P1112R3

EWGI
2020-01-12

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 away unless all members have the same access control.
This proposal attempts to remedy this situation with an opt-in syntax that express the intent and reduce waste or ambiguity.

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!

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"

Status

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.

The resolution of how to interact with the standard-layout related core wording is pursued in P1848 as that is not coupled to just this particular paper but any other that wants to tweak the layout and we have several such papers already and expect more.

R3 will supposedly be discussed in EWGI in Prague for a final round and gets blessing to move ahead to EWG.

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: *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.

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

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.

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. (see details later.)

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. (Note: if P1847 gets accepted then this will be the default state, so dropped as strategy, but possibly replaced by pubprotpriv strategy to ask placing public, then protected then private members in their declaration order matching EDG's already existing configurable behavior).

layout(pbo) invokes an implementation-defined strategy aiming to cover profile-based optimization use cases or other smart knowledge related to the platform. The implementation can define it any way it likes, including be equivalent of smallest, declorder.

We thought about many other sensible strategies that are not proposed in this paper until positive feedback. But the implementations can add their own keywords as extension and they can be standardized after as implementation and usage experience emerges.

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

```
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

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.

A new idea on "smallest"

In R0 "smallest" was seeking the minimal memory footprint. 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 specifically without that factor. That keeps the 0-offset base class at 0 offset if such exists, while beyond that work as the original. 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.

EWGI voted 1/6/0/1/0 to use this latter approach as definition of "smallest". With the A vote matching my concern, about the name "smallest" not telling the truth.

Other considered strategies (not proposed now)

"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.

"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).

"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.

"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)

"C++03" "C++17" if P1747 is accepted the ruleset defined in those standards (allow swap on access labels or state) could be asked explicitly invoking the behavior already in the implementation.

Interaction with standard layout

When this attribute is applied to a struct that is standard-layout without it, it evidently stops to be standard-layout if any member gets a different placement.

A more interesting question is what should happen if nothing is moved. In that compilation, that is, as the outcome may be different in another compile if some distant code changes, or even without change in case of pbo.

The discussion on this is moved to a separate paper, P1848, as the answers are not specific alterations made by this facility, but any, involved in tweaking. So it is more practical to resolve in a single place instead of every paper individually.

Beyond the global approach individual strategies can be defined to force standard-layout-ness in a direction like pbo to always false.

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(*)]] except where multiple public data members are specified. Or state that library classes' layout is unspecified. so users can not rely on member positions at all. A clear published decision is preferred compared to just say nothing.

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.

Most strategies will need to be tied with the ABI, and we expect the implementation help the user with non-stable workflows like actual pbo.

Summary of decision points

- bikeshed the strategy names
- add/remove some of the strategies

In related papers:
- P1847 to make declaration order the mandated default
- P1848 standard-layout interactions

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

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)?

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.

I added a simple "alpha" strategy to the non-proposed but interesting strategies that could gelp this use case. It may be considered for inclusion.