

UB and profile-defined behavior

Document Number: **P4401 R0**
Date: 2026-09-20
Reply-to: Herb Sutter (herb.sutter@gmail.com)
Audience: SG23

Contents

- 1 Motivation: What happens if a profile defines a previously-undefined behavior? 2
 - 1.1 Key question 2
- 2 Concrete instance of the canonical signed-overflow example 2
 - 2.1 Three profiles: Psat, Pwrap, Pthrow 2
 - 2.2 Situation: My team always enforces Psat, and I write code that relies on Psat-defined behavior 3
 - 2.3 The question is: Is that OK? 3
 - 2.4 Status quo summary 3
- 3 Possible ways to handle the situation 4
 - 3.1 Option 1: Say nothing (current default?).....4
 - 3.2 Option 2: Document “don’t rely on profile-defined behavior” 4
 - 3.3 Option 3: Enforce “don’t rely on profile-defined behavior” using Erroneous Behavior 4
 - 3.4 Do something else?.....4
- 4 Endnotes 5
- 5 References 5

1 Motivation: What happens if a profile defines a previously-undefined behavior?

Stroustrup gives the canonical example of how profiles might define signed overflow in [P3984R0] page 3:

“A profile cannot change the semantics of a program beyond defining the meaning of some forms of undefined behavior. For example, signed arithmetic overflow is UB so a profile can define it to be wraparound like unsigned arithmetic (though I wouldn’t do that), to be saturated arithmetic, or to throw an exception. However, wraparound of unsigned arithmetic is well-defined so a profile cannot define to be anything else. This implies that a program without UB will behave identically when compiled with or without a profile enforced.”

For convenience, let **profile-defined behavior** mean behavior that is undefined in the base language when no profile is enforced, but that is defined when a given profile is enforced.

In the above example, three different profiles could respectively define signed overflow to be wraparound, saturated, or throwing profile-defined behavior.

1.1 Key question

The key question is, can code rely on profile-defined behavior for correctness?

All profiles papers emphasize that a fundamental aim of a profile P is to give a set of “guarantees.” The crux of the question here is whether profile-defined behavior is one of those “guarantees.”

I offer two ways to state the question, that I think are equivalent and need to be answered:

- If I compile code cleanly in profile P and it relies on core language UB that P makes defined behavior, can my code **rely for correctness on** that added defined behavior?
- Can I write code that, by relying on P’s defined behavior, is **only guaranteed correct** when P is enforced?

2 Concrete instance of the canonical signed-overflow example

2.1 Three profiles: Psat, Pwrap, Pthrow

Stroustrup gives the canonical example in [P3984R0] page 3: Three Profiles that define signed overflow to saturate, wrap, or throw, respectively. For convenience, let’s give them names:

- When **Psat** is enforced, signed overflow saturates. (Q: Is that a “guarantee” I can rely on?)
- When **Pwrap** is enforced, signed overflow wraps.
- When **Pthrow** is enforced, signed overflow throws.

2.2 Situation: My team always enforces `Psat`, and I write code that relies on `Psat`-defined behavior

On my team, we love `Psat` and it's always enforced.

Let's say that yesterday I wrote some new code that boils down to this (our systems always use 32-bit ints):

```
// Example 1
if (int_with_value_2000000000 + int_with_value_2000000000 > 0) {
    printf("positive");
    // do more happy-path computation
}
else {
    printf("ERROR"); // or assert(false);
}
```

I want this code to always print “positive,” and I add unit tests and regression tests to check that.

Because my team always enforces a profile like `Psat`, my code **is guaranteed to work** as I expect, my tests pass, and I run it at my best customers' sites. I may not even notice that I'm relying on the profiles, especially if I'm a programmer less experienced with the details of C++.

However, my code **must now always** be compiled with `Psat` (or similar profile/flag) to get this guarantee and stay working, otherwise my code and the unit test may/will fail. [1]

(Bonus situation: This code happens to only use the C subset, so internally I share that code as a C library. Because my team compiles C as C++ and with `Psat` enforced, nobody notices that we're relying on the profile's behavior.)

2.3 The question is: Is that OK?

We can ask “is that OK?” two ways:

- Can I as the code author **rely for correctness on** always being compiled with a profile like `Psat` so I know my code prints “positive”?
- Can I legitimately write this code that is **only guaranteed correct** when `Psat` is enforced?

2.4 Status quo summary

According to the current profiles design and discussion, I think everyone agrees that:

- If compiled with `Psat`, this is well-defined and prints “positive.”
- If compiled with `Pwrap`, this is well-defined and prints “ERROR” (or asserts).
- If compiled with `Pthrow`, this is well-defined and throws an exception.
- If compiled without profiles, this is UB.

If anyone compiles this code without `Psat`, or ships it as a library that someone builds without `Psat`, or cut-and-pastes it into another source base with different/no profiles enforced, the code might not behave as intended and tested.

3 Possible ways to handle the situation

There are several possible ways to address this.

3.1 Option 1: Say nothing (current default?)

We could say we don't care about any code that is UB today and leave it to programmers to figure this out, such as by always compiling their whole programs with the same profiles they depend on, similar to how they take care today to compile with compiler flag settings that may enable nonstandard extensions/behaviors.

3.2 Option 2: Document “don't rely on profile-defined behavior”

We could say that programmers should not rely on any well-defined behavior added only when a profile is enforced.

However, there are two drawbacks we would have to answer:

- What example benefits from adding the specific well-defined behavior if they cannot rely on it?
- How do we avoid Hyrum's Law *de facto* sliding this back into Option 1?

3.3 Option 3: Enforce “don't rely on profile-defined behavior” using Erroneous Behavior

If we want to let two profiles define a given case of UB in different ways, we could:

- make that specific UB case instead **EB in the base language**, and
- say that a profile **must keep it as EB** but may change what the defined behavior is.

This way, code that performs what was formerly UB in the base language is now **always EB, in the base language and regardless which/no profiles are enforced**. Thus there is no divergence problem, because the code is always EB regardless whether any/all/no profiles are enforced, and is therefore always on notice that it cannot rely on any specific behavior (EB already means “this code is considered erroneous even though it compiles”).

Option 3 would enable us to achieve the nice clear statement that “for all non-erroneous programs, a profile is a pure subset.”

I think Option 3 could also let us link mixed programs (and even mixed TUs) compiled with either/any/all of [P_{sat}](#), [P_{wrap}](#), and [P_{throw}](#), saying “you can build with two profiles that define different behaviors for the same EB and you'll get one of them (you can't predict which one, similarly to COMDAT folding or different behaviors post-inline) and that's okay because what really matters is that the code is EB always so you're on notice that trying to rely on this is formally and officially ‘erroneous.’”

This still has the drawback to answer: Can we show two examples where each example benefits from adding a different specific well-defined behavior, that is still useful if neither example can rely on that behavior?

3.4 Do something else?

Other options exist, such as to make it ill-formed to build a program (or a TU) with a mix of profiles enforced if more than one of those profiles defines the same base language UB. Such options would need be specified in more detail (e.g., what kinds of conflicts are disallowed? how are the conflicts detected? is detection guaranteed or IF-NDR? etc.).

4 Endnotes

[1] I’m avoiding the word “dialect” only because it has repeatedly distracted from discussing the examples to requests to define “dialect.” For what it’s worth, though, Stroustrup and Direction Group introduced the term “dialect” to the profiles discussion and I think have given definitions of what they mean by it: They describe a dialect as something that *defines a superset* (adds features/behavior) and that *breaks the property that a piece of code means the same thing* regardless of which profile (if any) is enforced.

- In [P2687R0] page 3: “The problem then becomes that **the design of the safe dialect becomes the design of a superset to offer the safe features** plus the design of interfaces between the novel safe features and parts of the old language deemed safe.”
- In [P2759R1] page 9: “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). In particular, **a piece of code means the same in every profile (or no profiles). This property is the crucial difference between dialects and our approach.**” — The paragraph does not explain how the first half of the exception does not conflict with the crucial difference, in cases like Example 1.

I’m happy to use any term, or no term at all. The important thing to address is: What should Example 1 do?

5 References

[P2687R0] B. Stroustrup. “Design alternatives for type-and-resource safe C++” (WG21 paper, Oct 2022).

[P2759R1] H. Hinnant et al. “DG opinion on safety for ISO C++” (WG21 paper, Jan 2023).

[P3984R0] B. Stroustrup. “A type-safety profile” (WG21 paper, Feb 2026).