P1112R4

EWG
2023-05-18

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!

(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. And those so determined to go with the swaps anyway will then enjoy crippled git history. Dragging crippled reviews with all fallout of that.

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. And this all could be solved if we could just opt out of ORDERRULE and let the compiler just give us the best thing.

We have some hard evidence that this feature is wanted. In https://herbsutter.com/2019/07/25/survey-results-your-top-five-iso-c-feature-proposals/ **this paper got into top 50** with 7 votes. One of those is mine, but the other six was born in the most natural way, I did not mention the poll to anyone.

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. In the fist version of the paper we meant the attribute literally, with the [[]], but in the meantime decided to go with its broader sense, like alignas that is specified in [decl.attr] section of the standard, but is not ignorable and have semantics. (If the text of this paper refers to the facility as "this attribute", it's means just as a self-reference.)

The invocation syntax is open to bikeshedding and can be reworked later. The most recent idea is to use layout as a **context-dependent keyword** with () that appears between struct/class and the name, just as alignas or the [[]] attributes. Inside the () further syntax defines the strategy and its related arguments.

This paper wants to establish a *framework*. With just a small number of initial strategies (possibly just one). Opening the door for easy further extensions to add more strategies for specific needs. That provide good utility even just as vendor-specific extension, but later can be standardized from existing practice/use experience.

The names and composition of the initial strategies are up to bikeshedding. Here is the outline, more details follow in a later section.

layout(smallest) 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(explicit) is an implementation-defined strategy, that is allowed to do nothing or mirror smallest, but could cover profile-based optimization and user-defined layouts be injected from an external source.

We thought about many other sensible strategies that are not proposed in this paper until positive feedback or usage experience (see later). Special highlight would be the strategy that could invoke a consteval function that takes the "raw" layout and returns it with potential modification of the offsets. If we already had reflection in the standard, most cases could be covered in library, instead of the compiler core.

Es 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.

The names and composition of the included strategies are also open to bikeshedding.

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(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;
};

// same with layout(smallest), 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 \
    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>);
```

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

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

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

Interaction with the rest of the language

The most important clash is with standard layout. Unfortunately the attempt to sort it out in spin-off failed, so we have to cover it here individually. We propose the easiest approach: by default the presence of the attribute makes the class not standard layout. (Including when e.g. it was, and we asked smallest that did not alter the offsets.) This is augmented by the strategy itself, that can order otherwise. In the current state the layout(standard) makes it so.

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 "initial" layout candidate created before executing the strategy, but that would not be observed in any way. Those require a complete type. The strategy is inserted as a new last step as the class becomes complete.

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
- need adjusting structured bindings

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(smallest): [bikeshed names smallest, small, compact, minsize, size ...]

Aim is minimal memory usage. With one 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. 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.

An algorithm like sorting members by alignment(desc) and declorder + hoisting applicable items into the base classes tail padding will be specified. It can be made stable.

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.
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.

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.

It could even be used to mitigate ABI transition problems!

Other considered strategies (not proposed now)

"eval(func)" hand the layout to a consteval function that will patch up the offsets and return it for use. The user can just write his own strategy or invoke std::/third party ones. Requires reflection.

"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.

"strict_smallest" allow moving every base class to have the actually smallest footprint

"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.

Bulk specification (not proposed now)

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. 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).

Summary of decision points

For first EWG presentation:
- approve the motivation
- strategies to include/excude
- approve general syntax (one context sensitive keyword at class head, not [[]])
- approve standard layout approach (nuked ny default)

for later:
- bikeshed the strategy names and syntax nuances

Wording will be created after key elements approved.

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 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 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 "smallest" 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).

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