P1848R0

EWGI, EWG
2020-01-12

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

Improve rules of standard layout

Abstract

The spirit of standard layout (SL) is to have a memory footprint where all the members appear in the order[1] as they are declared in the source and beyond padding nothing else is there. Such memory chunk can then be used in interfacing with other languages in a reliable way (expecting them to be ruled by the same ABI that covers size, alignment and internal representation of types like int or double).

The rules did not cover this ideal right from the initial definition of the term. And these days we have serious demand for facilities that work on the layout, with several papers already in flight, that look for resolution on their interactions.

This paper aims to uncover the important questions, find out the rationale on how they are answered for the current state and then look for answers considering the incoming changes.

Motivation

The natural question is why we need a separate paper instead of discussing and resolving impact on the individual papers. And view the status quo as the state that served 2 decades well, so it must be good. And well understood by everyone.

Unfortunately the latter is not the case, even in CWG transcripts the question of what we mean by SL keep popping up. And we saw cases of misremembered rules even there, by people we can consider in the closest relation with the spirit and the letter of the standard. In other groups it's even easier to have a debate on nuances with incorrect verdicts surviving even fetching and reading the standard text. And it's just on what the state *is*, not *why* it is that way.

And the new papers typically want to do something smart with the layout itself, not directly interested in SL in any way. But the impact needs a resolution. In the ideal state they would not really have the impact in the first place, the rules would be already just work. But they are not, as definitions are done along implementation details instead of abstract principles and we have even redundancy issues like the recently fixed CWG2404 (http://wiki.edg.com/pub/Wg21belfast/CoreWorkingGroup/cwg_defects.html#2404) has shown.

Like DRY SPOT, it's more economic to figure out the general rule separately. And as by now we have good transcripts, unlike at many points in the past when the relevant rules were created, just the discussion will capture the relevant information. Even if no changes are made to the WP.

---

[1] * *From CWG discussion on P0840R1 in JV2018: " *****: 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."*

But we can discover some places where the behavior can be improved too. And the wording group can refactor the normative text to reflect the intent and the principles instead of symptoms and manifestations.

Changing the SL rules even by just removing artificial obstacles and overlooks extending the set imposes risks (see later), and some not easy to evaluate/estimate. For that reason we suggest a 2-step plan, first completely ignoring the risks and establish the "ideal", a "what would we do having full freedom" without worry or consequences. Then use step 2 to evaluate the risks attached to what actually is wanted and work out a solution with acceptable balance.

This revision of the paper covers input for phase 1 only, what it states "proposed" is not meant directly for the WP but for the phase 2 input.

History

In the initial standard we had POD (plain old data) to cover interfacing, mostly with C. (in C++03 see 8.5.1 [dcl.init.aggr] p1 and 9 [class] p4).

This got a major rework for C++11 with the bulk in the POD revisited paper N2342 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2342.htm) that created the initial definition of SL. (in C++11 see 9 [class] p7). Since then the spirit is unchanged while the text evolved to cover corner cases related to pointer-interconvertibility, strict aliasing, bitfields that were missed at wordsmithing.

Issue #1: access specifiers

In C++03 access specifier labels (ASL) were used for ordering the layout in [exp.rel] and [class.access.sped]: members separated by them could be placed in unspecified way. That was not consistent with POD, that required all members being public, but said nothing about ASL. So a struct with several public: labels inside counted as POD while was allowed to have incompatible layouting. (We assume that being a defect in the standard at the time.)

N2342 changed the rules at both points: the labels lost direct impact, and ordering was kept between with the same access. This ABI-breaking change was passed with CWG comment that no one really used the swapping license, so this change had no actual impact on reality, along with the previously mentioned bug.

While SL did no longer required being all-public, only have the same access for every member. This removed the inconsistency bug. And extended SL to new cases, like class with all private members.

The paper does not talk about the underlying principles that makes mixed access losing SL. But the change makes it clear that the private access itself is not a reason. We assume this is just an artifact and another wordsmithing mistake, that goes against the idea of SL. The ideal phrasing would define SL in terms of member placement. instead of the access levels that happened to influence the placement at the moment. And wording of the SL rules just copy/pasted the layouting rule. And we propose that if swapping on mixed ASL was not allowed the SL rule would not have that restriction either.

A current paper in flight, P1847 *"Make declaration order layout mandated"* (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1847r2.pdf) proposes to remove that swapping license, arguing it is not used in practice. We propose that if that change gets accepted we also remove the part related to access in the SL rules.

In particular we suggest that if class A { int a, b; int c; }; is SL then class B { int a, b; public: int c; }; that has the exact same layout being not SL makes no sense.

Issue #2: member swapping facilities

We have several papers in flight that aim to rearrange the member placement in the layout. One example is P1112 *"Language support for class layout control"*. (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1112r3.pdf).

For the sake of an example, suppose we can mark a struct with layout(alpha) that forces members placed in alphabetic order. Having it on struct BA { int b, a; }; would place A first. Obviously this struct is no longer SL and has to be treated that way. But what about struct AB { int a, b; }; ? This structure is created in the exact same way with or without the mark. Should it remain SL or lose it on the grounds that *was* reordered, just the new order is the same as old by accident?

In other words do we tie SL-ness to the outcome or the potential? If we look at SL as interfacing facility, from the perspective of the other party all we see is the outcome. And if it is good for the client why would it not be for C++? OTOH, the decision of N2342 might have been deliberate and fixed even for implementations strictly refusing to swap.

We propose to go with the former approach that looks closer to the expressed SL description and provides more utility for the programmers. Also this is consistent with the suggestion for issue #1. However with that we look ahead of way more work in phase 2. But if this verdict is taken but tuned back later to the other, it will be clear that the reason is pragmatism and risk management, not the idea itself.

Impact and risks from a change

While the spirit of SL may be what is stated at the beginning of the paper, it has plenty of strings attached in C++. Let's see that is the fallout if our example class B from above starts counting as SL:

- std::is_standard_layout_v<B> will change to true. This may result generating/selecting different functions, send code execution on a different path or create an indirect ABI break for a binary component if used.

- offsetof now works for B. It's a clear benefit for the user, but the compiler must provide the related core constants. (looks no extra burden compared to similar code using A)

- common initial sequence and layout compatibility kicks in, also pointer-interconvertibility with first member (can't see direct problem, but related queries like is_pointer_interconvertible_with_class same as first bullet).

- the mandatory empty base optimization kicks in (redundant)