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

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

Document #: P3100R9
Date: 2026-10-23
Project: Programming Language C++
Audience: EWG, LEWG

Abstract

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

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

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

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

Contents

1 Introduction . . . 4
2 History and polls . . . 4
3 Analysis . . . 6
3.1 Enumeration . . . 6
3.1.1 Methodology . . . 6
3.1.2 Granularity . . . 7
3.1.3 Scope . . . 7
3.2 Classification . . . 8
3.2.1 Categories . . . 8
3.2.2 Relevance for security . . . 10
3.3 Diagnosability . . . 10
3.3.1 Static vs. dynamic diagnosis . . . 11
3.3.2 Locality of diagnosis . . . 11
3.3.3 Cost of local diagnosis . . . 13
3.3.4 Cost of non-local diagnosis . . . 14
3.4 Existence of replacement behaviour . . . 15
4 Strategy . . . 18
4.1 Overview . . . 18
4.2 Scope . . . 18
4.3 Tools . . . 19
4.3.1 Feature removal . . . 19
4.3.2 Refined behaviour . . . 20
4.3.3 Erroneous behaviour . . . 20
4.3.4 Runtime checks . . . 21
4.3.5 Language subsetting . . . 21
4.3.6 Annotations . . . 21
4.3.7 New features . . . 22
4.4 Configuration . . . 22
5 Proposed design . . . 23
5.1 Defining implicit contract assertions . . . 23
5.2 Applying implicit contract assertions . . . 24
5.3 Defining replacement behaviour . . . 26
5.4 Providing an escape hatch . . . 26
5.5 Interaction with `noexcept` . . . 28
5.6 Extending the library API . . . 30
6 Proposed wording . . . 30
7 Future extensions . . . 94
7.1 Identifying the UB category . . . 94
7.2 Granular control of the evaluation semantic . . . 94
7.3 Integrating assertions and assumptions . . . 95

Acknowledgements . . . 96
References . . . 96
Appendix A: List of language UB . . . 100

Revision history

— **R9**, 2026-10-23: Added a library feature-test macro, rebased on latest draft and [P4284R1], added {conv.array.unsimilar.type}, wording changed to introduce implicit assertions before undefined behavior instead of altering the definition of undefined behavior.

— **R8**, 2026-08-14: Wording formatting improvements, rebased wording on latest draft and [P4284R0], updated discussion of interaction with `noexcept` (removed {basic.stc.alloc.dealloc.throw}, added {basic.compound.pointer.before.storage.duration} and {expr.reinterpret.cast.invalid.pointer.value}).

— **R7**, 2026-06-01: Wording formatting improvements; response to early wording review; clarified that implicit contract assertions have conditions and do not necessarily have predicates.

— **R6**, 2026-05-12: Completed wording; rebased on [P3596R2]; simplified diagnosability criteria; added history and polls section.

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

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

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

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

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

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

1 Introduction

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

While reducing the amount of UB on a case-by-case basis is useful to make the C++ programming language safer and more secure, what has been missing is a *holistic* strategy for systematically detecting, mitigating, and ultimately removing UB across the *entire* C++ programming language specification. In this paper, we present such a holistic strategy.

The remainder of this paper is structured as follows.

In Section 2, we summarise the history of this proposal and the polls taken by the various WG21 subgroups that reviewed it.

In Section 3, we identify and enumerate all core language UB explicitly specified in the current C++ working paper [N5054]. We group all core language UB into ten categories. We then classify cases of UB along several relevant criteria, such as whether they are checkable at compile time or at run time, whether a check can be performed locally, how expensive a check would be, and in which cases the UB can be replaced with meaningful, well-defined behaviour.

In Section 4, we present our holistic strategy for addressing UB across the entire C++ language. The proposed strategy is composed of seven basic tools: feature removal, refined behaviour, erroneous behaviour, insertion of runtime checks, language subsetting, the introduction of annotations, and the introduction of entirely new language features. We use the results of the analysis in Section 3 to identify which tools are applicable to which cases of UB. We find that the conditions under which UB will occur can, in many cases, be identified by a runtime check. In addition, there are a significant number of cases where UB can be replaced by well-defined erroneous behaviour following a failed check. These two techniques are applicable to a wide range of existing cases of UB and, importantly, do not require any source changes.

In Section 5, we present a generic framework for applying these two techniques across the entire C++ language. The proposed design has been reviewed and approved by SG21, SG23, and EWG. We describe how runtime checks can be systematically introduced via *implicit contract assertions*, building on the basic framework of Contracts adopted for C++26 via [P2900R14] and giving users complete control over what impact that undefined behaviour has on their programs. In addition to runtime checking, we replace UB by erroneous but well-defined behaviour that allows the program to continue execution past a violated implicit contract assertion wherever possible. We also propose an escape hatch to mitigate the runtime cost of such well-defined replacement behaviour and avoid performance regressions in existing C++ programs.

In Section 6, we propose wording that implements the framework presented in Section 5.

Finally, in Section 7, we discuss how future extensions, such as Labels [P3400R5], will enable programmatically identifying the category of UB that has occurred and provide us with granular, in-source control of the evaluation semantics for implicit contract assertions.

2 History and polls

The initial version R0 of this proposal, containing the basic ideas in this paper, was written up in early 2023, as a product of our work on designing a contract-assertion facility for C++ and understanding the many use cases of such a facility ([P1995R1]). During the WG21 meeting in June 2024 in St. Louis, these ideas were significantly refined, essentially taking on the shape they have

today, and published in revision R1. This revision was reviewed by SG21 at the WG21 meeting in November 2024 in Wrocław. SG21 unanimously supported the proposed direction:

SG21 Poll 6, Wrocław, 2024-11-22

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

At the WG21 February 2025 meeting in Hagenberg, EWG agreed on pursuing a systematic treatment of core language UB in C++. The ship vehicle for this effort was originally supposed to be a *core language UB white paper*, to be released in the C++26 timeframe, and covering erroneous behaviour (EB), Profiles, and Contracts:

EWG Poll, Hagenberg, 2025-02

Pursue a language safety white paper in the C++26 timeframe containing systematic treatment of core language Undefined Behavior in C++, covering Erroneous Behavior, Profiles, and Contracts. Appoint Herb and Gašper as editors.

SF	F	N	A	SA
32	31	6	4	4

Result: Consensus

The proposed process was to start with an empty white paper working draft and then iteratively get EWG approval for papers to be adopted into that working draft. This white paper working draft document, describing the proposed process, was published as [P3656R1].

We immediately realised that the present paper directly addresses the major work items proposed by [P3656R1], and prepared revision R2, now explicitly targeting the UB white paper, for EWG review.

At the WG21 meeting in June 2025 in Sofia, EWG reviewed R2 of this paper and approved its adoption into the core language UB white paper working draft:

EWG Poll, Sofia, 2025-06-20

EWG encourages more work on P3100R2 and wants a step-by-step systematic review of P3100R2 to do per-clause approval for inclusion in the core language UB whitepaper (in telecons)

SF	F	N	A	SA
15	27	2	0	0

Result: Consensus

Furthermore, in the same session we gave a presentation to EWG, published in [P3754R0], outlining the wider, holistic strategy to systematically address core language UB in the C++ language that underpins this proposal. EWG reached consensus to use the diagram on slide 53 from that presentation (a.k.a. the "magic slide"), visualising our strategy, as a basis for the core language UB white paper:

EWG Poll, Sofia, 2025-06-20

EWG agrees that Timur's (magic) slide 53 in P3754R0 is a good basis for the core language UB whitepaper, and asks that the Whitepaper editors make it so.

SF	F	N	A	SA
14	28	2	1	0

Result: Consensus

An updated version of the diagram and the strategy description is available in Section 4 of this paper.

However, the telecons for per-clause approval of this paper into the white paper were never scheduled, and the pursuit of the white paper as a ship vehicle has stalled. To make progress, we decided to instead target C++29 with the present proposal. While EWG was busy finishing C++26, SG23 (Safety & Security) reviewed revision R4, now targeting C++29, and approved it with strong consensus:

SG23 Poll, Kona, 2025-11-07

SG23 supports the direction of P3100R4 and recommends its inclusion in C++29

SF	F	N	A	SA
15	12	1	1	2

Result: Consensus

Finally, at the WG21 meeting in March 2026 in Croydon, EWG reviewed revision R5 of this paper and encouraged this direction. We were told to come back to EWG with full wording:

EWG Poll, Croydon, 2026-03-26

Update P3100R5 by applying the presented rules to all cases of runtime-checkable UB in the standard, as listed in appendix A, and bring it back to EWG for case-by-case wording review

SF	F	N	A	SA
39	19	5	2	1

Result: Strong consensus

The full wording requested by EWG is available in the present revision.

3 Analysis

3.1 Enumeration

3.1.1 Methodology

We manually inspected all occurrences of the words "undefined" and "assume" in revision [N5054] of the C++ working paper. We then constructed a list of all cases of explicitly specified core language UB. We found 81 instances of explicit language UB introduced with phrases containing the word "undefined" ("...the behavior is undefined", "...has undefined behavior", "...results in undefined behavior", etc.) and one instance of explicit language UB introduced with a phrase containing the

word "assume" ("the implementation may assume...").[1] We thus obtained a list with 82 cases of explicit language UB. [P4284R1] subsequently identified one further case, {conv.array.unsimilar.type}, which we have added, giving 83 cases; the full list can be found in Appendix A of this paper.

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

Each individual case of UB in our list has a stable identifier. Separately from this paper, [P3596R3], which was adopted in Brno, added a UB and IFNDR annexes to the C++ Standard. Various fixes and updates to that annex have been proposed in [P4284R1]. The enumeration of core language UB in that annex, and the stable identifiers used therein, are in sync with Appendix A of this paper. However, in this paper, we place those identifiers between {curly braces} to visually distinguish them from the C++ working paper's clause identifiers, which we place between [square brackets].

3.1.2 Granularity

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

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

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

As another example, instead of considering flowing off the end of a function ([stmt.return]/4) as a single case of UB, we could introduce separate cases for flowing off the end of an assignment operator (which has a proposed mitigation, see [P2973R0]), flowing off the end of a function that returns a built-in type (which could be mitigated via returning an erroneous value, see Section 3.4), and flowing off the end of any other non-`main` function that returns a non-`void` type.

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

3.1.3 Scope

In this paper, we consider only explicit UB, not implicit UB — that is, UB that exists by omission because the C++ working paper failed to specify the behaviour of a well-formed operation. We consider all cases of implicit UB that may be discovered in the future to be wording bugs that should

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

be addressed via Core issues, and then subsequently become explicit UB that can be addressed via the framework proposed here. Importantly, while explicit UB can be enumerated exhaustively, and we do so in Appendix A of this paper, enumerating all implicit UB is impossible in principle as we can never be sure we found all wording bugs in the C++ working paper.

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

Further, in this paper we consider only *runtime* UB, not compile-time or link-time issues. We therefore exclude all cases of IFNDR listed in [P3596R3] from our analysis. Although the effects of IFNDR can manifest at run time, whether a program is IFNDR is (unlike UB) fundamentally not a runtime property. We also exclude the case of infinite recursion during template instantiation, listed as {temp.inst.inf.recursion} in [P3596R3]. The current wording[3] for this case can be interpreted to mean that such infinite recursion could cause runtime UB; however, this interpretation makes little sense as failure during template instantiation is fundamentally a compile-time issue. We therefore consider this case a wording bug rather than an instance of UB; this issue is currently being addressed by Core issue [CWG3034].

Finally, we exclude *library undefined behaviour*. The natural mitigation approach for library UB is to make use of contract assertions (`pre`, `post`, and `contract_assert`) in library implementations and, where sensible, mandate such assertions through library hardening [P3471R4], both of which are out of scope for this paper. We, therefore, consider only UB that is specified in the core language part of the C++ working paper (Clauses 1–15). Further, we found one case of UB that is specified in the core language part of [N5054], listed as {basic.start.term.signal.handler} in [P3596R3], but actually represents a precondition on Standard Library functions;[4] that case is, therefore, also excluded from our list.

3.2 Classification

3.2.1 Categories

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

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

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

III. **Type and Lifetime** — 53 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

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

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

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

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

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

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

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

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

IX. **Replacement Functions** — 2 cases. Executing a user-defined replacement function (`operator new/delete`) that fails to meet the specified requirements. For example, returning `nullptr` from a user-defined placement `new`.

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

Figure 1 shows the distribution of the 83 identified cases of explicit core language UB across these ten basic categories.

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

Because unambiguously categorising a particular case of UB into either *type safety* or *lifetime safety* is often impossible since it concerns both, we grouped them into a single combined category, Type

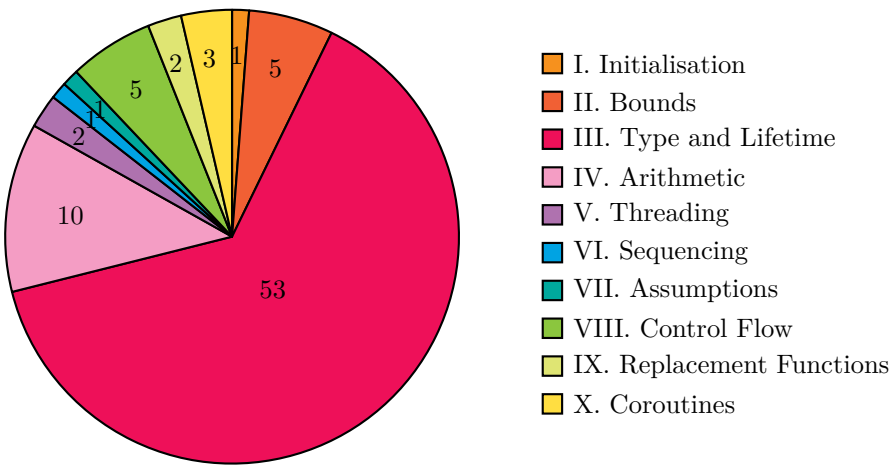

Figure 1: Distribution of identified cases of explicit language UB across specified categories

and Lifetime. While some cases of UB are primarily caused by type aliasing and others are primarily caused by out-of-lifetime accesses, they form a spectrum, and many common operations in C++ (e.g., using a reference) rely on *both* type and lifetime constraints to be satisfied.

Remarkably, these three categories related to memory safety account for 59 cases of UB, or 71.1% of all identified cases; the Type and Lifetime category alone accounts for 53 cases of UB, or 63.9% of all identified cases.

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

The following category, Sequencing, contains just one case of UB: unsequenced operations, such as `i++ + ++i`. Classifying UB due to data races and unsequenced operations into two separate categories might seem surprising at first since they have a very similar shape (except that one is inter-thread and the other is intra-thread), but as we will see in Section 4, these two categories actually require very different approaches to mitigation.

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

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

3.2.2 Relevance for security

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

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

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

3.3 Diagnosability

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

3.3.1 Static vs. dynamic diagnosis

A commonly asked question is "why does the committee not simply make all the UB ill-formed instead"? The answer is that in order for that to happen, it would be necessary to determine statically — i.e., at compile time — whether a given operation in a C++ program would lead to UB when executed at run time. However, whether a C++ program will have UB when executed is fundamentally a *runtime* property, i.e., the answer depends on runtime values unknown at compile time (for example, the runtime value of a pointer or an integer). Therefore, in the vast majority of cases, such a compile-time determination cannot be made. In fact, in our entire list of 83 cases of UB, we cannot identify *any* cases that can unconditionally be diagnosed at compile time.

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

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

As we will see in Section 4, despite these fundamental limitations there are things we can (and should) do in the C++ Standard to enable static analysis to construct a proof in more cases, such as subsetting the language (4.3.5), providing replacement features (4.3.7), and adding annotations (4.3.6). However, for existing C++ programs, reliably detecting cases of UB without rejecting correct code will inevitably have to leverage *runtime* detection, i.e., the insertion of additional runtime checks when compiling the program. We therefore focus on such runtime detection for the remainder of this analysis.

3.3.2 Locality of diagnosis

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

Most cases of UB in the Initialisation, Bounds, and Type and Lifetime categories are, in the *general* case, *not* locally diagnosable — and this is an inherent limitation of the C++ object model, see [Baxter2024] — but require instrumentation applied across the entire program to be reliably diagnosed.

Of course, "not locally diagnosable in general" does not mean "*never* locally diagnosable": under special circumstances, such cases of UB can be locally diagnosable — for example, if an array bound happens to be statically known, if a pointer happens to be null and not otherwise invalid/dangling, or if additional information about control flow and object values can be inferred from local analysis — and existing static analysis tools diagnose such cases today. This set of special cases could be further enlarged in the future with the introduction of novel features such as lifetime annotations [P2771R1] or "ghost data" [Lippincott2025], which are the subject of ongoing research. We discuss some of these approaches in Section 4; here, we focus on the status quo in Standard C++.

In the Initialisation category, no cases of UB are generally locally diagnosable, as C++ does not enforce initialisation and uninitialised objects can be passed around arbitrarily. In the Bounds category, no cases of UB are generally locally diagnosable because array bounds are generally not known.

In the Type and Lifetime category, {expr.mptr.oper.member.func.null} is locally diagnosable because this case requires *only* a null pointer check. {basic.align.object.alignment} is locally diagnosable by checking the alignment of storage when creating an object at run time. {expr.assign.overlap} is locally diagnosable by checking the overlap of the two address ranges. (The ranges are known because the address and `sizeof` are known at run time for both the source and the destination object.) {class.abstract.pure.virtual} is locally diagnosable by adding a runtime check to the pure virtual function stub to which the base class vtable points. {conv.lval.valid.representation} is locally diagnosable by validating the bits in the value representation of the object against its type. None of the other cases of UB in the Type and Lifetime category are locally diagnosable in general. Two recently introduced cases, {basic.compound.pointer.before.storage.duration} and {expr.reinterpret.cast.invalid.pointer.value}, depend on identifying numeric pointer values that cannot represent any possible object in or out of their lifetime. Due to the angelic nature of the undefined behaviour being removed it is challenging or impossible to identify diagnostic strategies for these. Similarly, {intro.object.implicit.create} is not diagnosable in general, since diagnosing it would require predicting whether the implicitly created objects will subsequently be used correctly as a given type.

However, all 10 cases of UB in the Arithmetic category are locally diagnosable since they are all cases of an arithmetic operation producing a value that is somehow inappropriate (mathematically invalid, not representable in the target type, etc.) and that value can be inspected at run time.

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

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

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

In the Replacement Function category, {expr.new.non.allocating.null} is locally diagnosable because we can check locally that a function does not exit via an exception, or that it does not return null. However, the more complex set of constraints specified in {basic.stc.alloc.dealloc.constraint} is not fully diagnosable (locally or at all).

Finally, one case of UB in the Coroutine category, {stmt.return.coroutine.flow.off}, is locally diagnosable in a way analogous to {stmt.return.flow.off} by inserting a check at the end of a coroutine for which no `return_void` function is provided. The remaining two cases of UB in that category are not locally diagnosable since being so would require tracking runtime state information that is not currently maintained within the coroutine handle in most implementations.

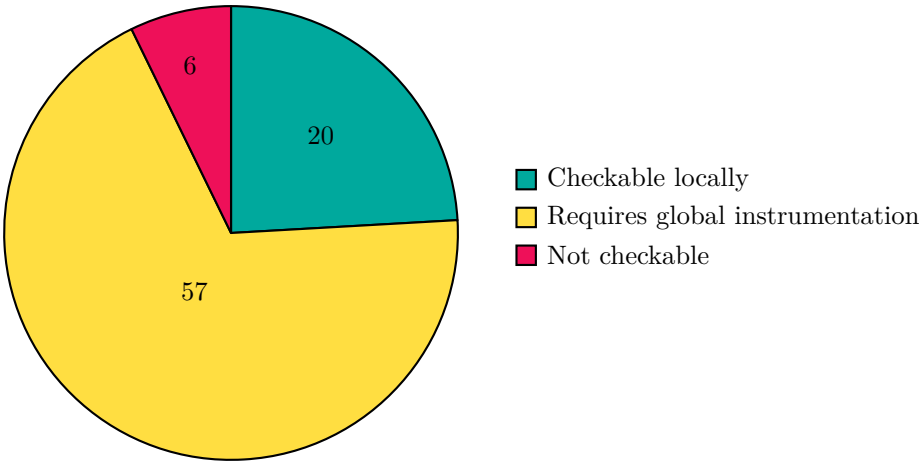

Figure 2: Runtime diagnosability of explicit core language UB for the general case

Overall, as shown in Figure 2, only 20 cases of UB (24.1% of all cases) are unconditionally locally diagnosable at run time, while 57 cases of UB (68.7%) are not locally diagnosable in the general case, requiring instrumentation of the code in order to be diagnosed at run time. We will discuss these requirements in more detail in Section 3.3.4. In addition, 6 cases of UB (7.2%) are not diagnosable at all in the general case, even with additional instrumentation.

3.3.3 Cost of local diagnosis

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

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

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

A slightly more expensive but still cheap and optimiser-friendly kind of check is a null check, required to diagnose the null pointer cases (the null pointer case of {expr.unary.dereference}, {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, i.e., a subset of {expr.add.out.of.bounds}, as well as {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, in varying degrees, to mostof the arithmetic UB category. The compiler would have to validate the bit patterns of values of arithmetic types according to knowledge it has about how values of such types are represented on the targeted platform. Such checks can be done locally, but they can slow operations involving built-in types and, in particular, floating-point types.

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

3.3.4 Cost of non-local diagnosis

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

To diagnose *all* cases of UB in the memory safety categories of Initialisation, 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 initialised
- For all addresses that point to functions, the type of the function

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

The non-locally-diagnosable UB in the Control Flow category concerns operations that are not allowed during construction and destruction of objects with static or thread-local storage duration ({stmt.dcl.local.static.init.recursive}, {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 being constructed or destroyed.

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

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

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

3.4 Existence of replacement behaviour

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

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

As we will discuss in more detail in Section 4, we can conceptually distinguish between two types of replacement behaviour — *refined* and *erroneous* behaviour — depending on whether the replacement behaviour is considered correct or incorrect (despite no longer being undefined). In any case, for either type of replacement behaviour to actually happen, the compiler must be able to lay down the necessary instructions at compile time. Simultaneously, as discussed in Section 3.3.1, in the vast majority of cases core language UB is fundamentally *not* diagnosable at compile time, as whether or not the UB will occur depends on runtime parameters. Replacement behaviour therefore cannot depend on knowing that an error occurred. For non-locally-diagnosable UB, replacement behaviour also cannot depend on any additional instrumentation being present.

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

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

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

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

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

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

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

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

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

For UB in the Threading category ({intro.races.data}), one could contemplate making all primitive memory accesses implicitly atomic, as in the Java memory model. However, such a model is neither specified nor ready to propose, so we do not count it as replacement behaviour. The overhead incurred by such a model would heavily depend on the memory model of the underlying hardware; on weakly-ordered platforms, such as ARM, it would be larger than on strongly-ordered platforms such as x86. Further, while such behaviour would be well-defined, it would still fail to prevent many

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

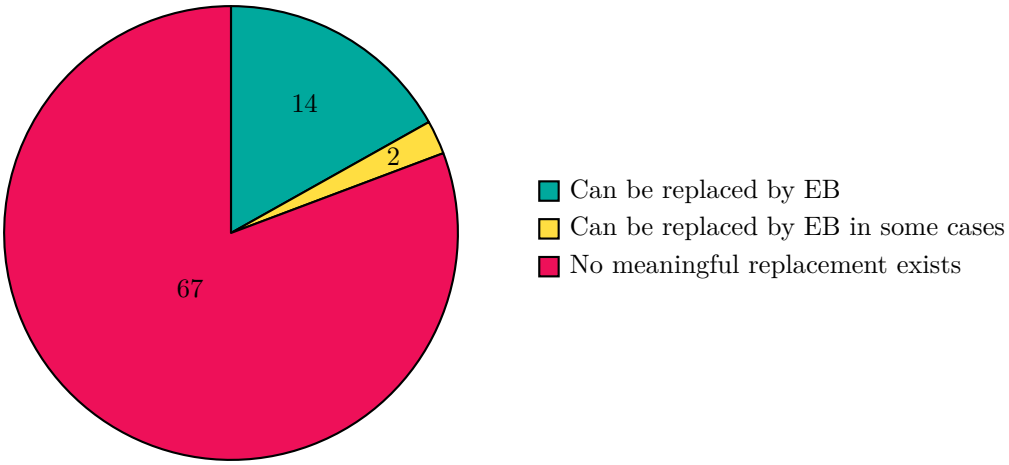

Figure 3: Existence of well-defined replacement behaviour for explicit core language UB

real bugs that result from incorrect application of concurrency since user-defined types with multiple members could still be easily observed with inconsistent ("torn") states if no proper synchronisation is performed.

The other case of UB in the Threading category, {intro.progress.stops}, has no meaningful replacement behaviour: since detecting whether a thread has stopped making progress is undecidable in general, the only option is to do nothing, i.e., leave the thread running as today, which does not replace the UB with anything.

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

The replacement behaviour for UB in the Assumption category ({dcl.attr.assume.false}) is trivial: just ignore the assumption, instead of optimising based on it. The performance overhead is limited to losing any optimisation opportunities from placing the assumption there. Of course, this mitigation makes the assumption itself completely useless. We will discuss this case in more detail in Section 5.4.

In the Control Flow category, we can define partial replacement behaviour for {stmt.return.flow.off}: as with automatic variables, we can initialise the storage for the return value with an erroneous value. This case is effectively handled in the same way as {basic.indet.value}; again, no plausible replacement behaviour exists for user-defined return types in the general case. We can also define replacement behaviour for {dcl.attr.noreturn.eventually.returns}: the function simply returns normally. This requires both the caller's and the callee's translation units to ignore, instead of assume, that the function does not return.

No analogous replacement behaviour exists for the corresponding case of UB in the Coroutines category ({stmt.return.coroutine.flow.off}): we cannot fabricate an argument of a type suitable to invoke one of possibly many overloads of `return_value` on the promise type.

Overall, as shown in Figure 3, we can define meaningful replacement behaviour for only 16 cases of UB (19.3% of all cases). Out of these 16 cases, for 2 cases this is only possible when the operation in question produces a value of built-in type. Unconditional replacement behaviour exists for only 14 cases of UB (16.9% of all cases). In all of these cases, the replacement behaviour is *erroneous* behaviour (and not refined, correct behaviour); in most cases, removing the UB in this manner introduces an unmeasured and possibly significant runtime cost.

4 Strategy

Having performed an in-depth analysis of all explicit core language UB in the C++ working paper in Section 3, we can use the results of this analysis to develop a holistic strategy for systematically detecting, mitigating, and ultimately removing UB across the entire C++ programming language specification. Our goal is for this strategy to guide the evolution of the C++ programming language.

The outline of this strategy is illustrated in Figure 4, an updated version of the diagram on slide 53 in [P3754R0] (a.k.a. the "magic slide") that we presented to EWG in Sofia, gaining strong approval from that group.

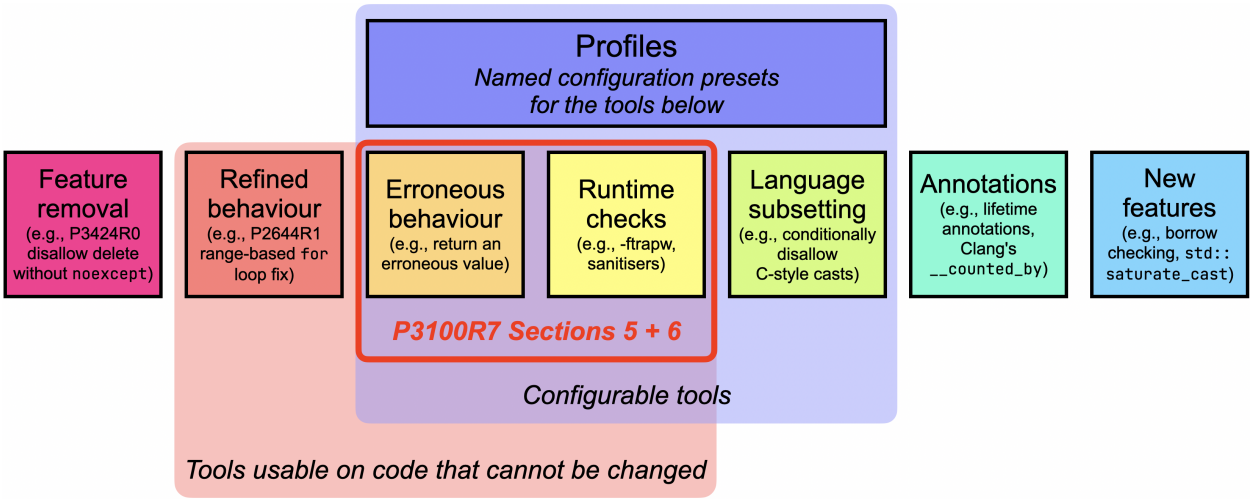

Figure 4: Overview of the proposed holistic strategy for removing UB from the C++ language: seven orthogonal tools plus Profiles as a higher-level feature specified on top of these tools. The red rectangle in the centre illustrates the scope of the proposal in Sections 5 and 6 of this paper.

4.1 Overview

The proposed strategy is composed of seven basic tools that are orthogonal to each other: feature removal, refined behaviour, erroneous behaviour, insertion of runtime checks, language subsetting, the introduction of annotations, and the introduction of entirely new language features. In Section 4.3, we describe each tool and discuss which cases of UB identified in this paper it can be applied to.

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

Further, in order to be usable effectively in practice, three of these tools (erroneous behaviour, runtime checks, and language subsetting) need to be configurable by the user, either via compiler options or directly in source. This design space is discussed in more detail in Section 4.4. *Profiles*, another proposed feature currently in development, could be used to group together particularly useful configuration presets across different tools in order to provide desirable sets of guarantees.

4.2 Scope

Our proposed strategy for removal of explicit core language UB focuses on tools that can be portably specified within the C++ abstract machine. We therefore do not consider, in this paper, efforts

that operate largely outside of the C++ abstract machine and the specification tools afforded by the C++ Standard. One noteworthy effort that falls into the latter category is [P3627R0], which proposes a profile for preventing remote code execution (RCE) by employing implementation-defined techniques such as stack isolation and address space layout randomisation (ASLR).

Further, as discussed in [P3700R0], making C++ "safe"[6] consists of more than mitigating explicit core language UB. We already touched upon implicit UB and library UB in Section 3.1.3. Beyond those, there are many other classes of bugs unrelated to UB that can compromise the functional safety, security, and correctness of a C++ program. Such bugs include resource leaks, termination errors, and logic errors.

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

4.3 Tools

4.3.1 Feature removal

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

However, we generally do not consider it acceptable to break an existing, *correct* program by using this tool (or any of the other proposed tools). If we want to strictly follow this principle, usage of the removal tool would have to be restricted to cases where we can determine at compile time that the given operation will definitely have UB when executed.[7] As discussed above in Section 3.3.1, there is not a single case of UB in our list where such a determination can generally be made.

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

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

We are currently not aware of any other areas in the C++ Standard where this tool could be successfully applied, but it is worth keeping it in our toolbox in case such areas will be discovered in

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

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

the future, and committing to the slow but effective process of deprecating prior to removal when the end result is meaningfully improved.

4.3.2 Refined behaviour

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

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

4.3.3 Erroneous behaviour

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

Unlike refined behaviour, it is acceptable to specify erroneous behaviour in a way that is relatively vague (e.g., return *some* erroneous value rather than return a specific value), as it represents an incorrect program that cannot be relied on. For the same reason, it makes sense to make erroneous behaviour configurable (see Section 4.4). Indeed, the definition of erroneous behaviour adopted for C++26 via [P2795R5] permits different possible behaviours.

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

In this paper, we go further and propose to introduce erroneous behaviour for almost all cases that we identified in Section 3.4 that some kind of plausible well-defined replacement behaviour exists — 12 of the 14 cases with unconditional replacement behaviour, plus the 2 cases for built-in types only. For the remaining two cases, unsequenced modification ({intro.execution.unsequenced.modification}) and a `[[noreturn]]` function that returns ({dcl.attr.noreturn.eventually.returns}), this revision of the wording leaves the behaviour undefined. In Sections 5 and 6 of this paper, we provide a concrete specification for how to perform the necessary replacements in the C++ working paper; the full list of proposed erroneous behaviours is provided in Appendix A.

4.3.4 Runtime checks

As we saw in Section 3.3, the vast majority of UB (77 cases out of 83, or 92.8% of all cases) can in principle be diagnosed by inserting and performing a suitable runtime check (which may or may not require additional instrumentation). The check verifies the conditions necessary for the operation in question to have well-defined behaviour at run time. It also acts as a guard against UB: if the check fails, the program can be terminated, thus preventing the undefined behaviour from occurring.

While such runtime checks invariably add runtime overhead, they are very effective at both diagnosing bugs and removing security vulnerabilities. In Sections 5 and 6 of this paper, we propose a generic framework for systematically adding runtime checks to C++ core language constructs via implicit contract assertions.

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

In order to be widely deployable, runtime checks — whether they are user-authored assertions, compiler-generated checks guarding against UB, or any other form of correctness check that is redundant in a correct program and has non-negligible overhead — need to be configurable; this is discussed in more detail in Section 4.4.

4.3.5 Language subsetting

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

However, in such cases, we cannot make these constructs *unconditionally* ill-formed, as this would break a significant number of correct C++ programs deployed in the field. Therefore, we need to use a tool different from simple removal (Section 4.3.1). The required tool is called *language subsetting*: specifying named *subsets* of the C++ language that do not contain the "unsafe" features. This tool can be very effective for avoiding UB in codebases that can be modernised or are being newly written. However, because much of existing C++ code cannot be changed easily, subsetting needs to be opt-in. Thus, it is the third tool in our toolbox that needs to be configurable by the user (see Section 4.4).

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

4.3.6 Annotations

Some cases of UB can be mitigated by language-level annotations that provide additional information that can be propagated across the interface boundaries of a C++ program. Such information can

turn cases of UB that are *not* locally diagnosable (which is the majority of UB today, see Section 3.3) into cases that *are* locally diagnosable.

Examples of standard proposals in this area include the lifetime annotations proposed in [P2771R1] and the `[[not_invalidating]]` attribute proposed in [P3984R0]. An example of such annotations deployed in the field as non-standard vendor extensions is the `__counted_by` attribute introduced in Clang 18. This attribute allows the user to propagate information about array bounds to enable automatic out-of-bounds checks. Such attributes are a great example of how different tools that form the proposed strategy, while being usable independently from each other, can also work together to help each other cover even more cases of UB.

4.3.7 New features

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

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

4.4 Configuration

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

Runtime checks with non-negligible overhead, in order to be widely deployable, necessarily need to be configurable in several ways; this is discussed in much detail in [P2899R1] and references therein. As we saw in Section 3.3, the majority of UB in C++ today is not locally diagnosable and requires expensive sanitiser-like instrumentation to perform the checks. Even for those 20 cases of UB that are always locally diagnosable and do not require additional instrumentation to insert runtime checks, in most cases the checks themselves will have a significant — and in some cases, unacceptable — runtime overhead. Since correctness checks serve no purpose in a program known to be correct — or for use cases where correctness is not the overriding concern — there needs to be an option to turn the checks off to avoid the overhead. This option is offered by every assertion facility that we know of.

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

Similar reasons apply to erroneous behaviour: in practically all cases, its introduction comes with non-negligible — and in some cases, even very large — performance overhead. Therefore, to avoid unacceptable performance regressions in existing, correct C++ code, we *must* offer an escape hatch that reverts to today's "unsafe" semantics. One such escape hatch is the `[[indeterminate]]` attribute for uninitialised values. Such a syntactic escape hatch is not applicable for all cases; we propose a generic escape hatch for erroneous behaviour that does not require syntax in Section 5.4.

Finally, as already discussed in Section 4.3.5, language subsetting needs to be opt-in in order to avoid breaking existing, correct C++ programs.

For all these features, we need to clearly specify the available configuration options and the mechanisms available for the user to select these options at different levels of granularity. For erroneous behaviour and runtime checks, we accomplish this via leveraging contract evaluation semantics (see Section 5); in-source configuration of these evaluation semantics at arbitrary granularity can be achieved with Labels (see Section 7). For subsetting, the design of suitable configuration mechanisms are is yet well understood; we expect future papers to make progress in this area.

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

There is no consensus yet on how Profiles relate to and compose with other proposals such as the holistic strategy for removing UB from C++ proposed here. That said, most Profiles proposed so far consist of some combination of specifying subsets of the language (Section 4.3.5), defining replacement behaviour for UB (Section 4.3.3), and/or introducing runtime checks guarding against UB (Section 4.3.4). It therefore seems logical to define Profiles as a higher-level feature building on top of these three basic tools (see Figure 4). Given that these three features are configurable, a concrete profile could be defined as being a named configuration preset for these features.

For example, we may define a language subset that excludes pointer arithmetic, and a set of implicit runtime checks for array bounds checking; further, the user may define explicit contract assertions on their own functions and declare them as bounds checks. We can then define a "bounds" profile whose purpose it is to allow the user to opt into all three of these features at once. Such profiles can be tailored to particular problem areas of the language, such as type, bounds, or arithmetic safety profiles, or to particular regulatory requirements, such as a MISRA profile. If we pursue such a "multi-level" strategy, we must make it clear which feature is responsible for providing the user-facing configuration mechanism for which tool (see also Section 7).

5 Proposed design

In this section, we propose a framework that systematically introduces runtime checks and well-defined replacement behaviour — two of the tools that form the strategy presented in Section 4 — to the C++ Standard. Runtime checks guarding against core language UB are realised as *implicit contract assertions*, leveraging the foundation laid by Contracts as adopted for C++26; in addition, replacement behaviour is added instead of UB wherever meaningfully possible. Both features are controlled and configured via use of the contract evaluation semantics introduced by [P2900R14] as their central vocabulary.

5.1 Defining implicit contract assertions

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

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

The precondition assertion `pre (index < size())` can be evaluated with a checked evaluation semantic (*observe*, *enforce*, or *quick-enforce*), which allows the user to opt into defined behaviour — program termination and/or a call to a contract-violation handler — when their vector is accessed out of bounds. Further, the contract-violation handler can be replaced by the user, allowing them

to query information about the error and implement their own mitigation strategy. Alternatively, the user can opt out of the runtime check by choosing an unchecked evaluation semantic (*ignore*) if their use case requires it.

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

As an example, let us consider indexing into a plain array rather than a user-defined, vector-like class. Let us further assume for the purpose of this example that the size (here, 10) of this array is statically known:

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

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

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

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

5.2 Applying implicit contract assertions

Having specified precisely what an implicit contract assertion is and how it behaves, we can now apply that specification to *every* case of UB that is — at least in principle — checkable at run time. As we saw in Section 3, this is true for 77 cases, that is, 92.8% of all identified cases of explicit core language UB in C++.

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

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

Note that no implementation is actually *required* to implement these checks: a valid implementation choice is to make all 77 cases always have the *assume* semantic (see Section 5.4), which mimics exactly the behaviour the compilers already implement today. It follows that all existing implementations of C++ are already conforming with this wording transformation.

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

One benefit is that implementations of such runtime checks will be able to leverage a shared paradigm and shared terminology for reasoning about incorrect programs. In addition, once we have Labels (see Section 7), for each case of UB guarded by an implicit contract assertion, implementations and users can refer to that category using portable standard names. Another benefit is that they will be able to integrate with the same unified standard contract-violation handling facility, significantly increasing the ability to deploy software to production systems that is hardened against entire categories of potential bugs.

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

Further, coding guidelines can place restrictions on which evaluation semantics are permitted for which kinds of implicit contract assertions; our proposal provides the necessary standard terminology for this. For example, in a safety-critical context, a set of coding guidelines may prescribe that unchecked semantics may not be used for certain kinds of implicit contract assertions. Further, we could add a syntactic way to render configurations not conforming with this requirement ill-formed (see also Section 7.3). Thus, the usage of toolchains and compiler options that could lead to the program exhibiting a particular kind of UB could be prevented by construction. Of course, this option requires alternatives to exist that offer checked semantics for the associated implicit contract assertions with acceptable performance tradeoffs.

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

The approach suggested in those papers is to constrain explicit contract-assertion predicates to expressions that can be statically proven to have no UB. However, this approach does not seem to be specifiable, implementable, or usable in practice (see [P3376R0], [P3386R0], and [P3499R1]) and has repeatedly been rejected by SG21, SG23, and EWG. What *does* work is to specify a framework

for mitigating UB across the entire language, as proposed here. Once we have this framework, it will then automatically also apply to the evaluation of explicit contract assertions.

5.3 Defining replacement behaviour

The next part of our proposal is to introduce well-defined replacement behaviour for 14 of the 16 cases of core language UB for which such replacement behaviour exists (see Section 3.4; the remaining two are discussed below). We accomplish this by modifying the specification of each affected operation as follows. If a condition that would have previously made the behaviour of the operation undefined (i.e., an implicit contract violation) occurs, and control flow continues past that violation (because the associated check was evaluated with the *ignore* or *observe* semantic), then the behaviour of the operation is defined to be that replacement behaviour instead of UB. Consider the following example:

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

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

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

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

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

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

In all 14 cases for which we propose replacement behaviour, this replacement behaviour will be some form of erroneous behaviour. For assumptions, it consists of ignoring the assumption; for all remaining cases, it consists of producing an erroneous value (which is only possible for values of built-in type). We do not yet propose replacement behaviour for unsequenced operations (performing the operations in some unspecified order) or for a `[[noreturn]]` function that returns (returning normally); both remain undefined behaviour in the proposed wording.

5.4 Providing an escape hatch

As discussed in Section 3.4, if we apply the transformation described in the previous section and do nothing further, we introduce significant — and in many cases, unacceptable — performance regressions to existing code. Therefore, we must offer an escape hatch to users that reverts to today's semantics: a violation of the implicit precondition leads to runtime UB.

For erroneous behaviour that arises from reading an indeterminate value, [P2795R5] introduced a syntactic escape hatch specific to this case: the `[[indeterminate]]` attribute. However, in many cases, such a syntactic escape hatch is simply nonviable. Consider, for example, arbitrary arithmetic

expressions where some integer operations may overflow; where would we place a syntactic escape hatch for a certain arithmetic operation within that expression? Instead, we need a *generic* escape hatch that works for all cases and does not require syntax.

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

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

Just like the *ignore* semantic, the *assume* semantic is a *nonchecking* semantic; i.e., its predicate is not evaluated. Further, just like with the *ignore* semantic, if the predicate would evaluate 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 two's complement representation, is a conforming implementation of the *ignore* semantic, silently executing well-defined replacement behaviour.

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

Just like with all other evaluation semantics, the mechanism by which the *assume* semantic is selected is implementation-defined and will, in practice, be accomplished by vendor-provided compiler flags. In addition, Labels (see Section 7.2) will provide the ability to choose and constrain the evaluation semantic in code with arbitrary granularity.

Importantly, in light of the sustained opposition in EWG to allowing the *assume* semantic for *explicit* contract assertions (`pre`, `post`, and `contract_assert`), we propose that the *assume* semantic is allowed for only *implicit* contract assertions. Explicit contract assertions may *not* be evaluated with the *assume* semantic (see also Section 7.3).

This restriction is important because, for explicit contract assertions, the *assume* semantic has the potential to silently add UB to an otherwise correct program if used incorrectly. This risk does not exist for implicit contract assertions since they are generated by the compiler; in all of those cases,

the *assume* semantic is merely a backwards-compatibility tool to achieve the same semantics that those operations already have in C++ today.

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

5.5 Interaction with `noexcept`

Consider:

```
int f(int x) {
  return x + 1;
}
```

In C++ today, signed integer addition is a non-throwing operation: `noexcept(x + 1) == true`. At the same time, this operation can have undefined behaviour due to signed integer overflow (which occurs if `x` has the value `INT_MAX`). With the addition of implicit contract assertions, such signed integer overflow may call the contract-violation handler, which may throw an exception.

On the language level, this is arguably fine and perfectly backwards-compatible, because such throwing from non-throwing core-language expressions can occur only in evaluations for which today the behaviour would be undefined — and undefined behaviour is already allowed to throw an exception today, thus implicit contract assertions are already conforming.

However, in order for such exception propagation from implicit contract assertions to work correctly *in all cases*, the compiler has to treat all core-language expressions that can result in implicit contract-assertion violations as potentially-throwing for the purposes of generating the exception-handling metadata, which will require compiler changes that affect codegen. We should thus carefully evaluate whether throwing from a contract-violation handler triggered by an implicit contract-assertion violation is indeed desirable.

Changing the value returned by the `noexcept` operator for a core-language expression such as `x + 1` above would be a breaking change to existing code and thus cannot be seriously contemplated. It is easy to construct a program with well-defined behaviour whose effects depend on the value of such a `noexcept` operator, and certainly many pieces of generic code that apply the `noexcept` operator to such expressions already exist in the wild. We therefore conclude that the addition of implicit contract assertions must not affect the result of the `noexcept` operator. We are currently aware of four design options for implicit contract assertions that satisfy this constraint:

Option A — "you get what you get". If the evaluation of a core-language expression leads to a violation of an implicit contract assertion and that assertion is evaluated with the *observe* or *enforce* semantic, the contract-violation handler is called; if that handler exits via an exception, that exception propagates out of the enclosing core-language expression. At the same time, the `noexcept` operator still returns the same value for that core-language expression as before; that value may be `true`.

Option B — handler called from implicit assertions cannot throw. The evaluation of an implicit contract assertion is not allowed to exit via an exception; if the contract-violation handler is called and exits via an exception, `std::terminate` is called instead. We could separately consider contract-terminating instead of invoking `std::terminate`, but that is orthogonal to the more general discussion of whether we permit exceptions to escape at all.

Option C — new non-throwing contract-evaluation semantics. Introduce new contract-evaluation semantics *observe-noexcept* and *enforce-noexcept* which have the same runtime effect as *observe* and *enforce*, respectively, except that if a contract-violation handler is called under those semantics, it can never exit via an exception and `std::terminate` is called instead. These

new semantics can be selected by the user via an implementation-defined mechanism, just like the existing ones.

Option D — only non-throwing semantics for implicit assertions. Introduce new contract-evaluation semantics as in Option C, but with the additional limitation that implicit contract assertions can never be evaluated with a semantic that could result in an exception escaping the contract check, thus excluding the regular *observe* or *enforce* semantics and avoiding the possibility of throwing. This is essentially the same as Option B, except that the user may also be given the ability to use the new non-throwing semantics for explicit contract assertions.

Option A is arguably the most straightforward option. Like all the other options, it does not change the existing behaviour of the `noexcept` operator — and thus does not constitute a breaking change — but it changes its conceptual meaning: rather than meaning "evaluating this expression cannot throw", it now effectively means "evaluating this expression cannot throw unless there is a contract violation".

Such throwing from core-language expressions has existing practice today: sanitisers such as ASan and UBSan allow the user to provide their own callbacks (for example, `__asan_on_error`), and those callbacks can throw, and be caught in user code.

This works today, but not in all cases: the compiler is not aware of the now potentially-throwing nature of core-language expressions and may therefore not correctly generate the necessary exception-handling metadata. This in turn can lead to strange effects such as exceptions thrown from these sanitiser callbacks flying through `noexcept` boundaries under certain conditions, destructors not being called correctly during stack unwinding, etc. A conforming implementation of P3100 with Option A would have to prevent such effects.

Option B avoids any changes to the conceptual meaning of the `noexcept` operator or to the exception-handling mechanism and associated codegen in existing compilers. However, it also precludes unwinding the stack in response to an implicit contract violation, which has important use cases (see [P3318R0]). Note further that with Option A, a user in control of their contract-violation handler will be able to obtain the behaviour of Option B by branching in that handler on whether the violated assertion is implicit (see Section 5.6).

Options A and B were first proposed in [P3541R1]. That paper was discussed in SG21 and resulted in strong consensus in favour of Option A and against Option B because the latter precludes the stack-unwinding approach to handling implicit contract violations.

Options B and D both remove the ability to do such unwinding that previous consensus was in favour of. Option C, introducing the new semantics but not otherwise restricting the ability to use throwing violation handlers, provides the greatest amount of freedom while giving users local control over the costs and implications of that freedom.

Options C and D both depend on introducing new contract-evaluation semantics. These will obviously require further design work to decide on their exact forms, and an approach to that has been put forth in [P4298R0]. If we choose to pursue either of those options, working out those finer details will be delegated to that paper, upon which this paper will then depend.

If we choose Option A today, that can evolve freely to Option C if [P4298R0] is adopted. If we choose Option B today, that can evolve freely to Option D if [P4298R0] is adopted.

In this paper, we thus follow SG21 consensus and propose Option A. Based on upcoming EWG discussions, we will adapt future revisions of this paper to record the result of those discussions and update the wording if a different option has stronger consensus.

5.6 Extending the library API

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

Alternatively, we could define its numerical value to be `0` since that value is not yet taken; however, we prefer to avoid using `0` and thus to retain the ability to detect the case in which the enum has not been explicitly initialised with a valid value.[8]

For completeness of having the enumeration represent the potential evaluation semantics, we also propose adding `assume` to `evaluation_semantic`, with similar reasoning for the value the enumerator should have.

No other changes to the library API for contract-violation handling are necessary. In particular, unlike earlier revisions of this paper and unlike [P3081R2], which adopted its library API from those earlier revisions, we no longer propose to add new enumerators to the enumeration `detection_mode` to encode the category of error (Initialisation, Bounds, and so on); instead, this encoding can be accomplished more effectively and flexibly via Labels (see Section 7.1).

Further, we propose minimal and flexible changes to the specification of `comment()` and no changes to `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 the user to repeat these things in multiple places is poor design. A user who wishes to use a different handler for implicit contract assertions can always branch on the `assertion_kind` in the global contract-violation handler and dispatch to a custom handler from there.

6 Proposed wording

After the adoption of [P3596R3], and with the follow-ups proposed by [P4284R1], each instance of undefined behaviour in the wording is annotated with a link to the annex. Wherever these occur, we have included notes containing our description and summary of how we are handling each particular instance of undefined behaviour. Those notes are called out with markers in this text (like *, †, or ‡, or multiples when we have particularly dense commentary on a section) along with explanations that follow in larger purple boxes.

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

In general when there is no fallback behaviour being introduced there is also no actual wording change, as the implicit contract assertion that is needed is introduced in basic.contract.implicit. This extra wording context and the corresponding purple boxes are to facilitate wording review by both EWG and CWG so that we can verify that the desired changes to existing undefined behaviour have been properly captured.

The wording in this document has been based on the changes in [P4284R1], which include the latest updates to the UB and IFNDR annexes on top of [P3596R3]. A companion paper, [P4277R1], contains significant additional commentary on all of the wording changes, along with details on the implementation experience with introducing runtime checks for each of these undefined behaviours.

These wording changes are relative to the C++29 draft sources with git hash f52f243f, last modified on Tue, 20 Oct 2026 21:10:04 -0400.

3 **Terms and definitions** [intro.defs]

3.66 **undefined behavior** [defns.undefined]

Modify definition 3.66[defns.undefined]:

> 3.66 [defns.undefined]
>
> **undefined behavior**
> behavior for which this document imposes no requirements
>
> [*Note to entry 1*: Undefined behavior may be expected when this document omits any explicit definition of behavior or when a program uses an incorrect construct or invalid data. Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a *diagnostic message* (defns.diagnostic)), to terminating a translation or execution (with the issuance of a diagnostic message). Many incorrect program constructs do not engender undefined behavior; they are required to be diagnosed. Evaluation of a constant expression (7.7.2[expr.const.core]) never exhibits behavior explicitly specified as undefined in Clause 4[intro] through Clause 15[cpp]. — *end note*]
>
> [*Note to entry 1+a*: There is a guarding contract assertion (6.11.2+a[basic.contract.implicit]) sequenced before any occurrence of undefined behavior. — *end note*]

6 **Basics** [basic]

6.8 **Memory and objects** [basic.memobj]

6.8.2 **Object model** [intro.object]

Notes regarding 6.8.2[intro.object], paragraphs 13-14:

> 13 Some operations are described as *implicitly creating objects* within a specified region of storage. 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 (6.9.1[term.implicit.lifetime.type]) 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 (F.2.1[ubx:intro.object.implicit.create]),* the behavior of the program is undefined. If multiple such sets of objects would give the program defined behavior, it is unspecified which such set of objects is created.
>
> [*Note*: Such operations do not start the lifetimes of subobjects of such objects that are not themselves of implicit-lifetime types. — *end note*]

Note

* **ubdef:**	intro.object.implicit.create
summary:	using storage in a way no choice of implicitly-created object could make well-defined.
runtime-checkable:	Not generally, requires predicting whether the object will be used correctly as a given type
locally-checkable:	No
checking:	None
replacement:	None
implementations:	None

14 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 (F.2.2[ubx:intro.object.implicit.pointer]).*If multiple such pointer values would give the program defined behavior, it is unspecified which such pointer value is produced.

Note

* **ubdef:**	intro.object.implicit.pointer
summary:	producing a pointer to an implicitly-created object when no pointer value into that storage is well-defined.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track whether storage can hold implicit lifetime objects
replacement:	None
implementations:	None

6.8.3 Alignment [basic.align]

Notes regarding 6.8.3[basic.align], paragraph 1:

1 Object types have *alignment requirements* (6.9.3[basic.fundamental], 6.9.5[basic.compound]) which place restrictions on the addresses at which an object of that type may be allocated. An *alignment* is an implementation-defined integer value representing the number of bytes between successive addresses at which a given object can be allocated. An object type imposes an alignment requirement on every object of that type; stricter alignment can be requested using the *alignment-specifier* (9.13.2[dcl.align]). Attempting to create an object (6.8.2[intro.object]) in storage that does not meet the alignment requirements of the object's type is undefined behavior (F.2.3[ubx:basic.align.object.alignment]).

Note

ubdef:	basic.align.object.alignment
summary:	placing an object at a storage address misaligned for its type.
runtime-checkable:	Yes
locally-checkable:	Yes
checking:	Insert alignment check when creating or reading objects.
replacement:	None
implementations:	UBSan's `alignment` check (stock and routed through the handler) catches a creation that stores to the object on both compilers, and one that performs no store on Clang only, though current Clang misses either through a pointer formed from a character array; the P3850 prototype's native check catches both on both compilers.

6.8.4 **Lifetime** **[basic.life]**

Notes regarding 6.8.4[basic.life], paragraphs 7-12:

7 Before the lifetime of an object has started but after the storage which the object will occupy has been allocated[16] 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. For an object under construction or destruction, see 11.9.5[class.cdtor]. Otherwise, such a pointer refers to allocated storage (6.8.6.5.2[basic.stc.dynamic.allocation]), and using the pointer as if the pointer were of type `void*` is well-defined. Indirection through such a pointer is permitted but the resulting lvalue may only be used in limited ways, as described below. The program has undefined behavior if:

— the pointer is used as the operand of a *delete-expression* (F.2.4[ubx:lifetime.outside.pointer.delete]),*

— the pointer is used to access a non-static data member or call a non-static member function of the object (F.2.5[ubx:lifetime.outside.pointer.member]), or†

— the pointer is converted (7.3.12[conv.ptr], 7.6.1.9[expr.static.cast]) to a pointer to a virtual base class (F.2.6[ubx:lifetime.outside.pointer.virtual]) or‡ to a base class thereof, or

— the pointer is used as the operand of a `dynamic_cast` (7.6.1.7[expr.dynamic.cast]) (F.2.7[ubx:lifetime.outside.pointer.dynamic.cast]).**

[*Example*:

```
struct B {
  virtual void f();
  void mutate();
  virtual ~B();
};

struct D1 : B { void f(); };
struct D2 : B { void f(); };

void B::mutate() {
  new (this) D2;     // reuses storage — ends the lifetime of *this
  f();               // undefined behavior
  ... = this;        // OK, this points to valid memory
}
```

```
void g() {
  void* p = std::malloc(sizeof(D1) + sizeof(D2));
  B* pb = new (p) D1;
  pb->mutate();
  *pb;              // OK, pb points to valid memory
  void* q = pb;     // OK, pb points to valid memory
  pb->f();          // undefined behavior: lifetime of *pb has ended
}
```

— *end example*]

---

[16] For example, before the dynamic initialization of an object with static storage duration (6.10.3.3[basic.start.dynamic]).

Note

∗ **ubdef:**	lifetime.outside.pointer.delete
summary:	`delete`-ing a pointer to an object whose lifetime has already ended.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track whether storage is associated with an object of correct type within its lifetime
replacement:	None
implementations:	Rejected during constant evaluation by Clang, and by GCC only with the P3850 GCC branch.

Note

† **ubdef:**	lifetime.outside.pointer.member
summary:	accessing a member through a pointer to an object outside its lifetime.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track whether storage is associated with an object of correct type within its lifetime
replacement:	None
implementations:	Rejected in constant evaluation by Clang; GCC rejects data-member access but not a non-virtual member-function call.

Note

‡ **ubdef:** lifetime.outside.pointer.virtual

summary: converting a pointer outside its lifetime into a virtual base class pointer.

runtime-checkable: Yes

locally-checkable: No

checking: Track whether storage is associated with an object of correct type within its lifetime

replacement: None

implementations: Rejected in constant evaluation by GCC, and by Clang only with the P3850 Clang branch; for polymorphic classes, heuristically caught by UBSan's `vptr` check on GCC and by MemorySanitizer on Clang.

Note

** **ubdef:** lifetime.outside.pointer.dynamic.cast

summary: `dynamic_cast` through a pointer to an object outside its lifetime.

runtime-checkable: Yes

locally-checkable: No

checking: Track whether storage is associated with an object of correct type within its lifetime

replacement: None

implementations: Rejected in constant evaluation by Clang, and by GCC for automatic objects only (GCC misses a destroyed heap object whose storage has not been released); no runtime detection.

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. For an object under construction or destruction, see 11.9.5[class.cdtor]. Otherwise, such a glvalue refers to allocated storage (6.8.6.5.2[basic.stc.dynamic.allocation]), and using the properties of the glvalue that do not depend on its value is well-defined. The program has undefined behavior if:

— the glvalue is used to access the object (F.2.8[ubx:lifetime.outside.glvalue.access]), or*

— the glvalue is used to call a non-static member function of the object (F.2.9[ubx:lifetime.outside.glvalue.member]), or†

— the glvalue is bound to a reference to a virtual base class (9.5.4[dcl.init.ref]) (F.2.10[ubx:lifetime.outside.glvalue.virtual]), or‡

— the glvalue is used as the operand of a `dynamic_cast` (7.6.1.7[expr.dynamic.cast]) or as the operand of `typeid` (F.2.11[ubx:lifetime.outside.glvalue.dynamic.cast]).**

[*Note*: Therefore, undefined behavior results if an object that is being constructed in one thread is referenced from another thread without adequate synchronization. — *end note*]

Note

∗ **ubdef:** lifetime.outside.glvalue.access

summary: using a glvalue to read or write an object outside its lifetime (before construction completes or after it ends).

runtime-checkable: Yes

locally-checkable: No

checking: Track whether storage is associated with an object of correct type within its lifetime

replacement: None

implementations: Caught by constant evaluation on Clang and on the P3850 GCC branch (stock GCC misses a write to a destroyed heap object); no runtime tool catches it.

Note

† **ubdef:** lifetime.outside.glvalue.member

summary: calling a non-static member function through a glvalue outside its lifetime.

runtime-checkable: Yes

locally-checkable: No

checking: Track whether storage is associated with an object of correct type within its lifetime

replacement: None

implementations: Caught by constant evaluation on Clang; GCC's constant evaluator accepts the call (miss).

Note

‡ **ubdef:** lifetime.outside.glvalue.virtual

summary: binding a glvalue outside its lifetime to a virtual base class reference.

runtime-checkable: Yes

locally-checkable: No

checking: Track whether storage is associated with an object of correct type within its lifetime

replacement: None

implementations: Rejected in constant evaluation by GCC except when the glvalue names the complete object, and by Clang only with the P3850 Clang branch; for polymorphic classes, heuristically caught by MemorySanitizer on Clang and, except for a named complete object, by UBSan's `vptr` check on GCC.

Note

** **ubdef:** lifetime.outside.glvalue.dynamic.cast
summary: using a glvalue to a dead object as the operand of `dynamic_cast` or `typeid`.
runtime-checkable: Yes
locally-checkable: No
checking: Track whether storage is associated with an object of correct type within its lifetime
replacement: None
implementations: Heuristically caught by MemorySanitizer on Clang for `typeid`; no runtime tool catches `dynamic_cast`. Constant evaluation rejects `dynamic_cast` on both compilers; `typeid` only on Clang.

11 If a program ends the lifetime of an object of type `T` with static (6.8.6.2[basic.stc.static]), thread (6.8.6.3[basic.stc.thread]), or automatic (6.8.6.4[basic.stc.auto]) storage duration and if `T` has a non-trivial destructor,[17] 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 (F.2.12[ubx:original.type.implicit.destructor]). This is true*even if the block is exited with an exception.

[*Example*:

```
struct B {
  ~B();
};

void h() {
  B b;
  new (&b) T;
}                                   // undefined behavior at block exit
```

— *end example*]

[17] That is, an object for which a destructor will be called implicitly—upon exit from the block for an object with automatic storage duration, upon exit from the thread for an object with thread storage duration, or upon exit from the program for an object with static storage duration.

Note

* **ubdef:** original.type.implicit.destructor
summary: an implicit destructor at scope exit running on storage reused for a different type.
runtime-checkable: Yes
locally-checkable: No
checking: Checking this would require maintaining type and lifetime information across an entire program.
replacement: None
implementations: None

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 (F.2.13[ubx:creating.within.const.complete.obj]).*

[*Example*:

```
  B();
  ~B();
};

const B b;

void h() {
  b.~B();
  new (const_cast<B*>(&b)) const B;     // undefined behavior
}
```

— *end example*]

Note

* **ubdef:**	creating.within.const.complete.obj
summary:	creating a new object in storage a `const` complete object occupies or used to occupy.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track whether storage is associated with a `const` object
replacement:	None
implementations:	Clang's constant evaluation catches both `.live` and `.reuse` subsets; GCC's catches neither.

6.8.5 **Indeterminate and erroneous values** **[basic.indet]**

Modify section 6.8.5[basic.indet], paragraphs 1-2:

1 When storage for an object with automatic or dynamic storage duration is obtained, the bytes comprising the storage for the object have the following initial value:

(1.1) — If the object has dynamic storage duration, or is the object associated with a variable or function parameter whose first declaration is marked with the `[[indeterminate]]` attribute (9.13.6[dcl.attr.indet]), the bytes have *indeterminate values*;

(1.2) — otherwise, the bytes have *erroneous values*, where each value is determined by the implementation independently of the state of the program.

If no initialization is performed for an object (including subobjects), such a byte retains its initial value until that value is replaced (9.5.1[dcl.init.general], 7.6.19[expr.assign]). If any bit in the value representation has an indeterminate value, the object has an indeterminate value; otherwise, if any bit in the value representation has an erroneous value, the object has an erroneous value.

[*Note 1*: Lvalue-to-rvalue conversion ~~has undefined behavior~~produces an unspecified erroneous value if the erroneous value ~~of an~~in the object's storage is not valid for its type (7.3.2[conv.lval]), with an implicit precondition assertion that this does not occur. — *end note*]

[*Note 2*: Objects with static or thread storage duration are zero-initialized, see 6.10.3.2[basic.start.static]. — *end note*]

2 If any operand of a built-in operator that produces a prvalue is evaluated, is not a discarded-value expression (7.2.3[expr.context]), and produces an erroneous or indeterminate value, then the value produced by that operator is indeterminate if any operand is indeterminate and erroneous otherwise. Except in the following cases, ~~if~~all evaluations have an implicit precondition assertion that they will not produce an indeterminate value ~~is produced by an evaluation, the behavior is undefined~~ (6.11.2+a[basic.contract.implicit]) (F.2.14[ubx:basic.indet.value]),*and if they do then the result of the evaluation is an indeterminate value; if an erroneous value is produced by an

evaluation, the behavior is erroneous and the result of the evaluation is that erroneous value:

(2.1) — If an indeterminate or erroneous value of unsigned ordinary character type (6.9.3[basic.fundamental]) or `std::byte` type (17.2.1[cstddef.syn]) is produced by the evaluation of:
  — the second or third operand of a conditional expression (7.6.16[expr.cond]),
  — the right operand of a comma expression (7.6.20[expr.comma]),
  — the operand of a cast or conversion (conv.integral, expr.type.conv, expr.static.cast, expr.cast) to an unsigned ordinary character type or `std::byte` type (17.2.1[cstddef.syn]), or
  — a discarded-value expression (7.2.3[expr.context]),

  then the result of the operation is an indeterminate value or that erroneous value, respectively.

(2.2) — If an indeterminate or erroneous value of unsigned ordinary character type or `std::byte` type is produced by the evaluation of the right operand of a simple assignment operator (7.6.19[expr.assign]) whose first operand is an lvalue of unsigned ordinary character type or `std::byte` type, an indeterminate value or that erroneous value, respectively, replaces the value of the object referred to by the left operand.

(2.3) — If an indeterminate or erroneous value of unsigned ordinary character type is produced by the evaluation of the initialization expression when initializing an object of unsigned ordinary character type, that object is initialized to an indeterminate value or that erroneous value, respectively.

(2.4) — If an indeterminate or erroneous value of unsigned ordinary character type or `std::byte` type is produced by the evaluation of the initialization expression when initializing an object of `std::byte` type, that object is initialized to an indeterminate value or that erroneous value, respectively.

Converting an indeterminate or erroneous value of unsigned ordinary character type or `std::byte` type produces an indeterminate or erroneous value, respectively. In the latter case, the result of the conversion is the value of the converted operand.

[*Example 1*:

```
int f(bool b) {
  unsigned char *c = new unsigned char;
  unsigned char d = *c;          // OK, d has an indeterminate value
  int e = d;                     // undefined behavior
  return b ? d : 0;              // undefined behavior if b is true
}

int g(bool b) {
  unsigned char c;
  unsigned char d = c;           // no erroneous behavior, but d has an erroneous value

  assert(c == d);                // holds, both integral promotions have erroneous behavior

  int e = d;                     // erroneous behavior
  return b ? d : 0;              // erroneous behavior if b is true
}

void h() {
  int d1, d2;

  int e1 = d1;                   // erroneous behavior
  int e2 = d1;                   // erroneous behavior

  assert(e1 == e2);              // holds
  assert(e1 == d1);              // holds, erroneous behavior
  assert(e2 == d1);              // holds, erroneous behavior

  std::memcpy(&d2, &d1, sizeof(int));   // no erroneous behavior, but d2 has an erroneous value
  assert(e1 == d2);              // holds, erroneous behavior
  assert(e2 == d2);              // holds, erroneous behavior
}
```

— *end example*]

Note

* **ubdef:**	basic.indet.value
summary:	reading an uninitialized (indeterminate) value.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track whether storage has been initialized
replacement:	Only for built-in types: initialize default-initialized variables with erroneous value
implementations:	Heuristically caught by MemorySanitizer on Clang (when the value is used); also rejected during constant evaluation on both compilers.

6.8.6 **Storage duration** [**basic.stc**]

6.8.6.5 **Dynamic storage duration** [**basic.stc.dynamic**]

6.8.6.5.1 **General** [**basic.stc.dynamic.general**]

Notes regarding 6.8.6.5.1[basic.stc.dynamic.general], paragraph 3:

3 If the behavior of an allocation or deallocation function does not satisfy the semantic constraints specified in 6.8.6.5.2[basic.stc.dynamic.allocation] and 6.8.6.5.3[basic.stc.dynamic.deallocation], the behavior is undefined (F.2.15[ubx:basic.stc.alloc.dealloc.constraint]).

Note

ubdef:	basic.stc.alloc.dealloc.constraint
summary:	a replacement or class-specific allocation or deallocation function violating its semantic constraints.
runtime-checkable:	Partially
locally-checkable:	No
checking:	Add postcondition assertions to allocation functions, evaluate them. Some constraints can be expressed this way, some (such as global distinctness) are not locally verifiable.
replacement:	None
implementations:	The null-return case is caught by UBSan's `returns-nonnull-attribute` check on Clang.

6.8.6.5.2 **Allocation functions** [**basic.stc.dynamic.allocation**]

Notes regarding 6.8.6.5.2[basic.stc.dynamic.allocation], paragraph 2:

2 An allocation function attempts to allocate the requested amount of storage. If it is successful, it returns the address of the start of a block of storage whose length in bytes is at least as large as the requested size. The order, contiguity, and initial value of storage allocated by successive calls to an allocation function are unspecified. Even if the size of the space requested is zero, the request can fail. If the request succeeds, the value returned by a replaceable allocation function is a non-null pointer value (6.9.5[basic.compound]) `p0` different from any previously returned value `p1`, unless that value `p1` was subsequently passed to a replaceable deallocation function. Furthermore, for the library allocation

functions in 17.6.3.2[new.delete.single] and 17.6.3.3[new.delete.array], `p0` represents the address of a block of storage disjoint from the storage for any other object accessible to the caller. The effect of indirecting through a pointer returned from a request for zero size is undefined (F.2.16[ubx:basic.stc.alloc.zero.dereference]).*[18]

[18] The intent is to have `operator new()` implementable by calling `std::malloc()` or `std::calloc()`, so the rules are substantially the same. C++ differs from C in requiring a zero request to return a non-null pointer.

Note

* **ubdef:**	basic.stc.alloc.zero.dereference
summary:	dereferencing the non-null pointer returned by a zero-size allocation
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track pointer provenance, insert bounds check
replacement:	None
implementations:	An access is heuristically caught by AddressSanitizer and rejected during constant evaluation, on both compilers; an indirection that accesses nothing is rejected during constant evaluation by Clang only.

6.9 Types [basic.types]

6.9.5 Compound types [basic.compound]

Notes regarding 6.9.5[basic.compound], paragraphs 5-6:

5 If an evaluation produces or causes an object to have (6.9.2[basic.types.trivial]) a pointer value to or past the end of an object O and happens before the beginning of the duration of the region of storage for O, the behavior is undefined (F.2.17[ubx:basic.compound.pointer.before.storage.duration]).*

[*Note*: Relaxed atomic operations (32.5.4[atomics.order]) can produce such values. Conversions from integers avoid producing them (7.6.1.10[expr.reinterpret.cast]). — *end note*]

Note

* **ubdef:**	basic.compound.pointer.before.storage.duration
summary:	forming a pointer to an object before its storage duration begins.
runtime-checkable:	No, requires knowing that undefined behavior will or will not be encountered with any potential value a pointer might have.
locally-checkable:	No
checking:	Would require properly identifying when angelic provenance is unable to apply.
replacement:	None
implementations:	None

6 A pointer value P is *valid in the context of* an evaluation E if P is a pointer to function or a null pointer value, or if it is a pointer to or past the end of an object O and E happens after the beginning and happens before the end of the duration of the region of storage for O. If a pointer value P is used in an evaluation E and P is not valid in the context of E:

— If E is an indirection (7.6.2.2[expr.unary.op]), the behavior is undefined (F.2.18[ubx:basic.compound.invalid.pointer]).*

— If E either

  — is a boolean conversion (7.3.15[conv.bool]) or

  — is performed by a unary +, additive, three-way comparison, relational, or equality operator (7.6[expr.compound]),

  the behavior is implementation-defined.

[*Note*: P can be valid in the context of E even if it points to a type unrelated to that of O or if O is not within its lifetime, although further restrictions apply to such pointer values (6.8.4[basic.life], 7.2.1[basic.lval], 7.6.6[expr.add]). — *end note*]

Note

* **ubdef:**	basic.compound.invalid.pointer
summary:	indirection through a pointer to storage that has been freed or whose duration has otherwise ended.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track object lifetimes across the entire program
replacement:	None
implementations:	An access through the pointer is heuristically caught by AddressSanitizer and rejected during constant evaluation, on both compilers; an indirection that accesses nothing is rejected during constant evaluation by Clang only.

6.10 Program execution [basic.exec]

6.10.1 Sequential execution [intro.execution]

Notes regarding 6.10.1[intro.execution], paragraph 10:

10 Except where noted, evaluations of operands of individual operators and of subexpressions of individual expressions are unsequenced.

[*Note*: In an expression that is evaluated more than once during the execution of a program, unsequenced and indeterminately sequenced evaluations of its subexpressions need not be performed consistently in different evaluations. — *end note*]

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 (6.10.2[intro.multithread]) (F.2.19[ubx:intro.execution.unsequenced.modification]) if*

— a side effect on a memory location (6.8.1[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 (6.10.2[intro.multithread]).

[*Note*: Starting the lifetime of an object in a memory location can end the lifetime of objects in other memory locations (6.8.4[basic.life]). — *end note*]

[*Note*: The next subclause imposes similar, but more complex restrictions on potentially concurrent computations. — *end note*]

[*Example*:

```
  i = 7, i++, i++;              // i becomes 9

  i = i++ + 1;                  // the value of i is incremented
  i = i++ + i;                  // undefined behavior
  i = i + 1;                    // the value of i is incremented

  union U { int x, y; } u;
  (u.x = 1, 0) + (u.y = 2, 0);  // undefined behavior
}
```

— *end example*]

Note

* **ubdef:**	intro.execution.unsequenced.modification
summary:	unsequenced side effects (or a side effect and a read) on the same scalar object, single-threaded.
runtime-checkable:	Yes
locally-checkable:	Yes
checking:	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
replacement:	Sequence operations in some unspecified order
note:	Needs more careful specification work, defer to future paper.
implementations:	None; constant evaluation misses it on both compilers, which only warn at compile time.

6.10.2 **Multi-threaded executions and data races** **[intro.multithread]**

6.10.2.2 **Data races** **[intro.races]**

Notes regarding 6.10.2.2[intro.races], paragraph 17:

17 Two actions are *potentially concurrent* if

— they are performed by different threads, or

— they are unsequenced, at least one is performed by a signal handler, and they are not both performed by the same signal handler invocation.

The execution of a program contains a *data race* if it contains two potentially concurrent conflicting actions, at least one of which is not atomic, and neither happens before the other, except for the special case for signal handlers described below. Any such data race

results in undefined behavior (F.2.20[ubx:intro.races.data]).*

[*Note*: It can be shown that programs that correctly use mutexes and `memory_order::seq_cst` operations to prevent all data races and use no other synchronization operations behave as if the operations executed by their constituent threads were simply interleaved, with each value computation of an object being taken from the last side effect on that object in that interleaving. This is normally referred to as "sequential consistency". However, this applies only to data-race-free programs, and data-race-free programs cannot observe most program transformations that do not change single-threaded program semantics. In fact, most single-threaded program transformations remain possible, since any program that behaves differently as a result has undefined behavior. — *end note*]

Note

* **ubdef:**	intro.races.data
summary:	unsynchronized, potentially concurrent accesses to the same memory location, at least one a write.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track from which threads memory is accessed and when accesses synchronize with each other; only practical for a subset of cases
replacement:	Could possibly make all primitive memory accesses implicitly atomic, but that is neither specified nor ready to propose
implementations:	Heuristically detected by ThreadSanitizer on both compilers.

6.10.2.3 **Forward progress** **[intro.progress]**

Notes regarding 6.10.2.3[intro.progress], paragraph 1:

1 The implementation may assume (F.2.21[ubx:intro.progress.stops]) that any thread* will eventually do one of the following:

— terminate,
— invoke the function `std::this_thread::yield` (32.4.5[thread.thread.this]),
— make a call to a library I/O function,
— perform an access through a volatile glvalue,
— perform an atomic or synchronization operation other than an atomic modify-write operation (32.5.4[atomics.order]), or
— continue execution of a trivial infinite loop (8.6.1[stmt.iter.general]).

[*Note*: This is intended to allow compiler transformations such as removal, merging, and reordering of empty loops, even when termination cannot be proven. An affordance is made for trivial infinite loops, which cannot be removed nor reordered. — *end note*]

Note

* **ubdef:**	intro.progress.stops
summary:	a non-terminated thread stops making progress – e.g., a busy-wait whose loop condition is not a constant expression (a trivial infinite loop is not undefined).
runtime-checkable:	No, requires (quite literally) solving the halting problem
locally-checkable:	No
checking:	No checking strategy exists as whether a thread will make progress is undecidable
replacement:	None
implementations:	None

6.10.3 **Start and termination** [**basic.start**]

6.10.3.1 `main` **function** [**basic.start.main**]

Notes regarding 6.10.3.1[basic.start.main], paragraph 4:

4 Terminating the program without leaving the current block (e.g., by calling the function `std::exit(int)` (17.5[support.start.term])) does not destroy any objects with automatic storage duration (11.4.7[class.dtor]). If `std::exit` is invoked during the destruction of an object with static or thread storage duration, the program has undefined behavior (F.2.22[ubx:basic.start.main.exit.during.destruction]).

Note

ubdef:	basic.start.main.exit.during.destruction
summary:	calling `std::exit` while an object with static or thread storage duration is being destroyed.
runtime-checkable:	Yes
locally-checkable:	No, needs new instrumentation of existing functionality
checking:	Track whether static or thread-local objects are currently being destroyed
replacement:	None
implementations:	None

6.10.3.4 **Termination** [**basic.start.term**]

Notes regarding 6.10.3.4[basic.start.term], paragraph 5:

5 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 (F.2.23[ubx:basic.start.term.use.after.destruction]) if the flow of control* passes through the definition of the previously destroyed block variable.

[*Note*: Likewise, the behavior is undefined if the block variable is used indirectly (e.g., through a pointer) after its destruction. — *end note*]

Note

* **ubdef:**	basic.start.term.use.after.destruction
summary:	passing through the definition of a block-scope static or thread-local variable after it has been destroyed.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track the lifetime of static objects
replacement:	None
implementations:	None

6.11 Contract assertions [basic.contract]

6.11.1 General [basic.contract.general]

Add a paragraph to and modify section 6.11.1[basic.contract.general], paragraphs 1-2:

1 *Contract assertions* ~~allow the programmer to~~ specify properties of the state of the program that are expected to hold at certain points during execution. ~~Contract assertions~~*Explicit contract assertions* are introduced by *precondition-specifier*s, *postcondition-specifier*s (9.4.1[dcl.contract.func]), and *assertion-statement*s (8.9[stmt.contract.assert]). *Implicit contract assertions* are introduced as parts of other language constructs.

2 Each explicit contract assertion has a *contract-assertion predicate*, which is an expression of type `bool`.

~~[*Note 1*: The value of the predicate is used to identify program states that are expected. — *end note*]~~

2+a Each implicit contract assertion has a *contract-assertion condition* under which it is considered satisfied.

[*Note a*: If it is determined during program execution that the predicate does not evaluate to `true`, or that the condition of an implicit contract assertion indicates that the contract assertion is not satisfied, a contract violation occurs. — *end note*]

6.11.2 Evaluation [basic.contract.eval]

Add paragraphs to and modify section 6.11.2[basic.contract.eval], paragraphs 1-7:

1 An evaluation of a contract assertion uses one of the following ~~four~~five *evaluation semantics*: *ignore*, *observe*, *enforce*, ~~or~~ *quick-enforce*, or *assume*. Observe, enforce, and quick-enforce are *checking semantics*; enforce and quick-enforce are *terminating semantics*.

2 ~~It~~Except that explicit contract assertions are never evaluated with the assume semantic, it is implementation-defined which evaluation semantic is used for any given evaluation of a contract assertion.

[*Note 1*: The range and flexibility of available choices of evaluation semantics depends on the implementation and need not allow all ~~four~~five evaluation semantics as possibilities. The evaluation semantics can differ for different evaluations of the same contract assertion, including evaluations during constant evaluation. — *end note*]

3 *Recommended practice*: An implementation should provide the option to translate a program such that all evaluations of contract assertions use the ignore semantic as well as the option to

translate a program such that all evaluations of contract assertions use the enforce semantic. By default, evaluations of contract assertions should use the enforce semantic.

4 The evaluation of a contract assertion using the ignore semantic has no effect. ~~[*Note 2*: The predicate is potentially evaluated (6.3[basic.def.odr]), but not evaluated. — *end note*]~~

4+a If the predicate of a contract assertion C evaluated with the assume semantic would not evaluate to `true`, or the condition of such a contract assertion is not satisfied, then for any evaluation E where C happens before E,

(4+a.1) — the evaluation of E has undefined behavior (4.1.2[intro.abstract]), and

(4+a.2) — if there are no evaluations that happen before E and after C, then C is the guarding contract assertion of E.

[*Note 1+a*: If a contract assertion has a predicate, and that contract assertion is evaluated with the ignore or assume semantic, the predicate is potentially evaluated (6.3[basic.def.odr]), but not evaluated. — *end note*]

4+c The evaluation A of a contract assertion C using a checking semantic determines whether a violation occurs.

5 ~~The evaluation A of a contract assertion using~~If C has a ~~checking semantic determines the value of the~~ predicate~~.~~, ~~It~~it is unspecified whether the predicate is evaluated. Let B be the value that would result from evaluating the predicate.

[*Note 2*: To determine whether a predicate would evaluate to `true` or `false`, an alternative evaluation that produces the same value as the predicate but has no side effects can occur.

[*Example 1*:

```
struct S {
  mutable int g = 5;
} s;
void f()
  pre(( s.g++, false ));    // #1
void g()
{
  f();  // Increment of s.g might not occur, even if #1 uses a checking semantic.
}
```

— *end example*]

— *end note*]

6 There is an observable checkpoint (4.1.2[intro.abstract]) CO that happens before A such that any other evaluation that happens before A also happens before CO.

7 A *contract violation* occurs when

(7.a) — the contract assertion has a condition that is not satisfied,

(7.1) — B is `false`,

(7.2) — the evaluation of the predicate exits via an exception, or

(7.3) — the evaluation of the predicate is performed in a context that is manifestly constant-evaluated (7.7.7[expr.const.defns]) and the predicate is not a core constant expression.

[*Note 4*: If B is `true`, or the contract assertion has a condition that is satisfied, no contract violation occurs and control flow continues normally after the point of evaluation of the contract assertion. The evaluation of the predicate can fail to produce a value without causing a contract violation, for example, by calling `longjmp` (17.14.3[csetjmp.syn]) or terminating the program. — *end note*]

6.11.2 + *a* **Implicit contract assertions** **[basic.contract.implicit]**

Add a new section 6.11.2+a[basic.contract.implicit] after 6.11.2[basic.contract.eval]

6.11.2+a Implicit contract assertions [basic.contract.implicit]

1 If an evaluation O has an *implicit precondition assertion* C applied to it, the evaluation in sequence of a list of contract assertions containing only C is sequenced before the evaluation of O and after the evaluation of all operands of O.

2 Any evaluation with undefined behavior has an implicit precondition assertion applied to it that is its *guarding contract assertion*. Unless otherwise specified, the condition of that precondition assertion is never satisfied.

3 [*Note 1*: Behavior is not undefined behavior until control continues past the evaluation of the guarding contract assertion that precedes it. — *end note*]

4 [*Note 2*: Behavior with an implicit precondition assertion that it does not occur results in undefined behavior when that assertion is evaluated with the assume semantic (6.11.2[basic.contract.eval]), but in all other cases behaves as specified. — *end note*]

Note

The above note's purpose is to make it clear that the places where we have replaced what was previously undefined behavior with erroneous behavior that has an implicit precondition can be implemented in a conforming manner by current compilers by dictating that the associated guarding contract assertion is evaluated with the assume semantic.

7 Expressions [expr]

7.1 Preamble [expr.pre]

Modify subclause 7.1[expr.pre], paragraph 4:

4 An *arithmetic expression* is

(4.1) — a unary plus or minus (7.6.2.2[expr.unary.op]),

(4.2) — an addition (7.6.6[expr.add]),

(4.3) — a subtraction (7.6.1.2[expr.sub]), or

(4.4) — a multiplication, division, or remainder (7.6.5[expr.mul])

expression where every (possibly converted) operand is of arithmetic type.

[*Note 3*: There exist non-arithmetic expressions such as compound assignment (7.6.19[expr.assign]) which are defined in terms of arithmetic expressions. — *end note*]

~~The behavior~~If the mathematical result of evaluating an arithmetic expression is ~~undefined (F.3.1[ubx:expr.expr.eval]) if the mathematical result is~~ neither

(4.1) — in the range of representable values for its type nor

(4.2) — a negative infinity, positive infinity, or NaN that is among the values of the type~~.~~,

then the result is an unspecified erroneous value with an implicit precondition assertion that this does not occur (6.11.2+a[basic.contract.implicit]) (F.3.1[ubx:expr.expr.eval]).*

[*Note 4*: If the operands are of a type that adheres to ISO/IEC 60559, division by zero is the only case where an arithmetic expression has erroneous or undefined behavior (7.6.5[expr.mul]). However, some well-defined arithmetic expressions are not core constant expressions (7.7.2[expr.const.core]).

[*Example 1*: The following example assumes that `std::float32_t` is supported (6.9.4[basic.extended.fp]).

```
constexpr std::float32_t min = std::numeric_limits<std::float32_t>::min();          // OK
constexpr std::float32_t max = std::numeric_limits<std::float32_t>::max();          // OK
constexpr std::float32_t inf = std::numeric_limits<std::float32_t>::infinity();     // OK
constexpr std::float32_t nan = std::numeric_limits<std::float32_t>::quiet_NaN();    // OK
```

```
// Furthermore, if arithmetic expressions with operands of type std::float32_t
// behave as specified in ISO/IEC 60559 for binary32:
constexpr std::float32_t inf2 = inf * 2;     // OK, also positive infinity
constexpr std::float32_t zero = min / max;   // OK, result cannot be represented, and is rounded to zero
constexpr std::float32_t oflo = max * 2;     // error: non-finite result but operands are finite (7.7.2[expr.const.
core])
constexpr std::float32_t nan2 = nan * 2;     // OK, propagating a NaN
constexpr std::float32_t udef = inf * 0;     // error: result is NaN but neither operand is NaN (7.7.2[expr.const.
core])
constexpr std::float32_t div0 = max / 0;     // error: division by zero is undefined (7.6.5[expr.mul], 7.7.2[expr.
const.core])
```

— *end example*]

— *end note*]

[*Note 5*: Treatment of division by zero, forming a remainder using a zero divisor, and all floating-point exceptions varies among machines, and is sometimes adjustable by a library function. — *end note*]

Note

* **ubdef:**	expr.expr.eval
summary:	signed integer `+`, `-`, `*`, and unary negation overflow (a subset of the general expression-evaluation rule).
runtime-checkable:	Yes
locally-checkable:	Yes
checking:	Insert a check of whether the given arguments will produce a valid value
replacement:	Coerce into erroneous value, either explicitly or by using the result of the underlying instruction as-is
implementations:	Exhaustively caught by UBSan's `signed-integer-overflow`, constant evaluation, and the P3850 prototype (GCC nonthrowing-only, Clang all semantics).

7.2 Properties of expressions [expr.prop]

7.2.1 Value category [basic.lval]

Notes regarding 7.2.1[basic.lval], paragraph 11:

11 An object of dynamic type $\mathtt{T}_{\text{obj}}$ is *type-accessible* through a glvalue of type $\mathtt{T}_{\text{ref}}$ if $\mathtt{T}_{\text{ref}}$ is similar (7.3.6[conv.qual]) to:

— $\mathtt{T}_{\text{obj}}$,

— a type that is the signed or unsigned type corresponding to $\mathtt{T}_{\text{obj}}$, or

— a `char`, `unsigned char`, or `std::byte` type.

If a program attempts to access (3.1[defns.access]) the stored value of an object through a glvalue through which it is not type-accessible, the behavior is undefined (F.3.2[ubx:expr.basic.lvalue.strict.aliasing.violation]).*[36] 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 (F.3.3[ubx:expr.basic.lvalue.union.initialization]).†

[*Note*: In C, an entire object of structure type can be accessed, e.g., using assignment. By contrast, C++ has no notion of accessing an object of class type through an lvalue of class type. — *end note*]

[36] The intent of this list is to specify those circumstances in which an object can or cannot be aliased.

Note

* **ubdef:**	expr.basic.lvalue.strict.aliasing.violation
summary:	accessing an object through a glvalue of a dissimilar type (strict-aliasing violation).
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track the dynamic type of the object in storage and verify the access glvalue's type is similar to it
replacement:	None
implementations:	None; cannot arise during constant evaluation.

Note

† **ubdef:**	expr.basic.lvalue.union.initialization
summary:	defaulted union copy/move construction or assignment whose argument is not a live object of that union type.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track whether storage is associated with an object of correct type within its lifetime
replacement:	None
implementations:	Rejected during constant evaluation on both GCC and Clang (inactive member or out-of-lifetime argument).

7.2.2 **Type** **[expr.type]**

Notes regarding 7.2.2[expr.type], paragraph 1:

1 If an expression initially has the type "reference to `T`" (9.3.4.3[dcl.ref], 9.5.4[dcl.init.ref]), the type is adjusted to `T` prior to any further analysis; the value category of the expression is not altered. Let X be the object or function denoted by the reference. If a pointer to X would be valid in the context of the evaluation of the expression (6.9.3[basic.fundamental]), the result designates X; otherwise, the behavior is undefined (F.3.4[ubx:expr.type.reference.lifetime]).*

[*Note*: Before the lifetime of the reference has started or after it has ended, the behavior is undefined (see 6.8.4[basic.life]). — *end note*]

Note

* **ubdef:**	expr.type.reference.lifetime
summary:	evaluating (naming) a reference whose referent's storage has been released.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track object and storage lifetimes
replacement:	None
implementations:	None; constant evaluation should reject it but does not on either compiler.

7.3 Standard conversions [conv]

7.3.2 Lvalue-to-rvalue conversion [conv.lval]

Modify section 7.3.2[conv.lval], paragraph 3:

3 The result of the conversion is determined according to the following rules:

(3.1) — If `T` is *cv* `std::nullptr_t`, the result is a null pointer constant (7.3.12[conv.ptr]). [*Note*: Since the conversion does not access the object to which the glvalue refers, there is no side effect even if `T` is volatile-qualified (6.10.1[intro.execution]), and the glvalue can refer to an inactive member of a union (11.5[class.union]). — *end note*]

(3.2) — Otherwise, if `T` has a class type, the conversion copy-initializes the result object from the glvalue.

(3.3) — 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~~prvalue result is ~~undefined~~an unspecified erroneous value, and there is an implicit precondition assertion that this does not occur (6.11.2+a[basic.contract.implicit]) (F.3.5[ubx:conv.lval.valid.representation]).*

[*Example 2*:

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

— *end example*]

(3.4) — Otherwise, the object indicated by the glvalue is read (3.1[defns.access]). Let `V` be the value contained in the object. The prvalue result is

- the value of type `T` congruent (6.9.3[basic.fundamental]) to `V` if `T` is an integer type,
- the result of `reinterpret_cast<T>(reinterpret_cast<std::uintptr_t>(V))` if `T` is volatile-qualified and `V` is a pointer value that is not valid in the context of the evaluation (6.9.5[basic.compound]), and
- `V` otherwise.

Note

* **ubdef:**	conv.lval.valid.representation
summary:	lvalue-to-rvalue conversion of an invalid bit pattern for an object of non-class type
runtime-checkable:	Yes
locally-checkable:	Yes
checking:	Insert a check of whether the value is a valid value representation for the object's type
replacement:	Coerce invalid value representations into erroneous values
implementations:	The `bool` and `enum` subgroups are caught natively by the P3850 prototype; GCC supports only the non-throwing semantics, Clang supports all seven. UBSan's `bool/enum` checks (both in the `undefined` group) together cover both categories. The `int` and `float` subgroups have no invalid representation on commonly available platforms.

7.3.3 **Array-to-pointer conversion** **[conv.array]**

Notes regarding 7.3.3[conv.array], paragraph 1:

1 An expression E of type "array of `N T`" or "array of unknown bound of `T`" can be converted to a prvalue of type "pointer to `T`". If E is a prvalue, the temporary materialization conversion (7.3.5[conv.rval]) is applied. If the result of E (possibly converted) is an object whose type is similar to the type of E, the result is a pointer to the first element of the array; otherwise, the behavior is undefined (F.3.6[ubx:conv.array.unsimilar.type]).

Note

ubdef:	conv.array.unsimilar.type
summary:	performing an array-to-pointer conversion on an array glvalue whose referenced object's type is not similar to the expression's type.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track the dynamic type of the object in storage and verify that it is similar to the type of the array expression
replacement:	None
implementations:	None

7.3.10 **Floating-point conversions** **[conv.double]**

Modify section 7.3.10[conv.double], paragraph 2:

2 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~~result is ~~undefined~~an unspecified erroneous value with an implicit precondition assertion that this does not occur (6.11.2+a[basic.contract.implicit]) (F.3.7[ubx:conv.double.out.of.range]).

Note

ubdef:	conv.double.out.of.range
summary:	converting a floating-point value to a type that cannot represent that value or values surrounding that value.
runtime-checkable:	Yes
locally-checkable:	Yes
checking:	Insert a check of whether the value is one that can be converted
replacement:	Coerce into erroneous value
implementations:	None

7.3.11 **Floating-integral conversions** **[conv.fpint]**

Modify section 7.3.11[conv.fpint], paragraphs 1-2:

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~~An unspecified erroneous value is ~~undefined (F.3.8[ubx:conv.fpint.float.not.represented])~~ produced if the truncated value cannot be represented in the destination type, with an implicit precondition assertion that this does not occur (6.11.2+a[basic.contract.implicit]) (F.3.8[ubx:conv.fpint.float.not.represented]).*

[*Note 1*: If the destination type is `bool`, see 7.3.15[conv.bool]. — *end note*]

Note

* **ubdef:**	conv.fpint.float.not.represented
summary:	float-to-integer conversion whose truncated value isn't representable in the destination type.
runtime-checkable:	Yes
locally-checkable:	Yes
checking:	Insert a check of whether the value is valid
replacement:	Coerce into erroneous value
implementations:	Checked by UBSan's `float-cast-overflow`, in constant evaluation, and implemented natively by the P3850 prototype.

2 A prvalue of an integer type or of an unscoped enumeration type can be converted to a prvalue of a floating-point type. The result is exact if possible. If the value being converted is in the range of values that can be represented but the value cannot be represented exactly, it is an implementation-defined choice of either the next lower or higher representable value.

[*Note 2*: Loss of precision occurs if the integral value cannot be represented exactly as a value of the floating-point type. — *end note*]

If the value being converted is outside the range of values that can be represented, ~~the behavior~~an unspecified erroneous value is ~~undefined~~produced with an implicit precondition assertion that this does not occur (6.11.2+a[basic.contract.implicit]) (F.3.9[ubx:conv.fpint.int.not.represented]).*If the source type is `bool`, the value `false` is converted to zero and the value `true` is converted to one.

Note

* **ubdef:**	conv.fpint.int.not.represented
summary:	integer/unscoped-enum to floating-point conversion whose value isn't representable.
runtime-checkable:	Yes
locally-checkable:	Yes
checking:	Insert a check of whether the value is valid
replacement:	Coerce into erroneous value
implementations:	None

7.3.12 Pointer conversions [conv.ptr]

Notes regarding 7.3.12[conv.ptr], paragraph 3:

3 A prvalue `v` of type "pointer to *cv* `D`", where `D` is a complete class type, can be converted to a prvalue of type "pointer to *cv* `B`", where `B` is a base class (11.7[class.derived]) of `D`. If `B` is an inaccessible (11.8[class.access]) or ambiguous (6.5.2[class.member.lookup]) base class of `D`, a program that necessitates this conversion is ill-formed. If `v` is a null pointer value, the result is a null pointer value. Otherwise, if `B` is a virtual base class of `D` or is a base class of a virtual base class of `D` and `v` does not point to an

object whose type is similar (7.3.6[conv.qual]) to `D` and that is within its lifetime or within its period of construction or destruction (11.9.5[class.cdtor]), the behavior is undefined (F.3.10[ubx:conv.ptr.virtual.base]).*Otherwise, the result is a pointer to the base class subobject of the derived class object.

Note

* **ubdef:**	conv.ptr.virtual.base
summary:	converting a derived-class pointer to a virtual base when the source isn't a valid, in-lifetime object.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track whether storage is associated with an object of correct type within its lifetime or an object currently being constructed or destroyed
replacement:	None
implementations:	UBSan's `vptr` check catches some cases (heuristic): GCC for `.past.end` and `.lifetime`, Clang for `.lifetime` and `.union`; AddressSanitizer catches `.past.end` on a stack array (heuristic). Constant evaluation rejects `.past.end` on both compilers and `.lifetime` on GCC; our P3850 Clang branch also rejects `.lifetime` and `.union`.

7.3.13 **Pointer-to-member conversions** **[conv.mem]**

Notes regarding 7.3.13[conv.mem], paragraph 2:

2 A prvalue of type "pointer to member of `B` of type *cv* `T`", where `B` is a class type, can be converted to a prvalue of type "pointer to member of `D` of type *cv* `T`", where `D` is a complete class derived (11.7[class.derived]) from `B`. If `B` is an inaccessible (11.8[class.access]), ambiguous (6.5.2[class.member.lookup]), or virtual (11.7.2[class.mi]) base class of `D`, or a base class of a virtual base class of `D`, a program that necessitates this conversion is ill-formed. 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 (F.3.11[ubx:conv.member.missing.member]).*Otherwise, the result of the conversion refers to the same member as the pointer to member before the conversion took place, but it refers to the base class member as if it were a member of the derived class. The result refers to the member in `D`'s instance of `B`. Since the result has type "pointer to member of `D` of type *cv* `T`", indirection through it with a `D` object is valid. The result is the same as if indirecting through the pointer to member of `B` with the `B` subobject of `D`. The null member pointer value is converted to the null member pointer value of the destination type.[40]

---

[40] The rule for conversion of pointers to members (from pointer to member of base to pointer to member of derived) appears inverted compared to the rule for pointers to objects (from pointer to derived to pointer to base) (7.3.12[conv.ptr], 11.7[class.derived]). This inversion is necessary to ensure type safety. Note that a pointer to member is not an object pointer or a function pointer and the rules for conversions of such pointers do not apply to pointers to members. In particular, a pointer to member cannot be converted to a `void*`.

Note

* **ubdef:**	conv.member.missing.member
summary:	converting a base pointer-to-member to a derived one that could not actually contain that member.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track which type the pointer to member originated from.
replacement:	None
implementations:	Clang's constant evaluation rejects the conversion itself; GCC's does not. No run-time tool detects it.

7.6 Compound expressions [expr.compound]

7.6.1 Postfix expressions [expr.post]

7.6.1.3 Function call [expr.call]

Notes regarding 7.6.1.3[expr.call], paragraph 6:

6 A type $\mathtt{T}_{\text{call}}$ is *call-compatible* with a function type $\mathtt{T}_{\text{func}}$ if $\mathtt{T}_{\text{call}}$ is the same type as $\mathtt{T}_{\text{func}}$ or if the type "pointer to $\mathtt{T}_{\text{func}}$" can be converted to type "pointer to $\mathtt{T}_{\text{call}}$" via a function pointer conversion (7.3.14[conv.fctptr]). 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 (F.3.12[ubx:expr.call.different.type]).*

[*Note*: This requirement allows the case when the expression has the type of a potentially-throwing function, but the called function has a non-throwing exception specification, and the function types are otherwise the same. — *end note*]

Note

* **ubdef:**	expr.call.different.type
summary:	calling a function through a pointer/reference type incompatible with its actual definition.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track type information of function based on address
replacement:	None
implementations:	Caught by Clang's UBSan `function/undefined` check (integrated on the P3850 Clang branch: *quick-enforce*, *`noexcept`-enforce*, and *`noexcept`-observe*); GCC has no equivalent; cannot arise during constant evaluation (requires `reinterpret_cast`).

7.6.1.5 Class member access [expr.ref]

Notes regarding 7.6.1.5[expr.ref], paragraph 10:

10 If `E2` designates a non-static member (possibly after overload resolution) or direct base class relationship and the result of `E1` is an object whose type is not similar (7.3.6[conv.

qual]) to the type of `E1`, the behavior is undefined (F.3.13[ubx:expr.ref.member.not.similar]).*

[*Example*:

```
struct B { int j; };
struct D : A, B {};
void f() {
  D d;
  static_cast<B&>(d).j;          // OK, object expression designates the B subobject of d
  reinterpret_cast<B&>(d).j;     // undefined behavior
}
```

— *end example*]

Note

* **ubdef:**	expr.ref.member.not.similar
summary:	accessing a member through a glvalue not similar to the object's actual type.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track whether storage is associated with an object of correct type within its lifetime
replacement:	None
implementations:	`.polymorphic` caught heuristically by UBSan's `vptr` check on GCC (integrated: *quick-enforce*, ***`noexcept`-enforce***, and ***`noexcept`-observe***), and on Clang except through a reference variable or parameter; `.nonpolymorphic` not caught.

7.6.1.7 Dynamic cast [expr.dynamic.cast]

Notes regarding 7.6.1.7[expr.dynamic.cast], paragraph 7:

7 If `v` has type "pointer to *cv* `U`" and `v` does not point to an object whose type is similar (7.3.6[conv.qual]) to `U` and that is within its lifetime or within its period of construction or destruction (11.9.5[class.cdtor]), the behavior is undefined (F.3.14[ubx:expr.dynamic.cast.pointer.lifetime]).*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 (F.3.15[ubx:expr.dynamic.cast.glvalue.lifetime]).

Note

* **ubdef:**	expr.dynamic.cast.pointer.lifetime
summary:	`dynamic_cast` on a non-null pointer to the wrong type or outside any object's lifetime.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track whether storage is associated with an object of correct type within its lifetime or an object currently being constructed or destroyed
replacement:	None
implementations:	Caught heuristically by Clang's UBSan `vptr` check; rejected during constant evaluation on Clang.

Note

ubdef: expr.dynamic.cast.glvalue.lifetime

summary: `dynamic_cast<T&>` on an object of the wrong type or outside its lifetime.

runtime-checkable: Yes

locally-checkable: No

checking: Track whether storage is associated with an object of correct type within its lifetime or an object currently being constructed or destroyed

replacement: None

implementations: Rejected during constant evaluation on both compilers.

7.6.1.9 **Static cast** [**expr.static.cast**]

Notes regarding 7.6.1.9[expr.static.cast], paragraph 2:

2 An lvalue of type "*cv1* `B`", where `B` is a class type, can be cast to type "reference to *cv2* `D`", where `D` is a complete class derived (11.7[class.derived]) from `B`, if *cv2* is the same cv-qualification as, or greater cv-qualification than, *cv1*. If `B` is a virtual base class of `D` or a base class of a virtual base class of `D`, or if no valid standard conversion from "pointer to `D`" to "pointer to `B`" exists (7.3.12[conv.ptr]), the program is ill-formed. An xvalue of type "*cv1* `B`" can be cast to type "rvalue reference to *cv2* `D`" with the same constraints as for an lvalue of type "*cv1* `B`". If the object of type "*cv1* `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 (F.3.16[ubx:expr.static.cast.base.class]).*

[*Example*:

```
struct D : public B { };
D d;
B &br = d;

static_cast<D&>(br);          // produces lvalue denoting the original d object
```

— *end example*]

Note

* **ubdef:** expr.static.cast.base.class

summary: `static_cast`-ing a base glvalue to a reference of the wrong sibling type.

runtime-checkable: Yes

locally-checkable: No

checking: Track whether storage is associated with an object of correct type within its lifetime

replacement: None

implementations: Polymorphic subset caught by UBSan's `vptr` check (Clang, and the P3850 GCC branch; integrated: *quick-enforce*, *`noexcept`-enforce*, and *`noexcept`-observe*) and constant evaluation (both); nonpolymorphic subset caught by constant evaluation (both).

Modify and note section 7.6.1.9[expr.static.cast], paragraphs 8-11:

8 A value of integral or enumeration type can be explicitly converted to a complete enumeration type. If the enumeration type has a fixed underlying type, the value is first converted to that type by integral promotion (7.3.7[conv.prom]) or integral conversion (7.3.9[conv.integral]), if necessary, and then to the enumeration type. 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 (9.8.1[dcl.enum]), and otherwise, ~~the behavior~~an unspecified erroneous value is ~~undefined~~produced with an implicit precondition assertion that this does not occur (6.11.2+a[basic.contract.implicit]) (F.3.17[ubx:expr.static.cast.enum.outside.range]).* A value of floating-point type can also be explicitly converted to an enumeration type. The resulting value is the same as converting the original value to the underlying type of the enumeration (7.3.11[conv.fpint]), and subsequently to the enumeration type.

Note

* **ubdef:**	expr.static.cast.enum.outside.range
summary:	`static_cast`-ing an out-of-range integral or floating-point value into an unscoped enum without a fixed underlying type.
runtime-checkable:	Yes
locally-checkable:	Yes
checking:	Insert a check of whether the value is in the valid range
replacement:	Coerce into erroneous value
implementations:	Integral sources are caught by Clang's constant evaluator, and floating-point sources by GCC's; both sources are caught by Clang's UBSan `enum` check (on the load) and by the P3850 prototype at the cast (both compilers, all semantics).

9 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~~an unspecified erroneous value is ~~undefined~~produced with an implicit precondition assertion that this does not occur (6.11.2+a[basic.contract.implicit]) (F.3.18[ubx:expr.static.cast.fp.outside.range]).

Note

ubdef:	expr.static.cast.fp.outside.range
summary:	an explicit `static_cast` of a floating-point value outside the target type's representable range.
runtime-checkable:	Yes
locally-checkable:	Yes
checking:	Insert a check of whether the value is valid
replacement:	Coerce into erroneous value
implementations:	None

10 A prvalue of type "pointer to *cv1* `B`", where `B` is a class type, can be converted to a prvalue of type "pointer to *cv2* `D`", where `D` is a complete class derived (11.7[class.derived])

from B, if *cv2* is the same cv-qualification as, or greater cv-qualification than, *cv1*. If B is a virtual base class of D or a base class of a virtual base class of D, or if no valid standard conversion from "pointer to D" to "pointer to B" exists (7.3.12[conv.ptr]), the program is ill-formed. The null pointer value (6.9.5[basic.compound]) is converted to the null pointer value of the destination type. If the prvalue of type "pointer to *cv1* 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 (F.3.19[ubx:expr.static.cast.downcast.wrong.derived.type]).

Note

ubdef:	expr.static.cast.downcast.wrong.derived.type
summary:	`static_cast`-ing a base pointer to a pointer of the wrong sibling type.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track whether storage is associated with an object of correct type within its lifetime or an object currently being constructed or destroyed
replacement:	None
implementations:	Polymorphic subset caught by UBSan's `vptr` check (both compilers; integrated: *quick-enforce*, *`noexcept`-enforce*, and *`noexcept`-observe*) and constant evaluation (both); nonpolymorphic subset caught by constant evaluation (both).

11 A prvalue of type "pointer to member of D of type *cv1* T" can be converted to a prvalue of type "pointer to member of B of type *cv2* T", where D is a complete class type and B is a base class (11.7[class.derived]) of D, if *cv2* is the same cv-qualification as, or greater cv-qualification than, *cv1*.

[*Note*: Function types (including those used in pointer-to-member-function types) are never cv-qualified (9.3.4.6[dcl.fct]). — *end note*]

If no valid standard conversion from "pointer to member of B of type T" to "pointer to member of D of type T" exists (7.3.13[conv.mem]), the program is ill-formed. The null member pointer value (7.3.13[conv.mem]) is converted to the null member pointer value of the destination type. 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 (F.3.20[ubx:expr.static.cast.does.not.contain.original.member]).*

[*Note*: Although class B need not contain the original member, the dynamic type of the object with which indirection through the pointer to member is performed must contain the original member; see 7.6.4[expr.mptr.oper]. — *end note*]

Note

* **ubdef:**	expr.static.cast.does.not.contain.original.member
summary:	`static_cast`-ing a pointer-to-member to a base lacking the original member.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track which type the pointer to member originated from
replacement:	None
implementations:	Caught by Clang's constant evaluator.

7.6.1.10 **Reinterpret cast** [expr.reinterpret.cast]

Notes regarding 7.6.1.10[expr.reinterpret.cast], paragraph 5:

5 A value of integral type or enumeration type can be explicitly converted to a pointer. If the value is one that can be produced by converting one or more pointer values (6.9.5[basic.compound]) to an integral type, the result is an unspecified choice among all such values that would result in the program having defined behavior. If no such value exists, the behavior is undefined (F.3.21[ubx:expr.reinterpret.cast.invalid.pointer.value]).*

[*Note*: It is possible for the result to be an invalid pointer value or to not be valid in the context of the conversion (6.9.5[basic.compound]) because it points to an object in a region of storage whose duration has ended or has not yet begun. — *end note*]

Otherwise, the result is implementation-defined.

[*Note*: It can be an invalid pointer value. — *end note*]

Note

* **ubdef:**	expr.reinterpret.cast.invalid.pointer.value
summary:	forming a pointer via `reinterpret_cast` with no real provenance.
runtime-checkable:	No, requires predicting whether any potential value will encounter undefined behavior
locally-checkable:	No
checking:	Would require properly identifying when angelic provenance does not apply
replacement:	None
implementations:	None

7.6.2 **Unary expressions** [expr.unary]

7.6.2.2 **Unary operators** [expr.unary.op]

Notes regarding 7.6.2.2[expr.unary.op], paragraph 1:

1 The unary `*` operator performs *indirection*. Its operand shall be a prvalue of type "pointer to `T`", where `T` is an object or function type. The operator yields an lvalue of type `T`. 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 7.6.1.8[expr.

typeid] (F.3.22[ubx:expr.unary.dereference]).*

[*Note*: Indirection through a pointer to an out-of-lifetime object is valid (6.8.4[basic.life]). — *end note*]

[*Note*: Indirection through a pointer to an incomplete type (other than *cv* `void`) is valid. The lvalue thus obtained can be used in limited ways (to initialize a reference, for example); this lvalue must not be converted to a prvalue, see 7.3.2[conv.lval]. — *end note*]

Note

* **ubdef:**	expr.unary.dereference
summary:	dereferencing a pointer that is null or otherwise does not point to an object or function.
runtime-checkable:	Yes
locally-checkable:	Only for the null pointer case
checking:	Track whether storage is associated with an object of correct type within its lifetime; track whether the address is associated with a function; insert a null pointer check
replacement:	None
implementations:	`.nullptr.access` caught by a native P3850 check (all available semantics on Clang; four non-throwing semantics on GCC), UBSan's `null` check, and constant evaluation, both compilers; `.invalid.access` caught heuristically by AddressSanitizer and by constant evaluation, both compilers; `.nullptr.indirection` caught in every form by the native P3850 check (same semantics) and only partially by UBSan and constant evaluation; `.invalid.indirection` not caught, and the P3850 prototype checks the `.invalid` sub-cases only for an array of statically known bound (see below).

7.6.2.8 **New** **[expr.new]**

Notes regarding 7.6.2.8[expr.new], paragraph 22:

22 [*Note*: Unless an allocation function has a non-throwing exception specification (14.5[except.spec]), it indicates failure to allocate storage by throwing a `std::bad_alloc` exception (6.8.6.5.2[basic.stc.dynamic.allocation], 14.2[except.throw], 17.6.4.1[bad.alloc]); it returns a non-null pointer otherwise. If the allocation function has a non-throwing exception specification, it returns null to indicate failure to allocate storage and a non-null pointer otherwise. — *end note*]

If the allocation function is a non-allocating form (17.6.3.4[new.delete.placement]) that returns null, the behavior is undefined (F.3.23[ubx:expr.new.non.allocating.null]).*Otherwise, if the allocation function returns null, initialization shall not be done, the deallocation function shall not be called, and the value of the *new-expression* shall be null.

Note

* **ubdef:**	expr.new.non.allocating.null
summary:	a non-allocating placement `new` whose pointer argument is null.
runtime-checkable:	Yes
locally-checkable:	Yes
checking:	Insert `post(r: r)`
replacement:	None
implementations:	Caught by UBSan's `null` check (part of the `undefined` group; stock, and integrated on the P3850 branches), which flags the null pointer returned by the non-allocating placement `new`, and by both compilers' constant evaluators, and by the P3850 prototype's native check (both compilers, all semantics).

7.6.2.9 **Delete** **[expr.delete]**

Notes regarding 7.6.2.9[expr.delete], paragraphs 2-3:

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 (F.3.24[ubx:expr.delete.mismatch]).* 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 (17.6.3.4[new.delete.placement]).[56] If not, the behavior is undefined (F.3.25[ubx:expr.delete.array.mismatch]).†

[*Note*: This means that the syntax of the *delete-expression* must match the type of the object allocated by `new`, not the syntax of the *new-expression*. — *end note*]

[*Note*: A pointer to a `const` type can be the operand of a *delete-expression*; it is not necessary to cast away the constness (7.6.1.11[expr.const.cast]) of the pointer expression before it is used as the operand of the *delete-expression*. — *end note*]

---

[56] For nonzero-length arrays, this is the same as a pointer to the first element of the array created by that *new-expression*. Zero-length arrays do not have a first element.

Note

* **ubdef:**	expr.delete.mismatch
summary:	applying single-object `delete` to a pointer from array `new[]`.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track pointer provenance, verify the delete form matches the allocation form (single-object versus array)
replacement:	None
implementations:	Exhaustively caught by ASan's `alloc_dealloc_mismatch` on both compilers (stock, and integrated on the P3850 branches); rejected during constant evaluation on both GCC and Clang.

Note

† **ubdef:**	expr.delete.array.mismatch
summary:	using `delete[]` on a pointer from single-object `new`.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track pointer provenance, verify the delete form matches the allocation form (single-object versus array)
replacement:	None
implementations:	Exhaustively caught by ASan's `alloc_dealloc_mismatch` on both compilers (stock, and integrated on the P3850 branches); rejected during constant evaluation on both GCC and Clang.

3 In a single-object delete expression, if the static type of the object to be deleted is not similar (7.3.6[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 (F.3.26[ubx:expr.delete.dynamic.type.differ]). 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 (F.3.27[ubx:expr.delete.dynamic.array.dynamic.type.differ]).

Note

* **ubdef:** expr.delete.dynamic.type.differ

summary: `delete` through a pointer whose static type differs from its dynamic type, with no virtual destructor.

runtime-checkable: Yes

locally-checkable: No

checking: Track dynamic type of non-polymorphic objects

replacement: None

implementations: Caught heuristically by ASan's `new-delete-type-mismatch` on both compilers (only when the sizes differ; stock, and integrated on the P3850 branches); rejected during constant evaluation by Clang and by the P3850 GCC branch (stock GCC accepts it).

Note

ubdef: expr.delete.dynamic.array.dynamic.type.differ

summary: `delete[]` whose static type differs from the array's dynamic type.

runtime-checkable: Yes

locally-checkable: No

checking: Track dynamic type of non-polymorphic objects

replacement: None

implementations: Caught heuristically by ASan's `new-delete-type-mismatch` on both compilers (only when the element sizes differ; stock, and integrated on the P3850 branches); rejected during constant evaluation on Clang.

7.6.4 **Pointer-to-member operators** **[expr.mptr.oper]**

Notes regarding 7.6.4[expr.mptr.oper], paragraphs 4-6:

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 (F.3.28[ubx:expr.mptr.oper.not.contain.member]).*The expression `E1` is sequenced before the expression `E2`.

Note

* **ubdef:**	expr.mptr.oper.not.contain.member
summary:	accessing a pointer-to-member whose target object lacks that member.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track which type the pointer to member originated from and the dynamic type of non-polymorphic objects
replacement:	None
implementations:	Caught heuristically by AddressSanitizer (both compilers; stock, and integrated on the P3850 branches) when the access lands outside the object; rejected during constant evaluation on Clang.

6 If the result of `.*` or `->*` is a function, then that result can be used only as the operand for the function call operator `()`.

[*Example*:

calls the member function denoted by `ptr_to_mfct` for the object pointed to by `ptr_to_obj`. — *end example*]

In a `.*` expression whose object expression is an rvalue, the program is ill-formed if the second operand is a pointer to member function whose *ref-qualifier* is `&`, unless its *cv-qualifier-seq* is `const`. In a `.*` expression whose object expression is an lvalue, the program is ill-formed if the second operand is a pointer to member function whose *ref-qualifier* is `&&`. The result of a `.*` expression whose second operand is a pointer to a data member is an lvalue if the first operand is an lvalue and an xvalue otherwise. 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 (7.3.13[conv.mem]), the behavior is undefined (F.3.29[ubx:expr.mptr.oper.member.func.null]).

Note

ubdef:	expr.mptr.oper.member.func.null
summary:	a pointer-to-member operator (`.*` or `->*`) whose second operand is a null member pointer.
runtime-checkable:	Yes
locally-checkable:	Yes
checking:	Insert null pointer check
replacement:	None
implementations:	Rejected during constant evaluation on both compilers for a member function, and on Clang for a data member; caught by the P3850 prototype's native check (both subgroups, both compilers, all semantics).

7.6.5 **Multiplicative operators** **[expr.mul]**

Modify section 7.6.5[expr.mul], paragraph 4:

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, then the ~~behavior~~operator yields an unspecified erroneous value and there is an implicit precondition assertion that ~~undefined~~this does not occur (6.11.2+a[basic.contract.implicit]) (F.3.30[ubx:expr.mul.div.by.zero]).*For integral operands, the / operator yields the algebraic quotient with any fractional part discarded;[57] 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~~ each of ~~both~~ a/b and a%b yield an unspecified erroneous value and there is ~~undefined~~an implicit precondition assertion that this does not occur (6.11.2+a[basic.contract.implicit]) (F.3.31[ubx:expr.mul.representable.type.result]).

Note

* **ubdef:**	expr.mul.div.by.zero
summary:	integer division/remainder and floating-point division by zero.
runtime-checkable:	Yes
locally-checkable:	Yes
checking:	Insert a check of whether the second operand is zero
replacement:	Produce an erroneous result after checking the second operand
implementations:	All three cases are caught by constant evaluation and the P3850 prototype (both compilers, all semantics); the integer cases also by UBSan's `integer-divide-by-zero` check, and floating-point division by zero by UBSan's `float-divide-by-zero` check, which is not part of the `undefined` group.

Note

ubdef:	expr.mul.representable.type.result
summary:	integer division or remainder whose quotient overflows, e.g., `INT_MIN / -1` or the paired `INT_MIN % -1`.
runtime-checkable:	Yes
locally-checkable:	Yes
checking:	Insert a check of whether the value is valid
replacement:	Coerce into erroneous value
implementations:	Exhaustively caught by UBSan's `signed-integer-overflow` check, constant evaluation, and the P3850 prototype (both compilers, all semantics).

[57] This is often called truncation towards zero.

7.6.6 Additive operators [expr.add]

Modify and note section 7.6.6[expr.add], paragraphs 4-6:

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 (9.3.4.5[dcl.array]),[58] 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 (F.3.32[ubx:expr.add.out.of.bounds]).*

[*Note*: Adding a value other than 0 or 1 to a pointer to a base class subobject, a member subobject, or a complete object results in undefined behavior. — *end note*]

Note

* **ubdef:** expr.add.out.of.bounds

summary: forming a pointer outside an array's elements and its one-past-the-end position.

runtime-checkable: Yes

locally-checkable: Only if the array bound is statically known

checking: Track pointer provenance, insert bounds check

replacement: None

implementations: `.known.subscript` checked by the P3850 bounds check (all semantics, both compilers) and UBSan's `bounds` check, and rejected during constant evaluation on both compilers; `.known.arithmetic` checked by the P3850 prototype (all semantics, both compilers) and Clang's UBSan `array-bounds` check, and rejected during constant evaluation on Clang only; `.unknown` only heuristically by AddressSanitizer, and rejected during constant evaluation on Clang only; `.pointer.arithmetic` by the P3850 prototype (all semantics, both compilers) and UBSan's `pointer-overflow` check on both compilers.

---

[58] As specified in 6.9.5[basic.compound], an object that is not an array element is considered to belong to a single-element array for this purpose and a pointer past the last element of an array of n elements is considered to be equivalent to a pointer to a hypothetical array element n for this purpose.

5 The result of subtracting two pointer expressions `P` and `Q` is a prvalue of type `std::ptrdiff_t` (17.2.4[support.types.layout]).

(5.1) — If `P` and `Q` both evaluate to null pointer values, the value is 0.

(5.2) — Otherwise, if `P` and `Q` point to, respectively, array elements i and j of the same array object `x`, the expression `P - Q` has the value $i - j$. If that value is not in the range of representable values of type `std::ptrdiff_t`, the ~~behavior~~result is ~~undefined~~an unspecified erroneous value and there is an implicit precondition assertion that this does not occur (6.11.2+a[basic.contract.implicit]) (F.3.33[ubx:expr.sub.pointers.rep]).*

(5.3) — Otherwise, the behavior is undefined (F.3.34[ubx:expr.add.sub.diff.pointers]).†

Note

∗ **ubdef:**	expr.sub.pointers.rep
summary:	subtracting two pointers into the same array when the index difference overflows `std::ptrdiff_t`.
runtime-checkable:	Yes
locally-checkable:	Yes
checking:	Verify that overflow does not occur when converting to `std::ptrdiff_t`
replacement:	Coerce into erroneous value
implementations:	None

Note

† **ubdef:**	expr.add.sub.diff.pointers
summary:	subtracting pointers that do not point into the same array (including different rows or members of one object).
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track pointer provenance, verify both operands point into the same array object
replacement:	None
implementations:	`.distinct.objects` caught at run time by AddressSanitizer's `pointer-subtract` check (stock and integrated) on both compilers, and rejected during constant evaluation on both; `.same.object` rejected during constant evaluation on Clang only.

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 (7.3.6[conv.qual]), the behavior is undefined (F.3.35[ubx:expr.add.not.similar]).∗

[*Example*:

```
unsigned int *p = reinterpret_cast<unsigned int*>(arr + 1);
unsigned int k = *p;            // OK, value of k is 2 (7.3.2[conv.lval])
unsigned int *q = p + 1;        // undefined behavior: p points to an int, not an unsigned int object
```

— *end example*]

Note

∗ **ubdef:**	expr.add.not.similar
summary:	pointer arithmetic advancing by the wrong per-element size (dissimilar pointer/array types).
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track whether storage is associated with an object of correct type
replacement:	None
implementations:	None; cannot arise during constant evaluation.

7.6.7 **Shift operators** **[expr.shift]**

Modify section 7.6.7[expr.shift], paragraph 1:

1 The shift operators << and >> group left-to-right.

shift-expression:
additive-expression
shift-expression << *additive-expression*
shift-expression >> *additive-expression*

The operands shall be prvalues of integral or unscoped enumeration type and integral promotions are performed. The type of the result is that of the promoted left operand. ~~The behavior~~There is ~~undefined (F.3.36[ubx:expr.shift.neg.and.width]) if~~an implicit precondition assertion that the right operand is non-negative~~, or greater~~and less than ~~or equal to~~ the width of the promoted left operand (6.11.2+a[basic.contract.implicit]) (F.3.36[ubx:expr.shift.neg.and.width]); if not, an unspecified erroneous value is produced.

Note

ubdef:	expr.shift.neg.and.width
summary:	shifting by a negative amount or by the type's bit width or more.
runtime-checkable:	Yes
locally-checkable:	Yes
checking:	Insert check whether right operand is valid
replacement:	Coerce into erroneous value
implementations:	Exhaustively caught by UBSan's `shift` check, constant evaluation, and the P3850 prototype (both compilers, all semantics).

7.6.19 **Assignment and compound assignment operators** **[expr.assign]**

Notes regarding 7.6.19[expr.assign], paragraph 7:

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 (F.3.37[ubx:expr.assign.overlap]).*

[*Note*: This restriction applies to the relationship between the left and right sides of the assignment operation; it is not a statement about how the target of the assignment can be aliased in general. See 7.2.1[basic.lval]. — *end note*]

Note

* **ubdef:**	expr.assign.overlap
summary:	an assignment whose source and destination overlap in storage.
runtime-checkable:	Yes
locally-checkable:	Yes
checking:	Check overlap of the two address ranges
replacement:	None
implementations:	The P3850 prototype (all semantics, both compilers); no existing tool, and accepted by both compilers' constant evaluators.

8 Statements [stmt]

8.8 Jump statements [stmt.jump]

8.8.4 The `return` statement [stmt.return]

Modify section 8.8.4[stmt.return], paragraph 4:

4 Flowing off the end of a constructor, a destructor, or a non-coroutine function with a *cv* `void` return type is equivalent to a `return` with no operand. Otherwise, flowing off the end of a function that is neither `main` (6.10.3.1[basic.start.main]) nor a coroutine (9.6.4[dcl.fct.def.coroutine]) ~~results in undefined behavior~~initializes the storage of the return value with unspecified erroneous values, with an implicit precondition assertion that this does not occur (6.11.2+a[basic.contract.implicit]) (F.4.1[ubx:stmt.return.flow.off]).

Note

ubdef:	stmt.return.flow.off
summary:	a value-returning function flowing off its end without a `return`.
runtime-checkable:	Yes
locally-checkable:	Yes
checking:	Equivalent to implicitly inserting a `contract_assert(false)` at end of the *function-body*.
replacement:	As with automatic variables, we can initialize the storage for the return value with erroneous values.
implementations:	Exhaustively caught by UBSan's `return` check, constant evaluation, and the P3850 prototype (all available semantics, both compilers).

8.8.5 The `co_return` statement [stmt.return.coroutine]

Notes regarding 8.8.5[stmt.return.coroutine], paragraph 3:

3 If overload resolution for *p*`.return_void()` succeeds, 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 (F.4.2[ubx:stmt.return.

coroutine.flow.off]).

Note

ubdef:	stmt.return.coroutine.flow.off
summary:	a value-returning coroutine flowing off its body without a `co_return`.
runtime-checkable:	Yes
locally-checkable:	Yes
checking:	Insert `contract_assert(false)` at end of *function-body* if no `return_void` function is provided
replacement:	No, we cannot fabricate an argument of a type suitable to invoke one of possibly many overloads of `return_value` on the promise type.
implementations:	Caught by the P3850 prototype.

8.10 Declaration statement [stmt.dcl]

Notes regarding 8.10[stmt.dcl], paragraph 3:

3 Dynamic initialization of a block variable with static storage duration (6.8.6.2[basic.stc.static]) or thread storage duration (6.8.6.3[basic.stc.thread]) is performed the first time control passes through its declaration; such a variable is considered initialized upon the completion of its initialization. If the initialization exits by throwing an exception, the initialization is not complete, so it will be tried again the next time control enters the declaration. If control enters the declaration concurrently while the variable is being initialized, the concurrent execution shall wait for completion of the initialization.

[*Note*: A conforming implementation cannot introduce any deadlock around execution of the initializer. Deadlocks might still be caused by the program logic; the implementation need only avoid deadlocks due to its own synchronization operations. — *end note*]

If control re-enters the declaration recursively while the variable is being initialized, the behavior is undefined (F.4.3[ubx:stmt.dcl.local.static.init.recursive]).*

[*Example*:

```
    static int s = foo(2*i);      // undefined behavior: recursive call
    return i+1;
  }
```

— *end example*]

Note

* **ubdef:**	stmt.dcl.local.static.init.recursive
summary:	recursively re-entering a block-scope `static` variable's initializer.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Insert a recursion counter into a guard for static and thread-local object construction
replacement:	None
implementations:	None

9 Declarations [dcl]

9.2 Specifiers [dcl.spec]

9.2.9 Type specifiers [dcl.type]

9.2.9.2 The *cv-qualifier*s [dcl.type.cv]

Notes regarding 9.2.9.2[dcl.type.cv], paragraphs 4-5:

4 Any attempt to modify (7.6.19[expr.assign], 7.6.1.6[expr.post.incr], 7.6.2.3[expr.pre.incr]) a const object (6.9.6[basic.type.qualifier]) during its lifetime (6.8.4[basic.life]) results in undefined behavior (F.5.1[ubx:dcl.type.cv.modify.const.obj]).*

[*Example*:

```
ci = 4;                                 // error: attempt to modify const

int i = 2;                              // not cv-qualified
const int* cip;                         // pointer to const int
cip = &i;                               // OK, cv-qualified access path to unqualified
*cip = 4;                               // error: attempt to modify through ptr to const

int* ip;
ip = const_cast<int*>(cip);             // cast needed to convert const int* to int*
*ip = 4;                                // defined: *ip points to i, a non-const object

const int* ciq = new const int (3);     // initialized as required
int* iq = const_cast<int*>(ciq);        // cast required
*iq = 4;                                // undefined behavior: modifies a const object
```

For another example,

```
  mutable int i;
  int j;
};
struct Y {
  X x;
  Y();
};

const Y y;
y.x.i++;                                // well-formed: mutable member can be modified
y.x.j++;                                // error: const-qualified member modified
Y* p = const_cast<Y*>(&y);              // cast away const-ness of y
p->x.i = 99;                            // well-formed: mutable member can be modified
p->x.j = 99;                            // undefined behavior: modifies a const subobject
```

— *end example*]

Note

* **ubdef:** dcl.type.cv.modify.const.obj

summary: modifying a `const` object, even via a `const_cast`-obtained pointer.

runtime-checkable: Yes

locally-checkable: No

checking: Track whether storage is associated with a `const` object

replacement: None

implementations: Rejected during constant evaluation on Clang; on GCC except for objects created by `new const T`.

5 The semantics of an access through a volatile glvalue are implementation-defined. 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 (F.5.2[ubx:dcl.type.cv.access.volatile]).

Note

ubdef:	dcl.type.cv.access.volatile
summary:	accessing a `volatile`-qualified object through a non-`volatile` glvalue.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track whether storage is associated with a `volatile` object
replacement:	None
implementations:	Rejected during constant evaluation on Clang; accepted by GCC's constant evaluator.

9.3 Declarators [dcl.decl]

9.3.4 Meaning of declarators [dcl.meaning]

9.3.4.3 References [dcl.ref]

Notes regarding 9.3.4.3[dcl.ref], paragraph 6:

6 Attempting to bind a reference to a function where the converted initializer is a glvalue whose type is not call-compatible (7.6.1.3[expr.call]) with the type of the function's definition results in undefined behavior (F.5.3[ubx:dcl.ref.incompatible.function]).* Attempting to bind a reference to an object where the converted initializer is a glvalue through which the object is not type-accessible (7.2.1[basic.lval]) results in undefined behavior (F.5.4[ubx:dcl.ref.incompatible.type]).†

[*Note*: The object designated by such a glvalue can be outside its lifetime (6.8.4[basic.life]). Because a null pointer value or a pointer past the end of an object does not point to an object, a reference in a well-defined program cannot refer to such things; see 7.6.2.2[expr.unary.op]. As described in 11.4.10[class.bit], a reference cannot be bound directly to a bit-field. — *end note*]

The behavior of an evaluation of a reference (7.5.5[expr.prim.id], 7.6.1.5[expr.ref]) that does not happen after (6.10.2.2[intro.races]) the initialization of the reference is undefined (F.5.5[ubx:dcl.ref.uninitialized.reference]).‡

[*Example*:

```
int &g();
extern int &ir3;
int *ip = 0;
int &ir1 = *ip;                 // undefined behavior: null pointer
int &ir2 = f(ir3);              // undefined behavior: ir3 not yet initialized
int &ir3 = g();
int &ir4 = f(ir4);              // undefined behavior: ir4 used in its own initializer

char x alignas(int);
int &ir5 = *reinterpret_cast<int *>(&x);    // undefined behavior: initializer refers to char object
```

— *end example*]

Note

∗ **ubdef:** dcl.ref.incompatible.function
summary: binding a reference to a function of a call-incompatible type.
runtime-checkable: Yes
locally-checkable: No
checking: Track the types of all functions based on their addresses
replacement: None
implementations: UBSan's `function` (part of `undefined`) check catches it on Clang.

Note

† **ubdef:** dcl.ref.incompatible.type
summary: binding a reference to an object via a type-incompatible (strict-aliasing) access.
runtime-checkable: Yes
locally-checkable: No
checking: Track whether storage is associated with an object of correct type
replacement: None
implementations: None

Note

‡ **ubdef:** dcl.ref.uninitialized.reference
summary: evaluating a reference before it has actually been initialized.
runtime-checkable: Yes
locally-checkable: No
checking: Track whether references have been initialized
replacement: None
implementations: Constant evaluation rejects `.member` on both compilers and `.self` on Clang. MemorySanitizer (Clang) catches `.self` and `.member`, and GCC's UBSan `null` check catches `.static` (both heuristic).

9.6 **Function definitions** [dcl.fct.def]

9.6.4 **Coroutine definitions** [dcl.fct.def.coroutine]

Notes regarding 9.6.4[dcl.fct.def.coroutine], paragraphs 8-11:

8 A suspended coroutine can be resumed to continue execution by invoking a resumption member function (17.13.4.6[coroutine.handle.resumption]) of a coroutine handle (17.13.4[coroutine.handle]) that refers to the coroutine. The evaluation that invoked a resumption member function is called the *resumer*. Invoking a resumption member function for a coroutine that is not suspended results in undefined behavior (F.5.6[ubx:dcl.fct.def.coroutine.resume.not.suspended]).

Note

ubdef:	dcl.fct.def.coroutine.resume.not.suspended
summary:	calling `resume()` on a coroutine handle that is not suspended.
runtime-checkable:	Yes
locally-checkable:	No, requires additional instrumentation
checking:	Track (in a thread-safe fashion) the suspension state associated with every coroutine handle
replacement:	None
implementations:	AddressSanitizer catches the self-resume case on both compilers (use-after-free or double-free; heuristic).

11 The coroutine state is destroyed when control flows off the end of the coroutine or the `destroy` member function (17.13.4.6[coroutine.handle.resumption]) of a coroutine handle (17.13.4[coroutine.handle]) that refers to the coroutine is invoked. In the latter case, control in the coroutine is considered to be transferred out of the function (8.10[stmt.dcl]). The storage for the coroutine state is released by calling a non-array deallocation function (6.8.6.5.3[basic.stc.dynamic.deallocation]). If `destroy` is called for a coroutine that is not suspended, the program has undefined behavior (F.5.7[ubx:dcl.fct.def.coroutine.destroy.not.suspended]).

Note

ubdef:	dcl.fct.def.coroutine.destroy.not.suspended
summary:	calling `destroy()` on a coroutine handle that is not suspended.
runtime-checkable:	Yes
locally-checkable:	No, requires additional instrumentation
checking:	Track (in a thread-safe fashion) the suspension state associated with every coroutine handle
replacement:	None
implementations:	AddressSanitizer (use-after-free/double-free) catches it on both compilers (heuristic).

9.13 Attributes [dcl.attr]

9.13.3 Assumption attribute [dcl.attr.assume]

Modify section 9.13.3[dcl.attr.assume], paragraphs 1-3:

1 The *attribute-token* `assume` may be applied to a null statement; such a statement is an *assumption*. An *attribute-argument-clause* shall be present and shall have the form:

(*conditional-expression*)

The expression is contextually converted to `bool` (7.3.1[conv.general]). The expression is not evaluated and the assumption has no effect. ~~If~~There is an implicit precondition assertion that 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~~ (6.11.2+a[basic.contract.implicit])

(F.5.8[ubx:dcl.attr.assume.false]).

Note

ubdef:	dcl.attr.assume.false
summary:	reaching an `[[assume(expr)]]` attribute while `expr` is false.
runtime-checkable:	Not in general, UB is based on the hypothetical result of an arbitrary expression that cannot be evaluated if it might have side effects.
locally-checkable:	Yes, if checkable at all.
checking:	No automatic checking strategy is possible because the predicate cannot be, in general, proven to be free of side effects; instead, the user has to change `[[assume(x)]]` to `contract_assert<may_be_assumed>(x)` and select an appropriate evaluation semantic
replacement:	Ignore the assumption
implementations:	P3850-native check covers the side-effect-free `.pure` subgroup with every semantic on both compilers, plus a `constexpr` catch on both compilers for `.pure`; the `.nonpure` subgroups support only *assume* and *ignore*, and only GCC's constant evaluation catches `.nonpure.opaque`. UBSan catches `.pure` too (GCC's `unreachable` check, Clang's `builtin` check).

2 [*Note*: The expression is potentially evaluated (6.3[basic.def.odr]). The use of assumptions is intended to allow implementations to analyze the form of the expression and deduce information used to optimize the program. Implementations are not required to deduce any information from any particular assumption. It is expected that the value of a *has-attribute-expression* for the `assume` attribute is `0` if an implementation does not attempt to deduce any such information from assumptions. — *end note*]

3 [*Example 1*:

```
int divide_by_32(int x) {
  [[assume(x >= 0)]];_______// Implicit precondition that x >= 0
  return x/32;□□□□            // ~~The instructions produced for the division~~If assumed, may omit handling of
□□□□                          // ~~may omit handling of~~ negative values~~.~~
}
int f(int y) {
  [[assume(++y == 43)]];□□□□  // y is not incremented
  return y;□□□□               // ~~statement~~If assumed, may ~~be replaced with return 42;~~return 42
}
```

— *end example*]

9.13.10 **Noreturn attribute** **[dcl.attr.noreturn]**

Notes regarding 9.13.10[dcl.attr.noreturn], paragraph 2:

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 (F.5.9[ubx:dcl.attr.noreturn.eventually.returns]).*

[*Note*: The function can terminate by throwing an exception. — *end note*]

Note

* **ubdef:**	dcl.attr.noreturn.eventually.returns
summary:	a `[[noreturn]]`-declared function returns to its caller anyway.
runtime-checkable:	Yes
locally-checkable:	Yes
checking:	Insert `post(false)`
replacement:	Return normally. This would require both the caller and callee translation unit to ignore instead of assume that the function does not return.
implementations:	UBSan's `unreachable` check (own diagnostic) catches it on both Clang and GCC (Clang only for direct calls). GCC's constant evaluation rejects it; Clang's accepts it. The P3850 prototype checks every normal exit of the function (both compilers, all semantics, direct and indirect calls alike).

11 **Classes** [**class**]

11.4 **Class members** [**class.mem**]

11.4.7 **Destructors** [**class.dtor**]

Notes regarding 11.4.7[class.dtor], paragraph 18:

18 Once a destructor is invoked for an object, the object's lifetime ends; the behavior is undefined (F.6.1[ubx:class.dtor.no.longer.exists]) if the destructor is invoked*for an object whose lifetime has ended (6.8.4[basic.life]).

[*Example*: If the destructor for an object with automatic storage duration is explicitly invoked, and the block is subsequently left in a manner that would ordinarily invoke implicit destruction of the object, the behavior is undefined. — *end example*]

Note

* **ubdef:**	class.dtor.no.longer.exists
summary:	invoking a destructor on an object whose lifetime has already ended.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track whether storage is associated with an object of correct type within its lifetime
replacement:	None
implementations:	`.polymorphic` is rejected during constant evaluation on both compilers and caught at runtime, heuristically, by UBSan's `vptr` check on GCC and by MemorySanitizer on Clang. The nonpolymorphic subgroups are caught only by constant evaluation: all three on Clang; `.automatic` on GCC, `.dynamic` only on the P3850 GCC branch, and `.empty` not on GCC.

11.7 **Derived classes** [**class.derived**]

11.7.4 **Abstract classes** [**class.abstract**]

Notes regarding 11.7.4[class.abstract], paragraph 6:

6 Member functions can be called from a constructor (or destructor) of an abstract class; the effect of making a virtual call (11.7.3[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 (F.6.2[ubx:class.abstract.pure.virtual]).

Note

ubdef:	class.abstract.pure.virtual
summary:	calling a pure virtual function from a constructor or destructor.
runtime-checkable:	Yes
locally-checkable:	Yes
checking:	Insert a `pre(false)` into the pure virtual stub pointed to from the base-class vtable
replacement:	None
implementations:	Rejected during constant evaluation, and checkable at run time in the P3850 prototype, on both compilers (GCC's `.direct` check catches the direct call it devirtualizes); GCC's constant evaluator misses a direct call to a pure virtual function that has a definition.

11.9 **Initialization** [**class.init**]

11.9.3 **Initializing bases and members** **[class.base.init]**

Notes regarding 11.9.3[class.base.init], paragraph 18:

18 Member functions (including virtual member functions, 11.7.3[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 (7.6.1.8[expr.typeid]) or of a `dynamic_cast` (7.6.1.7[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 (9.4.1[dcl.contract.func]),

the program has undefined behavior (F.6.3[ubx:class.base.init.mem.fun]).*

[*Example*:

```
public:
  A(int);
};

class B : public A {
  int j;
public:
  int f();
  B() : A(f()),      // undefined behavior: calls member function but base A not yet initialized
  j(f()) { }         // well-defined: bases are all initialized
};

class C {
public:
  C(int);
};

class D : public B, C {
  int i;
public:
  D() : C(f()),      // undefined behavior: calls member function but base C not yet initialized
  i(f()) { }         // well-defined: bases are all initialized
};
```

— *end example*]

Note

* **ubdef:**	class.base.init.mem.fun
summary:	calling a member function, or applying `typeid` or `dynamic_cast`, before the base classes are initialized.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track whether objects are currently being constructed or destroyed
replacement:	None
implementations:	A virtual call is caught, heuristically, by UBSan's `vptr` on GCC only; during constant evaluation Clang rejects a virtual call or `typeid`, both compilers reject `dynamic_cast`, and only the Clang P3850 prototype rejects a member call in a constructor precondition or destructor postcondition. A plain member call in a base-class mem-initializer is caught only by the P3850 prototype, which checks each of these operations on the object itself lexically (both compilers, all semantics).

11.9.5 **Construction and destruction** **[class.cdtor]**

Notes regarding 11.9.5[class.cdtor], paragraphs 1-6:

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 (F.6.4[ubx:class.cdtor.before.ctor]).* 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 (F.6.5[ubx:class.cdtor.after.dtor]).†

[*Example*:

```
struct Y : X { Y(); };                  // non-trivial
struct A { int a; };
struct B : public A { int j; Y y; };    // non-trivial

extern B bobj;
B* pb = &bobj;                          // OK
int* p1 = &bobj.a;                      // undefined behavior: refers to base class member
int* p2 = &bobj.y.i;                    // undefined behavior: refers to member's member

A* pa = &bobj;                          // undefined behavior: upcast to a base class type
B bobj;                                 // definition of bobj

extern X xobj;
int* p3 = &xobj.i;                      // OK, all constructors of X are trivial
X xobj;
```

For another example,

```
struct X : public virtual W { };
struct Y {
  int* p;
  X x;
  Y() : p(&x.j) {   // undefined, x is not yet constructed
    }
};
```

— *end example*]

Note

* **ubdef:**	class.cdtor.before.ctor
summary:	referring to a member or base of an object before its constructor has begun.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track whether objects are currently being constructed or destroyed
replacement:	None
implementations:	`.polymorphic` is rejected during constant evaluation on both compilers and caught, heuristically, by UBSan's `vptr` check on both compilers. `.nonpolymorphic.virtual.base` is rejected during constant evaluation only by the P3850 Clang branch and caught, heuristically, by MemorySanitizer on Clang. `.nonpolymorphic.nonvirtual` is caught by nothing.

Note

† **ubdef:**	class.cdtor.after.dtor
summary:	referring to a member or base of an object after its destructor has run.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track whether objects are currently being constructed or destroyed
replacement:	None
implementations:	`.polymorphic` and `.nonpolymorphic.access` are rejected during constant evaluation on both compilers and caught, heuristically, by MemorySanitizer on Clang; `.polymorphic` is also caught, heuristically, by UBSan's `vptr` check on GCC. `.nonpolymorphic.address` is caught by nothing.

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 (F.6.6[ubx:class.cdtor.convert.pointer]).* 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 (F.6.7[ubx:class.cdtor.form.pointer]).†

[*Example*:

```
struct B : virtual A { };
struct C : B { };
struct D : virtual A { D(A*); };
struct X { X(A*); };

struct E : C, D, X {
  E() : D(this),    // undefined behavior: upcast from E* to A* might use path E* → D* → A*
```

```
                    // but D is not constructed

                    // "D((C*)this)" would be defined: E* → C* is defined because E() has started,
                    // and C* → A* is defined because C is fully constructed

  X(this) {}        // defined: upon construction of X, C/B/D/A sublattice is fully constructed
};
```

— *end example*]

Note

* **ubdef:**	class.cdtor.convert.pointer
summary:	converting a pointer to a base class during that base's not-yet-constructed or already-destroyed window.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track whether objects are currently being constructed or destroyed
replacement:	None
implementations:	None

Note

† **ubdef:**	class.cdtor.form.pointer
summary:	forming a pointer to a member before construction starts or after destruction finishes.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track whether objects are currently being constructed or destroyed
replacement:	None
implementations:	For polymorphic objects, caught, heuristically, by UBSan's `vptr` check on Clang. Constant evaluation accepts both subgroups on both compilers.

4 Member functions, including virtual functions (11.7.3[class.virtual]), can be called during construction or destruction (11.9.3[class.base.init]). When a virtual function is called directly or indirectly from a constructor or from a destructor, including during the construction or destruction of the class's non-static data members, or during the evaluation of a postcondition assertion of a constructor or a precondition assertion of a destructor (9.4.1[dcl.contract.func]), and the object to which the call applies is the object (call it `x`) under construction or destruction, the function called is the final overrider in the constructor's or destructor's class and not one overriding it in a more-derived class. If the virtual function call uses an explicit class member access (7.6.1.5[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 (F.6.8[ubx:class.cdtor.virtual.not.x]).*

[*Example*:

```
  virtual void f();
  virtual void g();
};

struct A : virtual V {
  virtual void f();
};
```

```
struct B : virtual V {
  virtual void g();
  B(V*, A*);
};

struct D : A, B {
  virtual void f();
  virtual void g();
  D() : B((A*)this, this) { }
};

B::B(V* v, A* a) {
  f();              // calls V::f, not A::f
  g();              // calls B::g, not D::g
  v->g();           // v is base of B, the call is well-defined, calls B::g
  a->f();           // undefined behavior: a's type not a base of B
}
```

— *end example*]

Note

* **ubdef:**	class.cdtor.virtual.not.x
summary:	explicit-member-access virtual call during construction or destruction, outside the executing class or its bases.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track whether objects are currently being constructed or destroyed
replacement:	None
implementations:	None

5 The `typeid` operator (7.6.1.8[expr.typeid]) can be used during construction or destruction (11.9.3[class.base.init]). When `typeid` is used in a constructor (including the *mem-initializer* or default member initializer (11.4[class.mem]) for a non-static data member) or in a destructor, or used in a function called (directly or indirectly) from a constructor or destructor, if the operand of `typeid` refers to the object under construction or destruction, `typeid` yields the `std::type_info` object representing the constructor or destructor's class. 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 (F.6.9[ubx:class.cdtor.typeid]).

Note

ubdef:	class.cdtor.typeid
summary:	`typeid` on an object under construction or destruction, outside its own class or bases.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track whether objects are currently being constructed or destroyed
replacement:	None
implementations:	None

6 `dynamic_casts` (7.6.1.7[expr.dynamic.cast]) can be used during construction or destruction (11.9.3[class.base.init]). When a `dynamic_cast` is used in a constructor (including the *mem-initializer* or default member initializer for a non-static data member) or in a destructor, or used in a function called (directly or indirectly) from a constructor

or destructor, if the operand of the `dynamic_cast` refers to the object under construction or destruction, this object is considered to be a most derived object that has the type of the constructor or destructor's class. 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 (F.6.10[ubx:class.cdtor.dynamic.cast]).*

[*Example*:

```
  virtual void f();
};

struct A : virtual V { };

struct B : virtual V {
  B(V*, A*);
};

struct D : A, B {
  D() : B((A*)this, this) { }
};

B::B(V* v, A* a) {
  typeid(*this);                // type_info for B
  typeid(*v);                   // well-defined: *v has type V, a base of B yields type_info for B
  typeid(*a);                   // undefined behavior: type A not a base of B
  dynamic_cast<B*>(v);          // well-defined: v of type V*, V base of B results in B*
  dynamic_cast<B*>(a);          // undefined behavior: a has type A*, A not a base of B
}
```

— *end example*]

Note

* **ubdef:**	class.cdtor.dynamic.cast
summary:	`dynamic_cast` on an object under construction or destruction, outside its own class or bases.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track whether objects are currently being constructed or destroyed
replacement:	None
implementations:	None

13 Templates [temp]

13.9 Template instantiation and specialization [temp.spec]

13.9.2 Implicit instantiation [temp.inst]

Notes regarding 13.9.2[temp.inst], paragraph 18:

18 There is an implementation-defined quantity that specifies the limit on the total depth of recursive instantiations (Annex B[implimits]), which could involve more than one template. The result of an infinite recursion in instantiation is undefined (F.7.1[ubx:temp.inst.inf.recursion]).*

[*Example*:

```
  X<T>* p;          // OK
  X<T*> a;          // implicit generation of X<T> requires
                    // the implicit instantiation of X<T*> which requires
                    // the implicit instantiation of X<T**> which ...
};
```

— *end example*]

Note

* **ubdef:**	temp.inst.inf.recursion
summary:	infinite recursive template instantiation with no base case.

14 Exception handling [except]

14.4 Handling an exception [except.handle]

Notes regarding 14.4[except.handle], paragraph 11:

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 (F.8.1[ubx:except.handle.handler.ctor.dtor]).

Note

ubdef:	except.handle.handler.ctor.dtor
summary:	referring to a member/base of an object from its ctor/dtor function-try-block handler.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track whether objects are currently being constructed or destroyed
replacement:	None
implementations:	Direct references caught by the P3850 prototype's lexical check (both compilers, all semantics); GCC's constant evaluator rejects `.ended` but accepts `.initialized` and `.reference`, and Clang lacks `constexpr` exceptions; for `.ended`, MemorySanitizer (Clang) heuristically catches use of the value read; Clang warns at compile time for direct references in a constructor's handler.

14.5 Exception specifications [except.spec]

Modify subclause 14.5[except.spec], paragraph 6:

6 An expression E is *potentially-throwing* if

(6.1) — E is a function call (7.6.1.3[expr.call]) whose *postfix-expression* has a function type, or a pointer-to-function type, with a potentially-throwing exception specification, or

(6.2) — E implicitly invokes a function (such as an overloaded operator, an allocation function in a *new-expression*, a constructor for a function argument, or a destructor) that has a potentially-throwing exception specification, or

(6.3) — E is a *throw-expression* (7.6.18[expr.throw]), or

(6.4) — E is a `dynamic_cast` expression that casts to a reference type and requires a runtime check (7.6.1.7[expr.dynamic.cast]), or

(6.5) — E is a `typeid` expression applied to a (possibly parenthesized) built-in unary `*`

operator applied to a pointer to a polymorphic class type (7.6.1.8[expr.typeid]), or

(6.6) — any of the immediate subexpressions (6.10.1[intro.execution]) of E that is not an unevaluated operand is potentially-throwing.

[*Note 1*: The evaluation of an expression that is not potentially-throwing can nevertheless exit via an exception if, as part of that evaluation, the violation of an implicit contract assertion (6.11.1[basic.contract.general]) causes a call to the contract-violation handler (6.11.3[basic.contract.handler]) and that handler exits via an exception. — *end note*]

17 Language support library [support]

17.3 Implementation properties [support.limits]

17.3.2 Header `<version>` synopsis [version.syn]

Modify section 17.3.2[version.syn], paragraph 2:

2 Each of the macros defined in `<version>` is also defined after inclusion of any member of the set of library headers indicated in the corresponding comment in this synopsis.

[*Note 1*: Future revisions of this document might replace the values of these macros with greater values. — *end note*]

```
                                        ⋮  92 lines elided  ⋮
  // <deque>, <queue>, <stack>, <string>
#define __cpp_lib_contracts                        202502L // freestanding, also in <contracts>
#define __cpp_lib_contracts_implicit               XXXXXXL // freestanding, also in <contracts>↩
#define __cpp_lib_copyable_function                202306L // also in <functional>
#define __cpp_lib_coroutine                        201902L // freestanding, also in <coroutine>

                                        ⋮  231 lines elided  ⋮
```

17.10 Contract-violation handling [support.contract]

17.10.1 Header `<contracts>` synopsis [contracts.syn]

Modify section 17.10.1[contracts.syn], paragraph 1:

1 The header `<contracts>` defines types for reporting information about contract violations (6.11.2[basic.contract.eval]).

```
// all freestanding
namespace std::contracts {

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

  enum class evaluation_semantic : unspecified {
    ignore = 1,
    observe = 2,
    enforce = 3,
    quick_enforce = 4,
    assume = 5↩
  };

  enum class detection_mode : unspecified {
    predicate_false = 1,
    evaluation_exception = 2
```

```
  };

  class contract_violation {
    // no user-accessible constructor
  public:
    contract_violation(const contract_violation&) = delete;
    contract_violation& operator=(const contract_violation&) = delete;

    see below ~contract_violation();

    const char* comment() const noexcept;
    contracts::detection_mode detection_mode() const noexcept;
    bool is_terminating() const noexcept;
    assertion_kind kind() const noexcept;
    source_location location() const noexcept;
    evaluation_semantic semantic() const noexcept;
  };

  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

17.10.2 **Enumerations** [support.contract.enum]

Modify section 17.10.2[support.contract.enum], paragraph 2:

2 The enumerators of `assertion_kind` correspond to the ~~syntactic~~ forms of a contract assertion (6.11.1[basic.contract.general]), with meanings listed in Table 44[tab:support.contract.enum.kind].

Modify section 17.10.2[support.contract.enum], Table 44 [support.contract.enum.kind]:

Table 44 — Enum `assertion_kind` [tab:support.contract.enum.kind]

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

No changes to section 17.10.2[support.contract.enum], paragraph 3:

3 The enumerators of `evaluation_semantic` correspond to the evaluation semantics with which a contract assertion may be evaluated (6.11.2[basic.contract.eval]), with meanings listed in Table 45[tab:support.contract.enum.semantic].

Modify section 17.10.2[support.contract.enum], Table 45 [support.contract.enum.semantic]:

Table 45 — Enum `evaluation_semantic` [tab:support.contract.enum.semantic]

Name	Meaning
`ignore`	Ignore evaluation semantic
`observe`	Observe evaluation semantic
`enforce`	Enforce evaluation semantic
`quick_enforce`	Quick-enforce evaluation semantic
`assume`	Assume evaluation semantic

No changes to section 17.10.2[support.contract.enum], paragraph 4:

4 The enumerators of `detection_mode` correspond to the manners in which a contract violation can be identified (6.11.2[basic.contract.eval]), with meanings listed in Table tab:support.contract.enum.detection.

Modify section 17.10.2[support.contract.enum], Table 46 [support.contract.enum.detection]:

Table 46 — Enum `detection_mode` **[tab:support.contract.enum.detection]**

Name	Meaning
`predicate_false`	The predicate of the contract assertion evaluated to `false` ~~or~~, would have evaluated to `false`, or its condition is not satisfied.
`evaluation_exception`	An uncaught exception occurred during evaluation of the contract assertion.

17.10.3 Class `contract_violation` [support.contract.violation]

Modify section 17.10.3[support.contract.violation], paragraph 3:

3 *Recommended practice*: The string returned should contain a textual representation of the predicate of the violated contract assertion, a description of the condition that was detected by the assertion, or an empty string if storing a textual representation is undesired.

[*Note 1*: The string can represent a truncated, reformatted, or summarized rendering of the predicate, before or after preprocessing. — *end note*]

F Core undefined behavior [ub]

F.1 General [ub.general]

Modify subclause F.1[ub.general], paragraph 1:

1 This Annex documents undefined behavior explicitly called out in Clause 4[intro] through Clause 15[cpp] using the following phrases:

(1.1) — is undefined

(1.2) — the behavior is undefined

(1.3) — the behavior of the program is undefined

(1.4) — has undefined behavior

(1.5) — have undefined behavior

(1.6) — result has undefined behavior

(1.7) — results in undefined behavior

(1.7+a) — an implicit precondition assertion (6.11.2+a[basic.contract.implicit])

Undefined behavior that is implicit is not covered by this annex. Behavior with an implicit precondition assertion applied to it is listed due to the possibility of that assertion being evaluated with the assume evaluation semantic (6.11.2[basic.contract.eval]). Each entry contains a title, a cross-reference, a summary of the circumstances, and code examples. The code examples are not intended to exhaustively cover all possible ways of invoking that case.

F.2 Clause 6[basic]: Basics [ub.basic]

F.2.14 **basic.indet.value** [ub:basic.indet.value]

Modify UB description F.2.14[ub:basic.indet.value], paragraphs 1-2:

F.2.14 [ub:basic.indet.value]

Specified in: 6.8.5[ubx:basic.indet.value]

When the result of an evaluation is an indeterminate value (but not just an erroneous value) ~~the behavior~~ there is ~~undefined~~an implicit precondition assertion that this does not occur.

2 [*Example 1*:

```
void g() {
  int x [[ indeterminate ]];
  int y = x;    // undefined behaviorprecondition violation
}
```

— *end example*]

F.3 **Clause 7[expr]: Expressions** [ub.expr]

F.3.1 **expr.expr.eval** [ub:expr.expr.eval]

Modify UB description F.3.1[ub:expr.expr.eval], paragraphs 1-3:

F.3.1 [ub:expr.expr.eval]

Specified in: 7.1[ubx: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~~an erroneous value is ~~undefined~~produced with an implicit precondition assertion that this does not occur.

2 [*Example 1*: In the following example, assume that `std::int32_t` is supported (17.4.1[cstdint.syn]) and is `int`.

```
#include <cstdint>
#include <limits>
int main() {
  std::int32_t x1 = std::numeric_limits<std::int32_t>::max() + 1;
        // undefined behaviorerroneous value, 2,147,483,647 + 1 is not representable as a std::int32_t
  std::int32_t x2 = std::numeric_limits<std::int32_t>::min() / -1;
        // undefined behaviorerroneous value, −2,147,483,648/ − 1 is not representable as a std::int32_t
  std::int32_t x3 = std::numeric_limits<std::int32_t>::min() * -1;
        // undefined behaviorerroneous value, −2,147,483,648 ∗ −1 is not representable as a std::int32_t
}
```

— *end example*]

3 [*Example 2*: Assuming a `float` type that does not have NaN or infinities, operations that result in either ~~have undefined behavior~~produce an erroneous value, because their results are not in the range of `float`, and there is an implicit precondition assertion that this does not occur.

```
#include <limits>
int main() {
  float x = std::numeric_limits<float>::max() * 2;  // undefined behaviorerroneous value, infinite value
}
```

— *end example*]

F.3.5 **conv.lval.valid.representation** [ub:conv.lval.valid.representation]

Modify UB description F.3.5[ub:conv.lval.valid.representation], paragraphs 1-2:

F.3.5 [ub:conv.lval.valid.representation]

Specified in: 7.3.2[ubx:conv.lval.valid.representation]

Performing an lvalue-to-rvalue conversion on an object whose value representation is not valid for its type ~~has undefined behavior~~an erroneous value is produced with an implicit precondition assertion that this does not occur.

2 [*Example 1*:

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

— *end example*]

F.3.7 **conv.double.out.of.range** [**ub:conv.double.out.of.range**]

Modify UB description F.3.7[ub:conv.double.out.of.range], paragraphs 1-2:

F.3.7 [**ub:conv.double.out.of.range**]

Specified in: 7.3.10[ubx:conv.double.out.of.range]

Converting a floating-point value to a type that cannot represent that value or values surrounding that value ~~has undefined behavior~~produces an erroneous value with an implicit precondition assertion that this does not occur.

2 [*Example 1*: Assume that `double` has a larger range of representable values than `float` and that `float` has no representation for infinity for the following example.

```
#include <limits>

int main() {
  double d2 = std::numeric_limits<double>::max();
  float f = d2;      // ~~undefined behavior~~erroneous value, d2 is greater than
                     // the largest representable float
}
```

— *end example*]

F.3.8 **conv.fpint.float.not.represented** [**ub:conv.fpint.float.not.represented**]

Modify UB description F.3.8[ub:conv.fpint.float.not.represented], paragraphs 1-2:

F.3.8 [**ub:conv.fpint.float.not.represented**]

Specified in: 7.3.11[ubx:conv.fpint.float.not.represented]

When converting a floating-point value to an integer type, if the truncated value is not representable in the destination type, ~~the behavior~~an erroneous value is ~~undefined~~produced with an implicit precondition assertion that this does not occur.

2 [*Example 1*: In the following example, assume that `std::float64_t`, `std::int32_t`, and `std::uint32_t` are supported.

```
#include <cstdint>
#include <stdfloat>
#include <limits>

int main() {
  std::float64_t d  = static_cast<std::float64_t>(std::numeric_limits<std::int32_t>::max()) + 1;
  std::int32_t   x1 = d;          // ~~undefined behavior~~erroneous value, 2,147,483,647 + 1 is not representable as
std::int32_t
  std::int32_t   x2 = std::numeric_limits<std::float64_t>::infinity();
                                  // ~~undefined behavior~~erroneous value, infinity is not representable as std::int32_t
  std::int32_t   x3 = -std::numeric_limits<std::float64_t>::infinity();
                                  // ~~undefined behavior~~erroneous value, negative infinity is not representable as
std::int32_t
  std::int32_t   x4 = std::numeric_limits<std::float64_t>::quiet_NaN();
                                  // ~~undefined behavior~~erroneous value, NaN is not representable as std::int32_t
  std::uint32_t  x5 = -0.5f;      // OK, truncated value is 0
}
```

— *end example*]

F.3.9 **conv.fpint.int.not.represented** [**ub:conv.fpint.int.not.represented**]

Modify UB description F.3.9[ub:conv.fpint.int.not.represented], paragraphs 1-2:

F.3.9 **[ub:conv.fpint.int.not.represented]**

Specified in: 7.3.11[ubx:conv.fpint.int.not.represented]

When converting a value of integer or unscoped enumeration type to a floating-point type, if the value is not representable in the destination type ~~the behavior~~an erroneous value is ~~undefined~~produced with an implicit precondition assertion that this does not occur.

2 [*Example 1*:

```
int main() {
  unsigned long long x2 = -1;
  float f = x2; // undefined behavior erroneous value on systems where f does not include
                // a representation for infinity and the maximum value for float
                // is smaller than the maximum value for unsigned long long
}
```

— *end example*]

F.3.17 expr.static.cast.enum.outside.range [ub:expr.static.cast.enum.outside.range]

Modify UB description F.3.17[ub:expr.static.cast.enum.outside.range], paragraphs 1-2:

F.3.17 **[ub:expr.static.cast.enum.outside.range]**

Specified in: 7.6.1.9[ubx:expr.static.cast.enum.outside.range]

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 (9.8.1[dcl.enum]), and otherwise, an ~~the behavior~~erroneous value is ~~undefined~~produced with an implicit precondition assertion that this does not occur.

2 [*Example 1*:

```
enum A { e1 = 1, e2 };

void f() {
  enum A a = static_cast<A>(4); // undefined behavior erroneous value, 4 is not within the range of enumeration
values
}
```

— *end example*]

F.3.18 expr.static.cast.fp.outside.range [ub:expr.static.cast.fp.outside.range]

Modify UB description F.3.18[ub:expr.static.cast.fp.outside.range], paragraphs 1-2:

F.3.18 **[ub:expr.static.cast.fp.outside.range]**

Specified in: 7.6.1.9[ubx:expr.static.cast.fp.outside.range]

An explicit conversion of a floating-point value that is outside the range of the target type ~~has undefined behavior~~produces an erroneous value with an implicit precondition assertion that this does not occur.

2 [*Example 1*: If `float` does not adhere to ISO/IEC 60559 and cannot represent positive infinity, a sufficiently large `double` value will be outside the (finite) range of `float`.

```
void f() {
  double d = FLT_MAX;
  d *= 16;
  float f = static_cast<float>(d);  // undefined behavior erroneous value
}
```

— *end example*]

F.3.30 expr.mul.div.by.zero [ub:expr.mul.div.by.zero]

Modify UB description F.3.30[ub:expr.mul.div.by.zero], paragraphs 1-2:

F.3.30 **[ub:expr.mul.div.by.zero]**

Specified in: 7.6.5[ubx:expr.mul.div.by.zero]

Division by zero produces ~~has undefined behavior~~an erroneous value with an implicit precondition assertion that this does not occur.

2 [*Example 1*:

```
int main() {
  int x = 1 / 0;        // undefined behavior erroneous value
  double d = 1.0 / 0.0; // undefined behavior erroneous value
}
```

— *end example*]

F.3.31 expr.mul.representable.type.result [ub:expr.mul.representable.type.result]

Modify UB description F.3.31[ub:expr.mul.representable.type.result], paragraphs 1-2:

F.3.31 **[ub:expr.mul.representable.type.result]**

Specified in: 7.6.5[ubx:expr.mul.representable.type.result]

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` produce ~~is undefined~~an erroneous value with an implicit precondition assertion that this does not occur.

2 [*Example 1*: In the following example, assume that `std::int32_t` is supported (17.4.1[cstdint.syn]) and is `int`. Dividing the minimum `std::int32_t` value of $-2,147,483,648$ by -1 yields $2,147,483,648$ which is not representable by `std::int32_t`.

```
#include <cstdint>
#include <limits>

int main() {
  std::int32_t x = std::numeric_limits<std::int32_t>::min() / -1;   // undefined behavior erroneous value
  std::int32_t y = std::numeric_limits<std::int32_t>::min() % -1;   // undefined behavior erroneous value
}
```

— *end example*]

F.3.33 expr.sub.pointers.rep [ub:expr.sub.pointers.rep]

Modify UB description F.3.33[ub:expr.sub.pointers.rep], paragraphs 1-2:

F.3.33 **[ub:expr.sub.pointers.rep]**

Specified in: 7.6.6[ubx:expr.sub.pointers.rep]

Subtracting two expressions of pointer type where the difference is not representable in `std::ptrdiff_t` ~~has undefined behavior~~produce an erroneous value with an implicit precondition assertion that this does not occur.

2 [*Example 1*: This example requires being able to create a single object whose size is larger than the maximum value representable by `std::ptrdiff_t`, in order to get two pointers whose difference can be computed, and platforms that allow this are rare (6.8.2[intro.object], Annex B[implimits]).

```
#include <cstddef>
#include <cstdint>
char big[static_cast<std::size_t>(PTRDIFF_MAX) + 1];
std::ptrdiff_t d = big + sizeof(big) - big;     // undefined behavior erroneous value
```

— *end example*]

F.3.36 expr.shift.neg.and.width [ub:expr.shift.neg.and.width]

Modify UB description F.3.36[ub:expr.shift.neg.and.width], paragraphs 1-2:

F.3.36 **[ub:expr.shift.neg.and.width]**

Specified in: 7.6.7[ubx:expr.shift.neg.and.width]

Shifting by a negative amount or equal or greater than the bit-width of a type produces ~~has undefined behavior~~an erroneous value with an implicit precondition assertion that this does not occur.

2 [*Example 1*:

```
#include <limits>
unsigned int y1 = 1 << -1;  // undefined behaviorerroneous value, shift is negative
unsigned int y2 = 1 << std::numeric_limits<unsigned int>::digits;
                            // undefined behaviorerroneous value, shift is equal to bit width
unsigned int y3 = 1 >> std::numeric_limits<unsigned int>::digits;
                            // undefined behaviorerroneous value, shift is equal to bit width
```

— *end example*]

F.4 Clause 8[stmt]: Statements [ub.stmt]

F.4.1 stmt.return.flow.off [ub:stmt.return.flow.off]

Modify UB description F.4.1[ub:stmt.return.flow.off], paragraphs 1-2:

F.4.1 **[ub:stmt.return.flow.off]**

Specified in: 8.8.4[ubx:stmt.return.flow.off]

Flowing off the end of a function other than `main` or a coroutine initializes ~~results in undefined behavior~~the return value's storage with erroneous values if the return type is not *cv* `void`, with an implicit precondition assertion that this does not occur.

2 [*Example 1*:

```
int f(int x) {
  if (x)
    return 1;
  // undefined behavior if x is 0
}

void b() {
  int x = f(0);  // undefined behavior, using 0 as an argument will cause f(...) to flow
                 // off the end without a return statement↩
  int x = f(0);  // erroneous value and precondition violation, using 0 as an argument will
                 // cause f(...) to flow off the end without a return statement↩
}
```

— *end example*]

F.5 Clause 9[dcl]: Declarations [ub.dcl]

F.5.8 dcl.attr.assume.false [ub:dcl.attr.assume.false]

Modify UB description F.5.8[ub:dcl.attr.assume.false], paragraphs 1-2:

F.5.8 **[ub:dcl.attr.assume.false]**

Specified in: 9.13.3[ubx:dcl.attr.assume.false]

If an assumption expression would not evaluate to true at the point where it appears, ~~the behavior~~an implicit precondition assertion is ~~undefined~~violated.

2 [*Example 1*:

```
int g(int x) {
  [[assume(x >= 0)]];
  return x/32;
}

int f() {
  return g(-10);    // undefined behaviorprecondition violation, assumption in g is false
}
```

— *end example*]

7 Future extensions

We already briefly touched upon Labels in earlier sections of this paper. Here, we explore other extensions that rely on Labels as proposed in [P3400R5] and provide important additional functionality for implicit contract assertions not proposed in this paper.

7.1 Identifying the UB category

[P3400R5] 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 [P3400R5] Section 3.3.5). Such implicitly defined identification labels would make it possible to identify programmatically, 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.query(bounds_label_tag)) {
      // handle violation of assertion labelled with the bounds label
  }
}
```

Notably, the [P3400R5] approach has an important advantage over using the `detection_mode` enum, as proposed in [P3081R2] 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 5.1) 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.

7.2 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 [P3400R5] Section 4.2). Any possible label, such as "always enforce", "never enforce", etc., can be applied to any group of implicit contract assertions at any granularity: per file, per TU, per module, per namespace, per function, or per code block:

```
int f(int a, int b) {
  contract_control core arithmetic |= always_enforce;
  return a + b;
}
```

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

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

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

For granular, in-source control of the evaluation semantics of implicit contract assertions, we need to agree whether this happens via directives such as the ones proposed in [P3400R5] and shown here, or by using the syntax proposed in the Profiles framework as proposed in [P3589R2]. If we want to have both, we need to specify one in terms of the other to avoid an incoherent and messy design.

If we follow the idea in Section 4.4 and consider Profiles to be a higher-level feature defined in terms of the seven basic tools (the low-level features), then a Profile that enables or disables runtime checks can be defined as essentially a declaration that expands to [P3400R5] directives such as the one shown above. Alternatively, we could design Profiles as an auditing feature rather than a configuration feature: instead of actively enabling certain configuration options, the effect of a Profile would be that the program is ill-formed if the configuration options chosen via [P3400R5] directives or other mechanisms are not compatible with the guarantees that that Profile ensures.

7.3 Integrating assertions and assumptions

In Section 5.4, we introduced the *assume* semantic as a backwards-compatibility escape hatch for newly introduced erroneous behaviour; as such, it can only apply to implicit contract assertions, not to explicit ones.

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

However, Labels, as proposed in [P3400R5], open up the possibility of introducing an explicit label that would allow the *assume* semantic to apply to an explicit contract assertion as well. Consider the limiter example from [P1774R8]:

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

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

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

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

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

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

Acknowledgements

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

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

Claude (Anthropic) was used for editorial assistance during the preparation of this paper, as well as significant parts of the prototype implementations.

References

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

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

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

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

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

[N5054] Thomas Köppe. Working Draft, Standard for Programming Language C++. https://wg21.link/n5054, 2026-07-16.

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

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

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

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

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

[P1995R1] Joshua Berne, Timur Doumler, Andrzej Krzemieński, Ryan McDougall, and Herb Sutter. Contracts – Use Cases. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1995r1.html, 2020-03-02.

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

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

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

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

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

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

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

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

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

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

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

[P3038R0] Bjarne Stroustrup. Concrete suggestions for initial Profiles. https://wg21.link/p3038r0, 2023-12-16.

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

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

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

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

[P3227R0] Gašper Ažman and Timur Doumler. Fixing the library API for contract violation handling. https://wg21.link/p3227r0, 2024-10-16.

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

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

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

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

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

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

[P3400R5] Joshua Berne. Controlling Contract-Assertion Properties. https://wg21.link/p3400r5, 2026-10-23.

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

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

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

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

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

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

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

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

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

[P3596R2] Joshua Berne, Timur Doumler, Jens Maurer, and Shafik Yaghmour. Undefined Behavior and IFNDR Annexes. https://wg21.link/p3596r2, 2026-05-11.

[P3596R3] Joshua Berne, Timur Doumler, Jens Maurer, and Shafik Yaghmour. Undefined Behavior and IFNDR Annexes. https://wg21.link/p3596r3, 2026-06-11.

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

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

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

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

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

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

[P3984R0] Bjarne Stroustrup. A type-safety profile. https://wg21.link/p3984r0, 2026-02-22.

[P4277R1] Joshua Berne. Overview and Implementation Report for P3100. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2026/p4277r1.pdf, 2026-10-23.

[P4284R0] Joshua Berne. Addenda to the Undefined Behavior and IFNDR Annexes. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2026/p4284r0.pdf, 2026-08-13.

[P4284R1] Joshua Berne. Addenda to the Undefined Behavior and IFNDR Annexes. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2026/p4284r1.pdf, 2026-10-23.

[P4298R0] Joshua Berne. Nonthrowing Evaluation Semantics. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2026/p4298r0.pdf, 2026-07-15.

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

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

Appendix A: List of language UB

All wording is taken from the current working draft with [P4284R1] applied. Each row corresponds to one case of explicit core language UB. Rows are arranged by category, as defined in Section 3.2.1; within each category, rows follow the same order as that of the corresponding wording in that working draft. All identifiers are consistent with those listed in [P3596R3] and [P4284R1].

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

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

Identifier	Wording	Runtime-checkable	Locally checkable	Runtime checking strategy	Replacement behaviour
{expr.add.sub. diff.pointers}	[expr.add]/5: The result of subtracting two pointer expressions `P` and `Q` is a prvalue of type `std::ptrdiff_t` ([support.types.layout]). [...] Otherwise, if `P` and `Q` point to, respectively, array elements i and j of the same array object `x`, the expression `P - Q` has the value $i - j$. [...] Otherwise, the behavior is undefined.	Yes	No	Track pointer provenance, verify both operands point into the same array object	None
III. Type and Lifetime					
{intro.object. implicit. create}	[intro.object]/13: 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.	Not generally, requires predicting whether the object will be used correctly as a given type	No	None	None

Identifier	Wording	Runtime-checkable	Locally checkable	Runtime checking strategy	Replacement behaviour
{intro.object. implicit. pointer}	[intro.object]/14: 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 when creating or reading objects.	None

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

Identifier	Wording	Runtime-checkable	Locally checkable	Runtime checking strategy	Replacement behaviour
{lifetime. outside. glvalue. access}	[basic.life]/8: Similarly, before the lifetime of an object has started but after the storage which the object will occupy has been allocated or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any glvalue that refers to the original object may be used but only in limited ways. [...] The program has undefined behavior if the glvalue is used to access the object [...]	Yes	No	Track whether storage is associated with an object of correct type within its lifetime	None
{lifetime. outside. glvalue. member}	[basic.life]/8: [...] The program has undefined behavior if [...] the glvalue is used to call a non-static member function of the object [...]	Yes	No	Track whether storage is associated with an object of correct type within its lifetime	None
{lifetime. outside. glvalue. virtual}	[basic.life]/8: [...] The program has undefined behavior if [...] the glvalue is bound to a reference to a virtual base class ([dcl.init.ref]) [...]	Yes	No	Track whether storage is associated with an object of correct type within its lifetime	None
{lifetime. outside. glvalue. dynamic. cast}	[basic.life]/8: [...] The program has undefined behavior if [...] the glvalue is used as the operand of a `dynamic_cast` ([expr.dynamic.cast]) or as the operand of `typeid`.	Yes	No	Track whether storage is associated with an object of correct type within its lifetime	None

Identifier	Wording	Runtime-checkable	Locally checkable	Runtime checking strategy	Replacement behaviour
{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	Checking this would require maintaining type and lifetime information across an entire program.	None
{creating.within.const.complete.obj}	[basic.life]/12: Creating a new object within the storage that a const, complete object with static, thread, or automatic storage duration occupies, or within the storage that such a const object used to occupy before its lifetime ended, results in undefined behavior.	Yes	No	Track whether storage is associated with a `const` object	None

Identifier	Wording	Runtime-checkable	Locally checkable	Runtime checking strategy	Replacement behaviour
{basic. compound. pointer.before. storage. duration}	[basic.compound]/5: If an evaluation produces or causes an object to have ([basic.types.trivial]) a pointer value to or past the end of an object O and happens before the beginning of the duration of the region of storage for O, the behavior is undefined.	No, requires knowing that undefined behavior will or will not be encountered with any potential value a pointer might have.	No	Would require properly identifying when angelic provenance is unable to apply.	None
{basic. compound. invalid. pointer}	[basic.compound]/6: [...] If a pointer value P is used in an evaluation E and P is not valid in the context of E: If E is an indirection ([expr.unary.op]), the behavior is undefined. [...]	Yes	No	Track object lifetimes across the entire program	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 the dynamic type of the object in storage and verify the access glvalue's type is similar to it	None

Identifier	Wording	Runtime-checkable	Locally checkable	Runtime checking strategy	Replacement behaviour
{expr.basic. lvalue.union. initialization}	[basic.lval]/11.3: If a program invokes a defaulted copy/move constructor or copy/move assignment operator for a union of type `U` with a glvalue argument that does not denote an object of type *cv* `U` within its lifetime, the behavior is undefined.	Yes	No	Track whether storage is associated with an object of correct type within its lifetime	None
{expr.type. reference. lifetime}	[expr.type]/1: If a pointer to X would be valid in the context of the evaluation of the expression ([basic.fundamental]), the result designates X; otherwise, the behavior is undefined.	Yes	No	Track object and storage lifetimes	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	Yes	Insert a check of whether the value is a valid value representation for the object's type	Coerce invalid value representations into erroneous values
{conv.array. unsimilar. type}	[conv.array]/1: [...] If the result of E (possibly converted) is an object whose type is similar to the type of E, the result is a pointer to the first element of the array; otherwise, the behavior is undefined.	Yes	No	Track the dynamic type of the object in storage and verify that it is similar to the type of the array expression	None

Identifier	Wording	Runtime-checkable	Locally checkable	Runtime checking strategy	Replacement behaviour
{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	No	Track whether storage is associated with an object of correct type within its lifetime or an object currently being constructed or destroyed	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]/6: 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]/10: If `E2` is a non-static member and the result of `E1` is an object whose type is not similar ([conv.qual]) to the type of `E1`, the behavior is undefined.	Yes	No	Track whether storage is associated with an object of correct type within its lifetime	None

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

Identifier	Wording	Runtime-checkable	Locally checkable	Runtime checking strategy	Replacement behaviour
{expr.static. cast.downcast. wrong.derived. type}	[expr.static.cast]/10: 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	No	Track whether storage is associated with an object of correct type within its lifetime or an object currently being constructed or destroyed	None
{expr.static. cast.does. not.contain. original. member}	[expr.static.cast]/11: 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. reinterpret. cast.invalid. pointer.value}	[expr.reinterpret.cast]/5: A value of integral type or enumeration type can be explicitly converted to a pointer. If the value is one that can be produced by converting one or more pointer values ([basic.compound]) to an integral type, the result is an unspecified choice among all such values that would result in the program having defined behavior. If no such value exists, the behavior is undefined.	No, requires predicting whether any potential value will encounter undefined behavior	No	Would require properly identifying when angelic provenance does not apply	None
{expr.unary. dereference}	[expr.unary.op]/1: If the operand points to an object or function, the result denotes that object or function; otherwise, the behavior is undefined except as specified in [expr.typeid].	Yes	Only for the null pointer case	Track whether storage is associated with an object of correct type within its lifetime; track whether the address is associated with a function; insert a null pointer check	None

Identifier	Wording	Runtime-checkable	Locally checkable	Runtime checking strategy	Replacement behaviour
{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 the dynamic type of non-polymorphic objects	None

Identifier	Wording	Runtime-checkable	Locally checkable	Runtime checking strategy	Replacement behaviour
{expr.mptr. oper.member. func.null}	[expr.mptr.oper]/6: The result of a .* expression whose second operand is a pointer to a member function is a prvalue. If the second operand is the null member pointer value, the behavior is undefined.	Yes	Yes	Insert null pointer check	None
{expr.add.not. similar}	[expr.add]/6: For addition or subtraction, if the expressions `P` or `Q` have type "pointer to *cv* `T`", where `T` and the array element type are not similar, the behavior is undefined.	Yes	No	Track whether storage is associated with an object of correct type	None
{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

Identifier	Wording	Runtime-checkable	Locally checkable	Runtime checking strategy	Replacement behaviour
{dcl.ref. incompatible. function}	[dcl.ref]/6: Attempting to bind a reference to a function where the converted initializer is a glvalue whose type is not call-compatible ([expr.call]) with the type of the function's definition results in undefined behavior.	Yes	No	Track the types of all functions based on their addresses	None
{dcl.ref. incompatible. type}	[dcl.ref]/6: Attempting to bind a reference to an object where the converted initializer is a glvalue through which the object is not type-accessible ([basic.lval]) results in undefined behavior.	Yes	No	Track whether storage is associated with an object of correct type	None
{dcl.ref. uninitialized. reference}	[dcl.ref]/6: The behavior of an evaluation of a reference ([expr.prim.id], [expr.ref]) that does not happen after ([intro.races]) the initialization of the reference is undefined.	Yes	No	Track whether references have been initialised	None
{class.dtor.no. longer.exists}	[class.dtor]/18: Once a destructor is invoked for an object, the object's lifetime ends; the behavior is undefined if the destructor is invoked for an object whose lifetime has ended ([basic.life]).	Yes	No	Track whether storage is associated with an object of correct type within its lifetime	None

Identifier	Wording	Runtime-checkable	Locally checkable	Runtime checking strategy	Replacement behaviour
{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]/18: Member functions (including virtual member functions, [class.virtual]) can be called for an object under construction or destruction. Similarly, an object under construction or destruction can be the operand of the `typeid` operator ([expr.typeid]) or of a `dynamic_cast` ([expr.dynamic.cast]). However, if these operations are performed during evaluation of a *ctor-initializer* (or in a function called directly or indirectly from a *ctor-initializer*) before all the *mem-initializer*s for base classes have completed, a precondition assertion of a constructor, or a postcondition assertion of a destructor ([dcl.contract.func]), the program has undefined behavior.	Yes	No	Track whether objects are currently being constructed or destroyed	None

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

Identifier	Wording	Runtime-checkable	Locally checkable	Runtime checking strategy	Replacement behaviour
{class.cdtor. form.pointer}	[class.cdtor]/3: To form a pointer to (or access the value of) a direct non-static member of an object `obj`, the construction of `obj` shall have started and its destruction shall not have completed, otherwise the computation of the pointer value (or accessing the member value) results in undefined behavior.	Yes	No	Track whether objects are currently being constructed or destroyed	None
{class.cdtor. virtual.not.x}	[class.cdtor]/4: If the virtual function call uses an explicit class member access ([expr.ref]) and the object expression refers to the complete object of `x` or one of that object's base class subobjects but not `x` or one of its base class subobjects, the behavior is undefined.	Yes	No	Track whether objects are currently being constructed or destroyed	None
{class.cdtor. typeid}	[class.cdtor]/5: If the operand of `typeid` refers to the object under construction or destruction and the static type of the operand is neither the constructor or destructor's class nor one of its bases, the behavior is undefined.	Yes	No	Track whether objects are currently being constructed or destroyed	None

Identifier	**Wording**	**Runtime-checkable**	**Locally checkable**	**Runtime checking strategy**	**Replacement behaviour**
{class.cdtor. dynamic.cast}	[class.cdtor]/6: If the operand of the `dynamic_cast` refers to the object under construction or destruction and the static type of the operand is not a pointer to or object of the constructor or destructor's own class or one of its bases, the `dynamic_cast` results in undefined behavior.	Yes	No	Track whether objects are currently being constructed or destroyed	None
{except. handle. handler. ctor.dtor}	[except.handle]/11: Referring to any non-static member or base class of an object in the handler for a *function-try-block* of a constructor or destructor for that object results in undefined behavior.	Yes	No	Track whether objects are currently being constructed or destroyed	None

Identifier	Wording	Runtime-checkable	Locally checkable	Runtime checking strategy	Replacement behaviour
			IV. Arithmetic		
{expr.expr. eval}	[expr.pre]/4: An arithmetic expression is a unary plus or minus ([expr.unary.op]), an addition ([expr.add]), a subtraction ([expr.sub]), or a multiplication, division, or remainder ([expr.mul]) expression where every (possibly converted) operand is of arithmetic type. [...] The behavior of evaluating an arithmetic expression is undefined if the mathematical result is neither in the range of representable values for its type nor a negative infinity, positive infinity, or NaN that is among the values of the type.	Yes	Yes	Insert a check of whether the given arguments will produce a valid value	Coerce into erroneous value, either explicitly or by using the result of the underlying instruction as-is
{conv.double. out.of.range}	[conv.double]/2: A prvalue of floating-point type can be converted to a prvalue of another floating-point type with a greater or equal conversion rank ([conv.rank]). [...] If the source value can be exactly represented in the destination type, the result of the conversion is that exact representation. If the source value is between two adjacent destination values, the result of the conversion is an implementation-defined choice of either of those values. Otherwise, the behavior is undefined.	Yes	Yes	Insert a check of whether the value is one that can be converted	Coerce into erroneous value

Identifier	Wording	Runtime-checkable	Locally checkable	Runtime checking strategy	Replacement behaviour
{conv.fpint. float.not. represented}	[conv.fpint]/1: A prvalue of a floating-point type can be converted to a prvalue of an integer type. The conversion truncates; that is, the fractional part is discarded. The behavior is undefined if the truncated value cannot be represented in the destination type.	Yes	Yes	Insert a check of whether the value is valid	Coerce into erroneous value
{conv.fpint. int.not. represented}	[conv.fpint]/2: A prvalue of an integer type or of an unscoped enumeration type can be converted to a prvalue of a floating-point type. [...] If the value being converted is outside the range of values that can be represented, the behavior is undefined.	Yes	Yes	Insert a check of whether the value is valid	Coerce into erroneous value
{expr.static. cast.enum. outside. range}	[expr.static.cast]/8: If the enumeration type does not have a fixed underlying type, the value is unchanged if the original value is within the range of the enumeration values ([dcl.enum]), and otherwise, the behavior is undefined.	Yes	Yes	Insert a check of whether the value is in the valid range	Coerce into erroneous value

Identifier	Wording	Runtime-checkable	Locally checkable	Runtime checking strategy	Replacement behaviour
{expr.static. cast.fp. outside. range}	[expr.static.cast]/9: A prvalue of floating-point type can be explicitly converted to any other floating-point type. If the source value can be exactly represented in the destination type, the result of the conversion has that exact representation. If the source value is between two adjacent destination values, the result of the conversion is an implementation-defined choice of either of those values. Otherwise, the behavior is undefined.	Yes	Yes	Insert a check of whether the value is valid	Coerce into erroneous value
{expr.mul.div. by.zero}	[expr.mul]/4: The binary `/` operator yields the quotient, and the binary `%` operator yields the remainder from the division of the first expression by the second. If the second operand of `/` or `%` is zero, the behavior is undefined.	Yes	Yes	Insert a check of whether the second operand is zero	Produce an erroneous result after checking the second operand
{expr.mul. representable. type.result}	[expr.mul]/4: For integral operands, the `/` operator yields the algebraic quotient with any fractional part discarded; if the quotient `a/b` is representable in the type of the result, `(a/b)*b + a%b` is equal to `a`; otherwise, the behavior of both `a/b` and `a%b` is undefined.	Yes	Yes	Insert a check of whether the value is valid	Coerce into erroneous value

Identifier	Wording	Runtime-checkable	Locally checkable	Runtime checking strategy	Replacement behaviour
{expr.sub. pointers.rep}	[expr.add]/5: [...] Otherwise, if `P` and `Q` point to, respectively, array elements i and j of the same array object `x`, the expression `P - Q` has the value $i - j$. If that value is not in the range of representable values of type `std::ptrdiff_t`, the behavior is undefined.	Yes	Yes	Verify that overflow does not occur when converting to `std::ptrdiff_t`	Coerce into erroneous value
{expr.shift. neg.and. width}	[expr.shift]/1: The behavior is undefined if the right operand is negative, or greater than or equal to the width of the promoted left operand.	Yes	Yes	Insert check whether right operand is valid	Coerce into erroneous value
V. Threading					
{intro.races. data}	[intro.races]/17: Any such data race results in undefined behavior.	Yes	No	Track from which threads memory is accessed and when accesses synchronize with each other; only practical for a subset of cases	Could possibly make all primitive memory accesses implicitly atomic, but that is neither specified nor ready to propose

Identifier	Wording	Runtime-checkable	Locally checkable	Runtime checking strategy	Replacement behaviour
{intro. progress. stops}	[intro.progress]/1: The implementation may assume that any thread will eventually do one of the following: terminate, invoke the function `std::this_thread::yield` ([thread.thread.this]), make a call to a library I/O function, perform an access through a volatile glvalue, perform an atomic or synchronization operation other than an atomic modify-write operation ([atomics.order]), or continue execution of a trivial infinite loop ([stmt.iter.general]).	No, requires (quite literally) solving the halting problem	No	No checking strategy exists as whether a thread will make progress is undecidable	None
VI. Sequencing					
{intro. execution. unsequenced. modification}	[intro.execution]/10: The behavior is undefined if a side effect on a memory location ([intro.memory]) or starting or ending the lifetime of an object in a memory location is unsequenced relative to another side effect on the same memory location, starting or ending the lifetime of an object occupying storage that overlaps with the memory location, or a value computation using the value of any object in the same memory location, and the two evaluations are not potentially concurrent ([intro.multithread]).	Yes	Yes	Identify all potential read operations that are not sequenced with respect to each given write operation; insert checks to identify if those operations are referencing the same address	Sequence operations in some unspecified order

Identifier	Wording	Runtime-checkable	Locally checkable	Runtime checking strategy	Replacement behaviour
VII. Assumptions					
{dcl.attr. assume.false}	[dcl.attr.assume]/1: If the converted expression would evaluate to true at the point where the assumption appears, the assumption has no effect. Otherwise, evaluation of the assumption has runtime undefined behavior.	Not in general, UB is based on the hypothetical result of an arbitrary expression that cannot be evaluated if it might have side effects.	Yes, if checkable at all.	No automatic checking strategy is possible because the predicate cannot be, in general, proven to be free of side effects; instead, the user has to change `[[assume(x)]]` to `contract_assert<may_be_assumed>(x)` and select an appropriate evaluation semantic	Ignore the assumption
VIII. Control Flow					
{basic.start. main.exit. during. destruction}	[basic.start.main]/4: If `std::exit` is invoked during the destruction of an object with static or thread storage duration, the program has undefined behavior.	Yes	No, needs new instrumentation of existing functionality	Track whether static or thread-local objects are currently being destroyed	None

Identifier	Wording	Runtime-checkable	Locally checkable	Runtime checking strategy	Replacement behaviour
{basic.start.term.use.after.destruction}	[basic.start.term]/5: If a function contains a block variable of static or thread storage duration that has been destroyed and the function is called during the destruction of an object with static or thread storage duration, the program has undefined behavior if the flow of control passes through the definition of the previously destroyed block variable.	Yes	No	Track the lifetime of static objects	None
{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	Equivalent to implicitly inserting a `contract_assert(false)` at end of the *function-body*.	As with automatic variables, we can initialize the storage for the return value with erroneous values.
{stmt.dcl.local.static.init.recursive}	[stmt.dcl]/3: If control re-enters the declaration recursively while the variable is being initialized, the behavior is undefined.	Yes	No	Insert a recursion counter into a guard for static and thread-local object construction	None

Identifier	Wording	Runtime-checkable	Locally checkable	Runtime checking strategy	Replacement behaviour
{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)`	Return normally. This would require both the caller and callee translation unit to ignore instead of assume that the function does not return.

IX. Replacement Functions

{basic.stc. alloc.dealloc. constraint}	[basic.stc.dynamic.general]/3: If the behavior of an allocation or deallocation function does not satisfy the semantic constraints specified in [basic.stc.dynamic.allocation] and [basic.stc.dynamic.deallocation], the behavior is undefined.	Partially	No	Add postcondition assertions to allocation functions, evaluate them. Some constraints can be expressed this way, some (such as global distinctness) are not locally verifiable.	None
{expr.new. non.allocating. null}	[expr.new]/22: If the allocation function is a non-allocating form ([new.delete.placement]) that returns null, the behavior is undefined.	Yes	Yes	Insert `post(r: r)`	None

Identifier	Wording	Runtime-checkable	Locally checkable	Runtime checking strategy	Replacement behaviour
			X. Coroutines		
{stmt.return.coroutine.flow.off}	[stmt.return.coroutine]/3: If overload resolution for *p*`.return_void()` succeeds, flowing off the end of a coroutine's *function-body* is equivalent to a `co_return` with no operand; otherwise flowing off the end of a coroutine's *function-body* results in undefined behavior.	Yes	Yes	Insert `contract_assert(false)` at end of *function-body* if no `return_void` function is provided	No, we cannot fabricate an argument of a type suitable to invoke one of possibly many overloads of `return_value` on the promise type.
{dcl.fct.def.coroutine.resume.not.suspended}	[dcl.fct.def.coroutine]/8: Invoking a resumption member function for a coroutine that is not suspended results in undefined behavior.	Yes	No, requires additional instrumentation	Track (in a thread-safe fashion) the suspension state associated with every coroutine handle	None
{dcl.fct.def.coroutine.destroy.not.suspended}	[dcl.fct.def.coroutine]/11: If `destroy` is called for a coroutine that is not suspended, the program has undefined behavior.	Yes	No, requires additional instrumentation	Track (in a thread-safe fashion) the suspension state associated with every coroutine handle	None