P1847R3

EWG
2020-03-01

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

Make declaration order layout mandated

Changes from R2:

Add vendor information on Clang and gcc.
Removed feature test macro following SG10 guidance
Text hilighting

Changes from R1:

Rebased wording delta on the most recent WP.
Add collected information on current implementation's use of swapping license
Added more history of related changes.
Reworded the motivation.

Changes from R0:

Fixed the wording on review comments: un-strike "class" and add another delta in [expr.rel].
Added sectoin on C++03 version of the rule.

Abstract

The current rules allow implementations freedom to reorder members in the layout if they have different access control. But this is not really used in practice. We propose to make the standard reflect the actual industry practice, so remove that license and make the layout be created using the declaration order. As a baseline, a sturdy foundation for later-defined facilities that can allow reordering in a controlled way using some opt-in syntax.

Motivation

The idea emerged in discussion on P1112 *"Language support for class layout control"*. (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1112r2.pdf). It proposes to empower programmers with better control over the layout created for classes. The framework allows selecting a strategy to relax or strengthen the ordering compared to the current default. It shows examples why asking for strict declaration order can be desired.

While this paper was born to resolve a conflict with using attribute for that paper, following EWGI discussion in Cologne that suggested to split off this change in a separate paper with a 3/5/10/0/0 vote, it started its own, separate life.

The main argument is no longer to support that particular paper, and especially saving the attribute route for it (that is no longer pursued in R3).

We want this change for much better reasons: it seem to be the established industry practice. And also a cleanup/simplification of the rules. That will serve much better as a foundation to build on.

The other good reason is that the current rule just doesn't make sense. And seems established by an accident.

The first version of the standard had one fixed point: a POD struct needed to be compatible with C. Beyond that it could have allowed anything, but decided to use the access labels for control. That allowed the user who desired ordering between some members to put them is a block, or allow any swapping by inserting labels for each. On forums there were suggestions to use that style that looked surprisingly readable, edit-flexible and had potential to provide a better-optimized layout. *This paper would not have been written if we still had the rule this way*.

But it changed in a significant way for C++11: instead of labels it was reworded to work on just the effective access level. The paper (N2342) is explicit on that it wants it that way, though does not elaborate the explanation. And it is mostly tied to "standard-layout", not anything outside of that. So the extra restriction looks nothing but collateral damage.

This change happened after a good decade of C++ being standardized. On the review it was assumed as potential, bit not realistic ABI break. And we are not aware of related complaints, so assume that even using the wide license was not picked up.

Looking back we suggest that it was a suboptimal change, the ordering should have either stayed unchanged or pushed all the way to complete removal. And we will be better off correcting it. Going backwards is less practical, as there was a good reason of ignoring the license: it adds serious ABI headache, while without aimed control the benefits are moot. The labels are there for a different reason and the programmer is supposed to know its goals and be able to communicate to the compiler. And for the cover of those cases we have work in progress. The current rule is more in the way of those than the "clean slate" created by going all the way.

Proposal

We propose to remove the implementation's license of member reordering in case access control is mixed. In this paper we do not propose anything else. Probably easiest to look directly at the wording section.

Impact on standard layout

What counts as standard layout for C++ does not change from *this* paper, as [class.prop] bullet (3.3) does not change. This is deliberate as observable impact of that would bring in extra risk that is best handled separately. That supposed to happen in paper P1848 dedicated to recover information related to what SL means and tune it for needs of other papers and this one.

This means that is_standard_layout<T> still reports false (as did before, avoiding behavior change in code that issued the test). Also related benefits like offsetof or common initial sequence do not apply. But for interfacing with other languages the actual layout will be fine and immediately usable "by standard" instead of just "by observation".

What implementations use the license currently

As we have no view of all the existing or possible implementations, the answer can not be complete. I asked around on several forums for help, and got the following:

- Microsoft, Clang and gcc and one more major implementer said they do not use the swapping license and are not impacted from the change.
- EDG has a configuration flag (“targ_field_alloc_sequence_equals_decl_sequence”) that can be set to false, in which case we first lay out public field, followed by protected fields, followed by private fields. But not aware of an actual customer using it and can't recall a customer report with that mode used.

Impact on implementation that did not use the license

It is conforming without any change. The users who wanted declorder layout and this far only relied on hope or ABI documentation now can rely on the standard directly. And run no further risk whether the implementation decides to change its mind later on.

Impact if an implementation did use the license after all

Without further change it becomes non-conforming. If it doesn't track the standard, will likely not change the behavior at all, and the users will not notice any change. A more interesting risk for this case if new code starts to rely on the new rule and gets migrated to such a platform.

If it does track the standard, it will most likely offer mode for old compatibility. As the actual layout for the case was "unspecified", the users had not much to rely on beyond ABI specification and will see changes there.

It's important to note that tweaking the layout in various ways appear as a high demand target. We already have several papers in flight and people working on use cases. By the time this change is released in the standard we can expect other facilities that move this rule in the other direction. So through them they may be conforming again, with just some documentation work. For example to cover EDG's case I'm adding that strategy to the proposed list. So users will be able to request it on per/class basis with the opt-in syntax. And while bulk introduction is not active part of the paper, they can document the configuration as such and work as inserting the request on the code.

In a similar way the current rule could be added too, though unlike EDG's approach it is not inherently ABI-stable so probably not practical.

We can't just change such an old established rule

The interesting thing is that we did in fact change this rule, and I could not even locate the related papers and discussion allowing that. C++11 has similar text to current, but

C++03 9.2 p12: "Nonstatic data members of a (non-union) class declared without an intervening *access-specifier* are allocated so that later members have higher addresses within a class object. The order of allocation of nonstatic data members separated by an *access-specifier* is unspecified (11.1). ..."

allowed way more freedom to rearrangement. That was trimmed significantly. The old rule required only members between labels kept in a block, and those blocks placed in any way. The new ignores the labels and allows just 3 blocks with the public, protected and private members, and those zipped.

This change was made in the paper N2342 "POD revisited" (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2342.htm). The paper aimed to decompose "POD" into trivial and standard-layout. The motivation says "*As a consequence of allowing members of any access control in standard-layout types, the current requirement that POD data members have no intervening access-specifiers is changed to require only that such data members have the same access control. This change is believed to also be more in line with programmer expectations than the current requirements.*" This is not really what is reflected in the wording delta, as it was not only used in the definition of the new term of standard-layout, but also restricted the ways of how the layout is created. For either standard or non-standard layout types.

I could not find anything related from the EWG discussion. On CWG review the team did flag the change in paragraph as ABI-breaking (http://wiki.edg.com/bin/view/Wg21toronto/CoreWorkingGroup), but not used so far.

Benefits of accepting this change

The main benefit is creating a cleaner state that is 1/ aligned with existing practice and 2/ no longer carry a rule that looks arbitrary and lacks a good known motivation. The standard carries plenty of old baggage that is in the way, but we keep it to save backward compatibility. For this case we can make the change while remaining backward compatible.

Wording

(Delta against N4842)

In 7.6.9 [expr.rel] change p4.2 by deleting text:

(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.

In 11.3 [class.mem] change p19 by deleting text:

19 [*Note*: Non-static data members of a (non-union) class ~~with the same access control (11.9) and~~ of non-zero size (6.7.2) are allocated so that later members have higher addresses within a class object. ~~The order of allocation of non-static data members with different access control is unspecified.~~ Implementation alignment requirements might cause two adjacent members not to be allocated immediately after each other; so might requirements for space for managing virtual functions (11.7.2) and virtual base classes (11.7.1). —*end note*]

Acknowledgements

Thanks to Jens Maurer for the idea of this alternative path and for reviewing this paper. And everyone else who provided ABI information or commented on the proposal.