P1112R5

EWG
2024-05-21

Reply-to: Balog, Pal (pasa@lib.hu)
Target: C++26

Language support for class layout control

Abstract

The current rules on how layout is created uses rules inherited from pre-C era. Forcing an incremental order that is not a utility outside cases where standard layout is desired. In modern language usage it's often not applicable in the application at all, or if it is, limited to a small subset of classes. But for the rest a layout is forced that potentially wastes memory and CPU cycles. Paying for what is not uses is not in the spirit of C++ design.
This proposal attempts to remedy this situation with an opt-in syntax that express the intent explicitly so the implementation can provide better fitting solution. And where the standard layout is desired it can be invoked as the strategy making it well visible rather than "just happen" by matching a long list of arcane rules. And with this opt-in some additionally previously excluded cases could fit in too.

The effect of this facility is only that members appear at a different offset in the memory, for all other purposes, like the initialization order, nothing changes!

Change history

Changes from R4 (first version presented to EWG)

Motivation: expanded explanation on differences to C and why the using declaration order for this is not viable in C++;
New section: Why reflection will not solve this problem
New section: Design principles
Strategies: **'eval' promoted to be proposed**
Strategies: 'smallest' renamed to 'small' and add algorithm description
Strategies: add target audience
Wording: present strategy on how wording will be created

(pre-EWG revision and discussion history moved to the end of this paper.)

Motivation

This proposal is inspired by [Language support for empty objects] (http://wg21.link/P0840) 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: *7.6.9 [expr.rel] "*(4.2) — If two pointers point to different non-static data members of the same object, or to subobjects of such members, recursively, the pointer to the later declared member is required to compare greater provided the two members have the same access control (11.9), neither member is a subobject of zero size, and their class is not a union. " Repeated in p19 of [class.mem], that recently got turned into a note to avoid redundancy. (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.

The programmer could start fiddling with the order to address this problem, but that has significant fallout. Pre-C++23 the result was not even mandated unless all members were private or public. Now that is no longer a problem, but the declaration order influences not just the layout but the construction order too.

What changes the semantic of the program. Possibly introducing a bug. And likely triggering warnings to also change the constructors following old guidelines. And it messes up readability bigtime. Normally we want related data members appear in groups. And the order follows the features and semantics.

So such reordering is not desired even if we know all the sizes to reduce the waste. But the size is changing even for classes fully in our control. Never mind content of std:: and external things. And any change triggers a whole cascade.

In C the tool to control the layout is the declaration order. While it is impacted by the just mentioned cascade problem and messing the source-logical grouping, it's at least viable. Reordering only breaks code that uses list initialization, that can be countered by consistent use of designated initializers.

Not so in C++, that reused the declaration for a different purpose: lifetime control. Swapping members will change the order they are initialized what can be important for semantics. And the client code using list initialization will break even with using the designated form, as order different from member declaration makes the code ill-formed. We also break structured bindings and defaulted operator <=>. On top of that in C++ the class definition adds member functions and access specifiers, escalating the grouping problem. And typical code has way way way more structs/classes with more nesting. Even just looking at raw count, never mind what is created from templates.

We use C++ as a high level language. The data members are used for what they do and we almost never care where they happen to sit in the memory. If I have 7 doubles and 8 bools that can properly fit and work in a 64 byte frame, it's baffling to see that instead 72 -- or even 120 bytes could be used. Doing the same task, just much slower and consuming more memory. Most often we just swallow this waste as using the programmer's time and brain cycles is needed elsewhere.

Lately we can observe most compilers gained warnings on excessive padding. Indicating it is a thing programmers care about. But they can't really help resolving that warning.

Declaration order is just not a viable general tool for controlling the layout in C++. And we don't have another way. This proposal tries to remedy that.

(more details in 'why language support required' section)

Reflection will not solve this

We're (hopefully) getting reflection soon, and maybe layers on top of that like metaclasses. Will that not solve the problem without a specific facility?

The answer is a clear no. We may feed reflection with a class. It can walk through its members and figure out a good layout. And with enough maturity could splice the recipe only swapping the members leaving all else as is. That would change the declaration order and trigger the related problems.

For those who want the full power what reflection could do, this paper offers the eval strategy. That saves a lot of effort that would go into inspection by the compiler providing the relevant information in cooked form. And the "splicer" function only needs to patch up the offsets belonging to members. Then that recipe will be used as layout while keep the init/lifetime semantics unaltered.

However, when we get metaclasses, it would be a natural client of this feature, also working around the (for now) missing bulk application feature.

Proposal

We propose the addition of a Layout Control Syntax (LCS), `layout(`*`strategy`*`)` that can be applied to a class definition (at same position where `alignas` appears) that indicates that the programmer wants to override the language-default layout creation. Explicitly stating whether he wants the standard layout or opting out of ORDERRULE and looks for altered offsets for members according to selected strategy.

The LCS is viable as described above, using 'layout' as context-dependent keyword, but can be formed differently without altering the aim. Using [[]] attribute was considered in early versions of the paper but this facility has semantic impact so cant use that. The 'strategy' part can be a single word or have additional part in ().

This paper wants to establish a *framework*. With just a small number of initial strategies worded directly in the standard. Opening the door for easy further extensions to add more strategies for specific needs. While the eval strategy allows library-based extension by just using consteval functions. After usage experience some of those can be added into the standard library.

The names and composition of the initial strategies can be altered on feedback. Here is the outline, more details follow in a later section.

layout(small) wants the members reordered to minimize the memory footprint.

layout(standard) ensures that the struct is standard-layout (in more convenient way that appending a static_assert) and aims to extend standard-layout-ness to some additional cases.

layout(eval(consteval_func)) invokes the names consteval function that gets the original layout as an array of entries and can alter the offset field.

layout(explicit) is an implementation-defined strategy, that is allowed to do nothing or mirror small, but the recommended practice would allow to reach an external data source for the layout information. Allowing to cover profile-based optimization and user-defined layouts be injected.

We thought about many other sensible strategies that are not proposed in this paper until positive feedback or usage experience (see later).

As mentioned already, we see much desire in this area and a major aim is to put the framework itself in place, so further papers have easier time and need only to fiddle with the payload.

Examples

```
// 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(small) 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;
};

// same with layout(small), but reads
layout recipe from outside instead of using
algorytm
```

```
struct cell {
    int idx;
    double fortran_input;
    double fortran_output;
};
static_assert(std::is_standard_layout_v<
cell >);
```

```
struct layout(standard) cell {
    int idx;
    double fortran_input;
    double fortran_output;
};
```

```
// trick to simulate extension
#define CELL_MEMBERS \
```

```
struct layout(standard) cell {
    int idx;
    int idx; \
    double fortran_input; \
    double fortran_output;

struct cell {
   CELL_MEMBERS
};
static_assert(std::is_standard_layout_v<
cell >);

struct cell_ex {
    CELL_MEMBERS

    int extra_info;
};
static_assert(std::is_standard_layout_v<
cell_ex>);
```

```
    double fortran_input;
    double fortran_output;
};

struct layout(standard) cell_ex : cell {
   int extra_info;
};

// now works naturally and IS standard-
layout too; (almost) identical to left
```

Interaction with the rest of the language

The most important clash is with standard layout. P1948R0 attempted a to fix the related information-deficit and sort out the problems for all papers like this. EWGI did not like that approach, so the task must be solved here.   For thins we propose the simplest approach: LCS implies forcing standard-layoutness if the strategy is 'standard' and canceling it otherwise. Going with the latter even if the offsets are not actually altered.   Later evolution may add more explicit control of that aspect extending the LCS or make it come from the invoked function/source.

We want to take care to not break any core constants.   sizeof supposed to work and remain stable. And offsetof, where supported.   Those do change values compared to the "original" layout candidate created before executing the strategy, but that would not be observed in any way.

This feature is hooking at the point where the class transitions from incomplete to complete and gains the layout. The implementation creates the "original" layout as now, and if LCS is present, it is immediately amended. The following process only works with the amended state, the original can not be observed by any means.

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

on top of that, manual rearrangement would cause:
 - change in the order of initialization of members
   -- likely trigger warnings on initializer lists
   -- possibly breaking the code if it depended on the order
 - need adjusting brace-init lists (designated init not helping!)
 - need adjusting structured bindings
 - operator <=> can't be defaulted and if it was, look for drastic changes

What makes the effort extremely 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.

Details of the proposed strategies

layout(small: [bikeshed names smallest, smaller, small, compact, minsize, optsize, size ...]

Aim is minimal memory usage with some constraints. one important tweak: if we have a base class at 0 offset, it remains there. EWGI voted to prefer that over strict smallest that could move this too. (Even despite potential name discrepancy vs. 'smallest'.) Experience shows that programmers expect single inheritance make base and derived be reinterpret_cast compatible even if it's not guaranteed. Beyond avoiding surprise and critical bug potential if the base class is moved, the well-formed implicit/static casts will need instructions to adjust the offset. What may be more of a performance drain than the gain from the rearrange.

The algorithm of how to place members is to be specified directly. The most recent idea is as follows:

1. create the original layout
2. from it keep all the non-novable members and if a base class is at offset 0, that.
3. sort all remaining members (for bitfields the allocation unit is considered as one member) by alignment(desc), size and declorder.
4. add them to layout in that order placing at lowest legal offset (in a hole they fit or at the end)
5. compare sizeof and tail-padding of the result with the original and if the latter is better, use the original

Without over-aligned members and tail padding this provides the smallest footprint and maximum space at tail. Otherwise it can leave some bytes that could be filled with more sophisticated approach -- but it is economic with compile time and not just stable but have the bonus that the programmer can calculate the result in head. And it handles the most troublemaking cases with individual bools and shorts well. Over-aligned types are pretty rare and the missed filling can result from funny-sized arrays combined with lack of small items.

For real smallest we could order the implementation to create layout for all permutations and select the smallest, but it is not feasible. Better filling of the holes can be done, but adds hard to predict amount of processing time, complexity and we lose the trivial predictability. The last check ensures we can't get worse so it looks a fair balance.

(The algo can be improved by feedback. Before the wording is created we expect to implement the algorithm candidate in isolation and test it with layouts gathered from real-life code to have better picture.)

The implementation can transform this to eval(__intrinsic_small) under the hood.

Target audience: everyone just looking for the low-hanging fruit without any specific desire.

layout(standard):

The class will be standard-layout or the program is ill-formed. If the class is already so by the current rules, nothing happens. Else, look for special case that could be made into standard layout.

The access control labels are ignored.

If the class has exactly one base class, create a rewrite where this base class is removed and injected as the first member. Check if this complies as standard-layout. If so, accept this as the layout. All the facilities work as if we had this rewrite. (The pointer-convertibility between the whole and first member applies to the base; offsetof figures the actual and usable offset.)

This will be a challenge to specify in standardese, but less tricky than the last extension with the empty bases. And it does not really change anything of substance.

I found the need for this use case in almost all projects that were interested in standard_layout, and while the workaround exists, it's far from nice. Getting rid of one more legit use of preprocessor should count progress.

Later evolution can figure more rule relaxations.

Target audience: those who look for utility of standard layout either for interoperability or the offsetof/common initial sequence support.

layout(eval(func)):

This allows the programmer to invoke an arbitrary strategy. func is a consteval function that will receive the original layout's description as a collection/span of entries. The entry has a mutable member offset that can be altered in the function.

The function returns bool. If false is returned, that means the function decided to not alter the layout. The implementation will use the original. This is convenient for the implementation that may start making changes and can bail out later if stumbled on a problem.   No need to copy/restore.   If the nature of the problem considered fatal, it can throw or use some other tech that will render the program ill-formed.

If true is returned, the compiler will run sanity checks on the offsets and if some of those fail, the program is ill-formed. The details of mandatory checks to be decided later, good candidates are to look for obeying alignment and no overlapping.

The payload could be something like this: (values filled similarly as in xml in the next section)

(note: proxy/strawman names just for demonstration, real ones to be worked out with library group; content can also change on feedback)

```
struct layout_entry
{
   enum type_id { base, member, tech  };

   type_id type;
   string_view name;       // member's name or unspecified
   size_t index;           // index by declaration order
   size_t size;            // what sizeof() reports
   int alignment;          // alignas value
   bool noua;              // [[no_unique_address]]
   bool fixed;             //  can't move, shall stay at offset
   size_t orig_offset;     // offset in the original layout
   mutable size_t offset; // desired offset (orig_offset on invocaton)
};

bool my_layout_func(span<const layout_entry> entries);
```

Target audience: those looking for special layout that can be programmed.   Later, if functions get published in libraries, everyone with luck to find one fitting their needs.

layout(explicit): [bikeshed: pbo impdef ...]

invokes an implementation-defined strategy. That allows to do nothing, but the recommended practice is to allow injecting layout information from an external source (file, database). On encountering, the class identifier is looked up in the file and if found, the layout defined there is used (if passed sanity checks).
The file content can be filled by the implementation's optimizer, external tools or the user directly. Compilers already have switch to dump all class layouts as a text file. It just needs a format spec to be usable (json, xml+xsd). one tricky element is the unnamed namespaces, but a mapping based on source path can be figured. The other content is hardly rocket science.
The utility of this could be enormous as not just factory tools could be used to optimize, but the user could simply generate layout candidates with a python script and run benchmarks with minimal effort.
And there is no concern about stability.

The compiler could use format like this:

```
<class  name="foo" full_name="::foo" mangled_name="$%#$%#$%#$">
  <item type="base" name="" index="0" size="18" align="4" orig_offset="0"
  <item type="member" name="m_i" index="0" size="4" align="4" orig_offset="20"
  <item type="member" name="m_b" index="1" size="1" align="1" orig_offset="24"
  <item type="member" name="m_d" index="2" size="8" align="8" orig_offset="32"
  <item type="member" name="m_s" index="3" size="8" align="4" orig_offset="40"
</class>
```

```
(tricks: size -1 means 0 with no_unique_address; align 0 means not not
movable)
```

```
<class  name="foo" full_name="::foo" mangled_name="$%#$%#$%#$">
  <item type="tech" name="VMT" index="0" size="4" align="0" orig_offset="0"
  <item type="base" name="" index="0" size="18" align="1" orig_offset="22"
  <item type="base" name="" index="1" size="0" align="1" orig_offset="22"
</class>
```

to write the current state and read the same with offset="666" attribute inserted by a tool or manually

The compiler can reuse most of what is done for eval, the same structure can be mapped to the i/o format, and instead of invocation the offset is read into the entry corresponding to type+index.

It could even be used to mitigate ABI transition problems!

Target audience: toolmakers and users of those tools. Those looking for hand-crafted layout.

Design principles

1. Most value packed into one paper

Presentation experience shows that it takes a *lot* of time to just introduce the related concepts. Layout is used by everyone but most people appear not aware of the subtle related problems -- while the remaining group has lot of concerns (mostly about stability). A minimal version and small increments, i.e. split strategies into separate papers would lead to bad cost/impact ratio. The related use cases are in a wide range from people who have very specific idea where they want members to those who not care at all just want to avoid pessimization.

Original idea was to just have smallest and let the framework grow -- adding the other proposed strategies barely add to the EWGI/EWG time while widen the utility in a drastic way.

2. Something usable out-of-the-box for everyone

We could say 'small' is not really needed as anyone could just implement his own, probably better version with eval. But writing consteval functions is hardly for everyone and even if the implementation was easy it would be a waste of most clients time when the expressed use case is just get rid of the blatant waste.

3. Pure addition

The code not using LCS is not impacted in any way.

4. Stability and no surprises

All strategies are stable and predictable.

5. Improved portability

The layout is inherently implementation-defined and non-portable. With this facility it is possible to order the exact same layout with a single source through using eval or explicit strategy or cover special ABI needs.

6. Standard layout deserves some love

Standard layout carries cruft that we can't get rid of without potential breaks. The opt-in way allows that.

7. Not hosing implementations

The standard says very little about layout for a good reason: implementations may require things hard to foresee. This paper has no desire to break that. It builds on top of current layout and offer solutions for weird cases not breaking conformance.

8. Avoid non-essential complications now but allow future extensions

More strategies can be added later either with new keywords or std:: functions for eval.
eval itself could be enhanced by allowing extra parameters in the LCS that are passed to the function. That is not proposed to keep that part of LCS as simple as the function name and dodge questions about how to smuggle the arguments, lookup, ODR-use, etc. The client can still use abstractions and invoke them from a named function.
For similar reasons bitfields are treated at allocation unit level.

9. Avoid clashes

Alignment control would fall into the agenda of this paper. But it is not pursued to avoid interactions with existing compiler configuration and #pragma pack, neither controlled by the standard. The infrastructure would allow it, but resolving interactions would be too much burden for now.

Other considered strategies (not proposed)

"ABI(XXX)" replicate layout rules of FORTRAN, python, VS2012, C++98 or whatever external entity indicated by the keyword "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.

"alpha(N)" sort by the name (or first N letters) combined with smallest where it is tied. This would allow creation of groups of members to keep together (for locality) or apart (to avoid false sharing). This allows easy experimentation with reordering by just altering a prefix. Probably obsolete if "explicit" is implemented in the desired way, as then the source can be left alone and the external recipe altered.

" smallest***" various alterations of smallest with less opportunity for waste and no base-class fixing

"pubprotpriv" place public, then protected then private members in their declaration order (as currently implemented by EDG invocable by configuration)

"best" was in the original proposal to allow unspecified or implementation-defined magic to allow whatever goes. EWGI didn't like it for inherent ABI instability, between compiler versions, potentially even between compilations. Salvaging it as "pbo" where we expect that did not fare better. Now superceded by "explicit" that handles the stability problem.

"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.

Most of these can be covered by eval so should not burden the standard up front.

Bulk specification (not proposed now)

The programmers who want to use this facility 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. Though that is a thing the users would likely welcome.

And obviously the compiler can use configuration (command line args or a file), like it already happens for EDG.

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.

In EWGI most concerns were expressed about ABI and general stability. The current paper only proposes stable strategies. Certainly an inconsistent compilation could create a problem, but that is a GIGO case. (See more discussion in the appendix).

Wording

Wording will be created after key elements approved.

The idea is to have one section dedicated to facility (probably appearing after alignas) that introduces the LCS and the details of tweaking the offsets. Outside that only few existing lines get amended:
 - ORDERRULE to not apply upon LCS
 - standard layout rules to turn off if LCS present except (standard)

The eval strategy uses a few structures and enums that are passed as parameters into the invoked function. Those will be defined in the standard library section.

Acknowledgements

Many thanks to Richard Smith for championing this proposal on initial presentation.
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.
All folks in the incubator providing feedback.

Appendix

Q&A

Does it change the initialization order of members?

Absolutely not! The only change is the offset of the members within the memory. Any other semantic is unchanged. One of the major motivations of this paper is that manual rearrangement changes things that we want to avoid.

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 LCS 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 try 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 "small" could suffer an incompatible change.

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)?

The allocation units created from the original source are preserved verbatim, and those units can be moved around according to strategy. The strategy could specify reordering fields too, but the proposed ones currently leave them as is.

Pre-EWG change history

Changes from R3

Consolidating final EWGI feedback for first EWG presentation.
Consolidating effect of adopting P1847 into C++23 (declaration order is mandated ignoring access control)
Specifying interaction with standard layout.
Strategies: rework 'declorder' strategy to 'standard', 'pbo' strategy to 'explicit' with additional features.
Other strategies: +eval, +ABI

Changes from R2

Change syntax from attribute to contextual keyword
Delete parts that are no longer look relevant or important including attribute discussion, most of FAQ, wording
Strategies: +pbo, smallest redefined as 0base-preserving
Other strategies: +strict_smallest, -best, +pubprotpriv, +C++03, +C++17

Changes from R1

Reflect discussion at Cologne meeting.
- "declorder" strategy still discussed as motivation, but it is moved out to P1847 to be the default
- remove "best" strategy
+ refer to Herb's poll on desired papers for C++23

Changes from R0

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

Discussion history

R0 presented in EWGI by Richard Smith in San Diego (2018), passed teh motivation poll
R1 was discussed in EWGI in Cologne. Some of the previous decision points got polled. "declorder" is being pursued in separate paper P1847. Hopefully that passes, then this paper will only provide strategies to relax the strict ordering.
R2 was discussed in Belfast at SG7 providing good insight on what can be possibly made in the future using compile-time programming, including even user-provided consteval functions. That will not be pursued in this paper, but in follow-up after it is adopted. EDG appears to already work to support use cases

similar to ones in this paper. SG7 agrees that the facility is wanted and does not force a consteval-based approach, so the original one continues.
R2 was also discussed in EWGI and polled several open questions. Most importantly the attribute syntax lost 0/0/3/2/2 to context-sensitive keyword and the room voted 1/6/0/1/0 to use the 0-base-preserving version of smallest strategy.
R3 presented in Prague, gained forward to EWG. Directional polls suggest to have only the smallest strategy at start; research reflection-fbased solution allowing the strategy itself coded as library consteval function (possibly in follow-up).
R4 presented in Varna, gained consensus to proceed with some opposition that will need more convincing on motivation and value.

Related paper P1848 didn't gain traction so standard-layout related interactions must be embedded in this paper. http://wg21.link/P1848
Related paper P1847 got accepted and adopted making the previous 'declorder' strategy obsolete.
http://wg21.link/P1847