Implicit contract assertions

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

Document #: P3100R2
Date: 2025-05-19
Project: Programming Language C++
Audience: EWG

Abstract

In this paper, we enumerate all cases of core language undefined behaviour explicitly specified in C++, group them into categories, classify them along a number of relevant criteria, and discuss appropriate mitigation strategies. The conditions under which such undefined behaviour will occur can, in many cases, be identified by a runtime check. We describe how such 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 introduce well-defined fallback behaviour to replace undefined behaviour wherever possible. Building on Contracts as adopted for C++26, we provide a generic framework that can be incorporated into the ongoing core language UB white paper [P3656R1], fundamentally changing the landscape of how undefined behaviour is approached in C++.

Contents

1 Introduction

Eliminating or at least meaningfully reducing the amount of *undefined behaviour* (UB) is an important objective for the future evolution of C++ and crucial for improving the "safety"[1] and security of C++ programs. 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 framework for a systematic treatment of core language undefined behaviour in C++: the pursuit of a core language UB white paper in the C++26 timeframe, covering erroneous behaviour (EB), Profiles, and Contracts. The current version of that white paper is [P3656R1], proposing the process and major work items. The proposed process calls for papers to be adopted into the white paper working draft via EWG approval. This is the first such paper.

Further, as major work items, [P3656R1] proposes to enumerate and group all language UB in C++, identify tools to address them, and take a first pass at penciling in which tool to use for each UB case. The goal of this paper is to contribute to all the above major work items.

In Section 2, we identify and enumerate all core language UB explicitly specified in the C++ Standard. We then group all core language UB into broad categories such as "Arithmetic" and "Bounds". We then classify cases of UB along several relevant criteria: whether they are locally diagnosable, how expensive that diagnosis is, and whether there is well-defined fallback behaviour for each case. Finally, we discuss appropriate mitigation strategies for all identified cases of UB and find that runtime checking is an appropriate strategy for the large majority of cases.

In Section 3, we systematically introduce such runtime checks to C++ via *implicit contract assertions*, building on the basic framework of Contracts adopted for C++26 via [P2900R14]. We describe how implicit contract assertions should be specified and how to apply them to all appropriate cases of core language UB. For cases of UB where well-defined fallback behaviour exists, we discuss how specifying it allows the program to continue execution past a violated implicit contract assertion without UB. We conclude by proposing an escape hatch to mitigate the runtime cost of such fallback behaviour and avoid performance regressions.

In Section 4, 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. In Section 5, we propose wording for approval into the white paper that implements the design discussed in Section 3.

The first revision (R0) of this paper was published in May 2024. Following informal discussions at the St. Louis meeting, the paper was revised (R1) and presented to SG21 at the Wrocław meeting. SG21 voted *unanimously* in favour of our direction for implicit contract assertions.

SG21, Wrocław, 2024-11-22, Poll 6

We support the direction of P3100R1 and encourage the authors to come back with a fully specified proposal.

SF	F	N	A	SA
19	6	0	0	0

Result: Consensus

[1] In this paper, we place quotes around the term "safety" when unqualified due to the crucial importance of distinguishing between conflicting definitions of that term, such as functional safety, language safety, memory safety, and so on. See [P3376R0], [P3500R1], and [P3578R0] for a discussion of those definitions as well as recommendations regarding usage of the term "safety" in the context of C++.

The present revision (R2) is a complete rewrite of the paper that considers the above poll, the adoption of [P2900R14] into the C++26 working paper, EWG's decision to publish the core language UB white paper, the current state of that white paper [P3656R1], discussions at the Hagenberg meeting, and the feedback we received on the previous revision of this paper.

2 Analysis

2.1 Methodology and scope

For this paper, we manually inspected all occurrences of the word "undefined" in the current C++ working paper [N5008]. We then constructed a list of all cases of explicitly specified core language UB. Our complete list, containing 90 cases of UB, can be found in Appendix A of this paper.

Another group of WG21 members is currently engaged in an effort, based on work by Shafik Yaghmour ([P1705R1], [P3075R0]) and following the process outlined in [P3656R1], to enumerate cases of core language UB directly in the C++ Standard document LaTeX source. Our list has been created independently from that effort.

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++ Standard's clause identifiers, which we place between [square brackets]. Wherever possible, we use here the same stable identifiers as those used in the LaTeX-based effort. However, some differences occur between the two lists (and, therefore, also between the stable identifiers used) because we identified a number of potential defects and omissions in the other list. We are actively contributing to the LaTeX-based effort toward merging the two lists.

Note that unlike the LaTeX-based effort, our list excludes cases of IFNDR because this paper focuses on runtime mitigation strategies. While UB is fundamentally a runtime property of a particular program execution and thus runtime mitigation is a natural approach, IFNDR typically represents link-time issues and is, therefore, out of scope for this paper.

Note further that we exclude *library undefined behaviour* since the natural mitigation approach for it is to make use of contract assertions (`pre`, `post`, and `contract_assert`) in library implementations and, when possible, 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++ Standard (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; [2] that case is, therefore, also excluded from our list.

2.2 Basic categories of UB

We found that all identified cases of core language UB can be broadly classified into twelve 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.

[2][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.

III. **Type and Lifetime** — 45 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** — 1 case. Performing two concurrent accesses, at least one of which is modifying, to the same memory location from different threads where neither access happens before the other, i.e., a data race.

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** — 6 cases. Undefined behaviour due to errors in control flow. Examples: flowing off the end of a function; re-entering the same declaration recursively when initialising a static variable.

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** — 2 cases. Misusing coroutine machinery. Examples: destroying a coroutine that is not suspended; invoking a resumption member function for a coroutine that is not suspended.

XI. **Templates** — 1 case. Infinite recursion during template instantiation.

XII. **Preprocessor** — 8 cases. Misusing preprocessor directives. Examples: `#define`-ing a predefined macro name; passing an out-of-range integer to the `#line` directive.

The categories Initialisation, Bounds, and Type and Lifetime correspond to the common terms *initialisation safety*, *bounds safety*, *type safety*, and *lifetime safety*, respectively, and collectively represent undefined behaviour that is commonly referred to with the umbrella term *memory safety*.

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. Note further that this combined category of Type and Lifetime contains the majority (58%) of all cases of UB that we identified.

The next two categories, Arithmetic and Threading, correspond to the common terms *arithmetic safety* and *thread safety*, respectively; the latter contains only one case of UB, data races.

The following category, Sequencing, also contains just one case of UB: unsequenced operations, such as `i++ + ++i`. Grouping 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 2.7, 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 five categories (Control Flow, Replacement Functions, Coroutines, Templates, and Preprocessor) are less frequently discussed in the current discourse around UB. Nevertheless, they represent UB that needs to be mitigated.

2.3 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.

Other cases of UB, such as those in categories Arithmetic and Threading, are a common source of program defects, and those program defects do sizeable damage to existing software, so mitigating them offers a lot of value. To our knowledge, however, they are not commonly exploited by malicious attackers.

Eventually, mitigating all UB currently considered to be a 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, while 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++ Standard (see [Sutter2024], [P3500R1], and [P3578R0]).

2.4 Local checkability

The second question [P3656R1] asks is which cases of UB are "efficiently locally diagnosable". Here, we split this question into two separate questions: which cases of UB are locally diagnosable in principle (this subsection), and the estimated cost of that diagnosis (next subsection).

Most cases of UB in the security-critical Initialisation, Bounds, and Type and Lifetime categories are, in general, *not* locally diagnosable. 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 that the base class vtable points to. All other cases of UB in the Initialisation, Bounds, and Type and Lifetime categories require, to be diagnosable, additional instrumentation of the kind that is implemented in sanitisers such as ASan and UBSan (see Section 2.5 for further discussion).

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 ({intro.races.data}) is not locally diagnosable, but UB in the Sequencing category ({intro.execution.unsequenced.modification}) is.

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 not generally possible to do efficiently and is outright impossible in the presence of an opaque function call.

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

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.

All cases of UB in the Coroutine category are not locally diagnosable since being so would require tracking runtime state information that is not currently maintained within the coroutine handle in most implementations.

UB in the Templates and Preprocessor categories is unique in that it does not actually represent runtime UB, and therefore, runtime diagnosis makes no sense. In particular, UB in the Templates category ({temp.inst.inf.recursion}) is diagnosable at compile time, while UB in the Preprocessor category should be specified as IFNDR instead (see Section 2.7).

2.5 Cost of diagnosis

Considering locally diagnosable and not locally diagnosable cases of UB separately is useful to estimate the cost of diagnosis. 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 of diagnosis in real implementations, and we do not present benchmarks. This work is left for future studies.

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.

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 they 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.

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, and 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 practical, 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 UB 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 (at least) an order of magnitude, unless special hardware-acceleration or some other radically new technology for these checks becomes available. We discuss some of the consequences of this fundamental dilemma in Section 2.7.

Given the substantial overhead of the instrumentation itself, i.e., involving both a runtime cost and a cost in memory, how expensive the actual *checks* would be (whether a specific pointer is valid at a specific time, etc.) is not particularly important because the performance penalty would be dominated by the instrumentation overhead.

2.6 Well-defined fallback behaviour

If we want to turn UB into well-defined behaviour, a useful question is whether any well-defined behaviour actually exists that the affected operation could be defined to have instead of UB in the presence of a bug. Here, we call such well-defined behaviour *fallback behaviour*.

We could also use the term *erroneous behaviour* (EB), which is conceptually the same thing. However, since the approval of [P2795R5] for C++26, EB has very specific semantics. Here, we are considering the wider concept of introducing new well-defined behaviour for error cases, rather than the exact semantics that EB has in C++26, so we use a different term for now.

For fallback behaviour to happen, the compiler must supply the necessary instructions. However, in the vast majority of cases, core language UB is fundamentally *not* diagnosable at compile time (see Section 2.4); i.e., whether or not the UB will occur depends on runtime parameters. Fallback behaviour *cannot*, therefore, depend on knowing that an error occurred. For non-locally-diagnosable UB, fallback behaviour also cannot depend on any additional instrumentation being present.

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

For UB in the Initialization category ({basic.indet.value}), fallback behaviour is sometimes possible for built-in types: return an erroneous value instead. For variables with automatic storage duration, this fallback behaviour is already part of C++26 as EB via [P2795R5], because for this case, the fallback behaviour is particularly cheap. The same fallback behaviour could also be employed for dynamically allocated variables but at greater cost (see [P2723R1] Section 6 for discussion).

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 initialized) 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.

Further, for user-defined types, this fallback behaviour is not applicable in general. 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 fallback behaviour for the general case.

Practically *none* of the UB in the categories of Bounds and Type and Lifetime has fallback 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.[3] For example, in the code example given in the C++ Standard,

```
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 fallback behaviour: if an arithmetic operation would produce an inappropriate value, it can be coerced into an erroneous value instead, at the cost of incurring significant performance overhead on common arithmetic operations.

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

The fallback 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 3.5.

Finally, we can define partial fallback behaviour for two cases of UB in the Control Flow category ({stmt.return.flow.off} and {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.

[3]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.

This case is analogous to {basic.indet.value}; again, no fallback behaviour exists for user-defined return types in the general case.

For all other cases of runtime-checkable UB (63 cases in total), we cannot imagine any meaningful fallback behaviour.

2.7 Mitigation strategies

In this section, we attempt to systematically identify, at a very high level, candidate mitigation strategies for all cases of core language UB.

Arguably, the best mitigation strategy is to make the offending construct ill-formed, but we can do so only for cases in which we can unambiguously identify at *compile time* that UB will occur for all inputs; otherwise, we would break existing correct C++ code. Only one case of UB fits this particular situation: {temp.inst.inf.recursion}. This case should be specified as ill-formed instead of UB.

Similarly, UB in the Preprocessor category should not be specified as UB either. We defer to [P2843R2], which proposes to specify all cases in this category as IFNDR instead, placing them outside of the scope of this paper.

Two more cases of UB should not actually be considered UB. The first is {class.dtor.not.class.type}. While the wording for this case says that "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", this situation is not a new case of UB and is already omitted from the specification of other cases of UB elsewhere. This section should, therefore, be a non-normative note referring to those sections.

The second is {basic.stc.alloc.dealloc.throw}. There is no good reason why throwing an exception from a deallocation function should cause UB. Instead, we should enforce that deallocation functions have a nonthrowing exception specification. This solution is proposed in [P3424R0], and we refer to that paper for mitigating this case of UB.

We are left with 79 cases of UB for which we need to identify candidate mitigation strategies. All of those cases represent *runtime* UB that cannot be diagnosed at compile time. Therefore, one possible mitigation strategy for *all* those cases of UB is to insert *runtime* checks.

Fundamentally, inserting runtime checks is possible for 78 out of those 79 cases, the only exception being {dcl.attr.assume.false} where, as we saw in Section 2.4, no automated runtime checking is possible in the general case (see also Section 3.5) because we cannot prove that the assumption predicate is side-effect free.

However, as we saw in Section 2.4, the majority of those cases (60 out of 78) are not locally diagnosable and require expensive sanitiser-like instrumentation to perform the checks. And even for those 18 cases of UB that are locally diagnosable and do not require additional instrumentation to insert runtime checks, in most cases the checks themselves will have a significant runtime overhead. Therefore, the checks need to be *optional*: we need a mechanism to enable and disable each kind of check, and we cannot require an implementation to support all checks.

For example, a compiler may choose to support enabling runtime checks for arithmetic UB (they already do today for some cases; for example, GCC offers the `-ftrapv` flag, which enables checks for signed integer overflow) while not supporting any checks that require expensive instrumentation. On the other hand, a different compiler that comes with a suite of a sanitisers may choose to support some subset of those more expensive checks (and again, they already do today, just not in a standardised fashion).

Defining such optional runtime checks for all those 79 cases of UB is, therefore, useful in itself. These checks cost *nothing* unless they are turned on, and no implementation is actually *required* to implement them, yet specifying them in the Standard has a number of advantages: it allows us to assign standard names and categories to them (see also Section 4); it allows for implementations of such runtime checks (including existing compiler options and sanitisers) to leverage a shared paradigm and shared terminology; and it brings those tools into the scope of the C++ Standard.

The natural way to introduce such optional runtime checks to C++ is to leverage the Contracts framework. All the necessary machinery and terminology for optional runtime checks — called *contract assertions* — are already present in C++26, thanks to the foundation laid by [P2900R14]. The only missing part is to introduce compiler-generated checks, i.e., *implicit* contract assertions, in addition to the user-authored checks, i.e., *explicit* contract assertions, added via [P2900R14] and to hook those new implicit contract assertions into the same contract-checking and violation-handling machinery used by the explicit ones. We propose to do exactly this in Section 3. By integrating with the same contract-violation handling facility, we vastly increase the ability to deploy, to production systems, software that is hardened against entire categories of potential bugs.

In addition to introducing implicit contract assertions, which let us *diagnose* the UB, we can actually *remove* the UB for those 17 cases of runtime UB where meaningful, well-defined fallback behaviour exists (see Section 2.6). This removal can be accomplished by defining that the behaviour of the operation in question is fallback behaviour; we propose to do this in Section 3.4.

However, practically all this fallback behaviour 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. We propose such an escape hatch in Section 3.5.

For the 61 cases of runtime UB in which meaningful, well-defined fallback behaviour does not exist (and therefore, continuation after an error has occurred is not possible), only two known ways can, in principle, give all those cases defined behaviour at run time.

I. *Diagnose* them (and pay all the overhead for the associated runtime checks, including the required instrumentation) and then terminate the program.

II. Make the entire construct that could potentially exhibit the given case of UB ill-formed, and provide its functionality via a different, "safer" language feature.

The fundamental dilemma is that for many cases, neither alternative is acceptable. The instrumentation required to diagnose Bounds and Type and Lifetime UB in the general case already exists, but its overhead is prohibitively large for most production scenarios. On the other hand, replacement by "safer" alternative features — such as replacing pointers and references with borrow checking, as proposed in [P3390R0] — is viable for newly written code but fundamentally incompatible with legacy code because it would make vast swathes of existing, *correct* C++ ill-formed.

Such subsetting of the language is exactly where we see the role of Profiles. Enabling a particular profile would make the associated set of "unsafe" legacy features ill-formed. By leveraging Profiles, we can explicitly distinguish between newly written, "safe" parts of the code and legacy, "unsafe" parts, which is similar to how this is done in languages like Rust but offers us much greater granularity.

On the other hand, we do not believe that Profiles should intersect with runtime behaviour, as proposed by [P3081R1]. Enabling or disabling a profile should never change the runtime semantics of a C++ program for all the same reasons [P2900R14] strove to achieve its prime directive in its design: if the only way to have safer code is to change your code's behaviour entirely, you fail at improving the safety of software that cannot afford the cost of enabling the checks.

At most, a profile could reject the program if a certain runtime check required by the chosen "safety" level is not available (making Profiles a useful auditing tool). A profile should never dictate whether a runtime check is enabled or disabled or what should happen if that check fails because, as we will see in Section 3 of this paper, all the required machinery for configuring runtime checks is already provided — more cohesively and flexibly — by the Contracts framework.

3 Proposed design

3.1 Defining implicit contract assertions

In this section, we propose a framework for systematically introducing runtime checks that guard against core language UB to C++. This framework builds on top of Contracts.

For C++26, we adopted an initial subset of Contracts functionality via [P2900R14]. This initial subset 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 also 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, 8, 13, 21, 34, 55 };
  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 undefined behaviour, 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, and 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.

3.2 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`, as 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.[4]

No other changes to the library API for contract-violation handling are necessary. In particular, unlike the previous revision of this paper and unlike [P3081R1], which adopted its library API from that earlier revision, 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 4.1).

Further, we do not propose any 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 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 them to repeat these things in multiple places is not good design. If the user wishes to use a different handler for implicit contract assertions, they can always branch on the `assertion_kind` in the global contract-violation handler and dispatch to a custom handler from there.

3.3 Applying implicit contract assertions

Now that we have a framework in place for specifying what an implicit contract assertion is and how it behaves, we can apply that specification to *every* case of UB that is — at least in principle —

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

runtime checkable, i.e., per Section 2.7, 79 cases of UB, which is the vast majority of core language UB in C++ today.

The required transformation 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`; continuing execution past a violation of this precondition is undefined behaviour".

Note that we do not specify any restrictions on the evaluation semantics of any of these 79 newly introduced implicit contract assertions. Since the choice of evaluation semantic is implementation-defined, every implementation can choose for themselves which evaluation semantics to offer for which one and which should be the default semantic. One possible implementation choice is to simply make all 79 cases always have the *ignore* semantic, which makes all existing implementations of C++ already conforming with our proposal. Another possible choice is to say that *ignore* is the default, but other semantics are available. Yet another possible choice is to enable certain checks by default. All those choices are conforming with our proposal.

Since the choice of evaluation semantic is implementation-defined, implementations are further expected to document which semantics they support for which implicit contract assertions and which selection mechanism they offer. Once we have Labels (see Section 4) for each case of UB guarded by an implicit contract assertion, implementations and users can refer to each case by name, giving us a shared, portable, universally agreed standard framework with terminology for reasoning about runtime UB.

Many possible choices for the evaluation semantics of implicit contract assertions 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 the benefit that they can now opt into using the unified standard framework.

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, and a matching profile could render nonconforming programs ill-formed. 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 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: the fact that *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 thus been rejected by WG21. 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.

3.4 Specifying the fallback behaviour

The next part of our proposal is to introduce defined fallback behaviour for all 17 cases of core language UB for which such fallback behaviour exists (see Section 2.6). We accomplish this by modifying the specification of each affected operation such that, if the condition occurs that would have previously made the behaviour of the operation undefined, the behaviour is instead the defined fallback behaviour.

The required transformation 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 *<fallback behaviour>*".

As discussed in Section 2.6, if we make this change and do nothing further, it would introduce significant — and in many cases, unacceptable — performance regressions to existing code. Therefore, we must offer an escape hatch that reverts to today's semantics for cases in which a violation of the implicit precondition leads to undefined behaviour.

3.5 Providing an escape hatch

For indeterminate values, [P2795R5] introduced a specific escape hatch: the `[[indeterminate]]` attribute. However, in many cases, such a specific, 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", but in some cases, enabling the fallback behaviour by default will be infeasible or impractical due to the associated runtime overhead.

Considering all the above reasoning reveals that such a generic, nonsyntactic escape hatch to revert to today's semantics — i.e., a violation of the implicit precondition leads to undefined behaviour — 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 the safe fallback 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 4.2) will provide the ability to choose and constrain the evaluation semantic in code with arbitrary granularity.

Importantly, in light of the sustained opposition in WG21 to allowing the *assume* semantic for explicit contract assertions,[5] we propose that the *assume* semantic is allowed for only *implicit* contract assertions. *Explicit* contract assertions (`pre`, `post`, and `contract_assert`) may *not* be evaluated with the *assume* semantic.

This restriction is important because, for explicit contract assertions, the *assume* semantic has the potential to introduce undefined behaviour to an otherwise correct program if we wrote a buggy contract predicate. On the other hand, this risk does not exist for implicit contract assertions since they are generated by the compiler; for error cases that cause UB, the *assume* semantic is merely a tool to achieve the same semantics those error cases already have in C++ today.

Once we get Labels, as proposed in [P3400R1], we can introduce an explicit label that would allow the *assume* semantic to apply to an explicit contract assertion as well. For example, the limiter example from the `[[assume]]` paper, [P1774R8], could be written as follows:

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

To ensure language safety, the *assume* semantic would be allowed only when the `may_be_assumed` label is present; further, a "safe C++" profile would make such a label ill-formed. Thus, contract assertions without the label would be no less "safe" than they are in C++26. Such a label would be a vast improvement over `[[assume]]` since it would allow for *checkable* assumptions (see [P2064R0] for context). At that stage, we will have achieved the integration between assertions and assumptions that we failed to achieve in the C++20 cycle, and the `[[assume]]` attribute — a temporary solution that was introduced as a reaction to that failure — can be deprecated.

[5]This opposition is the reason why no such semantic was included in [P2900R14]. 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.

4 Future extensions

We already briefly touched upon Labels in the previous section. In this section, we explore other extensions that rely on Labels as proposed in [P3400R1] and provide important additional functionality for implicit contract assertions that is not proposed in this paper.

4.1 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 3.1) 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.

4.2 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”, and so on, can be applied to any group of implicit contract assertions at any granularity — per file, per namespace, per function, per block, and so on.

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, such as an “audit” label to identify expensive checks:

```
int f(int a, int b)
{
    // Add the audit label to all implicit arithmetic preconditions in this scope.
    contract_assert implicit arithmetic |= audit;

    return a + b;   // overflow checked if audit checks are enabled
}
```

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.

5 Proposed wording

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

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 has a value other than `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 [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 runtime-checkable UB *with* fallback 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 runtime-checkable UB *without* fallback 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 runtime-checkable UB listed in Appendix A can be provided in a future revision of this paper.

Appendix A: List of language UB

All wording 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; within each category, rows are ordered in the same order in which the corresponding wording appears in [N5008].

I. Initialization

Identifier	Wording	Runtime checkable	Locally checkable	Checking strategy	Fallback behaviour
{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

Identifier	Wording	Runtime checkable	Locally checkable	Checking strategy	Fallback behaviour
{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

{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

{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	Fallback 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

{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
{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 allocate1 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 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 object of correct type within its lifetime	None
{lifetime. outside. pointer. convert}	[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 object of correct type within its lifetime	None
{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 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 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 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 object of correct type within its lifetime	None
{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 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 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 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 `const` object	None
{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 / 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 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 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 / freed	None
{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 object of correct type within its lifetime	Coerce invalid value representations 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 object of correct type within its lifetime or object currently being constructed/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 object of correct type within its lifetime	None
{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 object of correct type within its lifetime or object currently being constructed/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 object of correct type within its lifetime or object currently being constructed/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 object of correct type within its lifetime	None
{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 object of correct type within its lifetime or object currently being constructed/destroyed; insert 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 object of correct type within its lifetime; track whether address is associated with a function; insert 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
{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 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 object of correct type	None
{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 types of all functions based on address	None

{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 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. not.class. type}	[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.	This should be a non-normative note	—	—	—
{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 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
{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/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/destroyed	None

{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/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/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/destroyed	None
{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/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/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/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/destroyed	None

IV. Arithmetic

Identifier	Wording	Runtime checkable	Locally checkable	Checking strategy	Fallback behaviour
{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 check whether 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 check whether 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 check whether value is valid	Coerce into erroneous value
{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 check whether 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 check whether 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 check whether value is valid	Coerce into erroneous value
{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 check whether 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 check whether 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

Identifier	Wording	Runtime checkable	Locally checkable	Checking strategy	Fallback behaviour
{intro.races. data}	[intro.races]/17: Any such data race results in undefined behavior.	Yes	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

VI. Sequencing

Identifier	Wording	Runtime checkable	Locally checkable	Checking strategy	Fallback behaviour
{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

VII. Assumptions

Identifier	Wording	Runtime checkable	Locally checkable	Checking strategy	Fallback behaviour
{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 automatic checking strategy possible because predicate cannot be in general proven to be side-effect free; instead, the user has to change `[[assume(x)]]` to `contract_assert<may_be_assumed>(x)` and select appropriate evaluation semantic	Ignore the assumption

VIII. Control Flow

Identifier	Wording	Runtime checkable	Locally checkable	Checking strategy	Fallback behaviour
{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 lifetime of static objects	None

{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. 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*	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 recursion counter into 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

IX. Replacement Functions

Identifier	Wording	Runtime checkable	Locally checkable	Checking strategy	Fallback behaviour

{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.	Address this via [P3424R0] instead	—	—	—
{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

X. Coroutines

Identifier	Wording	Runtime checkable	Locally checkable	Checking strategy	Fallback behaviour

{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 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 suspension state associated with every coroutine handle	None

XI. Templates

Identifier	Wording	Runtime checkable	Locally checkable	Checking strategy	Fallback behaviour
{temp. inst.inf. recursion}	[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.	No, make ill-formed instead	—	—	—

XII. Preprocessor

Identifier	Wording	Runtime checkable	Locally checkable	Checking strategy	Fallback behaviour
{cpp.cond. defined}	[cpp.cond]/11: If the preprocessing token `defined` is generated as a result of this replacement process or use of the `defined` unary operator does not match one of the two specified forms prior to macro replacement, the behavior is undefined.	No, make IFNDR instead	—	—	—

{cpp. include. one.of.two. forms}	[cpp.include]/4: The preprocessing tokens after `include` in the directive are processed just as in normal text ([...]). If the directive resulting after all replacements does not match one of the two previous forms, the behavior is undefined.	No, make IFNDR instead	—	—	—
{cpp. replace. macro. pptoken}	[cpp.replace.general]/13: If there are sequences of preprocessing tokens within the list of arguments that would otherwise act as preprocessing directives, the behavior is undefined.	No, make IFNDR instead	—	—	—
{cpp. stringize. invalid. char}	[cpp.stringize]/2: If the replacement that results is not a valid character string literal, the behavior is undefined.	No, make IFNDR instead	—	—	—
{cpp. concat. invalid. preprocessing. token}	[cpp.concat]/3: If the result is not a valid preprocessing token, the behavior is undefined.	No, make IFNDR instead	—	—	—
{cpp.line. zero.or. overflow}	[cpp.line]/3: If the digit sequence specifies zero or a number greater than 2147483647, the behavior is undefined.	No, make IFNDR instead	—	—	—
{cpp.line. pptoken. not.match}	[cpp.line]/5: If the directive resulting after all replacements does not match one of the two previous forms, the behavior is undefined; otherwise, the result is processed as appropriate.	No, make IFNDR instead	—	—	—

{cpp. predefined. define. undef}	[cpp.predefined]/4: If any of the predefined macro names in this subclause, or the identifier `defined`, is the subject of a `#define` or a `#undef` preprocessing directive, the behavior is undefined.	No, make IFNDR instead	—	—	—

Document history

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

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

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

Acknowledgements

Thanks to Herb Sutter, Oliver Rosten, Andrzej Krzemieński, and Roger Orr for their helpful feedback on a previous revision of this paper.

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

Bibliography

[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.

[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.

[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.

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

[P2843R2] Alisdair Meredith. Preprocessing is never undefined. https://wg21.link/p2843r2, 2025-03-15.

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

[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.

[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.

[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.

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

[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.

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

[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.

[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.