P1112R1

EWGI, EWG
2018-11-25

Reply-to: Balog, Pal (pasa@lib.hu)
Target: C++23

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.

Changes fron R0

+ status section
+ wording for bit-fields
+ Q&A to address questions rised on EVGI list
+ example showing visible semantic change from declorder
+ show a possible alternative approach instead of declorder
+ new idea to split "smallest"

Status

The R1 version was discussed by EWGI at San Diego. The verdict is that the proposal worth pursuing, however needs additional refinement and preferably experience with implementation/usage. It does not fit the C++20 timeframe. I expect to create R2 for Kona but it will not much different from this one just slight refinements based on reflector feedback and more concrete decision points. Probably will skip discussion in Kona or just run polls on open questions. Then pick up speed in Cologne, where I hope implementation experience can also be presented.

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. (Note: see also the "New idea on "smallest" later.)

[[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).

Examples

```
struct cell {
    int idx;
    double fortran_input;
    double fortran_output;
private:
    double f();
public: // should be private but we need
this
    // be standard layout!
    // PLEASE do not touch!
    mutable double memoized_f;
};
```

```
struct [[layout(declorder)]] cell {
    int idx;
    double fortran_input;
    double fortran_output;
private:
    double f();
    mutable double memoized_f;
};
```

```
// hand-optimized to save space!
// sorry for the mess
// please remember to re-work if add
// or change a member
struct Dog {
    std::string name;
    std::string bered;
    std::string owner;
    int age;
    bool sex_male;
    bool can_bark;
    bool bark_extra_deep;
    double weight;
    double bark_freq;
};
```

```
struct [[layout(smallest)]] Dog {
    std::string name;

    std::string bered;
    int age;
    bool sex_male;
    double weight;

    std::string owner;

    bool can_bark;
    double bark_freq;
    bool bark_extra_deep;
};
```

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. Also this seem the least intrusive way and allows the same source used with non-supporting compiler.

Jens Maurer provided this example:

```
struct [[layout(declorder)]] S {
 public:
   int x;
 private:
   int y;
   void f() { if (!(&x < &y)) abort(); }  // abort() is never called
};

int main() { S().f(); }
```

Here the program can potentially abort if the attribute is ignored and the implementation did put y before x as it is allowed.

This formally goes against the usual EWG approach on what is suitable as attribute. However that guideline is not universally used. Some attributes were not considered with it in the first place (alignas, contracts). And we can construct a similar example for no_unique_address too, unclear whether it is an overlook or not.

We suggest to evaluate the motivation of this guideline and its universality, and what it wants to protect from. The idea of attribute-with-semantics is already in the language.

We expect that a progremmer who relies on working of layout(declorder) would use the implied detection for its support and discocer the trouble at compile time.

An alternative to attribute would be to use grammar and a keyword, like

```
struct S layout(smallest){ ... };
```

that is less pleasant to read for both humans and tools and is hostile to usage of same source with older comilers (barring macro-based hacks that inject the construct).

Alternative solution for declorder

The original motivation can be covered from a different direction, that also addresses the just mentioned problem: by making declorder the default mandated behavior. IOW revoking the implementation's licence on reordering with mixed addess. And the layout attrubute would indicate opt-in to allow reordering. (A "by_access" strategy could mean allowing the previous behavior.)

We think this solution is more intrusive (even considering how little evidence we seen on actual using the reordering license), and prefer to allow the original way. We think this core change is prone to even more confusion on when the programmer can start to rely on it, and on that route suggest a specific feature-check.

A new idea on "smallest"

Some of the feedback is concerned on reinterpret_cast and general surprises related the "smallest" that is allowed to move the single base class down to save space. This can be addressed by providing a strategy specificially without that factor. (working name -- suggestions welcome) "minpadd" would require to keep the 0-offset base class at 0 offset if such exists, while beyond that work as "smallest". This might result wasting a handful of bytes, but beyond saving reinterpret_casters, would strictly prevent introducing offset-adjustment. With related potential of performance degradation.

Then "minpadd_full", "minpadd_all", "smallest" would look for really the minimal memory footprint.

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 its 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, that keeps mixed-access implying not standard-layout. If that way is chosen we suggest to mark that condition as deprecated.

* *From CWG discussion on P0840R1 in JV2018: " Jason: My understanding of "standard-layout" is that it the layout as-if reading the source. If we allow more subtle modifications, it's no longer standard-layout. Consensus."*

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; if choice is yes, consider to keep mark this condition as deprecated)
- tie standard-layoutness to actual created layout (yes) or consider potential reorder as breaker
- bikeshed the strategy names
- add/remove some of the strategies (i.e if "best" stands in the way, it can be moved out into a separate papaer depending on the base one.)

Plan for wording (draft)

(Reflects the base proposal in R1)

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). [*Note:* The order of the members can be changed compared to how they would appear without the attribute. – *end note*]

3 For bit-fields, each field shall remain in its allocation unit. [*Note:* Moving fields within the allocation unit and moving the allocation unit is allowed but not moving fields to a different unit or joining allocation units. – *end note*]

4 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*]

5 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*]

6 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*]

7 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.

Add layout *to table 15 in 14.1 [cpp.cond]*

Acknowledgements

Many thanks to Richard Smith for championing this proposal.
To James Dennett, Daniel J. Garcia and Roger Orr for reviewing the initial draft.
To Jens Maurer for clarification on "attribute" and the related source example.

Appendix

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.

Q&A

Is there implementation experience?

No/almost. I have implementation for all key parts, but not yet inserted them into an actual compiler. That was planned to happen before the SD meeting but didn't happen due to lack of time. I expect to catch up before the next meeting.

Implementability (with relative ease) was key element in design of this proposal. Many interesting ideas were dropped due to being unclear how to implement across variety of platforms or expected to collide with known compiler extensions.

Is there usage experience?

No

Why does this need to be an attribute? C++ attributes traditionally don't change semantics, and this changes offset in a manner that is observable in C++. “Why attribute?” tries to address this, but it's not really true. To be fair, neither is it true for [[no_unique_address]]. Could this be done with something else than an attribute?

It does not "need" to be an attribute, but that is the framework that is least intrusive and easiest for implementations. Also the proposal was directly inspired by [[no_unique_address]] thinking "this has the very same global impact".

If not attribute, then it leaves us with a keyword, that is way more intrusive. Even if it could be context-sensitive, looks like much heavier. That would also make the code hostile to cross-compile. The code written with this attribute passes the old compiler that does not support it, and the client will still get a working program. Except for one corner case, but there the user actively relies on the support so could not expect good results without.

Layout “smallest” is close to “packed” attributes. Please cite prior experience in various toolchains. Is there uniformity in implementations of “packed”? Why don't you propose that semantic? (example at https://godbolt.org/z/3wyjrn)

The packed "attribute" in the previous example changes the alignment of int. This proposal is about *reordering* within the layout and treats the alignment as given. It was considered and dropped due to the current standard has no direct control on the alignment (only cloning unspecified one.) While most compilers have several methods for alignment control (compiler switch, #pragma pack, __declspec(align) to name a few). And the control is already dependent on their interaction plus the requirements of the target platform. Moving that into the standard would be more than desirable, but did not happen in the last 20 years, probably due to all the interactions and major risk.

The control of the alignment is an orthogonal issue, and reordering is helpful exactly in the cases where alignment is mandatory (i.e. the SPARC and many other RISC architectures) or desired to avoid performance penalty.

Meanwhile, the infrastructure of this attribute allows easy addition of strategies and vendors can (and we hope will) provide such in a way that is fit in their environment. And eventually those can get into the standard as the established, working practice.

Layout “smallest” is not really smallest as above packed example shows

It is smallest within constraints imposed by the platform and the programmer through alignment control. But alternative names are considered like "minpadd".

Layout “best” is pretty handwavy. I absolutely expect implementation and usage experience before this can move forward. I want to see that there's an actionable benefit to the compiler when this attribute is used.

"best" has the weakest specification, that allows to do nothing or mirror "smallest", so there *can not* be problem with implementability. However it allows the implementation to provide some much better. It serves clients who know that they are not interested in any specific placement and allow full freedom for the implementation. And it can improve for them over time without a need to change the source code.

Benefit from manual rearrangement of class members was presented in many articles and talks. This proposal opens the way to save human effort and allow the compiler to take over.

If where compiling all the client using a struct is not a problem, it could be used in profile-based optimizations instead of simple heuristics. I really see no reason to doubt the compiler could do any better than it can now burdened with an ancient restriction.

Layout “declorder” isn't the current standard's mandated order, but it's effectively what implementations do. Is that correct? If so, can the paper instead standardize existing practice?

We don't know what all implementations do now or have in the pipeline. The fact is that the current standard allows reordering for mixed access, so the programmers can not rely on that it is not happening. While we are not aware of an actual case where it happens, the major vendors never specified that.

So to help the clients to be sure that attribute is needed. Unless we revoke the license of reordering completely for classes not marked. We considered it as a too intrusive move that alone could kill this proposal. But if the committee thinks it is viable or even preferred, we can go that way.

I'd like a discussion of ABI issues this paper can cause, and how users can avoid them (potentially with tooling help).

The ABI issues are the same as caused by [[no_unique_address]]. And usage of #pragma pack (+ alternatives). The latter is a thing we live together from the beginning of the C language. And the "tooling" is pretty weak on several major platforms. I.e. one can try to compile with MSVC switch setting the structure alignment to 1 instead of the default 4/8. And include <windows.h> and use something. The build is clean and the result will crash. As many structs will have a different layout in the program than in the system DLLs. (Because the source uses #pragma pack for control and pack(N) does not increase the alignment to N if it more than what comes from the switch...)

But tooling is certainly possible if the vendor provides it, i.e. on the same platform different values for ITERATOR_DEBUG_LEVEL, that cause different content in the standard classes has a chance to get an alert in linking.

The implementation can emit information on what attribute was used and in what way and internal identifier for strategy implementation and can check it too. Or an offset table. A related example https://docs.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-2012/k334t9xx(v=vs.110) is MSVC's warning C4742 that remembers alignment used for structure members. While the implementation is not open, the best guess is that the layout table is emitted to the .obj file as comment and is checked. The very same information can point out discrepancy on the member offsets. However this method is limited to cases where linking is involved. For a DLL+header there is probably no way to discover that the client compiled the header with incompatible options. (This latter problem is nothing new, just ry to build a WIN32 API application with the "default packing" option set to 1 and enjoy the crash related to system calls.)

This proposal does create an additional case, as the concrete algorithm even for "smallest" could suffer an incompatible change and "best" is more or less inviting it.

But the user who starts the project with arranging a solid build system that ensures everything compiled with same version and flags is protected from these problems too. While doing less is ill-advised. The libraries that ship as header+binary will probably stick to just the conservative layout control.

How does the proposal affect bit-field members (including zero-width bit-fields)?

See p3 in wording. The allocation units created from the original source are preserved, but the units are allowed to be moved around, so are fields within a unit. Constrained by the strategy senantics.

How can I keep certain members on the same cache line, or keep them in different cache lines?

This proposal only works on full structure, no mark for individual members. I have implementation plan for the cacheline strategy mentioned above that works well when the total (smallest) size is <= CL size.

For finer control a later proposal could add attribute to mark individual members and the strategy work on them. Or a strategy can describe some naming convention it use for guidance.

Until then the user has to figure out the layout manually and use the declorder strategy, that will apply it reliably.