P1112R0

EWG
2018-09-28

Balog, Pal (pasa@lib.hu)

Language support for class layout control

Abstract

The current rules on how layout is created for a class fall between chairs: if the user does not care about the order of members, they prevent optimal placement, while if the user cares, the control is taken unless all members have the same access control.
This proposal attempts to remedy this situation with an attribute that express the intent and reduce waste or ambiguity.

Motivation

This proposal is inspired by [Language support for empty objects] (http://open-std.org/JTC1/SC22/WG21/docs/papers/2017/p0840r1.html)] that allowed to turn off a layout-creating rule that requires distinct address for all members, including those taking up zero space. Causing wasted performance when the user never looks at member offsets.

We have another rule preventing optimal layout: p19 in [class.mem] stating " Non-static data members of a (non-union) class with the same access control (Clause 14) are allocated so that later members have higher addresses within a class object." (ORDERRULE) In practice that results in plenty of padding when the class has members of different size and alignment. That could be reduced if reordering the members was allowed.

On the other common use case the desired layout is compatible with another language and must have the members in a strict order. This can be achieved only by not using access control.

Proposal

We propose the addition of an attribute, [[layout(strategy)]] that can be applied to a class definition and indicates that the programmer wants to cancel ORDERRULE and orders the layout created in a certain way indicated by strategy.

[[layout(smallest)]] wants the members reordered to minimize the memory footprint. Minimizing the sum of inter-member padding and maximizing the tail-padding as tie-breaker if multiple variants have the same minimal sizeof(T), that may benefit in a subclass.

[[layout(best)]] allows reordering as the implementation decides optimal on the target platform, without specific preference.

[[layout(declorder)]] wants the members appear strictly in declaration order regardless access control, thus allowing standard-layout compatibility without interaction of the unrelated ACL functionality.

We thought about many other sensible strategies that are not proposed at this time, but the implementations can add their own keywords as extension and they can be standardized alter as implementation and usage experience emerges. But in the meantime, those can be put to use under (best).

Why language support is required?

Currently we have just one tool to get the best layout: arranging the members in the desired order. That brings in several problems:
- the source will be (way) less readable, the natural thing is to have members arranged by program logic
- the programmer must know the size and alignment of members; including 3rd party and std:: classes (that is next to impossible)
- if some member changed its content, what contains it needs rearrangement (recursively)
- such manual adjustment itself triggers need to rearrange the subsequent classes
- if the source targets several platforms, each may need a different order to be optimal

What makes the effort really infeasible in practice. We can ask the compiler to warn about padding or even dump the realized layout, but then many iterations are needed. And the work redone on a slight change. And we sacrificed much of readability and portability.

Therefore, in practice we mostly just ignore the layout and live with the waste as cost of using the high-level language. Against the design principles of C++. And this is really painful considering that cases where we use the address of members for anything is really rare. And that the compiler has all the info at hand when it is creating the layout to do the meaningful thing, just it is not allowed.

Why attribute?

It passes the "compiling a valid program with all instances of a particular attribute ignored must result in a correct interpretation of the original program" test. This also ensures that existing code will not change its meaning, the programmer must actively apply the attribute.

The attribute name

We considered a simpler approach with just an attribute to disable the ORDERRULE : [[no_incremental_address]]. That is great for language lawyers, but programmers would benefit more from reflecting the motivation. So next it was [[optimized_layout]] and [[bestlayout]] along with [[smallestlayout]].

But at defining the semantics we (obviously) discovered that "best" is an elusive thing. Smallest was simple to define but in some special cases the size reduction may result loss of performance, not gain (i.e. on modern platforms going from 64 byte to 60).

So we prefer an attribute family [[layout()]] with options already defined and ability to extend with relative ease. At the same time adding another strategy going the opposite direction, as it is trivial to define and implement, and cures an old pain.

Within the strategy keywords "best" is still somewhat fishy, but in this framework is more intuitive. The compiler optimizer options also started as Osmall Ofast but as a multitude of elementary optimizations got controllable introduced O1 O2 O3 umbrella for the regular user meaning "give me what you think is best"

without a special preference. It can be changed to "optimized" but we think it is just longer and not really more expressive.

For "declorder" we also considered "standard" that aligns with the likely intent of use. However, this might confuse the user when other requirements of standard-layout class-ness are not fulfilled beyond the same access. ("standard" may be considered a separate strategy making the program ill-formed if not applicable).

Other considered strategies (not proposed now)

"pack(N)" would invoke the effect of #pragma pack(N) finally bring this omnipresent facility in the standard. Not included because this proposal aims only at reordering members, not interacting with alignment.

"compact" would imply "smallest" "pack(1)" and implicit [[no_unique_address]] removing all possible waste.

"cacheline" a very powerful strategy for speed optimization aiming to set sizeof(T) be a divisor or multiple of the cacheline size. Not included before gathering experience with implementation and impact, especially for sizes over the cacheline size. Possibly needs additional tuning parameter, i.e. to control maximum extension.

Interaction with core

A major clash point is with 'standard-layout'. Rearranging the members shall render the class not standard-layout. The ambiguous point is when the [[layout(smallest)]] is present on a standard-layout class and no rearrangement actually happened. But that may change just due to size change of members later.

We believe that the spirit of standard-layout lies in the actual layout arrangement. The requirement on no mixed access was put in only as reflection of ORDERRULE. bullet (7.3) can be changed to this original intent stating it depends on all ints members laid out in declaration order.

Going this route would create a behavior change on existing code without using the attribute (the only such change): is_standard_layout for a class with mixed access always reported false and now can report true if the implementation did not use the license of reordering (likely). We consider preserving the way of such code has less value than the change that moves toward the original intent and give more clarity.

Full compatibility, if desired, can be achieved by an extra bullet in wording.

Interaction with library

The specification method of the library allows the implementation to use or not use the attribute without the user could detect it. The few cases where it is not evident are classes with public members, like std::pair.

Simplest and safest way is to explicitly state that the implementation is allowed to use [[layout(best)]] except where multiple public data members are specified. Or just state that library classes with multiple public members will have them in layout in declaration order.

Risks

This proposal does not create new kind of risk, as impact is similar to [[no_unique_address]]: if we mix code compiled with versions that implement it differently, the program will not work. (Similar mess can be created by inconsistent control of alignment through #pragma pack and related default packing control compiler switches.) But we add an extra item to those potential problems. For practice we consider these problems as an aspect of ODR violation.

The implementation of the attribute in general and a stable approach for "best" in particular, must become part of the ABI.

Summary of decision points

- preserve not standard-layoutness of mixed-access classes without attribute? (no)
- tie standard-layoutness to actual created layout (yes) or consider potential reorder as breaker
- bikeshed the strategy names
- add/remove some of the strategies

Plan for wording (draft)

Add a new subclause [dcl.attr.layout] after last attribute

9.11.12 Layout control attribute

1 The *attribute-token* **layout** specifies special rules regarding nonstatic member placement when the memory layout for a class is created. This attribute may appertain to struct or class definition. It shall appear at most once in each *attribute-list*.
It shall have an *attribute-argument-clause* of the following form:
(*layout-strategy*)

layout-strategy:
declorder
smallest
best

2 The attribute allows placing members in layout according to the indicated strategy, removing requirement set in (10.3 p19).

3 The **declorder** strategy requires members appear in the layout strictly in the order of declaration. [*Note:* Members are allocated so that later members have higher addresses within a class object or same if [[no_unique_address]] was used. – *end note*]

4 The **smallest** strategy aims for a layout with the smallest memory size. The size is considered smaller if sizeof(T) is smaller or sizeof(T) is equal and the amount of tail padding is greater. The implementation shall consider possible orders of the members and use one of the layouts with the smallest size. If the layout that is created by ignoring the layout attribute has the smallest size, that must be used. [*Note*: Gratuitous reordering without gain is not allowed. – *end note*]

5 The **best** strategy aims for a layout that the implementation considers ideal using its knowledge about the target platform. Any reordering is allowed. [*Note*: The result is up to quality of implementation. It can be identical to what the smallest strategy produces. It can leave the layout as if it had no layout attribute. It can increase the size if that is likely to increase the runtime performance. Quality is not considered good it makes the performance worse without a chance to be better. – *end note*]

6 The reordering shall be deterministic, so multiple translation units use the same layout from identical member definition and strategy.

[*Example*:

```
  [[layout(smallest)]] struct S {
    double d1;
    bool b1;
    double d2;
    bool b2;
    double d3;
    bool b3;
  };
```

If double is aligned on 8 bytes, without the attribute the struct has 7 bytes padding after b1, b2 and b3, having size 48 bytes. With the attribute it can be reordered by moving the three bools next to each other and have 5 byte padding, size 32 bytes. The bools must be placed last for the maximum tail padding. -- *end example*].

In 10.1 [class.prop] p3 change bullet (3.3)

(3.3) — has the same access control (10.8) for all non-static data members,
(3.3) — has such layout, that all non-static data members are allocated in the order of their declaration,

Add at end of first sentence of 10.3 Class members [class.mem] p19

Non-static data members of a (non-union) class with the same access control (10.8) are allocated so that later members have higher addresses within a class object, unless the class has the layout attribute (9.10.12). The order …

Add new section after 15.5.5.15 in 15.5.5 [conforming]

15.5.5.16 Layout control [lib.layoutcontrol]

1 The C++ library classes where multiple public data members are specified (like std::pair) must ensure a layout where these members appear in declaration order.

Possible improvements in the future

(Not part of the proposal at this time.)

Bulk specification

The programmers who want to use this attribute will likely want it on majority of their classes. So, a simple form that could add it to many places with little source change would be good. Like with extern "C" that can be applied to make a {} block that will apply it to all relevant elements inside.

We considered the attribute applicable to the extern "" {} block and namespace {} block with the semantics that it would be used on any class definition within the block without a layout attribute. Neither felt good enough.

The implementation could likely add a facility like current #pragma pack along with push and pop, but pragmas that is not fit for the standard.

Additional strategies

These were discussed earlier