Overview and Implementation Report for P3100

Document #:	P4277R0
Date:	2026-08-14
Project:	Programming Language C++
Audience:	SG21 (Contracts)

Reply-to: Joshua Berne <jberne4@bloomberg.net>

Abstract

EWG wishes to review each individual wording change in [P3100R8]; this paper gathers the relevant information about each instance of UB in one place for discussion so they can all be considered individually.

Contents

1 Introduction **2**

2 Individual Undefined Behavior Treatments **4**
- 2.1 Fallback to Erroneous Values 7
- 2.2 Fallback to nothing 45
- 2.3 Leave behavior as undefined 49
- 2.4 Excluded 208

3 Implementation Status **211**
- 3.1 Charts 218

4 Foundational Wording Changes **219**
- 4.1 Implicit Assertions as Part of Undefined Behavior 220
- 4.2 Implicit Assertions Before All Undefined Behavior 225
- 4.3 Common Changes 230

5 Conclusion **232**

Revision History

Revision 0 (August 2026 mailing)[1]:

- Ongoing draft for discussion in telecons.

1 Introduction

[P3100R8] (whose latest revision should be available concurrently with this paper) has been reviewed a number of times, with ever-growing consensus. Due to the widespread nature of the changes it proposes, the most recent poll on what to do next decided that there was interest in reviewing every line of the proposed changes prior to moving forward, to be sure everyone involved has a chance to be aware of the decision being made by adopting this proposal:

EWG, Brno, 2026-06-10, Poll

EWG Approves of the overall direction of [P3100R7], agrees to attend/spend time reviewing every line item in Telecons, and re-consider this in Búzios.

SF	F	N	A	SA
16	15	6	2	0

Result: consensus

This paper, built from the same source as [P3100R8], contains all of the wording changes in that paper along with descriptions of the changes, organized for consumption by category of change instead of the single complete set of wording instructions to the editors that [P3100R8] contains.

This paper is **NOT** a proposal of its own, and is provided as an illuminating companion to [P3100R8]. Any discrepancies between this paper and [P3100R8] or more recent revisions of that paper are unintentional and should be interpreted in the favor of the other paper. Any polls taken to ask for changes to contents to these papers or forward this paper should be targeted at [P3100R8]. Consider this paper and its contents to be a rather elaborate set of slides supporting the content of [P3100R8].[2]

The proposal in [P3100R8] consists of the following major parts:

- The introduction of *implicit contract assertions* that can be associated by the language with other operations, in contrast with *explicit contract assertions* that are introduced syntactically with `pre`, `post`, and `contract_assert`.

[1] An early first draft of this paper was shown at the 2026-07-26 EWG telecon. It contained the general skeleton of the paper and minimal information on the current implementation status. A second draft was shown at the 2026-08-10 EWG telecon. That draft added the local ub descriptions, implementation experience details, example compiler explorer links for each instance of UB, and the alternate wording strategy discussed in Section 4.2. This revision for the mailing has been edited for consistency and clarity and captures further implementation experience that has occurred since the telecon.

[2] If you are seeing a D version of this paper, it is quite likely to not have exactly the same wording as a version of [P3100R8] that exists somewhere other than one of the author's computers. We hope this does not cause undue confusion to our readers, but it is our goal to also avoid adding many highly redundant and mostly similar revisions of the same papers into the permanent WG21 record.

- The introduction of the *assume* semantic that an implicit contract assertion might be evaluated with.
- The introduction of guarding contract assertions that precede all instances of undefined behavior in the language.
  - Having a guarding contract assertion gives implementations a choice of providing a completely well-defined path to specifying how a program behaves when it misuses a core-language construct.
  - Like explicit contract assertions, the evaluation semantic on any given evaluation of an implicit contract assertion is implementation-defined.
- The transformation of a number of behaviors into well-defined behavior with an implicit precondition that they do not occur.
  - When that implicit precondition is assumed, the behavior has no constraints just as it does today.
  - In most cases this is applied to arithmetic operations where we now produce an erroneous value instead of having completely unconstrained behavior.
- [P3100R8] does not require that any particular semantic be available to choose for any given implicit contract assertion. All implementations already implement the *assume* semantic today for all implicit contract assertions.

2 Individual Undefined Behavior Treatments

On the following pages we will go through all of the wording changes in [P3100R8], reordered and annotated for easy review by EWG.

Some notes on the details of the wording presented here:

- The wording changes shown are the same as the wording changes in [P3100R8] — generated from the same preparatory branch revision to the C++ wording draft. They are shown here with additional reference information and context, but are not intended to supersede the contents of that paper.
- In Section 4 we include all of the foundational wording infrastructure that is independent of each instance of undefined behavior. Two alternatives for how guarding contract assertions get introduced are presented.
- In sections following that, we include information about each individual instance of undefined behavior, grouped generally by how [P3100R8] proposes to transform the specification of that behavior.
- For each instance of undefined behavior, we have included a few key items when available:
  - The **Notes** on the instance, identical to those presented in [P3100R8], summarizing whether this instance appears to be runtime-checkable, locally checkable, how it might be checked, and what kinds of fallback behaviors could be provided for it.

    These notes contain entries with the following categories and content:

    * **Summary:** A short description of the undefined behavior category
    * **Runtime-Checkable:** Whether determinining if there is undefined behavior would require universal knowledge of some sort, or if sufficient instrumentation (or no extra instrumentation) would be enough to implement a point-in-time check for this undefined behavior. This does not imply that the check, or the needed instrumentation, would be economical — merely that it would be possible.
    * **Locally-Checkable:** Whether a runtime check could be implemented using information that is available to the abstract machine. Note that this, again, says nothing about whether the check would be cheap or expensive, just whether it is notionally possible.
    * **Checking:** A description of how a check could be implemented.
    * **Replacement:** What form of replacement behavior might be selectable in. Note that replacement behavior might be chosable without paying the cost for checking, or for maintaining instrumentation data if that would be needed to execute a runtime check.
    * **Implementation:** Notes on whether sanitizers or the prototype p3850 implementation have checks that cover some or all of this category of undefined behavior.
  - The **Wording** that contains the definition of this instance of undefined behavior, with the particular point in the standard where the instance is defined highlighted.

- The **UB Annex Entry** that documents the undefined behavior, as gathered and merged into the standard originally by [P3596R3] and updated in [P4284R0].
- An **Implementation Experience** section describing what was involved with prototype implementions on the `p3850-contracts` branches of GCC and Clang, as well as what today's unmodified compilers already do for this category of undefined behavior. Where applicable, this section is broken into up to three parts: ***Existing Practice*** (what a stock sanitizer on an unmodified compiler already catches today, with no contracts feature involved), ***P3850 Sanitizer Integration*** (that same sanitizer integrated with the contract-violation handler), and ***P3850 Implementation*** (the native compiler frontend or middle-end prototype work). Details of what changes were needed to existing checks and code generation to implement the behavior, whether exception propagation could be easily supported, and links to compiler explorer are all provided.
  - The configuration for many of these compiler explorer examples uses the contracts configuration system described in [P3595R0].
  - Some checks do not support throwing violation handlers, and thus can only be configured with nonthrowing semantics. If `-fcontracts-p4298` is specified on the command line then the *`noexcept`-enforce* and *`noexcept`-observe* semantics may be available for these checks, as proposed in [P4298R0].
  - Some forms of UB can be detected by a sanitizer, and when a sanitizer identifies undefined behavior and emits its own diagnostics (or calls its own failure callback) that is, effectively, evaluating the assertion with a checked semantic. Sanitizer-specific callbacks are, effectively, local violation handlers (as described in [P3400R4]) that fully handle the violation. In most cases, the sanitizers do not support exception propagation (and terminate if the callback throws an exception), so it is the *`noexcept`-enforce* and sometimes the *`noexcept`-observe* semantics that the sanitizer provides.
  - Some sanitizers on the `p3850-contracts` branches have been fully integrated with the contract-violation handler. These support a new command-line argument, `-fsanitize-semantic=`. This argument takes a comma-separated list of `<check>:<semantic>` pairs, and will apply the given contract-evaluation semantic to those checks provided by the named sanitizer. We list below all checks that currently work and what language undefined behaviors those checks cover (completely or partially).

    On the integrated path the sanitizer itself emits nothing – no report printed to `stderr`, no diagnostic banner, nothing – so a violation on the integrated path looks, from the outside, like any other implicit assertion: the handler is the only thing that produces output. This matters because sanitizers such as AddressSanitizer normally print a detailed report (the faulting address, a symbolized stack trace, a shadow-memory map) directly to `stderr` and then terminate on their own, bypassing the contract machinery entirely; integration instead means the sanitizer must give that information to the handler rather than print it itself.

    Some sanitizers provide significant additional information when a violation is detected, well beyond what fits in the handful of fields `contract_violation` otherwise exposes. This information can be retrieved by the contract-violation handler using the new

`report()`[3] member function of `contract_violation`, which returns that sanitizer's full diagnostic as a string (or a null pointer if the current violation was not produced by a sanitizer that populates one). The string itself is only built the first time `report()` is called – a producer such as AddressSanitizer registers a callback rather than building the string up front – so a handler that never calls `report()` never pays the cost of rendering it, and the sanitizer never touches `stderr` regardless of whether the handler asks for the report. Both compiler branches integrate AddressSanitizer this way; examples are available for https://godbolt.org/z/1hGedKcj4 and https://godbolt.org/z/rcddMaYWG: AddressSanitizer detects a heap-buffer-overflow, delivers it to the handler as an implicit assertion, and the handler calls `report()` and prints the result itself.

The same integration extends to the UndefinedBehaviorSanitizer *runtime* checks, with the `vptr` (dynamic-type) check as the first one wired – `-fsanitize-semantic=vptr:<semantic>`. A `vptr` violation (a member access, member call, or downcast through a pointer whose runtime type is not the expected one) is genuinely detectable only at run time, so it is configured globally on the command line rather than through per-namespace configuration, exactly as for AddressSanitizer. On the integrated path UndefinedBehaviorSanitizer likewise emits nothing to `stderr`; the handler obtains its full "does not point to an object of type ..." diagnostic on demand through `report()`, and – unlike AddressSanitizer – the violation also carries the real source location of the faulting access. Both compiler branches integrate the vptr check this way; https://godbolt.org/z/6h8cW4e9h and https://godbolt.org/z/P5s4359v8 show the check delivered to the handler as an implicit assertion (the routed sanitizer emits nothing itself; the handler prints its `report()`).

∗ Note that on these compiler branches, all of the implemented proposals on the branch that are targetting C++29 can be enabled with `-fcontracts-p3850`.

[3] We have prepared a small paper proposing this functionality in [P4301R0].

2.1 Fallback to Erroneous Values

When a scalar value is produced by an operation that might have undefined behavior we have the option to fallback to producing an unspecified erroneous value erroneously as our fallback behavior. All of the items in this section do that.

[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: initialise default-initialised variables with erroneous value
implementations:	caught by MemorySanitizer on Clang; also rejected during constant evaluation.

Description

An object's storage can hold an *indeterminate* value – one that was never given a defined value by initialization or assignment. Producing an indeterminate value is not itself undefined behavior (a variable can be left uninitialized without consequence as long as it is never read), but the result of *evaluating* it – reading it as part of any expression other than a narrow set of byte-wise accesses – is.

```
void g() {
  int x [[indeterminate]];
  int y = x;    // undefined behavior: x's value is never read as raw
                // bytes, it is read as an int
}
```

This category is split into two subgroups by the kind of storage that is read:

- `basic.indet.value.attr.indeterminate` – reading an object whose value was never established, such as an `[[indeterminate]]` or default-initialized scalar.
- `basic.indet.value.object.padding` – reading the *padding* bytes of an object whose named members have all been assigned; the padding still holds an indeterminate value even though the object otherwise looks fully written.

Reliably implementing a replacement behavior would require, essentially, suppressing any traps that might ensue from invalid representations being used (for floating-point trapping NaNs, for example) and any optimizations which might be occurring based on those values. Traps, however, can be considered an application of the *quick-enforce* semantic and be left in place.

Wording

Modify section 6.8.5[basic.indet], paragraph 2:

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 erroneous or indeterminate respectively. 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~~ (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 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*]

UB Annex Entry

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 is undefined.

2 [*Example 1*:

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

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept-enforce***	***noexcept-observe***	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept-enforce***	***noexcept-observe***	
`basic.indet.value`		✔								✔					MSan MSan MSan	MSan MSan MSan	
↪ `...attr.indeterminate`	✔	✔							✔	✔					MSan MSan MSan	MSan MSan MSan	CE
↪ `...object.padding`		✔								✔					MSan MSan MSan	MSan MSan MSan	CE

Implementation Experience

Existing Practice

Reading an indeterminate (uninitialized) value is MemorySanitizer's (`-fsanitize=memory`) core

purpose: it tracks, bit for bit, whether every value in memory has been written, and reports as soon as an uninitialized value influences a branch, a system call, or other observable behavior. MSan is Clang/LLVM-only – GCC currently has no MemorySanitizer implementation at all.

MSan provides the equivalent of the *`noexcept`-enforce* semantic when used by default, and provides *`noexcept`-observe* when configured with `-fsanitize-recover=memory`. The p3850 branch integrates with the contract-violation handler and provides *quick-enforce* as well.

Full application of MSan in general requires a complete rebuild of the standard library being used as well, so larger-scale examples that use Standard Library types are not going to work properly in compiler explorer, but could be built locally.

[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).

Description

The expression-evaluation rule is the general catch-all: if evaluating an expression produces a result that is not mathematically defined, or is outside the range of representable values for its type, the behavior is undefined. More specific wording points refine particular operators (for example, division by zero and pointer-difference overflow have their own dedicated rules); the following subgroups cover what is left once those more specific rules are factored out – overflow in the three basic signed integer binary arithmetic operators and in unary negation.

```
int x = std::numeric_limits<int>::max();
int y = x + 1;   // undefined behavior: $2147483647 + 1$ is not representable as int
```

- `expr.expr.eval.signed.integer.add` – overflow evaluating `a + b`
- `expr.expr.eval.signed.integer.sub` – overflow evaluating `a - b`
- `expr.expr.eval.signed.integer.mul` – overflow evaluating `a * b`
- `expr.expr.eval.signed.integer.negate` – overflow evaluating `-a` (notably `-INT_MIN`)

Floating point operations do not result in out-of-range values on readily accessible platforms, so are not represented here as separate subcategories.

Wording

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 it 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 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*]

UB Annex Entry

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 is undefined.

2 [*Example 1*:

```
#include <cstdint>
#include <limits>
int main() {
  std::int32_t x1 = std::numeric_limits<std::int32_t>::max() + 1;
        // undefined behavior, 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 behavior, −2,147,483,648/ − 1 is not representable as a std::int32_t
}
```

— end example]

[*Example 2*: Assuming a `float` type that does not have NaN or infinities, the following operations have undefined behavior because their results are not in the range of `float`.

```
#include <limits>
int main() {
  float x1 = std::numeric_limits<float>::max() * 2; // undefined behavior, infinite value
  float x2 = 0.0f / 0.0f;                           // undefined behavior, NaN
}
```

— end example]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	*`noexcept`-enforce*	*`noexcept`-observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	*`noexcept`-enforce*	*`noexcept`-observe*	
`expr.expr.eval`	✔	✔	P3850	P3850 UBSan			P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	
↪ `...signed.integer.add`	✔	✔	P3850	P3850 UBSan			P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE
↪ `...signed.integer.sub`	✔	✔	P3850	P3850 UBSan			P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE
↪ `...signed.integer.mul`	✔	✔	P3850	P3850 UBSan			P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE
↪ `...signed.integer.negate`	✔	✔	P3850	P3850 UBSan			P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE

Implementation Experience

Existing Practice

UBSan's `signed-integer-overflow` check (part of the `undefined` group, so it is also enabled by plain `-fsanitize=undefined`) catches these cases on both compilers.

The same overflow is also caught during constant evaluation on both compilers.

P3850 Implementation

To enable this check in either compiler, `-fcontracts-p3100` must be specified and the group `ub:expr.expr.eval.signed.integer` given a semantic other than *assume*.

The check covers signed integer `+`, `-`, `*` (and unary `-`, `++`, and `--`). Because it does not cover every arithmetic-evaluation case this instance names, the group id is qualified to name only the subset checked. It is a middle-end check.

When continuing past a violation without terminating – the *ignore* and *noexcept-observe* semantics (and, on Clang, *observe*) – the defined value is the two's-complement wrapped result.

GCC: The check reuses the compiler's existing middle-end overflow instrumentation, so the change was small. Because that instrumentation runs too late to sit in an exception-handling context, the violation handler cannot propagate an exception, so only the nonthrowing semantics are available. The one care needed was to resolve the semantic before inlining, so that it stays correct once the operation is inlined; with that done, *assume* is byte-identical to today's code and still vectorizes, while *ignore* behaves as `-fwrapv`. Examples are available on https://godbolt.org/z/6d8fs74G1.

Clang: Clang emits the check during code generation of the arithmetic operators, in a context where exceptions may be thrown, so it supports all seven semantics, including the potentially-throwing *enforce* and *observe*. Examples are available on https://godbolt.org/z/z3rMaqd5e.

[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. With GCC potentially-throwing semantics are not supported while with Clang they are supported. UBSan's `bool`/`enum` checks (both in the `undefined` group) check both categories. The `int` and `float` subgroups have no invalid representation on commonly available platforms.

Description

An object's *value representation* – the bytes that determine its value – can, through manipulation of the underlying storage directly, end up holding a bit pattern that is not a valid value for the object's type (a `bool` byte that is neither `0` nor `1`, for instance). Performing an lvalue-to-rvalue conversion on such an object – reading it as a value of its declared type – is undefined behavior, even though writing and copying the invalid bytes around is not.

```
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 four subgroups distinguish the type being loaded:

- `conv.lval.valid.representation.bool` – a `bool` whose byte is neither `0` nor `1`
- `conv.lval.valid.representation.enum` – an unscoped enumeration with no fixed underlying type whose value is outside the enumeration's range
- `conv.lval.valid.representation.int` – an integer type
- `conv.lval.valid.representation.float` – a floating-point type

Only the `bool` and `enum` subgroups are undefined behavior on any available platform: both prototypes (and stock UBSan) detect an out-of-range `bool` or `enum` load, but on every platform available to either prototype every bit pattern is a valid value of an integer or floating-point type (no trap representations), so the `int` and `float` loads are well-defined and nothing checks them.

Note that the standard does not explicitly describe the value representations of any of its scalar types, nor that they are unique for any given value. A platform may treat a byte with any number

of non-zero bits set as a `true` boolean value, but it may also require that only the single lowest-order bit be set.

Wording

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 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;          // undefined 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.

UB Annex Entry

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.

2 [*Example 1*:

```
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
}
```

— *end example*]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept-enforce***	***noexcept-observe***	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept-enforce***	***noexcept-observe***	
conv.lval.valid. representation		✔	P3850	P3850 UBSan			P3850 UBSan UBSan	P3850 UBSan UBSan		✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	
↪ ...bool		✔	P3850	P3850 UBSan			P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE
↪ ...enum		✔	P3850	P3850 UBSan			P3850 UBSan UBSan	P3850 UBSan UBSan		✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE
↪ ...int																	N/A
↪ ...float																	N/A

Implementation Experience

Existing Practice

UBSan's `bool` and `enum` checks (both part of the `undefined` group, so both enabled by plain `-fsanitize=undefined`) cover the two halves of this category: `bool` flags an lvalue-to-rvalue load of a `bool` whose stored byte is neither `0` nor `1`, and `enum` flags a load of an unscoped, non-fixed enumeration whose value is outside the enumerated range. Both fire on both compilers.

GCC accepts invalid uses of `std::bit_cast` for enums at compile time, while Clang does not. Constant evaluation catches this category only partially. A `std::bit_cast` (or `__builtin_bit_cast`) producing an object whose value representation is invalid for its type is the constexpr-compatible spelling of the trigger: Clang rejects the `bool` case ("value 42 cannot be represented in type 'bool"') but accepts the `enum` case, and GCC accepts both.

P3850 Implementation

To enable this check in either compiler, `-fcontracts-p3100` must be specified and the group `ub:conv.lval.valid.representation` given a semantic other than *assume*. On GCC the potentially-throwing semantics additionally require `-fcontracts-p4298`; without it the check still fires and terminates under a terminating semantic, but the user's `handle_contract_violation` is not invoked.

The check covers the lvalue-to-rvalue load of a `bool` or non-fixed `enum` whose stored value is out of range; the cast-site spelling `expr.static.cast.enum.outside.range` is deferred. It is a middle-end check.

When continuing past a violation without terminating – the *ignore* and *`noexcept`-observe* semantics (and, on Clang, *observe*) – a defined valid value is substituted (`false` for `bool`, an in-range enumerator for an `enum`).

GCC: Because the check reuses the compiler's existing middle-end `bool`/`enum` load instrumentation (its raw-bits reload and range test), the handler cannot propagate an exception, so only the nonthrowing semantics are available: *assume*, *ignore*, *quick-enforce*, *`noexcept`-observe*, and *`noexcept`-enforce*. Examples are available on https://godbolt.org/z/Kxf1rWsq7.

Clang: Clang emits the guard during the scalar load, suppressing the load's range metadata so the optimizer cannot assume validity and discard the check; because that code generation is in an exception-handling context, it supports all seven semantics. Examples are available on https://godbolt.org/z/rv4hqTYe6.

[conv.double.out.of.range]

summary:	converting a floating-point value to a type that cannot represent it.
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

Description

Converting a floating-point value to another type (floating-point or integer) that cannot represent that value is undefined behavior. Whether a given conversion is actually out of range depends on the target platform's floating-point representation – on a platform whose floating type has no infinity, a magnitude beyond the finite range has nowhere to go; on a platform whose floating type does have infinity (as IEEE-754 does), the same conversion instead produces that representable infinite value and is well-defined.

```
double d = std::numeric_limits<double>::max();
float f = d;    // undefined behavior on a platform whose float cannot
                // represent this value (e.g., has no infinity); well-defined,
                // producing +inf, on a platform whose float can
int i = d;      // undefined behavior: no integer type can represent this
```

Wording

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 that this does not occur (F.3.6[ubx:conv.double.out.of.range]).

UB Annex Entry

F.3.6 **[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 the value has undefined behavior.

2 [*Example 1*: Assume a 32-bit `int`, 32-bit `float`, and a 64-bit `double` for the following example.

```
#include <limits>

int main() {
  double d2 = std::numeric_limits<double>::max();
  float f = d2; // undefined behavior on systems where the range of representable values
                // of float is [-max,+max]; on systems where the range of representable
                // values is [-inf,+inf] this would not be undefined behavior
  int i = d2;   // undefined behavior, the max value of double is not representable as int
}
```

— end example]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**`noexcept`**-*enforce*	**`noexcept`**-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**`noexcept`**-*enforce*	**`noexcept`**-*observe*	
`conv.double.out.of.range`																	N/A

Implementation Experience

Existing Practice

There is no platform available where this is undefined behavior. A floating-to-floating conversion that cannot represent the source value is UB only on a non-IEEE floating type with no infinity; on every platform available to either prototype (IEEE-754 `float`/`double`), an out-of-range conversion rounds to $\pm\infty$, which is representable in the destination type – so the conversion is defined, not UB. Constant evaluation is consistent with this rather than catching anything: the conversion compiles and evaluates to $\pm\infty$ in a constant expression on both compilers, which would not be possible if it were UB here.

[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.

Description

Converting a floating-point value to an integer type is undefined behavior if the value (after truncation toward zero) is not representable in the destination integer type – for example, a value with magnitude larger than the destination type's maximum, or a value that truncates to something outside the destination's range.

```
double d = static_cast<double>(std::numeric_limits<std::int32_t>::max()) + 1;
std::int32_t x = d;   // undefined behavior: not representable as std::int32_t
```

Wording

Modify section 7.3.11[conv.fpint], paragraph 1:

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.3.7[ubx:conv.fpint.float.not.represented]) produced if the truncated value cannot be represented in the destination type, with an implicit precondition that this does not occur (6.11.2+a[basic.contract.implicit]) (F.3.7[ubx:conv.fpint.float.not.represented]).

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

UB Annex Entry

F.3.7 **[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 value is not representable in the destination type, the behavior is undefined.

2 [*Example 1*:

```
#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, 2,147,483,647 + 1 is not representable as std::int32_t
}
```

— end example]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	
`conv.fpint.float.not.represented`	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE

Implementation Experience

Existing Practice

UBSan's `float-cast-overflow` check (part of the `undefined` group in Clang, while separate in GCC) covers a floating value whose truncation toward zero is out of range for the destination integer type, on both compilers. It is float-to-integer only by design. Because this undefined behavior is non-faulting – the conversion yields a garbage value rather than trapping – all three achievable-without-a-handler behaviors are compile-time-selectable: *quick-enforce* via `-fsanitize-trap=float-cast-overflow`, ***noexcept*-enforce** via `-fno-sanitize-recover=float-cast-overflow`, and ***noexcept*-observe** by default. Where it continues, the stock check resumes with the garbage value.

The same conversion is also rejected during constant evaluation on both compilers.

P3850 Implementation

To enable this check in either compiler, `-fcontracts-p3100` must be specified and the group `ub:conv.fpint.float.not.represented` given a semantic other than *assume*.

The check covers a floating value whose truncation toward zero is not representable in the destination integer type (plain integer destinations only, matching both compilers).

When continuing past a violation without terminating – the *ignore*, *observe*, and ***noexcept*-observe** semantics – the conversion is not performed and the defined value 0 is produced in its place.

[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

Description

Converting a value of integer or unscoped enumeration type to a floating-point type is undefined behavior if that value is not representable in the destination floating type.

```
unsigned long long x = -1;   // wraps to the type's maximum value
float f = x;   // undefined behavior on a platform where float cannot
               // represent this value and has no infinity to round to
```

Wording

Modify section 7.3.11[conv.fpint], paragraph 2:

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 that this does not occur (F.3.8[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.

UB Annex Entry

F.3.8 **[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 is undefined.

2 [*Example 1*:

```
int main() {
  unsigned long long x2 = -1;
  float f = x2; // undefined behavior 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]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**`noexcept`**-*enforce*	**`noexcept`**-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**`noexcept`**-*enforce*	**`noexcept`**-*observe*	
`conv.fpint.int.not.represented`																	N/A

Implementation Experience

Existing Practice

There is no platform available where this is undefined behavior. On every available (IEEE) platform, converting an integer to a floating type either produces a finite rounded value or, above the destination type's finite range, rounds to $\pm\infty$ – both representable, so both defined. Constant evaluation compiles the conversion on both compilers rather than rejecting it, consistent with the absence of UB on this target.

[expr.static.cast.enum.outside.range]

summary:	`static_cast`-ing an out-of-range integer 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:	caught by Clang via constant evaluation and UBSan's `enum` check (`-fsanitize=enum`) on the load; and, in the P3850 prototype, by a front-end check at the cast site on both compilers (all available semantics) – closing GCC's coverage gap.

Description

An unscoped enumeration type without a fixed underlying type has a value range determined by the smallest bit-field that can hold all of its enumerators – not just the enumerators' own values. `static_cast`-ing an integer into such an enumeration is well-defined as long as the source value falls within that range; a value outside it is undefined behavior.

```
enum A { e1 = 1, e2 };   // range is [0,3]: the smallest bit-field holding {1,2}

void f() {
  A a = static_cast<A>(4);   // undefined behavior: 4 is outside [0,3]
}
```

Wording

Modify and note section 7.6.1.9[expr.static.cast], paragraph 8:

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 that this does not occur (6.11.2+a[basic.contract.implicit]) (F.3.16[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.

UB Annex Entry

F.3.16 **[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, the behavior is undefined.

2 [*Example 1*:

```
enum A { e1 = 1, e2 };

void f() {
  enum A a = static_cast<A>(4); // undefined behavior, 4 is not within the range of enumeration values
}
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**`noexcept`**-*enforce*	**`noexcept`**-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**`noexcept`**-*enforce*	**`noexcept`**-*observe*	
`expr.static.cast.enum.outside.range`		✔	P3850	P3850	P3850	P3850	P3850	P3850	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE

Implementation Experience

Existing Practice

This is real undefined behavior on this target, and it shows a clean GCC/Clang divergence: **GCC has zero coverage** (its constant evaluator accepts an out-of-range unscoped-enum `static_cast` with no diagnostic, and `-fsanitize=enum` emits nothing for the same value), while **Clang catches it two ways**. Clang's constant evaluator rejects the cast ("not an integral constant expression"), and stock `-fsanitize=enum` (part of the `undefined` group) fires on the subsequent lvalue-to-rvalue *load* of the invalid enum value rather than on the cast itself. This is a deterministic check of the whole category, not a heuristic one.

P3850 Implementation

To enable this check in either compiler, `-fcontracts-p3100` must be specified and the group `ub:expr.static.cast.enum.outside.range` given a semantic other than *assume*.

The check is a front-end guard at the integer-to-enumeration conversion site – the same class of check as divide-by-zero, shift, and the float-to-integer conversion – so the semantic is resolved once, at the cast, and all seven semantics are supported, including throwing *enforce*/*observe*. It applies to a `static_cast` (or C-style/functional cast) whose target is an unscoped enumeration with no fixed underlying type and whose source is integral; a fixed-underlying-type enum, or a floating-point source, is converted through the underlying type and is not undefined behavior. The check fires only

when the enumeration's value range leaves some representable values invalid (when the range spans the whole storage there is nothing to check).

This closes the GCC coverage gap noted above: GCC catches an out-of-range unscoped-enum cast in no channel today, and here it gains a check across every checking semantic; on Clang it complements the existing load-site sanitizer by catching the violation at the cast itself. The erroneous-behavior substitute is the defined value 0 (always within an enumeration's value range): *ignore* silently coerces the result to 0, and *observe*/***noexcept*-observe** report and then continue with 0, while *enforce*/*quick-enforce*/***noexcept*-enforce** stop. The violation is reported as `assertion_kind::implicit`.

GCC: `build_implicit_enum_cast_check` (`cp/contracts.cc`), hooked at the enumeration-conversion site in `ocp_convert` (`cp/cvt.cc`) alongside the float-cast check; the value range is `TREE_TYPE (enumtype)`'s. Examples of every semantic are on https://godbolt.org/z/nd7MfW7aK.

Clang: the guard is emitted in `EmitScalarConversion` (`CGExprScalar.cpp`) via `EmitImplicitIntOpGuard`, using the `EnumDecl` value range; examples on https://godbolt.org/z/99a3z1oxs.

[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

Description

An explicit `static_cast` of a floating-point value to another floating type that is outside the target type's range is undefined behavior, on a platform-dependent basis (whether the target type has an infinity to round to).

```
void f() {
  double d = std::numeric_limits<float>::max();
  d *= 16;
  float f = static_cast<float>(d);   // undefined behavior on a platform
                                     // whose float cannot represent this
}
```

Wording

Modify and note section 7.6.1.9[expr.static.cast], paragraph 9:

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 that this does not occur (6.11.2+a[basic.contract.implicit]) (F.3.17[ubx:expr.static.cast.fp.outside.range]).

UB Annex Entry

F.3.17 **[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.

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
}
```

— end example]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**`noexcept`**-*enforce*	**`noexcept`**-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**`noexcept`**-*enforce*	**`noexcept`**-*observe*	
`expr.static.cast.fp.outside.range`																	N/A

Implementation Experience

Existing Practice

There is no platform available where this is undefined behavior. An out-of-range explicit floating-to-floating conversion rounds to $\pm\infty$ on every available IEEE platform, a representable destination value, so it is defined rather than UB. Constant evaluation compiles the conversion on both compilers instead of rejecting it, consistent with the absence of UB here. `-fsanitize=float-cast-overflow` does not apply (float-to-*integer* only); no other stock sanitizer targets this conversion.

[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:	the integer cases are caught by UBSan's `integer-divide-by-zero` check, constant evaluation, and the P3850 prototype (both compilers, all semantics); floating-point division by zero is caught by UBSan's `float-divide-by-zero` check, which is not part of the `undefined` group, and constant evaluation.

Description

Dividing (or taking the remainder of) a value by zero is undefined behavior. The wording point and its own annex example describe this generally, for both integer and floating-point division.

```
int x = 1 / 0;          // undefined behavior
double d = 1.0 / 0.0;   // undefined behavior
```

- `expr.mul.div.by.zero.int.div` – integer division by zero (`a / 0`)
- `expr.mul.div.by.zero.int.mod` – integer remainder by zero (`a \% 0`)
- `expr.mul.div.by.zero.fp.div` – floating-point division by zero (`a / 0.0`)

Floating-point division by zero is undefined behavior just as the integer cases are: the rule's "second operand is zero" text is unconditional, not restricted to integer operands. This holds even on an IEEE platform, where the hardware would otherwise yield an infinity or a NaN.

Wording

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 that ~~undefined~~this does not occur (F.3.29[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 of both `a/b`~~operator yields an unspecified erroneous value and ~~`a%b`~~there is ~~undefined~~an implicit precondition that this does not occur (6.11.2+a[basic.contract.implicit]) (F.3.30[ubx:expr.mul.representable.type.result]).

UB Annex Entry

F.3.29 **[ub:expr.mul.div.by.zero]**
Specified in: 7.6.5[ubx:expr.mul.div.by.zero]

Division by zero has undefined behavior.

2 [*Example 1*:

```
int main() {
  int x = 1 / 0;        // undefined behavior
  double d = 1.0 / 0.0; // undefined behavior
}
```

— *end example*]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-*enforce***	***noexcept*-*observe***	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-*enforce***	***noexcept*-*observe***	
`expr.mul.div.by.zero`	✔	✔							✔	✔							
↪ ... `int.div`	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE
↪ ... `int.mod`	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE
↪ ... `fp.div`	✔	✔		UBSan			UBSan UBSan	UBSan UBSan	✔	✔		UBSan			UBSan UBSan	UBSan UBSan	CE

Implementation Experience

Existing Practice

UBSan's `integer-divide-by-zero` check (part of the `undefined` group) covers integer `/` and `\%` by a zero divisor on both compilers.

Floating-point division by zero is covered by UBSan's separate `float-divide-by-zero` check. Because such a division yields an infinity or a NaN rather than trapping on IEEE hardware, that check is *not* part of the `undefined` group and must be requested explicitly.

Both the integer and floating-point cases are also rejected during constant evaluation on both compilers, independent of `-fcontracts-p3100`, `-fsanitize=`, or any evaluation semantic.

P3850 Implementation

To enable this check in either compiler, `-fcontracts-p3100` must be specified and the group `ub:expr.mul.div.by.zero` given a semantic other than *assume*.

The check covers integer `/` and `\%` only; it does not cover floating-point division. Floating-point `a / 0.0` is instead handled through the `float-divide-by-zero` sanitizer described above, which the P3850 branches route to the contract-violation handler.

When continuing past a violation without terminating – the *ignore*, *observe*, and *noexcept-observe* semantics – the division is not performed and the defined value 0 is produced in its place.

GCC: The GCC prototype supports all seven evaluation semantics via a new front-end guard on the division operators that skips the trapping operation and runs the reaction when the divisor is zero. Examples of every semantic are available on https://godbolt.org/z/61bonTsna.

Clang: The Clang prototype likewise supports all available semantics, emitting the guard during code generation of the division operators. Examples of every semantic are available on https://godbolt.org/z/b5EKh4fd3.

[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).

Description

Integer division and remainder are related by the identity `(a/b)*b + a\%b == a`. When the quotient `a/b` is not representable in the result type – the classic case being the most negative value of a signed integer type divided by `-1` – that identity cannot hold, so both `a/b` and `a\%b` are undefined behavior, even though the remainder's mathematical value (`0`) would itself be representable.

```
int x = std::numeric_limits<std::int32_t>::min() / -1;   // undefined behavior
int y = std::numeric_limits<std::int32_t>::min() % -1;   // undefined behavior
```

- `expr.mul.representable.type.result.int.div` – the division `a/b`
- `expr.mul.representable.type.result.int.mod` – the paired remainder `a\%b`

These two are exhaustive of this wording point: the rule speaks of exactly this one pair of related operators, and both are covered.

Wording

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 that ~~undefined~~this does not occur (F.3.29[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 of both a/b~~operator yields an unspecified erroneous value and ~~a%b~~there is ~~undefined~~an implicit precondition that this does not occur (6.11.2+a[basic.contract.implicit]) (F.3.30[ubx:expr.mul.representable.type.result]).

UB Annex Entry

F.3.30 **[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` is undefined.

2 [*Example 1*: 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` on such platforms.

```
#include <cstdint>
#include <limits>

int main() {
  std::int32_t x = std::numeric_limits<std::int32_t>::min() / -1;   // undefined behavior (F.3.1[ub:expr.expr.
eval])
  std::int32_t y = std::numeric_limits<std::int32_t>::min() % -1;   // undefined behavior
}
```

— *end example*]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	
`expr.mul.representable.type.result`	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	
↪ ...`int.div`	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE
↪ ...`int.mod`	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE

Implementation Experience

Existing Practice

UBSan's `signed-integer-overflow` check also covers the division-site overflow `INT_MIN / -1` (and `INT_MIN \% -1`) on both compilers.

The overflow is also rejected during constant evaluation on both compilers.

P3850 Implementation

To enable this check in either compiler, `-fcontracts-p3100` must be specified and the group `ub:expr.mul.representable.type.result` given a semantic other than *assume*.

This check guards the division-site case `INT_MIN / -1` (and `INT_MIN \% -1`) for signed `/` and `\%` – exactly the quotient-not-representable undefined behavior this instance names – so the group id is the unqualified `ub:expr.mul.representable.type.result`. The general signed `+`, `-`, `*` overflow is a separate instance, F.3.1[ub:expr.expr.eval].

When continuing past a violation without terminating – the *ignore*, *observe*, and *noexcept-observe* semantics – the division is not performed and the defined value 0 is produced in its place.

GCC: The GCC prototype supports all seven evaluation semantics. The check guards the single overflowing case (`INT_MIN / -1`), including the throwing *enforce* and *observe* semantics alongside the nonthrowing ones. Examples of every semantic are available on https://godbolt.org/z/xEbjKhrWd.

Clang: The Clang prototype likewise supports all available semantics. Examples of every semantic are available on https://godbolt.org/z/4GnrvaxWr.

[expr.sub.pointers.representable]

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

Description

Subtracting two pointers into the same array produces the difference between their indices, as a `std::ptrdiff_t`. If that difference is too large to represent in `std::ptrdiff_t`, the behavior is undefined. As the annex itself notes, reaching this case at all requires a single object whose size exceeds `PTRDIFF_MAX` – a size most platforms cannot construct.

```
char big[static_cast<std::size_t>(std::numeric_limits<std::ptrdiff_t>::max()) + 1];
std::ptrdiff_t d = std::end(big) - std::begin(big);   // undefined behavior
```

Wording

Modify and note section 7.6.6[expr.add], paragraph 5:

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 the value $i-j$ 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 that this does not occur (F.3.32[ubx:expr.sub.pointers.representable]).

(5.3) — Otherwise, the behavior is undefined (F.3.33[ubx:expr.add.sub.diff.pointers]).

UB Annex Entry

F.3.32 **[ub:expr.sub.pointers.representable]**

Specified in: 7.6.6[ubx:expr.sub.pointers.representable]

Subtracting pointers when the result is too large to represent in `std::ptrdiff_t` has undefined behavior.

2 [*Example 1*: Note that 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 <limits>
#include <iterator>
```

```
char big[static_cast<std::size_t>(std::numeric_limits<std::ptrdiff_t>::max())+1];
std::ptrdiff_t d = std::end(big) - std::begin(big);      // undefined behavior
```

— *end example*]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	
`expr.sub.pointers.representable`		✔								✔							CE

Implementation Experience

Existing Practice

This is real UB, but it is not reproducible on any platform we have ready access to; the annex itself notes that "platforms that allow this are rare." On readily available platforms, it is impossible to create single objects whose size exceeds the maximum value of `ptrdiff_t`, and any subtraction that would overflow is already undefined behavior for subtracting pointers that do not point to the same underlying storage.

[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).

Description

Shifting a value by a negative amount, or by an amount greater than or equal to the bit width of its type, is undefined behavior – there is no well-defined meaning for shifting further than the value's own representation.

```
std::uint32_t y  = std::uint32_t(1) << -1;   // undefined behavior: negative shift
std::uint32_t y1 = std::uint32_t(1) << 32;   // undefined behavior: shift == bit width
std::uint32_t y2 = std::uint32_t(1) >> 32;   // undefined behavior: shift == bit width
```

- `expr.shift.neg.and.width.shl` – left shift, `a << b`
- `expr.shift.neg.and.width.shr` – right shift, `a >> b`

These two are exhaustive: every shift expression is either a left shift or a right shift, and both are covered.

Wording

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.3.35[ubx:expr.shift.neg.and.width]) ~~if~~an implicit precondition that the right operand is ~~negative, or greater~~nonnegative and less than ~~or equal to~~ the width of the promoted left operand (6.11.2+a[basic.contract.implicit]) (F.3.35[ubx:expr.shift.neg.and.width]) ; if not, an unspecified erroneous value is produced.

UB Annex Entry

F.3.35 **[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 has undefined behavior.

2 [*Example 1*: In the following example, assume that `std::uint32_t` is supported (17.4.1[cstdint.syn]).

```
#include <cstdint>
std::uint32_t y = std::uint32_t(1) << -1;   // undefined behavior, shift is negative
std::uint32_t y1 = std::uint32_t(1) << 32;  // undefined behavior, shift is equal to bit width
std::uint32_t y2 = std::uint32_t(1) >> 32;  // undefined behavior, shift is equal to bit width
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	
`expr.shift.neg.and.width`	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	
↪ ...`shl`	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE
↪ ...`shr`	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE

Implementation Experience

Existing Practice

UBSan's `shift` check (part of the `undefined` group) covers `<<` and `>>` with a negative or too-large shift amount on both compilers. This undefined behavior is non-faulting – it produces a garbage value rather than trapping – so all three achievable-without-a-handler behaviors are compile-time-selectable: *quick-enforce* via `-fsanitize-trap=shift`, ***noexcept*-enforce** via `-fno-sanitize-recover=shift`, and ***noexcept*-observe** by default. Where it continues, the stock check resumes with the garbage value; the P3850 check substitutes a defined 0. None of this involves any contract-violation handler.

The same shift is also rejected during constant evaluation on both compilers, independent of `-fcontracts-p3100`, `-fsanitize=`, or any evaluation semantic.

P3850 Implementation

To enable this check in either compiler, `-fcontracts-p3100` must be specified and the group `ub:expr.shift.neg.and.width` given a semantic other than *assume*.

The check covers `<<` and `>>` when the shift amount is negative or at least the width of the promoted left operand.

When continuing past a violation without terminating – the *ignore*, *observe*, and *noexcept-observe* semantics – the shift is not performed and the defined value 0 is produced in its place.

GCC: The GCC prototype supports all seven evaluation semantics, via a new front-end guard on the shift operators. Examples of every semantic are available on https://godbolt.org/z/hM1xPenzs.

Clang: The Clang prototype likewise supports all available semantics, emitting the guard during code generation of the shift operators. Examples of every semantic are available on https://godbolt.org/z/YbMsrMb49.

[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).

Description

A function whose return type is not `void` must return a value along every path that can be executed. Reaching the closing brace without having executed a `return` statement – other than in `main` or a coroutine, both of which have their own defined behavior for this case – is undefined behavior, since there is no value to give back to the caller.

```
int f(int x) {
  if (x)
    return 1;
  // undefined behavior if x is 0: falls off the end with no return
}
```

Wording

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]) .

UB Annex Entry

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 results in undefined behavior if the return type is not *cv* `void`.

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
}
```

— end example]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	
`stmt.return.flow.off`	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan	CE

Implementation Experience

Existing Practice

UBSan's `return` check (part of the `undefined` group) instruments the end of a value-returning function and fires when control flows off it without a `return`, on both compilers. Like `unreachable`, this check has no recoverable variant – `-fsanitize-recover=return` is rejected at compile time on both compilers – so only two of the achievable-without-a-handler behaviors apply, both compile-time-selectable: *quick-enforce* (`-fsanitize-trap=return`, a silent trap) and ***noexcept*-enforce** (the default, a diagnostic then terminate). Neither involves any contract-violation handler.

P3850 Implementation

To enable this check in either compiler, `-fcontracts-p3100` must be specified and the group `ub:stmt.return.flow.off` given a semantic other than *assume*.

The check is synthesized at the end of a value-returning function and fires when control would flow off that end.

When continuing past a violation without terminating – the *ignore*, *observe*, and ***noexcept*-observe** semantics – the function returns a byte-zeroed result object of its return type (a zero scalar, or memset-cleared storage for a class or array return, including the sret case) with no data leak. The violation is reported as `assertion_kind::implicit`. A compiler could provide the option to initialize this erroneous data with other values if it so chose.

GCC: The GCC prototype supports all seven evaluation semantics; the check is synthesized in the front end at the end of the function. A function-try-block is handled specifically: a throwing *observe* or *enforce* reaction emitted when the try body falls off is caught by the construct's own handler. Examples of every semantic are available on https://godbolt.org/z/M1xhYe8qv.

Clang: The Clang prototype likewise supports all available semantics, emitting the reaction during code generation of the function body. Examples of every semantic are available on https://godbolt.org/z/eer6xxdE8.

2.2 Fallback to nothing

In one instance we have the option for an attribute to have no effect as our fallback behavior — an `[[assume]]` attribute introduces an implicit contract assertion that it is not violated. Its fallback is to do nothing (the assumption is simply dropped); and when the predicate is side-effect-free, it can additionally be *evaluated*, so the full set of checking semantics is available for it as well.

[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; the opaque `.nonpure` subgroup supports only *assume* and *ignore*.

Description

The `[[assume(expr)]]` attribute tells the optimizer that `expr` is always true at that point in the program, without generating any code to check it. If control ever reaches the assumption while `expr` is actually false, the behavior is undefined – the program has broken a promise the compiler was entitled to rely on.

```
int g(int x) {
  [[assume(x >= 0)]];
  return x / 32;
}

int f() {
  return g(-10);    // undefined behavior: the assumption in g is false
}
```

There are two distinct sub-categories of this form of UB, depending on the compiler's understanding of the nature of the expression:

- `dcl.attr.assume.false.pure` – the assumption's predicate is side-effect- free and non-trapping (a constant, a simple expression, or a call to a `[[gnu::pure]]`/`[[gnu::const]]` function), so it can be evaluated at run time without changing the program's behavior; in such cases the compiler is free to evaluate the predicate to determine its result, and hence every semantic is viable
- `dcl.attr.assume.false.nonpure` – the predicate is *opaque*: evaluating it might have observable side effects, so it cannot be checked without changing the program, and only *assume* and *ignore* apply (a configured checking semantic clamps to *ignore*)

Wording

Modify section 9.13.3[dcl.attr.assume], paragraph 1:

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]).

UB Annex Entry

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 is undefined.

2 [*Example 1*:

```
int g(int x) {
  [[assume(x >= 0)]];
  return x/32;
}

int f() {
  return g(-10);    // undefined behavior, assumption in g is false
}
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	*noexcept-enforce*	*noexcept-observe*	*constexpr*	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	*noexcept-enforce*	*noexcept-observe*	
`dcl.attr.assume.false`		✔	P3850							✔	P3850						
↪ `...pure`	✔	✔	P3850	P3850	P3850	P3850	P3850	P3850	✔	✔	P3850	P3850	P3850	P3850	P3850	P3850	CE
↪ `...nonpure`		✔	P3850							✔	P3850						CE

Implementation Experience

P3850 Implementation

To enable this configurability in either compiler, `-fcontracts-p3100` must be specified and the group `ub:dcl.attr.assume.false` given a semantic other than *assume*. It is *not* gated on

`-fcontracts-allow-assume`: honoring an `[[assume]]` is the status quo and introduces no new undefined behavior.

Which semantics are available depends on the predicate. A side-effect-free, non-trapping predicate – a constant, a simple expression, or a call to a `[[gnu::pure]]` or `[[gnu::const]]` function – can be evaluated at run time, so every semantic is available; this subset is named with the qualified group `ub:dcl.attr.assume.false.pure`. An opaque predicate, whose evaluation might have side effects, or a predicate with obvious side effects, cannot be freely evaluated, so only *assume* and *ignore* apply.

During constant evaluation, compilers are given the same freedom to determine the results of the predicate of `[[assume]]`, and both compilers do so.

GCC: The front end resolves the semantic and either emits the status-quo optimizer hint, drops it, or emits an explicit test of the predicate followed by the reaction. A predicate is treated as checkable when it has no side effects and cannot trap. Examples spanning the predicate spectrum are on https://godbolt.org/z/qev9sPT1v.

Clang: Clang does the same during code generation of the `[[assume]]` statement, using its own side-effect analysis to decide checkability. Because that code generation is in an exception-handling context, the throwing *enforce* and *observe* semantics are supported directly. Examples are on https://godbolt.org/z/aMEPKMq7P.

2.3 Leave behavior as undefined

In the vast majority of cases there is no viable fallback behavior and so we simply rely on the updated definition of undefined behavior to introduce the needed implicit contract assertion and make no local changes to the wording.

[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

Description

Certain operations (such as `malloc`) are specified as implicitly creating objects of implicit-lifetime type in the storage they return, but only to the extent needed to give the program defined behavior. If no choice of implicitly-created objects would make the program's subsequent use of that storage well-defined – for example, because it is used as two incompatible types at once – the behavior is undefined.

```
void f() {
  void *p = malloc(sizeof(int) + sizeof(float));
  *reinterpret_cast<int*>(p) = 0;
  *reinterpret_cast<float*>(p) = 0.0f;  // undefined behavior: no set of
                                        // implicitly-created objects lets
                                        // both accesses be well-defined
}
```

Wording

Notes regarding 6.8.2[intro.object], paragraph 13:

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*]

UB Annex Entry

F.2.1 **[ub:intro.object.implicit.create]**

Specified in: 6.8.2[ubx:intro.object.implicit.create]

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[basic.

types]) 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.

2 [*Example 1*:

```
void f()
{
  void *p = malloc(sizeof(int) + sizeof(float));
  *reinterpret_cast<int*>(p) = 0;
  *reinterpret_cast<float*>(p) = 0.0f;  // undefined behavior, cannot create
                                        // both int and float in same place
}
```

— *end example*]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	
intro.object.implicit.create		✔								✔							CE

Implementation Experience

Existing Practice

No tool covers this category, on either compiler. Checking for this undefined behavior at runtime would require tracking the set of potential objects that *might* exist at a given storage location, and then erroring if conflicting operations reduce that set of potential types to an empty set.

Constant evaluation does not admit errors of this type due to the lack of `reinterpret_cast`.

[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

Description

Once objects have been implicitly created in a region of storage, some operations produce a pointer to one of those objects – specifically, the one whose address is the start of the region – but only if that pointer value gives the program defined behavior. If no such pointer value would, the behavior is undefined.

```
struct X {
  int a, b;
  ~X() = delete;      // deleted destructor: X is not an implicit-lifetime type
};

X* make_x() {
  X* p = (X*)std::malloc(sizeof(X));   // cannot implicitly create an X here
  p->a = 1;    // undefined behavior: no implicitly-created object exists
  return p;
}
```

- `intro.object.implicit.pointer.deleted.dtor` – the pointer's static type is not an implicit-lifetime type at all, so no object can ever be implicitly created there
- `intro.object.implicit.pointer.type.pun` – an implicitly-created object of one type already occupies the storage, and it is accessed through a pointer to a different, not-similar type

These two are not claimed to be exhaustive: they are the two shapes the annex illustrates with its own examples, not a derivation of every way this rule's pointer-selection step can fail to produce a usable value.

Wording

Notes regarding 6.8.2[intro.object], paragraph 14:

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.

UB Annex Entry

F.2.2 **[ub:intro.object.implicit.pointer]**

Specified in: 6.8.2[ubx:intro.object.implicit.pointer]

After implicitly creating objects within a specified region of storage, some operations are described as producing a pointer to a suitable created object (6.9[basic.types]). 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.

2 [*Example 1*:

```
#include <cstdlib>
struct X {
  int a, b;
  ~X() = delete;        // deleted destructor makes X a non-implicit-lifetime class
};

X* make_x() {
  X* p = (X*)std::malloc(sizeof(struct X));     // Cannot implicitly create an object because X
                                                // is not an implicit-lifetime class.
  p->a = 1;     // undefined behavior, no set of objects give us defined behavior
  return p;
}
```

— *end example*]

[*Example 2*:

```
#include <cstdlib>

struct X {
  int a;
};

struct Y {
  int x;
};

Y* make_y() {
  X* p1 = (X*)std::malloc(sizeof(struct X));    // object #1
  p1->a = 1;
  Y* p2 = (Y*)p1;
  p2->x = 2;    // undefined behavior, p2 points to an in-lifetime object of type X,
                // which is not similar to Y (7.6.1.5[expr.ref])
  return p2;
}
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-*enforce***	***noexcept*-*observe***	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-*enforce***	***noexcept*-*observe***	
`intro.object.implicit.pointer`		✔								✔							
↪ ...`deleted.dtor`		✔								✔							CE
↪ ...`type.pun`		✔								✔							CE

Implementation Experience

Existing Practice

Two shapes of this category were examined, matching the standard's two examples, and no tool covers either on either compiler, in any channel. Constant evaluation rejects each trigger only because each needs `std::malloc` or a similar tool, along with the use of `reinterpret_cast`.

[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); and, in the P3850 prototype, a native middle-end check on both compilers.

Description

Every object must be created in storage whose address satisfies its type's alignment requirement. Placing an object (for example, via placement `new`) at an address that does not meet that requirement is undefined behavior, regardless of how the misaligned address was obtained.

```
struct alignas(4) S {};

void make_misaligned() {
  alignas(S) char s[sizeof(S) + 1];
  new (&s + 1) S();   // undefined behavior: &s+1 is one byte past a
                      // 4-aligned address, so it cannot itself be 4-aligned
}
```

Wording

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]).

UB Annex Entry

F.2.3 **[ub:basic.align.object.alignment]**

Specified in: 6.8.3[ubx:basic.align.object.alignment]

All instances of a type must be created in storage that meets the alignment requirement of that type.

2 [*Example 1*:

```
struct alignas(4) S {};

void make_misaligned()
{
    alignas(S) char s[sizeof(S) + 1];
```

```
    new (&s+1) S();     // undefined behavior, &s+1 will yield a pointer to
                        // a char which is 1 byte away from an address with
                        // an alignment of 4 and so cannot have an alignment of 4
}
```

— end example]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	
basic.align.object.alignment		✔	P3850	P3850 UBSan			P3850 UBSan UBSan	P3850 UBSan UBSan		✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE

Implementation Experience

Existing Practice

UBSan's `alignment` check (part of the `undefined` group, so it is also enabled by plain `-fsanitize=undefined`) instruments every pointer dereference and reference binding with an alignment requirement, and fully covers this category on both compilers.

This behavior is structurally prevented during constant evaluation because the casts that would result in a misaligned pointer or reference in the first place are not allowed during constant evaluation.

P3850 Implementation

To enable this check in either compiler, `-fcontracts-p3100` must be specified and the group `ub:basic.align.object.alignment` given a semantic other than *assume*.

This is a middle-end check that reuses the compilers' existing null/alignment instrumentation. Like the null-dereference check it is therefore restricted to the non-throwing semantics: a configured throwing *enforce*/*observe* is clamped to the corresponding `noexcept` variant under `-fcontracts-p4298`, else to *quick-enforce*. A misaligned access is an lvalue with no defined substitute, so *ignore* is the raw access and *observe*/***noexcept*-observe** report and then proceed into the still-misaligned access (which, on a target that tolerates misaligned scalar accesses, completes; the point is that the violation has been reported). The violation is reported as `assertion_kind::implicit`.

GCC: the alignment sub-condition of `IFN_UBSAN_NULL` carries its own reaction, resolved for `ub:basic.align.object.alignment` (`instrument_mem_ref`/`ubsan_expand_null_ifn`, `gcc/ubsan.cc`), independent of the null-dereference reaction; the expander routes the null and misaligned edges to separate then-blocks, so a null-dereference contract and an alignment contract at the same access

each fire with their own semantic (enabling one never disables the other). Example on https://godbolt.org/z/hT9d1YK3K.

Clang: `EmitImplicitMisalignedGuard` (`CGContracts.cpp`) emits an independent `if ((ptr & (align-1)) != 0) <reaction>` at each load/store type check (`EmitTypeCheck`, `CGExpr.cpp`), so the null and alignment contracts can both apply at one site; Clang's codegen is in a valid EH context, so it supports all seven semantics directly (no `-fcontracts-p4298` needed for the throwing forms). Example on https://godbolt.org/z/zj5rEanxh.

[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:	caught by constant evaluation on Clang.

Description

This rule is one of a family of related rules governing what a *pointer* to an object may be used for once that object's lifetime has ended. This specific rule covers using such a pointer as the operand of a *delete-expression*, when the object was (or will be) of a class type with a non-trivial destructor: the delete-expression would call that destructor a second time.

```
struct S {
  float f = 0;
  ~S() {}
};

void f() {
  S* p = new S;
  p->~S();
  delete p;    // undefined behavior: lifetime has already ended, and S
               // has a non-trivial destructor
}
```

Wording

Notes regarding 6.8.4[basic.life], paragraph 7:

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]).

UB Annex Entry

F.2.4 [ub:lifetime.outside.pointer.delete]

Specified in: 6.8.4[ubx:lifetime.outside.pointer.delete]

For a pointer pointing to an object outside of its lifetime, behavior is undefined if the object will be or was of a class type with a non-trivial destructor and the pointer is used as the operand of a *delete-expression*.

2 [*Example 1*:

```
struct S {
  float f = 0;
  ~S() {}
};

void f() {
  S* p = new S;
  p->~S();
  delete p;     // undefined behavior, operand of delete, lifetime has ended and S
                // has a non-trivial destructor
}
```

— end example]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**`noexcept`**-*enforce*	**`noexcept`**-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**`noexcept`**-*enforce*	**`noexcept`**-*observe*	
`lifetime.outside.pointer.delete`		✔							✔	✔							CE

Implementation Experience

Existing Practice

Deleting an object whose lifetime has already ended (a double `delete`-expression) is rejected during constant evaluation on Clang (GCC accepts it there), but no stock runtime sanitizer catches it – the single `operator delete` call is not a double-free, so ASan and UBSan stay silent.

[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:	caught by constant evaluation on both GCC and Clang.

Description

Using a pointer to an object whose lifetime has ended to access one of the object's non-static data members, or to call one of its non-static member functions, is undefined behavior.

```
struct S {
  float f = 0;
};

float f() {
  S s;
  S* p = &s;
  s.~S();
  return p->f;   // undefined behavior: accessing a non-static data member
                 // after the object's lifetime has ended
}
```

Wording

Notes regarding 6.8.4[basic.life], paragraph 7:

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]).

UB Annex Entry

F.2.5 [ub:lifetime.outside.pointer.member]

Specified in: 6.8.4[ubx:lifetime.outside.pointer.member]

For a pointer pointing to an object outside of its lifetime, behavior is undefined if the pointer is used to access a non-static data member or call a non-static member function of the object.

2 [*Example 1*:

```
struct S {
  float f = 0;
};

float f() {
  S s;
  S* p = &s;
  s.~S();
  return p->f;  // undefined behavior, accessing non-static data member after end of lifetime
}
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept-enforce***	***noexcept-observe***	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept-enforce***	***noexcept-observe***	
`lifetime.outside.pointer.member`	✔	✔							✔	✔							CE

Implementation Experience

Existing Practice

Accessing a data member through a pointer to an object outside its lifetime (automatic storage, no heap free) is rejected during constant evaluation on both compilers; no stock runtime sanitizer catches it (the storage is still live).

[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:	heuristically caught by UBSan's `vptr` check (detection relies on the ended object's vtable still being recognizably invalid) and exhaustively caught by constant evaluation, on both GCC and Clang.

Description

Implicitly converting a pointer to an object whose lifetime has ended into a pointer to a virtual base class (or a base of a virtual base) is undefined behavior – that conversion needs to consult the object's virtual-base layout information, which is no longer guaranteed to be valid once the object's lifetime has ended.

```
struct B {};
struct D : virtual B {};

void f() {
  D d;
  D* p = &d;
  d.~D();
  B* b = p;    // undefined behavior
}
```

Wording

Notes regarding 6.8.4[basic.life], paragraph 7:

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]).

UB Annex Entry

F.2.6 [ub:lifetime.outside.pointer.virtual]

Specified in: 6.8.4[ubx:lifetime.outside.pointer.virtual]

For a pointer pointing to an object outside of its lifetime, behavior is undefined if the pointer is implicitly converted (7.3.12[conv.ptr]) to a pointer to a virtual base class (or base class of a virtual base class).

2 [*Example 1*:

```
struct B {};
struct D : virtual B {};
void f() {
  D d;
  D* p = &d;
  d.~D();
  B* b = p;     // undefined behavior
}
```

— *end example*]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-*enforce***	***noexcept*-*observe***	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-*enforce***	***noexcept*-*observe***	
`lifetime.outside.pointer.virtual`	✔	✔					UBSan† UBSan†	UBSan†	✔	✔					UBSan† UBSan†	UBSan†	CE

Implementation Experience

Existing Practice

Binding a pointer to a virtual base of an object outside its lifetime is inherently polymorphic (the conversion reads the object's vtable for the base offset). UBSan's `vptr` check catches it on both compilers, integrated with the handler (`-fsanitize-semantic=vptr`, both ***noexcept*-*enforce*** and ***noexcept*-*observe***; no *quick-enforce* – `-fsanitize-trap=vptr` unsupported), and the constant evaluator rejects it too. That runtime detection is only heuristic, however: it relies on the dead object's vtable slot still being recognizably invalid, which is not guaranteed for a pointer to arbitrary (for example, reused heap) storage.

[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:	heuristically caught by UBSan's `vptr` check on Clang (detection relies on the ended object's vtable) and rejected in constant evaluation on Clang; GCC has no detection channel (it does not instrument `dynamic_cast` for `vptr`, and its constant evaluator accepts the cast).

Description

Using a pointer to an object whose lifetime has ended as the operand of a `dynamic_cast` is undefined behavior – the cast needs to consult the object's dynamic type information, which a polymorphic object's now-invalid vptr can no longer reliably provide.

```
struct B { virtual ~B() = default; };
struct D : B {};

void f() {
  D d;
  B* bp = &d;
  d.~D();
  D* dp = dynamic_cast<D*>(bp);   // undefined behavior
}
```

Wording

Notes regarding 6.8.4[basic.life], paragraph 7:

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]).

UB Annex Entry

F.2.7 **[ub:lifetime.outside.pointer.dynamic.cast]**

Specified in: 6.8.4[ubx:lifetime.outside.pointer.dynamic.cast]

For a pointer pointing to an object outside of its lifetime, behavior is undefined if the pointer is used as the operand of a `dynamic_cast` (7.6.1.7[expr.dynamic.cast]).

2 [*Example 1*:

```
struct B { virtual ~B() = default; };
struct D : B {};
void f()
{
  D d;
  B* bp = &d;
  d.~D();
  D* dp = dynamic_cast<D*>(bp);      // undefined behavior
}
```

— *end example*]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-*enforce***	***noexcept*-*observe***	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-*enforce***	***noexcept*-*observe***	
`lifetime.outside.pointer.dynamic.cast`		✔							✔	✔					UBSan† UBSan†	UBSan†	CE

Implementation Experience

Existing Practice

A `dynamic_cast` through a pointer to an object outside its lifetime shows a clean compiler divergence. Clang detects it in two channels – the constant evaluator rejects it, and `-fsanitize=vptr` catches it at run time (both ***noexcept*-*enforce*** and ***noexcept*-*observe***), though only heuristically, since detection relies on the dead object's vtable slot still being recognizably invalid. GCC has no detection channel at all: it does not instrument `dynamic_cast` for `vptr`, and its constant evaluator silently returns a null pointer.

[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:	heuristically caught by AddressSanitizer on both compilers (default heap-use-after-free; extra flags for stack/overflow variants) and exhaustively caught by constant evaluation.

Description

Using a *glvalue* that refers to an object outside its lifetime – either before its construction has completed or after its lifetime has ended – to access that object's value is undefined behavior. This is the general case: any read or write through such a glvalue, not a specific operation like `delete` or a cast.

```
void f() {
  int x = int{10};
  using T = int;
  x.~T();
  int y = x;   // undefined behavior: glvalue used to access the object
               // after its lifetime has ended
}
```

Wording

Notes regarding 6.8.4[basic.life], paragraph 8:

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*]

UB Annex Entry

F.2.8 **[ub:lifetime.outside.glvalue.access]**

Specified in: 6.8.4[ubx:lifetime.outside.glvalue.access]

Behavior is undefined if a glvalue referring to an object outside of its lifetime is used to access the object.

2 [*Example 1*:

```
void f() {
  int x = int{10};
  using T = int;
  x.~T();
  int y = x;     // undefined behavior, glvalue used to access the
                 // object after the lifetime has ended
}
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept-enforce***	***noexcept-observe***	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept-enforce***	***noexcept-observe***	
`lifetime.outside.glvalue.access`	✔	✔					ASan† ASan†	ASan† ASan†	✔	✔					ASan† ASan†	ASan† ASan†	CE

Implementation Experience

Existing Practice

Accessing a glvalue whose object's lifetime has ended is AddressSanitizer's (`-fsanitize=address`) headline use case: heap-use-after-free is caught by default (a freed allocation's memory is quarantined and poisoned rather than reused immediately), as is a read or write past the end of a heap, global, or (with `-fstack-protector`-style redzones already in place) stack allocation. Two related cases need an extra flag: stack-use-after-scope (access after a block-scoped automatic variable's block exits, but before the enclosing function returns) needs `-fsanitize-address-use-after-scope`; stack- and heap-use-after-return (access after the frame or coroutine that owned the storage returns) needs the allocation to be moved to a heap-backed "fake stack" via `-fsanitize-address-use-after-return=always` (or the runtime default, which decides based on `ASAN_OPTIONS=detect_stack_use_after_return`). It is available on both compilers.

Detection is heuristic rather than a deterministic check of the whole category: a freed heap allocation is only poisoned while it sits in ASan's bounded quarantine, so an access after enough further allocations have evicted it from that quarantine and the memory has been reused is not reported.

This case is also caught during constant evaluation on both compilers: accessing an object after its storage has been deleted is not a constant expression.

[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 both GCC and Clang.

Description

Using a glvalue that refers to an object whose lifetime has ended to call one of the object's non-static member functions is undefined behavior.

```
struct A {
  void f() {}
};

void f() {
  A a;
  a.~A();
  a.f();   // undefined behavior: glvalue used to call a non-static member
           // function after the object's lifetime has ended
}
```

Wording

Notes regarding 6.8.4[basic.life], paragraph 8:

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*]

UB Annex Entry

F.2.9 **[ub:lifetime.outside.glvalue.member]**

Specified in: 6.8.4[ubx:lifetime.outside.glvalue.member]

Behavior is undefined if a glvalue referring to an object outside of its lifetime is used to call a non-static member function of the object.

2 [*Example 1*:

```
struct A {
  void f() {}
};

void f() {
  A a;
  a.~A();
  a.f();    // undefined behavior, glvalue used to access a non-static member
            // function after the lifetime has ended
}
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	
`lifetime.outside.glvalue.member`	✔	✔							✔	✔							CE

Implementation Experience

Existing Practice

Calling a non-virtual member function through a glvalue outside its lifetime is rejected during constant evaluation on both compilers; no stock runtime sanitizer catches it (no vtable is read, so even `-fsanitize=vptr` is silent).

[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:	heuristically caught by UBSan's `vptr` check (detection relies on the ended object's vtable still being recognizably invalid) and exhaustively caught by constant evaluation, on both GCC and Clang.

Description

Binding a glvalue that refers to an object whose lifetime has ended to a reference to a virtual base class is undefined behavior – the binding needs the object's virtual-base layout information, which is no longer guaranteed valid.

```
struct B {};
struct D : virtual B {};

void f() {
  D d;
  d.~D();
  B& b = d;   // undefined behavior
}
```

Wording

Notes regarding 6.8.4[basic.life], paragraph 8:

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*]

UB Annex Entry

F.2.10 **[ub:lifetime.outside.glvalue.virtual]**

Specified in: 6.8.4[ubx:lifetime.outside.glvalue.virtual]

Behavior is undefined if a glvalue referring to an object outside of its lifetime is bound to a reference to a virtual base class.

2 [*Example 1*:

```
struct B {};
struct D : virtual B {
};

void f() {
  D d;
  d.~D();
  B& b = d;     // undefined behavior
}
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	
`lifetime.outside.glvalue.virtual`	✔	✔					UBSan† UBSan†	UBSan†	✔	✔					UBSan† UBSan†	UBSan†	CE

Implementation Experience

Existing Practice

Binding a glvalue outside its lifetime to a virtual base reference is inherently polymorphic (the binding reads the object's vtable for the base offset). UBSan's `vptr` check catches it on both compilers. Runtime detection is only heuristic, however: it relies on the dead object's vtable slot still being recognizably invalid, which is not guaranteed for a glvalue denoting arbitrary (for example, reused heap) storage.

This behavior is rejected during constant evaluation on both compilers.

[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:	Clang heuristically catches it at run time via UBSan's `vptr` check (both `noexcept`-enforce and `noexcept`-observe; detection relies on the ended object's vtable); GCC has no run-time channel. Both compilers reject it in constant evaluation, though only incidentally – the virtual base makes the class ineligible for a constexpr constructor/destructor, so evaluation fails before the offending `dynamic_cast` is reached.

Description

Using a glvalue that refers to an object whose lifetime has ended as the operand of a `dynamic_cast` or of `typeid` is undefined behavior, since both need the object's now-possibly-invalid dynamic type information.

```
struct B { virtual ~B(); };
struct D : virtual B {};

void f() {
  D d;
  B& br = d;
  d.~D();
  D& dr = dynamic_cast<D&>(br);   // undefined behavior
}
```

Wording

Notes regarding 6.8.4[basic.life], paragraph 8:

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*]

UB Annex Entry

F.2.11 **[ub:lifetime.outside.glvalue.dynamic.cast]**

Specified in: 6.8.4[ubx:lifetime.outside.glvalue.dynamic.cast]

Behavior is undefined if a glvalue referring to an object outside of its lifetime is used as the operand of a `dynamic_cast` or as the operand of `typeid`.

2 [*Example 1*:

```
struct B { virtual ~B(); };
struct D : virtual B {};

void f() {
  D d;
  B& br = d;
  d.~D();
  D& dr = dynamic_cast<D&>(br);      // undefined behavior
}
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept-enforce***	***noexcept-observe***	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept-enforce***	***noexcept-observe***	
`lifetime.outside.glvalue.dynamic.cast`	✔	✔							✔	✔					UBSan† UBSan†	UBSan†	CE

Implementation Experience

Existing Practice

Clang catches this UB at run time via `-fsanitize=vptr`, though only heuristically – detection relies on the dead object's vtable slot still being recognizably invalid. GCC has no run-time channel here, as it does not instrument `dynamic_cast` for `vptr`. Both compilers do reject the trigger in constant evaluation.

[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

Description

When automatic storage duration ends (at the end of a block, for example), an implicit non-trivial destructor call happens for the object originally declared in that storage. If the storage has since been reused, via placement `new`, for an object of a different type, that implicit destructor call runs the wrong destructor – for the storage's *original* type, not whatever object now actually occupies it – which is undefined behavior.

```
struct T { int x; };  // unrelated, trivial type

struct B {
  int val;
  ~B();
};

void h(int x) {
  B b;
  ::new (&b) T{x + 1};
}    // undefined behavior at block exit: implicit call to B:: B() runs
     // on storage that now holds a T
```

Wording

Notes regarding 6.8.4[basic.life], paragraph 11:

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.

UB Annex Entry

F.2.12 **[ub:original.type.implicit.destructor]**

Specified in: 6.8.4[ubx:original.type.implicit.destructor]

The behavior is undefined if a non-trivial implicit destructor call occurs when the type of the object inhabiting the associated storage is not the original type associated with that storage.

2 [*Example 1*:

```
class T {};

struct B {
  ~B();
};

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

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	
`original.type.implicit.destructor`		✔								✔							CE

Implementation Experience

Existing Practice

No sanitizer checks exist that detect this form of violation. Attempting to reproduce it during constant evaluation is hindered by not being able to `reinterpret_cast` the storage for an object to contain a different type.

[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.

Description

A `const` complete object's storage carries a restriction that outlives the object itself: creating a new object either in the storage such an object currently occupies, or in the storage it used to occupy before its lifetime ended, is undefined behavior.

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

- `creating.within.const.complete.obj.live` – the const object is still alive (its lifetime has not ended) when the new object is created in its storage
- `creating.within.const.complete.obj.reuse` – the const object's lifetime has already ended (its destructor has run) when its former storage is reused

These two are exhaustive: the wording is exactly a disjunction of these two clauses ("occupies" vs. "used to occupy before its lifetime ended"), with no third state a const complete object's storage can be in.

Wording

Notes regarding 6.8.4[basic.life], paragraph 12:

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*]

UB Annex Entry

F.2.13 **[ub:creating.within.const.complete.obj]**

Specified in: 6.8.4[ubx:creating.within.const.complete.obj]

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.

2 [*Example 1*:

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

const B b;

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

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	*noexcept-enforce*	*noexcept-observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	*noexcept-enforce*	*noexcept-observe*	
creating.within.const.complete.obj		✔							✔	✔							
↪ ...live		✔							✔	✔							CE
↪ ...reuse		✔							✔	✔							CE

Implementation Experience

Existing Practice

This rule has two clauses – creating a new object in the storage a const, complete object *occupies*, or in the storage such an object *used to occupy* before its lifetime ended – and splits accordingly. The *.live* subset places a new object directly over a still-alive const object; the *.reuse* subset is the annex's own example, where the const object's destructor runs first and the now-ended object's storage is reused. Both are caught during constant evaluation on Clang, which rejects the write into the const storage regardless of whether the destructor was called first; GCC accepts both. Clang's diagnostic here is its general "modification of a const object" rejection rather than distinct machinery for this rule's own after-lifetime-end extension, but it still rejects the exact buggy code

in both subsets, so it counts as coverage. Neither subset is caught by a stock runtime sanitizer (`-fsanitize=undefined` or `-fsanitize=address`) on either compiler, and there is no P3850 native check.

[basic.stc.alloc.dealloc.constraint]

summary:	a user-replaced 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:	Some caught by UBSan's `undefined` check on Clang.

Description

A user-replaced allocation or deallocation function must honor the semantic constraints the standard places on such functions (for example: a zero-size allocation request must still return a valid, distinct, non-null pointer if it succeeds; a deallocation function must not exit via an exception). If a user's replacement function violates one of those constraints, the behavior is undefined.

```
void* operator new(std::size_t sz) {
  if (sz == 0)
    return nullptr;         // undefined behavior: must return a non-null pointer
  return std::malloc(1);    // undefined behavior if this succeeds: the
                             // allocation must be at least sz bytes
}
void operator delete(void* ptr) noexcept {
  throw 0;                  // undefined behavior: a replaceable deallocation
                             // function must not terminate by throwing
}
```

Wording

Notes regarding 6.8.6.5.1[basic.stc.dynamic.general], paragraph 4:

4 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]).

UB Annex Entry

F.2.15 **[ub:basic.stc.alloc.dealloc.constraint]**

Specified in: 6.8.6.5.1[ubx:basic.stc.alloc.dealloc.constraint]

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.

2 [*Example 1*:

```
#include <new>
void* operator new(std::size_t sz) {
```

```
  if (sz == 0)
    return nullptr;             // undefined behavior, should return non-null pointer

  return std::malloc(1);        // undefined behavior, if successful should return allocation with
                                // length in bytes is at least as large as the requested size
}
void operator delete(void* ptr) noexcept {
  throw 0;                      // undefined behavior, terminates by throwing an exception
}
```

— *end example*]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	
`basic.stc.alloc.dealloc.constraint`		✔								✔					UBSan UBSan	UBSan	CE

Implementation Experience

Existing Practice

This category is a user-replaced allocation or deallocation function that violates its semantic constraints; the canonical example is a throwing-form (non-`nothrow`) `operator new` that returns null instead of returning a non-null pointer or exiting via an exception. Coverage differs between the two compilers.

Clang: stock UBSan's implicit `returns_nonnull` check on a replaceable global `operator new` flags the violation at the point of return – "null pointer returned from function declared to never return null" – before any caller uses the bad pointer. This is a targeted catch of the allocation function's own constraint, where a postcondition assertion is essentially added to the function declaration by the attribute.

GCC: GCC does not similarly annotate the allocation functions or provide the corresponding check for the return value.

Other aspects of the behavior are not checked by existing tools. Replacement allocation functions are not used during constant evaluation, so *constexpr* coverage is not applicable.

[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:	caught by AddressSanitizer on both compilers.

Description

An allocation function called with a size of zero must still return a valid, non-null pointer, but that pointer does not point to any actual storage the program is allowed to use – dereferencing it is undefined behavior.

```
void test() {
  char* c = static_cast<char*>(operator new(0z));
  c[0] = 'X';   // undefined behavior
}
```

Wording

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.

UB Annex Entry

F.2.16 **[ub:basic.stc.alloc.zero.dereference]**
Specified in: 6.8.6.5.2[ubx:basic.stc.alloc.zero.dereference]
The pointer returned when invoking an allocation function with a size of zero cannot be dereferenced.

2 [*Example 1*:

```
void test()
{
    char* c = static_cast<char*>(operator new(0z));
    c[0] = 'X';     // undefined behavior
}
```

— *end example*]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**noexcept**-*enforce*	**noexcept**-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**noexcept**-*enforce*	**noexcept**-*observe*	
basic.stc.alloc.zero. dereference		✔					ASan† ASan	ASan		✔					ASan† ASan	ASan	CE

Implementation Experience

Existing Practice

AddressSanitizer catches dereferences of pointers returned from zero-sized allocations, reporting the error as a `heap-buffer-overflow`. This is heuristic, and not guaranteed, as the access itself needs to land in the region AddressSanitizer has poisoned. (A member access to a member with a sufficiently large offset might jump out of the poisoned region and access a nearby non-poisoned region of memory, avoiding the checks that AddressSanitizer introduces).

[**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

Description

Computing a pointer value to, or one-past-the-end of, an object before that object's storage duration has even begun is undefined behavior. A consequence of this rule is subtle: a pointer value computed before an object's storage exists can never be treated as pointing to that object, even if the address later turns out to coincide with where the object is placed.

```
void f() {
  int x;
  std::uintptr_t pval = reinterpret_cast<std::uintptr_t>((&x) + 1);
  int *p = reinterpret_cast<int*>(pval);
  {
    int y;   // y's storage duration begins here, so p cannot be &y
    if (pval == reinterpret_cast<std::uintptr_t>(&y)) {
      *p = 17;   // undefined behavior
    }
  }
}
```

Wording

Notes regarding 6.9.5[basic.compound], paragraph 5:

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*]

UB Annex Entry

F.2.17 [**ub:basic.compound.pointer.before.storage.duration**]
Specified in: 6.9.5[ubx:basic.compound.pointer.before.storage.duration]
Producing a pointer value to or past the end of an object before the beginning of the duration of that object's storage has undefined behavior. Therefore any pointer created before the storage duration of an object cannot be a pointer to that object.

2 [*Example 1*: In the following example, initializing the pointer `p` has undefined behavior because all possible values of `p`, including `&y`, lead to undefined behavior.

```
#include <cstdint>
void f() {
  int x;
  std::uintptr_t pval = reinterpret_cast<std::uintptr_t>((&x)+1);
  int *p = reinterpret_cast<int*>(pval);
  {
    int y;          // storage duration of y begins, so p is not &y
    if (pval == reinterpret_cast<std::uintptr_t>(&y)) { // consider when x and y are consecutive
      *p = 17;      // undefined behavior, so p is not (&x)+1
    }
  }
}
```

— *end example*]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	
`basic.compound.pointer.before.storage.duration`		✔								✔							CE

[basic.compound.invalid.pointer]

summary:	dereferencing a pointer to storage that has already been freed.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track object lifetimes across the entire program
replacement:	None
implementations:	heuristically caught by AddressSanitizer, on both compilers; also rejected during constant evaluation on both compilers.

Description

Once storage has been freed, a pointer that referenced it becomes invalid. Most uses of an invalid pointer value are only implementation-defined, but *indirection* through it – dereferencing it – is undefined behavior.

```
void f() {
  int *x = new int{5};
  delete x;
  int y = *x;   // undefined behavior
}
```

Wording

Notes regarding 6.9.5[basic.compound], paragraph 6:

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*]

UB Annex Entry

F.2.18 **[ub:basic.compound.invalid.pointer]**

Specified in: 6.9.5[ubx:basic.compound.invalid.pointer]

Indirection with a pointer value referencing storage that has been freed has undefined behavior. (Most other uses of such a pointer have implementation-defined behavior.)

2 [*Example 1*:

```
void f()
{
    int *x = new int{5};
    delete x;
    int y = *x; // undefined behavior
}
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	
`basic.compound.invalid.pointer`	✔	✔					ASan† ASan†	ASan† ASan†	✔	✔					ASan† ASan†	ASan† ASan†	CE

Implementation Experience

Existing Practice

Indirection through a pointer to freed storage is AddressSanitizer's (`-fsanitize=address`) headline case: the heap-use-after-free in this category's example is caught by default on both compilers, because a freed allocation's memory is quarantined and poisoned rather than reused immediately. Detection is heuristic rather than a deterministic check of the whole category – once enough further allocations evict the freed block from ASan's bounded quarantine and its memory is reused, the invalid access is no longer reported.

The default behavior when a violation is detected is to terminate after reporting an error, which is the ***noexcept*-enforce** semantic with a local violation handler. Using the `-fsanitize-recover` option provides the equivalent of the ***noexcept*-observe** semantic.

This case is also caught during constant evaluation on both compilers: dereferencing an object after its storage has been deleted is not a constant expression.

P3850 Sanitizer Integration

On the `p3850-contracts` branches, AddressSanitizer is integrated with the contract-violation handler via `-fsanitize-semantic=address:<semantic>`: the same heap-use-after-free ASan detects is delivered to the handler instead of printed directly. Both non-throwing checked semantics are available.

The potentially throwing *enforce* and *observe* semantics are not supported yet — AddressSanitizer's reporting happens within a `noexcept` function (a destructor), and removing that `noexcept` would require all potential call sites to be instrumented to support exception propagation (or to be aware of the presence or absence of checks introduced by AddressSanitizer).

[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

Description

If two side effects on the same memory location are unsequenced relative to each other – or a side effect and a value computation reading that same location are unsequenced – and they are not potentially concurrent (running on different threads), the behavior is undefined. This is the classic "multiple unsequenced modifications of a scalar" rule, which applies to single-threaded execution rather than to genuinely concurrent accesses.

```
void g(int i) {
  i = i++ + 1;   // well-defined since C++17: the RHS value computation and
                 // the i++ side effect are both sequenced before the
                 // assignment to i
  i = i++ + i;   // undefined behavior: the ++ side effect and the read of
                 // i for the right-hand + i are unsequenced
}
```

Wording

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*]

UB Annex Entry

F.2.19 **[ub:intro.execution.unsequenced.modification]**

Specified in: 6.10.1[ubx:intro.execution.unsequenced.modification]

If a side effect on a memory location (6.8.1[intro.memory]) is unsequenced relative to either another side effect on the same memory location or a value computation using the value of any object in the same memory location, and they are not potentially concurrent (6.10.2[intro.multithread]), the behavior is undefined.

2 [*Example 1*:

```
void g(int i) {
  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
}
```

— *end example*]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	
`intro.execution.` `unsequenced.modification`		✔								✔							CE

Implementation Experience

Existing Practice

No sanitizer or native check catches an unsequenced modification (e.g. `i = i++ + 1;`) on either compiler, and the constant evaluator does NOT catch it either (both compilers silently pick an evaluation order). The only stock signal is a compile-time warning: GCC's `-Wsequence-point` (needs `-Wall`) and Clang's `-Wunsequenced` (default-on).

[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.

Description

A *data race* occurs when two threads perform unsynchronized, potentially concurrent accesses to the same memory location, at least one of which is a write. This rule covers accesses that genuinely run on different threads with no ordering imposed between them; the behavior is undefined.

```
int total = 0;
void worker() {
  while (total < 10000000)
    ++total;      // unsynchronized read-modify-write, unsynchronized with
                  // the other thread's identical access
}
// spawning two or more threads that each run worker() is undefined
// behavior: every ++total races every other thread's ++total
```

Wording

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*]

UB Annex Entry

F.2.20 **[ub:intro.races.data]**

Specified in: 6.10.2.2[ubx:intro.races.data]

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 in 6.10.2.2[intro.races]. Any such data race results in undefined behavior.

2 [*Example 1*: In the following example, `t1`, `t2` and `t3` have a data race on access of variable `count`.

```
int count = 0;
auto f = [&] { count++; };
std::thread t1{f}, t2{f}, t3{f};     // undefined behavior
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	
`intro.races.data`		✔					TSan† TSan	TSan		✔					TSan† TSan	TSan† TSan	CE

Implementation Experience

Existing Practice

ThreadSanitizer (`-fsanitize=thread`) is purpose-built to detect exactly this category: two unsynchronized accesses to the same memory location, at least one a write. Detection is heuristic rather than a deterministic check of every data race in the program – TSan instruments every memory access and every synchronization primitive it can see, but it only *reports* a race that is actually scheduled to overlap on the particular execution being observed; a race that exists in the program but does not happen to manifest on a given run produces no report at all. It is available on both compilers.

This is demonstrated on both compilers by the runnable example: `-fsanitize=thread` cleanly reports "WARNING: ThreadSanitizer: data race" with both racing stack traces and exits nonzero.

[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

Description

The implementation is entitled to *assume* that every thread of execution will eventually do one of: terminate, call `std::this_thread::yield`, make a call to a library I/O function, access through a volatile glvalue, perform an atomic or synchronization operation (other than an atomic read-modify-write), or continue execution of a *trivial infinite loop*. A thread that has not terminated and does none of these – so it stops making execution steps without ever satisfying the assumption – has undefined behavior.

The trivial-infinite-loop affordance is the key subtlety: the loop shown is undefined *only because* its condition is not a constant expression, so the compiler cannot treat it as a trivial infinite loop. A genuine trivial infinite loop – a constant `true` condition with a trivial body – is explicitly *not* undefined (the implementation may keep, remove, or reorder it).

```
bool stop() { return false; }     // not a constant expression
void busy_wait() {
  while (!stop())                 // spins forever; not a trivial infinite
    ;                             // loop, so the non-terminating thread is
}                                 // undefined behavior
```

A reasonable replacement behavior that does not break the fundamental concurrency model of C++ does not seem to exist.

Wording

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*]

UB Annex Entry

F.2.21 **[ub:intro.progress.stops]**

Specified in: 6.10.2.3[ubx:intro.progress.stops]

The behavior is undefined if a thread of execution that has not terminated stops making execution steps.

2 [*Example 1*: In the following example, the loop condition `!stop()` is not a constant expression, so the infinite loop which results is undefined behavior.

```
bool stop() { return false; }

void busy_wait_thread() {
  while (!stop());  // undefined behavior
}

int main() {
  std::thread t(busy_wait_thread);
  t.join();
}
```

— *end example*]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	
`intro.progress.stops`		✔								✔							

[basic.start.main.exit.during.destruction]

summary:	calling `std::exit` recursively during the destruction sequence it would restart.
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

Description

Calling `std::exit` recursively re-enters the very static/thread destruction sequence it is meant to trigger. If `std::exit` is called while a static- or thread-storage-duration object is already being destroyed as part of that sequence, the behavior is undefined.

```
struct Exiter {
  ~Exiter() { std::exit(0); }
};
Exiter ex;

int main() {}
// after main returns, destroying ex calls std::exit -- but ex is
// itself being destroyed as part of the very sequence std::exit would
// restart: undefined behavior
```

Wording

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]).

UB Annex Entry

F.2.22 **[ub:basic.start.main.exit.during.destruction]**

Specified in: 6.10.3.1[ubx:basic.start.main.exit.during.destruction]

If `std::exit` is called to end a program during the destruction of an object with static or thread storage duration, the program has undefined behavior.

2 [*Example 1*:

```
#include <cstdlib>

struct Exiter {
  ~Exiter() { std::exit(0); }
};

Exiter ex;
```

```
int main() {}
```

During the destruction of static variables after `main` completes, the destructor of the variable `ex`, which invokes `std::exit`, has undefined behavior. — *end example*]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	
`basic.start.main.exit.during.destruction`		✔								✔							CE

Implementation Experience

Existing Practice

Calling `std::exit` during the destruction of a static object is caught by no stock tool on either compiler (both runtimes handle the recursive `exit` defensively without crashing), there is no native check, and it is not expressible in constant evaluation. No tool covers this category.

[basic.start.term.use.after.destruction]

summary:	using a static-storage-duration object after its destructor has run.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track the lifetime of static objects
replacement:	None
implementations:	None

Description

Static-storage-duration objects are destroyed in the reverse order their construction completed. If, during that destruction sequence, an object that has *already* been destroyed is used again – either because control re-enters a function containing a block-scope static whose lifetime has ended, or because the destroyed object is reached indirectly through a pointer saved to it – the behavior is undefined.

```
struct A {};
void f() { static A a; }

struct B { B() { f(); } };
struct C { ~C() { f(); } };

C c;
B b;   // f() during B's construction begins a's lifetime

int main() {}
// destruction order is b, then a, then c; c's destructor calls
// f() again, passing through a's (already-destroyed) definition:
// undefined behavior
```

Wording

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*]

UB Annex Entry

F.2.23 **[ub:basic.start.term.use.after.destruction]**

Specified in: 6.10.3.4[ubx:basic.start.term.use.after.destruction]

If a function contains a block-scope object 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-scope object. Likewise, the behavior is undefined if the block-scope object is used indirectly (i.e., through a pointer) after its destruction.

2 [*Example 1*:

```
struct A {};
void f() {
  static A a;
}

struct B {
  B() { f(); }
};
struct C {
  ~C() { f(); }
};

C c;
B b;    // call to f() in constructor begins lifetime of a

int main() {}   // undefined behavior
```

Objects with static storage duration are destroyed in reverse order of the completion of their construction. In the above example, first `c` then `a` and finally `b` completes construction. When the destructor of `c` is called after first destroying `b` and then `a`, it calls `f()` which passes through the definition of the previously destroyed block-scope object `a`. — *end example*]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	*noexcept-enforce*	*noexcept-observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	*noexcept-enforce*	*noexcept-observe*	
`basic.start.term.use.after.destruction`		✔								✔							CE

Implementation Experience

Existing Practice

Using a static object after its destruction during program termination is caught by no stock tool on either compiler (AddressSanitizer does not poison static storage on destruction the way it does heap), there is no native check, and it is not expressible in constant evaluation.

[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

Description

This is the general "strict aliasing" rule: accessing an object's stored value through a glvalue whose type is not *similar* to the object's actual (dynamic) type is undefined behavior. It permits the optimizer to assume that accesses through unrelated pointer/reference types cannot alias the same storage.

```
int foo(float* f, int* i) {
  *i = 1;
  *f = 0.f;   // undefined behavior: glvalue type (float) is not similar
              // to the object's type (int)
  return *i;
}

int main() {
  int x = 0;
  x = foo(reinterpret_cast<float*>(&x), &x);
}
```

Wording

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

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

— $\texttt{T}_{\text{obj}}$,

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

UB Annex Entry

F.3.2 **[ub:expr.basic.lvalue.strict.aliasing.violation]**

Specified in: 7.2.1[ubx:expr.basic.lvalue.strict.aliasing.violation]

If a program attempts to access (3.1[defns.access]) the stored value of an object whose dynamic type is T through a glvalue whose type is not similar (7.3.6[conv.qual]) to T (or its corresponding signed or unsigned types) the behavior is undefined.

2 [*Example 1*:

```
int foo(float* f, int* i) {
  *i = 1;
  *f = 0.f;      // undefined behavior, glvalue is not similar to int

  return *i;
}

int main() {
  int x = 0;

  x = foo(reinterpret_cast<float*>(&x), &x);
}
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	
`expr.basic.lvalue.strict.aliasing.violation`		✔								✔							CE

Implementation Experience

Existing Practice

No stock sanitizer catches this category on either compiler.

This category has no *constexpr* entry: the trigger can only be formed through a `reinterpret_cast` between unrelated pointer types, which is not permitted in constant evaluation.

[expr.basic.lvalue.union.initialization]

summary:	defaulted union copy/move construction whose argument isn't a live similar-type object.
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.

Description

A defaulted copy/move constructor or assignment operator of a union is implemented as a bitwise copy of the argument. If that argument is not actually a live object of a type similar to the union, there is nothing valid for the bitwise copy to read, and the behavior is undefined.

```
union U { int x; };
void f() {
  char u[sizeof(U)];
  U o = reinterpret_cast<U&>(u);   // undefined behavior: u is not a live
                                   // object of a type similar to U
}
```

Wording

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

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

— $\texttt{T}_{\text{obj}}$,

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

UB Annex Entry

F.3.3 **[ub:expr.basic.lvalue.union.initialization]**

Specified in: 7.2.1[ubx:expr.basic.lvalue.union.initialization]

If a program invokes a defaulted copy/move constructor or defaulted copy/move assignment operator of a union with an argument that is not an object of a similar type within its lifetime, the behavior is undefined.

2 [*Example 1*:

```
union U { int x; };
void f()
{
   char u[sizeof(U)];
   U o = reinterpret_cast<U&>(u);   // undefined behavior
}
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	
`expr.basic.lvalue.union.initialization`	✔	✔							✔	✔							CE

[expr.type.reference.lifetime]

summary:	evaluating (naming) a reference to an object whose lifetime has ended.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track object and storage lifetimes
replacement:	None
implementations:	None

Description

References are conceptually pointers that have already been dereferenced. This rule makes that precise: merely *evaluating* a reference (naming it, without necessarily reading through it) is undefined behavior whenever the equivalent dereference of a pointer to the same object would be – for example, because the referenced object's lifetime has ended.

```
void g() {
  int* ip = new int(5);
  int& i = *ip;
  delete ip;
  i;    // undefined behavior: evaluating i is equivalent to dereferencing
        // a pointer to a deleted object
}
```

Wording

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*]

UB Annex Entry

F.3.4 **[ub:expr.type.reference.lifetime]**

Specified in: 7.2.2[ubx:expr.type.reference.lifetime]

Evaluating a reference when an equivalent use of a pointer denoting the same object would be invalid has undefined behavior.

2 [*Example 1*:

```
void g()
{
  int* ip = new int(5);
  int& i = *ip;
```

```
  delete ip;
  i;    // undefined behavior
}
```

— *end example*]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**`noexcept`**-*enforce*	**`noexcept`**-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**`noexcept`**-*enforce*	**`noexcept`**-*observe*	
`expr.type.reference.lifetime`		✔								✔							CE

Implementation Experience

Existing Practice

Nothing catches this category on either compiler. The annex's bare naming of the reference emits no instructions, so no runtime sanitizer has anything to instrument, and the constant evaluator tracks lifetime exactly for any operation that does read through the reference – so the acceptance here is not a gap; there is simply no observable operation to check.

[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:	Only for the null pointer case
checking:	Track whether storage is associated with an object of correct type within its lifetime or an object currently being constructed or destroyed; insert null pointer check
replacement:	None
implementations:	Many cases caught by UBSan's `vptr` check on both compilers.

Description

Converting a pointer to a derived class into a pointer to a virtual base class requires consulting that specific object's virtual-base layout information (the offset can differ between objects of the same type, since virtual inheritance is diamond-shaped and layout-dependent). If the source pointer does not point to a valid, in-lifetime object, the conversion is undefined behavior.

```
struct B {};
struct D : virtual B {};
void f() {
  D ds[1];
  B* b = &ds[1];   // undefined behavior: ds[1] is one past the array,
                   // not a valid D object
}
```

Wording

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.3.9[ubx:conv.ptr.virtual.base]). Otherwise, the result is a pointer to the base class subobject of the derived class object.

UB Annex Entry

F.3.9 **[ub:conv.ptr.virtual.base]**

Specified in: 7.3.12[ubx:conv.ptr.virtual.base]

Converting a pointer to a derived class `D` to a pointer to a virtual base class `B` that does not point to a valid object within its lifetime has undefined behavior.

2 [*Example 1*:

```
struct B {};
struct D : virtual B {};
void f()
{
  D ds[1];
  B* b = &ds[1];     // undefined behavior
}
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**`noexcept`**-*enforce*	**`noexcept`**-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**`noexcept`**-*enforce*	**`noexcept`**-*observe*	
`conv.ptr.virtual.base`		✔					UBSan† UBSan†	UBSan†		✔					UBSan† UBSan†	UBSan†	CE

Implementation Experience

Existing Practice

Converting a pointer to a derived class to one of its virtual bases must read the object's vtable for the base offset, so this undefined behavior is inherently polymorphic (there is no non-polymorphic subcategory where this UB can apply). When the source does not point to a valid object within its lifetime and the vtable has been successfully cleared or poisoned, UBSan's `vptr` check rejects the conversion on both compilers.

[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:	GCC's `object-size` check flags some cases (heuristic; stock and integrated); constant evaluation exhaustive on both compilers.

Description

A pointer-to-member of a base class can be converted to a pointer-to-member of a derived class, adjusting the stored offset along the way. If the derived class named in the conversion could not actually contain the member in question (for example, because the member belongs to a different sibling class entirely), the conversion is undefined behavior.

```
struct B {};
struct D1 : B { int d1; };
struct D2 : B {};
void f() {
  int (D1::*pd1) = &D1::d1;
  int (B::*pb) = static_cast<int (B::*)>(pd1);
  int (D2::*pd2) = pb;   // undefined behavior: D2 cannot contain D1::d1
}
```

Wording

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.3.10[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*`.

UB Annex Entry

F.3.10 **[ub:conv.member.missing.member]**

Specified in: 7.3.13[ubx:conv.member.missing.member]

The conversion of a pointer to a member of a base class to a pointer to member of a derived class that could not contain that member has undefined behavior.

2 [*Example 1*:

```
struct B {};
struct D1 : B { int d1; };
struct D2 : B {};
void f()
{
  int (D1::*pd1) = &D1::d1;
  int (B::*pb) = static_cast<int (B::*)>(pd1);
  int (D2::*pd2) = pb;  // undefined behavior
}
```

— *end example*]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	
`conv.member.missing.member`	✔	✔					UBSan	UBSan[†] UBSan	✔	✔							CE

Implementation Experience

Existing Practice

No dedicated sanitizer check for member-pointer-conversion validity exists on either compiler. GCC's `-fsanitize=undefined` does, however, incidentally catch some of the *manifestations* of this UB – the out-of-object load performed when the resulting `D2::*` is applied to a real `D2` object – through its generic object-size bounds instrumentation, reporting "load of address ... with insufficient space for an object of type 'int'." This fires only when the object's size and the (compile-time-constant) member-pointer offset are both statically known. Clang's `-fsanitize=undefined` emits nothing on the same trigger. None of this involves any contract-violation handler.

Constant evaluation catches this category on both compilers: converting a member pointer to a class that cannot contain the member, and then applying it, is rejected as not a constant expression – both evaluators validate that the target class actually contains a compatible member at the converted offset.

[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; rejected during constant evaluation on both GCC and Clang.

Description

Calling a function through a function pointer (or reference) whose type is not call-compatible with the function's actual definition – for example, after a `reinterpret_cast` between unrelated function pointer types – is undefined behavior, even if the call happens to "work" at the ABI level for a particular argument.

```
using f_int = int (*)(int);
int f1(float) { return 10; }

int main() {
  return reinterpret_cast<f_int>(f1)(20);   // undefined behavior: the
                                              // expression's function type
                                              // differs from f1's actual type
}
```

Wording

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

6 A type $\texttt{T}_{\text{call}}$ is *call-compatible* with a function type $\texttt{T}_{\text{func}}$ if $\texttt{T}_{\text{call}}$ is the same type as $\texttt{T}_{\text{func}}$ or if the type "pointer to $\texttt{T}_{\text{func}}$" can be converted to type "pointer to $\texttt{T}_{\text{call}}$" 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.3.11[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*]

UB Annex Entry

F.3.11 **[ub:expr.call.different.type]**

Specified in: 7.6.1.3[ubx:expr.call.different.type]

Calling a function through an expression whose function type is not call-compatible with the function type of the called function's definition results in undefined behavior.

2 [*Example 1*:

```
using f_float = int (*)(float);
using f_int = int (*)(int);

int f1(float) { return 10; }

int f2(int) { return 20; }

int main() {
  return reinterpret_cast<f_int>(f1)(20);   // undefined behavior, the function type of the expression
                                            // is different from the called function's definition
}
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**`noexcept`**-*enforce*	**`noexcept`**-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**`noexcept`**-*enforce*	**`noexcept`**-*observe*	
`expr.call.different.type`	✔	✔							✔	✔		UBSan			UBSan UBSan	UBSan UBSan	CE

Implementation Experience

Existing Practice

Coverage of this category diverges between the two compilers. Clang's UBSan `function` check (part of the `undefined` group, so also enabled by plain `-fsanitize=undefined`) detects an indirect call made through a function-pointer type that is not call-compatible with the callee's definition, reporting "call to function ... through pointer to incorrect function type" at the call site.

This case cannot be fully reconstructed at compile time due to the need to use a `reinterpret_cast`.

[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:	caught by both compilers' constant evaluators.

Description

In a member access `E1.E2`, `E1` is expected to actually refer to an object of a type similar to `E1`'s own expression type. If it does not – typically because `E1` was reached through a `reinterpret_cast` to an unrelated type – the behavior is undefined, even for a member name (`E2`) that happens to exist on both types.

```
struct A { int i; };
struct B { int j; };
A a;
int x = reinterpret_cast<B&>(a).j;   // undefined behavior: a's type
                                     // (A) is not similar to B
```

Wording

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.3.12[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*]

UB Annex Entry

F.3.12 **[ub:expr.ref.member.not.similar]**

Specified in: 7.6.1.5[ubx:expr.ref.member.not.similar]

In a class member access `E1.E2`, where `E2` is a non-static member, the behavior is undefined if `E1` refers to an object whose type is not similar to the type of the expression `E1`.

2 [*Example 1*:

```
struct A { int i; };
struct B { int j; };
A a;
int x = reinterpret_cast<B&>(a).j;  // undefined behavior
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	
`expr.ref.member.not.similar`	✔	✔							✔	✔							CE

Implementation Experience

Existing Practice

No stock sanitizer catches this category on either compiler. There is no dedicated UBSan check for a class member access through a reference whose type is not similar to the referred-to object's type; `-fsanitize=undefined` emits nothing on both compilers once the trigger is constructed so that the two class types share size and alignment (a type pun across types of differing alignment would instead trip UBSan's unrelated `alignment` check, a different category). ASan is likewise silent, since the underlying object is live and in scope.

Constant evaluation, however, catches it on both compilers, though with a per-compiler caveat. GCC's rejection is a real, trigger-specific check: it names the exact mismatch ("accessing value of an `A` object through a `B` glvalue in a constant expression") and accepts the otherwise-identical same-type cast, which is not undefined behavior. Clang instead bans `reinterpret_cast` in a constant expression categorically, so its rejection would fire even for the well-defined same-type case and carries no category-specific information.

[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:	Only for the null pointer case
checking:	Track whether storage is associated with an object of correct type within its lifetime or an object currently being constructed or destroyed; insert null pointer check
replacement:	None
implementations:	None

Description

`dynamic_cast` needs to consult the dynamic type of the object a non-null pointer points to. If that pointer instead points to an object of the wrong type, or to storage that is not within any object's lifetime at all, the cast is undefined behavior rather than simply failing gracefully.

```
struct B { virtual ~B(); };
void f() {
  B bs[1];
  B* dp = dynamic_cast<B*>(bs + 1);   // undefined behavior: bs+1 is one
                                      // past the array, not a valid object
}
```

Wording

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.3.13[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.3.14[ubx:expr.dynamic.cast.glvalue.lifetime]).

UB Annex Entry

F.3.13 **[ub:expr.dynamic.cast.pointer.lifetime]**

Specified in: 7.6.1.7[ubx:expr.dynamic.cast.pointer.lifetime]

Evaluating a `dynamic_cast` on a non-null pointer that points to an object (of polymorphic type) of the wrong type or to an object not within its lifetime has undefined behavior.

2 [*Example 1*:

```
struct B { virtual ~B(); };
void f() {
  B bs[1];
  B* dp = dynamic_cast<B*>(bs+1);   // undefined behavior
}
```

— end example]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**`noexcept`**-*enforce*	**`noexcept`**-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**`noexcept`**-*enforce*	**`noexcept`**-*observe*	
`expr.dynamic.cast.pointer.lifetime`		✔								✔							CE

[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 Clang.

Description

`dynamic_cast<T&>` on a reference that denotes an object whose type is not similar to the referenced type, or an object not within its lifetime, is undefined behavior – a reference cast has no null-like value to fall back to, so the failure mode is UB rather than a defined "cast failed" signal.

```
struct B { virtual ~B(); };
void f() {
  B bs[1];
  B& dr = dynamic_cast<B&>(bs[1]);   // undefined behavior: bs[1] is one
                                     // past the array, not a valid object
}
```

Wording

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.3.13[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.3.14[ubx:expr.dynamic.cast.glvalue.lifetime]).

UB Annex Entry

F.3.14 **[ub:expr.dynamic.cast.glvalue.lifetime]**

Specified in: 7.6.1.7[ubx:expr.dynamic.cast.glvalue.lifetime]

Evaluating a `dynamic_cast` on a reference that denotes an object (of polymorphic type) of a type that is not similar to the referenced type or an object not within its lifetime has undefined behavior.

2 [*Example 1*:

```
struct B { virtual ~B(); };
void f() {
  B bs[1];
  B& dr = dynamic_cast<B&>(bs[1]);  // undefined behavior
}
```

— end example]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**`noexcept`**-*enforce*	**`noexcept`**-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**`noexcept`**-*enforce*	**`noexcept`**-*observe*	
`expr.dynamic.cast.glvalue.lifetime`		✔							✔	✔							CE

Implementation Experience

Existing Practice

The annex uses an identity-conversion shape (`dynamic_cast<B&>` on a `B`); at run time no check is emitted and nothing catches it on either compiler. The one channel that does react is constant evaluation, and it does so asymmetrically. The reference form obtains its operand by subscripting (`bs[1]`), which is itself a one-past-the-end dereference: Clang's constant evaluator rejects that access ("read of dereferenced one-past-the-end pointer is not allowed"). GCC accepts it – a real gap, because a bare `bs[1]` (no cast) *is* rejected by GCC's evaluator, but wrapping the identical subscript in `dynamic_cast<B&>(...)` suppresses GCC's own out-of-bounds-subscript diagnostic.

[**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, integrated, both `noexcept`-enforce and `noexcept`-observe) and constant evaluation (Clang); nonpolymorphic subset caught by constant evaluation (Clang).

Description

`static_cast` can convert a glvalue of a base type `B` to a reference to a derived type `D`, but it performs no runtime check that the referenced object is actually a `D` – it simply trusts the programmer and adjusts the pointer as if it were. If the object is actually some other type sharing base `B` (a sibling class, for instance), the cast is undefined behavior.

```
struct B {};
struct D1 : B {};
struct D2 : B {};

void f() {
  D1 d;
  B &b = d;
  static_cast<D2 &>(b);   // undefined behavior: b actually refers to a D1
}
```

- `expr.static.cast.base.class.polymorphic` – `B` and `D1`/`D2` are polymorphic (have a vtable), so a runtime check against the object's dynamic type is at least possible in principle
- `expr.static.cast.base.class.nonpolymorphic` – no vtable exists, so no runtime information distinguishes the sibling types at all

These two are exhaustive: every class is either polymorphic or not, and that is exactly the axis that determines whether any runtime check could ever exist for this cast.

Wording

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.3.15[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*]

UB Annex Entry

F.3.15 **[ub:expr.static.cast.base.class]**

Specified in: 7.6.1.9[ubx:expr.static.cast.base.class]

A glvalue of type `B` can be cast to the type "reference to `D`" if `B` is a base class of `D`, otherwise the behavior is undefined.

2 [*Example 1*:

```
struct B {};
struct D1 : B {};
struct D2 : B {};

void f() {
  D1 d;
  B &b = d;
  static_cast<D2 &>(b); // undefined behavior, base class object of type D1 not D2
}
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept-enforce***	***noexcept-observe***	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept-enforce***	***noexcept-observe***	
`expr.static.cast.base.class`		✔							✔	✔							
↪ `...polymorphic`		✔							✔	✔					UBSan UBSan	UBSan	CE
↪ `...nonpolymorphic`		✔							✔	✔							CE

Implementation Experience

Existing Practice

For the *polymorphic* objects, UBSan's `vptr` check applies in principle, but only Clang instruments it here: Clang catches the bad reference cast. GCC does not instrument the *reference* form of a

`static_cast` downcast at all – GCC's `vptr` coverage of these casts is limited to the pointer form. The *nonpolymorphic* subset (the annex's own example) has no vtable, so `vptr` is inapplicable and no stock runtime tool catches it.

Both subsets are rejected during constant evaluation by Clang ("cannot cast object of dynamic type 'D1' to type 'D2"') but accepted silently by GCC, whose constant evaluator does not cross-check a `static_cast`'s target type against the operand's real constructed type.

[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:	Only for the null pointer case
checking:	Track whether storage is associated with an object of correct type within its lifetime or an object currently being constructed or destroyed; insert a null pointer check
replacement:	None
implementations:	polymorphic subset caught by UBSan's `vptr` check (both compilers, integrated, both `noexcept`-enforce and `noexcept`-observe) and constant evaluation (Clang); nonpolymorphic subset caught by constant evaluation (Clang).

Description

`static_cast`-ing a pointer to a base class down to a pointer to a derived class performs no runtime check, and is undefined behavior if there is no enclosing object of that derived type at the pointed-to location.

```
struct B {};
struct D1 : B {};
struct D2 : B {};

void f() {
  B *bp = new D1;
  static_cast<D2 *>(bp);   // undefined behavior: bp actually points to a D1
}
```

- `expr.static.cast.downcast.wrong.derived.type.polymorphic` – the classes involved are polymorphic, so a vtable exists that could in principle be consulted at runtime
- `expr.static.cast.downcast.wrong.derived.type.nonpolymorphic` – no vtable exists, so there is no runtime information to check against at all

These two are exhaustive: every class is either polymorphic or not, and polymorphic-or-not is exactly the axis controlling whether a runtime check could ever exist here.

Wording

Modify and note section 7.6.1.9[expr.static.cast], paragraph 10:

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.3.18[ubx:expr.static.cast.downcast.wrong.derived.type]).

UB Annex Entry

F.3.18 **[ub:expr.static.cast.downcast.wrong.derived.type]**

Specified in: 7.6.1.9[ubx:expr.static.cast.downcast.wrong.derived.type]

Casting from a pointer to a base class to a pointer to a derived class when there is no enclosing object of that derived class at the specified location has undefined behavior.

2 [*Example 1*:

```
struct B {};
struct D1 : B {};
struct D2 : B {};

void f() {
  B *bp = new D1;
  static_cast<D2 *>(bp);        // undefined behavior, base class object of type D1 not D2
}
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	*noexcept-enforce*	*noexcept-observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	*noexcept-enforce*	*noexcept-observe*	
`expr.static.cast.downcast.wrong.derived.type`		✔							✔	✔							
↪ `...polymorphic`		✔					UBSan UBSan	UBSan	✔	✔					UBSan UBSan	UBSan	CE
↪ `...nonpolymorphic`		✔							✔	✔							CE

Implementation Experience

Existing Practice

This is a `static_cast<D2*>` applied to a pointer that actually points to a `D1`, and, for the *polymorphic* subset, is the archetypal `vptr` case: UBSan's `vptr` check catches it on both compilers ("downcast of address ... which does not point to an object of type 'D2"'). GCC catches this pointer form even though it does not instrument the logically identical reference-form cast on the same object layout, so GCC's `vptr` coverage of `static_cast` downcasts is pointer-expression-specific.

The *nonpolymorphic* subset has no vtable and is caught by no stock runtime tool.

Both subsets are rejected during constant evaluation by Clang and accepted silently by GCC, whose constant evaluator does not cross-check a `static_cast`'s target type against the operand's real constructed type.

[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.

Description

A pointer-to-member of a derived class can be `static_cast` to a pointer-to-member of a base class, adjusting the stored offset – but only if the target base class actually contains the original member (directly, or is otherwise related to the class that declared it). Casting to an unrelated class that happens to share a common derived type is undefined behavior.

```
struct A { char c; };
struct B { int i; };
struct D : A, B {};

int main() {
  char D::*p1 = &D::c;
  char B::*p2 = static_cast<char B::*>(p1);   // undefined behavior: B
                                                // does not contain c
}
```

Wording

Modify and note section 7.6.1.9[expr.static.cast], paragraph 11:

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.3.19[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*]

UB Annex Entry

F.3.19 [ub:expr.static.cast.does.not.contain.original.member]

Specified in: 7.6.1.9[ubx:expr.static.cast.does.not.contain.original.member]

A pointer to member of derived class `D` can be cast to a pointer to member of base class `B` (with certain restrictions on *cv* qualifiers) as long as `B` contains the original member, or is a base or derived class of the class containing the original member; otherwise the behavior is undefined.

2 [*Example 1*:

```
struct A {
  char c;
};

struct B {
  int i;
};

struct D : A, B {};

int main() {
  char D::*p1 = &D::c;
  char B::*p2 = static_cast<char B::*>(p1); // undefined behavior, B does not contain the
                                            // original member c
}
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	
`expr.static.cast.does.not.contain.original.member`		✔							✔	✔							CE

Implementation Experience

Existing Practice

GCC's constant evaluator accepts examples of this behavior silently, while Clang rejects it during constant evaluation; nothing catches it at run time.

[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

Description

Converting an integral or enumeration value to a pointer with `reinterpret_cast` is only meaningful if some actual pointer exists that would convert back to that same integral value. If no such pointer exists (the value corresponds to no valid provenance at all), the conversion is undefined behavior – this is a rule about pointer *provenance*, not merely about the numeric address value.

```
void f() {
  int x, y;             // may be at consecutive addresses
  int *p = (&x) + 1;    // one past x
  std::uintptr_t pval = reinterpret_cast<std::uintptr_t>(p);
  if (pval == reinterpret_cast<std::uintptr_t>(&y)) {
    int *q = reinterpret_cast<int*>(pval);   // undefined behavior
  }
}
```

Wording

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.3.20[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*]

UB Annex Entry

F.3.20 **[ub:expr.reinterpret.cast.invalid.pointer.value]**
Specified in: 7.6.1.10[ubx:expr.reinterpret.cast.invalid.pointer.value]
Converting a value of integral or enumeration type to a pointer using `reinterpret_cast`

where there is no pointer that would convert back to that integral or enumeration value has undefined behavior.

2 [*Example 1*: In the following example, undefined behavior will occur when converting `pval` to a pointer. The results based on `x` or `y` would lead to undefined behavior, as would any result based on other objects (that share storage with `y`) that are outside of lifetime.

```
#include <cstdint>
void f()
{
  int x,y;          // can be at consecutive addresses in memory
  int *p = (&x)+1;  // pointer past the end of x
  std::uintptr_t pval = reinterpret_cast<std::uintptr_t>(p);
  if (pval == reinterpret_cast<std::uintptr_t>(&y)) {   // consider when x and y are consecutive
    int *q = reinterpret_cast<int*>(pval);  // undefined behavior
    q-1;                                    // undefined behavior if q is &y
    *q;                                     // undefined behavior if q is (&x)+1 or refers to some int outside f
  }
}
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**noexcept**-*enforce*	**noexcept**-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**noexcept**-*enforce*	**noexcept**-*observe*	
`expr.reinterpret.cast.invalid.pointer.value`		✔								✔							CE

Implementation Experience

Existing Practice

No stock tool catches this behavior at runtime. Forming the example during constant evaluation requires uses of `reinterpret_cast` that are not possible during constant evaluation.

[expr.unary.dereference]

summary:	dereferencing a pointer that is null or otherwise does not point to a live 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` caught by a native P3850 check (all available semantics on Clang; four non-throwing semantics on GCC) and constant evaluation, both compilers; `.invalid` caught heuristically by AddressSanitizer and by constant evaluation, both compilers.

Description

Dereferencing a pointer that does not currently point to an object or function is undefined behavior. The clearest case is a null pointer, but the rule is broader: any pointer that has stopped pointing to a live object (for example, one to freed storage) falls under it too.

```
int f() {
  int *p = nullptr;
  return *p;   // undefined behavior
}
```

- `expr.unary.dereference.nullptr` – the pointer is null
- `expr.unary.dereference.invalid` – the pointer is non-null but does not point to a live object (for example, a dangling pointer to freed storage)

These two are exhaustive: a pointer either is null or is not, and a non-null pointer that fails to point to an object is exactly the "invalid" case.

Wording

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.3.21[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*]

UB Annex Entry

F.3.21 **[ub:expr.unary.dereference]**

Specified in: 7.6.2.2[ubx:expr.unary.dereference]

Dereferencing a pointer that does not point to an object or function has undefined behavior.

2 [*Example 1*:

```
int f()
{
  int *p = nullptr;
  return *p;    // undefined behavior
}
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	
`expr.unary.dereference`	✔	✔							✔	✔							
↪ `...nullptr`	✔	✔	P3850	P3850 UBSan			P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE
↪ `...invalid`	✔	✔					ASan† ASan†	ASan† ASan†	✔	✔					ASan† ASan†	ASan† ASan†	CE

Implementation Experience

Existing Practice

The `.invalid` subset – dereferencing a non-null but otherwise invalid pointer (dangling, freed, or never validly formed) – is `ub:expr.unary.dereference.invalid`. AddressSanitizer (`-fsanitize=address`) catches many instances of it whenever the invalid pointer traces back to a tracked allocation (heap-use-after-free by default; stack-use-after-scope/-return and global/heap/stack-buffer-overflow with the corresponding flags), reporting then terminating (***noexcept*-enforce**) by default, and continuing (***noexcept*-observe**) under `ASAN_OPTIONS=halt_on_error=0`. It does not cover a pointer that was never validly formed at all (e.g., one built from an arbitrary integer) – such a pointer typically dereferences to a plain segmentation fault, which ASan reports generically rather than diagnosing as a specific error kind. Detection is heuristic – it depends on the freed allocation still being in ASan's quarantine (which evicts by size) and on the access landing where a redzone or poisoned shadow can notice it. The same example demonstrates this case: https://godbolt.org/z/rx8sc79c4 and https://godbolt.org/z/M3aavEfh6. The representative use-after-free is additionally caught during constant evaluation on both compilers – dereferencing a deleted object is not a constant expression – which is why the `.invalid` row also carries a *constexpr* check.

For the `.nullptr` subset specifically, UBSan's `null` check (part of the `undefined` group) instruments the dereference directly on both compilers.

A null dereference is also rejected during constant evaluation on both compilers.

P3850 Implementation

To enable this check in either compiler, `-fcontracts-p3100` must be specified and the group `ub:expr.unary.dereference.nullptr` given a semantic other than *assume*.

The check covers only a *null*-pointer dereference (`*p` with `p` null), not every invalid unary dereference this instance names, so the group id is qualified accordingly (`.nullptr`). It is a middle-end check. Because `*p` is an lvalue with no defined substitute, there is no value replacement: the continuing semantics run the handler, if any, and then proceed into the real (still undefined) dereference, while the enforcing semantics stop before it.

GCC: The GCC prototype reuses the compiler's existing null-check instrumentation, resolving the semantic per site. Because that check runs after exception-handling has been lowered, the handler cannot propagate an exception, so only the nonthrowing semantics are available: *assume*, *ignore*, *quick-enforce*, ***noexcept**-observe*, and ***noexcept**-enforce*. Examples are available on https://godbolt.org/z/W5cMsGvT6.

Clang: Clang reuses its existing null-check code generation together with a no-value reaction; because that code generation is in an exception-handling context, it supports all available semantics – ahead of GCC here. Examples are available on https://godbolt.org/z/vx3Kbfazf.

[expr.new.non.allocating.null]

summary:	a user-replaced non-allocating placement-`new` returning 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), which flags the null pointer returned by the non-allocating placement `new`, and by both compilers' constant evaluators.

Description

A non-allocating placement-`new` form (such as `operator new(size_t, void*)`) is defined to never fail and always return its `ptr` argument. If a user-replaced version of such a function returns null anyway, the subsequent placement-`new` expression using it is undefined behavior – there is no allocation-failure protocol for a form that is specified as unable to fail.

```
struct A { int x; };

int main() {
  char *p = nullptr;
  A *a = new (p) A;   // undefined behavior: non-allocating placement new
                      // "returned" a null pointer
}
```

Wording

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.3.22[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.

UB Annex Entry

F.3.22 **[ub:expr.new.non.allocating.null]**
Specified in: 7.6.2.8[ubx:expr.new.non.allocating.null]
If the allocation function is a non-allocating form (17.6.3.4[new.delete.placement]) that returns null, the behavior is undefined.

2 [*Example 1*:

```
#include <new>
[[nodiscard]] void* operator new(std::size_t size, void* ptr) noexcept {
  return nullptr;       // undefined behavior, should return non-null pointer
}
```

— *end example*]

[*Example 2*:

```
#include <new>

struct A {
  int x;
};

int main() {
  char *p = nullptr;
  A *a = new (p) A;     // undefined behavior, non-allocating new returning nullptr
}
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	
`expr.new.non.allocating.null`	✔	✔		UBSan			UBSan UBSan	UBSan UBSan	✔	✔		UBSan			UBSan UBSan	UBSan UBSan	CE

Implementation Experience

Existing Practice

UBSan's `null` check (part of the `undefined` group) instruments the result of a non-allocating placement `new` the same way it instruments any other null-pointer formation, and covers this category on both compilers.

The construction is additionally rejected during constant evaluation on both compilers ("construction of dereferenced null pointer is not allowed in a constant expression").

[expr.delete.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; rejected during constant evaluation on both GCC and Clang.

Description

`new` and `new[]` use different bookkeeping (an array allocation typically stores the element count alongside the data); the corresponding delete form must match. Using `delete[]` on a pointer obtained from a single-object `new` expression is undefined behavior.

```
int* x = new int;
delete[] x;   // undefined behavior: x was allocated with single-object new
```

Wording

Notes regarding 7.6.2.9[expr.delete], paragraph 2:

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.3.23[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.3.24[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.

UB Annex Entry

F.3.23 **[ub:expr.delete.mismatch]**

Specified in: 7.6.2.9[ubx:expr.delete.mismatch]

Using array delete on the result of a single object new expression has undefined behavior.

2 [*Example 1*:

```
int* x = new int;
delete[] x;          // undefined behavior, allocated using single object new expression
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**`noexcept`**-*enforce*	**`noexcept`**-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**`noexcept`**-*enforce*	**`noexcept`**-*observe*	
`expr.delete.mismatch`	✔	✔					ASan ASan	ASan ASan	✔	✔					ASan ASan	ASan ASan	CE

Implementation Experience

Existing Practice

When the `new` and the `delete[]` are visible together, both compilers already reject it at compile time with a warning (`-Wmismatched-new-delete`, part of `-Wall`).

At run time, AddressSanitizer's allocator records how each live allocation was made – single-object `new`, array `new[]`, or `malloc` – and compares that against the deallocation form used, via its `alloc_dealloc_mismatch` check, catching the mismatch consistently as long as the allocation and deallocation are both in translation units where AddressSanitizer has been enabled. Unlike ASan's use-after-free and buffer-overflow checks, this is a precise, deterministic detection (no quarantine or redzone heuristic).

The mismatch is also rejected during constant evaluation on both compilers ("array deallocation of object allocated with non-array allocation").

[expr.delete.array.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; rejected during constant evaluation on both GCC and Clang.

Description

`new` and `new[]` use different bookkeeping (an array allocation typically stores the element count alongside the data so that the proper number of objects may have their destructors invoked); the corresponding delete form must match. Using single-object `delete` on a pointer obtained from an array `new[]` expression is undefined behavior.

```
int* x = new int[10];
delete x;   // undefined behavior: x was allocated with array new
```

Wording

Notes regarding 7.6.2.9[expr.delete], paragraph 2:

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.3.23[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.3.24[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.

UB Annex Entry

F.3.24 **[ub:expr.delete.array.mismatch]**

Specified in: 7.6.2.9[ubx:expr.delete.array.mismatch]

Using single object delete on the result of an array new expression has undefined behavior.

2 [*Example 1*:

```
int* x = new int[10];
delete x;          // undefined behavior, allocated using array new expression
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-*enforce***	***noexcept*-*observe***	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-*enforce***	***noexcept*-*observe***	
`expr.delete.array.mismatch`	✔	✔					ASan ASan	ASan ASan	✔	✔					ASan ASan	ASan ASan	CE

Implementation Experience

Existing Practice

A single-object `delete` on a pointer obtained from an array `new[]` is caught by three channels: the `-Wmismatched-new-delete` compile warning (when the two are statically visible), constant evaluation ("non-array deallocation of object allocated with array allocation"), and ASan's `alloc_dealloc_mismatch` check at run time (default-on, precise, both compilers; ***noexcept*-*enforce*** by default, with the stock ***noexcept*-*observe*** continuing case needing the `ASAN_OPTIONS=halt_on_error=0` environment variable, and no *quick-enforce*).

[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 by ASan's `new-delete-type-mismatch` on both compilers via allocation-size mismatch; rejected during constant evaluation on Clang.

Description

`delete` on a pointer whose static type differs from the pointed-to object's actual (dynamic) type needs a virtual destructor to find the right destructor and the object's full extent – without one (and without a destroying operator delete), the deallocation has no way to know it, and the behavior is undefined.

```
struct B { int a; };
struct D : public B { int b; };

void f() {
  B* b = new D;
  delete b;   // undefined behavior: B has no virtual destructor
}
```

Wording

Notes regarding 7.6.2.9[expr.delete], paragraph 3:

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.3.25[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.3.26[ubx:expr.delete.dynamic.array.dynamic.type.differ]).

UB Annex Entry

F.3.25 **[ub:expr.delete.dynamic.type.differ]**

Specified in: 7.6.2.9[ubx:expr.delete.dynamic.type.differ]

If the static type of the object to be deleted is different from its dynamic type and the selected deallocation function 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.

2 [*Example 1*:

```
struct B {
  int a;
};

struct D : public B {
  int b;
};

void f() {
  B* b = new D;
  delete b;              // undefined behavior, no virtual destructor
}
```

— end example]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept**-enforce*	***noexcept**-observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept**-enforce*	***noexcept**-observe*	
`expr.delete.dynamic.type.differ`		✔					ASan ASan	ASan	✔	✔					ASan ASan	ASan	CE

Implementation Experience

Existing Practice

AddressSanitizer's `new-delete-type-mismatch` check reports this issue. It is caught on both compilers, deterministically, and prints a report and terminates by default (***noexcept**-enforce*). The mismatch is also rejected during constant evaluation on Clang, though GCC accepts it.

[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:	Partially caught by ASan's `new-delete-type-mismatch` on both compilers when element sizes differ; rejected during constant evaluation on both GCC and Clang.

Description

An array `delete[]` expression requires the static and dynamic element types to match exactly: a virtual destructor does not help here, since there is no "array vtable slot" to fix up the element stride or count when the static element type differs from the dynamic one, and the behavior is undefined.

```
struct B { virtual ~B(); void operator delete[](void*, std::size_t); };
struct D : B { void operator delete[](void*, std::size_t); };

void f(int i) {
  B* bp = new D[i];
  delete[] bp;   // undefined behavior: static type B differs from
                 // dynamic type D[]
}
```

Wording

Notes regarding 7.6.2.9[expr.delete], paragraph 3:

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.3.25[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.3.26[ubx:expr.delete.dynamic.array.dynamic.type.differ]).

UB Annex Entry

F.3.26 **[ub:expr.delete.dynamic.array.dynamic.type.differ]**
Specified in: 7.6.2.9[ubx:expr.delete.dynamic.array.dynamic.type.differ]
In an array delete expression, if the dynamic type of the object to be deleted differs from its static type, the behavior is undefined.

2 [*Example 1*:

```
struct B {
  virtual ~B();
  void operator delete[](void*, std::size_t);
};
```

```
struct D : B {
  void operator delete[](void*, std::size_t);
};

void f(int i) {
  D* dp = new D[i];
  delete[] dp;          // uses D::operator delete[](void*, std::size_t)
  B* bp = new D[i];
  delete[] bp;          // undefined behavior
}
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept-enforce***	***noexcept-observe***	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept-enforce***	***noexcept-observe***	
`expr.delete.dynamic.array.dynamic.type.differ`	✔	✔					ASan ASan	ASan	✔	✔					ASan ASan	ASan	CE

Implementation Experience

Existing Practice

AddressSanitizer's `new-delete-type-mismatch` check catches this problem on both compilers when the size of the types differ. This requires the two element types to differ in size; a same-layout base/derived pair produces no observable divergence and no report, since the mismatch check keys on allocation size rather than on static-versus-dynamic type identity alone. Constant evaluation also rejects it on both compilers. There is no P3850 native check.

[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) and by both compilers' constant evaluators.

Description

In a pointer-to-member access `E1.*E2`, the object named by `E1` must actually contain the member `E2` refers to. The wording states this as a requirement on the class relationship: the class of which `E2`'s member is a member must be similar to, or a base class of, the dynamic type of `E1`. If `E1`'s dynamic type does not contain that member – typically because `E1` was reached via an under-qualified `static_cast` – the access is undefined behavior.

```
struct B {};
struct D : B { int x; };

void f() {
  B *b = new B;
  D *d = static_cast<D *>(b);
  int D::*p = &D::x;
  (*d).*p = 1;   // undefined behavior: the dynamic type (B) does not
                 // contain x
}
```

Wording

Notes regarding 7.6.4[expr.mptr.oper], paragraph 4:

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

UB Annex Entry

F.3.27 **[ub:expr.mptr.oper.not.contain.member]**
Specified in: 7.6.4[ubx:expr.mptr.oper.not.contain.member]
Abbreviating *pm-expression*.***cast-expression* as `E1.*E2`, `E1` is called the object expression. If the dynamic type of `E1` does not contain the member to which `E2` refers, the behavior is undefined.

2 [*Example 1*:

```
struct B {};
struct D : B {
  int x;
};

void f() {
  B *b = new B;
  D *d = static_cast<D *>(b);
  int D::*p = &D::x;
  (*d).*p = 1;          // undefined behavior, dynamic type B does not contain x
}
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**noexcept**-*enforce*	**noexcept**-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**noexcept**-*enforce*	**noexcept**-*observe*	
expr.mptr.oper.not.contain.member	✔	✔					ASan† ASan	ASan	✔	✔					ASan† ASan	ASan	CE

Implementation Experience

Existing Practice

This category applies a pointer-to-member through a glvalue whose *dynamic* type does not contain the member – here, a plain `B` reached through an unchecked `static_cast<D*>` (with `D` deriving from `B` and adding the member `x`). Both compilers' constant evaluators reject the trigger, though for different reasons: GCC flags the write as reaching outside the constructed object, while Clang flags the invalid downcast itself.

At run time, whenever the missing member is one the derived class *adds* (rather than merely aliases within an already-same-sized layout), an object whose true dynamic type is the base is necessarily too small to hold that member, so reaching it is physically a read or write past the end of the real object. AddressSanitizer (`-fsanitize=address`) catches that overrun on both compilers. Detection is heuristic rather than a deterministic check of the category – it depends on the missing member falling outside the smaller object's storage and on ASan's redzone being positioned to notice, not on any direct "wrong dynamic type" test.

[expr.mptr.oper.member.func.null]

summary:	calling through a null pointer-to-member-function.
runtime-checkable:	Yes
locally-checkable:	Yes
checking:	Insert null pointer check
replacement:	None
implementations:	Caught by both compilers' constant evaluators.

Description

A pointer-to-member-function can hold the null member-pointer value, the member-pointer analogue of a null pointer. Using `.*` (or `->*`) with a null member pointer as the second operand – effectively "calling through" it – is undefined behavior.

```
struct S { int f(); };

void f() {
  S cs;
  int (S::*pm)() = nullptr;
  (cs.*pm)();   // undefined behavior: pm is the null member-pointer value
}
```

Wording

Notes regarding 7.6.4[expr.mptr.oper], paragraph 6:

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.3.28[ubx:expr.mptr.oper.member.func.null]).

UB Annex Entry

F.3.28 **[ub:expr.mptr.oper.member.func.null]**
Specified in: 7.6.4[ubx:expr.mptr.oper.member.func.null]

If the second operand in a `.*` expression is the null member pointer value (7.3.13[conv.mem]), the behavior is undefined.

2 [*Example 1*:

```
struct S {
  int f();
};

void f() {
  S cs;
  int (S::*pm)() = nullptr;
  (cs.*pm)();           // undefined behavior, the second operand is null
}
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept-enforce***	***noexcept-observe***	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept-enforce***	***noexcept-observe***	
`expr.mptr.oper.member.func.null`	✔	✔							✔	✔							CE

Implementation Experience

Existing Practice

Calling through a null pointer-to-member-function is caught during constant evaluation, where both compilers reject it (GCC: the null value "does not designate a `constexpr` function"; Clang: "subexpression not valid in a constant expression").

[expr.add.out.of.bounds]

summary:	forming a pointer beyond an array's one-past-the-end bound.
runtime-checkable:	Yes
locally-checkable:	Only if the array bound is statically known
checking:	Track pointer provenance, insert bounds check
replacement:	None
implementations:	`.known` fully checked by the P3850 bounds check (all semantics, both compilers) and rejected during constant evaluation on both compilers; `.unknown` only heuristically by AddressSanitizer, and also rejected during constant evaluation on both compilers; `.pointer.arithmetic` by UBSan's `pointer-overflow` check on both compilers.

Description

Adding an integer to (or subtracting one from) a pointer can only validly produce a pointer to an element of the same array, or one past its end. *Forming* a pointer further out of bounds than that is undefined behavior – independent of whether the resulting pointer is ever dereferenced.

```
static const int arr[10]{};
int main() {
  const int *x = arr + 11;   // undefined behavior: out of bounds, even
                               // though x is never dereferenced
}
```

- `expr.add.out.of.bounds.known` – the array's extent is known at compile time (a fixed-size array type)
- `expr.add.out.of.bounds.unknown` – the array's extent is not statically known (for example, a pointer into heap-allocated storage)
- `expr.add.out.of.bounds.pointer.arithmetic` – the address computation *itself* overflows the pointer's value representation (for example, adding a large offset to a pointer near the top of the address space), independent of any array bound

The first two are exhaustive for landing outside a *determinable array*: such arithmetic is performed either on an array whose bound the compiler can see, or one whose bound it cannot. The third is a distinct failure of the same [expr.add] rule – the arithmetic *overflowing* rather than merely leaving an array – and is the case UBSan's dedicated `pointer-overflow` check reports.

Wording

Modify and note section 7.6.6[expr.add], paragraph 4:

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.3.31[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*]

UB Annex Entry

F.3.31 **[ub:expr.add.out.of.bounds]**

Specified in: 7.6.6[ubx:expr.add.out.of.bounds]

Creating an out of bounds pointer has undefined behavior.

2 [*Example 1*:

```
static const int arr[10]{};
static const int arrs[10][10]{};

int main() {
  const int *x = arr + 11;          // undefined behavior, creating an out of bounds pointer
  const int(*y)[10] = arrs + 11;    // undefined behavior, creating an out of bounds pointer
}
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	*noexcept-enforce*	*noexcept-observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	*noexcept-enforce*	*noexcept-observe*	
`expr.add.out.of.bounds`		✔								✔							
↪ `...known`	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan **UBSan**	P3850 UBSan **UBSan**	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan **UBSan**	P3850 UBSan **UBSan**	CE
↪ `...unknown`	✔	✔					ASan† **ASan†**	ASan† **ASan†**	✔	✔					ASan† **ASan†**	ASan† **ASan†**	CE
↪ `...pointer.arithmetic`		✔		UBSan			UBSan **UBSan**	UBSan **UBSan**		✔		UBSan			UBSan **UBSan**	UBSan **UBSan**	CE

Implementation Experience

Existing Practice

For the `.known` subset (statically-known array extent), UBSan's `bounds` check (`-fsanitize=bounds`, independent of `-fsanitize=address`) catches every instance on both compilers. All three

achievable-without-a-handler behaviors are compile-time-selectable on GCC (*quick-enforce* via `-fsanitize-trap=bounds`, *`noexcept`-enforce* via `-fno-sanitize-recover=bounds`, *`noexcept`-observe* by default); Clang supports *quick-enforce* and *`noexcept`-enforce* (its default) but its `bounds` check cannot recover at all – `-fsanitize-recover=bounds` has no effect and produces no warning. Examples are available on https://godbolt.org/z/bW1EM1ave and https://godbolt.org/z/zq4scWWhc.

Pointer arithmetic whose bound is not statically known – the `.unknown` subset – is not covered by `bounds` at all (it needs a statically-typed array, not a raw pointer from `new[]`), but AddressSanitizer (`-fsanitize=address`) incidentally catches many instances of it on an unmodified compiler: a resulting pointer that is later dereferenced, and that lands inside a poisoned redzone or another live allocation's shadow region, is reported as a heap-buffer-overflow (or global/stack-buffer-overflow, for the corresponding storage durations). This is heuristic rather than a deterministic check of the whole subset – ASan only reports what a subsequent access actually touches, an out-of-range offset beyond its redzone (capped at roughly 2048 bytes) can land silently inside another live allocation, and a pointer that is only ever compared, never dereferenced, is never checked at all.

ASan also catches part of `.known` – a stack-array overrun is reported as a `stack-buffer-overflow` through the exact same instrumented-access path that reports `.unknown`'s heap-buffer-overflow. This is not evidence that ASan's own coverage lines up with the `.known`/`.unknown` split – it doesn't. ASan's boundary is storage duration (stack/global vs. heap), which cuts across both subsets; `bounds`'s boundary (statically-typed array vs. raw pointer) is what actually defines `.known`/`.unknown` here.

The `.known` and `.unknown` subsets are also caught during constant evaluation, on both compilers. The `.pointer.arithmetic` subset has no constant-evaluation counterpart: the wild base pointer it needs can only be formed by a `reinterpret_cast` from an integer, which is not a constant expression, so the overflow can never arise in a `constexpr` context in the first place.

P3850 Implementation

The `.known` category can be checked using the integrated frontend checks in the P3850 prototypes. To enable this check in either compiler, `-fcontracts-p3100` must be specified and the group `ub:expr.add.out.of.bounds.known` given a semantic other than *assume*.

When continuing past a violation without terminating – the *ignore*, *observe*, and *`noexcept`-observe* semantics – an out-of-range subscript is redirected to the defined valid index 0 (for reads and writes alike). This behavior is still unconstrained, but does avoid out-of-bounds editing of memory.

GCC: The GCC prototype supports all seven evaluation semantics, via a front-end guard hooked into the compiler's existing `-fsanitize=bounds` array-reference walk, so that walk's address-of and one-past context is honored (a legal one-past `&a[N]` is not redirected).

Clang: The Clang prototype likewise supports all available semantics, emitting the guard during code generation of the array subscript.

[expr.add.sub.diff.pointers]

summary:	subtracting pointers into two different, unrelated arrays.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track pointer provenance, verify both operands point into the same array object
replacement:	None
implementations:	caught at run time by AddressSanitizer's `pointer-subtract` check (stock and integrated) on both compilers, and rejected during constant evaluation on both GCC and Clang.

Description

Pointer subtraction is only meaningful for two pointers into the same array object. Subtracting pointers that point into two different, unrelated arrays is undefined behavior – there is no shared index space for the result to describe.

```
void f() {
  int x[2];
  int y[2];
  int* p1 = x;
  int* p2 = y;
  std::ptrdiff_t off = p1 - p2;   // undefined behavior: different arrays
}
```

Wording

Modify and note section 7.6.6[expr.add], paragraph 5:

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 the value $i-j$ 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 that this does not occur (F.3.32[ubx:expr.sub.pointers.representable]).

(5.3) — Otherwise, the behavior is undefined (F.3.33[ubx:expr.add.sub.diff.pointers]).

UB Annex Entry

F.3.33 **[ub:expr.add.sub.diff.pointers]**

Specified in: 7.6.6[ubx:expr.add.sub.diff.pointers]

Subtracting pointers that are not part of the same array has undefined behavior.

2 [*Example 1*:

```
#include <cstddef>
void f() {
  int x[2];
  int y[2];
  int* p1 = x;
  int* p2 = y;
  std::ptrdiff_t off = p1 - p2; // undefined behavior, p1 and p2 point to different arrays
}
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept-enforce***	***noexcept-observe***	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept-enforce***	***noexcept-observe***	
`expr.add.sub.diff.pointers`	✔	✔					ASan ASan	ASan	✔	✔					ASan ASan	ASan	CE

Implementation Experience

Existing Practice

Constant evaluation rejects this category: subtracting two pointers into different arrays is a compile error on both compilers' constant evaluators, with an object-identity-aware diagnostic ("arithmetic involving unrelated objects ... has unspecified value" on Clang). Bare `-fsanitize=undefined` and `-fsanitize=address` emit nothing – the subtraction looks like ordinary pointer arithmetic – but AddressSanitizer's dedicated `pointer-subtract` check *does* catch it at run time: with the runtime flag `detect_invalid_pointer_pairs` set, ASan recognizes that the two operands belong to different objects. That check rides the ASan runtime (hence it must be combined with `-fsanitize=address`), and under `-fcontracts-p3100` + `-fsanitize-semantic=pointer-subtract:...` it is routed to the contract-violation handler as an implicit assertion, exactly like the address check itself.

[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:	Rejected during constant evaluation on both compilers.

Description

Pointer arithmetic (`P + J`) advances by multiples of the *pointer's* pointee type size, on the assumption that the pointer walks through an array of that same type. If the pointer's type is not similar to the array's actual element type – for example, a base-class pointer walking through an array of a larger derived type – the arithmetic silently computes the wrong address, and the behavior is undefined.

```
struct S { int i; };
struct T : S { double d; };   // sizeof(T) > sizeof(S)

void f(const S* s, std::size_t count) {
  for (const S* end = s + count; s != end; ++s) {
    // undefined behavior: T is not similar to S, but s actually
    // walks a T[]
  }
}
```

Wording

Modify and note section 7.6.6[expr.add], paragraph 6:

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.3.34[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*]

UB Annex Entry

F.3.34 **[ub:expr.add.not.similar]**

Specified in: 7.6.6[ubx:expr.add.not.similar]

For addition or subtraction of two expressions P and Q, if P or Q have type "pointer to *cv* `T`", where `T` and the array element type are not similar (7.3.5[conv.rval]), the behavior is undefined.

2 [*Example 1*:

```
struct S     {  int i;    };
struct T : S {  double d; };

void f(const S* s, std::size_t count) {
  for (const S* end = s + count; s != end; ++s) {
        // undefined behavior, T is not similar to S but s points to a T[]
    /* ... */
  }
}

int main() {
  T test[5];
  f(test, 5);
}
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	
`expr.add.not.similar`	✔	✔							✔	✔							CE

Implementation Experience

Existing Practice

This category is pointer arithmetic on a "pointer to `S`" that actually walks an array of some `T` not similar to `S` (here `T` derives from `S`, so the base-class pointer conversion is implicit): the pointer advances by `sizeof(S)` per step and so does not track array elements at all.

It is caught only during constant evaluation, where both compilers reject it (GCC: the dereference "is not a constant expression"; Clang: "cannot access field of pointer past the end of object") – the general "pointer arithmetic cannot walk off a non-array subobject" rule happens to subsume this not-similar-type shape. No stock runtime sanitizer catches it: the walk stays entirely within the bounds of the overall array's storage, so `-fsanitize=address` has no redzone to trip, and `-fsanitize=undefined` ships no "wrong element type / not similar" check. The behavior can be undefined entirely *within* valid storage, so nothing at run time notices.

[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:	None

Description

An assignment's source and destination are generally assumed not to overlap in storage (unless the specific operation is documented to tolerate it, like a class's own copy assignment). When the two do overlap – typically because a `reinterpret_cast` has aliased one object's storage as another type – the result of the assignment can be undefined behavior.

```
int x = 1;
char* c = reinterpret_cast<char*>(&x);
x = *c;   // undefined behavior: the source (*c) overlaps the storage of x
```

Wording

Notes regarding 7.6.19[expr.assign], paragraph 8:

8 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.3.36[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*]

UB Annex Entry

F.3.36 **[ub:expr.assign.overlap]**

Specified in: 7.6.19[ubx:expr.assign.overlap]

Overlap in the storage between the source and destination can result in undefined behavior.

2 [*Example 1*:

```
int x = 1;
char* c = reinterpret_cast<char*>(&x);
x = *c;         // undefined behavior, source overlaps storage of destination
```

— *end example*]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	
`expr.assign.overlap`		✔								✔							CE

Implementation Experience

Existing Practice

No tool detects this category. The undefined behavior also cannot be manifested during `constexpr` evaluation as it would require the use of `reinterpret_cast`.

[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.

Description

A coroutine's body does not have an ordinary `return`, so its promise type's `return_void`/`return_value` establishes the result instead. If the promise type requires a value (has `return_value`, not `return_void`) and the coroutine body flows off the end without a `co_return`, there is no way to supply that value, and the behavior is undefined.

```
resumable foo() {
  std::cout << "Hello" << std::endl;
  co_await std::suspend_always();
  // undefined behavior: falls off the end of a coroutine whose promise
  // type requires a return value (return_value, not return_void)
}
```

Wording

Notes regarding 8.8.5[stmt.return.coroutine], paragraph 4:

4 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]).

UB Annex Entry

F.4.2 **[ub:stmt.return.coroutine.flow.off]**

Specified in: 8.8.5[ubx:stmt.return.coroutine.flow.off]

Flowing off the end of a coroutine function body that does not return `void` has undefined behavior.

2 [*Example 1*:

```
#include <cassert>
#include <coroutine>
#include <iostream>
#include <vector>

class resumable {
 public:
  struct promise_type;
  using coro_handle = std::coroutine_handle<promise_type>;
  resumable(coro_handle handle) : handle_(handle) { assert(handle); }
```

```
  resumable(resumable&) = delete;
  resumable(resumable&&) = delete;
  bool resume() {
    if (not handle_.done())
      handle_.resume();
    return not handle_.done();
  }
  ~resumable() { handle_.destroy(); }
  const char* return_val();

 private:
  coro_handle handle_;
};

struct resumable::promise_type {
  using coro_handle = std::coroutine_handle<promise_type>;
  const char* string_;
  auto get_return_object() { return coro_handle::from_promise(*this); }
  auto initial_suspend() { return std::suspend_always(); }
  auto final_suspend() noexcept { return std::suspend_always(); }
  void unhandled_exception() { std::terminate(); }
  void return_value(const char* string) { string_ = string; }
};

const char* resumable::return_val() {
  return handle_.promise().string_;
}

resumable foo() {
  std::cout << "Hello" << std::endl;
  co_await std::suspend_always();
  // undefined behavior, falling off the end of coroutine that does not return void
}

int main() {
  resumable res = foo();
  while (res.resume())
    ;
  std::cout << res.return_val() << std::endl;
}
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	
`stmt.return.coroutine.flow.off`		✔	P3850	P3850	P3850	P3850	P3850	P3850		✔	P3850	P3850	P3850	P3850	P3850	P3850	CE

Implementation Experience

Existing Practice

No stock sanitizer catches this category. In particular, UBSan's `return` check does *not* fire when control flows off the end of a coroutine.

Constant evaluation does not apply: a coroutine cannot be a `constexpr` or `consteval` function, so there is no constexpr-compatible trigger.

P3850 Implementation

To enable this check in either compiler, `-fcontracts-p3100` must be specified and the group `ub:stmt.return.coroutine.flow.off` given a semantic other than *assume*.

The check is synthesized at the point where control flows off the end of a coroutine whose promise type has no usable `return_void`. There is no return value to substitute, so the continuing semantics (*ignore*, *observe*, *`noexcept`-observe*) simply proceed to the coroutine's final suspend. The fall-off point is inside the coroutine's `try` block, so a throwing violation handler unwinds the body (destroying in-scope automatic objects) and is caught by `promise.unhandled_exception()`. The violation is reported as `assertion_kind::implicit`.

GCC: The GCC prototype supports all seven evaluation semantics. Examples of every semantic are available on https://godbolt.org/z/Kn5Pjf99h.

Clang: The Clang prototype likewise supports all available semantics. Examples of every semantic are available on https://godbolt.org/z/9edW7c63o.

[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

Description

A block-scope `static` variable is guarded so its initializer runs exactly once, the first time control reaches the declaration. If control recursively re-enters that same declaration while the initializer is still running – the guard has been entered but not yet exited – the behavior is undefined; there is no well-defined value for the variable to have yet.

```
int foo(int i) {
  static int s = foo(2 * i);   // undefined behavior: recursively re-enters
                                 // the declaration of s while initializing it
  return i + 1;
}
```

Wording

Notes regarding 8.10[stmt.dcl], paragraph 4:

4 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*]

UB Annex Entry

F.4.3 **[ub:stmt.dcl.local.static.init.recursive]**

Specified in: 8.10[ubx:stmt.dcl.local.static.init.recursive]

If control re-enters the declaration recursively while the variable is being initialized, the behavior is undefined.

2 [*Example 1*:

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

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**`noexcept`**-*enforce*	**`noexcept`**-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**`noexcept`**-*enforce*	**`noexcept`**-*observe*	
`stmt.dcl.local.static.init.recursive`		✔								✔							CE

Implementation Experience

Existing Practice

Objects with constant initialization cannot encounter this undefined behavior — a recursive initializer would not be a constant expression because it is attempting to initialize a function-local static variable.

For objects with dynamic initialization, a guard must be introduced that ensures that the initialization is done exactly once in a thread-safe fashion. That guard on the Itanium ABI protects against re-entering it while one initialization is in progress. The exact response is dependent on the particular standard library providing the Itanium implementation:

- **GCC / libstdc++** throws the runtime's recursive-initialization error (a `std::exception` subclass) from inside the guard-acquire routine. With nothing to catch it, the program terminates (`terminate called after throwing ...`).
- **Clang / libc++abi** does not throw: the guard-acquire routine issues a fatal diagnostic directly ("detected recursive initialization ...") and aborts. There is no exception to catch.

[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 both compilers.

Description

`const` is not merely a compiler-enforced typing discipline; it is also a runtime promise the implementation may rely on. Modifying a `const` object during its lifetime is undefined behavior, even when a `const_cast` is used to obtain a non-`const` pointer or reference through which to do it.

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

Wording

Notes regarding 9.2.9.2[dcl.type.cv], paragraph 5:

5 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*]

UB Annex Entry

F.5.1 **[ub:dcl.type.cv.modify.const.obj]**

Specified in: 9.2.9.2[ubx:dcl.type.cv.modify.const.obj]

Any attempt to modify a const object during its lifetime results in undefined behavior.

2 [*Example 1*:

```
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
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	
`dcl.type.cv.modify.const.obj`	✔	✔							✔	✔							CE

Implementation Experience

Existing Practice

This category is detected only during constant evaluation: modifying an object declared `const` is rejected as a compile error by both compilers' constant evaluators. No stock sanitizer catches it at run time – the write appears to succeed – because detecting it would require tracking a per-object const flag, which no shipping sanitizer does.

[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.

Description

Accessing an object defined with a `volatile`-qualified type through a non-volatile glvalue is undefined behavior – the optimizer is entitled to assume such an access is not subject to `volatile`'s ordering and not-elidable guarantees, which may not hold if the underlying object really is volatile.

```
volatile int x = 0;
int& y = const_cast<int&>(x);
std::cout << y;   // undefined behavior: accessing volatile x through the
                  // non-volatile glvalue y
```

Wording

Notes regarding 9.2.9.2[dcl.type.cv], paragraph 6:

6 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]).

UB Annex Entry

F.5.2 **[ub:dcl.type.cv.access.volatile]**

Specified in: 9.2.9.2[ubx:dcl.type.cv.access.volatile]

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

2 [*Example 1*:

```
volatile int x = 0;
int& y = const_cast<int&>(x);
std::cout << y;         // undefined behavior, accessing volatile through non-volatile glvalue
```

— *end example*]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-*enforce***	***noexcept*-*observe***	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-*enforce***	***noexcept*-*observe***	
`dcl.type.cv.access.volatile`		✔							✔	✔							CE

Implementation Experience

Existing Practice

Accessing a `volatile`-qualified object through a non-`volatile` glvalue (the annex's `const_cast` example) is detected only during constant evaluation, and there the two compilers diverge. Clang's evaluator tracks the *actual* underlying object and rejects the read regardless of the accessing glvalue's type ("read of volatile object ... is not allowed in a constant expression"). GCC's evaluator does not track "the underlying object is `volatile`" once the access is spelled through a non-`volatile` reference; it treats the read as an ordinary `int` read and accepts the program.

[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.

Description

A reference to a function type is a binding contract about how that function may later be called. Initializing such a reference with a function that is not call-compatible with the reference's own type – for example, via a `reinterpret_cast` between unrelated function types – is undefined behavior at the point of binding, though in practice the condition is only *detected* when the reference is actually called through.

```
void f(float x);
void (&g)(int) = reinterpret_cast<void (&)(int)>(f);   // undefined behavior
```

Wording

Notes regarding 9.3.4.3[dcl.ref], paragraph 7:

7 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]

UB Annex Entry

F.5.3 **[ub:dcl.ref.incompatible.function]**

Specified in: 9.3.4.3[ubx:dcl.ref.incompatible.function]

Initializing a reference to a function with a value that is a function that is not call-compatible (7.6.1.3[expr.call]) with the type of the reference has undefined behavior.

2 *[Example 1*:

```
void f(float x);
void (&g)(int) = reinterpret_cast<void (&)(int)>(f);    // undefined behavior
```

— end example]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	
`dcl.ref.incompatible.function`		✔								✔		UBSan			UBSan UBSan	UBSan UBSan	CE

Implementation Experience

Existing Practice

Binding (and then calling through) a reference to a function whose type is not call-compatible with the referent – the annex's `void(float)` reinterpreted as `void(&)(int)` – is covered by UBSan's Clang-only `function` check.

On Clang the check is part of the `undefined` group (so plain `-fsanitize=undefined` enables it) and diagnoses the call with an exact message ("call to function ... through pointer to incorrect function type"), supporting all three achievable-without-a-handler behaviors with compile-time flags alone: *quick-enforce* (`-fsanitize-trap=function`, silent trap), ***noexcept*-enforce** (`-fno-sanitize-recover=function`, diagnostic then terminate), and ***noexcept*-observe** (the default, which recovers and continues).

GCC has no check for function types.

Constant evaluation does not apply: the mistyped reference can only be formed through a `reinterpret_cast`, which cannot be done at compile time.

[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

Description

Initializing a reference to an object with a value that is not type-accessible through the reference's own type – the strict-aliasing rule applied at the point of reference initialization – is undefined behavior.

```
float g;
int& i = reinterpret_cast<int&>(g);   // undefined behavior
```

Wording

Notes regarding 9.3.4.3[dcl.ref], paragraph 7:

7 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*]

UB Annex Entry

F.5.4 **[ub:dcl.ref.incompatible.type]**

Specified in: 9.3.4.3[ubx:dcl.ref.incompatible.type]

Initializing a reference to an object with a value that is not type-accessible (7.2.1[basic.lval]) through the type of the reference has undefined behavior.

2 [*Example 1*:

```
float g;
int& i = reinterpret_cast<int&>(g);      // undefined behavior
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	
`dcl.ref.incompatible.type`		✔								✔							CE

Implementation Experience

Existing Practice

No tool detects this category on either compiler. Binding an `int&` to a `float` object via `reinterpret_cast` (the annex's example) and then reading through it is a scalar type-punning ("strict aliasing") violation, for which no stock check ships: `-fsanitize=undefined` has no such check, LLVM's experimental TySan (`-fsanitize=type`) is not part of the default `undefined` group and is not evaluated here, and `-fsanitize=address` has nothing out of bounds to flag (valid, in-bounds storage throughout).

Constant evaluation likewise does not apply: the mistyped reference can only be formed through a `reinterpret_cast`, not permitted in constant evaluation – inapplicable here, not unchecked. With no stock detector on either compiler, this is the cleanest "nothing catches this" case in this group.

[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 initialised
replacement:	None
implementations:	None

Description

A reference has no state of its own before it is bound to something; it is purely an alias. Evaluating a reference before it has actually been initialized – for example, an `extern` reference used before the translation unit that initializes it has run – is undefined behavior.

```
extern int &ir1;
int i2 = ir1;   // undefined behavior: ir1 is not yet initialized
int i3 = 17;
int &ir1 = i3;
```

Wording

Notes regarding 9.3.4.3[dcl.ref], paragraph 7:

7 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]

UB Annex Entry

F.5.5 **[ub:dcl.ref.uninitialized.reference]**

Specified in: 9.3.4.3[ubx:dcl.ref.uninitialized.reference]

Evaluating a reference prior to initializing that reference has undefined behavior.

2 *[Example 1:*

```
extern int &ir1;
int i2 = ir1;   // undefined behavior, ir1 not yet initialized
int i3 = 17;
int &ir1 = i3;
```

— end example]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	*noexcept-enforce*	*noexcept-observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	*noexcept-enforce*	*noexcept-observe*	
`dcl.ref.uninitialized.reference`		✔								✔							CE

Implementation Experience

Existing Practice

No tool detects this category on either compiler. The annex's example reads an `extern` reference (`ir1`) in one namespace-scope initializer before a later declaration in the same translation unit binds it. Because binding a reference to a named object of static storage duration needs only that object's *address* (a compile-time constant), both compilers bind `ir1` during static initialization – so at the point of the too-early read the reference is validly bound to a real, live object, and the read simply returns that object's pre-initialization (zero) state rather than the value its own dynamic initializer will later assign.

This is why no memory-safety sanitizer has anything to flag: there is no invalid access, only a *temporal* ordering error over valid storage, which is strictly harder for any tool to notice than an out-of-bounds or use-after-free access. Both `-fsanitize=undefined` and `-fsanitize=address` emit nothing on both compilers. There is no way to construct an example during constant evaluation because every variable must have a visible initializer prior to being usable in a constant expression at all.

[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 will catch many cases as a use-after-free.

Description

A coroutine handle's `resume()` member function expects the coroutine to actually be suspended – waiting to be resumed. Calling it on a coroutine that has already run to completion (past `final_suspend`) or is otherwise not suspended is undefined behavior, even though the handle itself remains a valid object right up until `destroy()` is called on it.

```
// generator g is fully exhausted: g.done() is true after the last
// co_yield
while (g.move_next())
  sum += g.current_value();

g.move_next();   // undefined behavior: move_next() calls coro.resume(),
                  // but the coroutine already ran past final_suspend
```

Wording

Notes regarding 9.6.4[dcl.fct.def.coroutine], paragraph 9:

9 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]).

UB Annex Entry

F.5.6 **[ub:dcl.fct.def.coroutine.resume.not.suspended]**

Specified in: 9.6.4[ubx:dcl.fct.def.coroutine.resume.not.suspended]

Invoking a resumption member function for a coroutine that is not suspended results in undefined behavior.

2 [*Example 1*:

```
#include <coroutine>

struct minig {
  struct promise_type {
    int val;
    minig get_return_object() { return {*this}; }
    constexpr suspend_always initial_suspend() noexcept { return {}; }
```

```
    constexpr suspend_always final_suspend() noexcept { return {}; }
    constexpr void return_void() noexcept {}
    [[noreturn]] void unhandled_exception() noexcept { throw; }
    suspend_always yield_value(int v) noexcept {
      val = v;
      return {};
    }
  };
  using HDL = coroutine_handle<promise_type>;
  HDL coro;
  minig(promise_type& p) : coro(HDL::from_promise(p)) {}
  ~minig() { coro.destroy(); }
  bool move_next() {
    coro.resume();
    return !coro.done();
  }
  int current_value() { return coro.promise().val; }
};

static minig f(int n) {
  for (int i = 0; i < n; ++i)
    co_yield i;
}

int main() {
  auto g = f(10);
  int sum = 0;
  while (g.move_next())
    sum += g.current_value();

  g.move_next();     // undefined behavior, will call coro.resume() but final_suspend
                     // has already returned, even though it returned suspend_always

  return sum;
}
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	
`dcl.fct.def.coroutine.resume.not.suspended`		✔								✔							CE

Implementation Experience

Existing Practice

Many cases where a coroutine's frame has been destroyed will be caught by AddressSanitizer as a use-after-free or a null pointer dereference. In general, there is no implementation experience with guarding coroutines against invalid calls to `resume()`.

Constant evaluation is inapplicable: a coroutine cannot be a `constexpr` or `consteval` function.

[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.

Description

Calling `destroy()` on a coroutine handle while that coroutine is not suspended – most dangerously, calling it on a coroutine from within its own still-executing frame – is undefined behavior, since it tears down the frame that is actively running the call.

```
Task self_destroy() {
  auto self = co_await get_own_handle{};  // obtain this frame's own handle
  self.destroy();   // undefined behavior: destroys the coroutine while its
                    // frame is still executing this call (not suspended)
  co_return;        // never reached
}
```

Wording

Notes regarding 9.6.4[dcl.fct.def.coroutine], paragraph 12:

12 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]).

UB Annex Entry

F.5.7 **[ub:dcl.fct.def.coroutine.destroy.not.suspended]**

Specified in: 9.6.4[ubx:dcl.fct.def.coroutine.destroy.not.suspended]

Invoking `destroy()` on a coroutine that is not suspended has undefined behavior.

2 [*Example 1*:

```
#include <coroutine>

struct minig {
  struct promise_type {
    int val;
    minig get_return_object() { return {*this}; }
    constexpr suspend_always initial_suspend() noexcept { return {}; }
    constexpr suspend_always final_suspend() noexcept { return {}; }
```

```
    constexpr void return_void() noexcept {}
    [[noreturn]] void unhandled_exception() { throw; }
    suspend_always yield_value(int v) noexcept {
      val = v;
      return {};
    }
  };
  using HDL = coroutine_handle<promise_type>;
  HDL coro;
  minig(promise_type& p) : coro(HDL::from_promise(p)) {}
  ~minig() { coro.destroy(); }
  bool move_next() {
    coro.resume();
    return !coro.done();
  }
  int current_value() {
    // this coroutine is not suspended therefore a call to destroy has undefined behavior
    coro.destroy();
    return coro.promise().val;
  }
};

static minig f(int n) {
  for (int i = 0; i < n; ++i)
    co_yield i;
}

int main() {
  auto g = f(10);
  int sum = 0;
  while (g.move_next())
    sum += g.current_value();

  return sum;
}
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**noexcept-enforce**	**noexcept-observe**	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**noexcept-enforce**	**noexcept-observe**	
dcl.fct.def.coroutine. destroy.not.suspended		✔					ASan† ASan†	ASan†		✔					ASan† ASan†	ASan†	CE

Implementation Experience

Existing Practice

There is no direct support for catching this category in existing sanitizers, but AddressSanitizer does catch the ways in which this issue manifests as either a use-after-free or double-free of a stack frame. Because it is ASan, the check is heuristic, not precise.

Constant evaluation is inapplicable: a coroutine cannot be a `constexpr` or `consteval` function.

[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.

Description

The `[[noreturn]]` attribute tells callers and the optimizer that a function never returns control to its caller (it always throws, calls `std::terminate`, loops forever, and so on). If a function so declared returns anyway, the behavior is undefined – the optimizer may have already generated code downstream of the call that assumes execution never resumes there.

```
[[noreturn]] void f(int i) {
  if (i > 0)
    throw "positive";
}

int main() {
  f(0);   // undefined behavior: f is declared [[noreturn]] but returns
}
```

Wording

Notes regarding 9.13.10[dcl.attr.noreturn], paragraph 3:

3 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*]

UB Annex Entry

F.5.9 **[ub:dcl.attr.noreturn.eventually.returns]**

Specified in: 9.13.10[ubx:dcl.attr.noreturn.eventually.returns]

If a function `f` is called where `f` was previously declared with the `noreturn` attribute and `f` eventually returns, the behavior is undefined.

2 [*Example 1*:

```
[[noreturn]] void f(int i) {
  if (i > 0)
    throw "positive";
}

int main() {
  f(0);         // undefined behavior, returns from a [[noreturn]] function
```

```
}
```

— end example]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept-enforce***	***noexcept-observe***	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept-enforce***	***noexcept-observe***	
`dcl.attr.noreturn.eventually.returns`		✔		UBSan			UBSan UBSan	UBSan		✔		UBSan			UBSan UBSan	UBSan	CE

Implementation Experience

Existing Practice

UBSan's `unreachable` check (part of the `undefined` group) instruments the point immediately after a call to a function declared `[[noreturn]]`, and fully covers this category on both compilers. Unlike most other UBSan checks, `unreachable` cannot recover – there is no defined way to "continue" past a `[[noreturn]]` function actually returning, matching this category's own *replacement: none* – so only two of the three achievable-without-a-handler behaviors apply, both compile-time-selectable:

- *quick-enforce*: `-fsanitize-trap=unreachable` traps silently, with no diagnostic and no callback.
- ***noexcept-enforce***: the default (there is no recoverable variant to request; `-fsanitize-recover=unreachable` is rejected at compile time on both compilers) prints a diagnostic and terminates.

The `[[noreturn]]` attribute has no effect at compile time, so no `constexpr`-based detection of violating the attribute's promise is feasible.

[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:	`.nonpolymorphic` caught only by constant evaluation (both compilers); `.polymorphic` caught at runtime by UBSan's `vptr` check on GCC only.

Description

Invoking an object's destructor ends its lifetime. Invoking a destructor again for an object whose lifetime has already ended – a double destroy – is undefined behavior, whether the second call is explicit or implicit (at scope exit).

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

int main() {
  A a;
  a.~A();
}   // undefined behavior: the implicit destructor call at scope exit runs
    // on an object whose lifetime already ended
```

- `class.dtor.no.longer.exists.polymorphic` – the object is polymorphic (has a vtable), so a runtime check against the object's dynamic type is at least possible in principle by poisoning the vtable upon destruction
- `class.dtor.no.longer.exists.nonpolymorphic` – no vtable exists, so no runtime information distinguishes a live object from an already-destroyed one

These two are exhaustive: every object is either polymorphic or not, and that is exactly the axis controlling whether any runtime check could ever exist here.

Wording

Notes regarding 11.4.7[class.dtor], paragraph 19:

19 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*]

UB Annex Entry

F.6.1 **[ub:class.dtor.no.longer.exists]**

Specified in: 11.4.7[ubx:class.dtor.no.longer.exists]

Once a destructor is invoked for an object, the object's lifetime has ended; the behavior is undefined if the destructor is invoked for an object whose lifetime has ended.

2 [*Example 1*:

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

int main() {
  A a;
  a.~A();
}   // undefined behavior, lifetime of a already ended before implicit destructor
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	
`class.dtor.no.longer.exists`		✔								✔							
↪ ...`polymorphic`		✔					UBSan†			✔							CE
↪ ...`nonpolymorphic`	✔	✔					UBSan†		✔	✔							CE

Implementation Experience

Existing Practice

This category splits by whether the object is polymorphic. The *nonpolymorphic* subset – a double destruction of an object with no vtable – is rejected during constant evaluation on both compilers but caught by no stock runtime sanitizer (the storage is still live and nothing distinguishes the second destructor call), so it is constexpr-only. The *polymorphic* subset – a second *virtual* destructor dispatch, whose invalidated vptr no longer describes a live object – can be caught at run time by UBSan `vptr`, but on GCC only; Clang does not catch it. `vptr` offers only ***noexcept*-enforce** here (no *quick-enforce* – `-fsanitize-trap=vptr` is unsupported; a recovering *observe* leaves the dead object's vptr in place, so it does not fire).

[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 on both compilers; and, in the P3850 prototype, checkable at run time on both compilers.

Description

A pure virtual function has no implementation to fall back on unless one is separately provided and called by its (unmangled) qualified name. Calling one from a constructor or destructor – directly, or indirectly through another member function – when the most-derived class has not yet overridden it (during construction) or no longer has (during destruction) is undefined behavior.

```
struct AbstractBase {
  virtual void pureVirtual() = 0;
  AbstractBase() {
    pureVirtual();   // undefined behavior: called from the constructor,
                     // before any override could apply
  }
};
struct ConcreteDerived : AbstractBase {
  void pureVirtual() override;
};
```

Wording

Notes regarding 11.7.4[class.abstract], paragraph 7:

7 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]).

UB Annex Entry

F.6.2 **[ub:class.abstract.pure.virtual]**

Specified in: 11.7.4[ubx:class.abstract.pure.virtual]

Calling a pure virtual function from a constructor or destructor in an abstract class has undefined behavior.

2 [*Example 1*:

```
struct B {
  virtual void f() = 0;
  B() {
    f();        // undefined behavior, f is pure virtual and is being called from the constructor
```

```
  }
};

struct D : B {
  void f() override;
};
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept-enforce***	***noexcept-observe***	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept-enforce***	***noexcept-observe***	
`class.abstract.pure.virtual`	✔	✔		P3850	P3850	P3850	P3850	P3850	✔	✔		P3850	P3850	P3850	P3850	P3850	CE

Implementation Experience

Existing Practice

Calling a pure virtual function from a constructor is caught during constant evaluation on both compilers (GCC: "call to non-`constexpr` function ..."; Clang: "pure virtual function ... called").

At run time both GCC and Clang invoke a stub that aborts the program in these cases, essentially always applying the *quick-enforce* semantic by default.

GCC, in some circumstances, will even devirtualize such calls and result in a link-time failure because the pure virtual function is never defined.

P3850 Implementation

To enable this check in either compiler, `-fcontracts-p3100` must be specified and the group `ub:class.abstract.pure.virtual` given a semantic other than *assume* or *ignore*.

Unlike every other implicit check, there is no per-call site to guard: a pure-virtual dispatch is an ordinary indirect call through the vtable, and only the runtime object's *current* vtable knows a slot is pure. So the check lives in the vtable itself. Where the vtable is emitted, the pure virtual's slot – which normally holds the address of `__cxa_pure_virtual` – is instead pointed at a semantic-specific terminus. The semantic is resolved at that point using the pure virtual's *declaring (base) class* location and namespace, so per-file/line and per-namespace configuration selects it per class. (The vtable is emitted once in the key function's translation unit, or per-translation-unit for a keyless class, so the configuration active where the vtable is built governs and may differ across translation units.) The slot remains a plain function pointer – only its value changes – so the vtable ABI is untouched.

GCC: `cp_pure_virtual_contract_terminus` (`cp/contracts.cc`) selects the terminus at the pure-virtual slot fill in `build_vtbl_initializer` (`cp/class.cc`); the termini are defined in `libstdc++`'s contracts runtime. Examples of every semantic are available on https://godbolt.org/z/hhrKa83cT.

Clang: `CodeGenModule::getPureVirtualContractTerminusName` (`CGContracts.cpp`) mirrors this at the slot fill in `addVTableComponent` (`CGVTables.cpp`); the termini are defined in `libc++`'s contracts runtime. Examples of every semantic are available on https://godbolt.org/z/vbjodWWGG.

[class.base.init.mem.fun]

summary:	calling a member function outside the base classes' constructed lifetime.
runtime-checkable:	Yes
locally-checkable:	No
checking:	Track whether objects are currently being constructed or destroyed
replacement:	None
implementations:	For polymorphic types, this can be caught by UBSan's `vptr`, generally consistently for GCC when nothing else has written to the object's storage.

Description

Calling a member function before all base-class mem-initializers have completed, or after base classes have already been destroyed, is undefined behavior – this includes constructor precondition assertions and destructor postcondition assertions, which run before/after construction/destruction in exactly this window.

```
class A { public: A(int); };
class B : public A {
  int j;
public:
  int f();
  B() : A(f()),   // undefined behavior: calls f() but base A is not
                  // yet initialized
        j(f()) {} // defined: all bases are initialized by now
};
```

- `class.base.init.mem.fun.polymorphic` – the call is a virtual call through a pointer to polymorphic storage, so a runtime vtable check is at least possible in principle
- `class.base.init.mem.fun.nonpolymorphic` – no vtable is involved, so no runtime information distinguishes this case at all

These two are exhaustive: the axis is exactly whether the call is a virtual dispatch through a vtable or not.

Wording

Notes regarding 11.9.3[class.base.init], paragraph 19:

19 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*]

UB Annex Entry

F.6.3 **[ub:class.base.init.mem.fun]**

Specified in: 11.9.3[ubx:class.base.init.mem.fun]

Calling a member function before all the *mem-initializer*s for base classes have completed, or after base classes have been destroyed, has undefined behavior. This includes precondition assertions of the constructor and postcondition assertions of the destructor.

2 [*Example 1*:

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

class B : public A {
  int j;

public:
  int f();
  B()
      pre( this->f() )  // undefined behavior, calls member function in constructor precondition
      : A(f()),         // undefined behavior, calls member function but base A not yet initialized
        j(f()) {}       // defined, bases are all initialized

  B(int x)
      pre( this->f() )  // undefined behavior, calls member function in delegating constructor precondition
      : B() {}

  ~B()
      post( this->f() ) // undefined behavior, calls member function in destructor postcondition
      {}
```

```
};

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()) {}        // defined, bases are all initialized
};
```

— end example]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	
`class.base.init.mem.fun`		✔								✔							
↪ `...polymorphic`		✔					UBSan† UBSan†	UBSan†		✔							CE
↪ `...nonpolymorphic`	✔	✔							✔	✔							CE

Implementation Experience

Existing Practice

This category splits by whether the call is polymorphic. The *nonpolymorphic* subset – a plain member function reading a not-yet-initialized scalar member – is rejected during constant evaluation on both compilers and caught by no stock runtime sanitizer, so it is constexpr-only. The *polymorphic* subset – a *virtual* call before the base is initialized, so the vptr does not yet describe a valid object – can be caught at run time by UBSan `vptr`, but on GCC only; Clang does not catch it.

[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:	Accesses in non-polymorphic objects are not generally caught. Accesses in polymorphic types are caught, heuristically, by UBSan's `vptr` check on both compilers. GCC accepts non-polymorphic cases at compile time while Clang does not.

Description

For an object with a non-trivial constructor, referring to any non-static member or base class of the object before its constructor has begun executing is undefined behavior – nothing has established the object's invariants yet, and (for a virtual base reached through a not-yet-started subobject) even the layout information needed to locate the member may not be ready.

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

- `class.cdtor.before.ctor.polymorphic` – the access is a virtual call through a pointer to the not-yet-constructed storage, reading an uninitialized vptr, so a runtime check is at least possible in principle
- `class.cdtor.before.ctor.nonpolymorphic` – forming a pointer to a member through a not-yet-constructed non-polymorphic (virtual base) subobject, with no vtable involved at all

These two are exhaustive: the axis is exactly whether the offending access goes through a vtable or not.

Wording

Notes regarding 11.9.5[class.cdtor], paragraph 2:

2 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*]

UB Annex Entry

F.6.4 **[ub:class.cdtor.before.ctor]**

Specified in: 11.9.5[ubx:class.cdtor.before.ctor]

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.

2 [*Example 1*:

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

— *end example*]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	
`class.cdtor.before.ctor`		✔								✔							
↪ `...polymorphic`		✔					UBSan† UBSan†	UBSan†		✔					UBSan† UBSan†	UBSan†	CE
↪ `...nonpolymorphic`		✔							✔	✔							CE

Implementation Experience

Existing Practice

This category splits by whether the object is polymorphic. UBSan's `vptr` check instruments virtual member-function calls and non-static member accesses on a polymorphic object: if the object's vptr does not describe a live object of the expected type, the check fails. The *polymorphic* subset – a virtual call through a pointer to polymorphic storage not yet constructed, can read an uninitialized vptr – which in many cases is caught by UBSan on both compilers. The *nonpolymorphic* subset is instead rejected during constant evaluation on Clang only (GCC accepts it in constant evaluation), with no stock runtime detector.

[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:	split by polymorphism – `.polymorphic` caught by UBSan's `vptr` check on GCC only; `.nonpolymorphic` rejected during constant evaluation on both compilers.

Description

For an object with a non-trivial destructor, referring to any non-static member or base class of the object after its destructor has finished executing is undefined behavior – the object's invariants have already been torn down.

```
struct X {
  int i;
  ~X();   // non-trivial
};
void f() {
  X* px = new X;
  px->~X();
  int j = px->i;   // undefined behavior
}
```

- `class.cdtor.after.dtor.polymorphic` – a virtual call through a pointer whose object's destructor has run, reading an overwritten or invalid vptr, so a runtime check is at least possible in principle
- `class.cdtor.after.dtor.nonpolymorphic` – a member read of a non-polymorphic object after its destructor finishes, with no vtable involved at all

These two are exhaustive: the axis is exactly whether the offending access goes through a vtable or not.

Wording

Notes regarding 11.9.5[class.cdtor], paragraph 2:

2 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*]

UB Annex Entry

F.6.5 **[ub:class.cdtor.after.dtor]**

Specified in: 11.9.5[ubx:class.cdtor.after.dtor]

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 has undefined behavior.

2 [*Example 1*:

```
struct X {
  int i;
  ~X();     // non-trivial
};
X& g()
{
  static X x;
  return x;
}
void f()
{
  X* px = &g();
  px->~X();
  int j = px->i;    // undefined behavior
}
```

— *end example*]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	
`class.cdtor.after.dtor`		✔								✔							
↪ `...polymorphic`		✔					UBSan† UBSan†			✔							CE
↪ `...nonpolymorphic`	✔	✔							✔	✔							CE

Implementation Experience

Existing Practice

This category splits by whether the object is polymorphic. The *polymorphic* subset – a virtual call through a pointer whose object's destructor has run, reading an overwritten/invalid vptr – is caught by UBSan `vptr` on GCC only, though that requires the invalidated vptr to not have been overwritten with something valid since the object's destruction, and is thus heuristic and not reliable. The non-polymorphic case is not checked by any sanitizers as long as the storage is still allocated.

[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

Description

Converting a pointer to a base class of an object, via a path whose construction has not started (or whose destruction has already finished), is undefined behavior – the conversion needs base-offset information that is only guaranteed valid for a base subobject that is actually under construction.

```
struct A {};
struct B : virtual A {};
struct D : virtual A { D(A*); };
struct X { X(A*); };
struct E : B, D, X {
  E() : D(this),   // undefined behavior: converts E* to A* via D's
                   // not-yet-constructed path
        X(this) {} // defined: by now the B/A sublattice is fully
                   // constructed
};
```

Wording

Notes regarding 11.9.5[class.cdtor], paragraph 4:

4 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*]

UB Annex Entry

F.6.6 [ub:class.cdtor.convert.pointer]

Specified in: 11.9.5[ubx:class.cdtor.convert.pointer]

Converting a pointer to a base class of an object, when construction of that object has not started or destruction of that object has finished, has undefined behavior.

2 [*Example 1*:

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

In the above constructor for `E`, upcast from `E*` to `A*` might use the path `E*` → `D*` → `A*`, but the `D` subobject has not yet been constructed. If the member initializer was instead "`D((C*)this)`" then that would be defined, because `E*` → `C*` is defined (`E()` has started) and `C*` → `A*` is defined (the `C` subobject is fully constructed). — *end example*]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-*enforce***	***noexcept*-*observe***	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-*enforce***	***noexcept*-*observe***	
`class.cdtor.convert.pointer`		✔								✔							CE

[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 by UBSan's `vptr` check on Clang.

Description

Merely *forming* a pointer to a direct non-static member of a class – not even using it – is undefined behavior if construction of the enclosing object has not yet started, or destruction has already finished.

```
struct A { int i = 0; };
struct B {
  int *p;
  A a;
  B() : p(&a.i) {}   // undefined behavior: forms a pointer to a.i before
                     // a's construction has started
};
```

- `class.cdtor.form.pointer.polymorphic` – the member's own class is polymorphic, so a runtime vtable check is at least possible in principle
- `class.cdtor.form.pointer.nonpolymorphic` – the member's class has no vtable, so no runtime information distinguishes this case at all

These two are exhaustive: the axis is exactly whether the class of the member being pointed to has a vtable or not.

Wording

Notes regarding 11.9.5[class.cdtor], paragraph 4:

4 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*]

UB Annex Entry

F.6.7 **[ub:class.cdtor.form.pointer]**

Specified in: 11.9.5[ubx:class.cdtor.form.pointer]

When forming a pointer to a direct non-static member of a class, construction must have started and destruction must not have finished otherwise the behavior is undefined.

2 [*Example 1*:

```
struct A {
  int i = 0;
};
struct B {
  int *p;
  A a;
  B() : p(&a.i) {}  // undefined behavior
};
```

— *end example*]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	*noexcept-enforce*	*noexcept-observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	*noexcept-enforce*	*noexcept-observe*	
`class.cdtor.form.pointer`		✔								✔							
↪ `...polymorphic`		✔								✔					UBSan† UBSan†	UBSan†	CE
↪ `...nonpolymorphic`		✔								✔							CE

Implementation Experience

Existing Practice

Forming a pointer to a non-static member before construction begins shows a compiler asymmetry. For the *polymorphic* subset – where the member's own class is polymorphic – Clang's `vptr` member-

access check rejects it when the uninitialized vtable entry is read; GCC does not catch it. The *nonpolymorphic* subset – where the member's class has no vtable – is caught by no stock tool.

[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

Description

During construction or destruction, a virtual call resolves to the final overrider in the *constructor's or destructor's own class*, not any more-derived override – that part is well-defined. But if the call is written as an explicit member access (`E1.E2` or `E1->E2`) and `E1`'s static type is not the executing constructor/destructor's own class or one of its bases, the call itself is undefined behavior.

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

B::B(V* v, A* a) {
  f();      // calls V::f -- fine, resolves within B's own class
  a->f();   // undefined behavior: a's static type is not a base of B
}
```

Wording

Notes regarding 11.9.5[class.cdtor], paragraph 5:

5 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*]

UB Annex Entry

F.6.8 **[ub:class.cdtor.virtual.not.x]**

Specified in: 11.9.5[ubx:class.cdtor.virtual.not.x]

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, 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.

2 [*Example 1*:

```
struct V {
  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 defined, calls B::g
  a->f();       // undefined behavior, a's type not a base of B
```

```
}
```

— end example]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**`noexcept`**-*enforce*	**`noexcept`**-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**`noexcept`**-*enforce*	**`noexcept`**-*observe*	
`class.cdtor.virtual.not.x`		✔								✔							CE

[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

Description

If the operand of `typeid` refers to the object under construction or destruction, and its static type is neither the constructor/destructor's own class nor one of its bases, the behavior is undefined.

```
struct V { 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(*v);   // defined: v's type V is a base of B
  typeid(*a);   // undefined behavior: a's type A is not a base of B
}
```

Wording

Notes regarding 11.9.5[class.cdtor], paragraph 6:

6 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]).

UB Annex Entry

F.6.9 **[ub:class.cdtor.typeid]**

Specified in: 11.9.5[ubx:class.cdtor.typeid]

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.

2 [*Example 1*:

```
struct V {
```

```
  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);        // std::type_info for B
  typeid(*v);           // defined, *v has type V, a base of B yields std::type_info for B
  typeid(*a);           // undefined behavior, type A not a base of B
}
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**noexcept**-*enforce*	**noexcept**-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**noexcept**-*enforce*	**noexcept**-*observe*	
`class.cdtor.typeid`		✔								✔							CE

[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

Description

If the operand of a `dynamic_cast` refers to the object under construction or destruction, and its static type is not the constructor/destructor's own class (or a pointer/object of one of its bases), the cast is undefined behavior.

```
struct V { 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) {
  dynamic_cast<B *>(v);   // defined: v's type V is a base of B
  dynamic_cast<B *>(a);   // undefined behavior: a's type A is not a
                          // base of B
}
```

Wording

Notes regarding 11.9.5[class.cdtor], paragraph 7:

7 `dynamic_cast`s (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*]

UB Annex Entry

F.6.10 **[ub:class.cdtor.dynamic.cast]**

Specified in: 11.9.5[ubx:class.cdtor.dynamic.cast]

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.

2 [*Example 1*:

```
struct V {
  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) {
  dynamic_cast<B *>(v);         // 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*]

Implementation Status (full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	
`class.cdtor.dynamic.cast`		✔								✔							CE

[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:	Clang emits a compile-time warning for direct references in the handler, GCC does not, and there is no run-time or constant-evaluation detection.

Description

A function-try-block on a constructor or destructor catches exceptions thrown while initializing (or destroying) the object's bases and members – but by the time the handler runs, those bases and members are gone (or not yet built). Referring to any non-static member or base class of the object from within that handler is undefined behavior.

```
struct A {
  A(int v) try : x(v == 1 ? throw 1 : 1) {
  } catch (int) {
    int r = y;   // undefined behavior: refers to non-static member y of an
                 // object whose initialization did not complete
    (void)r;
    throw;
  }
  int x;
  int y = 42;
};
```

Wording

Notes regarding 14.4[except.handle], paragraph 12:

12 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]).

UB Annex Entry

F.8.1 **[ub:except.handle.handler.ctor.dtor]**

Specified in: 14.4[ubx:except.handle.handler.ctor.dtor]

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.

2 [*Example 1*:

```
#include <iostream>
```

```
struct A {
  A() try : x(0 ? 1 : throw 1) {
  } catch (int) {
    std::cout << "y: " << y << std::endl;   // undefined behavior, referring to non-static member y in
                                            // the handler of function-try-block
  }
  int x;
  int y = 42;
};
```

— *end example*]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	***noexcept*-enforce**	***noexcept*-observe**	
except.handle.handler.ctor.dtor		✔								✔							CE

Implementation Experience

Existing Practice

Clang detects this at compile time: it warns that one cannot refer to a non-static member from the handler of a constructor *function-try-block* (`-Wexceptions`), pointing at the offending member reference. GCC issues no such warning.

GCC currently supports `constexpr` exception handling, but does not identify this behavior as undefined behavior.

2.4 Excluded

The wording of undefined behaviors in this section is likely to be removed or changed to no longer classify the situation as undefined behavior.

[temp.inst.inf.recursion]

summary: infinite recursive template instantiation with no base case.

Description

Implicitly instantiating a template can require instantiating other templates in turn. If that process recurses infinitely – each instantiation triggering another, with no base case – the result is undefined.

This undefined behavior does not actually make the behavior of any evaluation become undefined, but simply states that compiling code with such an infinite recursive template has undefined behavior. Specification of this sort is what the language has "*ill-formed, no diagnostic required*" for, and this change is considered in [CWG3034]. Therefore we exclude this case from further consideration in relation to the behavior of evaluations.

An example of code that would exhibit this problem is:

```
template <class T>
class X {
  X<T> *p;   // OK: a pointer doesn't require completing X<T>
  X<T*> a;   // instantiating X<T> requires instantiating X of
             // pointer-to-T, which requires X of pointer-to-pointer-to-T,
             // and so on forever
};

int main() {
  X<int> x;   // undefined behavior: instantiation recurses infinitely
}
```

Existing practice is that the above code is just ill-formed on most compilers — either with an explicit warning about the recursive definition or a compiler invocation that never completes.

Wording

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*]

UB Annex Entry

F.7.1 [ub:temp.inst.inf.recursion]

Specified in: 13.9.2[ubx:temp.inst.inf.recursion]

The result of an infinite recursion in template instantiation is undefined.

2 [*Example 1*:

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

int main() {
  X<int> x;     // undefined behavior, instantiation will lead to infinite recursion
}
```

— *end example*]

3 Implementation Status

Many of the checks named in [P3100R8] have been implemented in prototype branches of Clang and GCC, along with many of the other proposals that have been called out in the plan laid out in [P3850R1]. For each category of undefined behavior in the standard, a number of implementation possibilities exist:

- If a check can be emitted by compiler frontends that captures the entire category then the compilers can readily support all evaluation semantics, with whatever configuration granularity the compiler supports for other contract assertions.
- If a check can be readily emitted after exception-handling and language-specific knowledge has been discarded by the compiler then it is possible to support the non-throwing evaluation semantics — *assume*, *quick-enforce*, *`noexcept`-enforce*, and *`noexcept`-observe*.
- If a supported sanitizer guards the named category of undefined behavior then some subset of semantics are choosable by enabling that sanitizer with the given semantic (using `-fsanitize-semantic=<sanitizer>:<semantic>`).
- For operations that have no fallback behavior the ignore and assume semantics are interchangeable. If a fallback behavior has been specified in [P3100R8] then the ignore semantic is separate and might or might not be supported.
- If a subset of that category meets any of the above criteria, we have named a new category with a further qualifier (such as `ub:expr.add.out.of.bounds.known` which is a check for a subset of the conditions in the `ub:expr.add.out.of.bounds` category, or `ub:dcl.attr.assume.false.pure` which is the side-effect-free subset of `ub:dcl.attr.assume.false`).

In the table below we list all categories that have implementation support in the prototype implementations. In the semantic columns there is a green check (✔) for those check+semantic columns that have full support (and which can be controlled through the `.json` configuration file as described in [P3595R0]). Those semantics supported by enabling a given sanitizer have the name of the sanitizer. (Some cells have both, because the check can be enabled by a sanitizer and might also be enabled by setting a semantic for that implicit assertion group).

Each compiler's block of columns leads with a *constexpr* column, separate from the seven evaluation-semantic columns: a green check there means we have confirmed that compiler's constant evaluator rejects the undefined behavior when the same operation is forced into a `static_assert` – independent of `-fcontracts-p3100`, `-fsanitize=`, or any of the evaluation semantics, since constant evaluation always has exact knowledge of the real object model regardless of what a runtime optimizer could prove. A blank *constexpr* cell means either that this hasn't been checked yet, or that no constexpr-compatible trigger exists for the category at all – the two aren't distinguished in the table itself; see the category's own Implementation Status or Implementation Experience prose for which applies.

Every row shows a green check under *assume* for both compilers: *assume* is the status-quo baseline for every category (today's behavior, unchanged), so it is trivially available even where no other semantic has been wired yet.

Rows are grouped and ordered exactly as in Appendix A of [P3100R8]; an indented row such as ↪ `known` names a qualified sub-group of the (unindented) group directly above it.

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	
I. Initialization																	
`basic.indet.value`		✔								✔					MSan MSan	MSan MSan	
↪ `...attr.indeterminate`	✔	✔							✔	✔					MSan MSan	MSan MSan	CE
↪ `...object.padding`		✔								✔					MSan MSan	MSan MSan	CE
II. Bounds																	
`basic.stc.alloc.zero.dereference`		✔					ASan† ASan	ASan		✔					ASan† ASan	ASan	CE
`expr.delete.mismatch`	✔	✔					ASan ASan	ASan ASan	✔	✔					ASan ASan	ASan ASan	CE
`expr.delete.array.mismatch`	✔	✔					ASan ASan	ASan ASan	✔	✔					ASan ASan	ASan ASan	CE
`expr.add.out.of.bounds`		✔								✔							
↪ `...known`	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE
↪ `...unknown`	✔	✔					ASan† ASan†	ASan† ASan†	✔	✔					ASan† ASan†	ASan† ASan†	CE
↪ `...pointer.arithmetic`		✔		UBSan			UBSan UBSan	UBSan UBSan		✔		UBSan			UBSan UBSan	UBSan UBSan	CE
`expr.add.sub.diff.pointers`	✔	✔					ASan ASan	ASan	✔	✔					ASan ASan	ASan	CE
`expr.sub.pointers.representable`		✔								✔							CE
III. Type and Lifetime																	
`intro.object.implicit.create`		✔								✔							CE
`intro.object.implicit.pointer`		✔								✔							
↪ `...deleted.dtor`		✔								✔							CE
↪ `...type.pun`		✔								✔							CE
`basic.align.object.alignment`		✔	P3850	P3850 UBSan			P3850 UBSan UBSan	P3850 UBSan UBSan		✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE
`lifetime.outside.pointer.delete`		✔							✔	✔							CE
`lifetime.outside.pointer.member`	✔	✔							✔	✔							CE
`lifetime.outside.pointer.virtual`	✔	✔					UBSan† UBSan†	UBSan†	✔	✔					UBSan† UBSan†	UBSan†	CE
`lifetime.outside.pointer.dynamic.cast`		✔							✔	✔					UBSan† UBSan†	UBSan†	CE
`lifetime.outside.glvalue.access`	✔	✔					ASan† ASan†	ASan† ASan†	✔	✔					ASan† ASan†	ASan† ASan†	CE

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**noexcept**-*enforce*	**noexcept**-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	**noexcept**-*enforce*	**noexcept**-*observe*	
`lifetime.outside.glvalue.member`	✔	✔							✔	✔							CE
`lifetime.outside.glvalue.virtual`	✔	✔					UBSan† UBSan†	UBSan†	✔	✔					UBSan† UBSan†	UBSan†	CE
`lifetime.outside.glvalue.dynamic.cast`	✔	✔							✔	✔					UBSan† UBSan†	UBSan†	CE
`original.type.implicit.destructor`		✔								✔							CE
`creating.within.const.complete.obj`		✔							✔	✔							
↪ `...live`		✔							✔	✔							CE
↪ `...reuse`		✔							✔	✔							CE
`basic.compound.pointer.before.storage.duration`		✔								✔							CE
`basic.compound.invalid.pointer`	✔	✔					ASan† ASan†	ASan† ASan†	✔	✔					ASan† ASan†	ASan† ASan†	CE
`expr.basic.lvalue.strict.aliasing.violation`		✔								✔							CE
`expr.basic.lvalue.union.initialization`	✔	✔							✔	✔							CE
`expr.type.reference.lifetime`		✔								✔							CE
`conv.lval.valid.representation`		✔	P3850	P3850 UBSan			P3850 UBSan UBSan	P3850 UBSan UBSan		✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	
↪ `...bool`		✔	P3850	P3850 UBSan			P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE
↪ `...enum`		✔	P3850	P3850 UBSan			P3850 UBSan UBSan	P3850 UBSan UBSan		✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE
↪ `...int`																	N/A
↪ `...float`																	N/A
`conv.ptr.virtual.base`		✔					UBSan† UBSan†	UBSan†		✔					UBSan† UBSan†	UBSan†	CE
`conv.member.missing.member`	✔	✔					UBSan	UBSan† UBSan	✔	✔							CE
`expr.call.different.type`	✔	✔							✔	✔		UBSan			UBSan UBSan	UBSan UBSan	CE
`expr.ref.member.not.similar`	✔	✔							✔	✔							CE
`expr.dynamic.cast.pointer.lifetime`		✔								✔							CE
`expr.dynamic.cast.glvalue.lifetime`		✔							✔	✔							CE

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	
`expr.static.cast.base.class`		✔							✔	✔							
↪ `...polymorphic`		✔							✔	✔					UBSan UBSan	UBSan	CE
↪ `...nonpolymorphic`		✔							✔	✔							CE
`expr.static.cast.downcast.wrong.derived.type`		✔							✔	✔							
↪ `...polymorphic`		✔					UBSan UBSan	UBSan	✔	✔					UBSan UBSan	UBSan	CE
↪ `...nonpolymorphic`		✔							✔	✔							CE
`expr.static.cast.does.not.contain.original.member`		✔							✔	✔							CE
`expr.reinterpret.cast.invalid.pointer.value`		✔								✔							CE
`expr.unary.dereference`	✔	✔							✔	✔							
↪ `...nullptr`	✔	✔	P3850	P3850 UBSan			P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE
↪ `...invalid`	✔	✔					ASan† ASan†	ASan† ASan†	✔	✔					ASan† ASan†	ASan† ASan†	CE
`expr.delete.dynamic.type.differ`		✔					ASan ASan	ASan	✔	✔					ASan ASan	ASan	CE
`expr.delete.dynamic.array.dynamic.type.differ`	✔	✔					ASan ASan	ASan	✔	✔					ASan ASan	ASan	CE
`expr.mptr.oper.not.contain.member`	✔	✔					ASan† ASan	ASan	✔	✔					ASan† ASan	ASan	CE
`expr.mptr.oper.member.func.null`	✔	✔							✔	✔							CE
`expr.add.not.similar`	✔	✔							✔	✔							CE
`expr.assign.overlap`		✔								✔							CE
`dcl.type.cv.modify.const.obj`	✔	✔							✔	✔							CE
`dcl.type.cv.access.volatile`		✔							✔	✔							CE
`dcl.ref.incompatible.function`		✔								✔		UBSan			UBSan UBSan	UBSan UBSan	CE
`dcl.ref.incompatible.type`		✔								✔							CE
`dcl.ref.uninitialized.reference`		✔								✔							CE
`class.dtor.no.longer.exists`		✔								✔							
↪ `...polymorphic`		✔					UBSan† UBSan†			✔							CE
↪ `...nonpolymorphic`	✔	✔							✔	✔							CE
`class.abstract.pure.virtual`	✔	✔		P3850	P3850	P3850	P3850	P3850	✔	✔		P3850	P3850	P3850	P3850	P3850	CE

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	
`class.base.init.mem.fun`		✔								✔							
↪ `...polymorphic`		✔					UBSan† UBSan†	UBSan†		✔							CE
↪ `...nonpolymorphic`	✔	✔							✔	✔							CE
`class.cdtor.before.ctor`		✔								✔							
↪ `...polymorphic`		✔					UBSan† UBSan†	UBSan†		✔					UBSan† UBSan†	UBSan†	CE
↪ `...nonpolymorphic`		✔							✔	✔							CE
`class.cdtor.after.dtor`		✔								✔							
↪ `...polymorphic`		✔					UBSan† UBSan†			✔							CE
↪ `...nonpolymorphic`	✔	✔							✔	✔							CE
`class.cdtor.convert.pointer`		✔								✔							CE
`class.cdtor.form.pointer`		✔								✔							
↪ `...polymorphic`		✔								✔					UBSan† UBSan†	UBSan†	CE
↪ `...nonpolymorphic`		✔								✔							CE
`class.cdtor.virtual.not.x`		✔								✔							CE
`class.cdtor.typeid`		✔								✔							CE
`class.cdtor.dynamic.cast`		✔								✔							CE
`except.handle.handler.ctor.dtor`		✔								✔							CE
IV. Arithmetic																	
`expr.expr.eval`	✔	✔	P3850	P3850 UBSan			P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	
↪ `...signed.integer.add`	✔	✔	P3850	P3850 UBSan			P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE
↪ `...signed.integer.sub`	✔	✔	P3850	P3850 UBSan			P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE
↪ `...signed.integer.mul`	✔	✔	P3850	P3850 UBSan			P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE
↪ `...signed.integer.negate`	✔	✔	P3850	P3850 UBSan			P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE
`conv.double.out.of.range`																	N/A
`conv.fpint.float.not.represented`	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE
`conv.fpint.int.not.represented`																	N/A
`expr.static.cast.enum.outside.range`		✔	P3850	P3850	P3850	P3850	P3850	P3850	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	
`expr.static.cast.fp.outside.range`																	N/A
`expr.mul.div.by.zero`	✔	✔							✔	✔							
↪ ... `int.div`	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE
↪ ... `int.mod`	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE
↪ ... `fp.div`	✔	✔		UBSan			UBSan UBSan	UBSan UBSan	✔	✔		UBSan			UBSan UBSan	UBSan UBSan	CE
`expr.mul.representable.type.result`	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	
↪ ... `int.div`	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE
↪ ... `int.mod`	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE
`expr.shift.neg.and.width`	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	
↪ ... `shl`	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE
↪ ... `shr`	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan UBSan	CE
V. Threading																	
`intro.races.data`		✔					TSan† TSan	TSan		✔					TSan† TSan	TSan† TSan	CE
`intro.progress.stops`		✔								✔							
VI. Sequencing																	
`intro.execution.unsequenced.modification`		✔								✔							CE
VII. Assumptions																	
`dcl.attr.assume.false`		✔	P3850							✔	P3850						
↪ ... `pure`	✔	✔	P3850	P3850	P3850	P3850	P3850	P3850	✔	✔	P3850	P3850	P3850	P3850	P3850	P3850	CE
↪ ... `nonpure`		✔	P3850							✔	P3850						CE
VIII. Control Flow																	
`basic.start.main.exit.during.destruction`		✔								✔							CE
`basic.start.term.use.after.destruction`		✔								✔							CE
`stmt.return.flow.off`	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan	✔	✔	P3850	P3850 UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan	CE

Group	GCC								Clang								compiler explorer
	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	constexpr	*assume*	*ignore*	*quick-enforce*	*enforce*	*observe*	`noexcept`-*enforce*	`noexcept`-*observe*	
`stmt.dcl.local.static.init.recursive`		✔								✔							CE
`dcl.attr.noreturn.eventually.returns`		✔		UBSan			UBSan UBSan	UBSan		✔		UBSan			UBSan UBSan	UBSan	CE
IX. Replacement Functions																	
`basic.stc.alloc.dealloc.constraint`		✔								✔					UBSan UBSan	UBSan	CE
`expr.new.non.allocating.null`	✔	✔		UBSan			UBSan UBSan	UBSan UBSan	✔	✔		UBSan			UBSan UBSan	UBSan UBSan	CE
X. Coroutines																	
`stmt.return.coroutine.flow.off`		✔	P3850	P3850	P3850	P3850	P3850	P3850		✔	P3850	P3850	P3850	P3850	P3850	P3850	CE
`dcl.fct.def.coroutine.resume.not.suspended`		✔								✔							CE
`dcl.fct.def.coroutine.destroy.not.suspended`		✔					ASan† ASan†	ASan†		✔					ASan† ASan†	ASan†	CE

3.1 Charts

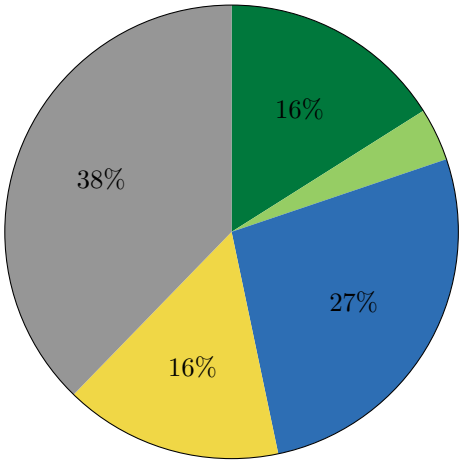
Now, of course, nothing is more fun than charts. It's important to note that the data underlying these charts is, inherently, not weighted in a manner that makes it particularly applicable to reality. Real exploitable problems in production systems are far, far more likely to be the result of integer overflow or invalid pointer dereference, but those are only two of the many places in the C++ Standard where distinct forms of undefined behavior are introduced.

But, looking at how progress has been made on identifying and mitigating each of the categories we have explored does at least give us an idea of where the progress is in the state of the art. To that end, we're going to take the above data and visualize it, showing how well the categories of undefined behavior can be managed, and how well those categories can be managed in a fashion that is fully integrated with the contract-violation handler by [P3100R8].

In the below charts, all subcategories are weighted equally and both open-source compilers are weighted equally.

Runtime-Checkable UB Categories

GCC and Clang counted separately (106 categories × 2 compilers)



Prototype fully integrated with contracts	34 (16%)
Contracts integration, no exception support	8 (4%)
Integrated sanitizer	57 (27%)
Legacy sanitizer	0 (0%)
Identified only during constant evaluation	33 (16%)
No checks	80 (38%)

4 Foundational Wording Changes

The primary proposal of [P3100R8] is that all undefined behavior has an implicit contract assertion introduced prior to actual behavior with no restrictions at all occurring. That implicit contract assertion may be ignored or assumed, but in either case until the implicit contract assertion is evaluated with some semantic, program behavior can still be reasoned about.

4.1 Implicit Assertions as Part of Undefined Behavior

The currently proposed wording strategy for introducing an implicit contract assertion as a part of all undefined behavior is to make it part of the definition of undefined behavior. The wording changes needed for this approach are presented in this section.

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

3.65 + *a* **unconstrained behavior** **[defns.unconstrained]**

Add a new definition 3.65+a[defns.unconstrained] after 3.65[defns.unblock]

> 3.65 + *a* **[defns.unconstrained]**
>
> **unconstrained behavior**
> behavior for which this document imposes no requirements
>
> [*Note to entry 1*: Unconstrained behavior is expected whenever a contract assertion with the assume semantic (6.11.2[basic.contract.eval]) is violated or control continues after the guarding contract assertion (6.11.2+a[basic.contract.implicit]) of undefined behavior. — *end note*]
>
> [*Note to entry 2*: Permissible unconstrained 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). — *end note*]

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

Modify definition 3.66[defns.undefined]:

> 3.66 **[defns.undefined]**
>
> **undefined behavior**
> behavior ~~for which this document~~that begins with an implicit contract assertion ~~imposes no requirements~~that the behavior cannot occur (6.11.2+a[basic.contract.implicit]) and, if control continues after the contract assertion, has unconstrained behavior.
>
> [*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. There is no requirement that any particular semantic choice be available for the implicit contract assertion. — *end note*]
>
> [*Note to entry 1+a*: 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*]

4 **General principles** **[intro]**

4.1 **Implementation compliance** [intro.compliance]

4.1.1 **General** [intro.compliance.general]

Modify section 4.1.1[intro.compliance.general], paragraphs 1-2:

1 The set of *diagnosable rules* consists of all syntactic and semantic rules in this document except for those rules containing an explicit notation that "no diagnostic is required" or which are described as resulting in "~~undefined~~unconstrained behavior".

Note

Many behaviors are described as resulting in undefined behavior, and when control continues past the guarding contract assertion of that behavior they then result in undefined behavior.

2 Although this document states only requirements on C++ implementations, those requirements are often easier to understand if they are phrased as requirements on programs, parts of programs, or execution of programs. Such requirements have the following meaning:

(2.1) — If a program contains no violations of the rules in Clause 5[lex] through Clause 33[exec] as well as those specified in Annex D[depr], a conforming implementation shall accept and correctly execute[3] that program, except when the implementation's limitations (see below) are exceeded.

(2.2) — If a program contains a violation of a rule for which no diagnostic is required, this document places no requirement on implementations with respect to that program.

(2.3) — Otherwise, if a program contains

- a violation of any diagnosable rule,
- a preprocessing translation unit with a `#warning` preprocessing directive (15.9[cpp.error]),
- an occurrence of a construct described in this document as "conditionally-supported" when the implementation does not support that construct, or
- a contract assertion (6.11.2[basic.contract.eval]) evaluated with a checking semantic in a manifestly constant-evaluated context (7.7.7[expr.const.defns]) resulting in a contract violation,

a conforming implementation shall issue at least one diagnostic message.

[*Note 1*: During template argument deduction and substitution, certain constructs that in other contexts require a diagnostic are treated differently; see 13.10.3[temp.deduct]. — *end note*]

Furthermore, a conforming implementation shall not accept

(2.1) — a preprocessing translation unit containing a `#error` preprocessing directive (15.9[cpp.error]),

(2.2) — a translation unit with a *static_assert-declaration* that fails (9.1[dcl.pre]), or

(2.3) — a contract assertion evaluated with a terminating semantic (6.11.2[basic.contract.eval]) in a manifestly constant-evaluated context (7.7.7[expr.const.defns]) resulting in a contract violation.

---

[3] "Correct execution" can include undefined behavior, unconstrained behavior, and erroneous behavior, depending on the data being processed; see Clause 3[intro.defs] and 6.10.1[intro.execution].

4.1.2 **Abstract machine** **[intro.abstract]**

Add a paragraph to and modify section 4.1.2[intro.abstract], paragraphs 4-6:

3+a Certain operations are described in this document as undefined behavior (for example, the effect of attempting to modify a const object). Such operations evaluate a *guarding contract assertion*, which is an implicit contract assertion that they do not occur (6.11.2+a[basic.contract.implicit]). If control continues normally after a guarding contract assertion, the behavior is unconstrained.

4 Certain ~~other~~ operations are described in this document as ~~undefined~~unconstrained behavior (for example, control continuing after evaluating the ~~effect of~~guarding contract assertion ~~attempting to modify a const object~~associated with an instance of undefined behavior (6.11.2+a[basic.contract.implicit])).

5 Certain events in the execution of a program are termed *observable checkpoints*.

[*Note 1*: A call to `std::observable_checkpoint` (22.2.9[utility.undefined]) is an observable checkpoint, as are certain parts of the evaluation of contract assertions (6.11[basic.contract]). — *end note*]

[*Note 1+a*: Undefined behavior whose guarding contract assertion is evaluated with the ignore or assume semantic does not imply an observable checkpoint. — *end note*]

6 The *defined prefix* of an execution comprises the operations O for which for every ~~undefined~~unconstrained operation U there is an observable checkpoint C such that O happens before C and C happens before U.

[*Note 2*: The undefined behavior that arises from a data race (6.10.2.2[intro.races]) occurs on all participating threads. — *end note*]

A conforming implementation executing a well-formed program shall produce the observable behavior of the defined prefix of one of the possible executions of the corresponding instance of the abstract machine with the same program and the same input. If the selected execution contains an ~~undefined~~unconstrained operation, the implementation executing that program with that input may produce arbitrary additional observable behavior afterwards. If the execution of an operation is specified as having erroneous behavior, the implementation is permitted to issue a diagnostic and is permitted to terminate the execution of the program.

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 evaluated with the assume semantic would not evaluate to `true`, or the condition of such a contract assertion is not satisfied, the evaluation has unconstrained behavior (4.1.2[intro.abstract]); otherwise the evaluation has no effect.

4+b [*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*]

5 The evaluation A of a contract assertion with a predicate using a checking semantic determines the value of the predicate. 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]) C that happens before A such that any other evaluation that happens before A also happens before C.

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

2 All undefined behavior has a *guarding contract assertion* that is an implicit contract assertion whose condition is never satisfied.

3 [*Note 1*: If the guarding contract assertion of undefined behavior is

(3.1) — evaluated with the ignore semantic,

(3.2) — evaluated with the assume semantic, or

(3.3) — evaluated with the observe semantic and the contract-violation handler returns normally

then the behavior is unconstrained (4.1.2[intro.abstract]). — *end note*]

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

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*]

4.2 Implicit Assertions Before All Undefined Behavior

This alternative avoids separating out the parts of undefined behavior as part of the definition and instead declares that a guarding contract assertion is evaluated immediately prior to any undefined behavior occurring.

These wording changes are relative to the C++29 draft sources with git hash cfa21fec, last modified

on Thu, 13 Aug 2026 13:55:07 -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 occurence of undefined behavior. — *end note*]

6 **Basics** **[basic]**

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*]

5 The evaluation A of a contract assertion with a predicate using a checking semantic determines the value of the predicate. 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]) C that happens before A such that any other evaluation that happens before A also happens before C.

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 An evaluation O may have an *implicit precondition assertion* C applied to it. If so, the evaluation in sequence of C is sequenced before the evaluation of O and after the evaluation of all operands of O.

2 Any evaluation with undefined behavior has a *guarding contract assertion* applied to it that is an implicit precondition assertion whose condition 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 specifies behavior. — *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.

14 **Exception handling** **[except]**

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 ([basic.contract.general]) causes a call to the contract-violation handler ([basic.contract.handler]) and that handler exits via an exception. — *end note*]

4.3 Common Changes

Changes to the library are independent of the wording mechanism chosen to introduce guarding contract assertions whenever undefined behavior occurs.

17 **Language support library** [**support**]

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
  };

  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], 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

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 (6.11.2+a[basic.contract.implicit])

Undefined behavior that is implicit is not covered by this annex. 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.

5 Conclusion

EWG asked for a line-item review of [P3100R8]. There have also been significant questions about the implementatability and meaning of the changes proposed by [P3100R8].

This paper has presented, as thoroughly as we can manage, the information needed to understand those details and be ready to make a final decision on adopting [P3100R8].

Acknowledgments

Thank you to everyone in EWG for giving time to consideration of this information.

Claude (Anthropic) was used for editorial assistance during the preparation of this paper, as well as significant parts of the prototype implementations.

Bibliography

[CWG3034] Timur Doumler, "Infinite recursion should hit an implementation limit"
https://wg21.link/cwg3034

[P3100R7] Timur Doumler and Joshua Berne, "A framework for systematically addressing undefined behaviour in the C++ Standard", 2026
http://wg21.link/P3100R7

[P3100R8] Timur Doumler and Joshua Berne, "A framework for systematically addressing undefined behaviour in the C++ Standard", 2026
http://wg21.link/P3100R8

[P3400R4] Joshua Berne, "Controlling Contract-Assertion Properties", 2026
http://wg21.link/P3400R4

[P3595R0] Joshua Berne and Iain Sandoe, "Configuration of Contract Evaluation Semantics", 2026
http://wg21.link/P3595R0

[P3596R3] Joshua Berne, Timur Doumler, Jens Maurer, and Shafik Yaghmour, "Undefined Behavior and IFNDR Annexes", 2026
http://wg21.link/P3596R3

[P3850R1] Timur Doumler and Joshua Berne, "A proposed plan for extending Contracts in C++29", 2026
http://wg21.link/P3850R1

[P4284R0] Joshua Berne, "Addenda to the Undefined Behavior and IFNDR Annexes", 2026
http://wg21.link/P4284R0

[P4298R0] Joshua Berne, "Nonthrowing Evaluation Semantics", 2026
http://wg21.link/P4298R0

[P4301R0] Joshua Berne, “Context Reports for the Contract-Violation Handler”, 2026
http://wg21.link/P4301R0