A framework for systematically addressing undefined behaviour in the C++ Standard

Timur Doumler (papers@timur.audio)
Joshua Berne (jberne4@bloomberg.net)

Document #:	D3100R5
Date:	2025-11-07
Project:	Programming Language C++
Audience:	EWG

Abstract

In this paper, we enumerate all cases of core language undefined behaviour explicitly specified in the C++ Standard, group them into ten categories, and classify them along a number of relevant criteria.

We then present a holistic strategy for systematically detecting, mitigating, and ultimately eliminating such undefined behaviour from the C++ Standard. This strategy is built on top of seven basic tools: feature removal, refined behaviour, erroneous behaviour, insertion of runtime checks, language subsetting, the introduction of annotations, and the introduction of entirely new language features. We discuss which tools are applicable to which cases of core language undefined behaviour.

We find that two of these tools — erroneous behaviour and runtime checks — are applicable to a wide range of existing cases and do not require any source changes. We describe how runtime checks can be systematically introduced via *implicit contract assertions*, giving users complete control over what impact that undefined behaviour has on their programs. In addition to runtime checking, we replace undefined behaviour with erroneous, but well-defined behaviour that allows the program to continue execution past a violated implicit contract assertion wherever possible.

Building on Contracts as adopted for C++26, we provide a generic framework for applying these two techniques across the *entire* C++ language specification, fundamentally changing the landscape of how undefined behaviour is approached in C++.

Contents

Revision history

— **R5**, 2025-11-07: Added one more case of UB that was initially overlooked because it did not use the word “undefined’ in the wording

— **R4**, 2025-08-13: Updated the paper to align with [P3754R0]; incorporated feedback from the WG21 meeting in Sofia; removed items from UB list that should not be specified as UB and have Core issues to make it so; added discussion of interaction with `noexcept`; improved paper structure; changed title to reflect the wider scope.

— **R3**, 2025-06-28: Removed preprocessor UB due to adoption of [P2843R3] into C++26.

— **R2**, 2025-05-19: Complete rewrite after the WG21 meeting in Hagenberg.

— **R1**, 2024-10-16: Complete rewrite after the WG21 meeting in St. Louis.

— **R0**, 2023-03-08: Initial version.

1 Introduction

Eliminating or at least meaningfully reducing the amount of *undefined behaviour* (UB) is an important objective for the future evolution of C++. WG21 has been continuously working in that direction. For a recent status update, see [Sutter2025] and references therein; for background, see [Sutter2024] and references therein.

At WG21's February 2025 meeting in Hagenberg, EWG agreed on a ship vehicle for a systematic treatment of core language UB in C++: the pursuit of a *core language UB white paper* [P3656R1] in the C++26 timeframe, covering erroneous behaviour (EB), Profiles, and Contracts. The proposed process involves starting with an empty white paper working draft and then iteratively get EWG approval for papers to be adopted into that working draft.

This paper directly addresses the major work items proposed by [P3656R1]. At WG21's June 2025 meeting in Sofia, EWG reviewed revision R2 of this paper and approved its adoption into the core language UB white paper working draft.

In Section 2, we identify and enumerate all core language UB explicitly specified in the current C++ working paper [N5008]. We group all core language UB into ten categories. We then classify cases of UB along several relevant criteria, such as whether they are locally diagnosable, how expensive that diagnosis is, and in which cases the UB can be replaced with meaningful, well-defined behaviour.

In Section 3, we present a holistic strategy for systematically detecting, mitigating, and ultimately removing UB across the entire C++ programming language specification. The present revision of this paper incorporates the presentation [P3754R0] we gave to EWG in Sofia outlining this strategy. It also includes an updated version of the diagram on slide 53 from that presentation (a.k.a. the "magic slide") visualising this strategy. EWG reached consensus in Sofia to use this diagram as a basis for the core language UB white paper.

The proposed strategy is composed of seven basic tools: feature removal, refined behaviour, erroneous behaviour, insertion of runtime checks, language subsetting, the introduction of annotations, and the introduction of entirely new language features. We use the results of the analysis in Section 2 to take a first pass at identifying which tools would be applicable to which cases of UB. We find that the conditions under which UB will occur can, in many cases, be identified by a runtime check. In addition, there are a significant number of cases where UB can be replaced by well-defined erroneous behaviour following a failed check. These two techniques are applicable to a wide range of existing cases of UB and, importantly, do not require any source changes.

In Section 4, we present a generic framework for applying these two techniques across the entire C++ language. The proposed design has been reviewed and approved by SG21, SG23, and EWG. We describe how runtime checks can be systematically introduced via *implicit contract assertions*, building on the basic framework of Contracts adopted for C++26 via [P2900R14] and giving users complete control over what impact that undefined behaviour has on their programs. In addition to runtime checking, we replace UB by erroneous, but well-defined behaviour that allows the program to continue execution past a violated implicit contract assertion wherever possible. We also propose an escape hatch to mitigate the runtime cost of such well-defined replacement behaviour and avoid performance regressions in existing C++ programs.

In Section 5, we propose wording for approval into the core language UB white paper [P3656R1] that implements the framework presented in Section 4.

Finally, in Section 6, we discuss how future extensions, such as Labels [P3400R1], will enable programmatically identifying the category of UB that has occurred and provide us with granular, in-source control of the evaluation semantics for implicit contract assertions.

2 Analysis

2.1 Enumeration

2.1.1 Methodology

We manually inspected all occurrences of the words “undefined” and “assume” in revision [N5008] of the C++ working paper. We then constructed a list of all cases of explicitly specified core language UB. We found 80 instances of explicit language UB introduced with phrases containing the word “undefined” (“...the behaviour is undefined”, “...has undefined behaviour”, “...results in undefined behaviour”, etc.) and one instance of explicit language UB introduced with a phrase containing the word “assume” (“the implementation may assume...”).[1] We thus obtained a list with 81 cases of explicit language UB; it can be found in Appendix A of this paper.

Each individual case of UB in our list has a stable identifier. We place those identifiers between {curly braces} to visually distinguish them from the C++ working paper’s clause identifiers, which we place between [square brackets].

There is currently an open pull request against the C++ working paper that proposes to add these identifiers directly to the C++ working paper alongside a new appendix to that document enumerating all cases of UB and providing explanation and code examples. This pull request and the stable identifiers used therein are fully consistent with Appendix A of this paper.

Originally, we constructed our list independently from another effort to enumerate core language UB led by Shafik Yaghmour (see [P1705R1], [P3075R0]). The current list in Appendix A and in the pull request mentioned above represents a merger of the lists produced by both efforts, further increasing confidence that we in fact exhaustively covered all core language UB explicitly specified in the current C++ working paper.

2.1.2 Granularity

The granularity of our enumeration — i.e., when a piece of core language specification is considered a distinct case of UB — is somewhat arbitrary. As a general rule, we consider a single sentence in the C++ working paper that specifies a condition under which a core language operation has UB to be one case of UB; if such a sentence contains a bulleted list where each item specifies such a condition (e.g., [basic.life]/7), we consider each item to be a separate case of UB.

We could consider dividing up the cases of UB in different ways and with greater granularity. In particular, we could more closely align the enumeration with possible mitigation strategies, rather than with the lexical appearance of the word “undefined” in the C++ working paper.

For example, for [basic.life]/7, instead of considering each bullet to be a separate case of UB, we could consider each different way in which a pointer can be invalid (lifetime of the object not started yet, lifetime of the object ended already, pointer is null, etc.) to be a separate case of UB.

As another example, instead of considering flowing off the end of a function ([stmt.return]/4) as a single case of UB, we could introduce separate cases for flowing off the end of an assignment operator (which has a proposed mitigation, see [P2973R0]), flowing off the end of a function that returns a built-in type (which could be mitigated via returning an erroneous value, see Section 2.4), and flowing off the end of any other non-`main` function that returns a non-`void` type.

[1] Except for this one occurrence, which is in [intro.progress]/1, the verb “assume” in core wording does not imply that there may be undefined behaviour, but instead means things like “this case will be treated as the default” or “other cases will be ignored”. To remove ambiguity, [CWG2816] proposes to change the wording in [intro.progress]/1 to use the phrase “the behaviour is undefined” instead of “the implementation may assume”.

However, such alternative approaches would make the enumeration more difficult to map to existing wording, which in turn would make it more difficult to track changes and reason about whether the enumeration is exhaustive. Further, keeping the enumeration of cases of UB separate from the enumeration of possible mitigation strategies reduces interdependencies and complexity. Instead, whenever a mitigation strategy does not fully align with a case of UB, we point this out explicitly (for example, inserting an implicit null pointer check is considered only a *partial* mitigation for dereferencing an invalid pointer).

2.1.3 Scope

In this paper, we consider only explicit UB, not implicit UB — that is, UB that exists by omission because the C++ working paper failed to specify the behaviour of a well-formed operation. We consider all cases of implicit UB that may be discovered in the future to be wording bugs that should be addressed via Core issues, and then subsequently become explicit UB that can be addressed via the framework proposed here. Importantly, while explicit UB can be enumerated exhaustively, and we do so in Appendix A of this paper, enumerating all implicit UB is impossible in principle as we can never be sure we found all wording bugs in the C++ working paper.

We exclude one case of explicit UB from our enumeration that does not actually represent a separate case of UB. The current wording in this case[2] normatively states that "the program has undefined behaviour" but merely refers to cases of UB already specified elsewhere in the Standard, rather than specifying any new such cases. The relevant wording should instead be a non-normative note; this issue is currently being addressed by Core issue [CWG3022].

Further, in this paper we consider only *runtime* UB, not compile-time or link-time issues. We therefore exclude all cases of IFNDR from our analysis: although the effects of IFNDR can manifest at run time, whether a program is IFNDR is (unlike UB) fundamentally not a runtime property. We also exclude the case of infinite recursion during template instantiation. The current wording[3] for this case can be interpreted to mean that such infinite recursion could cause runtime UB; however, this interpretation makes little sense as failure during template instantiation is fundamentally a compile-time issue. We therefore consider this case a wording bug rather than an instance of UB; this issue is currently being addressed by Core issue [CWG3034].

Finally, we exclude *library undefined behaviour*. The natural mitigation approach for library UB is to make use of contract assertions (`pre`, `post`, and `contract_assert`) in library implementations and, where sensible, mandate such assertions through library hardening [P3471R4], both of which are out of scope for this paper. We, therefore, consider only UB that is specified in the core language part of the C++ working paper (Clauses 1–15). Further, we found one case of UB that is specified in the core language part of [N5008] but actually represents a precondition on Standard Library functions;[4] that case is, therefore, also excluded from our list.

[2][class.dtor]/16: The invocation of a destructor is subject to the usual rules for member functions ([class.mfct]); that is, if the object is not of the destructor's class type and not of a class derived from the destructor's class type (including when the destructor is invoked via a null pointer value), the program has undefined behavior.

[3][temp.inst]/16: There is an implementation-defined quantity that specifies the limit on the total depth of recursive instantiations ([implimits]), which could involve more than one template. The result of an infinite recursion in instantiation is undefined.

[4][basic.start.term]/6: If there is a use of a standard library object or function not permitted within signal handlers ([support.runtime]) that does not happen before ([intro.multithread]) completion of destruction of objects with static storage duration and execution of `std::atexit` registered functions ([support.start.term]), the program has undefined behavior.

2.2 Classification

2.2.1 Categories

We found that all identified cases of core language UB can be classified into ten basic categories:

I. **Initialisation** — 1 case. Evaluating an expression that produces an indeterminate value.

II. **Bounds** — 5 cases. Using a pointer in a way that fails to respect the range of the pointed-to object or array. Examples: incrementing a pointer beyond the past-the-end position; performing single-object delete on an operand obtained from an array-new expression; dereferencing a pointer returned from a request for zero size.

III. **Type and Lifetime** — 51 cases. Operations that access storage and/or use pointers or references to storage in an inappropriate way that is not already covered by Initialisation and Bounds. Examples: attempting to access a value of one type through a pointer of a different, incompatible type; attempting to access the value of an object after its lifetime has ended.

IV. **Arithmetic** — 9 cases. Executing an arithmetic operation whose operands fail to meet certain preconditions. Examples: division by zero; conversion of a value to a different arithmetic type that cannot represent that value.

V. **Threading** — 2 cases. Data races; non-trivial infinite loops with no side effects.

VI. **Sequencing** — 1 case. Performing two concurrent accesses, at least one of which is modifying, to the same memory location from the same thread where neither access is sequenced before the other.

VII. **Assumptions** — 1 case. Reaching an `[[assume]]` declaration whose operand would not evaluate to `true`.

VIII. **Control Flow** — 5 cases. Errors in control flow. Examples: flowing off the end of a function; re-entering the same declaration recursively when initialising a static variable.

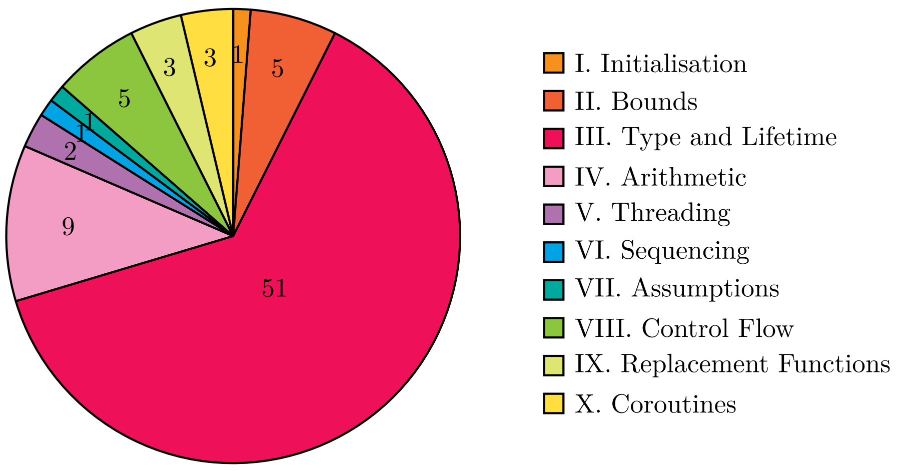

Figure 1: Distribution of identified cases of explicit language UB across specified categories

IX. **Replacement Functions** — 3 cases. Executing a user-defined replacement function (`operator new/delete`) that fails to meet the specified requirements. Examples: returning `null` from a user-defined placement `new`; throwing an exception from a user-defined `delete`.

X. **Coroutines** — 3 cases. Misusing coroutine machinery. Examples: destroying a coroutine that is not suspended; failing to provide a `return_void` function for a coroutine that does not return a value.

Figure 1 shows the distribution of the 81 identified cases of explicit core language UB across these ten basic categories.

The categories of Initialisation, Bounds, and Type and Lifetime correspond to the common terms *initialisation safety*, *bounds safety*, *type safety*, and *lifetime safety*, respectively, and collectively represent UB that is commonly referred to with the umbrella term *memory safety*. Much of the ongoing work around how to "make C++ safe" is focused on these categories (see [P3081R2], [P3700R0], and references therein).

Because unambiguously categorising a particular case of UB into either *type safety* or *lifetime safety* is often impossible since it concerns both, we grouped them into a single combined category, Type and Lifetime. While some cases of UB are primarily caused by type aliasing and others are primarily caused by out-of-lifetime accesses, they form a spectrum, and many common operations in C++ (e.g., using a reference) rely on *both* type and lifetime constraints to be satisfied.

Remarkably, these three categories related to memory safety account for 57 cases of UB, or 70.4% of all identified cases; the Type and Lifetime category alone accounts for 51 cases of UB, or 63.0% of all identified cases.

The next two categories, Arithmetic and Threading, correspond to the common terms *arithmetic safety* and *thread safety*, respectively. Note that the term thread safety is often used to refer to data races specifically and does not include the other case of UB we placed in the Threading category (infinite loops with no side effects).

The following category, Sequencing, contains just one case of UB: unsequenced operations, such as `i++ + ++i`. Classifying UB due to data races and unsequenced operations into two separate categories might seem surprising at first since they have a very similar same shape (except that one is inter-thread and the other is intra-thread), but as we will see in Section 3, these two categories actually require very different approaches to mitigation.

The next category, Assumptions, also contains just one case of UB: reaching an `[[assume]]` declaration whose operand would not evaluate to `true`. As we will see later, this case of UB is of a different nature than the others and warrants its own category.

The final three categories (Control Flow, Replacement Functions, and Coroutines) contain a handful of cases of UB that are less frequently discussed in the current "safe C++" discourse.

2.2.2 Relevance for security

[P3656R1] asks which cases of UB are security related. The paper suggests having security experts indicate which cases of UB have security impact and use "always", "never", and "sometimes" tags. We are not security experts, so we do not attempt to do this here. However, we note that cases of UB commonly associated with security vulnerabilities (see, for example, the CWE list at `https://cwe.mitre.org/`) fall into the Initialisation, Bounds, and Type and Lifetime categories.

UB in other categories is not commonly exploited by malicious attackers to our knowledge. Nevertheless, some of these cases, for example those in categories Arithmetic and Threading, are a common source of program defects that do sizeable damage to existing software.

In principle, with aggressive optimising compilers any form of UB can lead to unpredictable defects and vulnerabilities. Mitigating cases of UB currently considered to be the most critical security concern will simply remove the easiest routes of attack from the table, and any UB not yet addressed may become the new major candidate for attackers to leverage for nefarious purposes. Therefore, prioritising implementation based on current trends amongst malicious actors, though helpful, should not be used to limit the scope of our work on improving the C++ language specification (see [Sutter2024], [P3500R1], and [P3578R0]).

2.3 Diagnosability

The second question [P3656R1] asks is which cases of UB are "efficiently locally diagnosable". In this paper, we split this question into three separate questions: whether diagnosis can happen statically or whether it needs to happen dynamically (i.e., at run time), whether such diagnosis can be performed locally, and the runtime cost of such diagnosis (whether performed locally or not).

2.3.1 Static vs. dynamic diagnosis

A commonly asked question is "why does the committee not simply make all the UB ill-formed instead"? The answer is that in order for that to happen, it would be necessary to determine statically — i.e., at compile time — whether a given operation in a C++ program would lead to UB when executed at run time. However, whether a C++ program will have UB when executed is fundamentally a *runtime* property, i.e., the answer depends on runtime values unknown at compile time (for example, the runtime value of a pointer or an integer). Therefore, in the vast majority of cases, such a compile-time determination cannot be made. In fact, in our entire list of 81 cases of UB, we cannot identify *any* cases that can unconditionally be diagnosed at compile time.

Of course, static analysis is still useful and widely used in the field. There are many situations where static analysis *can* detect a bug that would lead to UB when executed, because the relevant values or conditions happen to be known at compile time in the particular program at hand. However, given an existing C++ program, any approach based on static analysis fundamentally has to choose between *false positives* (rejecting code for which no proof can be constructed one way or another, even if that code happens to be correct) and *false negatives* (accepting incorrect code that will lead to UB when executed).

Crucially, whether a pointer or reference refers to a valid object of the correct type within its lifetime at a given point in time ("memory safety"), the relevant property for addressing UB in the Initialisation, Bounds, and Type and Lifetime categories, seems to be fundamentally unprovable at compile time in the general case (see [Baxter2024]).

As we will see in Section 3, despite these fundamental limitations there are things we can (and should) do in the C++ Standard to enable static analysis to construct a proof in more cases, such as subsetting the language (3.3.5), providing replacement features (3.3.7), and adding annotations (3.3.6). However, for existing C++ programs, reliably detecting cases of UB without rejecting correct code will inevitably have to leverage *runtime* detection, i.e., the insertion of additional runtime checks when compiling the program. We therefore focus on such runtime detection for the remainder of this analysis.

2.3.2 Locality of diagnosis

An important property for diagnosis of UB is whether such diagnosis (whether static or dynamic) can be performed *locally*, i.e. without keeping track of additional information across the entire program that is not available within the C++ abstract machine (achievable with additional instrumentation of the kind that is implemented in sanitisers, such as ASan and UBSan) and without analysing code in other branches (which is limited by the Halting problem) or on the other side of a function call boundary (which might be located in another TU and therefore inaccessible).

Most cases of UB in the Initialisation, Bounds, and Type and Lifetime categories are, in general, *not* locally diagnosable. They could potentially be made locally diagnosable in the future with the introduction of novel features such as lifetime annotations [P2771R1] or "ghost data" [Lippincott2025] which are the subject of ongoing research. We discuss some of these approaches in Section 3; here, we focus on the status quo in Standard C++.

In the Bounds category, {expr.add.out.of.bounds} and {expr.add.sub.diff.pointers} are partially locally diagnosable (only if the array bound is statically known). In the Type and Lifetime category, {expr.static.cast.downcast.wrong.derived.type}, {expr.unary.dereference}, {conv.ptr.virtual.base}, and {expr.dynamic.cast.lifetime} are partially locally diagnosable (for the null pointer case). {expr.mptr.oper.member.func.null} is locally diagnosable because this case requires *only* a null pointer check. {basic.align.object.alignment} is locally diagnosable by checking the alignment of storage when creating an object at run time. {expr.assign.overlap} is locally diagnosable by checking the overlap of the two address ranges. (The ranges are known because the address and `sizeof` are known at run time for both the source and the destination object.) {class.abstract.pure.virtual} is locally diagnosable by adding a runtime check to the pure virtual function stub to which the base class vtable points. None of the other cases of UB in the Initialisation, Bounds, and Type and Lifetime categories are locally diagnosable.

All cases of UB in the Arithmetic category are locally diagnosable since they are all cases of an arithmetic operation producing a value that is somehow inappropriate (mathematically invalid, not representable in the target type, etc.) and that value can be inspected at run time.

UB in the Threading category is either not locally diagnosable ({intro.races.data}) or not diagnosable at all ({intro.progress.stops}. However, UB in the Sequencing category ({intro.execution.unsequenced.modification}) is locally diagnosable.

UB in the Assumption category ({dcl.attr.assume.false}) is, in principle, locally diagnosable by evaluating the operand of the assumption and verifying that the resulting value, contextually converted to `bool`, equals `true`. However, if that evaluation has any side effects, such a check could alter the observable state of the program. Therefore, even if the given assumption holds and no UB occurs, the check itself might render the program invalid by altering its state. Thus, this case of UB is meaningfully diagnosable in any automated fashion only if the operand has no side effects when evaluated. However, proving that the operand has no side effects is generally impossible to do efficiently and is outright impossible in the presence of an opaque function call. We therefore consider this case not diagnosable at all.

Two cases of UB in the Control Flow category are locally diagnosable. {stmt.return.flow.off} can be diagnosed by inserting a check at the end of every function body that does not end with a `return` statement. {dcl.attr.noreturn.eventually.returns} can be diagnosed by inserting a check into every function declared `[[noreturn]]`. The remaining three cases of UB in that category are not locally diagnosable.

Some cases of UB in the Replacement Function category are partially or fully locally diagnosable. In particular, some of the constraints specified in {basic.stc.alloc.dealloc.constraint} and {expr.new.non.allocating.null} are locally diagnosable, while others are not. In particular, we can check locally that a deallocation function does not exit via an exception and that an allocation function does not return null. However, checking the other constraints (locally or at all) is generally not possible.

Finally, one case of UB in the Coroutine category, stmt.return.coroutine.flow.off}, is locally diagnosable in a way analogous to {stmt.return.flow.off} by inserting a check at the end of a coroutine for which no `return_void` function is provided. The remaining two cases of UB in that category are

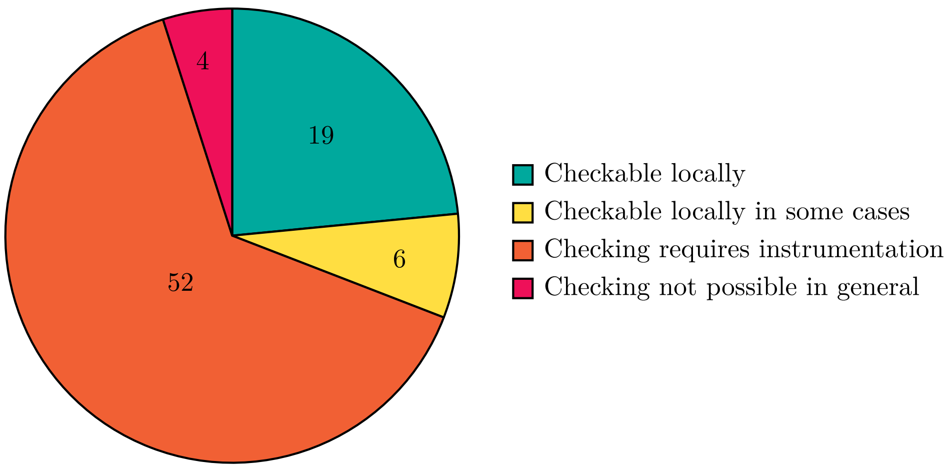

Figure 2: Runtime diagnosability of explicit core language UB

not locally diagnosable since being so would require tracking runtime state information that is not currently maintained within the coroutine handle in most implementations.

Overall, as shown in Figure 2, only 19 cases of UB (23.5% of all cases) are unconditionally locally diagnosable at run time, while 58 cases of UB (72.5%) are not; these cases require instrumentation of the code in order to be diagnosed at run time. We will discuss these requirements in more detail in Section 2.3.4. In addition, 4 cases of UB (4.9%) are not checkable at all in the general case, even with additional instrumentation.

2.3.3 Cost of local diagnosis

Considering locally checkable cases of UB separately from non-locally checkable ones is useful to estimate the cost of diagnosis. For locally diagnosable cases, some kind of runtime check — an *assertion* — could be inserted by the implementation and then evaluated at run time. The total cost of diagnosis is, therefore, equal to the cost of evaluating that check multiplied by the number of times the check needs to be evaluated.

Note that in this paper, we study the theoretical, relative cost based on the current specification of the C++ language. We do not, however, measure the actual cost in existing tooling that implements such checks, nor do we present benchmarks in this paper; such studies are left for future work.

That said, the cheapest kind of check — and the only one that has (almost) no overhead for the happy path — is the "fail if you get here" check, equivalent to a `pre/post/contract_assert(false)`. This kind of check is sufficient to diagnose {class.abstract.pure.virtual}, {stmt.return.flow.off}, {stmt.return.coroutine.flow.off}, and {dcl.attr.noreturn.eventually.returns}.

A slightly more expensive but still cheap and optimiser-friendly kind of check is a null check, required to diagnose the null pointer cases ({expr.static.cast.downcast.wrong.derived.type}, {expr.unary.dereference}, {conv.ptr.virtual.base}, {expr.dynamic.cast.lifetime}, {expr.mptr.oper.member.func.null}, and {expr.new.non.allocating.null}) as well as division by zero ({expr.mul.div.by.zero}).

Integer comparisons are similarly cheap and optimiser-friendly and are required for bounds checks with statically known array bounds ({expr.add.out.of.bounds} and {expr.add.sub.diff.pointers}) as well as for {expr.shift.neg.and.width} and {intro.execution.unsequenced.modification}.

Beyond this, a number of UB cases can still be checked by a straightforward arithmetic expression but with increasingly expensive expressions: {expr.assign.overlap} requires computing whether two integer ranges overlap, and {basic.align.object.alignment} requires computing an integer modulo.

At the expensive end of the locally diagnosable UB spectrum are runtime checks for which there is no corresponding C++ expression; instead, the compiler would have to generate more complex "magic" checks based on knowledge unavailable in the C++ abstract machine. In particular, this case applies to all arithmetic UB except {expr.add.out.of.bounds} and {expr.add.sub.diff.pointers}. The compiler would have to validate the bit patterns of values of arithmetic types according to knowledge it has about how values of such types are represented on the targeted platform. Such checks can be done locally, but they can slow operations involving built-in types and, in particular, floating-point types.

In addition to the cost of the check itself, we need to consider the frequency with which these checks would need to be done. Checks that would need to happen once when a function is called or when a function returns are likely to be acceptable in most scenarios. Extensive checks for arithmetic UB will probably be acceptable in fewer scenarios because such checks have the potential to significantly slow arithmetic operations, which are performance sensitive in many contexts. On the extreme end, if we wanted to diagnose {intro.execution.unsequenced.modification} via a runtime check, the check itself would be fairly inexpensive, but the compiler would have to identify all potential read operations that are not sequenced with respect to each given write operation and then insert checks to identify if those operations are actually going to reference the same address.

2.3.4 Cost of non-local diagnosis

For UB that is not locally diagnosable (which is most of the UB in C++), we need to consider the cost of the required additional instrumentation. To get an idea of that cost, we must nail down exactly which additional properties that are not normally known from within the C++ abstract machine would need to be tracked by such instrumentation. This tracking would need to happen at run time throughout the *entire* program; checks relying on the tracked information would have to be inserted for *every* runtime operation that may be affected by such UB. The full list is available in Appendix A; we provide an overview below.

To diagnose *all* cases of UB in the memory safety categories of Initialization, Bounds, and Type and Lifetime, instrumentation would have to track all the following properties:

— Provenance of all pointers and pointers-to-member

— For all storage, whether it has been allocated or freed

— For all storage, whether it has been initialised

— For all storage, whether it has been created such that it can hold implicit lifetime objects

— For all storage, the type of the object associated with it (if any), including whether it is `const` or `volatile`

— For all objects, whether their lifetime has been started or ended

— For all objects, whether they are currently being constructed or destroyed

— The dynamic type of all *non*-polymorphic objects of class type

— For all references, whether they have been initialized

— For all addresses that point to functions, the type of the function

To diagnose UB in the Threading category, instrumentation would have to track, for *all* memory accesses, from which threads that memory is accessed and when these accesses synchronise with each other. Doing this exhaustively is not practically possible; however, instrumentation that is capable of diagnosing a subset of cases exists in the form of sanitisers (TSan).

The non-locally-diagnosable UB in the Control Flow category concerns operations that are not allowed during construction and destruction of objects with static or thread-local storage duration ({basic.start.main.exit.during.destruction} and {basic.start.term.use.after.destruction}). To diagnose these, instrumentation would have to insert guards tracking whether such objects are currently being constructed and destroyed.

Finally, to diagnose {dcl.fct.def. coroutine. resume.not. suspended} and {dcl.fct.def. coroutine.destroy.not.suspended} in the Coroutine category, instrumentation would have to track the suspension state associated with every coroutine handle.

As we know from existing sanitisers, such instrumentation is expensive enough that it is almost never affordable in production. If we were to add instrumentation covering *all* of the above, we would remove vast swathes of UB from the language, but performance would worsen by an order of magnitude, unless special hardware-acceleration or some other radically new technology for these checks becomes available.

Given the substantial overhead of such instrumentation in both runtime cost and additional memory consumption, the cost of the actual checks themselves (whether a specific pointer is valid at a specific time, etc.) is not particularly important for non-local diagnosis because the performance penalty would be dominated by the instrumentation overhead.

2.4 Existence of replacement behaviour

For existing code that cannot be modified in-source, removing runtime UB requires redefining the semantics of the affected C++ operations, for the cases where UB would occur today, to have well-defined behaviour instead. A useful question is therefore: for which cases of UB is it actually possible to specify such well-defined *replacement behaviour* in a meaningful way?

For the purposes of this analysis, we need to be careful with delineating what exactly we mean by replacement behaviour. If it is possible to insert a runtime check guarding a particular case of UB (e.g., a bounds check, a null pointer check), we can specify well-defined behaviour for the case when this check fails (e.g., terminate the program, throw an exception) which guarantees that we never actually execute the operation that would have runtime UB, thus avoiding it. However, that does not mean giving well-defined behaviour to the operation *itself.* What we mean by replacement behaviour is that, regardless of the existence of the check, continuing execution and evaluating the operation no longer leads to runtime UB *even if the check failed, or would have failed.*

As we will discuss in more detail in Section 3, we can conceptually distinguish between two types of replacement behaviour — *refined* and *erroneous* behaviour — depending on whether the replacement behaviour is considered correct or incorrect (despite no longer being undefined). In any case, for either type of replacement behaviour to actually happen, the compiler must be able to lay down the necessary instructions at compile time. Simultaneously, as discussed in Section 2.3.1, in the vast majority of cases core language UB is fundamentally *not* diagnosable at compile time, as whether or not the UB will occur depends on runtime parameters. Replacement behaviour can therefore not depend on knowing that an error occurred. For non-locally-diagnosable UB, replacement behaviour also cannot depend on any additional instrumentation being present.

For this paper, we systematically identified all cases of core language UB for which either form of replacement behaviour can be meaningfully defined. This section gives an overview; the full list can be found in Appendix A. As we will see, for most cases of UB, replacement behaviour does not exist, and if it does, it is often not cheap.

For UB in the Initialization category ({basic.indet.value}), replacement behaviour is sometimes possible for built-in types: an operation that would currently return an indeterminate value can be specified to return *some* value instead.

We could consider returning a specific value such as 0, or returning some unspecified value (as a form of refined behaviour). However, doing so removes the ability for tools to recognise that a program defect is present (see [P2754R0]). The most meaningful option is to make it return an erroneous value (a form of erroneous behaviour). For variables with automatic storage duration, this replacement behaviour is already part of C++26 as EB via [P2795R5] because for this case, the replacement behaviour is particularly cheap. The same behaviour could also be employed for dynamically allocated variables but at greater cost (see [P2723R1] Section 6 for discussion).

On the other hand, producing an erroneous value (instead of, for example, the value that happened to be in memory where an object was incorrectly presumed to have been initialised) requires having a point in time where a fallback value can be unconditionally placed in memory, such as when passing the declaration of an automatic variable; there are cases where such a point cannot be determined.

Further, we cannot in general define replacement behaviour for uninitialised variables of user-defined type. Even if we could zero out all the underlying storage for user-defined types (or overwrite it with some other known bit pattern), doing so does not always produce, for that type, a valid value that can be accessed without UB. (Consider a user-defined type that relies on a member pointer always being dereferenceable.) Therefore, {basic.indet.value} does not have replacement behaviour for the general case.

Practically *none* of the UB in the categories of Bounds and Type and Lifetime have any plausible replacement behaviour. The only exception is {conv.lval.valid.representation}: if the bits in the value representation of an object of built-in type are not valid for that type, the compiler could instead coerce the value into an erroneous value.[5] For example, in the code example given in the C++ working paper,

```
bool f() {
  bool b = true;
  char c = 42;
  memcpy(&b, &c, 1);
  return b;          // undefined behavior if 42 is not a valid value representation for bool
}
```

the UB could be replaced by well-defined behaviour by appropriately bit-masking every accessed `bool` value (and considering the result erroneous if the bit-mask operation changed the value). Similar mitigations could be put in place for other built-in types since the space of allowed bit representations for values of those types, for the targeted platform, are known to the compiler. The caveat is that such mitigations would potentially incur a significant performance overhead on many simple operations that involve built-in types.

All UB in the Arithmetic category has the same possible replacement behaviour: if an arithmetic operation would produce an inappropriate value, it can be coerced into some other value instead. We could contemplate refined behaviour in the form of a concrete value (e.g., saturate or wraparound for signed integer overflow, choose the closest valid value for invalid conversions) or erroneous behaviour in the form of an erroneous value being produced. In either case, such replacement behaviour will incur significant performance overhead on common arithmetic operations.

Defining replacement behaviour for UB in the Threading category ({intro.races.data}) is in principle possible: we could make all primitive memory accesses implicitly atomic, as in the Java memory model. The overhead incurred by such a model will heavily depend on the memory model of

[5]This property of {conv.lval.valid.representation} is a potential argument for placing this case of UB into the Arithmetic category instead of the Type and Lifetime category as we did here.

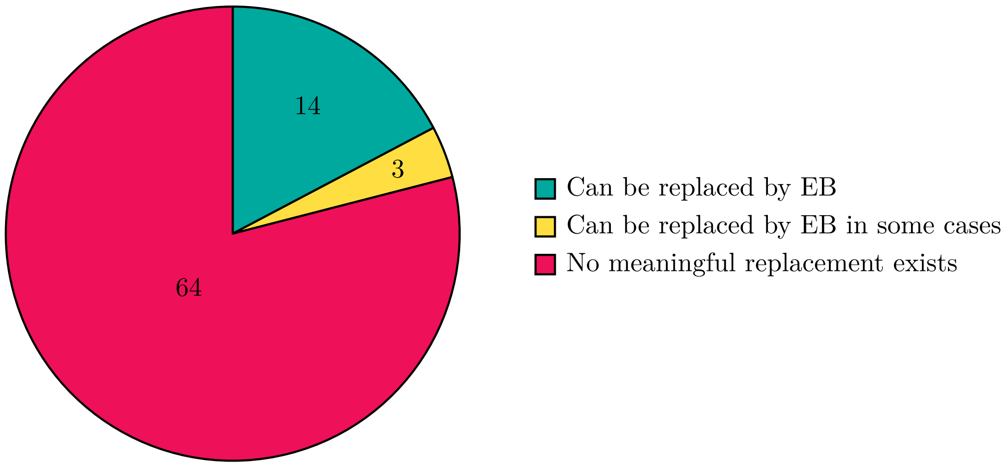

Figure 3: Existence of well-defined replacement behaviour for explicit core language UB

the underlying hardware; on weakly-ordered platforms, such as ARM, it will be larger than on strongly-ordered platforms such as x86. Note that while such replacement behaviour is well-defined, it still fails to prevent many real bugs that result from incorrect application of concurrency since user-defined types with multiple members can still be easily observed with inconsistent ("torn") states if no proper synchronisation is performed.

The replacement behaviour for UB in the Sequencing category ({intro.execution.unsequenced.modification}) is much more straightforward: we can define that the unsequenced operations happen in some unspecified order. This behaviour can still have performance overhead in the form of losing optimisation opportunities, but such overhead will likely be manageable.

The replacement behaviour for UB in the Assumption category ({dcl.attr.assume.false}) is trivial: just ignore the assumption, instead of optimising based on it. The performance overhead is limited to losing any optimisation opportunities from placing the assumption there. Of course, this mitigation makes the assumption itself completely useless. We will discuss this case in more detail in Section 4.4.

Finally, we can define partial replacement behaviour for one cases of UB in the Control Flow category ({stmt.return.flow.off}) and an analogous case of UB in the Coroutines category ({stmt.return.coroutine.flow.off}): when the function or coroutine would return a value of built-in type, we can define that flowing off the end returns an erroneous value. This case is effectively handled in the same way as {basic.indet.value}; again, no plausible replacement behaviour exists for user-defined return types in the general case.

Overall, as shown in Figure 3, we can define meaningful replacement behaviour for only 17 cases of UB (1.0% of all cases). Out of these 17 cases, for 3 cases this is only possible when the operation in question produces a value of built-in type. Unconditional replacement behaviour exists for only 14 cases of UB (17.3% of all cases). In all of these cases, the replacement behaviour consists of erroneous behaviour; in most cases, removing the UB in this manner introduces significant runtime cost.

3 Strategy

Having performed an in-depth analysis of all explicit core language UB in the C++ working paper in Section 2, we can use the results of this analysis to develop a holistic strategy for systematically detecting, mitigating, and ultimately removing UB across the entire C++ programming language specification. Our goal is for this strategy to guide the development of the core language UB white paper [P3656R1] as well as future versions of Standard C++.

The outline of this strategy is illustrated in Figure 4, an updated version of the diagram on slide 53 in [P3754R0] (a.k.a. the "magic slide") that we presented to EWG in Sofia and that EWG approved as a basis for the core language UB white paper.

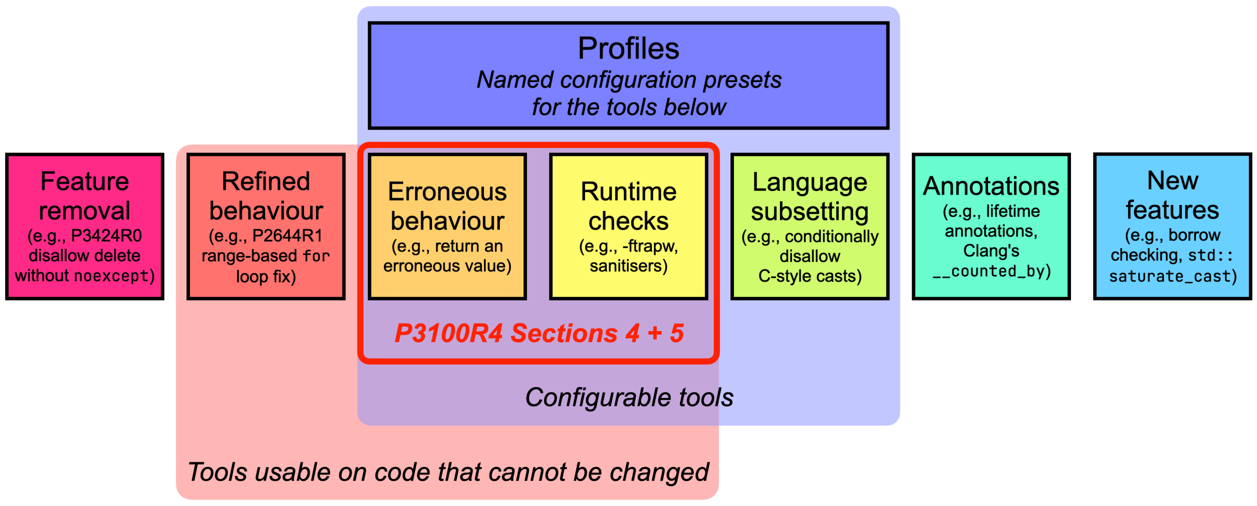
Profiles
Named configuration presets for the tools below
Feature removal (e.g., P3424R0 disallow delete without noexcept)
Refined behaviour (e.g., P2644R1 range-based for loop fix)
Erroneous behaviour (e.g., return an erroneous value)
Runtime checks (e.g., -ftrapw, sanitisers)
Language subsetting (e.g., conditionally disallow C-style casts)
Annotations (e.g., lifetime annotations, Clang's __counted_by)
New features (e.g., borrow checking, std::saturate_cast)
P3100R4 Sections 4 + 5
Configurable tools
Tools usable on code that cannot be changed

Figure 4: Overview of the proposed holistic strategy for removing UB from the C++ language: seven orthogonal tools plus Profiles as a higher-level feature specified on top of these tools. The red rectangle in the centre illustrates the scope of the proposal in Sections 4 and 5 of this paper.

3.1 Overview

The proposed strategy is composed of seven basic tools that are orthogonal to each other: feature removal, refined behaviour, erroneous behaviour, insertion of runtime checks, language subsetting, the introduction of annotations, and the introduction of entirely new language features. In section 3.3, we describe each tool and discuss which cases of UB identified in this paper it can be applied to.

Three of these tools (refined behaviour, erroneous behaviour, and runtime checks) have the interesting property that they are applicable even in cases where the source code cannot be modified for whatever reason, and thus can be used to remove UB from existing legacy C++ programs.

Further, in order to be usable effectively in practice, three of these tools (erroneous behaviour, runtime checks, and language subsetting) need to be configurable by the user, either via compiler options or directly in source. This design space is discussed in more detail in Section 3.4. *Profiles*, another proposed feature currently in development, could be used to group together particularly useful configuration presets across different tools in order to provide desirable sets of guarantees.

3.2 Scope

Our proposed strategy for removal of explicit core language UB focuses on tools that can be portably specified within the C++ abstract machine. We therefore do not consider, in this paper, efforts that operate largely outside of the C++ abstract machine and the specification tools afforded by the C++ Standard. One noteworthy effort that falls into the latter category is [P3627R0], which proposes a profile for preventing remote code execution (RCE) by employing implementation-defined techniques such as stack isolation and address space layout randomisation (ASLR).

Further, as discussed in [P3700R0], making C++ "safe"[6] consists of more than mitigating explicit core language UB. We already touched upon implicit UB and language UB in Section 2.1.3. Beyond those, there are many other classes of bugs unrelated to UB that can compromise the functional safety, security, and correctness of a C++ program. Such bugs include resource leaks, termination errors, and logic errors.

While there are tools available to address these classes of bugs (for example, logic errors can often be avoided by using strongly typed utilities such as [P3045R6]), and there is interesting ongoing work in those areas, they are out of scope for the strategy proposed here. At least for now, we explicitly target the core language UB white paper [P3656R1] and therefore limit the scope of the strategy described here to explicit core language UB.

3.3 Tools

3.3.1 Feature removal

The first and most blunt tool in our toolbox is to make a C++ operation that would otherwise lead to runtime UB unconditionally ill-formed, i.e., to remove it from the language — either immediately or via a deprecate-remove cycle spanning multiple releases of the C++ Standard.

However, we generally do not consider it acceptable to break an existing, *correct* program by using this tool (or any of the other proposed tools). If we want to strictly follow this principle, usage of the removal tool would have to be restricted to cases where we can determine at compile time that the given operation will definitely have UB when executed.[7] As discussed above in Section 2.3.1, there is not a single case of UB in our list where such a determination can generally be made.

A slightly less conservative approach is to unconditionally remove a feature if we can determine at compile time that it always *either* causes UB *or* does nothing useful. This removal can happen in a single step; alternatively, a language construct can first be deprecated, and then removed in a later Standard. Historically, the committee has been very wary of acting on such deprecations if there is a belief that the code broken by a removal will be frequent, especially if the downside of the bad construct is not generally catastrophic. In general, even breaking incorrect code such that it would fail to compile can make the cost of migrating a large codebase to a new C++ standard unbearably high if the problematic code is ubiquitous and its negative impact often benign.

A current example of using this tool is the proposal [P3424R0]. When a deallocation function (i.e., a user-defined `delete` operator) exits via an exception, the behaviour is currently undefined. Therefore, a throwing or potentially-throwing exception specification on such a deallocation function always either causes UB (if an exception ends up being thrown from it) or does nothing useful (if no such exception is ever being thrown). The paper proposes to deprecate deallocation functions with an explicit non-throwing *noexcept-specifier* and making deallocation functions with a potentially-throwing *noexcept-specifier* ill-formed.

We are currently not aware of any other areas in the C++ Standard where this tool could be successfully applied, but it is worth keeping it in our toolbox in case such areas will be discovered in the future, and committing to the slow but effective process of deprecating prior to removal when the end result is meaningfully improved.

[6] In this paper, we avoid unqualified uses of the terms "safe" and "safety" because of their ambiguity. As discussed in [P3376R0], [P3500R1], and [P3578R0], it is critically important to distinguish between conflicting usages of those terms, such as functional safety, language safety, memory safety, etc.

[7] One could argue that even then, the program is not necessarily incorrect unless that operation actually ends up being executed at run time. This is not something a C++ compiler can reason about unless the operation in question is either lexically inside `main` or is provably being called from `main`.

3.3.2 Refined behaviour

Some cases of UB can be addressed by unconditionally changing their runtime semantics to some well-defined behaviour. We have extensively used this option in the past to gradually remove UB from the language; examples of successfully applying this tool in the C++ Standard are the introduction of implicit lifetime types in C++20 [P0593R6] which gave defined behaviour to certain situations when a pointer to raw allocated memory is being cast to a pointer of object type, as well as the range-based for loop fix in C++23 [P2644R1], which extended the lifetime of certain temporary objects across the duration of executing the loop, thus avoiding UB due to dangling references.

We expect that there are relatively few remaining situations in the C++ working paper where this tool can be used effectively. To avoid creating language dialects, refined behaviour needs to be unconditional. For example, GCC has an option `-fwrapv` which turns signed integer overflow into wraparound. We cannot make that the new behaviour of signed integer addition unconditionally for two reasons. First, the associated runtime overhead would be unacceptable for many users; and second, in many cases the result will still be incorrect but this new behaviour would mask the bug, making it more difficult for users and tools to diagnose it. We also cannot have two different language dialects where the same expression means two different things (overflow or wraparound). In such cases, another tool is more appropriate — erroneous behaviour (see next section).

3.3.3 Erroneous behaviour

Some cases of UB can be addressed by replacing it with well-defined replacement behaviour, but specifying that replacement behaviour as *erroneous*, that is, well-defined but still considered incorrect. The purpose of introducing erroneous behaviour is to not remove the bug, and to leave tools with the possibility of diagnosing it, but at the same time to place a limit on the program behaviour in the face of the bug and in particular to prevent the bug from creating a security vulnerability.

Unlike refined behaviour, it is acceptable to specify erroneous behaviour in a way that is relatively vague (e.g., return *some* erroneous value rather than return a specific value), as it represents an incorrect program that cannot be relied on. For the same reason, it makes sense to make erroneous behaviour configurable (see Section 3.4). Indeed, the definition of erroneous behaviour adopted for C++26 via [P2795R5] permits different possible behaviours.

Together with that definition, we also adopted one instance of erroneous behaviour for C++26: producing an erroneous value (instead of exhibiting undefined behaviour) when reading a default-initialised automatic variable of arithmetic type [P2795R5]. Beyond that one instance, [P2795R5] contains a section with a tentative list of other cases of UB that could be replaced with erroneous behaviour, and [P2973R0] proposes to do so for erroneous behaviour for missing return from assignment.

In this paper, we go further and propose to introduce erroneous behaviour for *all* cases for which we identified in Section 2.4 that some kind of plausible well-defined replacement behaviour exists — 14 cases unconditionally plus 3 cases for built-in types only. In Sections 4 and 5 of this paper, we provide a concrete specification for how to perform the necessary replacements in the C++ working paper; the full list of proposed erroneous behaviours is provided in Appendix A.

3.3.4 Runtime checks

As we saw in Section 2.3, the large majority of UB (77 cases out of 81) can in principle be diagnosed by inserting and performing a suitable runtime check. The check verifies the conditions necessary for the operation in question to have well-defined behaviour at run time. It also acts as a guard against

UB: if the check fails, the program can be terminated, thus preventing the undefined behaviour from occurring.

While such runtime checks invariably add runtime overhead, they are very effective at both diagnosing bugs and removing security vulnerabilities. In Sections 4 and 5 of this paper, we propose a generic framework for systematically adding runtime checks to C++ core language constructs via implicit contract assertions.

Implicitly generated runtime checks are widely deployed in the field today. Checks that can be generated locally by the compiler are often provided via compiler flags, for example the `-ftrapv` flag in GCC that checks for signed integer overflow and terminates the program on failure. Checks that require additional instrumentation to perform are provided by various flavours of sanitisers such as ASan, UBSan, etc.

In order to be widely deployable, runtime checks — whether they are user-authored assertions, compiler-generated checks guarding against UB, or any other form of correctness check that is redundant in a correct program and has non-negligible overhead — need to be configurable; this is discussed in more detail in Section 3.4.

3.3.5 Language subsetting

While practically no C++ operation can, in the general case, be proven at compile time to exhibit UB at run time, there are a number of operations that are particularly prone to exhibiting UB at run time when not used correctly; such operations are colloquially known as “unsafe”. Examples of such operations are C-style casts (which can silently fall back to `reinterpret_cast`) and pointer arithmetics. Compilers and linters already provide options to statically flag the usage of such features as a potential source of bugs. We could go one step further and make such constructs ill-formed, particularly if there are “safer” alternative features that provide equivalent functionality.

However, in such cases, we cannot make these constructs *unconditionally* ill-formed, as this would break a significant number of correct C++ programs deployed in the field. Therefore, we need to use a tool different from simple removal (Section 3.3.1). The required tool is called *language subsetting*: specifying named *subsets* of the C++ language that do not contain the “unsafe” features. This tool can be very effective for avoiding UB in codebases that can be modernised or are being newly written. However, because much of existing C++ code cannot be changed easily, subsetting needs to be opt-in. Thus, it is the third tool in our toolbox that needs to be configurable by the user (see Section 3.4).

While we do not propose a concrete specification for language subsetting in this paper, such a specification is being developed in [P3716R0]. As described in that paper, a number of principles need to be considered for designing this tool correctly: subsetting the language should never be allowed to alter the semantics of well-formed code, subsets must always combine orthogonally, and so forth.

3.3.6 Annotations

Some cases of UB can be mitigated by language-level annotations that provide additional information that can be propagated across the interface boundaries of a C++ program. Such information can turn cases of UB that are *not* locally diagnosable (which is the majority of UB today, see Section 2.3) into cases that *are* locally diagnosable.

An example of a standard proposal in this area are the lifetime annotations proposed in [P2771R1]. An example of such annotations deployed in the field as non-standard vendor extensions is the `__counted_by` attribute introduced in Clang 18. This attribute allows the user to propagate information about array bounds to enable automatic out-of-bounds checks. Such attributes are a great example for how different tools that form the proposed strategy, while being usable independently from each other, can also work together to help each other cover even more cases of UB.

3.3.7 New features

Finally, some cases of UB can be addressed by providing entirely new language features that provide functionality equivalent to that of existing "unsafe" features but without the possibility of UB. Examples of such proposals are `std::saturate_cast` [P0543R3] (approved for C++26), enabling conversion from one integer type to another without the danger of UB due to values not representable in the target type, and borrow checking [P3390R0], a memory-safe alternative to pointers and references based on the Rust borrow checker.

Note that combining the new features tool with the language subsetting tool is equivalent to the approach encouraged by Bjarne Stroustrup, "superset then subset" (see [P3650R0] and references therein).

3.4 Configuration

Three of the seven tools discussed above need to be *configurable* by the user in order to be usable in practice: runtime checks, erroneous behaviour, and language subsetting.

Runtime checks with non-negligible overhead, in order to be widely deployable, necessarily need to be configurable in several ways; this is discussed in much detail in [P2899R1] and references therein. As we saw in Section 2.3, the majority of UB in C++ today is not locally diagnosable and requires expensive sanitiser-like instrumentation to perform the checks. Even for those 19 cases of UB that are always locally diagnosable and do not require additional instrumentation to insert runtime checks, in most cases the checks themselves will have a significant — and in some cases, unacceptable — runtime overhead. Since correctness checks serve no purpose in a program known to be correct — or for use cases where correctness is not the overriding concern — there needs to be an option to turn the checks off to avoid the overhead. This option is offered by every assertion facility that we know of.

Further, the behaviour of a failed check needs to be configurable as well. In some scenarios, the most appropriate behaviour following a failed check is immediate termination; in others, termination is unacceptable even in the face of bugs. In some scenarios, it is desirable to have detailed diagnostics describing the failure; in others, the overhead of generating such diagnostics is undesirable.

Similar reasons apply to erroneous behaviour: in practically all cases, its introduction comes with non-negligible — and in some cases, even very large — performance overhead. Therefore, to avoid unacceptable performance regressions in existing, correct C++ code, we *must* offer an escape hatch that reverts to today's "unsafe" semantics. One such escape hatch is the `[[indeterminate]]` attribute for uninitialised values. Such a syntactic escape hatch is not applicable for all cases; we propose a generic escape hatch for erroneous behaviour that does not require syntax in Section 4.4.

Finally, as already discussed in Section 3.3.5, language subsetting needs to be opt-in in order to avoid breaking existing, correct C++ programs.

For all these features, we need to clearly specify the available configuration options and the mechanisms available for the user to select these options at different levels of granularity. For erroneous behaviour and runtime checks, we accomplish this via leveraging contract evaluation semantics (see Section 4); in-source configuration of these evaluation semantics at arbitrary granularity can be achieved with Labels (see Section 6). For subsetting, the design of suitable configuration mechanisms are not yet well understood; we expect future papers to make progress in this area.

Another proposed feature in this space is *Profiles*. We can distinguish between individual profiles — collections of rules that aim to provide a guarantee that a C++ program exhibit certain qualities (e.g., [P3081R2], [P3038R0], [P3402R3], and [P3446R0]) — and the *Profiles framework* [P3589R2], a set of mechanisms to enable and disable a named profile at various levels of granularity.

There is no consensus yet on how Profiles relate to and compose with other proposals such as the holistic strategy for removing UB from C++ proposed here. That said, most Profiles proposed so far consist of some combination of specifying subsets of the language (Section 3.3.5), defining replacement behaviour for UB (Section 3.3.3), and/or introducing runtime checks guarding against UB (Section 3.3.4). It therefore seems logical to define Profiles as a higher-level feature building on top of these three basic tools (see Figure 4). Given that these three features are configurable, a concrete profile could be defined as being a named configuration preset for these features.

For example, we may define a language subset that excludes pointer arithmetics, and a set of implicit runtime checks for array bounds checking; further, the user may define explicit contract assertions on their own functions and declare them as bounds checks. We can then define a "bounds" profile whose purpose it is to allow the user to opt into all three of these features at once. Such profiles can be tailored to particular problem areas of the language, such as type, bounds, or arithmetic safety profiles, or to particular regulatory requirements, such as a MISRA profile. If we pursue such a "multi-level" strategy, we must make it clear which feature is responsible for providing the user-facing configuration mechanism for which tool (see also Section 6).

4 Proposed design

In this section, we propose a framework that systematically introduces runtime checks and well-defined replacement behaviour — two of the tools that form the strategy presented in Section 3 — to the C++ Standard. Runtime checks guarding against core language UB are realised as *implicit contract assertions*, leveraging the foundation laid by Contracts as adopted for C++26; in addition, replacement behaviour is added instead of UB wherever meaningfully possible. Both features are controlled and configured via the evaluation semantics introduced by [P2900R14].

4.1 Defining implicit contract assertions

The initial subset of Contracts functionality added to C++26 via [P2900R14] contains three kinds of *contract assertions*: `pre`, `post`, and `contract_assert`. Since these contract assertions are specified by the user with explicit syntax, in this paper we call them *explicit* contract assertions. For example, the author of a vector-like class can add a precondition assertion to its subscript operator to guard against out-of-bounds access:

```
T& operator[] (size_t index)
  pre (index < size());
```

The precondition assertion `pre (index < size())` can be evaluated with a checked assertion (*observe*, *enforce*, or *quick-enforce*), which allows the user to opt into defined behaviour — program termination and/or a call to a contract-violation handler — when their vector is accessed out of bounds. Further, the contract-violation handler can be replaced by the user, allowing them to query information about the error and implement their own mitigation strategy. Alternatively, the user can opt out of the runtime check by choosing an unchecked evaluation semantic (*ignore*) if their use case requires it.

To implement runtime checks that guard against core language UB, we propose to introduce *implicit contract assertions*, which are added implicitly by the implementation, rather than explicitly by the user. In all other aspects, they work exactly the same as explicit contract assertions.

As an example, let us consider indexing into a plain array rather than a user-defined, vector-like class. Let us further assume for the purpose of this example that the size `N` of this array is statically known:

```
int main() {
  int a[10] = { 1, 1, 2, 3, 5 };
  std::size_t i;
  std::cin >> i;
  return a[i];
}
```

In C++ today, the behaviour of this program is undefined if the value of `i` is not smaller than 10 ({expr.add.out.of.bounds}). However, instead of saying that out-of-bounds access into a plain array is UB, we can say that access into a plain array has an *implicit precondition assertion* that the index is not out of bounds. Then, the program behaves as-if the compiler had wrapped every raw array subscript operation for which it statically knows the array bound `N` into an inline function with a precondition assertion:

```
template <typename T, std::size_t N>
T& __index_into_array(T (&a)[N], std::size_t i)
pre (i < N) {
  return *(&a + i);
}
```

Other than being an implicit precondition assertion automatically generated by the compiler, `pre (i < N)` behaves the same as an explicit precondition assertion. That is, the user has the same choice of four evaluation semantics (*ignore*, *observe*, *enforce*, or *quick-enforce*) to specify the desired behaviour depending on the tradeoffs that are most suitable for their application. When an out-of-bounds access is detected and the semantic is *observe* or *enforce*, the same contract-violation handler is called that is used for explicit contract assertions.

4.2 Applying implicit contract assertions

Having specified precisely what an implicit contract assertion is and how it behaves, we can now apply that specification to *every* case of UB that is — at least in principle — checkable at run time. As we saw in Section 2, this is true for 77 cases, that is, 95% of all identified cases of explicit core language UB in C++.

This feat can be accomplished by applying the following transformation across the entire C++ Standard: change every occurrence of "if A is not `true`, operation X has undefined behaviour" to "operation X has an implicit precondition that A is `true`; continuing execution past a violation of this precondition is undefined behaviour".

Note that the choice of evaluation semantic is implementation-defined; therefore, there are no restrictions on the evaluation semantics of any of these 77 newly introduced implicit contract assertions beyond the requirement that an implementation document which semantics they support for which implicit contract assertions and which selection mechanism they offer. These options, and the choice of the default, depend on the particular case.

Note that no implementation is actually *required* to implement these checks: a valid implementation choice is to make all 77 cases always have the *ignore* semantic. It follows that all existing implementations of C++ are already conforming with this wording transformation.

Many of the other possible choices map directly to existing compiler and sanitiser options. For example, for signed integer overflow, the GCC flag `-ftrapw` is a conforming implementation of the *quick-enforce* semantic; sanitisers like ASan and UBSan are conforming implementations of the *enforce* semantic for those cases of UB that they identify. These tools can continue to work in the way they do; however, bringing them into the scope of the C++ Standard as proposed here has many benefits.

One benefit is that implementations of such runtime checks will be able to leverage a shared paradigm and shared terminology for reasoning about incorrect programs. In addition, once we have Labels (see Section 6), for each case of UB guarded by an implicit contract assertion, implementations and users can refer to each case and each category using portable standard names. Another benefit is that they will be able to integrate with the same unified standard contract-violation handling facility, significantly increasing the ability to deploy software to production systems that is hardened against entire categories of potential bugs.

This is significant because today, the integration between such tools and user code tends to be poor. For example, all Clang sanitisers have a callback, `__sanitizer_set_death_callback`, but this callback takes no arguments. It can be used to inform us that the process is about to terminate, but it does not provide an API to programmatically query what happened or where. ASan has a slightly more sophisticated callback, `__asan_set_error_report_callback`, which takes a single argument of type `const char*`. This argument provides a string that contains the generated error report. With our proposal, all these tools can instead hook into the standard contract-violation-handling API. This API provides not only a user callback in the form of a program-wide replaceable contract-violation handler, but also programmatically accessible information about the defect via the `contract_violation` object passed into the contract-violation handler. This more comprehensive API can serve as a uniform, standard callback mechanism for sanitisers and other tools.

Further, coding guidelines can place restrictions on which evaluation semantics are permitted for which kinds of implicit contract assertions; our proposal provides the necessary standard terminology for this. For example, in a safety-critical context, a set of coding guidelines may prescribe that unchecked semantics may not be used for certain kinds of implicit contract assertions. Further, we could add a syntactic way to render configurations not conforming with this requirement ill-formed (see also Section 6.3). Thus, the usage of toolchains and compiler options that could lead to the program exhibiting a particular kind of UB could be prevented by construction. Of course, this option requires alternatives to exist that offer checked semantics for the associated implicit contract assertions with acceptable performance tradeoffs.

Finally, applying implicit contract assertions throughout the language in the proposed fashion addresses another much-discussed issue: *explicit* contract assertions in C++26, as specified in [P2900R14], can themselves have UB when checked because explicit contract-assertion predicates are boolean expressions and thus follow the usual rules for evaluating expressions in C++. This property has been repeatedly raised as a concern (see [P2680R1], [P3173R0], [P3285R0], and [P3362R0]).

The approach suggested in those papers is to constrain explicit contract-assertion predicates to expressions that can be statically proven to have no UB. However, this approach does not seem to be specifiable, implementable, or usable in practice (see [P3376R0], [P3386R0], and [P3499R1]) and has repeatedly been rejected by SG21, SG23, and EWG. What *does* work is to specify a framework for mitigating UB across the entire language, as proposed here. Once we have this framework, it will then automatically also apply to the evaluation of explicit contract assertions.

4.3 Defining replacement behaviour

The next part of our proposal is to introduce well-defined replacement behaviour for all 17 cases of core language UB for which such replacement behaviour exists (see Section 2.4). We accomplish this by modifying the specification of each affected operation as follows. If a condition that would have previously made the behaviour of the operation undefined (i.e., an implicit contract violation) occurs, and control flow continues past that violation (because the associated check was evaluated with the *ignore* or *observe* semantic), then the behaviour of the operation is defined to be that replacement behaviour instead of UB. Consider the following example:

```
int g(int i) {
  return i + 100;
}
```

This function returns the result of adding two signed integers, which may or may not exhibit UB depending on whether the addition will overflow — which is unknown at compile time as one of the integers in question is a runtime parameter of the function.

This program now behaves as-if the compiler performed every signed integer addition with a built-in `operator+` that is guarded by an implicit precondition assertion against overflow, *and* whose implicitly defined function body always has well-defined behaviour as follows:

```
int operator+(int a, int b)
pre ((b >= 0 && a <= INT_MAX - b) || (b < 0 && a >= INT_MIN - b)) {
  // return the result of the addition or an erroneous value
}
```

The fact that this implicit function body always has well-defined behaviour is the major difference between cases where meaningful replacement behaviour exists (such as this one), and cases where it does not (such as array subscripting).

The wording transformation required to implement the above replacement across the entire C++ Standard is to change every occurrence of "if A is not `true`, operation X has undefined behaviour" to "operation X has an implicit precondition that A is `true`; if this precondition is violated, the behaviour is *<replacement behaviour>*".

In all 17 cases identified in this paper, this replacement behaviour will be some form of erroneous behaviour. For data races, the erroneous behaviour consists of performing all primitive memory accesses atomically in some unspecified order; for unsequenced operations, it consists of performing the operations in question in sone unspecified order; for assumptions, it consists of ignoring the assumption; for all remaining cases, it consists of returning an erroneous value (which is only possible in cases where the operation in question returns a value of built-in type).

4.4 Providing an escape hatch

As discussed in Section 2.4, if we apply the transformation describe in the previous section and do nothing further, we introduce significant — and in many cases, unacceptable — performance regressions to existing code. Therefore, we must offer an escape hatch to users that reverts to today's semantics: a violation of the implicit precondition leads to runtime UB.

For erroneous behaviour that arises from reading an indeterminate value, [P2795R5] introduced a semantic escape hatch specific for this case: the `[[indeterminate]]` attribute. However, in many cases, such a syntactic escape hatch is simply nonviable. Consider, for example, arbitrary arithmetic expressions where some integer operations may overflow; where would we place a syntactic escape hatch for a certain arithmetic operation within that expression? Instead, we need a *generic* escape hatch that works for all cases and does not require syntax.

Further, this escape hatch needs to be flexible enough that implementations can choose whether or not it should be engaged by default. Engaging the escape hatch by default seems counterintuitive because doing so would fail to provide a "safe default"; however, in some cases, it will be necessary as defaulting to the well-defined replacement behaviour and silently incurring the associated runtime overhead would be too user-hostile.

As it turns out, such a generic, nonsyntactic escape hatch that reverts to today's semantics — a violation of the implicit precondition leads to UB — is nothing other than a new, fifth evaluation semantic in addition to the four existing ones (*ignore*, *observe*, *enforce*, *quick-enforce*) that can be applied to the evaluation of the affected implicit contract assertions. This evaluation semantic is called the *assume* semantic.

Just like the *ignore* semantic, the *assume* semantic is a *nonchecking* semantic; i.e., its predicate is not evaluated. Further, just like with the *ignore* semantic, if the predicate evaluates to `true` at the point where the contract assertion is placed, the *assume* semantic has no effect; i.e., the program behaves exactly as if the contract assertion were not there. However, unlike the *ignore* semantic, if the predicate does *not* evaluate to `true`, the behaviour is undefined. This semantic allows compilers to optimise on the assumption that the predicate is `true`, just like they do today for those cases of core language UB.

With this definition, we can map all five evaluation semantics for implicit contract assertions that guard against core language UB to concrete behaviours. For example, for signed integer overflow, this mapping is as follows:

— The GCC compiler option `-ftrapv`, which aborts the program on signed integer overflow, is a conforming implementation of the *quick_enforce* semantic.

— A sanitiser that detects signed integer overflow and prints a diagnostic is a conforming implementation of the *enforce* or *observe* semantic (depending on whether the process is terminated or execution continues after printing the diagnostic).

— The GCC compiler option `-fwrapv`, which implements wraparound for signed integer addition using twos-complement representation, is a conforming implementation of the *ignore* semantic, silently executing well-defined replacement behaviour.

— The default behaviour in C++ today, which is to assume that signed integer addition never overflows and to optimise based on this assumption when the appropriate optimisation flags are selected by the user, is a conforming implementation of the *assume* semantic.

Just like with all other evaluation semantics, the mechanism by which the *assume* semantic is selected is implementation-defined and will, in practice, be accomplished by vendor-provided compiler flags. In addition, Labels (see Section 6.2) will provide the ability to choose and constrain the evaluation semantic in code with arbitrary granularity.

Importantly, in light of the sustained opposition in EWG to allowing the *assume* semantic for *explicit* contract assertions (`pre`, `post`, and `contract_assert`), we propose that the *assume* semantic is allowed for only *implicit* contract assertions. Explicit contract assertions may *not* be evaluated with the *assume* semantic (see also Section 6.3).

This restriction is important because, for explicit contract assertions, the *assume* semantic has the potential to silently add UB to an otherwise correct program if used incorrectly. This risk does not exist for implicit contract assertions since they are generated by the compiler; in all of those cases, the *assume* semantic is merely a backwards-compatibility tool to achieve the same semantics that those operations already have in C++ today.

Note that this set of five evaluation semantics has the interesting property that it provides a single mechanism to configure two different tools in our toolbox simultaneously: the replacement behaviour as well as the associated implicit contract assertion.

4.5 Interaction with `noexcept`

Consider:

```
bool f() {
  int x;
```

```
    return noexcept(x + 1);
  }
```

In C++ today, calling `f()` has defined behaviour (the indeterminate value is never accessed; the operand of `noexcept` is an unevaluated operand) and returns `true` (adding two integers can never throw an exception unless the behaviour is undefined). If we want to avoid breaking changes to the existing language, the result of the `noexcept` operator must remain the same with this proposal.

However, since `x` has an erroneous value, evaluating `x + 1` may call the contract violation handler, which in turn may throw an exception. With our proposal, it is therefore no longer true that evaluating the core language expression `x + 1` can never throw an exception unless UB occurs due to signed integer overflow.

A detailed discussion of this problem can be found in [P3541R1]. Fundamentally, in order to address this problem, we need to choose between the following three options: either accept the breaking change to the `noexcept` operator, or do not allow throwing violation handlers for implicit contract assertions,[8] or redefine the meaning of the `noexcept` operator to be "can never throw an exception *unless there is a contract violation*".

SG21 discussed this problem at great length. The first option introduces unacceptable breaking changes to existing C++ programs. The second option precludes unwinding the stack in response to a contract violation, which is not expected to be a very common strategy, but has important use cases (see [P3318R0]). SG21 therefore concluded that the only acceptable solution is the third one.

For the proposal in this paper, we follow the SG21 consensus. It is therefore possible for an implicit contract assertion to call a throwing contract-violation handler when violated, and for the evaluation of the expression to exit via that exception, even if the `noexcept` operator returns `true` for that expression.

4.6 Extending the library API

To give the user a way to programmatically distinguish explicit and implicit contract assertions in the contract-violation handler, we propose to add a new enum value, `implicit`, to the enum `assertion_kind`. We simply append the new enumerator to the existing ones, which gives it the numerical value `4`, without attaching any particular meaning to that numerical value.

Alternatively, we could define its numerical value to be `0` since that value is not yet taken; however, we prefer to avoid using `0` and thus to retain the ability to detect the case in which the enum has not been explicitly initialised with a valid value.[9]

No other changes to the library API for contract-violation handling are necessary. In particular, unlike earlier revisions of this paper and unlike [P3081R1], which adopted its library API from those earlier revisions, we no longer propose to add new enumerators to the enumeration `detection_mode` to encode the category of error (Initialization, Bounds, and so on); instead, this encoding can be accomplished more effectively and flexibly via Labels (see Section 6.1).

Further, we propose no changes to the specification of `comment()` and `location()`. C++26 non-normatively recommends that these functions return a textual representation of the expression that triggered the contract violation and the source location of the contract violation, respectively. While returning such a representation is, in principle, possible for violations of implicit contract assertions, generating a textual representation for every expression in the program that could lead to UB is likely to cause an unacceptable amount of code bloat. However, generating some other string

[8] As a variation of this option, [P3577R0] proposes that the *default* contract-violation handler should be normatively prohibited from exiting via an exception, however a *user-defined* contract-violation handler is still free to do so. However, this paper did not get consensus in SG21 or EWG.

[9] See also [P3227R0], which was adopted into [P2900R14] and made the same argument for adding new enumerators to the enumeration `evaluation_semantic`.

that may help us identify the problem, such as the diagnostic message already printed by existing sanitisers, is equally conforming, as is simply returning an empty string and a default-constructed source location if no information is available or if the information cannot be made programmatically accessible in the contract-violation handler (for example, because it is located in a separate debug information file).

Finally, we do not propose a separate contract-violation handler for implicit contract assertions. Having a single, program-wide handler for all contract violations is a central aspect of the [P2900R14] design. By standardising on a central reporting mechanism, we clearly separate the responsibility for reporting from the responsibility of knowing all the different mechanisms within a program by which a bug might be detected. For example, the user might want to hard-code a particular form of termination or to use a particular logger. Forcing the user to repeat these things in multiple places is poor design. A user who wishes to use a different handler for implicit contract assertions can always branch on the `assertion_kind` in the global contract-violation handler and dispatch to a custom handler from there.

5 Proposed wording

The proposed wording is relative to revision [N5008] of the C++ working paper.

Modify [basic.contract.general] as follows:

> Contract assertions ~~allow the programmer to~~ specify properties of the state of the program that are expected to hold at certain points during execution. Explicit c~~C~~ontract assertions are introduced by *precondition-specifiers*, *postcondition-specifiers* ([dcl.contract.func]), and *assertion-statements* ([stmt.contract.assert]). *Implicit* contract assertions are applied to operations by the implementation.
>
> Each contract assertion has a predicate, which is an expression of type `bool`. [*Note:* ~~The value of the predicate is used to identify program states that are expected.~~ If it is determined during program execution that the predicate does not evaluate to `true`, a contract violation occurs. A contract violation is always the consequence of incorrect program code. — *end note*]

Modify [basic.contract.eval] as follows:

> An evaluation of a contract assertion uses one of the following five~~four~~ evaluation semantics: *assume,* *ignore*, *observe*, *enforce*, or *quick-enforce*. Observe, enforce, and quick-enforce are checking semantics; enforce and quick-enforce are terminating semantics.
>
> It is implementation-defined which evaluation semantic is used for any given evaluation of a contract assertion. Explicit contract assertions are never evaluated with the assume semantic.
>
> [...]
>
> The evaluation of a contract assertion using the ignore or assume semantic has no effect. If the semantic is assume and the predicate would not evaluate to `true`, evaluation of the contract assertion has runtime undefined behaviour.

Add a new section, [basic.contract.implicit] after [basic.contract.eval]:

> A built-in operation O may have an *implicit precondition assertion* C applied to it. If so, the evaluation of C is sequenced before the evaluation of O and after the evaluation of all operands of O.
>
> A built-in operation O may have an *implicit postcondition assertion* C applied to it. If so, the evaluation of C is sequenced after the evaluation of O.

Modify [except.spec] as follows:

> An expression *E* is *potentially-throwing* if [...] or any of the immediate subexpressions of *E* is potentially-throwing. [*Note*: The evaluation of an expression that is not potentially-throwing may nevertheless exit via an exception if, as part of that evaluation, the violation of an implicit contract assertion ([basic.contract.general]) causes a call to the contract-violation handler ([basic.contract.handler]) and that handler exits via an exception. — *end note*]

Modify [contracts.syn] as follows:

```
enum class assertion_kind :  unspecified {
  pre = 1,
  post = 2,
  assert = 3,
  implicit = 4
};
```

Modify [support.contract.enum] as follows:

Name	Meaning
`pre`	A precondition assertion
`post`	A postcondition assertion
`assert`	An *assertion-statement*
`implicit`	An implicit contract assertion

Modify all cases of UB checkable at run time *with* replacement behaviour, as listed in Appendix A, according to the following pattern.

— Example [expr.expr.eval]:

> ~~If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined.~~Evaluation of an expression has an implicit postcondition assertion that the result is mathematically defined and in the range of representable values for its type; if this precondition assertion is violated, the result is an erroneous value.

— Example [conv.rank]:

> The value computations of the operands of an operator are sequenced before the value computation of the result of the operator. ~~The behavior is undefined if~~There is an implicit contract assertion that ~~a~~no side effect on a memory location ([intro.memory]) or starting or ending the lifetime of an object in a memory location is unsequenced relative to another side effect on the same memory location, starting or ending the lifetime of an object occupying storage that overlaps with the memory location, or a value computation using the value of any object in the same memory location, and the two evaluations are not potentially concurrent ([intro.multithread]); if this precondition assertion is violated, the value computations are sequenced in an unspecified order.

Modify all cases of UB checkable at run time *without* replacement behaviour, as listed in Appendix A, according to the following pattern.

— Example [basic.stc.dynamic.allocation]:

> ~~The effect of i~~Indirecting through a pointer has an implicit precondition assertion that the pointer was not returned from a request for zero size; continuing execution past a violation of this precondition assertion is undefined.

— Example [class.cdtor]:

For an object with a non-trivial destructor, referring to any non-static member or base class of the object has an implicit precondition assertion that the destructor has not yet finished~~after the destructor finishes~~ execution; continuing execution past a violation of this precondition assertion results in undefined behavior.

Written-out wording for all 79 cases of UB checkable at run time listed in Appendix A can be provided in a future revision of this paper.

6 Future extensions

We already briefly touched upon Labels in earlier sections of this paper. Here, we explore other extensions that rely on Labels as proposed in [P3400R1] and provide important additional functionality for implicit contract assertions not proposed in this paper.

6.1 Identifying the UB category

[P3400R1] proposes the addition of *identification labels* to contract assertions. These identification labels can be used to identify groups of contract assertions by name. For explicit contract assertions, we must introduce these identification labels manually; however, for implicit contract assertions, we can define and assign such identification labels directly in the C++ Standard (see [P3400R1] Section 2.2.8). Such implicitly defined identification labels would make possible programmatically identifying, in the contract-violation handler, whether the violated implicit contract assertion is related to an out-of-bounds issue, an arithmetic issue, and so forth; for example:

```
void handle_contract_violation(const std::contracts::contract_violation& violation)
{
  if (auto* bounds_label =
      violation.getLabel<std::contracts::labels::bounds_label>()) {
      // handle violation of assertion labelled with the bounds label
  }
}
```

Notably, the [P3400R1] approach has an important advantage over using the `detection_mode` enum, as proposed in [P3081R1] and in earlier versions of this paper: a single implicit contract assertion can belong to multiple groups. We identified cases of UB, such as {expr.dynamic.cast.glvalue.lifetime}, that are simultaneously type and lifetime issues.

In addition, users (and, more importantly, libraries) can use such labels to annotate their own explicit contract assertions, enabling the same policies to guide handling of core language bounds violations and violations of higher-level functions. For example, the indexing operator of a user-defined container (such as the one shown in Section 4.1) can have an explicit precondition labelled to belong to the same Bounds category as bounds checks defined by the C++ Standard itself. The same identification labels can be defined for hardened preconditions in the C++ Standard Library.

6.2 Granular control of the evaluation semantic

Another important feature enabled by Labels is the possibility to control and constrain the evaluation semantic in code. This possibility also extends to implicit contract assertions (see [P3400R1] Section 2.2.8). Any possible label, such as "always enforce", "never enforce", etc., can be applied to any group of implicit contract assertions at any granularity: per file, per TU, per module, per namespace, per function, or per code block:

```
int f(int a, int b) {
  contract_assert implicit arithmetic |= always_enforce;
```

```
    return a + b;
  }
```

In addition to labels that specify or constrain the evaluation semantics directly, there are labels that give the user higher-level control of the evaluation semantics based on meaningful decisions, for example an “audit” label to identify expensive checks.

Labels used in this way provide granular control when needed, allow the Standard to specify useful groupings of different sources of program defects, and give developers the freedom they need to control mitigations for those defects based on exactly the criteria needed for their environments.

Such a control mechanism for runtime checks (or for other tools in our toolbox such as language subsetting) needs to be designed carefully and take into account the overall strategy (Figure 4). In particular, we need to make it clear which feature is responsible for providing the user-facing configuration mechanism for which tool, and avoid ending up in a situation where the same functionality is provided simultaneously by different features in incompatible ways.

For granular, in-source control of the evaluation semantics of implicit contract assertions, we need to agree whether this happens via directives such as the ones proposed in [P3400R1] and shown here, or by using the syntax proposed in the Profiles framework as proposed in [P3589R2]. If we want to have both, we need to specify one in terms of the other to avoid an incoherent and messy design.

If we follow the idea in Section 3.4 and consider Profiles to be a higher-level feature defined in terms of the seven basic tools (the low-level features), then a Profile that enables or disables runtime checks can be defined as essentially a declaration that expands to [P3400R1] directives as the one shown above. Alternatively, we could design Profiles as an auditing feature rather than a configuration feature: instead of actively enabling certain configuration options, the effect of a Profile would be that the program is ill-formed if the configuration options chosen via [P3400R1] directives or other mechanisms are not compatible with the guarantees that that Profile ensures.

6.3 Integrating assertions and assumptions

In Section 4.4, we introduced the *assume* semantic as a backwards-compatibility escape hatch for newly introduced erroneous behaviour; as such, it can only apply to implicit contract assertions, not to explicit ones.

Allowing the *assume* semantic on explicit contract assertions has met sustained opposition in EWG due to the possibility of inadvertently *adding* new UB to a C++ program instead of removing it. The presence of the *assume* semantic in the C++2a Contracts proposal [P0542R5] contributed to that proposal being removed from the C++20 Working Draft. In response to this opposition, no *assume* semantic was included in C++26 Contracts [P2900R14]. Assumptions were instead standardised as a separate feature in the form of the `[[assume]]` attribute [P1774R8] to enable the required functionality.

However, Labels, as proposed in [P3400R1], open up the possibility of introducing an explicit label that would allow the *assume* semantic to apply to an explicit contract assertion as well. Consider the limiter example from [P1774R8]:

```
void limiter(float* data, size_t size) {
  [[assume(size > 0)]];
  [[assume(size % 32 == 0)]];
  // implementation
}
```

With a `may_be_assumed` label, we could instead write:

```
void limiter(float* data, size_t size)
  pre<may_be_assumed> (size > 0);
  pre<may_be_assumed> (size % 32 == 0);
```

Now, the assumptions are not only visible on the *declaration* of the function, but also benefit from all other features of explicit precondition assertions, such as the ability to select evaluation semantics other than *assume*.

To avoid the possibility of introducing an assumption by accident, the *assume* semantic would be allowed on explicit contract assertions only when the `may_be_assumed` label is present; further, a "safe C++" profile could make such a label ill-formed. Thus, contract assertions without the explicit label would be no less "safe" than they are in C++26.

Such a label would be a vast improvement over today's `[[assume]]` attribute since it would allow for *checkable* assumptions (see [P2064R0] for context), achieving the integration between assertions and assumptions that we failed to achieve in the C++20 cycle. The `[[assume]]` attribute — a temporary solution that was introduced as a reaction to that failure — could then be deprecated.

Acknowledgements

Thanks to Gašper Ažman, Herb Sutter, Oliver Rosten, Andrzej Krzemieński, Roger Orr, Phil Nash, and Peter Bindels for their helpful feedback on previous revisions of this paper.

Thanks to Lori Hughes for reviewing a previous revision this paper and providing editorial feedback.

Bibliography

[Baxter2024] Sean Baxter. Why Safety Profiles Failed. https://www.circle-lang.org/draft-profiles.html, 2024-10-24.

[CWG2816] Jiang An. Core Issue 2816: Unclear phrasing "may assume ... eventually". https://www.open-std.org/jtc1/sc22/wg21/docs/cwg_active.html#2816, 2023-04-26.

[CWG3022] Timur Doumler. Core Issue 3022: Redundant specification of explicit destructor calls. https://cplusplus.github.io/CWG/issues/3022.html, 2025-04-13.

[CWG3034] Timur Doumler. Core Issue 3034: Infinite recursion should hit an implementation limit. https://cplusplus.github.io/CWG/issues/3034.html, 2025-07-26.

[Lippincott2025] Lisa Lippincott. Balancing the Books. C++Now talk, 2025-04-30.

[N5008] Thomas Köppe. Working Draft, Standard for Programming Language C++. https://wg21.link/n5008, 2025-03-15.

[P0542R5] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, and B. Stroustrup. Support for contract based programming in C++. https://wg21.link/p0542r5, 2018-06-08.

[P0543R3] Jens Maurer. Saturation arithmetic. https://wg21.link/p0543r3, 2023-07-19.

[P0593R6] Richard Smith. Implicit creation of objects for low-level object manipulation. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0593r6.html, 2020-02-14.

[P1705R1] Shafik Yaghmour. Enumerating Core Undefined Behavior. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1705r1.html, 2019-09-28.

[P1774R8] Timur Doumler. Portable assumptions. https://wg21.link/p1774r8, 2022-06-14.

[P2064R0] Herb Sutter. Assumptions. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2064r0.pdf, 2020-01-13.

[P2644R1] Nicolai Josuttis, Herb Sutter, Titus Winter, Hana Dusíková, Fabio Fracassi, Victor Zverovich, Bryce Adelstein Lelbach, and Peter Sommerlad. Final Fix of Broken Range-based for Loop, Rev 1. https://wg21.link/p2644r1, 2022-11-11.

[P2680R1] Gabriel Dos Reis. Contracts for C++: Prioritizing Safety. https://wg21.link/p2680r1, 2022-12-15.

[P2723R1] JF Bastien. Zero-initialize objects of automatic storage duration. https://wg21.link/p2723r1, 2023-01-15.

[P2754R0] Jake Fevold. Deconstructing the Avoidance of Uninitialized Reads of Auto Variables. https://wg21.link/p2754r0, 2023-01-24.

[P2771R1] Thomas Neumann. Towards memory safety in C++. https://wg21.link/p2771r1, 2023-05-17.

[P2795R5] Thomas Köppe. Erroneous behaviour for uninitialized reads. https://wg21.link/p2795r5, 2024-03-22.

[P2843R3] Alisdair Meredith. Preprocessing is never undefined. https://wg21.link/p2843r3, 2025-06-20.

[P2899R1] Joshua Berne, Timur Doumler, Rostislav Khlebnikov, and Andrzej Krzemieński. Contracts for C++ — Rationale. https://wg21.link/p2899r1, 2025-03-14.

[P2900R14] Joshua Berne, Timur Doumler, and Andrzej Krzemieński. Contracts for C++. https://wg21.link/p2900r14, 2025-02-13.

[P2973R0] Jonathan Wakely and Thomas Köppe. Erroneous behaviour for missing return from assignment. https://wg21.link/p2973r0, 2023-09-15.

[P3038R0] . . https://wg21.link/p3038r0, 202.

[P3045R6] Mateusz Pusz, Dominik Berner, Johel Ernesto Guerrero Pe∼na, Chip Hogg, Nicolas Holthaus, Roth Michaels, and Vincent Reverdy. Quantities and units library. https://wg21.link/p3045r6, 2025-06-19.

[P3075R0] Shafik Yaghmour. Adding an Undefined Behavior and IFNDR Annex. https://wg21.link/p3075r0, 2023-12-15.

[P3081R1] Herb Sutter. Core safety profiles for C++26. https://wg21.link/p3081r1, 2025-01-06.

[P3081R2] Herb Sutter. Core safety profiles for C++26. https://wg21.link/p3081r2, 2025-02-04.

[P3173R0] Gabriel Dos Reis. P2900R6 May Be Minimal, but It Is Not Viable. https://wg21.link/p3173r0, 2024-02-15.

[P3227R0] Gašper Ažman and Timur Doumler. Fixing the library API for contract violation handling. https://wg21.link/p3227r0, 2024-10-15.

[P3285R0] Gabriel Dos Reis. Contracts: Protecting The Protector. https://wg21.link/p3285r0, 2024-05-15.

[P3318R0] Ville Voutilainen. Throwing violation handlers, from an application programming perspective. https://wg21.link/p3318r0, 2024-05-22.

[P3362R0] Ville Voutilainen and Richard Corden. Static analysis and 'safety' of Contracts, P2900 vs. P2680/P3285. https://wg21.link/p3362r0, 2024-08-11.

[P3376R0] Andrzej Krzemieński. Contract assertions versus static analysis and 'safety'. https://wg21.link/p3376r0, 2024-10-14.

[P3386R0] Joshua Berne. Static Analysis of Contracts with P2900. https://wg21.link/p3386r0, 2024-10-15.

[P3390R0] Sean Baxter and Christian Mazakas. Safe C++. https://wg21.link/p3390r0, 2024-09-11.

[P3400R1] Joshua Berne. Controlling Contract-Assertion Properties. https://wg21.link/p3400r1, 2025-02-28.

[P3402R3] Marc-André Laverdière, Christopher Lapkowski, and Charles-Henri Gros. A Safety Profile Verifying Initialization. https://wg21.link/p3402r3, 2025-05-16.

[P3424R0] Alisdair Meredith. Define Delete With Throwing Exception Specification. https://wg21.link/p3424r0, 2024-12-17.

[P3446R0] Bjarne Stroustrup. Profile invalidation – eliminating dangling pointers. https://wg21.link/p3446r0, 2024-10-14.

[P3471R4] Konstantin Varlamov and Louis Dionne. Standard library hardening. https://wg21.link/p3471r4, 2025-02-14.

[P3499R1] Timur Doumler, Lisa Lippincott, and Joshua Berne. Exploring strict contract predicates. https://wg21.link/p3499r1, 2025-02-09.

[P3500R1] Timur Doumler, Gašper Ažman, Joshua Berne, and Ryan McDougall. Are Contracts "safe"? https://wg21.link/p3500r1, 2025-02-09.

[P3541R1] Andrzej Krzemieński. Violation handlers vs noexcept. https://wg21.link/p3541r1, 2025-01-06.

[P3577R0] John Lakos. Require a non-throwing default contract-violation handler. https://wg21.link/p3577r0, 2025-01-13.

[P3578R0] Ryan McDougall. What is Safety? https://wg21.link/p3578r0, 2024-12-12.

[P3589R2] Gabriel Dos Reis. C++ Profiles: The Framework. https://wg21.link/p3589r2, 2025-05-19.

[P3627R0] Ulfar Erlingsson. Easy-to-adopt security profiles for preventing RCE (remote code execution) in existing C++ code. https://wg21.link/p3627r0, 2025-02-11.

[P3650R0] Bjarne Stroustrup. 21st century C++. https://wg21.link/p3650r0, 2025-03-06.

[P3656R1] Herb Sutter and Gašper Ažman. Initial draft proposal for core language UB white paper: Process and major work items. https://wg21.link/p3656r1, 2025-03-23.

[P3700R0] Peter Bindels. Making Safe C++ Happen. https://wg21.link/p3700r0, 2025-05-19.

[P3716R0] Peter Bindels. Subsetting. https://wg21.link/p3716r0, 2025-05-19.

[P3754R0] Timur Doumler and Joshua Berne. Slides for EWG presentation of P3100R2 Implicit Contract Assertions. https://wg21.link/p3754r0, 2025-06-20.

[Sutter2024] Herb Sutter. C++ safety, in context. https://herbsutter.com/2024/03/11/safety-in-context/, 2024-03-11.

[Sutter2025] Herb Sutter. Crate-training Tiamat, un-calling Cthulhu: Taming the UB monsters in C++. https://herbsutter.com/2025/03/30/crate-training-tiamat-un-calling-cthulhutaming-the-ub-monsters-in-c/, 2025-03-30.

Appendix A: List of language UB

All wording is taken from the current C++ working paper [N5008]. Each row corresponds to one case of explicit core language UB. Rows are arranged by category, as defined in Section 2.2.1; within each category, rows follow the same order as that of the corresponding wording in [N5008].

Identifier	**Wording**	**Runtime-checkable**	**Locally checkable**	**Checking strategy**	**Replacement behaviour**
	I. Initialization				
{basic.indet. value}	[basic.indet]/2: Except in the following cases, if an indeterminate value is produced by an evaluation, the behavior is undefined, [...]	Yes	No	Track whether storage has been initialized	Only for built-in types: initialise default-initialised variables with erroneous value
	II. Bounds				
{basic.stc. alloc.zero. dereference}	[basic.stc.dynamic.allocation]/2: The effect of indirecting through a pointer returned from a request for zero size is undefined.	Yes	No	Track pointer provenance, insert bounds check	None
{expr.delete. mismatch}	[expr.delete]/2: In a single-object delete expression, the value of the operand of delete may be a null pointer value, a pointer value that resulted from a previous non-array new-expression, or a pointer to a base class subobject of an object created by such a new-expression. If not, the behavior is undefined.	Yes	No	Track pointer provenance, insert bounds check	None

Identifier	**Wording**	**Runtime-checkable**	**Locally checkable**	**Checking strategy**	**Replacement behaviour**
{expr. delete.array. mismatch}	[expr.delete]/2: In an array delete expression, the value of the operand of delete may be a null pointer value or a pointer value that resulted from a previous array new-expression whose allocation function was not a non-allocating form ([new.delete.placement]). If not, the behavior is undefined.	Yes	No	Track pointer provenance, insert bounds check	None
{expr.add.out. of.bounds}	[expr.add]/4: When an expression `J` that has integral type is added to or subtracted from an expression `P` of pointer type, the result has the type of `P`. If `P` evaluates to a null pointer value and `J` evaluates to 0, the result is a null pointer value. Otherwise, if `P` points to a (possibly-hypothetical) array element i of an array object `x` with n elements ([dcl.array]), the expressions `P + J` and `J + P` (where `J` has the value j) point to the (possibly-hypothetical) array element $i+j$ of `x` if $0 \leq i+j \leq n$ and the expression `P - J` points to the (possibly-hypothetical) array element $i-j$ of `x` if $0 \leq i-j \leq n$. Otherwise, the behavior is undefined.	Yes	Only if the array bound is statically known	Track pointer provenance, insert bounds check	None

Identifier	**Wording**	**Runtime-checkable**	**Locally checkable**	**Checking strategy**	**Replacement behaviour**
{expr.add.sub.diff.pointers}	[expr.add]/4: When an expression `J` that has integral type is added to or subtracted from an expression `P` of pointer type, the result has the type of `P`. If `P` evaluates to a null pointer value and `J` evaluates to 0, the result is a null pointer value. Otherwise, if `P` points to a (possibly-hypothetical) array element i of an array object `x` with n elements ([dcl.array]), the expressions `P + J` and `J + P` (where `J` has the value j) point to the (possibly-hypothetical) array element $i + j$ of `x` if $0 \leq i + j \leq n$ and the expression `P - J` points to the (possibly-hypothetical) array element $i - j$ of `x` if $0 \leq i - j \leq n$. Otherwise, the behavior is undefined.	Yes	Only if the array bound is statically known	Track pointer provenance, insert bounds check	None

III. Type and Lifetime

Identifier	**Wording**	**Runtime-checkable**	**Locally checkable**	**Checking strategy**	**Replacement behaviour**
{intro.object.implicit.create}	[intro.object]/11: For each operation that is specified as implicitly creating objects, that operation implicitly creates and starts the lifetime of zero or more objects of implicit-lifetime types ([basic.types.general]) in its specified region of storage if doing so would result in the program having defined behavior. If no such set of objects would give the program defined behavior, the behavior of the program is undefined.	Yes	No	Track whether storage can hold implicit lifetime objects	None

Identifier	**Wording**	**Runtime-checkable**	**Locally checkable**	**Checking strategy**	**Replacement behaviour**
{intro.object. implicit. pointer}	[intro.object]/11: Further, after implicitly creating objects within a specified region of storage, some operations are described as producing a pointer to a suitable created object. These operations select one of the implicitly-created objects whose address is the address of the start of the region of storage, and produce a pointer value that points to that object, if that value would result in the program having defined behavior. If no such pointer value would give the program defined behavior, the behavior of the program is undefined.	Yes	No	Track whether storage can hold implicit lifetime objects	None
{basic. align.object. alignment}	[basic.align]/1: Attempting to create an object ([intro.object]) in storage that does not meet the alignment requirements of the object's type is undefined behavior.	Yes	Yes	Insert alignment check	None

Identifier	**Wording**	**Runtime-checkable**	**Locally checkable**	**Checking strategy**	**Replacement behaviour**
{lifetime. outside. pointer. delete}	[basic.life]/7: Before the lifetime of an object has started but after the storage which the object will occupy has been allocated or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any pointer that represents the address of the storage location where the object will be or was located may be used but only in limited ways. [...] The program has undefined behavior if the pointer is used as the operand of a *delete-expression* [...]	Yes	No	Track whether storage is associated with an object of correct type within its lifetime	None
{lifetime. outside. pointer. member}	[basic.life]/7: [...] The program has undefined behavior if [...] the pointer is used to access a non-static data member or call a non-static member function of the object, [...]	Yes	No	Track whether storage is associated with an object of correct type within its lifetime	None
{lifetime. outside. pointer. virtual}	[basic.life]/7: [...] The program has undefined behavior if [...] the pointer is implicitly converted ([conv.ptr]) to a pointer to a virtual base class [...]	Yes	No	Track whether storage is associated with an object of correct type within its lifetime	None

Identifier	**Wording**	**Runtime-checkable**	**Locally checkable**	**Checking strategy**	**Replacement behaviour**
{lifetime.outside.pointer.static.cast}	[basic.life]/7: [...] The program has undefined behavior if [...] the pointer is used as the operand of a `static_cast` ([expr.static.cast]), except when the conversion is to pointer to *cv* `void`, or to pointer to *cv* `void` and subsequently to pointer to *cv* `char`, *cv* `unsigned char`, or *cv* `std::byte` ([cstddef.syn]) [...]	Yes	No	Track whether storage is associated with an object of correct type within its lifetime	None
{lifetime.outside.pointer.dynamic.cast}	[basic.life]/7: [...] The program has undefined behavior if [...] the pointer is used as the operand of a `dynamic_cast` ([expr.dynamic.cast]).	Yes	No	Track whether storage is associated with an object of correct type within its lifetime	None
{lifetime.outside.glvalue.access}	[basic.life]/8: Similarly, before the lifetime of an object has started but after the storage which the object will occupy has been allocated or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any glvalue that refers to the original object may be used but only in limited ways. [...] The program has undefined behavior if the glvalue is used to access the object [...]	Yes	No	Track whether storage is associated with an object of correct type within its lifetime	None
{lifetime.outside.glvalue.member}	[basic.life]/8: [...] The program has undefined behavior if [...] the glvalue is used to call a non-static member function of the object [...]	Yes	No	Track whether storage is associated with an object of correct type within its lifetime	None

Identifier	**Wording**	**Runtime-checkable**	**Locally checkable**	**Checking strategy**	**Replacement behaviour**
{lifetime.outside.glvalue.ref.virtual}	[basic.life]/8: [...] The program has undefined behavior if [...] the glvalue is bound to a reference to a virtual base class ([dcl.init.ref]) [...]	Yes	No	Track whether storage is associated with an object of correct type within its lifetime	None
{lifetime.outside.glvalue.dynamic.cast}	[basic.life]/8: [...] The program has undefined behavior if [...] the glvalue is used as the operand of a `dynamic_cast` ([expr.dynamic.cast]) or as the operand of `typeid`.	Yes	No	Track whether storage is associated with an object of correct type within its lifetime	None
{original.type.implicit.destructor}	[basic.life]/11: If a program ends the lifetime of an object of type `T` with static ([basic.stc.static]), thread ([basic.stc.thread]), or automatic ([basic.stc.auto]) storage duration and if `T` has a non-trivial destructor, and another object of the original type does not occupy that same storage location when the implicit destructor call takes place, the behavior of the program is undefined.	Yes	No	Track whether storage is associated with an object of correct type within its lifetime	None
{creating.within.const.complete.obj}	[basic.life]/12: Creating a new object within the storage that a const, complete object with static, thread, or automatic storage duration occupies, or within the storage that such a const object used to occupy before its lifetime ended, results in undefined behavior.	Yes	No	Track whether storage is associated with a `const` object	None

Identifier	**Wording**	**Runtime-checkable**	**Locally checkable**	**Checking strategy**	**Replacement behaviour**
{basic. compound. invalid. pointer}	[basic.compound]/4: If a pointer value P is used in an evaluation E and P is not valid in the context of E, then the behavior is undefined if E is an indirection ([expr.unary.op]) or an invocation of a deallocation function ([basic.stc.dynamic.deallocation]) [...]	Yes	No	Track whether storage has been allocated and freed	None
{expr.basic. lvalue.strict. aliasing. violation}	[basic.lval]/11.3: If a program attempts to access ([defns.access]) the stored value of an object through a glvalue through which it is not type-accessible, the behavior is undefined.	Yes	No	Track whether storage is associated with an object of correct type within its lifetime	None
{expr.basic. lvalue.union. initialization}	[basic.lval]/11.3: If a program invokes a defaulted copy/move constructor or copy/move assignment operator for a union of type `U` with a glvalue argument that does not denote an object of type *cv* `U` within its lifetime, the behavior is undefined.	Yes	No	Track whether storage is associated with an object of correct type within its lifetime	None
{expr.type. reference. lifetime}	[expr.type]/1: If a pointer to X would be valid in the context of the evaluation of the expression ([basic.fundamental]), the result designates X; otherwise, the behavior is undefined.	Yes	No	Track whether storage has been allocated and freed	None

Identifier	Wording	Runtime-checkable	Locally checkable	Checking strategy	Replacement behaviour
{conv. lval.valid. representation}	[conv.lval]/3.4: Otherwise, if the bits in the value representation of the object to which the glvalue refers are not valid for the object's type, the behavior is undefined.	Yes	No	Track whether storage is associated with an object of correct type within its lifetime	Coerce invalid value represen-tations into erroneous values
{conv.ptr. virtual.base}	[conv.ptr]/3: Otherwise, if `B` is a virtual base class of `D` and `v` does not point to an object whose type is similar ([conv.qual]) to `D` and that is within its lifetime or within its period of construction or destruction ([class.cdtor]), the behavior is undefined.	Yes	Only for the null pointer case	Track whether storage is associated with an object of correct type within its lifetime or an object currently being constructed or destroyed; insert null pointer check	None
{conv. member. missing. member}	[conv.mem]/2: If class `D` does not contain the original member and is not a base class of the class containing the original member, the behavior is undefined.	Yes	No	Track which type the pointer to member originated from	None
{expr.call. different. type}	[expr.call]/5: Calling a function through an expression whose function type is not call-compatible with the type of the called function's definition results in undefined behavior.	Yes	No	Track type information of function based on address	None
{expr.ref. member.not. similar}	[expr.ref]/9: If `E2` is a non-static member and the result of `E1` is an object whose type is not similar ([conv.qual]) to the type of `E1`, the behavior is undefined.	Yes	No	Track whether storage is associated with an object of correct type within its lifetime	None

Identifier	**Wording**	**Runtime-checkable**	**Locally checkable**	**Checking strategy**	**Replacement behaviour**
{expr.dynamic.cast.pointer.lifetime}	[expr.dynamic.cast]/7: If `v` has type "pointer to *cv* `U`" and `v` does not point to an object whose type is similar ([conv.qual]) to `U` and that is within its lifetime or within its period of construction or destruction ([class.cdtor]), the behavior is undefined.	Yes	Only for the null pointer case	Track whether storage is associated with an object of correct type within its lifetime or an object currently being constructed or destroyed; insert null pointer check	None
{expr.dynamic.cast.glvalue.lifetime}	[expr.dynamic.cast]/7: If `v` is a glvalue of type `U` and `v` does not refer to an object whose type is similar to `U` and that is within its lifetime or within its period of construction or destruction, the behavior is undefined.	Yes	No	Track whether storage is associated with an object of correct type within its lifetime or an object currently being constructed or destroyed	None
{expr.static.cast.base.class}	[expr.static.cast]/2: An xvalue of type "*cv*1 `B`" can be cast to type "rvalue reference to *cv*2 `D`" with the same constraints as for an lvalue of type "*cv*1 `B`". If the object of type "*cv*1 `B`" is actually a base class subobject of an object of type `D`, the result refers to the enclosing object of type `D`. Otherwise, the behavior is undefined.	Yes	No	Track whether storage is associated with an object of correct type within its lifetime	None

Identifier	**Wording**	**Runtime-checkable**	**Locally checkable**	**Checking strategy**	**Replacement behaviour**
{expr.static. cast.downcast. wrong.derived. type}	[expr.static.cast]/11: If the prvalue of type "pointer to *cv*1 `B`" points to a `B` that is actually a base class subobject of an object of type `D`, the resulting pointer points to the enclosing object of type `D`. Otherwise, the behavior is undefined.	Yes	Only for the null pointer case	Track whether storage is associated with an object of correct type within its lifetime or an object currently being constructed or destroyed; insert a null pointer check	None
{expr.static. cast.does.not. contain. orignal. member}	[expr.static.cast]/12: If class `B` contains the original member, or is a base class of the class containing the original member, the resulting pointer to member points to the original member. Otherwise, the behavior is undefined.	Yes	No	Track which type the pointer to member originated from	None
{expr.unary. dereference}	[expr.unary.op]/1: If the operand points to an object or function, the result denotes that object or function; otherwise, the behavior is undefined except as specified in [expr.typeid].	Yes	Only for the null pointer case	Track whether storage is associated with an object of correct type within its lifetime; track whether the address is associated with a function; insert a null pointer check	None
{expr.delete. dynamic.type. differ}	[expr.delete]/3: In a single-object delete expression, if the static type of the object to be deleted is not similar ([conv.qual]) to its dynamic type and the selected deallocation function (see below) is not a destroying operator delete, the static type shall be a base class of the dynamic type of the object to be deleted and the static type shall have a virtual destructor or the behavior is undefined.	Yes	No	Track dynamic type of non-polymorphic objects	None

Identifier	**Wording**	**Runtime-checkable**	**Locally checkable**	**Checking strategy**	**Replacement behaviour**
{expr.delete.dynamic.array.dynamic.type.differ}	[expr.delete]/3: In an array delete expression, if the dynamic type of the object to be deleted is not similar to its static type, the behavior is undefined.	Yes	No	Track dynamic type of non-polymorphic objects	None
{expr.mptr.oper.not.contain.member}	[expr.mptr.oper]/4: Abbreviating *pm-expression*`.*`*cast-expression* as `E1.*E2`, `E1` is called the object expression. If the result of `E1` is an object whose type is not similar to the type of `E1`, or whose most derived object does not contain the member to which `E2` refers, the behavior is undefined.	Yes	No	Track which type the pointer to member originated from and the dynamic type of non-polymorphic objects	None
{expr.mptr.oper.member.func.null}	[expr.mptr.oper]/6: The result of a .* expression whose second operand is a pointer to a member function is a prvalue. If the second operand is the null member pointer value, the behavior is undefined.	Yes	Yes	Insert null pointer check	None
{expr.add.not.similar}	[expr.add]/6: For addition or subtraction, if the expressions `P` or `Q` have type "pointer to *cv* `T`", where `T` and the array element type are not similar, the behavior is undefined.	Yes	No	Track whether storage is associated with an object of correct type	None

Identifier	**Wording**	**Runtime-checkable**	**Locally checkable**	**Checking strategy**	**Replacement behaviour**
{expr.assign. overlap}	[expr.assign]/7: If the value being stored in an object is read via another object that overlaps in any way the storage of the first object, then the overlap shall be exact and the two objects shall have the same type, otherwise the behavior is undefined.	Yes	Yes	Check overlap of the two address ranges	None
{dcl.type.cv. modify.const. obj}	[dcl.type.cv]/4: Any attempt to modify ([expr.assign], [expr.post.incr], [expr.pre.incr]) a const object ([basic.type.qualifier]) during its lifetime ([basic.life]) results in undefined behavior.	Yes	No	Track whether storage is associated with a `const` object	None
{dcl.type. cv.access. volatile}	[dcl.type.cv]/5: If an attempt is made to access an object defined with a volatile-qualified type through the use of a non-volatile glvalue, the behavior is undefined.	Yes	No	Track whether storage is associated with a `volatile` object	None
{dcl.ref. incompatible. function}	[dcl.ref]/6: Attempting to bind a reference to a function where the converted initializer is a glvalue whose type is not call-compatible ([expr.call]) with the type of the function's definition results in undefined behavior.	Yes	No	Track the types of all functions based on their addresses	None

Identifier	**Wording**	**Runtime-checkable**	**Locally checkable**	**Checking strategy**	**Replacement behaviour**
{dcl.ref. incompatible. type}	[dcl.ref]/6: Attempting to bind a reference to an object where the converted initializer is a glvalue through which the object is not type-accessible ([basic.lval]) results in undefined behavior.	Yes	No	Track whether storage is associated with an object of correct type	None
{dcl.ref. uninitialized. reference}	[dcl.ref]/6: The behavior of an evaluation of a reference ([expr.prim.id], [expr.ref]) that does not happen after ([intro.races]) the initialization of the reference is undefined.	Yes	No	Track whether references have been initialised	None
{class.dtor.no. longer.exists}	[class.dtor]/18: Once a destructor is invoked for an object, the object's lifetime ends; the behavior is undefined if the destructor is invoked for an object whose lifetime has ended ([basic.life]).	Yes	No	Track whether storage is associated with an object of correct type within its lifetime	None
{class. abstract. pure.virtual}	[class.abstract]/6: Member functions can be called from a constructor (or destructor) of an abstract class; the effect of making a virtual call ([class.virtual]) to a pure virtual function directly or indirectly for the object being created (or destroyed) from such a constructor (or destructor) is undefined.	Yes	Yes	Insert a `pre(false)` into the pure virtual stub pointed to from the base-class vtable	None

Identifier	**Wording**	**Runtime-checkable**	**Locally checkable**	**Checking strategy**	**Replacement behaviour**
{class.base.init.mem.fun}	[class.base.init]/16: Member functions (including virtual member functions, [class.virtual]) can be called for an object under construction or destruction. Similarly, an object under construction or destruction can be the operand of the `typeid` operator ([expr.typeid]) or of a `dynamic_cast` ([expr.dynamic.cast]). However, if these operations are performed during evaluation of a *ctor-initializer* (or in a function called directly or indirectly from a *ctor-initializer*) before all the *mem-initializer*s for base classes have completed, a precondition assertion of a constructor, or a postcondition assertion of a destructor ([dcl.contract.func]), the program has undefined behavior.	Yes	No	Track whether objects are currently being constructed or destroyed	None
{class.cdtor.before.ctor.after.dtor}	[class.cdtor]/1: For an object with a non-trivial constructor, referring to any non-static member or base class of the object before the constructor begins execution results in undefined behavior.	Yes	No	Track whether objects are currently being constructed or destroyed	None

Identifier	**Wording**	**Runtime-checkable**	**Locally checkable**	**Checking strategy**	**Replacement behaviour**
{class.cdtor. before.ctor. after.dtor}	[class.cdtor]/1: For an object with a non-trivial destructor, referring to any non-static member or base class of the object after the destructor finishes execution results in undefined behavior.	Yes	No	Track whether objects are currently being constructed or destroyed	None
{class.cdtor. convert.or. form.pointer}	[class.cdtor]/3: To explicitly or implicitly convert a pointer (a glvalue) referring to an object of class `X` to a pointer (reference) to a direct or indirect base class `B` of `X`, the construction of `X` and the construction of all of its direct or indirect bases that directly or indirectly derive from `B` shall have started and the destruction of these classes shall not have completed, otherwise the conversion results in undefined behavior.	Yes	No	Track whether objects are currently being constructed or destroyed	None
{class.cdtor. convert.or. form.pointer}	[class.cdtor]/3: To form a pointer to (or access the value of) a direct non-static member of an object `obj`, the construction of `obj` shall have started and its destruction shall not have completed, otherwise the computation of the pointer value (or accessing the member value) results in undefined behavior.	Yes	No	Track whether objects are currently being constructed or destroyed	None

Identifier	**Wording**	**Runtime-checkable**	**Locally checkable**	**Checking strategy**	**Replacement behaviour**
{class.cdtor. virtual.not.x}	[class.cdtor]/4: If the virtual function call uses an explicit class member access ([expr.ref]) and the object expression refers to the complete object of `x` or one of that object's base class subobjects but not `x` or one of its base class subobjects, the behavior is undefined.	Yes	No	Track whether objects are currently being constructed or destroyed	None
{class.cdtor. typeid}	[class.cdtor]/5: If the operand of `typeid` refers to the object under construction or destruction and the static type of the operand is neither the constructor or destructor's class nor one of its bases, the behavior is undefined.	Yes	No	Track whether objects are currently being constructed or destroyed	None
{class.cdtor. dynamic.cast}	[class.cdtor]/6: If the operand of the `dynamic_cast` refers to the object under construction or destruction and the static type of the operand is not a pointer to or object of the constructor or destructor's own class or one of its bases, the `dynamic_cast` results in undefined behavior.	Yes	No	Track whether objects are currently being constructed or destroyed	None
{except. handle. handler. ctor.dtor}	[except.handle]/11: Referring to any non-static member or base class of an object in the handler for a *function-try-block* of a constructor or destructor for that object results in undefined behavior.	Yes	No	Track whether objects are currently being constructed or destroyed	None

Identifier	**Wording**	**Runtime-checkable**	**Locally checkable**	**Checking strategy**	**Replacement behaviour**
IV. Arithmetic					
{expr.expr. eval}	[expr.pre]/4: If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined.	Yes	Yes	Insert a check of whether the value is valid	Coerce into erroneous value
{conv.double. out.of.range}	[conv.double]/2: A prvalue of floating-point type can be converted to a prvalue of another floating-point type with a greater or equal conversion rank ([conv.rank]). [...] If the source value can be exactly represented in the destination type, the result of the conversion is that exact representation. If the source value is between two adjacent destination values, the result of the conversion is an implementation-defined choice of either of those values. Otherwise, the behavior is undefined.	Yes	Yes	Insert a check of whether the value is valid	Coerce into erroneous value
{conv.fpint. float.not. represented}	[conv.fpint]/1: A prvalue of a floating-point type can be converted to a prvalue of an integer type. The conversion truncates; that is, the fractional part is discarded. The behavior is undefined if the truncated value cannot be represented in the destination type.	Yes	Yes	Insert a check of whether the value is valid	Coerce into erroneous value

Identifier	**Wording**	**Runtime-checkable**	**Locally checkable**	**Checking strategy**	**Replacement behaviour**
{conv.fpint.int.not.represented}	[conv.fpint]/2: A prvalue of an integer type or of an unscoped enumeration type can be converted to a prvalue of a floating-point type. [...] If the value being converted is outside the range of values that can be represented, the behavior is undefined.	Yes	Yes	Insert a check of whether the value is valid	Coerce into erroneous value
{expr.static.cast.enum.outside.range}	[expr.static.cast]/9: If the enumeration type does not have a fixed underlying type, the value is unchanged if the original value is within the range of the enumeration values ([dcl.enum]), and otherwise, the behavior is undefined.	Yes	Yes	Insert a check of whether the value is valid	Coerce into erroneous value
{expr.static.cast.fp.outside.range}	[expr.static.cast]/10: A prvalue of floating-point type can be explicitly converted to any other floating-point type. If the source value can be exactly represented in the destination type, the result of the conversion has that exact representation. If the source value is between two adjacent destination values, the result of the conversion is an implementation-defined choice of either of those values. Otherwise, the behavior is undefined.	Yes	Yes	Insert a check of whether the value is valid	Coerce into erroneous value

Identifier	**Wording**	**Runtime-checkable**	**Locally checkable**	**Checking strategy**	**Replacement behaviour**
{expr.mul.div.by.zero}	[expr.mul]/4: The binary `/` operator yields the quotient, and the binary `%` operator yields the remainder from the division of the first expression by the second. If the second operand of `/` or `%` is zero, the behavior is undefined.	Yes	Yes	Insert a check of whether the second operand is zero	Coerce into erroneous value
{expr.mul.representable.type.result}	[expr.mul]/4: For integral operands, the `/` operator yields the algebraic quotient with any fractional part discarded; if the quotient `a/b` is representable in the type of the result, `(a/b)*b + a%b` is equal to `a`; otherwise, the behavior of both `a/b` and `a%b` is undefined.	Yes	Yes	Insert a check of whether the value is valid	Coerce into erroneous value
{expr.shift.neg.and.width}	[expr.shift]/1: The behavior is undefined if the right operand is negative, or greater than or equal to the width of the promoted left operand.	Yes	Yes	Insert check whether right operand is valid	Coerce into erroneous value

V. Threading

{intro.races.data}	[intro.races]/17: Any such data race results in undefined behavior.	Partially	No	Track from which threads memory is accessed and when accesses synchronise with each other; only practical for a subset of cases (see TSan)	Make all primitive memory accesses implicitly atomic

Identifier	**Wording**	**Runtime-checkable**	**Locally checkable**	**Checking strategy**	**Replacement behaviour**
{intro. progress. stops}	[intro.progress]/1: The implementation may assume that any thread will eventually do one of the following: terminate, invoke the function `std::this_thread::yield` ([thread.thread.this]), make a call to a library I/O function, perform an access through a volatile glvalue, perform an atomic or synchronization operation other than an atomic modify-write operation ([atomics.order]), or continue execution of a trivial infinite loop ([stmt.iter.general]).	No	No	No checking strategy exists as whether a thread will make progress is undecidable	Do nothing

VI. Sequencing

{intro. execution. unsequenced. modification}	[conv.rank]/10: The behavior is undefined if a side effect on a memory location ([intro.memory]) or starting or ending the lifetime of an object in a memory location is unsequenced relative to another side effect on the same memory location, starting or ending the lifetime of an object occupying storage that overlaps with the memory location, or a value computation using the value of any object in the same memory location, and the two evaluations are not potentially concurrent ([intro.multithread]).	Yes	Yes	Identify all potential read operations that are not sequenced with respect to each given write operation; insert checks to identify if those operations are referencing the same address	Sequence operations in some unspecified order

Identifier	**Wording**	**Runtime-checkable**	**Locally checkable**	**Checking strategy**	**Replacement behaviour**
VII. Assumptions					
{dcl.attr.assume.false}	[dcl.attr.assume]/1: If the converted expression would evaluate to true at the point where the assumption appears, the assumption has no effect. Otherwise, evaluation of the assumption has runtime undefined behavior.	No	No	No automatic checking strategy is possible because the predicate cannot be, in general, proven to be free of side effects; instead, the user has to change `[[assume(x)]]` to `contract_assert<may_be_assumed>(x)` and select an appropriate evaluation semantic	Ignore the assumption
VIII. Control Flow					
{basic.start.main.exit.during.destruction}	[basic.start.main]/4: If `std::exit` is invoked during the destruction of an object with static or thread storage duration, the program has undefined behavior.	Yes	No	Track whether static or thread-local objects are currently being destroyed	None
{basic.start.term.use.after.destruction}	[basic.start.term]/4: If a function contains a block variable of static or thread storage duration that has been destroyed and the function is called during the destruction of an object with static or thread storage duration, the program has undefined behavior if the flow of control passes through the definition of the previously destroyed block variable.	Yes	No	Track the lifetime of static objects	None

Identifier	**Wording**	**Runtime-checkable**	**Locally checkable**	**Checking strategy**	**Replacement behaviour**
{stmt.return.flow.off}	[stmt.return]/4: Otherwise, flowing off the end of a function that is neither `main` ([basic.start.main]) nor a coroutine ([dcl.fct.def.coroutine]) results in undefined behavior.	Yes	Yes	Insert `contract_assert(false)` at end of *function-body*	Only for built-in return types: return erroneous value
{stmt.dcl.local.static.init.recursive}	[stmt.dcl]/3: If control re-enters the declaration recursively while the variable is being initialized, the behavior is undefined.	Yes	No	Insert a recursion counter into a guard for static and thread-local object construction	None
{dcl.attr.noreturn.eventually.returns}	[dcl.attr.noreturn]/2: If a function `f` is invoked where `f` was previously declared with the `noreturn` attribute and that invocation eventually returns, the behavior is runtime-undefined.	Yes	Yes	Insert `post(false)`	None

Identifier	**Wording**	**Runtime-checkable**	**Locally checkable**	**Checking strategy**	**Replacement behaviour**
IX. Replacement Functions					
{basic.stc.alloc.dealloc.constraint}	[basic.stc.dynamic.general]/3: If the behavior of an allocation or deallocation function does not satisfy the semantic constraints specified in [basic.stc.dynamic.allocation] and [basic.stc.dynamic.deallocation], the behavior is undefined.	Partially: some constraints can be checked locally (e.g., allocation function does not return null); others cannot be checked at all.	Partially	Insert checks where possible	None
{basic.stc.alloc.dealloc.throw}	[basic.stc.dynamic.deallocation]/4: If a deallocation function terminates by throwing an exception, the behavior is undefined.	Make ill-formed via [P3424R0]	—	—	
{expr.new.non.allocating.null}	[expr.new]/22: If the allocation function is a non-allocating form ([new.delete.placement]) that returns null, the behavior is undefined.	Yes	Yes	Insert `post(r:   r)`	None

Identifier	**Wording**	**Runtime-checkable**	**Locally checkable**	**Checking strategy**	**Replacement behaviour**
		X. Coroutines			
{stmt.return.coroutine.flow.off}	[stmt.return.coroutine]/3: If a search for the name `return_void` in the scope of the promise type finds any declarations, flowing off the end of a coroutine's *function-body* is equivalent to a `co_return` with no operand; otherwise flowing off the end of a coroutine's *function-body* results in undefined behavior.	Yes	Yes	Insert `contract_assert(false)` at end of *function-body* if no `return_void` function is provided	Only for built-in return types: return erroneous value
{dcl.fct.def.coroutine.resume.not.suspended}	[dcl.fct.def.coroutine]/9: Invoking a resumption member function for a coroutine that is not suspended results in undefined behavior.	Yes	No	Track the suspension state associated with every coroutine handle	None
{dcl.fct.def.coroutine.destroy.not.suspended}	[dcl.fct.def.coroutine/12: If `destroy` is called for a coroutine that is not suspended, the program has undefined behavior.	Yes	No	Track the suspension state associated with every coroutine handle	None