Strategy for removing safety-related UB by default

Document Number: **P3436 R0**
Date: 2024-10-12
Reply-to: Herb Sutter (herb.sutter@gmail.com)
Audience: SG23, EWG

Contents

Abstract

This paper proposes that we take a first pass toward removing safety-related undefined behavior (UB) by default from C++ by:

a) systematically cataloging the UB already prevented in `constexpr` code, and

b) making each such UB case prevented by default in the regular language in C++26/29 in one of two ways:

i) if the cost is cheap enough, make it the language default (as C++26 did for uninitialized locals)

ii) otherwise, make it prevented by default when an applicable safety Profile is active

and providing a way to opt out.

1 Motivation

I think we agree we want to remove *safety-related* undefined behavior (not all UB) by default if possible.

This is specifically a goal of Profiles. The Direction Group's paper [P2759R1] section 5 says (**emphasis** added):

> *"Profiles impose restrictions on use where they are activated. They do not change the semantics of a valid program (**except to turn UB into a specific well-defined behavior** or vice versa)."*

There is similar phrasing in:

- Stroustrup and Dos Reis's [P2687R0] section 6 paragraph 2
- Stroustrup's [P3038R0] section 14, pasting the entire section for convenience:

    14. Undefined behavior

    Undefined behavior (UB) is a difficult and often misunderstood phenomenon. I will not go into details here. UB is being re-examined in the committee (SG12). For the type_safety profile the only UB that absolutely must be eliminated is the so-called "time-travel optimization" where an occurrence of UB is used to eliminate a test on the path leading to it. The range checking and pointer dereference checking turns UB into a well-defined response (§13). Undefined just means that the standard doesn't define the meaning, so giving a well-defined meaning is among the valid alternatives.

2 Observation: `constexpr` already prevents much UB

`constexpr` already does a huge amount of exactly that safety-related UB elimination.

However, we can't just blindly move all those same checks to execution time, because some would incur unacceptable costs (e.g., every `int+int` overflow/underflow; not even C# enables that by default) and be unusable even with an opt-out (users would have to opt-out too often and the language would be effectively too slow by default and no longer really C++).

But I think it could work if we used Profiles to be selective about the default.

3 Proposed approach: Apply `constexpr` UB preventions to the regular language, by default or in a safety Profile

I think this section's approach is a direct expansion of what Stroustrup wrote in [P3038R0]'s short section 14.

This paper proposes that for each case of safely-related UB that is already prevented in `constexpr` code:

- **Prevent it by default in C++,** universally if possible, otherwise in a safety Profile:
    - if the cost is cheap enough, make it prevented (either checked, or changed to specified behavior) as the language default for all code (e.g., as we just did for uninitialized local reads becoming erroneous behavior in C++26)

    - otherwise, make it prevented if a safety Profile is enabled: either by banning the construct that could lead to the UB (e.g., banning all unsafe pointer arithmetic) or by specifying the behavior (e.g., a bounds check violation reporting of some sort)

- **Provide a way to opt out.** All code will need to opt out of almost every UB sometimes, such as in hot loops, just like C++26 provides the `[[indeterminate]]` opt-out to get uninitialized locals.

Whatever the granularity of Profiles is, have each Profile include its related UB, and what the result is if the potential UB is encountered.

3.1 Example: Integer overflow

Integer overflow is a useful example in that could apply to more than one Profile.

1) If an `arithmetic_safety` Profile is enabled, require that all integer operations that could overflow are checked. (And define what happens if a violation occurs.)

2) If a `bounds_safety` Profile is enabled, check only integer overflows that could lead to a bounds-unchecked subscript operation. (Presumably `bounds_safety` would requires subscript operations to be bounds-checked by default, but if the programmer opts out and performs an unchecked subscript, then we should additionally by default prevent an overflowed value from being used as a subscript unless the programmer opts out of that too.)

4 References

[P2816R0] B. Stroustrup and Gabriel Dos Reis. “Safety Profiles: Type-and-resource safe programming in ISO standard C++” (WG21 paper and SG23/EWG presentation, February 2023).

[P2687R0] B. Stroustrup and Gabriel Dos Reis. “Design alternatives for type-and-resource safe C++” (WG21 paper, October 2023).

[P2759R1] H. Hinnant, R. Orr, B. Stroustrup, D. Vandevoorde, M. Wong. “DG Opinion on Safety for ISO C++” (WG21 paper, 2023-01-22.

[P3038R0] B. Stroustrup. “Concrete suggestions for initial Profiles” (WG21 paper, December 2023).