

Overview and Implementation Report for P3100

Document #: P4277R1
Date: 2026-10-23
Project: Programming Language C++
Audience: SG21 (Contracts)
Reply-to: Joshua Berne <jberne4@bloomberg.net>

Abstract

EWG wishes to review each individual wording change in [P3100R9]; 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	50
2.4	Excluded: Compile-Time Template Recursion	231
3	Implementation Status	234
3.1	Implementation Coverage Charts	243
4	Foundational Wording Changes	244
4.1	Implicit Assertions Before All Undefined Behavior	245
4.2	Implicit Assertions as Part of Undefined Behavior	249
4.3	Wording Changes Common to Both Strategies	256
5	Conclusion	263

Revision History

Revision 1: October 2026 mailing

- Corrected the semantics listed as available for sanitizers and prototypes.
- Significant additional implementation experience recorded.
- Rebased on [P3100R9], which in turn is based on [P4284R1].
- Previous alternate wording strategy is now proposed by [P3100R9] and the original strategy is presented here as an alternate.

Revision 0: August 2026 mailing¹

- Ongoing draft for discussion in telecons.

1 Introduction

[P3100R9] (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 [P3100R9], 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 [P3100R9] contains.

This paper is **NOT** a proposal of its own, and is provided as an illuminating companion to [P3100R9]. Any discrepancies between this paper and [P3100R9] or more recent revisions of that paper are unintentional and should be interpreted in favor of the other paper. Any polls asking for changes to the contents of these papers, or to forward the proposal, should target [P3100R9]. Consider this paper and its contents to be a rather elaborate set of slides supporting the content of [P3100R9].²

¹An early first draft of this paper was shown at the 2026-07-27 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 of introducing guarding contract assertions before undefined behavior, which [P3100R9] has since adopted (Section 4). This revision for the mailing has been edited for consistency and clarity and captures further implementation experience that has occurred since the telecon.

²If you are seeing a D version of this paper, it is quite likely to not have exactly the same wording as a version

The proposal in [P3100R9] 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`.
- 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 undefined behavior.
- [P3100R9] 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.

of [P3100R9] 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.

2 Individual Undefined Behavior Treatments

On the following pages we will go through all of the wording changes in [P3100R9], 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 [P3100R9] — 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; the first (Section 4) is the one [P3100R9] proposes.
- In the subsections of this section, we include information about each individual instance of undefined behavior, grouped generally by how [P3100R9] 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 [P3100R9], 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 determining 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. Note that replacement behavior might be choosable 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.

2 INDIVIDUAL UNDEFINED BEHAVIOR TREATMENTS

- The **UB Annex Entry** that documents the undefined behavior, as gathered and merged into the standard originally by [P3596R3] and updated in [P4284R1].
- An **Implementation Experience** section describing what was involved with prototype implementations 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; when a sanitizer identifies undefined behavior and emits its own diagnostics (or calls its own failure callback), it is effectively evaluating the assertion with a checked semantic. Sanitizer-specific callbacks are, effectively, local violation handlers (as described in [P3400R5]) 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. The checks integrated so far are AddressSanitizer and the UndefinedBehaviorSanitizer `vptr` check; each category’s Implementation Experience section says which checks cover it.

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.

2 INDIVIDUAL UNDEFINED BEHAVIOR TREATMENTS

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 [GCC](#) and [Clang](#): `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; the [GCC](#) and [Clang](#) examples show the check delivered to the handler as an implicit assertion (the routed sanitizer emits nothing itself; the handler prints its `report()`).

One limitation of the integrated path remains. Some sanitizer checks are inserted by the compiler only after the optimizer has run: `AddressSanitizer` and `ThreadSanitizer` on both compilers, and also `MemorySanitizer` and the `local-bounds` check on `Clang`. When such a check is evaluated with the *noexcept-observe* semantic and the handler returns, the code following the check is not guaranteed to see the handler’s side effects at `-O1` and above (a variable that only the handler modifies may still be read with its old value). Checks inserted earlier, including every native implicit assertion, are not affected.

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

³We have prepared a small paper proposing this functionality in [\[P4301R0\]](#).

2.1 Fallback to Erroneous Values

When an operation that might have undefined behavior produces a scalar value, we can instead produce an unspecified erroneous value as the fallback. Every item in this section does so (for [ub:basic.indet.value] and [ub:stmt.return.flow.off], for built-in types only).

[basic.indet.value]

summary: reading an uninitialized (indeterminate) value.

runtime-checkable: Yes

locally-checkable: No

checking: Track whether storage has been initialized

replacement: Only for built-in types: initialize default-initialized variables with erroneous value

implementations: Heuristically caught by MemorySanitizer on Clang (when the value is used); also rejected during constant evaluation on both compilers.

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: reads an indeterminate 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]]` variable or a scalar created by `new` without an initializer.
- `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 (and copying the object's bytes, for example with `std::bit_cast`, carries that indeterminacy along).

A default-initialized automatic variable without `[[indeterminate]]` is not in this category: since C++26 reading it is erroneous behavior, not undefined.

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 indeterminate if any operand is indeterminate and erroneous otherwise. Except in the following cases, if all evaluations

have an implicit precondition assertion that they will not produce an indeterminate value ~~is produced by an evaluation, the behavior is undefined~~ (6.11.2+a[`basic.contract.implicit`]) (F.2.14[`ubx:basic.indet.value`]), and if they do then the result of the evaluation is an indeterminate value; if an erroneous value is produced by an evaluation, the behavior is erroneous and the result of the evaluation is that erroneous value:

- (2.1) — If an indeterminate or erroneous value of unsigned ordinary character type (6.9.3[`basic.fundamental`]) or `std::byte` type (17.2.1[`cstddef.syn`]) is produced by the evaluation of:
- the second or third operand of a conditional expression (7.6.16[`expr.cond`]),
 - the right operand of a comma expression (7.6.20[`expr.comma`]),
 - the operand of a cast or conversion (`conv.integral`, `expr.type.conv`, `expr.static.cast`, `expr.cast`) to an unsigned ordinary character type or `std::byte` type (17.2.1[`cstddef.syn`]), or
 - a discarded-value expression (7.2.3[`expr.context`]),
- then the result of the operation is an indeterminate value or that erroneous value, respectively.
- (2.2) — If an indeterminate or erroneous value of unsigned ordinary character type or `std::byte` type is produced by the evaluation of the right operand of a simple assignment operator (7.6.19[`expr.assign`]) whose first operand is an lvalue of unsigned ordinary character type or `std::byte` type, an indeterminate value or that erroneous value, respectively, replaces the value of the object referred to by the left operand.
- (2.3) — If an indeterminate or erroneous value of unsigned ordinary character type is produced by the evaluation of the initialization expression when initializing an object of unsigned ordinary character type, that object is initialized to an indeterminate value or that erroneous value, respectively.
- (2.4) — If an indeterminate or erroneous value of unsigned ordinary character type or `std::byte` type is produced by the evaluation of the initialization expression when initializing an object of `std::byte` type, that object is initialized to an indeterminate value or that erroneous value, respectively.

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

[Example 1:

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

int g(bool b) {
    unsigned char c;
```

```

unsigned char d = c;           // no erroneous behavior, but d has an erroneous value

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

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

void h() {
    int d1, d2;

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

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

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

```

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 there is undefined~~ an implicit precondition assertion that this does not occur.

2

[Example 1:

```

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

```

— 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 [†]	
↪ ...attr.indeterminate	✓	✓							✓	✓		MSan [†]			MSan [†]	MSan [†]	CE
↪ ...object.padding	✓	✓							✓	✓		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. It reports when an uninitialized value is *used*, not when it is copied, so a read whose result is never used is missed; the detection is therefore heuristic. MSan is Clang/LLVM-only – GCC currently has no MemorySanitizer implementation at all.

Clang does not implement `[[indeterminate]]` (it warns that the attribute is unknown and ignores it); there the variable in the example is simply uninitialized, and its read is caught the same way.

Both compilers reject reading an indeterminate value during constant evaluation, including a padding byte copied out by `std::bit_cast`.

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 Clang 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 covers arithmetic expressions whose mathematical result is neither in the range of representable values for its type nor an infinity or NaN that is among the values of the type; P3100 makes such a result an unspecified erroneous value, with an implicit precondition assertion that this does not occur. 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`)

Each subgroup includes the compound-assignment and `++/--` forms, which are defined in terms of these operators.

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 type₁

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

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

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

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

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

— end example]

— end note]

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

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~~an erroneous value is ~~undefined~~produced with an implicit precondition assertion that this does not occur.

2

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

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

— end example]

3

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

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

— end example]

Implementation Status

(full table in Section 3)

Group	GCC								Clang								compiler explorer
	constexpr	assume	ignore	quick-enforce	enforce	observe	no except-enforce	no except-observe	constexpr	assume	ignore	quick-enforce	enforce	observe	no except-enforce	no except-observe	
expr.expr.eval	✓	✓	P3850	P3850 UBSan UBSan			P3850 UBSan UBSan	P3850 UBSan	✓	✓	P3850	P3850 UBSan UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan	
↪ ...signed.integer.add	✓	✓	P3850	P3850 UBSan UBSan			P3850 UBSan UBSan	P3850 UBSan	✓	✓	P3850	P3850 UBSan UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan	CE
↪ ...signed.integer.sub	✓	✓	P3850	P3850 UBSan UBSan			P3850 UBSan UBSan	P3850 UBSan	✓	✓	P3850	P3850 UBSan UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan	CE
↪ ...signed.integer.mul	✓	✓	P3850	P3850 UBSan UBSan			P3850 UBSan UBSan	P3850 UBSan	✓	✓	P3850	P3850 UBSan UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 UBSan	CE
↪ ...signed.integer.negate	✓	✓	P3850	P3850 UBSan UBSan			P3850 UBSan UBSan	P3850 UBSan	✓	✓	P3850	P3850 UBSan UBSan	P3850	P3850	P3850 UBSan UBSan	P3850 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. Each of the four subgroups is a group of its own on both compilers (`ub:expr.expr.eval.signed.integer.add`, `.sub`, `.mul` and `.negate`; `++` is an `.add` and `--` a `.sub`), so each can be configured separately.

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

GCC: On GCC it is a middle-end check. 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/r383YP6zY>.

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/qPYEP9afz>.

[conv.lval.valid.representation]

summary: lvalue-to-rvalue conversion of an invalid bit pattern for an object of non-class type

runtime-checkable: Yes

locally-checkable: Yes

checking: Insert a check of whether the value is a valid value representation for the object's type

replacement: Coerce invalid value representations into erroneous values

implementations: The `bool` and `enum` subgroups are caught natively by the P3850 prototype; GCC supports only the non-throwing semantics, Clang supports all seven. UBSan's `bool/enum` checks (both in the `undefined` group) together cover both categories. The `int` and `float` subgroups have no invalid representation on commonly available platforms.

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 (pointer and pointer-to-member types are alike: no invalid representations here)
- `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. The exception is x87 80-bit `long double`, whose pseudo-NaN, pseudo-infinity and unnormal encodings are invalid; no tool checks those either.

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 undefinedan unspecified erroneous value, and there is an implicit precondition assertion that this does not occur (6.11.2+a[basic.contract.implicit]) (F.3.5[ubx:conv.lval.valid.representation]).
- [*Example 2:*
- ```
bool f() {
 bool b = true;
 char c = 42;
 memcpy(&b, &c, 1);
 return b; // undefinederroneous behavior if 42 is not a valid value representation for bool
}
```
- *end example*]
- (3.4) — Otherwise, the object indicated by the glvalue is read (3.1[defs.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~~an erroneous value is produced with an implicit precondition assertion that this does not occur.

2

[Example 1:

```

bool f() {
 bool b = true;
 char c = 42;
 memcpy(&b, &c, 1);
 return b; // undefined behaviorerroneous value 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<br>UBSan |         |         | P3850<br>UBSan   | P3850<br>UBSan   | —         | ✓      | P3850  | P3850<br>UBSan | P3850<br>UBSan | P3850   | P3850<br>UBSan   | P3850<br>UBSan   |                   |
| ↪ ...bool                      | —         | ✓      | P3850  | P3850<br>UBSan |         |         | P3850<br>UBSan   | P3850<br>UBSan   | —         | ✓      | P3850  | P3850<br>UBSan | P3850          | P3850   | P3850<br>UBSan   | P3850<br>UBSan   | CE                |
| ↪ ...enum                      | —         | ✓      | P3850  | P3850<br>UBSan |         |         | P3850<br>UBSan   | P3850<br>UBSan   | —         | ✓      | P3850  | P3850<br>UBSan | P3850          | P3850   | P3850<br>UBSan   | P3850<br>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.

Constant evaluation cannot reach this conversion, hence the `---` cells: the only `constexpr` way to manufacture an invalid `bool` or `enum` is `std::bit_cast`, which is itself undefined when no value corresponds (`[bit.cast]`), and `memcpy` between types is not a constant expression. Clang diagnoses that `bit_cast` for `bool`, not for `enum` ([llvm#68489](#)); GCC diagnoses neither.

## 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 a configured `enforce` or `observe` is clamped to the corresponding `noexcept` variant under `-fcontracts-p4298`, else to `quick-enforce`, so the user’s `handle_contract_violation` is invoked only with `-fcontracts-p4298`.

The `bool` and `enum` subgroups are separately configurable on both compilers, as `ub:conv.lval.valid.representation.bool` and `ub:conv.lval.valid.representation.enum`.

The check covers the lvalue-to-rvalue load of a `bool` or non-fixed `enum` whose stored value is out of range; a cast that produces such a value is checked separately at the cast, as `expr.static.cast.enum.outside.range`. It is a middle-end check.

When continuing past a violation without terminating – the *ignore* and *noexcept-observe* semantics (and, on Clang, *observe*) – a valid erroneous 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/Kxf1rWs7>.

**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 that value or values surrounding that value.

**runtime-checkable:** Yes

**locally-checkable:** Yes

**checking:** Insert a check of whether the value is one that can be converted

**replacement:** Coerce into erroneous value

**implementations:** None

**Description**

Converting a floating-point value to another floating-point type that cannot represent it 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 is well-defined, its result an implementation-defined choice between the two adjacent representable values (the largest finite value and infinity).

```
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
 // (an implementation-defined choice of FLT_MAX or +inf)
 // on a platform whose float has an infinity
```

**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 assertion that this does not occur (6.11.2+a[**basic.contract.implicit**]) (F.3.7[ubx:conv.double.out.of.range]).

**UB Annex Entry****F.3.7****[ub:conv.double.out.of.range]****Specified in:** 7.3.10[ubx:conv.double.out.of.range]

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

- 2 *[Example 1:* Assume that double has a larger range of representable values than float and that float has no representation for infinity for the following example.

```
#include <limits>

int main() {
```

```
double d2 = std::numeric_limits<double>::max();
float f = d2; // undefined behaviorerroneous value, d2 is greater than
 // the largest representable float
}
```

— 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 yields an implementation-defined choice between the largest finite value and  $\pm\infty$ , both 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. The table’s --- cells record that no trigger exists on any available platform.

**[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: a value too large for the destination, a negative value converted to an unsigned type, or a NaN or infinity; a fractional part alone never is (-0.5 converts to `unsigned 0`).

```
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:

- <sup>1</sup> A prvalue of a floating-point type can be converted to a prvalue of an integer type. The conversion truncates; that is, the fractional part is discarded. ~~The behavior~~ An unspecified erroneous value is ~~undefined~~ (F.3.8[ubx:conv.fpint.float.not.represented]) produced if the truncated value cannot be represented in the destination type, with an implicit precondition assertion that this does not occur (6.11.2+a[basic.contract.implicit]) (F.3.8[ubx:conv.fpint.float.not.represented]).

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

**UB Annex Entry****F.3.8** **[ub:conv.fpint.float.not.represented]**

**Specified in:** 7.3.11[ubx:conv.fpint.float.not.represented]

When converting a floating-point value to an integer type, if the truncated value is not representable in the destination type, ~~the behavior~~ an erroneous value is ~~undefined~~ produced with an implicit precondition assertion that this does not occur.

- <sup>2</sup> [Example 1: In the following example, assume that `std::float64_t`, `std::int32_t`, and `std::uint32_t` are supported.

```
#include <stdint>
#include <stdfloat>
#include <limits>

int main() {
 std::float64_t d = static_cast<std::float64_t>(std::numeric_limits<std::int32_t>::max()) + 1;
 std::int32_t x1 = d; // undefined behavior erroneous value, 2,147,483,647 + 1 is not representable as
```

```

std::int32_t
std::int32_t x2 = std::numeric_limits<std::float64_t>::infinity();
// undefined behaviorerroneous value, infinity is not representable as

std::int32_t
std::int32_t x3 = -std::numeric_limits<std::float64_t>::infinity();
// undefined behaviorerroneous value, negative infinity is not representable as

std::int32_t
std::int32_t x4 = std::numeric_limits<std::float64_t>::quiet_NaN();
// undefined behaviorerroneous value, NaN is not representable as std::int32_t
std::uint32_t x5 = -0.5f; // OK, truncated value is 0
}

— 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<br>UBSan<br>UBSan | P3850<br>UBSan | P3850   | P3850   | P3850<br>UBSan   | P3850<br>UBSan   | ✓         | ✓      | P3850<br>UBSan | P3850<br>UBSan | P3850   | P3850<br>UBSan | P3850<br>UBSan   | P3850<br>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 stack 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 type, for every integer and enumeration destination.

When continuing past a violation without terminating – the *ignore*, *observe*, and *noexcept-observe* semantics – the conversion is not performed and the erroneous 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 only on a platform whose float has no
 // infinity and a maximum below ULLONG_MAX (IEEE float)
 // holds this value)
```

**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 assertion that this does not occur (6.11.2+a[basic.contract.implicit]) (F.3.9[ubx:conv.fpint.int.not.represented]). If the source type is bool, the value false is converted to zero and the value true is converted to one.

**UB Annex Entry****F.3.9****[ub:conv.fpint.int.not.represented]**

**Specified in:** 7.3.11[ubx:conv.fpint.int.not.represented]

When converting a value of integer or unscoped enumeration type to a floating-point type, if the value is not representable in the destination type ~~the behavior~~an erroneous value is undefined produced with an implicit precondition assertion that this does not occur.

2 [Example 1:

```
int main() {
 unsigned long long x2 = -1;
 float f = x2; // undefined behaviorerroneous value on systems where f does not include
```

```

 // a representation for infinity and the maximum value for float
 // is smaller than the maximum value for unsigned long long
}
— end example]
```

## 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. The table's --- cells record that no trigger exists on any available platform.

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

**summary:** static\_cast-ing an out-of-range integral or floating-point value into an unscoped enum without a fixed underlying type.

**runtime-checkable:** Yes

**locally-checkable:** Yes

**checking:** Insert a check of whether the value is in the valid range

**replacement:** Coerce into erroneous value

**implementations:** Integral sources are caught by Clang’s constant evaluator, and floating-point sources by GCC’s; both sources are caught by Clang’s UBSan enum check (on the load) and by the P3850 prototype at the cast (both compilers, all semantics).

**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]
}
```

A floating-point value cast to such an enumeration is first converted to the underlying type and then to the enumeration, so the undefined behavior arises in that second conversion: static\_cast<A>(4.0) is undefined behavior just as static\_cast<A>(4) is. An enumeration with a fixed underlying type has the range of that type, so a cast to it is never this undefined behavior.

- `expr.static.cast.enum.outside.range.integral` – the source is of integral or enumeration type
- `expr.static.cast.enum.outside.range.floating` – the source is of floating-point type

**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 assertion that this does not occur (6.11.2+a[basic].

contract.implicit) (F.3.17[ub: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.3.17** [ub:expr.static.cast.enum.outside.range]

**Specified in:** 7.6.1.9[ub:expr.static.cast.enum.outside.range]

If the enumeration type does not have a fixed underlying type, the value is unchanged if the original value is within the range of the enumeration values (9.8.1[dcl.enum]), and otherwise, an ~~the behavior~~ erroneous value is ~~undefined~~ produced with an implicit precondition assertion that this does not occur.

2

[Example 1:

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

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

— end example]

## 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<br>UBSan | P3850<br>UBSan | P3850   | P3850<br>UBSan   | P3850<br>UBSan   |                   |
| ↪ ...integral                       | ✗         | ✓      | P3850  | P3850         | P3850   | P3850   | P3850            | P3850            | ✓         | ✓      | P3850  | P3850<br>UBSan | P3850<br>UBSan | P3850   | P3850<br>UBSan   | P3850<br>UBSan   | CE                |
| ↪ ...floating                       | ✓         | ✓      | P3850  | P3850         | P3850   | P3850   | P3850            | P3850            | ✗         | ✓      | P3850  | P3850<br>UBSan | P3850<br>UBSan | P3850   | P3850<br>UBSan   | P3850<br>UBSan   | CE                |

## Implementation Experience

### Existing Practice

GCC and Clang diverge cleanly here. Clang’s constant evaluator rejects an out-of-range integral source (“integer value 4 is outside the valid range of values [0, 3] for the enumeration type ‘A’”) but accepts a floating-point source (stock Clang and ours; not yet reported upstream). GCC’s

constant evaluator does the reverse: it accepts every out-of-range integral or enumeration source (GCC PR95701), and rejects a floating-point source, though with an unhelpful message (“(unsigned int)4.0e+0’ is not a constant expression”).

At run time, Clang’s `-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, for either source. GCC’s `-fsanitize=enum` reports nothing for these enumerations.

### 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 subgroups are configurable on their own, as `ub:expr.static.cast.enum.outside.range.integral` and `ub:expr.static.cast.enum.outside.range.floating`, by the type of the source.

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, for an integral source and for a floating-point source after its conversion to the underlying type. 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). The second seed of each example exercises an enumeration with a negative enumerator (range `[-2,1]`); the range test is made one bit wider than both the source and the range, so an unsigned source cannot wrap into it.

On GCC this is the only run-time check; 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 is unspecified an unspecified erroneous value is undefined produced with an implicit precondition assertion that this does not occur (6.11.2+a[basic.contract.implicit]) (F.3.18[ubx:expr.static.cast.fp.outside.range]).

**UB Annex Entry****F.3.18****[ub:expr.static.cast.fp.outside.range]****Specified in:** 7.6.1.9[ubx:expr.static.cast.fp.outside.range]

An explicit conversion of a floating-point value that is outside the range of the target type has-undefined-behavior produces an erroneous value with an implicit precondition assertion that this does not occur.

- 2 [Example 1: If `float` does not adhere to ISO/IEC 60559 and cannot represent positive infinity, a sufficiently large double value will be outside the (finite) range of `float`.

```
void f() {
 double d = FLT_MAX;
 d *= 16;
 float f = static_cast<float>(d); // undefined behavior erroneous value
}
```

— *end example*]**Implementation Status**

(full table in Section 3)

| Group                                          | GCC              |               |               |                      |                |                |                         |                         | Clang            |               |               |                      |                |                |                         |                         | compiler explorer |
|------------------------------------------------|------------------|---------------|---------------|----------------------|----------------|----------------|-------------------------|-------------------------|------------------|---------------|---------------|----------------------|----------------|----------------|-------------------------|-------------------------|-------------------|
|                                                | <i>constexpr</i> | <i>assume</i> | <i>ignore</i> | <i>quick-enforce</i> | <i>enforce</i> | <i>observe</i> | <i>noexcept-enforce</i> | <i>noexcept-observe</i> | <i>constexpr</i> | <i>assume</i> | <i>ignore</i> | <i>quick-enforce</i> | <i>enforce</i> | <i>observe</i> | <i>noexcept-enforce</i> | <i>noexcept-observe</i> |                   |
| <code>expr.static.cast.fp.outside.range</code> | —                |               |               |                      |                |                |                         |                         | —                |               |               |                      |                |                |                         |                         | N/A               |

**Implementation Experience***Existing Practice*

No available platform has this undefined behavior, so the row is empty: the `constexpr` cells are --- and the Compiler Explorer cell is N/A. An out-of-range explicit floating-to-floating conversion rounds to  $\pm\infty$  on every available IEEE platform, a representable destination value. Constant evaluation accepts the conversion on both compilers, consistent with the absence of undefined behavior 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:** All three cases are caught by constant evaluation and the P3850 prototype (both compilers, all semantics); the integer cases also by UBSan’s `integer-divide-by-zero` check, and floating-point division by zero by UBSan’s `float-divide-by-zero` check, which is not part of the `undefined` group.

**Description**

Dividing (or taking the remainder of) a value by zero is undefined behavior; P3100 makes the result an unspecified erroneous value, with an implicit precondition assertion that this does not occur. 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`)

Each subgroup includes the compound-assignment forms (`/=` and `%=`).

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

**UB Annex Entry****F.3.30**

[ub:expr.mul.div.by.zero]

Specified in: 7.6.5[ubx:expr.mul.div.by.zero]

Division by zero produces an erroneous value with an implicit precondition assertion that this does not occur.

2

[Example 1:

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

— end example]

**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 | ✓         | ✓      | P3850  | P3850<br>UBSan | P3850   | P3850   | P3850<br>UBSan   | P3850<br>UBSan   | ✓         | ✓      | P3850  | P3850<br>UBSan | P3850<br>UBSan | P3850   | P3850<br>UBSan   | P3850<br>UBSan   |                   |
| ↪ ...int.div         | ✓         | ✓      | P3850  | P3850<br>UBSan | P3850   | P3850   | P3850<br>UBSan   | P3850<br>UBSan   | ✓         | ✓      | P3850  | P3850<br>UBSan | P3850<br>UBSan | P3850   | P3850<br>UBSan   | P3850<br>UBSan   | CE                |
| ↪ ...int.mod         | ✓         | ✓      | P3850  | P3850<br>UBSan | P3850   | P3850   | P3850<br>UBSan   | P3850<br>UBSan   | ✓         | ✓      | P3850  | P3850<br>UBSan | P3850<br>UBSan | P3850   | P3850<br>UBSan   | P3850<br>UBSan   | CE                |
| ↪ ...fp.div          | ✓         | ✓      | P3850  | P3850<br>UBSan | P3850   | P3850   | P3850<br>UBSan   | P3850<br>UBSan   | ✓         | ✓      | P3850  | P3850<br>UBSan | P3850<br>UBSan | P3850   | P3850<br>UBSan   | P3850<br>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*. Each subgroup is a group of its own on both compilers – `ub:expr.mul.div.by.zero.int.div`, `ub:expr.mul.div.by.zero.int.mod` and `ub:expr.mul.div.by.zero.fp.div` – so each can be configured separately; `ub:expr.mul.div.by.zero.int` selects both integer cases.

The floating-point check tests whether the divisor compares equal to zero, so `-0.0` is caught and a NaN divisor is not. The `float-divide-by-zero` sanitizer described above is also routed to the contract-violation handler by the P3850 branches.

When continuing past a violation without terminating – the *ignore*, *observe*, and *noexcept-observe* semantics – the division is not performed and the erroneous 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. P3100 makes the result an unspecified erroneous value, with an implicit precondition assertion that this does not occur.

```
int x = std::numeric_limits<int>::min() / -1; // undefined behavior
int y = std::numeric_limits<int>::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 (including `/=` and `%=`). The division `INT_MIN / -1` also falls under the general arithmetic-overflow rule (`expr.expr.eval`); the P3850 check attributes it to this group.

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

**F.3.31**

[ub:expr.mul.representable.type.result]

**Specified in:** 7.6.5[ubx:expr.mul.representable.type.result]

If the quotient  $a/b$  is representable in the type of the result,  $(a/b)*b + a\%b$  is equal to  $a$ ; otherwise, ~~the behavior of~~ both  $a/b$  and  $a\%b$  ~~produce is undefined~~ an erroneous value with an implicit precondition assertion that this does not occur.

2

[*Example 1:* In the following example, assume that `std::int32_t` is supported (17.4.1[cstdint.syn]) and is `int`. Dividing the minimum `std::int32_t` value of  $-2,147,483,648$  by  $-1$  yields  $2,147,483,648$  which is not representable by `std::int32_t`.

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

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

— end example]

**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<br>UBSan | P3850<br>UBSan | P3850   | P3850   | P3850<br>UBSan   | P3850<br>UBSan   | ✓         | ✓      | P3850  | P3850<br>UBSan | P3850<br>UBSan | P3850   | P3850<br>UBSan   | P3850<br>UBSan   |                   |
| ↪ ...int.div                       | ✓         | ✓      | P3850<br>UBSan | P3850<br>UBSan | P3850   | P3850   | P3850<br>UBSan   | P3850<br>UBSan   | ✓         | ✓      | P3850  | P3850<br>UBSan | P3850          | P3850   | P3850<br>UBSan   | P3850<br>UBSan   | CE                |
| ↪ ...int.mod                       | ✓         | ✓      | P3850<br>UBSan | P3850<br>UBSan | P3850   | P3850   | P3850<br>UBSan   | P3850<br>UBSan   | ✓         | ✓      | P3850  | P3850<br>UBSan | P3850          | P3850   | P3850<br>UBSan   | P3850<br>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`; each subgroup is a group of its own on both compilers (`.int.div` and `.int.mod`), so each can be configured separately. The general signed `+`, `-`, `*` overflow is checked under `[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 erroneous value 0 is produced in its place.

**GCC:** The GCC prototype supports all seven evaluation semantics, including the throwing *enforce* and *observe*; the check guards the single overflowing case (`INT_MIN / -1`). Examples of every semantic are available on <https://godbolt.org/z/fn9c4cz3q>.

**Clang:** The Clang prototype likewise supports all available semantics. Examples of every semantic are available on <https://godbolt.org/z/1scx816P4>.

**[expr.sub.pointers.rep]**

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

**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. Reaching this case requires a single object whose size exceeds `PTRDIFF_MAX`.

```
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 that value is not in the range of representable values of type `std::ptrdiff_t`, the **behavior** result is **undefined** an unspecified erroneous value and there is an implicit precondition assertion that this does not occur (6.11.2+a[basic.contract.implicit]) (F.3.33[ubx:expr.sub.pointers.rep]).
- (5.3) — Otherwise, the behavior is undefined (F.3.34[ubx:expr.add.sub.diff.pointers]).

**UB Annex Entry****F.3.33****[ub:expr.sub.pointers.rep]**

**Specified in:** 7.6.6[ubx:expr.sub.pointers.rep]

Subtracting two expressions of pointer type where the difference is not representable in `std::ptrdiff_t` **has undefined behavior** produce an erroneous value with an implicit precondition assertion that this does not occur.

- 2 [Example 1: This example requires being able to create a single object whose size is larger than the maximum value representable by `std::ptrdiff_t`, in order to get two pointers whose difference can be computed, and platforms that allow this are rare (6.8.2[intro.object], Annex B[implimits]).

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

```
char big[static_cast<std::size_t>(PTRDIFF_MAX) + 1];
std::ptrdiff_t d = big + sizeof(big) - big; // undefined behavior erroneous value
```

— end example]

## 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.rep | —         | ✓      |        |               |         |         |                  |                  | —         | ✓      |        |               |         |         |                  |                  | CE                |

## Implementation Experience

### Existing Practice

This undefined behavior is unreachable on every platform we have access to; as the annex notes, “platforms that allow this are rare.” Both compilers reject the example’s array as too large, and the Compiler Explorer example’s allocation of such an object always fails. For the same reason no constexpr trigger exists (hence the ---).

**[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.

```
unsigned int y1 = 1 << -1; // undefined behavior: negative shift
unsigned int y2 = 1 << std::numeric_limits<unsigned int>::digits;
 // undefined behavior: shift == bit width
unsigned int y3 = 1 >> std::numeric_limits<unsigned int>::digits;
 // 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`

The width is that of the promoted left operand (a `char` shifted by 8 is fine; by 32 it is not), and the compound forms `<<=` and `>>=` are included.

**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.3.36[ubx:expr.shift.neg.and.width]) ~~if an implicit precondition assertion that the right operand is non-negative, or greater and less than or equal to the width of the promoted left operand (6.11.2+a[basic.contract.implicit]) (F.3.36[ubx:expr.shift.neg.and.width])~~; if not, an unspecified erroneous value is produced.

**UB Annex Entry**

**F.3.36**

[ub:expr.shift.neg.and.width]

**Specified in:** 7.6.7[ubx:expr.shift.neg.and.width]

Shifting by a negative amount or equal or greater than the bit-width of a type produces ~~has-undefined-behavior~~ an erroneous value with an implicit precondition assertion that this does not occur.

2

[Example 1:

```
#include <limits>
unsigned int y1 = 1 << -1; // undefined behaviorerroneous value, shift is negative
unsigned int y2 = 1 << std::numeric_limits<unsigned int>::digits;
// undefined behaviorerroneous value, shift is equal to bit width
unsigned int y3 = 1 >> std::numeric_limits<unsigned int>::digits;
// undefined behaviorerroneous value, shift is equal to bit width
```

— end example]

**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<br>UBSan | P3850   | P3850   | P3850<br>UBSan   | P3850<br>UBSan   | ✓         | ✓      | P3850  | P3850<br>UBSan | P3850<br>UBSan | P3850   | P3850<br>UBSan   | P3850<br>UBSan   |                   |
| ↪ ... shl                | ✓         | ✓      | P3850  | P3850<br>UBSan | P3850   | P3850   | P3850<br>UBSan   | P3850<br>UBSan   | ✓         | ✓      | P3850  | P3850<br>UBSan | P3850<br>UBSan | P3850   | P3850<br>UBSan   | P3850<br>UBSan   | CE                |
| ↪ ... shr                | ✓         | ✓      | P3850  | P3850<br>UBSan | P3850   | P3850   | P3850<br>UBSan   | P3850<br>UBSan   | ✓         | ✓      | P3850  | P3850<br>UBSan | P3850<br>UBSan | P3850   | P3850<br>UBSan   | P3850<br>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. The stock check can continue past its report, 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. Each subgroup is a group of its own on both compilers (`ub:expr.shift.neg.and.width.shl` and `.shr`), so each can be configured separately.

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/GKM3xG118>.

**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/7PbPTaY9x>.

**[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**

Reaching the closing brace of a function whose return type is not `void` – other than `main` (which returns 0) or a coroutine (see [ub:stmt.return.coroutine.flow.off]) – without executing a `return` statement is undefined behavior: 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 initializes ~~results in undefined behavior~~ the return value’s storage with erroneous values if the return type is not *cv* void, with an implicit precondition assertion that this does not occur.

- 2 [Example 1:

```
int f(int x) {
 if (x)
 return 1;
 // undefined behavior if x is 0
}
```

```

void b() {
 int x = f(0); // undefined behavior; using 0 as an argument will cause f(...) to flow
 off the end without a return statement
 int x = f(0); // erroneous value and precondition violation, using 0 as an argument will
 cause f(...) to flow off the end without a return statement
}

```

— end example]

## 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<br>UBSan<br>UBSan | P3850   | P3850   | P3850<br>UBSan<br>UBSan | P3850<br>UBSan   | ✓         | ✓      | P3850  | P3850<br>UBSan<br>UBSan | P3850   | P3850   | P3850<br>UBSan<br>UBSan | P3850<br>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 for `.pure`; the `.nonpure` subgroups support only *assume* and *ignore*, and only GCC's constant evaluation catches `.nonpure.opaque`. UBSan catches `.pure` too (GCC's `unreachable` check, Clang's `builtin` check).

**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 three 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.opaque` – the predicate calls a function the compiler cannot prove side-effect-free (neither `[[gnu::pure]]` nor `[[gnu::const]]`), so it cannot be checked without possibly changing the program, and only *assume* and *ignore* apply (a configured checking semantic clamps to *ignore*)
- `dcl.attr.assume.false.nonpure.side.effects` – the predicate has actual side effects (for example `++x > 100`); as for `.opaque`, only *assume* and *ignore* apply

## 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~~ an implicit precondition assertion is ~~undefined~~ violated.

- 2 [Example 1:

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

int f() {
 return g(-10); // undefined behavior precondition violation, 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                |
|                           |           |        | UBSan  |               |         |         | UBSan            | UBSan            |           |        |        | UBSan         |         |         | UBSan            | UBSan            |                   |
|                           |           |        | UBSan  |               |         |         | UBSan            | UBSan            |           |        |        | UBSan         |         |         | UBSan            | UBSan            |                   |
| ↪ ...nonpure.opaque       | ✓         | ✓      | P3850  |               |         |         |                  |                  | ✗         | ✓      | P3850  |               |         |         |                  |                  | CE                |
| ↪ ...nonpure.side.effects | ✗         | ✓      | P3850  |               |         |         |                  |                  | ✗         | ✓      | P3850  |               |         |         |                  |                  | CE                |

---

## Implementation Experience

---

### *Existing Practice*

UBSan catches a false assumption whose predicate the compiler evaluates (`.pure`), on both compilers and with no handler: GCC lowers such an assumption to an unreachable point, which its `unreachable` check reports (“execution reached an unreachable program point”), and Clang’s builtin check reports it directly (“assumption is violated during execution”). GCC’s check cannot recover (`-fsanitize-recover=unreachable` is rejected), so it offers *quick-enforce* (`-fsanitize-trap=unreachable`) and *noexcept-enforce* (the default); Clang’s also offers *noexcept-observe* (the default, which recovers). Neither checks a `.nonpure` predicate, which neither compiler evaluates. Both checks are integrated with the contract-violation handler on the P3850 branches; as for `[[noreturn]]`, GCC’s *noexcept-observe* aborts once the handler returns.

### *P3850 Implementation*

To enable this check 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.

These two subsets are named `ub:dcl.attr.assume.false.nonpure.opaque` and `ub:dcl.attr.assume.false.nonpure.side.effects`; a predicate containing an assignment, an increment or decrement, or a volatile access is in the latter. The group `ub:dcl.attr.assume.false.nonpure` also selects both.

During constant evaluation, compilers are given the same freedom to evaluate the predicate. Both compilers reject a false side-effect-free predicate; GCC also evaluates calls to `constexpr` functions, which Clang ignores (with a `-Wassume` warning), and neither evaluates a predicate with actual side effects. Since CWG2763 and CWG2924, whether a false assumption makes an expression non-constant is implementation-defined, so these acceptances are permitted.

**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>. Clang’s analysis treats a predicate that may trap but has no side effect, such as `*p > 0`, as checkable (`.pure`), where GCC treats it as `.nonpure.opaque`. For a

predicate it cannot check, Clang's status-quo *assume* emits no optimizer hint, so on Clang *assume* and *ignore* behave the same for both `.nonpure` subgroups.

### 2.3 Leave Behavior as Undefined

In the vast majority of cases there is no viable fallback behavior and so we simply rely on having an implicit contract assertion introduced before the pre-existing undefined behavior and make no other local changes to the wording. This includes two cases, [ub:intro.execution.unsequenced.modification] and [ub:dcl.attr.noreturn.eventually.returns], for which a replacement behavior exists but [P3100R9] leaves the behavior undefined.

**[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 objects to an empty set.

No constexpr trigger exists: constant evaluation does not model implicit object creation (no malloc/memcpy, no reinterpret\_cast, and a static\_cast from void\* must find an existing object of similar type), hence ---.

**[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 storage is used as a type that is not an implicit-lifetime type, so no suitable object can be implicitly created
- `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             | —         | ✓      |        |               |         |         |                  |                  | —         | ✓      |        |               |         |         |                  |                  | <a href="#">CE</a> |
| ↪ ...type.pun                 | —         | ✓      |        |               |         |         |                  |                  | —         | ✓      |        |               |         |         |                  |                  | <a href="#">CE</a> |

**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. No constexpr trigger exists: constant evaluation has no operation that implicitly creates objects and returns a pointer to one (`std::malloc` is not constexpr), hence ---.

**[basic.align.object.alignment]**

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

**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
}
```

- `basic.align.object.alignment.store` – the creation stores to the object (a non-trivial constructor, or `new (m) int(5)`)
- `basic.align.object.alignment.nostore` – the creation performs no store (default-initializing an `int` or a trivially constructible class)

**Wording**

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

- <sup>1</sup> 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                   |         |         | P3850                   | P3850                   | —         | ✓      | P3850  | P3850<br>UBSan          | P3850<br>UBSan | P3850   | P3850<br>UBSan          | P3850<br>UBSan          |                   |
| ↪ ...store                   | —         | ✓      | P3850  | P3850<br>UBSan<br>UBSan |         |         | P3850<br>UBSan<br>UBSan | P3850<br>UBSan<br>UBSan | —         | ✓      | P3850  | P3850<br>UBSan<br>UBSan | P3850<br>UBSan | P3850   | P3850<br>UBSan<br>UBSan | P3850<br>UBSan<br>UBSan | CE                |
| ↪ ...nostore                 | —         | ✓      | P3850  | P3850                   | P3850   | P3850   | P3850                   | P3850                   | —         | ✓      | P3850  | P3850<br>UBSan<br>UBSan | P3850          | P3850   | P3850<br>UBSan<br>UBSan | P3850<br>UBSan<br>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 so catches `.store` on both compilers, with one gap on Clang: it checks the alignment it derives for the address rather than the one the type requires, and an address formed from a character array keeps the array’s alignment of 1, so the access is not checked. `*(int *)&buf[o]` has never been checked (Clang 18 through trunk), and since Clang 22 neither has `*(int *)&(buf + o)`, nor a placement `new` that stores through `buf + o` (llvm#177682, the access regression; the other forms are not yet reported upstream); the same accesses through a pointer variable are checked. Clang also checks the pointer passed to placement `new`, so it catches `.nostore`, including the creation of an empty class, subject to the same gap: `new (s + 1) S()` above is missed since Clang 22 (Clang

18 through 21 caught it), `new (&s[1]) S()` always, though the same creation through a pointer variable is caught. GCC checks only the typed stores of the initialization, never the pointer passed to placement `new`, so it misses every `.nostore` creation (not yet reported upstream).

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, so neither subgroup has a constant-evaluation trigger (---).

### *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*. The subgroups are configurable on their own, as `ub:basic.align.object.alignment.store` and `ub:basic.align.object.alignment.nostore`.

`.store` is checked at each access, reusing the compilers' existing null/alignment instrumentation. `.nostore` is checked on the pointer passed to a placement `new` whose new-expression stores nothing (no initialization is performed, or the class is empty), before the object is created; a placement `new` whose initialization stores is left to the `.store` check, so a misaligned creation is reported once. Both checks catch the character-array forms UBSan misses (`*(int *) (buf + o)`, `*(int *)&buf[o]`, `new (s + o) S()`, `new (&s[o]) S()`) on both compilers.

On GCC the `.store` check is a middle-end check; 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*. GCC's `.nostore` check is a front-end check and supports all seven semantics, as both of Clang's checks do. 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). The `.nostore` check is built with the placement `new` itself (`build_new_1, cp/init.cc`), after its null check. 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). The pointer of a placement `new` is checked against the alignment of the type created, as `.nostore` or `.store` by whether the initialization stores (`EmitImplicitPlacementNewGuards`), which catches an empty class too. 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:** Rejected during constant evaluation by Clang, and by GCC only with the P3850 GCC branch.

**Description**

This rule is one of the family governing what a pointer to an object outside its lifetime may be used for: here, as the operand of a *delete-expression*. For a class with a non-trivial destructor, the delete-expression runs 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
}
```

**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<sup>16</sup> 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]

---

<sup>16</sup> 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 pointer is used as the operand of a *delete-expression*.

[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
}
```

— end example]

Leave Behavior as Undefined

`lifetime.outside.pointer.delete`

## Implementation Status

(full table in Section 3)

| Group                                        | GCC              |               |               |                      |                |                |                         |                         | Clang            |               |               |                      |                |                |                         |                         | compiler explorer |
|----------------------------------------------|------------------|---------------|---------------|----------------------|----------------|----------------|-------------------------|-------------------------|------------------|---------------|---------------|----------------------|----------------|----------------|-------------------------|-------------------------|-------------------|
|                                              | <i>constexpr</i> | <i>assume</i> | <i>ignore</i> | <i>quick-enforce</i> | <i>enforce</i> | <i>observe</i> | <i>noexcept-enforce</i> | <i>noexcept-observe</i> | <i>constexpr</i> | <i>assume</i> | <i>ignore</i> | <i>quick-enforce</i> | <i>enforce</i> | <i>observe</i> | <i>noexcept-enforce</i> | <i>noexcept-observe</i> |                   |
| <code>lifetime.outside.pointer.delete</code> | ✗<br>P3850       | ✓             |               |                      |                |                |                         |                         | ✓                | ✓             |               |                      |                |                |                         |                         | CE                |

## Implementation Experience

### *Existing Practice*

Deleting an object whose lifetime has already ended is rejected during constant evaluation by Clang, whether or not the destructor is trivial. Stock GCC accepts it ([GCC PR126425](#)); the P3850 GCC branch rejects it.

No stock runtime sanitizer catches the example: the single `operator delete` call is not a double-free, so ASan stays 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:** Rejected in constant evaluation by Clang; GCC rejects data-member access but not a non-virtual member-function call.

**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
}
```

- `lifetime.outside.pointer.member.data` – reading or writing a non-static data member (the example above)
- `lifetime.outside.pointer.member.call` – calling a non-static member function

**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<sup>16</sup> 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.ctor]. 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]

---

<sup>16</sup> 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.

[Example 1:

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

float f() {
 S s;
 S* p = &s;
 s.~S();
}
```

2

```
 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 | no-exception-enforce | no-exception-observe | constexpr | assume | ignore | quick-enforce | enforce | observe | no-exception-enforce | no-exception-observe |                   |
| lifetime.outside.pointer.member | ✗         | ✓      |        |               |         |         |                      |                      | ✓         | ✓      |        |               |         |         |                      |                      |                   |
| ↪ ...data                       | ✓         | ✓      |        |               |         |         |                      |                      | ✓         | ✓      |        |               |         |         |                      |                      | CE                |
| ↪ ...call                       | ✗         | ✓      |        |               |         |         |                      |                      | ✓         | ✓      |        |               |         |         |                      |                      | CE                |

## Implementation Experience

### Existing Practice

Both compilers' constant evaluators reject the `.data` form. For the `.call` form, Clang rejects any member call, but GCC (stock and the P3850 GCC branch) accepts a non-virtual call (not yet reported upstream); it rejects a virtual call, which reads the `vp`tr.

The storage is still allocated, so ASan does not apply. Clang's MemorySanitizer, which poisons an object's storage when its non-trivial destructor runs, reports a virtual call through the poisoned `vp`tr, and a later use of a value read from the dead object, but neither a data-member access itself nor a non-virtual call; the table shows no sanitizer for these rows.

**[lifetime.outside.pointer.virtual]**

**summary:** converting a pointer outside its lifetime into a virtual base class pointer.  
**runtime-checkable:** Yes  
**locally-checkable:** No  
**checking:** Track whether storage is associated with an object of correct type within its lifetime  
**replacement:** None  
**implementations:** Rejected in constant evaluation by GCC, and by Clang only with the P3850 Clang branch; for polymorphic classes, heuristically caught by UBSan's `vptra` check on GCC and by MemorySanitizer on Clang.

**Description**

Converting (implicitly or with `static_cast`) 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<sup>16</sup> 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.ctor]. 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]

---

<sup>16</sup> 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 converted (7.3.12[conv.ptr], 7.6.1.9[expr.static.cast]) 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* b1 = p; // undefined behavior, implicit conversion
 B* b2 = static_cast<B*>(p); // undefined behavior, explicit conversion
}
```

— 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 <sup>†</sup> |         |         | UBSan <sup>†</sup><br>UBSan <sup>†</sup> | UBSan <sup>†</sup> | ✗<br>P3850 | ✓      |        | MSan <sup>†</sup> |         |         | MSan <sup>†</sup><br>MSan <sup>†</sup> | MSan <sup>†</sup><br>MSan <sup>†</sup> | CE                |

**Implementation Experience***Existing Practice*

The conversion reads the virtual-base offset through the object's `vptr`. For a polymorphic class with a non-trivial destructor, two sanitizers therefore detect it, heuristically:

- GCC's UBSan `vptr` check reports it, because the destroyed object's `vptr` no longer names its most-derived class. Clang's `vptr` check does not instrument conversions to a virtual base.
- Clang's MemorySanitizer reports it: Clang poisons an object's storage, `vptr` included, when its destructor runs (`-fsanitize-memory-use-after-dtor`, on by default), and the offset load reads the poisoned `vptr`. MSan provides *noexcept-enforce* by default, *noexcept-observe* with `-fsanitize-recover=memory`, and, integrated on the P3850 Clang branch, *quick-enforce*.

Neither detects the non-polymorphic example above, nor storage that has been rewritten since the destructor ran; ASan does not apply, as the storage is still allocated.

GCC's constant evaluator rejects the conversion, including through `static_cast` to a base of a virtual base. Stock Clang accepts it (not yet reported upstream); the P3850 Clang branch rejects it.

**[lifetime.outside.pointer.dynamic.cast]**

**summary:** `dynamic_cast` through a pointer to an object outside its lifetime.  
**runtime-checkable:** Yes  
**locally-checkable:** No  
**checking:** Track whether storage is associated with an object of correct type within its lifetime  
**replacement:** None  
**implementations:** Rejected in constant evaluation by Clang, and by GCC for automatic objects only (GCC misses a destroyed heap object whose storage has not been released); no runtime detection.

**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
}
```

- `lifetime.outside.pointer.dynamic.cast.automatic` – the object has automatic storage duration (the example above)
- `lifetime.outside.pointer.dynamic.cast.dynamic` – the object was created in allocated storage, destroyed, and its storage not yet released

**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<sup>16</sup> 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.ctor]. 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]

---

<sup>16</sup> 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 | no-exception-enforce | no-exception-observe | constexpr | assume | ignore | quick-enforce | enforce | observe | no-exception-enforce | no-exception-observe |                   |
| lifetime.outside.pointer.dynamic.cast | ✗         | ✓      |        |               |         |         |                      |                      | ✓         | ✓      |        |               |         |         |                      |                      |                   |
| ↪ ...automatic                        | ✓         | ✓      |        |               |         |         |                      |                      | ✓         | ✓      |        |               |         |         |                      |                      | CE                |
| ↪ ...dynamic                          | ✗         | ✓      |        |               |         |         |                      |                      | ✓         | ✓      |        |               |         |         |                      |                      | CE                |

## Implementation Experience

### Existing Practice

No stock sanitizer reports this undefined behavior. GCC does not instrument `dynamic_cast` for `vp_ptr`; neither Clang’s `vp_ptr` check nor its `MemorySanitizer` reports the cast; and the storage is still allocated, so ASan does not apply.

The constant evaluators of both compilers reject the `.automatic` form. GCC accepts the `.dynamic` form, stock and the P3850 GCC branch alike (not yet reported upstream; the umbrella report [GCC PR107529](#) covers destroyed heap objects); Clang rejects it.

**[lifetime.outside.glvalue.access]**

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

**runtime-checkable:** Yes

**locally-checkable:** No

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

**replacement:** None

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

**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.ctor]. 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 | ✗<br>P3850 | ✓      |        |               |         |         |                  |                  | ✓         | ✓      |        |               |         |         |                  |                  | CE                |

## Implementation Experience

### Existing Practice

No runtime tool catches this category on either compiler. AddressSanitizer cannot: [basic.life] restricts this category to storage that has not been released, which ASan treats as live. (Use after the storage is released is [ub:basic.compound.invalid.pointer].) MemorySanitizer’s use-after-destructor poisoning applies only to non-trivial destructors, whose member accesses fall under [ub:class.cdtor.after.dtor] first.

Constant evaluation rejects the annex example (and reads or writes of a destroyed local object) on both compilers. Stock GCC misses a write to a destroyed heap object whose storage is still allocated, which the conformance example uses, and a read of one crashes its checking builds (GCC PR107529); the P3850 GCC branch rejects both.

**[lifetime.outside.glvalue.member]**

**summary:** calling a non-static member function through a glvalue outside its lifetime.  
**runtime-checkable:** Yes  
**locally-checkable:** No  
**checking:** Track whether storage is associated with an object of correct type within its lifetime  
**replacement:** None  
**implementations:** Caught by constant evaluation on Clang; GCC's constant evaluator accepts the call (miss).

**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 by Clang but not by GCC, which rejects the annex example only at the later scope-exit destruction of `a`, and accepts the call itself (also before construction, or on a destroyed heap object); this is not yet reported upstream. 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:** Rejected in constant evaluation by GCC except when the glvalue names the complete object, and by Clang only with the P3850 Clang branch; for polymorphic classes, heuristically caught by MemorySanitizer on Clang and, except for a named complete object, by UBSan's `vp` check on GCC.

**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
}
```

- `lifetime.outside.glvalue.virtual.named` – the glvalue names the complete object, so its most-derived type is statically known (the example above)
- `lifetime.outside.glvalue.virtual.unnamed` – the most-derived type is not statically known (for example, the glvalue is a reference parameter)

**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.ctor]. 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 | ✗         | ✓      |        |               |         |         |                  |                  | ✗<br>P3850 | ✓      |        | MSan†         |         |         | MSan†<br>MSan†    | MSan†<br>MSan†   |                  |
| ↪ ...named                       | ✗         | ✓      |        |               |         |         |                  |                  | ✗<br>P3850 | ✓      |        | MSan†         |         |         | MSan†<br>MSan†    | MSan†<br>MSan†   | CE               |
| ↪ ...unnamed                     | ✓         | ✓      |        | UBSan†        |         |         | UBSan†           | UBSan†           | ✗<br>P3850 | ✓      |        | MSan†         |         |         | MSan†<br>MSan†    | MSan†<br>MSan†   | CE               |

## Implementation Experience

### Existing Practice

Unless the most-derived type is statically known, the binding reads the virtual-base offset through

the object's `vp`tr. For a polymorphic class with a non-trivial destructor, two sanitizers therefore detect it, heuristically:

- GCC's UBSan `vp`tr check reports the `.unnamed` form, because the destroyed object's `vp`tr no longer names its most-derived class. It does not see the `.named` form, whose offset GCC folds. Clang's `vp`tr check does not instrument conversions to a virtual base.
- Clang's MemorySanitizer reports both forms: Clang poisons an object's storage, `vp`tr included, when its destructor runs (`-fsanitize-memory-use-after-dtor`, on by default), and the offset load reads the poisoned `vp`tr. MSan provides *noexcept-enforce* by default, *noexcept-observe* with `-fsanitize-recover=memory`, and, integrated on the P3850 Clang branch, *quick-enforce*.

Neither detects the non-polymorphic example above, nor storage that has been rewritten since the destructor ran; ASan does not apply, as the storage is still allocated.

During constant evaluation, GCC rejects the `.unnamed` form but folds the virtual-base offset for a named complete object, so it accepts the `.named` form, the example above (not yet reported upstream; the umbrella report [GCC PR102284](#) covers uses of an object after an explicit destructor call). Stock Clang accepts both forms (not yet reported upstream); the P3850 Clang branch rejects both.

**[lifetime.outside.glvalue.dynamic.cast]**

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

**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
}
```

- `lifetime.outside.glvalue.dynamic.cast.dynamic.cast` – the glvalue is the operand of `dynamic_cast`
- `lifetime.outside.glvalue.dynamic.cast.typeid` – the glvalue is the operand of `typeid`

**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.ctor`]. 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 | ✗         | ✓      |        |               |         |         |                  |                  | ✓         | ✓      |        |                   |         |         |                   |                   |                   |
| ↪ ...dynamic.cast                     | ✓         | ✓      |        |               |         |         |                  |                  | ✓         | ✓      |        |                   |         |         |                   |                   | CE                |
| ↪ ...typeid                           | ✗         | ✓      |        |               |         |         |                  |                  | ✓         | ✓      |        | MSan <sup>†</sup> |         |         | MSan <sup>†</sup> | MSan <sup>†</sup> | CE                |

## Implementation Experience

### Existing Practice

No runtime tool catches the `dynamic_cast` form: the object's storage is still allocated, so AddressSanitizer is silent, and after `d.~D()` the `vp`tr points at `B`'s vtable, which Clang's `-fsanitize=vp`tr

accepts (GCC does not instrument `dynamic_cast` for `vp_ptr` at all). Clang's `vp_ptr` check fires only once the storage has been freed, which is a different undefined behavior ([ub:expr.type.reference.lifetime]).

Clang's MemorySanitizer catches the `.typeid` form, heuristically: Clang poisons an object's storage, `vp_ptr` included, when its destructor runs (`-fsanitize-memory-use-after-dtor`, on by default), and `typeid` reads the poisoned `vp_ptr` in instrumented code. The `vp_ptr` read inside `dynamic_cast` happens in the uninstrumented runtime library, so the `.dynamic.cast` form is not reported, and at `-O2` an unused `typeid` result is removed along with the read. MSan provides *noexcept-enforce* by default, *noexcept-observe* with `-fsanitize-recover=memory`, and, integrated on the P3850 Clang branch, *quick-enforce*.

In constant evaluation both compilers reject `dynamic_cast` on an object outside its lifetime. Clang also rejects `typeid`; GCC accepts `typeid` when its result is unused (a miss, not yet reported upstream). Stock GCC's checking builds crash on either operation applied to a destroyed heap object whose storage is still allocated ([GCC PR107529](#)); the P3850 GCC branch rejects both.

**[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. The same applies when the lifetime ended without reuse (for example, by an explicit destructor call), and to objects of static or thread storage duration, whose implicit destructor runs at program or thread exit.

```
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,<sup>17</sup> 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]
```

<sup>17</sup> 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.

There is no constant-evaluation trigger (---): the different-type form cannot be written, as placement `new` in a constant expression may not change the type of the storage. The no-reuse form (`B b; b.~B(); }`) is rejected by both compilers, but it is the same code as `[ub:class.dtor.no.longer.exists]`, where that detection is shown.

**[creating.within.const.complete.obj]**

**summary:** creating a new object in storage a `const` complete object occupies or used to occupy.

**runtime-checkable:** Yes

**locally-checkable:** No

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

**replacement:** None

**implementations:** Clang’s constant evaluation catches both `.live` and `.reuse` subsets; GCC’s catches neither.

**Description**

The storage of a `const` complete object with static, thread, or automatic storage duration 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 creating any

object in the const object’s storage (reported as “modification of object of const-qualified type”), whether or not the destructor ran first and whether or not the new object’s construction writes anything; GCC accepts both ([GCC PR119304](#)). Only automatic storage duration can be exercised in constant evaluation. 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 replacement or class-specific allocation or deallocation function violating its semantic constraints.

**runtime-checkable:** Partially

**locally-checkable:** No

**checking:** Add postcondition assertions to allocation functions, evaluate them. Some constraints can be expressed this way, some (such as global distinctness) are not locally verifiable.

**replacement:** None

**implementations:** The null-return case is caught by UBSan's `returns-nonnull-attribute` check on Clang.

**Description**

An allocation or deallocation function – a replacement of a global one, or a class-specific one – 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 such a function violates one of those constraints, the behavior is undefined. The other constraints (the result's alignment, its distinctness from other live allocations, and a failure exception matching `std::bad_alloc`) are covered the same way.

```
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
}

struct Widget {
 static void operator delete(void* ptr) noexcept(false) {
 ::operator delete(ptr);
 throw 0; // undefined behavior: a deallocation function
 // must not terminate by throwing
 }
};
```

A replaceable global deallocation function must be `noexcept`, so a `throw` out of one calls `std::terminate` instead; the last constraint is reachable only through a class-specific deallocation function declared `noexcept(false)`.

**Wording**

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

- <sup>3</sup> 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
}
```

*— 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<br>UBSan   | UBSan<br>UBSan   | CE                |

**Implementation Experience***Existing Practice*

This category is an 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. The check recovers with `-fsanitize-recover=returns-nonnull-attribute`, giving *noexcept-observe*.

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

Both compilers warn at compile time when such an `operator new` returns a null pointer constant directly, as in the conformance example; a null pointer computed at run time is not diagnosed.

Other aspects of the behavior are not checked by existing tools. Replacement and class-specific 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:** An access is heuristically caught by AddressSanitizer and rejected during constant evaluation, on both compilers; an indirection that accesses nothing is rejected during constant evaluation by Clang only.

**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(0uz));
 c[0] = 'X'; // undefined behavior
}
```

The same holds for storage from `operator new[] (0)`, `new T[0]`, or `std::allocator<T>::allocate(0)` (which obtains its storage from `::operator new`).

- `basic.stc.alloc.zero.dereference.access` – the indirection reads or writes through the pointer (`c[0] = 'X';`)
- `basic.stc.alloc.zero.dereference.indirection` – the indirection accesses nothing (binding `T& r = *p;`)

**Wording**

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

- <sup>2</sup> 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]).<sup>18</sup>

<sup>18</sup> 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 | ✗         | ✓      |        |                   |         |         |                   |                   | ✓         | ✓      |        |                   |         |         |                   |                   |                   |
| ↪ ...access                      | ✓         | ✓      |        | ASan <sup>†</sup> |         |         | ASan <sup>†</sup> | ASan <sup>†</sup> | ✓         | ✓      |        | ASan <sup>†</sup> |         |         | ASan <sup>†</sup> | ASan <sup>†</sup> | CE                |
| ↪ ...indirection                 | ✗         | ✓      |        |                   |         |         |                   |                   | ✓         | ✓      |        |                   |         |         |                   |                   | CE                |

## Implementation Experience

### Existing Practice

AddressSanitizer catches accesses through 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). ASan checks only loads and stores, so it never sees `.indirection`. It terminates by default (*noexcept-enforce*); `-fsanitize-recover=address` together with ASan's `halt_on_error=0` option (supplied in-source by defining `__asan_default_options` to return it, or through `ASAN_OPTIONS`) gives *noexcept-observe*, and the P3850 branches route it to the contract-violation handler.

During constant evaluation the same access through a zero-length allocation (`std::allocator<T>::allocate(0)` or `new T[0]`) is rejected by both compilers as an out-of-bounds access. Binding a reference through such a pointer (`.indirection`) is rejected by Clang and accepted by GCC (not yet reported upstream).

**[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. Relaxed atomic operations can also produce such pointer values.

```
void f() {
 int x;
 std::uintptr_t pval = reinterpret_cast<std::uintptr_t>(&x) + 1;
 int *p = reinterpret_cast<int*>(pval); // undefined behavior: every value
 // p could take leads to UB

 {
 int y; // y's storage duration begins here, so p cannot be &y
 if (pval == reinterpret_cast<std::uintptr_t>(&y)) {
 *p = 17; // so p is not (&x)+1 either (past-the-end write)
 }
 }
}
```

**Wording**

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

- <sup>5</sup> 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 beginning of 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 <stdint>
void f() {
 int x;
 std::uintptr_t pval = reinterpret_cast<std::uintptr_t>(&x)+1;
 int *p = reinterpret_cast<int*>(pval); // undefined behavior
 {
 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                |

## Implementation Experience

### Existing Practice

No tool checks this category, and none can: the undefined behavior is a constraint on which object a converted pointer may be taken to point to, not a property of any run-time value. It has no constant-evaluation trigger (---) because integer-to-pointer `reinterpret_cast` is not a core constant expression.

**[basic.compound.invalid.pointer]**

**summary:** indirection through a pointer to storage that has been freed or whose duration has otherwise ended.

**runtime-checkable:** Yes

**locally-checkable:** No

**checking:** Track object lifetimes across the entire program

**replacement:** None

**implementations:** An access through the pointer is heuristically caught by AddressSanitizer and rejected during constant evaluation, on both compilers; an indirection that accesses nothing is rejected during constant evaluation by Clang only.

**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. The storage need not be dynamic: a pointer to an automatic variable becomes invalid in the same way when the variable’s block exits.

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

- `basic.compound.invalid.pointer.access` – the indirection reads or writes the object (`int y = *x;`)
- `basic.compound.invalid.pointer.indirection` – the indirection accesses nothing (binding `int& r = *x;`, or forming `&p->m`)

**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 | ✗         | ✓      |        |                   |         |         |                   |                   | ✓         | ✓      |        |                   |         |         |                   |                   |                   |
| ↪ ...access                    | ✓         | ✓      |        | ASan <sup>†</sup> |         |         | ASan <sup>†</sup> | ASan <sup>†</sup> | ✓         | ✓      |        | ASan <sup>†</sup> |         |         | ASan <sup>†</sup> | ASan <sup>†</sup> | CE                |
| ↪ ...indirection               | ✗         | ✓      |        |                   |         |         |                   |                   | ✓         | ✓      |        |                   |         |         |                   |                   | 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. ASan checks only loads and stores, so it never sees `.indirection`.

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 `-fsanitize-recover=address` together with ASan's `halt_on_error=0` option (supplied in-source by defining `__asan_default_options` to return it, or through `ASAN_OPTIONS`) provides the equivalent of the *noexcept-observe* semantic.

During constant evaluation both compilers reject `.access`. For `.indirection`, Clang rejects binding a reference through a pointer to deleted heap storage and GCC accepts it. Neither compiler rejects forming `&p->m`, or binding a reference through a pointer to an automatic variable whose block has exited. None of these misses has been reported upstream yet.

### *P3850 Sanitizer Integration*

On the P3850 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. All three non-throwing checked semantics (*quick-enforce*, *noexcept-enforce*, *noexcept-observe*) are available.

The potentially throwing *enforce* and *observe* semantics are not supported yet — AddressSanitizer's checks are emitted at arbitrary loads and stores, including in `noexcept` functions, and letting them throw would require every instrumented access to be a potential exception-propagation point.

**[intro.execution.unsequenced.modification]**

**summary:** unsequenced side effects (or a side effect and a read) on the same scalar object, single-threaded.

**runtime-checkable:** Yes

**locally-checkable:** Yes

**checking:** Identify all potential read operations that are not sequenced with respect to each given write operation; insert checks to identify if those operations are referencing the same address

**replacement:** Sequence operations in some unspecified order

**note:** Needs more careful specification work, defer to future paper.

**implementations:** None; constant evaluation misses it on both compilers, which only warn at compile time.

**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 (on different threads, or one in a signal handler), 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. The rule also covers starting or ending an object’s lifetime unsequenced with another access to the same storage, e.g.,  $(u.x = 1, 0) + (u.y = 2, 0)$  for a union  $u$ .

```
void g(int i) {
 i = i++ + 1; // OK, 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++ + i;`) on either compiler, and the constant evaluator does not catch it either: both compilers silently pick an evaluation order, for every variation tried (`i++ + i++`, `(i += 1) + i`, `f(i++, i)`, the union example above), although the core-language undefined behavior makes the expression not a constant expression. The constant-evaluation miss is reported upstream as [GCC PR87275](#) and [llvm#37768](#). The only stock signal is a compile-time warning: GCC's `-Wsequence-point` (needs `-Wall`) and Clang's `-Wunsequenced` (default-on; it does not fire on the union example).

**[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 potentially concurrent accesses to the same memory location (on different threads, or one in a signal handler unsequenced with the other), at least one a write and at least one non-atomic, are not ordered by happens-before; the behavior is undefined.

```
int total = 0;
void worker() {
 while (total < 10000000)
 ++total; // read-modify-write not synchronized 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};
```

— *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†<br>TSan†   | TSan†            | —         | ✓      |        | TSan†         |         |         | TSan†<br>TSan†   | TSan†<br>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.

TSan’s detection is demonstrated on both compilers by the runnable example: `-fsanitize=thread` cleanly reports “WARNING: ThreadSanitizer: data race” with both racing stack traces. By default TSan then *continues* – the program runs to completion and only its exit status (66) records the report – which is *noexcept-observe*. No compile-time flag changes this on either compiler: `-fno-sanitize-recover=thread` still continues, Clang accepts `-fsanitize-recover=thread` as a no-op,

and GCC rejects it. Halting at the first report (*noexcept-enforce*) needs TSan's `halt_on_error=1` option, supplied either in-source by defining `__tsan_default_options` to return it or through the `TSAN_OPTIONS` environment variable. TSan has no *quick-enforce* (no `-fsanitize-trap=thread`).

**[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 modify-write operation such as `store_add`), 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 below 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:

<sup>1</sup> 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 resulting infinite loop has 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 | ---       | ✓      |        |               |         |         |                  |                  | ---       | ✓      |        |               |         |         |                  |                  |                   |

## Implementation Experience

### Existing Practice

Nothing checks this category. Constant evaluation rejects any non-terminating loop by its step limit, including well-defined trivial infinite loops, so it does not detect this undefined behavior as such; the *constexpr* cells are ---.

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

**summary:** calling `std::exit` while an object with static or thread storage duration is being destroyed.

**runtime-checkable:** Yes

**locally-checkable:** No, needs new instrumentation of existing functionality

**checking:** Track whether static or thread-local objects are currently being destroyed

**replacement:** None

**implementations:** None

**Description**

Calling `std::exit` while an object with static or thread storage duration is being destroyed – for example from such an object’s destructor, after `main` has returned or `std::exit` has already been called – is undefined behavior. The call need not be recursive: a `thread_local` object’s destructor that calls `std::exit` at a thread’s exit is covered too.

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

int main() {}
// after main returns, destroying ex calls std::exit while ex is
// being destroyed: 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), and there is no native check. It has no constant-evaluation trigger (---): `std::exit` is not `constexpr`, and no static destruction happens during constant evaluation.

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

**summary:** passing through the definition of a block-scope static or thread-local variable after it has been destroyed.

**runtime-checkable:** Yes

**locally-checkable:** No

**checking:** Track the lifetime of static objects

**replacement:** None

**implementations:** None

**Description**

Objects with static storage duration are destroyed in the reverse order their construction completed (thread-local objects likewise, at thread exit). If a function is called during that destruction and control passes through the definition of a block variable that has already been destroyed, the behavior is undefined. (Using the destroyed object indirectly, through a saved pointer, is undefined as well, by the general lifetime rules.)

```
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` complete 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), and there is no native check. It has no constant-evaluation trigger (---): no static or thread-local destruction happens during 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; cannot arise during constant evaluation.

**Description**

This undefined behavior is the general “strict aliasing” rule: accessing an object’s stored value through a glvalue through which it is not *type-accessible* – one whose type is not similar to the object’s dynamic type, its signed/unsigned counterpart, or `char`, `unsigned char`, or `std::byte` – 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  $T_{\text{obj}}$  is *type-accessible* through a glvalue of type  $T_{\text{ref}}$  if  $T_{\text{ref}}$  is similar (7.3.6[conv.qual]) to:

- $T_{\text{obj}}$ ,
- a type that is the signed or unsigned type corresponding to  $T_{\text{obj}}$ , or
- a `char`, `unsigned char`, or `std::byte` type.

If a program attempts to access (3.1[defs.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]).<sup>36</sup> 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*]

<sup>36</sup> 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[defs.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 sanitizer in the default groups catches this category. LLVM’s experimental TypeSanitizer (`-fsanitize=type`, Clang only, not evaluated in the table) does report the example at `-O0`, but not at `-O2` once the accesses are optimized. GCC has no equivalent.

This category cannot arise during constant evaluation: the trigger can only be formed through a `reinterpret_cast` between unrelated pointer types, or a `static_cast` from `void*` to a dissimilar type, neither of which is permitted in constant evaluation.

**[expr.basic.lvalue.union.initialization]**

**summary:** defaulted union copy/move construction or assignment whose argument is not a live object of that union type.

**runtime-checkable:** Yes

**locally-checkable:** No

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

**replacement:** None

**implementations:** Rejected during constant evaluation on both GCC and Clang (inactive member or out-of-lifetime argument).

**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
}

union W { U u; int i; };
W w{.i = 1};
U o = w.u; // undefined behavior: w.u is not within its lifetime
```

**Wording**

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

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

- $T_{\text{obj}}$ ,
- a type that is the signed or unsigned type corresponding to  $T_{\text{obj}}$ , or
- a `char`, `unsigned char`, or `std::byte` type.

If a program attempts to access (3.1[defs.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]).<sup>36</sup> 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*

<sup>36</sup> 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                |

Implementation Experience

Existing Practice

Constant evaluation rejects this on both compilers when the argument is an inactive union member or an object outside its lifetime (the second example above). The `reinterpret_cast` form of the first example is not a constant expression. No sanitizer detects this category at run time.

**[expr.type.reference.lifetime]**

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

**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 – that is, once the storage the referenced object occupied has been released. A reference to an object whose lifetime merely ended, with its storage intact, is not covered here.

```
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
}
```

The same holds when automatic storage ends, e.g., for a reference to a local variable of a block that has exited or of a function that has returned.

**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. Naming the reference emits no memory access, so no runtime sanitizer has anything to instrument. Constant evaluation should reject it (it is core-language undefined behavior) but neither compiler does: the annex example, `&i`, and binding another reference to `i` are all accepted; only a subsequent read through `i` is rejected, as a use-after-free. The miss is not yet reported upstream for either compiler.

**[conv.array.unsimilar.type]**

**summary:** performing an array-to-pointer conversion on an array glvalue whose referenced object’s type is not similar to the expression’s type.

**runtime-checkable:** Yes

**locally-checkable:** No

**checking:** Track the dynamic type of the object in storage and verify that it is similar to the type of the array expression

**replacement:** None

**implementations:** None

**Description**

An array-to-pointer conversion of an array glvalue *E* yields a pointer to the first element of the array *E* refers to. That array object is expected to have a type similar to *E*’s own type. If it does not – typically because *E* was reached through a `reinterpret_cast` to an array type with a different bound or element type – the behavior is undefined, even when every element subsequently accessed through the resulting pointer lies within the actual object.

```
int arr[4] = {};
int* p = reinterpret_cast<int(&)[2]>(arr); // undefined behavior: arr is
 // an array of a different size
```

**Wording**

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

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

**UB Annex Entry****F.3.6****[ub:conv.array.unsimilar.type]****Specified in:** 7.3.3[ubx:conv.array.unsimilar.type]

Performing an array-to-pointer conversion on an expression that refers to an object whose type is not similar (7.3.6[conv.qual]) to the type of the expression has undefined behavior.

- 2 [Example 1:

```
void f() {
 int arr[4] = {};
 int* p = reinterpret_cast<int(&)[2]>(arr); // undefined behavior, arr is
 // an array of a different size
}
```

— 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.array.unsimilar.type | —         | ✓      |        |               |         |         |                  |                  | —         | ✓      |        |               |         |         |                  |                  | CE                |

**Implementation Experience***Existing Practice*

No sanitizer catches this undefined behavior on either compiler: UBSan (including `-fsanitize=bounds`) and ASan are silent, since the underlying object is live and every element access stays within it. There is no native check on either P3850 branch.

No constexpr trigger exists (hence the ---): every way to obtain such an array glvalue (`reinterpret_cast`, or a cast from `void*` to a non-similar array type) is already not a core constant expression, and both compilers reject the `void*` cast as such. GCC accepts a reference `reinterpret_cast` in constant evaluation (not yet reported upstream), which lets it reach this undefined behavior; it then accepts the conversion without diagnosis.

**[conv.ptr.virtual.base]**

**summary:** converting a derived-class pointer to a virtual base when the source isn't a valid, in-lifetime object.

**runtime-checkable:** Yes

**locally-checkable:** No

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

**replacement:** None

**implementations:** UBSan's `vptra` check catches some cases (heuristic): GCC for `.past.end` and `.lifetime`, Clang for `.lifetime` and `.union`; AddressSanitizer catches `.past.end` on a stack array (heuristic). Constant evaluation rejects `.past.end` on both compilers and `.lifetime` on GCC; our P3850 Clang branch also rejects `.lifetime` and `.union`.

**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 depends on the most-derived type of the object). 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];
 D* p = ds + 1; // OK, one past the end
 B* b = p; // undefined behavior: p does not point to a D object
}
```

- `conv.ptr.virtual.base.past.end` – the source is a past-the-end pointer
- `conv.ptr.virtual.base.lifetime` – the source points to storage whose object has not yet been constructed or has already been destroyed
- `conv.ptr.virtual.base.union` – the source points to an inactive union member

The same rule also covers a source that points to an object of an unrelated type; that case is not separately exercised. (The annex's `&ds[1]` form first applies `*` to the past-the-end pointer, so the example above forms the pointer with `ds + 1` instead.)

**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.ctor]), the behavior is undefined (F.3.10[ubx:conv.ptr.virtual.base]). Otherwise, the result is a pointer to the base class subobject of the derived class object.

## UB Annex Entry

### F.3.10

[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 | ✗         | ✓      |        |                                         |         |         |                                                                                    |                                         | ✗<br>P3850 | ✓      |        |                    |         |         |                                          |                    |                   |
| ↪ ...past.end         | ✓         | ✓      |        | UBSan <sup>†</sup><br>ASan <sup>†</sup> |         |         | UBSan <sup>†</sup><br>ASan <sup>†</sup><br>UBSan <sup>†</sup><br>ASan <sup>†</sup> | UBSan <sup>†</sup><br>ASan <sup>†</sup> | ✓          | ✓      |        | ASan <sup>†</sup>  |         |         | ASan <sup>†</sup><br>ASan <sup>†</sup>   | ASan <sup>†</sup>  | CE                |
| ↪ ...lifetime         | ✓         | ✓      |        | UBSan <sup>†</sup>                      |         |         | UBSan <sup>†</sup><br>UBSan <sup>†</sup>                                           | UBSan <sup>†</sup>                      | ✗<br>P3850 | ✓      |        | UBSan <sup>†</sup> |         |         | UBSan <sup>†</sup><br>UBSan <sup>†</sup> | UBSan <sup>†</sup> | CE                |
| ↪ ...union            | ✗         | ✓      |        |                                         |         |         |                                                                                    |                                         | ✗<br>P3850 | ✓      |        | UBSan <sup>†</sup> |         |         | UBSan <sup>†</sup><br>UBSan <sup>†</sup> | UBSan <sup>†</sup> | 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 every class involved has a vtable, even a non-polymorphic one like the example's

D. UBSan's `vptr` check validates that vtable pointer, so it reports only when the storage happens not to hold a valid one:

- *.past.end*: GCC's `vptr` check reports the garbage past the array. Clang's `vptr` runtime instead crashes inside `__dynamic_cast` while inspecting it, which is not a report, so the Clang `vptr` cells are empty. AddressSanitizer reports the read past a stack array as a stack-buffer-overflow on both compilers.
- *.lifetime*: both compilers' `vptr` checks report zeroed storage that never held an object, but not an object destroyed in place, whose vtable pointer still looks valid.
- *.union*: Clang's `vptr` check reports the inactive member (its vtable pointer overlaps the active `int`); GCC folds the conversion for a union member of known type and checks nothing.

There is no P3850 native check.

Constant evaluation rejects *.past.end* on both compilers. GCC also rejects *.lifetime* (storage allocated, never constructed), but accepts *.union* (not yet reported upstream). Stock Clang accepts both *.lifetime* and *.union* (not yet reported upstream); our P3850 Clang branch rejects them.

**[conv.member.missing.member]**

**summary:** converting a base pointer-to-member to a derived one that could not actually contain that member.

**runtime-checkable:** Yes

**locally-checkable:** No

**checking:** Track which type the pointer to member originated from.

**replacement:** None

**implementations:** Clang’s constant evaluation rejects the conversion itself; GCC’s does not. No run-time tool detects it.

**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:

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

<sup>40</sup> 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.3.11

[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 | ✗         | ✓      |        |               |         |         |                  |                  | ✓         | ✓      |        |               |         |         |                  |                  | CE                |

## Implementation Experience

### Existing Practice

No sanitizer check for member-pointer-conversion validity exists on either compiler. GCC’s object-size check can flag a later *application* of the converted pointer that reads outside the object (“load of address ... with insufficient space for an object of type ‘int’”), but that access is a different undefined behavior (expr.mptr.oper.not.contain.member); the conversion itself is pure offset arithmetic and nothing reports it. The same holds for a pointer to member function.

Clang’s constant evaluator rejects the conversion itself. GCC’s accepts it, and even accepts applying the result when the target class happens to have a same-typed member at that offset; it only rejects

an application that reads outside the object, which is a different undefined behavior (related: [GCC PR109243](#), which covers the conversion to a base; this derived-direction form is not yet reported upstream).

**[expr.call.different.type]**

**summary:** calling a function through a pointer/reference type incompatible with its actual definition.

**runtime-checkable:** Yes

**locally-checkable:** No

**checking:** Track type information of function based on address

**replacement:** None

**implementations:** Caught by Clang’s UBSan *function/undefined* check (integrated on the P3850 Clang branch: *quick-enforce*, *noexcept-enforce*, and *noexcept-observe*); GCC has no equivalent; cannot arise during constant evaluation (requires `reinterpret_cast`).

**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  $T_{\text{call}}$  is *call-compatible* with a function type  $T_{\text{func}}$  if  $T_{\text{call}}$  is the same type as  $T_{\text{func}}$  or if the type “pointer to  $T_{\text{func}}$ ” can be converted to type “pointer to  $T_{\text{call}}$ ” via a function pointer conversion (7.3.14[conv.fctptr]). Calling a function through an expression whose function type is not call-compatible with the type of the called function’s definition results in undefined behavior (F.3.12[ub: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.3.12****[ub:expr.call.different.type]****Specified in:** 7.6.1.3[ub: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<br>UBSan |         |         | UBSan<br>UBSan   | UBSan<br>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. On the P3850 Clang branch that check can be routed to the contract-violation handler (`-fsanitize-semantic=function:...`, the nonthrowing semantics only). GCC has no `-fsanitize=function`.

This category cannot arise during constant evaluation: forming the mistyped function pointer requires 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:** .polymorphic caught heuristically by UBSan's `vptra` check on GCC (integrated: *quick-enforce*, *noexcept-enforce*, and *noexcept-observe*), and on Clang except through a reference variable or parameter; .nonpolymorphic not caught.

**Description**

In a member access `E1.E2` naming a non-static member (data member or member function), `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
```

- `expr.ref.member.not.similar.polymorphic` – the object's type is polymorphic, so its vtable pointer identifies its dynamic type at run time
- `expr.ref.member.not.similar.nonpolymorphic` – the object's type is not polymorphic, so there is no runtime type information to check against

**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.3.13[ubx:expr.ref.member.not.similar]).

[Example:

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

— end example]

**UB Annex Entry****F.3.13**

[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 | —         | ✓      |        |                    |         |         |                    |                    | —         | ✓      |        |               |         |         |                  |                  |                   |
| ↪ ...polymorphic            | —         | ✓      |        | UBSan <sup>†</sup> |         |         | UBSan <sup>†</sup> | UBSan <sup>†</sup> | —         | ✓      |        |               |         |         |                  |                  | CE                |
| ↪ ...nonpolymorphic         | —         | ✓      |        |                    |         |         | UBSan <sup>†</sup> | UBSan <sup>†</sup> | —         | ✓      |        |               |         |         |                  |                  | CE                |

**Implementation Experience***Existing Practice*

For the *polymorphic* subset, UBSan’s `vp` check catches the access on both compilers (“member access within address ... which does not point to an object of type ‘B’”, or “member call on ...” for a member function). The detection is heuristic: neither compiler checks a member of a polymorphic base class shared by both types (not yet reported upstream). Clang also does not check a data member read through a reference variable or parameter bound to the reinterpreted glvalue (`B& b = reinterpret_cast<B&>(a); b.j`, the shape used in the conformance example), which GCC does (not yet reported upstream), so Clang’s cells for this subset are empty. On both P3850 branches the `vp` reports, “member access within” and “member call on” included, can be routed to the contract-violation handler (`-fsanitize-semantic=vp:...`); there is no native check. The *nonpolymorphic* subset has no vtable, so no stock sanitizer catches it. (A type pun across types of differing alignment would instead trip UBSan’s unrelated `alignment` check.) ASan is silent, since the underlying object is live.

No `constexpr` trigger exists (hence the ---): every way to obtain such a glvalue (`reinterpret_cast`, or a cast from `void*` to a non-similar type) is already not a core constant expression. GCC accepts

a reference `reinterpret_cast` in constant evaluation (not yet reported upstream), which lets it reach this undefined behavior; it then names the mismatch for a data-member access but accepts a member-function call.

**[expr.dynamic.cast.pointer.lifetime]**

**summary:** `dynamic_cast` on a non-null pointer to the wrong type or outside any object's lifetime.

**runtime-checkable:** Yes

**locally-checkable:** No

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

**replacement:** None

**implementations:** Caught heuristically by Clang's UBSan `vptr` check; rejected during constant evaluation on Clang.

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

A null operand is well-defined (the result is null). The same cast on a pointer to an object outside its lifetime is also `lifetime.outside.pointer.dynamic.cast`.

**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.ctor`]), the behavior is undefined (F.3.14[`ubx:expr.dynamic.cast.pointer.lifetime`]). If `v` is a glvalue of type `U` and `v` does not refer to an object whose type is similar to `U` and that is within its lifetime or within its period of construction or destruction, the behavior is undefined (F.3.15[`ubx:expr.dynamic.cast.glvalue.lifetime`]).

**UB Annex Entry****F.3.14****[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(); };
struct D : B {};
void f() {
 B* bp = new B;
 bp->~B();
 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   |                   |
| expr.dynamic.cast.pointer.lifetime | ✗         | ✓      |        |               |         |         |                  |                  | ✓         | ✓      |        | UBSan <sup>†</sup> |         |         | UBSan <sup>†</sup><br>UBSan <sup>†</sup> | UBSan <sup>†</sup> | CE                |

## Implementation Experience

### Existing Practice

There is no run-time check on GCC. Clang’s UBSan `vp`ptr check (part of `undefined`) reports a `dynamic_cast` whose operand does not point to an object of the operand’s static type, such as the past-the-end pointer above. The check is heuristic: it inspects whatever vtable pointer is found in the storage, so it misses an object destroyed in place, whose vtable pointer still designates a base class.

Constant evaluation rejects a past-the-end, destroyed, dangling, or inactive-union operand on Clang (“dynamic\_cast of dereferenced one-past-the-end pointer”). GCC’s rejects only the dangling and inactive-union cases and accepts a past-the-end or destroyed operand (not yet reported upstream).

**[expr.dynamic.cast.glvalue.lifetime]**

**summary:** `dynamic_cast<T&>` on an object of the wrong type or outside its lifetime.  
**runtime-checkable:** Yes  
**locally-checkable:** No  
**checking:** Track whether storage is associated with an object of correct type within its lifetime or an object currently being constructed or destroyed  
**replacement:** None  
**implementations:** Rejected during constant evaluation on both compilers.

**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(); };
struct D : B {};
void f() {
 D d;
 B& b = d;
 d.~D();
 D& r = dynamic_cast<D&>(b); // undefined behavior: b refers to an object
 // outside its lifetime
 new (&d) D;
}
```

The same cast on a glvalue that is not within its lifetime is also `lifetime.outside.glvalue.dynamic.cast`.

**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.ctor`]), the behavior is undefined (F.3.14[`ubx:expr.dynamic.cast.pointer.lifetime`]). If `v` is a glvalue of type `U` and `v` does not refer to an object whose type is similar to `U` and that is within its lifetime or within its period of construction or destruction, the behavior is undefined (F.3.15[`ubx:expr.dynamic.cast.glvalue.lifetime`]).

**UB Annex Entry****F.3.15**

[`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(); };
struct D : B {};
void f() {
 B* bp = new B;
 B& br = *bp;
 bp->~B();
 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 |
| expr.dynamic.cast.glvalue.lifetime | ✓         | ✓      |        |               |         |         |                  |                  | ✓         | ✓      |        |               |         |         | UBSan <sup>†</sup> | UBSan <sup>†</sup> | CE               |

## Implementation Experience

### Existing Practice

No run-time check catches the lifetime case on either compiler: neither sanitizer reports a `dynamic_cast` on a destroyed object, whose vtable pointer still designates a base class. For the wrong-type case (for example, a `reinterpret_cast<B&>` of a live object of an unrelated polymorphic class), Clang’s `-fsanitize=vptr` heuristically reports the operand (“dynamic operation on address ... which does not point to an object of type ‘B’”; *noexcept-observe* by default, *noexcept-enforce* with `-fno-sanitize-recover=vptr`, and no trap mode); GCC’s reports nothing.

Constant evaluation rejects a destroyed, dangling, or inactive-union operand on both compilers (Clang: “dynamic\_cast of object outside its lifetime”; GCC: “accessing d outside its lifetime”). For an object destroyed in place in dynamically allocated storage, stock GCC instead crashes (an internal compiler error; not yet reported upstream, related [GCC PR107529](#)); the P3850 GCC branch rejects it.

**[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 `vptra` check (Clang, and the P3850 GCC branch; integrated: *quick-enforce*, *noexcept-enforce*, and *noexcept-observe*) and constant evaluation (both); nonpolymorphic subset caught by constant evaluation (both).

**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 B object is not a base class subobject of a D (for instance it belongs to a sibling class, or is a complete B), 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

**Wording**

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

- <sup>2</sup> An lvalue of type “*cv1* B”, where B is a class type, can be cast to type “reference to *cv2* D”, where D is a complete class derived (11.7[class.derived]) from B, if *cv2* is the same cv-qualification as, or greater cv-qualification than, *cv1*. If B is a virtual base class of D or a base class of a virtual base class of D, or if no valid standard conversion from “pointer to D” to “pointer to B” exists (7.3.12[conv.ptr]), the program is ill-formed. An xvalue of type “*cv1* B” can be cast to type “rvalue reference to *cv2* D” with the same constraints as for an lvalue of type “*cv1* B”. If the object of type “*cv1* B” is actually a base class subobject of an object of type D, the result refers to the enclosing object of type D. Otherwise, the

behavior is undefined (F.3.16[ubx:expr.static.cast.base.class]).

[Example:

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

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

— end example]
```

## UB Annex Entry

### F.3.16

[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            | ✓         | ✓      |        | UBSan         |         |         | UBSan            | UBSan            | CE                |
| ↪ ...nonpolymorphic         | ✓         | ✓      |        |               |         |         |                  |                  | ✓         | ✓      |        |               |         |         |                  |                  | CE                |

## Implementation Experience

### Existing Practice

For the *polymorphic* objects, UBSan’s `vptr` check applies. Clang catches the bad reference cast. Stock GCC does not instrument the *reference* form of a `static_cast` downcast at all – its `vptr` coverage of

these casts is limited to the pointer form ([GCC PR116102](#)); the P3850 GCC branch fixes this, instrumenting the reference form after the conversion as it does the pointer form. On both P3850 branches every `vptr` report can be routed to the contract-violation handler (`-fsanitize-semantic=vptr:...`). The *nonpolymorphic* subset (the annex’s own example) has no vtable, so `vptr` is inapplicable and no stock runtime tool catches it.

Clang’s `vptr` check also fires on a *defined* downcast: a `static_cast` from `B&` to `F&` inside the constructor of an intermediate base `E` while an `F` is being constructed reports “does not point to an object of type ‘F’” ([llvm#59060](#); GCC has the same false positive for the pointer form, see `[ub:expr.static.cast.downcast.wrong.derived.type]`). Under an enforcing semantic this terminates a valid program.

Both subsets are rejected during constant evaluation by both compilers: Clang reports “cannot cast object of dynamic type ‘D1’ to type ‘D2’”, GCC “‘B’ operand (of dynamic type ‘D1’) is not a base class subobject of a ‘D2’ object”. GCC only acquired this recently – its constant evaluator used to accept both subsets silently, and gained the cross-check of a `static_cast`’s target type against the operand’s real constructed type in GCC 17. Both constant evaluators also reject the `xvalue` form (`B&&` to `D2&&`) and a `B` that is a complete object or a data member.

**[expr.static.cast.downcast.wrong.derived.type]**

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

**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

A null pointer converts to a null pointer and is not undefined behavior.

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

## UB Annex Entry

**F.3.19** [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

The example is a `static_cast<D2*>` applied to a pointer that actually points to a `D1`, and, for the *polymorphic* subset, is the archetypal `vp_ptr` case: UBSan’s `vp_ptr` check catches it on both compilers (“downcast of address ... which does not point to an object of type ‘D2’”). Stock GCC instruments only this pointer form (the P3850 GCC branch instruments the reference form too); see [ub:expr.static.cast.base.class] for the reference form.

On both compilers `vp_ptr` also fires on a *defined* downcast: a `static_cast` from `B*` to `F*` inside the constructor of an intermediate base `E` while an `F` is being constructed reports “does not point to an object of type ‘F’” (llvm#59060; for GCC not yet reported upstream). Under an enforcing semantic this terminates a valid program.

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

Both subsets are rejected during constant evaluation by both compilers. GCC only acquired this recently – its constant evaluator used to accept both subsets silently, and gained the cross-check of a `static_cast`'s target type against the operand's real constructed type in GCC 17.

**[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 D can be `static_cast` to a pointer to member of a base class B – but only if B contains the original member or is a base class of the class that contains it. Casting to a base unrelated to the member’s class (for instance a sibling base of D) is undefined behavior; this applies to pointers to member functions as well as data members.

```
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.3.20[ubx:expr.static.cast.does.not.contain.original.member]).

[Note: Although class B need not contain the original member, the dynamic type of the object

with which indirection through the pointer to member is performed must contain the original member; see 7.6.4[expr.mptr.oper]. — end note]

UB Annex Entry

**F.3.20** [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 this, for data and member-function pointers alike ([GCC PR109243](#)); Clang rejects it at the cast (“subexpression not valid in a constant expression”). Nothing catches it at run time: a pointer to member does not record the class of its original member.

**[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` yields an unspecified choice among the pointer values that would convert to that integer, restricted to those that give the rest of the program defined behavior. If no such choice exists, the conversion itself is undefined behavior – a rule about pointer *provenance*, not merely the numeric address.

```
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
 q - 1; // undefined behavior if q is &y
 *q; // undefined behavior if q is (&x)+1
 }
}
```

**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.3.21[ubx:expr.reinterpret.cast.invalid.pointer.value]).

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

Otherwise, the result is implementation-defined.

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

**UB Annex Entry**

**F.3.21****[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` when at least one pointer value converts to that integral or enumeration value, but no such pointer value would give the program defined behavior, 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 their lifetime.

```
#include <stdint>
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 run time. The example on Compiler Explorer uses two adjacent members of a struct, so that the past-the-end pointer and the pointer to the next object always convert to the same integer. No `constexpr` trigger exists (hence the ---): an integer-to-pointer `reinterpret_cast` is never a core constant expression.

**[expr.unary.dereference]**

**summary:** dereferencing a pointer that is null or otherwise does not point to an object or function.

**runtime-checkable:** Yes

**locally-checkable:** Only for the null pointer case

**checking:** Track whether storage is associated with an object of correct type within its lifetime; track whether the address is associated with a function; insert a null pointer check

**replacement:** None

**implementations:** `.nullptr.access` caught by a native P3850 check (all available semantics on Clang; four non-throwing semantics on GCC), UBSan's `null` check, and constant evaluation, both compilers; `.invalid.access` caught heuristically by AddressSanitizer and by constant evaluation, both compilers; `.nullptr.indirection` caught in every form by the native P3850 check (same semantics) and only partially by UBSan and constant evaluation; `.invalid.indirection` not caught, and the P3850 prototype checks the `.invalid` sub-cases only for an array of statically known bound (see below).

**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 non-null pointer that does not point to an object (for example, a past-the-end pointer) falls under it too. A pointer to an object whose lifetime has ended still points to that object, so indirection through it is not covered here, and indirection through a dangling pointer (one whose storage has been released) is [ub:basic.compound.invalid.pointer] first.

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

- `expr.unary.dereference.nullptr.access` – the pointer is null and the result is read or written (`*p`, `p->m`)
- `expr.unary.dereference.nullptr.indirection` – the pointer is null and the result is not accessed (`int& r = *p;`, `&*p`, `&p->m`, `p->sf()` for a static member function `sf`, `*fp` for a function pointer `fp`)
- `expr.unary.dereference.invalid.access` – the pointer is non-null but does not point to an object (for example, a past-the-end pointer) and the result is read or written
- `expr.unary.dereference.invalid.indirection` – as `.invalid.access`, but the result is not accessed (`int& r = *p;`, `&*p`, `&a[N]` for an array `a` of `N` elements)

These are exhaustive: a pointer either is null or is not, and the result of the indirection either is accessed or is not.

## 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.3.22[ubx:expr.unary.dereference]).

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

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

## UB Annex Entry

### F.3.22

[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.access      | ✓         | ✓      | P3850  | P3850<br>UBSan    |         |         | P3850<br>UBSan    | P3850<br>UBSan    | ✓         | ✓      | P3850  | P3850<br>UBSan    | P3850<br>UBSan | P3850   | P3850<br>UBSan    | P3850<br>UBSan    | CE                |
| ↳ ...nullptr.indirection | ✗         | ✓      | P3850  | P3850             |         |         | P3850             | P3850             | ✗         | ✓      | P3850  | P3850             | P3850          | P3850   | P3850             | P3850             | CE                |
| ↳ ...invalid.access      | ✓         | ✓      |        | ASan <sup>†</sup> |         |         | ASan <sup>†</sup> | ASan <sup>†</sup> | ✓         | ✓      |        | ASan <sup>†</sup> |                |         | ASan <sup>†</sup> | ASan <sup>†</sup> | CE                |
| ↳ ...invalid.indirection | ✗         | ✓      |        |                   |         |         |                   |                   | ✗         | ✓      |        |                   |                |         |                   |                   | CE                |

---

## Implementation Experience

---

### *Existing Practice*

For `.nullptr.access`, UBSan's null check (part of the `undefined` group) instruments the dereference on both compilers, and constant evaluation rejects it on both.

For `.nullptr.indirection` the coverage of existing practice depends on the form, so the row reflects the least-covered one (a static member function called through a null pointer, which no stock tool catches). UBSan's null check catches binding a reference (`int& r = *p;`) and forming a member address (`&p->m`) on both compilers, and binding a reference to a function through a null function pointer on GCC only; it does not instrument `&*p`, a discarded `*p`, a static member named through `p->`, or a call `(*fp)()`. In constant evaluation both compilers reject binding a reference through `*p`, forming a member address `&p->m`, and a call `(*fp)()`; Clang also rejects `*p`, `&*p`, and `(*p).sf()`; GCC also rejects `F& r = *fp;` (a reference to a function) and `int& r = p[0];`, both of which Clang accepts; neither rejects `p->sf()`, `&*fp`, or a discarded `*fp`. The GCC misses are reported upstream as [GCC PR115592](#), [GCC PR126360](#), and [GCC PR113047](#); the Clang misses are not yet reported upstream.

For `.invalid.access`, AddressSanitizer (`-fsanitize=address`) catches a read or write through a past-the-end pointer when it lands in a redzone (heap/stack/global-buffer-overflow), reporting then terminating (*noexcept-enforce*) by default, and continuing (*noexcept-observe*) under `-fsanitize-recover=address` together with ASan's `halt_on_error=0` option (supplied in-source by defining `__asan_default_options` to return it, or through `ASAN_OPTIONS`). Detection is heuristic: an access far enough past the end can land in another live allocation, and a pointer built from an arbitrary integer typically dereferences to a plain segmentation fault. Constant evaluation rejects such a read on both compilers.

For `.invalid.indirection` nothing catches the representative `&*p` (with `p` past the end): no memory is accessed, so no sanitizer has anything to instrument, and both constant evaluators accept it – Clang deliberately ([llvm#65119](#), closed as not planned). Clang does reject `int& r = *p;`, but GCC accepts that form as well as `&*p` (not yet reported upstream).

### *P3850 Implementation*

To enable the null-pointer check in either compiler, `-fcontracts-p3100` must be specified and the group `ub:expr.unary.dereference.nullptr` given a semantic other than *assume*. Each sub-case is also a group of its own: `ub:expr.unary.dereference.nullptr.access` is reported where the result is read or written (including a member read or written through `p->m` or `p->*pm`, and a virtual function call or a `dynamic_cast` to a reference, which read the object's vtable pointer), and `ub:expr.unary.dereference.nullptr.indirection` where it is not (binding a reference, `&*p`, `&p->m`, a discarded `*p`, a non-virtual member function call, a static member, enumerator or destructor named through `p->`, and any indirection through a null pointer to function, the call `(*fp)()` included).

The check covers only a *null*-pointer dereference (`*p` with `p` null), not every invalid unary dereference this instance names. Within that, both compilers check every indirection, accessed or not, the row's representative static member call included. A member of a base class used through a null pointer is checked on the object before its conversion to the base, so a base at a nonzero offset or a virtual

base is caught (a virtual base's member under `.access` in every form, since reaching it reads the object), as is a downcast of `*p` to a reference; and one null dereference is reported once: a read or write through a reference bound to `*p` is not checked again after the binding, and no check is made through `this` inside a member function called through a null pointer. An element at a constant offset (`p[1].m`, `*(p + 1)`) is checked on `p` itself. The two compilers differ on one form, both conforming: for a subscript or pointer arithmetic with a non-constant offset used as an object (`p[i].m`, `*(p + i)`), Clang checks `p` while GCC checks the element's address `p + i`, so GCC does not report it with `p` null and `i` nonzero (which is also null pointer arithmetic). 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.

For a non-null pointer, both compilers check the indirection of a past-the-end pointer into an array of statically known bound `N` – `&a[N]`, `*(a + N)`, `&*(a + N)`, a reference bound to `*(a + N)`, `(a + N)->m`, `(a + N)->f()`, an empty class copied from or stored to `a[N]`, and nested arithmetic such as `*(a + i + j)` – under the group `ub:expr.unary.dereference.invalid.known`, in all available semantics; the substitute is the array's first element. This covers only part of both `.invalid` sub-cases: a pointer whose bound is not statically known, such as the heap pointers of this row's examples, is not checked, so the `.invalid` rows show no P3850 cell.

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

**Description**

A non-allocating placement-`new` form (such as `operator new(size_t, void*)`) cannot fail; it returns its pointer argument. If that argument is null, the allocation function returns null and the placement new-expression has undefined behavior – there is no allocation-failure protocol for a form that cannot fail. These forms are not replaceable, so a null argument is the only way to reach this undefined behavior.

```
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.3.23[ubx:expr.new.non.allocating.null]). Otherwise, if the allocation function returns null, initialization shall not be done, the deallocation function shall not be called, and the value of the *new-expression* shall be null.

**UB Annex Entry****F.3.23****[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 | ✓         | ✓      | P3850  | P3850<br>UBSan | P3850   | P3850   | P3850<br>UBSan   | P3850<br>UBSan   | ✓         | ✓      | P3850  | P3850<br>UBSan | P3850   | P3850   | P3850<br>UBSan   | P3850<br>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 (GCC: “destroying `*0` from outside current evaluation”; Clang: “construction of dereferenced null pointer is not allowed in a constant expression”). GCC accepts a null placement `new` of an empty class in constant evaluation, which no initialization reaches through the pointer (not yet reported upstream).

### P3850 Implementation

To enable this check in either compiler, `-fcontracts-p3100` must be specified and the group

ub:`expr.new.non.allocating.null` given a semantic other than *assume*. Both compilers check the pointer returned by the standard non-allocating placement operator `new(size_t, void*)` before the object is created; a class's own placement operator `new` is not checked. All seven semantics are supported. There is no defined substitute: execution continuing past a violation creates the object at the null address.

**GCC:** The check is built with the new-expression (`build_new_1, cp/init.cc`). Its condition is a constant expression, so with the check configured a null placement `new` in constant evaluation is reported as undefined behavior, the empty class included.

**Clang:** `EmitImplicitPlacementNewGuards` (`CGContracts.cpp`) checks the pointer before initialization, in place of the null assertion of the new-expression's type check.

**[expr.delete.mismatch]**

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

**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:

- <sup>2</sup> In a single-object delete expression, the value of the operand of `delete` may be a null pointer value, a pointer value that resulted from a previous non-array *new-expression*, or a pointer to a base class subobject of an object created by such a *new-expression*. If not, the behavior is undefined (F.3.24[ubx:expr.delete.mismatch]). In an array delete expression, the value of the operand of `delete` may be a null pointer value or a pointer value that resulted from a previous array *new-expression* whose allocation function was not a non-allocating form (17.6.3.4[new.delete.placement]).<sup>56</sup> If not, the behavior is undefined (F.3.25[ubx:expr.delete.array.mismatch]).

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

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

<sup>56</sup> 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.mismatch]****Specified in:** 7.6.2.9[ubx:expr.delete.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.mismatch | ✓         | ✓      |        | ASan          |         |         | ASan<br>ASan     | ASan             | ✓         | ✓      |        | ASan          |         |         | ASan<br>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” on GCC, “non-array delete used to delete pointer to array object” on Clang), 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.array.mismatch]**

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

**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.3.24[ubx:expr.delete.mismatch]). In an array delete expression, the value of the operand of `delete` may be a null pointer value or a pointer value that resulted from a previous array *new-expression* whose allocation function was not a non-allocating form (17.6.3.4[new.delete.placement]).<sup>56</sup> If not, the behavior is undefined (F.3.25[ubx:expr.delete.array.mismatch]).

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

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

<sup>56</sup> 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.3.25**

[ub:expr.delete.array.mismatch]

**Specified in:** 7.6.2.9[ubx:expr.delete.array.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.array.mismatch | ✓         | ✓      |        | ASan          |         |         | ASan<br>ASan     | ASan             | ✓         | ✓      |        | ASan          |         |         | ASan<br>ASan     | ASan             | CE                |

**Implementation Experience***Existing Practice*

When the `new` and the `delete[]` are visible together, both compilers already diagnose 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 report is *noexcept-enforce* by default, with the stock *noexcept-observe* continuing case needing the `ASAN_OPTIONS=halt_on_error=0` environment variable (alongside `-fsanitize-recover=address`), and no *quick-enforce*.

The mismatch is also rejected during constant evaluation on both compilers (GCC: “array deallocation of object allocated with non-array allocation”; Clang: “array delete used to delete pointer to non-array object”).

**[expr.delete.dynamic.type.differ]**

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

**runtime-checkable:** Yes

**locally-checkable:** No

**checking:** Track dynamic type of non-polymorphic objects

**replacement:** None

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

**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.3.26[ubx:expr.delete.dynamic.type.differ]). In an array delete expression, if the dynamic type of the object to be deleted is not similar to its static type, the behavior is undefined (F.3.27[ubx:expr.delete.dynamic.array.dynamic.type.differ]).

**UB Annex Entry****F.3.26****[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 | ✗<br>P3850 | ✓      |        | ASan <sup>†</sup> |         |         | ASan <sup>†</sup><br>ASan <sup>†</sup> | ASan <sup>†</sup> | ✓         | ✓      |        | ASan <sup>†</sup> |         |         | ASan <sup>†</sup><br>ASan <sup>†</sup> | ASan <sup>†</sup> | CE                |

## Implementation Experience

### Existing Practice

AddressSanitizer’s `new-delete-type-mismatch` check reports this issue on both compilers whenever the static and dynamic types differ in size: it compares the sized-deallocation size with the allocation size, so a same-size derived class goes unreported and the detection is heuristic. By default it prints a report and terminates (*noexcept-enforce*).

Clang’s constant evaluator rejects the deletion. Stock GCC’s accepts it ([GCC PR97647](#)); the the P3850 GCC branch rejects it (“delete of an object of dynamic type D through a pointer to B, whose destructor is not virtual”).

**[expr.delete.dynamic.array.dynamic.type.differ]**

**summary:** delete[] whose static type differs from the array's dynamic type.  
**runtime-checkable:** Yes  
**locally-checkable:** No  
**checking:** Track dynamic type of non-polymorphic objects  
**replacement:** None  
**implementations:** Caught heuristically by ASan's `new-delete-type-mismatch` on both compilers (only when the element sizes differ; stock, and integrated on the P3850 branches); rejected during constant evaluation on Clang.

**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.3.26[ubx:expr.delete.dynamic.type.differ]). In an array delete expression, if the dynamic type of the object to be deleted is not similar to its static type, the behavior is undefined (F.3.27[ubx:expr.delete.dynamic.array.dynamic.type.differ]).

**UB Annex Entry**

**F.3.27** [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.<br>dynamic.type.differ | ✗         | ✓      |        | ASan <sup>†</sup> |         |         | ASan <sup>†</sup><br>ASan <sup>†</sup> | ASan <sup>†</sup> | ✓         | ✓      |        | ASan <sup>†</sup> |         |         | ASan <sup>†</sup><br>ASan <sup>†</sup> | ASan <sup>†</sup> | CE                |

## Implementation Experience

### Existing Practice

AddressSanitizer’s `new-delete-type-mismatch` check catches this problem on both compilers only when the static and dynamic element types differ in size: it compares the size passed to the sized array deallocation function with the allocation size, so a same-size base/derived pair goes unreported; the detection is therefore heuristic.

Clang’s constant evaluator rejects the deletion (“delete of pointer to subobject”). GCC’s accepts it: it rejects only incidentally, when a larger derived type makes the wrong-stride destruction of a later element leave the allocation, which a one-element array (as in the conformance example) avoids (not yet reported upstream). 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; stock, and integrated on the P3850 branches) when the access lands outside the object; rejected during constant evaluation on Clang.

**Description**

In a pointer-to-member access `E1.*E2`, the object denoted by `E1` must have a type similar to that of `E1`, and its most derived object must contain the member `E2` designates; otherwise the access is undefined behavior. The second case arises without any prior undefined behavior by applying a member pointer converted to a base class to an object that is only a base:

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

void f() {
 int B::*pb = static_cast<int B::*>(&D::x); // OK, B is a base of D
 B b;
 b.*pb = 1; // undefined behavior: the most derived object (B) does not
 // contain D::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.3.28[ubx:expr.mptr.oper.not.contain.member]). The expression `E1` is sequenced before the expression `E2`.

**UB Annex Entry****F.3.28****[ub:expr.mptr.oper.not.contain.member]****Specified in:** 7.6.4[ubx:expr.mptr.oper.not.contain.member]

In a *pm-expression* of the form `E1.*E2`, if `E1` refers to an object whose type is not similar to the type of `E1`, or whose most derived object (6.8.2[intro.object]) 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() {
 int B::*pb = static_cast<int B::*>(&D::x);
 B b;
 b.*pb = 1; // undefined behavior, 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 <sup>†</sup> |         |         | ASan <sup>†</sup><br>ASan <sup>†</sup> | ASan <sup>†</sup> | ✓         | ✓      |        | ASan <sup>†</sup> |         |         | ASan <sup>†</sup><br>ASan <sup>†</sup> | ASan <sup>†</sup> | CE                |

## Implementation Experience

### Existing Practice

For a member-pointer conversion as above, Clang’s constant evaluator rejects the access (“subexpression not valid in a constant expression”). GCC’s rejects it only when a read or write lands outside the object, and accepts `&(b.*pb)` or a member function call that touches no data (GCC PR121250).

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:** a pointer-to-member operator (`.` or `->`) whose second operand is a null member pointer.

**runtime-checkable:** Yes

**locally-checkable:** Yes

**checking:** Insert null pointer check

**replacement:** None

**implementations:** Rejected during constant evaluation on both compilers for a member function, and on Clang for a data member; caught by the P3850 prototype’s native check (both subgroups, both compilers, all semantics).

**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
}
```

Despite the category name, the rule applies to any null member pointer, including a pointer to data member:

- `expr.mptr.oper.member.func.null.function` – null member function pointer
- `expr.mptr.oper.member.func.null.data` – null data member pointer

**Wording**

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

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

[Example:

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

In a `.*` expression whose object expression is an rvalue, the program is ill-formed if the second operand is a pointer to member function whose *ref-qualifier* is `&`, unless its *cv-qualifier-seq* is `const`. In a `.*` expression whose object expression is an lvalue, the program is ill-formed if the second operand is a pointer to member function whose *ref-qualifier* is `&&`. The result of a `.*` expression whose second operand is a pointer to a

data member is an lvalue if the first operand is an lvalue and an xvalue otherwise. The result of a `.*` expression whose second operand is a pointer to a member function is a prvalue. If the second operand is the null member pointer value (7.3.13[conv.mem]), the behavior is undefined (F.3.29[ubx:expr.mptr.oper.member.func.null]).

## UB Annex Entry

### F.3.29

[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 | ✗         | ✓      | P3850  | P3850         | P3850   | P3850   | P3850            | P3850            | ✓         | ✓      | P3850  | P3850         | P3850   | P3850   | P3850            | P3850            |                   |
| ↪ ...function                   | ✓         | ✓      | P3850  | P3850         | P3850   | P3850   | P3850            | P3850            | ✓         | ✓      | P3850  | P3850         | P3850   | P3850   | P3850            | P3850            | CE                |
| ↪ ...data                       | ✗         | ✓      | P3850  | P3850         | P3850   | P3850   | P3850            | P3850            | ✓         | ✓      | P3850  | P3850         | P3850   | P3850   | P3850            | P3850            | CE                |

## Implementation Experience

### Existing Practice

No run-time check exists on either compiler; a call through a null pointer to member function typically jumps to address zero.

Calling through a null pointer to member function is rejected during constant evaluation on both compilers (GCC: “pm is not a constant expression”, or “does not designate a `constexpr` function” for `->*` and virtual targets; Clang: “subexpression not valid in a constant expression”). For a null pointer to data member, Clang rejects every use; GCC rejects a read or write only incidentally (the

access lands outside the object) and accepts `&(s.*pmd)` (not yet reported upstream; related [GCC PR126360](#)).

### *P3850 Implementation*

To enable this check in either compiler, `-fcontracts-p3100` must be specified and the group `ub:expr.mptr.oper.member.func.null` given a semantic other than *assume*; the subgroups are configurable on their own, as `ub:expr.mptr.oper.member.func.null.data` and `ub:expr.mptr.oper.member.func.null.function`. Both compilers test the member pointer at the `.*` or `->*` operator (a null pointer to data member is the offset `-1`; a null pointer to member function has a zero function pointer and no virtual bit), so `&(s.*pmd)` is caught too. A constant member pointer is not checked. All seven semantics are supported; there is no defined substitute, so execution continuing past a violation uses the null member pointer as written (for a call, typically a jump to address zero).

**GCC:** The guards are built with the operator (`build_m_component_ref`, `cp/typeck2.cc`) and with the call (`get_member_function_from_ptrfunc`, `cp/typeck.cc`); they are evaluated only at run time, so constant evaluation is unchanged.

**Clang:** `EmitImplicitNullMemberPointerGuard` (`CGContracts.cpp`) emits the guard, using the C++ ABI's own null test of a member pointer, from the code generation of `.*` and `->*` and of a call through a pointer to member function.

**[expr.add.out.of.bounds]**

|                           |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|---------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>summary:</b>           | forming a pointer outside an array's elements and its one-past-the-end position.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>runtime-checkable:</b> | Yes                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>locally-checkable:</b> | Only if the array bound is statically known                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>checking:</b>          | Track pointer provenance, insert bounds check                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>replacement:</b>       | None                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>implementations:</b>   | <code>.known.subscript</code> checked by the P3850 bounds check (all semantics, both compilers) and UBSan's <code>bounds</code> check, and rejected during constant evaluation on both compilers; <code>.known.arithmetic</code> checked by the P3850 prototype (all semantics, both compilers) and Clang's UBSan <code>array-bounds</code> check, and rejected during constant evaluation on Clang only; <code>.unknown</code> only heuristically by AddressSanitizer, and rejected during constant evaluation on Clang only; <code>.pointer.arithmetic</code> by the P3850 prototype (all semantics, both compilers) and UBSan's <code>pointer-overflow</code> 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.subscript` – the array's extent is known at compile time (a fixed-size array type) and the pointer is formed by a subscript (`a[i]`, `&a[i]`)
- `expr.add.out.of.bounds.known.arithmetic` – the array's extent is known at compile time and the pointer is formed by pointer arithmetic applied directly to the array (`a + i`, `a - i`)
- `expr.add.out.of.bounds.unknown` – the array's extent is not statically known at the arithmetic (for example, a pointer into heap-allocated storage, or `p += i` on a pointer variable)
- `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 three 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. Pointers to non-array objects (complete objects, members, base subobjects) count as one-element arrays and fall under `.known`; adding a nonzero value to a null pointer is also this undefined behavior. The last 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  $\mathbf{x}$  with  $n$  elements (9.3.4.5[dcl.array]),<sup>58</sup> the expressions  $\mathbf{P} + \mathbf{J}$  and  $\mathbf{J} + \mathbf{P}$  (where J has the value  $j$ ) point to the (possibly-hypothetical) array element  $i + j$  of  $\mathbf{x}$  if  $0 \leq i + j \leq n$  and the expression  $\mathbf{P} - \mathbf{J}$  points to the (possibly-hypothetical) array element  $i - j$  of  $\mathbf{x}$  if  $0 \leq i - j \leq n$ .
  - Otherwise, the behavior is undefined (F.3.32[ub: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.3.32**

[ub:expr.add.out.of.bounds]

**Specified in:** 7.6.6[ub: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 |
|-------------------------|-----------|--------|--------|-------------------|---------|---------|----------------------------------------|-------------------|-----------|--------|--------|-------------------|---------|---------|----------------------------------------|-------------------|-------------------|
|                         | contextpr | assume | ignore | quick-enforce     | enforce | observe | noexcept-enforce                       | noexcept-observe  | contextpr | assume | ignore | quick-enforce     | enforce | observe | noexcept-enforce                       | noexcept-observe  |                   |
| expr.add.out.of.bounds  | ✗         | ✓      |        |                   |         |         |                                        |                   | ✓         | ✓      |        |                   |         |         |                                        |                   |                   |
| ↪ ...known.subscript    | ✓         | ✓      | P3850  | P3850<br>UBSan    | P3850   | P3850   | P3850<br>UBSan                         | P3850<br>UBSan    | ✓         | ✓      | P3850  | P3850<br>UBSan    | P3850   | P3850   | P3850<br>UBSan                         | P3850<br>UBSan    | CE                |
| ↪ ...known.arithmetic   | ✗         | ✓      | P3850  | P3850             | P3850   | P3850   | P3850<br>UBSan                         | P3850<br>UBSan    | ✓         | ✓      | P3850  | P3850<br>UBSan    | P3850   | P3850   | P3850<br>UBSan                         | P3850<br>UBSan    | CE                |
| ↪ ...unknown            | ✗         | ✓      |        | ASan <sup>†</sup> |         |         | ASan <sup>†</sup><br>ASan <sup>†</sup> | ASan <sup>†</sup> | ✓         | ✓      |        | ASan <sup>†</sup> |         |         | ASan <sup>†</sup><br>ASan <sup>†</sup> | ASan <sup>†</sup> | CE                |
| ↪ ...pointer.arithmetic | —         | ✓      | P3850  | P3850<br>UBSan    | P3850   | P3850   | P3850<br>UBSan                         | P3850<br>UBSan    | —         | ✓      | P3850  | P3850<br>UBSan    | P3850   | P3850   | P3850<br>UBSan                         | P3850<br>UBSan    | CE                |

**Implementation Experience***Existing Practice*

For the `.known.subscript` subset, 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). On Clang the bounds group also contains the trap-only local-bounds check, so it cannot recover; its `array-bounds` component provides all three. For `.known.arithmetic`, Clang’s `array-bounds` check also instruments pointer arithmetic applied directly to an array, while GCC’s bounds check does not. 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’s report is *noexcept-enforce* by default, with the stock *noexcept-observe* continuing case needing the `ASAN_OPTIONS=halt_on_error=0` environment variable (alongside `-fsanitize-recover=address`), and no *quick-enforce*.

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. That overlap 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.

Clang's constant evaluator rejects the formation of any out-of-bounds pointer. GCC's accepts the formation (and even evaluates  $(a + 4) - 4$ ), rejecting only a subscript of a known-bound array or a read of an out-of-range element, so only `.known.subscript` is caught on GCC ([GCC PR70151](#), [GCC PR11242](#)). 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.subscript`, `.known.arithmetic` and `.pointer.arithmetic` categories can be checked using the integrated frontend checks in the P3850 prototypes. To enable these checks in either compiler, `-fcontracts-p3100` must be specified and the group `ub:expr.add.out.of.bounds` given a semantic other than *assume*; each category is also a group of its own (`ub:expr.add.out.of.bounds.known.subscript`, `ub:expr.add.out.of.bounds.known.arithmetic`, `ub:expr.add.out.of.bounds.pointer.arithmetic`), and `ub:expr.add.out.of.bounds.known` selects the first two.

The subscript guard covers array subscripts (`a[i]`, `&a[i]`). It also fires on a read of `a[N]`, which forms a valid one-past pointer and is a past-the-end dereference rather than this undefined behavior. The arithmetic guard covers `a + i`, `i + a` and `a - i` on an array of known bound `N` decayed to a pointer, which must stay within `[0, N]`, and arithmetic or a subscript applied to such a result (`(a + i) + j`, and `(a + k)[i]` checked as `a[k + i]`); arithmetic on a pointer variable, or through a cast to another element type, is not known-bound and not checked. The `.pointer.arithmetic` guard checks whether the byte offset wraps the address space, for `p + i`, `p - i`, `p += i` and the increment and decrement operators on any pointer.

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), and out-of-range or wrapping arithmetic yields the unmoved pointer (for nested known-bound arithmetic, the array's first element). This is not a proposed replacement – the program still has undefined behavior – but it does avoid out-of-bounds writes.

**GCC:** The GCC prototype supports all seven evaluation semantics. The subscript guard is 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); the arithmetic guards are built by the front end with the pointer addition.

**Clang:** The Clang prototype likewise supports all available semantics, emitting the guards during code generation of the array subscript and of the pointer arithmetic.

**[expr.add.sub.diff.pointers]**

**summary:** subtracting pointers that do not point into the same array (including different rows or members of one object).

**runtime-checkable:** Yes

**locally-checkable:** No

**checking:** Track pointer provenance, verify both operands point into the same array object

**replacement:** None

**implementations:** `.distinct.objects` caught at run time by AddressSanitizer's `pointer-subtract` check (stock and integrated) on both compilers, and rejected during constant evaluation on both; `.same.object` rejected during constant evaluation on Clang only.

**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
}
```

- `expr.add.sub.diff.pointers.distinct.objects` – the two pointers point into different complete objects, as in the example above
- `expr.add.sub.diff.pointers.same.object` – the two pointers point into different subobjects of one complete object, such as two members of a class or elements of different rows of a multidimensional array

**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 that value is not in the range of representable values of type `std::ptrdiff_t`, the behavior result is undefined an unspecified erroneous value and there is an implicit precondition assertion that this does not occur (6.11.2+a[basic.contract.implicit]) (F.3.33[ubx:expr.sub.pointers).

rep])).

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

**UB Annex Entry****F.3.34**

[ub:expr.add.sub.diff.pointers]

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

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

2

[Example 1:

```
#include <stddef>
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 | no-exception-enforce | no-exception-observe | constexpr | assume | ignore | quick-enforce | enforce | observe | no-exception-enforce | no-exception-observe |                   |
| expr.add.sub.diff.pointers | ✗         | ✓      |        |               |         |         |                      |                      | ✓         | ✓      |        |               |         |         |                      |                      |                   |
| ↪ ...distinct.objects      | ✓         | ✓      |        | ASan          |         |         | ASan                 | ASan                 | ✓         | ✓      |        | ASan          |         |         | ASan                 | ASan                 | CE                |
| ↪ ...same.object           | ✗         | ✓      |        |               |         |         |                      |                      | ✓         | ✓      |        |               |         |         |                      |                      | CE                |

**Implementation Experience***Existing Practice*

Constant evaluation rejects `.distinct.objects`: 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). For `.same.object`, GCC’s evaluator accepts the subtraction of pointers to different subobjects of the same complete object – two members of one class, or elements of different rows of a multidimensional array – which Clang rejects (“subtracted pointers are not elements of the same array”) (GCC PR126060). 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. It compares allocations, so it does not report `.same.object`, whose operands lie in one allocation.

**[expr.add.not.similar]**

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

**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, an `int` array walked through an `unsigned int*` obtained by `reinterpret_cast` – the behavior is undefined.

```
int arr[5] = {1, 2, 3, 4, 5};
unsigned int* p = reinterpret_cast<unsigned int*>(arr + 1);
unsigned int k = *p; // OK, value of k is 2
unsigned int* q = p + 1; // undefined behavior: p points to an int, not
 // an unsigned int object
```

The annex's example, a base-class pointer walking an array of a larger derived type, is better read as [ub:expr.add.out.of.bounds]: under the [expr.add] note, a pointer to a base-class subobject behaves as a pointer into a one-element array of the base type, so `s + count` leaves that array.

**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.3.35[ub: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.3.35****[ub:expr.add.not.similar]**

**Specified in:** 7.6.6[ub: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 | —         | ✓      |        |               |         |         |                  |                  | —         | ✓      |        |               |         |         |                  |                  | <a href="#">CE</a> |

Implementation Experience

Existing Practice

No stock runtime sanitizer catches this category: the arithmetic stays within the array’s storage, so `-fsanitize=address` has no redzone to trip, and `-fsanitize=undefined` ships no “wrong element type” check.

This category cannot arise during constant evaluation: a pointer to an array element of a dissimilar type can only be formed with `reinterpret_cast`, or with `static_cast` from `void*`, which constant evaluation rejects unless the types are similar. (GCC’s constant evaluator instead crashes on the `static_cast` form when the `void*` came from `arr + 1`: [GCC PR125199](#).)

**[expr.assign.overlap]**

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

**Description**

A built-in assignment whose stored value is read from an object that overlaps the destination is undefined behavior unless the overlap is exact and the two objects have the same type – for example, assigning one member of a union to a differently-typed member.

```
union U { int i; short s; };
U u;
u.s = 1;
u.i = u.s; // undefined behavior: u.s overlaps u.i inexactly and has a
 // different type
```

The annex's example, which reads the source through a `reinterpret_cast<char*>` of the destination, is weaker: the `char` read accesses the destination object itself, so whether a distinct overlapping object exists is debatable.

**Wording**

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

- 7 If the value being stored in an object is read via another object that overlaps in any way the storage of the first object, then the overlap shall be exact and the two objects shall have the same type, otherwise the behavior is undefined (F.3.37[ub: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.3.37****[ub:expr.assign.overlap]****Specified in:** 7.6.19[ub: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]

Leave Behavior as Undefined

`expr.assign.overlap`

## Implementation Status

(full table in Section 3)

| Group                            | GCC              |               |               |                      |                |                |                         |                         | Clang            |               |               |                      |                |                |                         |                         | compiler explorer |
|----------------------------------|------------------|---------------|---------------|----------------------|----------------|----------------|-------------------------|-------------------------|------------------|---------------|---------------|----------------------|----------------|----------------|-------------------------|-------------------------|-------------------|
|                                  | <i>constexpr</i> | <i>assume</i> | <i>ignore</i> | <i>quick-enforce</i> | <i>enforce</i> | <i>observe</i> | <i>noexcept-enforce</i> | <i>noexcept-observe</i> | <i>constexpr</i> | <i>assume</i> | <i>ignore</i> | <i>quick-enforce</i> | <i>enforce</i> | <i>observe</i> | <i>noexcept-enforce</i> | <i>noexcept-observe</i> |                   |
| <code>expr.assign.overlap</code> | ✗                | ✓             | P3850         | P3850                | P3850          | P3850          | P3850                   | P3850                   | ✗                | ✓             | P3850         | P3850                | P3850          | P3850          | P3850                   | P3850                   | CE                |

## Implementation Experience

### Existing Practice

No tool detects this category, and neither compiler’s constant evaluator diagnoses it: the union form above is accepted in a `static_assert` by both (not yet reported upstream).

### P3850 Implementation

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

The check covers a simple assignment of scalar type whose right operand is, through conversions of the value read, a glvalue: before the store, the address ranges of the two objects are compared, and any overlap between objects of different types, or an inexact one between objects of the same type, is a violation. It does not cover bit-fields, two different variables named directly (which cannot overlap), compound assignment, or a stored value that is computed rather than read. An operand whose address computation has side effects is evaluated once. The comparison is not a constant expression, so constant evaluation is unchanged. There is no substitute: when execution continues past a violation, the assignment is performed as written.

**GCC:** The GCC prototype supports all seven evaluation semantics via a front-end guard built with the assignment.

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

**[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 overload resolution for `p.return_void()` fails (the promise type declares `return_value` instead, or neither) and the coroutine body flows off the end, there is no `co_return` to emulate, 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 3:

- 3 If overload resolution for `p.return_void()` succeeds, flowing off the end of a coroutine's *function-body* is equivalent to a `co_return` with no operand; otherwise flowing off the end of a coroutine's *function-body* results in undefined behavior (F.4.2[ubx:stmt.return.coroutine.flow.off]).

**UB Annex Entry****F.4.2****[ub:stmt.return.coroutine.flow.off]****Specified in:** 8.8.5[ubx:stmt.return.coroutine.flow.off]

In a coroutine whose promise object is `p` and for which overload resolution for `p.return_void()` fails, flowing off the end of the *function-body* 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 cannot co_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 `constexpr` 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 variable with static or thread storage duration is guarded so its dynamic 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 3:

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

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

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

[*Example:*

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

— *end example*]

**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, and there is no constant-evaluation trigger (`---`): the rule applies only to dynamic initialization, and constant evaluation cannot pass through the declaration of a static or `thread_local` 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 depends on the runtime library providing the guard functions, not on the compiler (stock Clang on Linux uses `libstdc++` and so throws):

- **libstdc++** (**libsupc++**) 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 ...).
- **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.

With `-fno-threadsafe-statics`, or for a `thread_local` variable, no guard detects re-entry and the recursion typically overflows the stack.

**[dcl.type.cv.modify.const.obj]**

**summary:** modifying a `const` object, even via a `const_cast`-obtained pointer.  
**runtime-checkable:** Yes  
**locally-checkable:** No  
**checking:** Track whether storage is associated with a `const` object  
**replacement:** None  
**implementations:** Rejected during constant evaluation on Clang; on GCC except for objects created by `new const T`.

**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
```

- `dcl.type.cv.modify.const.obj.static.automatic` – the `const` object has static or automatic storage duration (or is a `const` subobject)
- `dcl.type.cv.modify.const.obj.dynamic` – the `const` object was created by `new const T`, as in the example above

**Wording**

Notes regarding 9.2.9.2[dcl.type.cv], paragraph 4:

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

[Example:

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

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

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

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

For another example,

```
mutable int i;
int j;
};
```

```

struct Y {
 X x;
 Y();
};

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

```

## 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 | ✗         | ✓      |        |               |         |         |                  |                  | ✓         | ✓      |        |               |         |         |                  |                  |                   |
| ↪ ...static.automatic        | ✓         | ✓      |        |               |         |         |                  |                  | ✓         | ✓      |        |               |         |         |                  |                  | CE                |
| ↪ ...dynamic                 | ✗         | ✓      |        |               |         |         |                  |                  | ✓         | ✓      |        |               |         |         |                  |                  | CE                |

## Implementation Experience

### Existing Practice

This category is detected only during constant evaluation. Clang’s constant evaluator rejects every such modification. GCC’s rejects modifications of `const` objects with automatic or static storage duration and of `const` subobjects, but accepts (and evaluates) a modification of a `const` object created by `new const T`, as in the example above (not yet reported upstream). 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; accepted by GCC's constant evaluator.

**Description**

Accessing an object defined with a `volatile`-qualified type through a non-`volatile` glvalue is undefined behavior: the compiler may cache, merge, reorder, or elide such an access, which defeats the guarantees the object's `volatile` definition was meant to provide.

```
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 5:

- 5 The semantics of an access through a `volatile` glvalue are implementation-defined. If an attempt is made to access an object defined with a `volatile`-qualified type through the use of a non-`volatile` glvalue, the behavior is undefined (F.5.2[ubx:dcl.type.cv.access.volatile]).

**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 (GCC PR126359). Clang rejects writes, increments, and accesses to `volatile` members alike.

**[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 6:

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

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

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

[*Example:*

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

char x alignas(int);
```

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

— end example]

## 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<br>UBSan |         |         | UBSan<br>UBSan    | UBSan<br>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 is not a constant expression. Clang rejects it, but GCC accepts a reference

`reinterpret_cast` during constant evaluation and then calls through it unchecked, an unrelated GCC flaw (not yet reported upstream; related: [GCC PR92411](#)).

**[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 6:

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

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

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

[*Example:*

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

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

— *end example*]

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: every route to a mistyped glvalue (`reinterpret_cast`, or `static_cast` from `void*` to a non-similar type) is not a constant expression. Clang rejects them all; GCC rejects the `static_cast` but accepts a reference `reinterpret_cast`, an unrelated GCC flaw (not yet reported upstream; related: [GCC PR92411](#)), and only rejects a later read through the reference.

**[dcl.ref.uninitialized.reference]**

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

**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 read before its initialization has run (in this or another translation unit) – is undefined behavior.

```
int i3 = 17;
int& pick() { return i3; }
extern int &ir1;
int i2 = ir1; // undefined behavior: ir1 is not yet initialized
int &ir1 = pick();
```

- `dcl.ref.uninitialized.reference.self` – a reference is read in its own initializer (`int& r = f(r);`)
- `dcl.ref.uninitialized.reference.member` – a reference member is read before its member initializer has run
- `dcl.ref.uninitialized.reference.static` – a reference with static storage duration is read before its dynamic initialization

A read from another thread that is not ordered after the initialization is also covered but not exercised.

**Wording**

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

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

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

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

[Example:

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

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

— end example]

## 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& dynamicValue() { return i3; } // not constexpr
int &ir1 = dynamicValue(); // not constant-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  |                   |
| dcl.ref.uninitialized.reference | ✗         | ✓      |        |                    |         |         |                    |                    | ✓         | ✓      |        |                   |         |         |                   |                   |                   |
| ↪ ...self                       | ✗         | ✓      |        |                    |         |         |                    |                    | ✓         | ✓      |        | MSan <sup>†</sup> |         |         | MSan <sup>†</sup> | MSan <sup>†</sup> | CE                |
| ↪ ...member                     | ✓         | ✓      |        |                    |         |         |                    |                    | ✓         | ✓      |        | MSan <sup>†</sup> |         |         | MSan <sup>†</sup> | MSan <sup>†</sup> | CE                |
| ↪ ...static                     | —         | ✓      |        | UBSan <sup>†</sup> |         |         | UBSan <sup>†</sup> | UBSan <sup>†</sup> | —         | ✓      |        |                   |         |         |                   |                   | CE                |

---

**Implementation Experience**

---

*Existing Practice*

Before its dynamic initialization, a reference with static storage duration occupies zero-initialized storage, so reading it (`.static`) is a load through a null pointer. GCC's UBSan `null` check reports that load; Clang does not instrument it and the program simply crashes. A local or member reference read before initialization (`.self`, `.member`) holds an indeterminate value; MemorySanitizer (Clang only) reports that value when it is passed to a function (parameters are checked eagerly by default) or otherwise used. All of these are heuristic. Neither compiler has a P3850 native check.

During constant evaluation, Clang rejects reading a reference in its own initializer and reading a reference member before its mem-initializer; GCC rejects only the latter ([GCC PR119339](#)). The namespace-scope form cannot be constant-evaluated: a variable is usable in a constant expression only after its initializer.

**[dcl.fct.def.coroutine.resume.not.suspended]**

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

**Description**

A coroutine handle's `resume()` member function expects the coroutine to be suspended. Calling it on a coroutine that is currently running – from inside its own frame, or concurrently from another thread – is undefined behavior. (Resuming a coroutine suspended at its final suspend point is a separate library precondition violation, [coroutine.handle.resumption].)

```
Task self_resume() {
 auto self = co_await get_own_handle{}; // obtain this frame's own handle
 self.resume(); // undefined behavior: the coroutine is running, not
 // suspended
}
```

**Wording**

Notes regarding 9.6.4[dcl.fct.def.coroutine], paragraph 8:

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

**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 task {
 struct promise_type {
 task get_return_object() { return {}; }
 std::suspend_never initial_suspend() noexcept { return {}; }
 std::suspend_never final_suspend() noexcept { return {}; }
 void return_void() noexcept {}
 };
};
```

```

 void unhandled_exception() noexcept {}
};

struct get_handle {
 std::coroutine_handle<> h;
 bool await_ready() noexcept { return false; }
 bool await_suspend(std::coroutine_handle<> c) noexcept { h = c; return false; }
 std::coroutine_handle<> await_resume() noexcept { return h; }
};

task f() {
 std::coroutine_handle<> self = co_await get_handle{};
 self.resume(); // undefined behavior, f is running, not suspended
}

```

— 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 | —         | ✓      |        | ASan <sup>†</sup> |         |         | ASan <sup>†</sup><br>ASan <sup>†</sup> | ASan <sup>†</sup> | —         | ✓      |        | ASan <sup>†</sup> |         |         | ASan <sup>†</sup><br>ASan <sup>†</sup> | ASan <sup>†</sup> | CE                |

## Implementation Experience

### Existing Practice

There is no direct support for catching this category in existing sanitizers. When a coroutine resumes itself, the nested activation can run to completion and free the frame that the outer activation is still using; AddressSanitizer reports that as a heap-use-after-free (GCC) or double-free (Clang). Because it is ASan, the check is heuristic, not precise.

Constant evaluation is inapplicable: a coroutine cannot be a `constexpr` or `constexpr` 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 (heuristic).

**Description**

Calling `destroy()` on a coroutine handle while that coroutine is not suspended – for example, from within its own still-executing frame, or while another thread is running it – is undefined behavior, since it tears down a frame that is actively running.

```
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 11:

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

**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 task {
 struct promise_type {
 task get_return_object() { return {}; }
 std::suspend_never initial_suspend() noexcept { return {}; }
 };
};
```

```

 std::suspend_never final_suspend() noexcept { return {}; }
 void return_void() noexcept {}
 void unhandled_exception() noexcept {}
};

struct get_handle {
 std::coroutine_handle<> h;
 bool await_ready() noexcept { return false; }
 bool await_suspend(std::coroutine_handle<> c) noexcept { h = c; return false; }
 std::coroutine_handle<> await_resume() noexcept { return h; }
};

task f() {
 std::coroutine_handle<> self = co_await get_handle{};
 self.destroy(); // undefined behavior, f is running, not suspended
}

```

— 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.<br>destroy.not.suspended | —         | ✓      |        | ASan <sup>†</sup> |         |         | ASan <sup>†</sup><br>ASan <sup>†</sup> | ASan <sup>†</sup> | —         | ✓      |        | ASan <sup>†</sup> |         |         | ASan <sup>†</sup><br>ASan <sup>†</sup> | ASan <sup>†</sup> | 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 the coroutine frame. Because it is ASan, the check is heuristic, not precise.

Constant evaluation is inapplicable: a coroutine cannot be a `constexpr` or `constexpr` 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 (Clang only for direct calls). GCC's constant evaluation rejects it; Clang's accepts it. The P3850 prototype checks every normal exit of the function (both compilers, all semantics, direct and indirect calls alike).

**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
}
```

- `dcl.attr.noreturn.eventually.returns.direct` – the function is called directly
- `dcl.attr.noreturn.eventually.returns.indirect` – the function is called through a function pointer

**Wording**

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

- <sup>2</sup> 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.<br>eventually.returns | ✓         | ✓      | P3850  | P3850<br>UBSan | P3850   | P3850   | P3850<br>UBSan   | P3850<br>UBSan   | ✗         | ✓      | P3850  | P3850<br>UBSan | P3850   | P3850   | P3850<br>UBSan   | P3850            |                   |
| ↪ ...direct                              | ✓         | ✓      | P3850  | P3850<br>UBSan | P3850   | P3850   | P3850<br>UBSan   | P3850<br>UBSan   | ✗         | ✓      | P3850  | P3850<br>UBSan | P3850   | P3850   | P3850<br>UBSan   | P3850            | CE                |
| ↪ ...indirect                            | ✓         | ✓      | P3850  | P3850<br>UBSan | P3850   | P3850   | P3850<br>UBSan   | P3850<br>UBSan   | ✗         | ✓      | P3850  | P3850<br>UBSan | P3850   | P3850   | P3850<br>UBSan   | P3850            | CE                |

## Implementation Experience

### Existing Practice

UBSan’s `unreachable` check (part of the `undefined` group) instruments the point after a direct call to a `[[noreturn]]` function (Clang) or the end of the `[[noreturn]]` function itself (GCC); Clang therefore misses calls through a function pointer (`.indirect`). Unlike most other UBSan checks, `unreachable` cannot recover – there is no defined way to “continue” past a `[[noreturn]]` function actually returning – so with stock flags 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.

GCC’s constant evaluator rejects a `[[noreturn]]` call that returns, direct or indirect. Clang’s accepts it with only a `-Winvalid-noreturn` warning ([llvm#211272](#)); since CWG2763 and CWG2924 whether

such a call makes an expression non-constant is implementation-defined, so this is permitted.

### *P3850 Implementation*

To enable this check in either compiler, `-fcontracts-p3100` must be specified and the group `ub:dcl.attr.noreturn.eventually.returns` given a semantic other than *assume*.

Both prototypes check in the callee: every normal exit of a `[[noreturn]]` function – a `return` statement or flowing off the end, after the function’s local variables are destroyed and its postconditions evaluated – is a contract violation, and an exception leaving the function is not. Calls through a function pointer are therefore caught as well as direct calls. Because the callee cannot tell how it was called, the check reports the group `ub:dcl.attr.noreturn.eventually.returns` itself rather than `.direct` or `.indirect`, so a configuration must name that group (or one above it) to select it.

There is no valid way to continue after such a return, so under *observe* and *noexcept-observe* the program aborts once the violation handler returns; a violation handler that throws under *enforce* or *observe* propagates its exception to the caller. *Ignore* emits no check.

**GCC:** The front end wraps the body of each `[[noreturn]]` function so that the reaction runs on every normal exit.

**Clang:** Code generation emits the reaction at the function’s return block, after the postconditions where the epilogue evaluates them.

**[class.dtor.no.longer.exists]**

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

**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.dynamic` – a non-polymorphic object with data members, in dynamically allocated storage
- `class.dtor.no.longer.exists.nonpolymorphic.automatic` – a non-polymorphic object with data members, with automatic storage duration
- `class.dtor.no.longer.exists.nonpolymorphic.empty` – a non-polymorphic object with no non-static data members (as in the example above), in any storage

The split is by whether a `vptra` exists that a check could poison and, for constant evaluation, by where GCC tracks the object's lifetime.

**Wording**

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

18

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

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

## 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 <sup>†</sup> |         |         | UBSan <sup>†</sup> | UBSan <sup>†</sup> | ✓         | ✓      |        | MSan <sup>†</sup> |         |         | MSan <sup>†</sup> | MSan <sup>†</sup> | CE                |
| ↪ ...nonpolymorphic.dynamic   | ✗<br>P3850 | ✓      |        |                    |         |         |                    |                    | ✓         | ✓      |        |                   |         |         |                   |                   | CE                |
| ↪ ...nonpolymorphic.automatic | ✓          | ✓      |        |                    |         |         |                    |                    | ✓         | ✓      |        |                   |         |         |                   |                   | CE                |
| ↪ ...nonpolymorphic.empty     | ✗          | ✓      |        |                    |         |         |                    |                    | ✓         | ✓      |        |                   |         |         |                   |                   | CE                |

## Implementation Experience

### Existing Practice

Clang rejects every double destruction during constant evaluation. GCC rejects it for an automatic object with data members and for a polymorphic object; for a heap object it is rejected only on the

P3850 GCC branch (stock GCC accepts it), and an object with no data members is accepted in any storage ([GCC PR102284](#)), hence the red X cells.

No stock runtime sanitizer catches the nonpolymorphic subgroups (the storage is still live and nothing distinguishes the second destructor call). 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* in stock form (`-fsanitize-trap=vptr` is unsupported; the P3850 GCC branch adds an integrated *quick-enforce*; a recovering *noexcept-observe* leaves the dead object's vptr in place, so it does not fire). On Clang, MemorySanitizer (`-fsanitize=memory`, whose use-after-destruction poisoning is on by default) catches the polymorphic subset instead: the first destructor poisons the object's storage, vptr included, so the second dispatch reads a poisoned vptr. This is heuristic – anything that writes to the storage in between unpoisons it – and MSan provides *noexcept-enforce* by default, *noexcept-observe* with `-fsanitize-recover=memory`, and, integrated on the P3850 Clang branch, *quick-enforce*. MSan does not catch the nonpolymorphic double destruction itself, only a later use of a value read from the dead object.

**[class.abstract.pure.virtual]**

**summary:** calling a pure virtual function from a constructor or destructor.

**runtime-checkable:** Yes

**locally-checkable:** Yes

**checking:** Insert a `pre(false)` into the pure virtual stub pointed to from the base-class vtable

**replacement:** None

**implementations:** Rejected during constant evaluation, and checkable at run time in the P3850 prototype, on both compilers (GCC's `.direct` check catches the direct call it devirtualizes); GCC's constant evaluator misses a direct call to a pure virtual function that has a definition.

**Description**

During the constructor or destructor of an abstract class, the object's dynamic type is that class, so a virtual call to one of its pure virtual functions – directly, or indirectly through another member function – has no final overrider to dispatch to and is undefined behavior (even if the pure virtual function has a definition, which only a qualified call can reach).

```
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;
};
```

- `class.abstract.pure.virtual.indirect` – the call is made through another member function
- `class.abstract.pure.virtual.direct` – the call appears directly in the constructor or destructor, so the compiler can resolve it statically

**Wording**

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

- <sup>6</sup> 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            |                    |
| ↪ ...indirect               | ✓         | ✓      |        | P3850         | P3850   | P3850   | P3850            | P3850            | ✓         | ✓      |        | P3850         | P3850   | P3850   | P3850            | P3850            | <a href="#">CE</a> |
| ↪ ...direct                 | ✗         | ✓      |        | P3850         | P3850   | P3850   | P3850            | P3850            | ✓         | ✓      |        | P3850         | P3850   | P3850   | P3850            | P3850            | <a href="#">CE</a> |

**Implementation Experience***Existing Practice*

Calling a pure virtual function from a constructor or destructor is caught during constant evaluation on both compilers (GCC: “call to non-constexpr function `__cxa_pure_virtual`”; 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 devirtualizes a direct call to a pure virtual function from the class’s own constructor or destructor (`.direct`): without a definition this is a link-time error ([GCC PR124834](#)), and with one GCC silently calls the definition ([GCC PR59246](#)), so the runtime stub never sees the call. GCC’s constant evaluator likewise calls a `constexpr` definition and accepts the expression (not yet reported upstream). Clang dispatches through the vtable and catches both subgroups.

*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*. The subgroups are configurable on their own, as `ub:class.abstract.pure.virtual.indirect` and `ub:class.abstract.pure.virtual.direct`.

For `.indirect` 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. A call GCC resolves statically to a pure virtual function – an unqualified call on `*this` in a constructor or destructor of a class that has no final overrider for it, which bypasses the vtable – is checked where the call is resolved (`build_over_call`, `cp/call.cc`) and reported as `.direct`; it reacts as the terminus does: *observe* aborts after the handler returns, and a `noexcept` pure virtual promotes *enforce* and *observe* to their `noexcept` forms. Constant evaluation is unchanged. Examples of every supported 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. Clang does not devirtualize a direct call: it is dispatched to the terminus too, and so is reported as `.indirect` (configuring `.direct` alone leaves the stock `__cxa_pure_virtual`; the `.direct` row's Clang cells configure the parent group). Examples of every supported semantic are available on <https://godbolt.org/z/vbjodWWGG>.

**[class.base.init.mem.fun]**

**summary:** calling a member function, or applying `typeid` or `dynamic_cast`, before the base classes are initialized.

**runtime-checkable:** Yes

**locally-checkable:** No

**checking:** Track whether objects are currently being constructed or destroyed

**replacement:** None

**implementations:** A virtual call is caught, heuristically, by UBSan's `vptr` on GCC only; during constant evaluation Clang rejects a virtual call or `typeid`, both compilers reject `dynamic_cast`, and only the Clang P3850 prototype rejects a member call in a constructor precondition or destructor postcondition. A plain member call in a base-class mem-initializer is caught only by the P3850 prototype, which checks each of these operations on the object itself lexically (both compilers, all semantics).

**Description**

Calling a member function on an object, or applying `typeid` or `dynamic_cast` to it, is undefined behavior while its constructor's base-class mem-initializers are still running (including in functions they call), and in a precondition assertion of its constructor or a postcondition assertion of its destructor.

```
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()) {} // OK, all bases are initialized by now
};
```

- `class.base.init.mem.fun.polymorphic.virtual.call` – a virtual call on the object in a base-class mem-initializer
- `class.base.init.mem.fun.polymorphic.typeid` – `typeid` applied to the (polymorphic) object in a base-class mem-initializer
- `class.base.init.mem.fun.polymorphic.dynamic.cast` – `dynamic_cast` applied to the object in a base-class mem-initializer
- `class.base.init.mem.fun.nonpolymorphic.mem.init` – a non-virtual member call in a base-class mem-initializer
- `class.base.init.mem.fun.nonpolymorphic.contract` – a member call in a constructor precondition or a destructor postcondition

The polymorphic operations consult the `vptr`, which does not yet describe the object, so a run-time check is possible in principle; nothing in the object identifies the nonpolymorphic error.

---

**Wording**

---

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

18

Member functions (including virtual member functions, 11.7.3[class.virtual]) can be called for an object under construction or destruction. Similarly, an object under construction or destruction can be the operand of the typeid operator (7.6.1.8[expr.typeid]) or of a dynamic\_cast (7.6.1.7[expr.dynamic.cast]). However, if these operations are performed during evaluation of

- a *ctor-initializer* (or in a function called directly or indirectly from a *ctor-initializer*) before all the *mem-initializers* 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-initializers* for base classes have completed, or after base classes have been destroyed, has undefined behavior. This includes calls in 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()
 : A(f()), // undefined behavior, calls member function in constructor precondition
 j(f()) {} // undefined behavior, calls member function but base A not yet initialized
 // defined, bases are all initialized

 B(int x)
 : B() {} // undefined behavior, calls member function in delegating constructor precondition

 ~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       | ✗         | ✓      | P3850  | P3850              | P3850   | P3850   | P3850              | P3850              | ✗         | ✓      | P3850  | P3850         | P3850   | P3850   | P3850            | P3850            |                   |
| ↪ ...polymorphic.virtual.call | ✗         | ✓      | P3850  | P3850              | P3850   | P3850   | P3850              | P3850              | ✓         | ✓      | P3850  | P3850         | P3850   | P3850   | P3850            | P3850            | CE                |
|                               |           |        |        | UBSan <sup>†</sup> |         |         | UBSan <sup>†</sup> | UBSan <sup>†</sup> |           |        |        |               |         |         |                  |                  |                   |
| ↪ ...polymorphic.typeid       | ✗         | ✓      | P3850  | P3850              | P3850   | P3850   | P3850              | P3850              | ✓         | ✓      | P3850  | P3850         | P3850   | P3850   | P3850            | P3850            | CE                |
| ↪ ...polymorphic.dynamic.cast | ✓         | ✓      | P3850  | P3850              | P3850   | P3850   | P3850              | P3850              | ✓         | ✓      | P3850  | P3850         | P3850   | P3850   | P3850            | P3850            | CE                |
| ↪ ...nonpolymorphic.mem.init  | ✗         | ✓      | P3850  | P3850              | P3850   | P3850   | P3850              | P3850              | ✗         | ✓      | P3850  | P3850         | P3850   | P3850   | P3850            | P3850            | CE                |
| ↪ ...nonpolymorphic.contract  | ✗         | ✓      | P3850  | P3850              | P3850   | P3850   | P3850              | P3850              | ✗         | ✓      | P3850  | P3850         | P3850   | P3850   | P3850            | P3850            | CE                |
|                               |           |        |        |                    |         |         |                    |                    | P3850     |        |        |               |         |         |                  |                  |                   |

---

## Implementation Experience

---

### *Existing Practice*

A virtual call before the base is initialized can be caught at run time by UBSan `vptr`, but on GCC only, and heuristically: it relies on the storage not already holding a valid-looking `vptr`. Clang's `vptr` check does not fire, and neither compiler's check instruments `typeid` or `dynamic_cast` (a `dynamic_cast` may instead crash on the uninitialized `vptr`). A nonpolymorphic call that reads a not-yet-initialized member is a separate undefined behavior (an indeterminate read), which MemorySanitizer can catch; the call itself is not detected.

During constant evaluation Clang rejects a virtual call and `typeid` in a base-class mem-initializer; GCC accepts both (not yet reported upstream; related: [GCC PR126357](#)). Both compilers reject a `dynamic_cast` that consults the `vptr`. Both accept a non-virtual call that reads no member ([GCC PR126357](#), [llvm#211286](#)). A member call in a constructor precondition or destructor postcondition is rejected only by the Clang P3850 prototype; GCC accepts it (not yet reported upstream), and stock Clang does not implement contract assertions.

### *P3850 Implementation*

To enable this check in either compiler, `-fcontracts-p3100` must be specified and the group `ub:class.base.init.mem.fun` (or one of its five subgroups) given a semantic other than *assume*.

Both prototypes check lexically and unconditionally: a member function call (named or an operator), `typeid` of a polymorphic `*this`, or a `dynamic_cast` of a polymorphic `this` or `*this`, written directly in a base-class or delegating mem-initializer, in the predicate of a constructor precondition, or in the predicate of a destructor postcondition, is a contract violation wherever it is reached. Each is reported under the subgroup it belongs to: a virtual call, `typeid` and `dynamic_cast` under the `.polymorphic` subgroups, any other member call under `.nonpolymorphic.mem.init` or `.nonpolymorphic.contract` by where it appears. Only direct forms are found (not a call through a pointer or reference to the object saved elsewhere). There is no replacement: execution continuing past a violation performs the operation as written.

**GCC:** The front end inserts the check where it builds the call, the `typeid` or the `dynamic_cast`, and in a constructor template's precondition when it is instantiated. Only a manifestly constant evaluation skips the check (accepting the call, as above), so a construction that is not manifestly constant-evaluated, even one GCC could fold to a constant (as for the `.nonpolymorphic.mem.init` example, whose constructor call has constant arguments), keeps the call and its check.

**Clang:** Code generation inserts the check when it emits the call, the `typeid` or the `dynamic_cast` in a mem-initializer or a contract predicate.

**[class.ctor.before.ctor]**

**summary:** referring to a member or base of an object before its constructor has begun.

**runtime-checkable:** Yes

**locally-checkable:** No

**checking:** Track whether objects are currently being constructed or destroyed

**replacement:** None

**implementations:** `.polymorphic` is rejected during constant evaluation on both compilers and caught, heuristically, by UBSan's `vptra` check on both compilers. `.nonpolymorphic.virtual.base` is rejected during constant evaluation only by the P3850 Clang branch and caught, heuristically, by MemorySanitizer on Clang. `.nonpolymorphic.nonvirtual` is caught by nothing.

**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.ctor.before.ctor.polymorphic` – the access is a virtual call through a pointer to the not-yet-constructed storage, reading an uninitialized `vptra`, so a runtime check is at least possible in principle
- `class.ctor.before.ctor.nonpolymorphic.virtual.base` – forming a pointer to a member through a not-yet-constructed virtual base, which needs the (not yet initialized) `vptra` to find the base, although no virtual function is involved
- `class.ctor.before.ctor.nonpolymorphic.nonvirtual` – forming a pointer to a member of a not-yet-constructed object reached without a virtual base (a member of a non-virtual base, or a member of a member)

The split is by what locating the member needs: a virtual dispatch, the `vptra` of a virtual-base path, or neither. Converting to a base before construction is `class.ctor.convert.pointer`; forming a pointer to a direct member is also `class.ctor.form.pointer`.

**Wording**

Notes regarding 11.9.5[class.ctor], paragraph 1:

1

For an object with a non-trivial constructor, referring to any non-static member or base class of the object before the constructor begins execution results in undefined behavior (F.6.4[ub:class.ctor.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[ub:class.ctor.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.ctor.before.ctor]

**Specified in:** 11.9.5[ub:class.ctor.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.ctor.before.ctor               | ✗         | ✓      |        |                    |         |         |                    |                    | ✗         | ✓      |        |                    |         |         |                    |                    |                   |
| ↪ ...polymorphic                     | ✓         | ✓      |        | UBSan <sup>†</sup> |         |         | UBSan <sup>†</sup> | UBSan <sup>†</sup> | ✓         | ✓      |        | UBSan <sup>†</sup> |         |         | UBSan <sup>†</sup> | UBSan <sup>†</sup> | CE                |
| ↪ ...nonpolymorphic.<br>virtual.base | ✗         | ✓      |        |                    |         |         |                    |                    | ✗         | ✓      |        | MSan <sup>†</sup>  |         |         | MSan <sup>†</sup>  | MSan <sup>†</sup>  | CE                |
| ↪ ...nonpolymorphic.<br>nonvirtual   | ✗         | ✓      |        |                    |         |         |                    |                    | ✗         | ✓      |        |                    |         |         |                    |                    | CE                |

**Implementation Experience***Existing Practice*

UBSan’s `vp`tr check instruments virtual member-function calls and non-static member accesses on a polymorphic object: if the object’s `vp`tr 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, which reads an uninitialized `vp`tr – is in many cases caught by UBSan on both compilers, and is rejected during constant evaluation on both.

In constant evaluation the virtual-base shape is rejected only by the P3850 Clang branch (a fix not yet upstream); GCC and stock Clang accept it (GCC PR126357, llvm#211286, both partial). The non-virtual shapes are accepted by all four compilers (GCC PR126358; related for Clang: llvm#211286). At run time, the virtual-base instance is caught, heuristically, by MemorySanitizer (`-fsanitize=memory`, Clang only), provided the storage has not been written before: MSan provides *noexcept-enforce* by default, *noexcept-observe* with `-fsanitize-recover=memory`, and, integrated on the P3850 Clang branch, *quick-enforce*. The non-virtual instance reads nothing, so no sanitizer sees it.

**[class.ctor.after.dtor]**

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

**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.ctor.after.dtor.polymorphic` – a virtual call through a pointer whose object's destructor has run, which reads the dead object's `vptra`
- `class.ctor.after.dtor.nonpolymorphic.access` – a read or write of a data member of an object whose destructor has run
- `class.ctor.after.dtor.nonpolymorphic.address` – forming a pointer to a member, or converting to a base, after the destructor has run, with no access

The split is by what the reference does: a virtual dispatch, an access to a data member, or only the formation of an address. A non-virtual member call that touches no member falls between them: Clang rejects it in constant evaluation, GCC accepts it (not yet reported upstream).

**Wording**

Notes regarding 11.9.5[class.ctor], paragraph 1:

- 1 For an object with a non-trivial constructor, referring to any non-static member or base class of the object before the constructor begins execution results in undefined behavior (F.6.4[ubx.class.ctor.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.ctor.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.ctor.after.dtor]

**Specified in:** 11.9.5[ubx:class.ctor.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 <sup>†</sup> |         |         | UBSan <sup>†</sup> |                  | ✓         | ✓      |        | MSan <sup>†</sup> |         |         | MSan <sup>†</sup> | MSan <sup>†</sup> | CE               |
| ↪ ...nonpolymorphic.access  | ✓         | ✓      |        |                    |         |         | UBSan <sup>†</sup> |                  | ✓         | ✓      |        | MSan <sup>†</sup> |         |         | MSan <sup>†</sup> | MSan <sup>†</sup> | CE               |
| ↪ ...nonpolymorphic.address | ✗         | ✓      |        |                    |         |         |                    |                  | ✗         | ✓      |        |                   |         |         |                   |                   | CE               |

**Implementation Experience***Existing Practice*

Constant evaluation on both compilers rejects a virtual call and a data-member read or write on an object after its destructor has run. Forming a member's address or converting to a base after destruction is accepted by both ([GCC PR102284](#); not yet reported upstream for Clang), hence the red X on `.nonpolymorphic.address`. Every shape here is also a use of a pointer to an object outside its lifetime (`lifetime.outside.*`).

At run time, the *polymorphic* subset – a virtual call through a pointer whose object's destructor has run, reading an overwritten/invalid vp<sub>tr</sub> – is caught by UBSan vp<sub>tr</sub> on GCC only, though that requires the invalidated vp<sub>tr</sub> to not have been overwritten with something valid since the object's destruction, and is thus heuristic and not reliable. The non-polymorphic cases are not checked by UBSan or ASan as long as the storage is still allocated.

On Clang, MemorySanitizer (`-fsanitize=memory`) catches the polymorphic and `.nonpolymorphic.access` subsets: its use-after-destruction poisoning (`-fsanitize-memory-use-after-dtor`, on by default) marks an object's storage, vp<sub>tr</sub> included, as uninitialized once its destructor finishes, so a virtual call through the dead object, or a use of a member value read from it, is reported. This too is heuristic – anything that writes to the storage after the destructor unpoisons it, and a value read but never used, or a write, is not reported. MSan provides *noexcept-enforce* by default, *noexcept-observe* with `-fsanitize-recover=memory`, and, integrated on the P3850 Clang branch, *quick-enforce*. Forming an address reads nothing, so MSan cannot see `.nonpolymorphic.address`.

**[class.ctor.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 an object of class **X** to a pointer to its base **B** is undefined behavior unless the construction of **X** and of every base of **X** that derives from **B** has started and none of their destructions has completed – 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* while D
 // (which derives from A) has not started construction
 X(this) {} // OK, by now B, D and A are all constructed
};
```

**Wording**

Notes regarding 11.9.5[class.ctor], paragraph 3:

- <sup>3</sup> 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.ctor.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.ctor.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.ctor.convert.pointer]

**Specified in:** 11.9.5[ubx:class.ctor.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, the 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 were 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.ctor.convert.pointer | ✗         | ✓      |        |               |         |         |                  |                  | ✗         | ✓      |        |               |         |         |                  |                  | CE                |

## Implementation Experience

### Existing Practice

No stock tool checks this conversion, and constant evaluation accepts it on both compilers (not yet reported upstream), hence the red X.

**[class.ctor.form.pointer]**

**summary:** forming a pointer to a member before construction starts or after destruction finishes.

**runtime-checkable:** Yes

**locally-checkable:** No

**checking:** Track whether objects are currently being constructed or destroyed

**replacement:** None

**implementations:** For polymorphic objects, caught, heuristically, by UBSan's `vptra` check on Clang. Constant evaluation accepts both subgroups on both compilers.

**Description**

Merely *forming* a pointer to (or accessing the value of) a direct non-static member of an object is undefined behavior if construction of that object has not yet started, or its destruction has already finished. The examples here illustrate forming the pointer before construction.

```
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
};
```

Because A's constructor is non-trivial, the example is also `class.ctor.before.ctor`.

- `class.ctor.form.pointer.polymorphic` – the member's own class is polymorphic, so a runtime vtable check is at least possible in principle
- `class.ctor.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.ctor], paragraph 3:

3

To explicitly or implicitly convert a pointer (a glvalue) referring to an object of class X to a pointer (reference) to a direct or indirect base class B of X, the construction of X and the construction of all of its direct or indirect bases that directly or indirectly derive from B shall have started and the destruction of these classes shall not have completed, otherwise the conversion results in undefined behavior (F.6.6[ubx:class.ctor.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.ctor.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.ctor.form.pointer]

**Specified in:** 11.9.5[ubx:class.ctor.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.ctor.form.pointer | ✗         | ✓      |        |               |         |         |                  |                  | ✗         | ✓      |        |                    |         |         |                    |                    |                   |
| ↪ ...polymorphic        | ✗         | ✓      |        |               |         |         |                  |                  | ✗         | ✓      |        | UBSan <sup>†</sup> |         |         | UBSan <sup>†</sup> | UBSan <sup>†</sup> | 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.

Constant evaluation accepts forming such a pointer on both compilers, before construction or after destruction, but rejects reading the member’s value, hence the red X on both subgroups ([GCC PR126358](#); related for Clang: [llvm#211286](#)).

**[class.ctor.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(); // OK, calls V::f, 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.ctor], paragraph 4:

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

**Specified in:** 11.9.5[ubx:class.ctor.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.ctor.virtual.not.x | ✗         | ✓      |        |               |         |         |                  |                  | ✗         | ✓      |        |               |         |         |                  |                  | CE                |

Implementation Experience

Existing Practice

No stock tool checks this call: at run time it silently dispatches to some overrider (`A::f` on GCC, `D::f`, which the linked example adds, on Clang). Constant evaluation accepts it on both compilers (not yet reported upstream), hence the red X.

**[class.ctor.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); // OK, 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.ctor], paragraph 5:

- 5 The typeid operator (7.6.1.8[expr.typeid]) can be used during construction or destruction (11.9.3[class.base.init]). When typeid is used in a constructor (including the *mem-initializer* or default member initializer (11.4[class.mem])) for a non-static data member) or in a destructor, or used in a function called (directly or indirectly) from a constructor or destructor, if the operand of typeid refers to the object under construction or destruction, typeid yields the std::type\_info object representing the constructor or destructor's class. If the operand of typeid refers to the object under construction or destruction and the static type of the operand is neither the constructor or destructor's class nor one of its bases, the behavior is undefined (F.6.9[ubx.class.ctor.typeid]).

**UB Annex Entry****F.6.9****[ub:class.ctor.typeid]****Specified in:** 11.9.5[ubx.class.ctor.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                |

Implementation Experience

Existing Practice

No stock tool checks this use: at run time `typeid` silently yields the type of the constructor’s class. Constant evaluation accepts it on both compilers (not yet reported upstream), hence the red X.

**[class.ctor.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); // OK, 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.ctor], paragraph 6:

- 6 `dynamic_casts` (7.6.1.7[expr.dynamic.cast]) can be used during construction or destruction (11.9.3[class.base.init]). When a `dynamic_cast` is used in a constructor (including the *mem-initializer* or default member initializer for a non-static data member) or in a destructor, or used in a function called (directly or indirectly) from a constructor or destructor, if the operand of the `dynamic_cast` refers to the object under construction or destruction, this object is considered to be a most derived object that has the type of the constructor or destructor's class. If the operand of the `dynamic_cast` refers to the object under construction or destruction and the static type of the operand is not a pointer to or object of the constructor or destructor's own class or one of its bases, the `dynamic_cast` results in undefined behavior (F.6.10[ubx:class.ctor.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.ctor.dynamic.cast]

**Specified in:** 11.9.5[ubx:class.ctor.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.ctor.dynamic.cast | ✗         | ✓      |        |               |         |         |                  |                  | ✗         | ✓      |        |               |         |         |                  |                  | CE                |

---

**Implementation Experience**

---

*Existing Practice*

No stock tool checks this cast: at run time it silently yields a null pointer on both compilers. Constant evaluation accepts it on both compilers, which even disagree on the result (GCC yields a non-null pointer, Clang a null one), hence the red X ([GCC PR93096](#); not yet reported upstream for Clang).

**[except.handle.handler.ctor.dtor]**

**summary:** referring to a member/base of an object from its ctor/dtor function-try-block handler.

**runtime-checkable:** Yes

**locally-checkable:** No

**checking:** Track whether objects are currently being constructed or destroyed

**replacement:** None

**implementations:** Direct references caught by the P3850 prototype's lexical check (both compilers, all semantics); GCC's constant evaluator rejects `.ended` but accepts `.initialized` and `.reference`, and Clang lacks `constexpr` exceptions; for `.ended`, MemorySanitizer (Clang) heuristically catches use of the value read; Clang warns at compile time for direct references in a constructor's handler.

**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;
};
```

- `except.handle.handler.ctor.dtor.ended` – the handler accesses a member or base that was never initialized (as in the example above), a class-type member or base that has already been destroyed, or any member or base in a destructor's handler
- `except.handle.handler.ctor.dtor.initialized` – a constructor's handler reads a scalar member whose initialization completed before the exception was thrown
- `except.handle.handler.ctor.dtor.reference` – the handler refers to a member without accessing it (for example, `&y`)

**Wording**

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

- 11 Referring to any non-static member or base class of an object in the handler for a *function-try-block* of a constructor or destructor for that object results in undefined behavior (F.8.1[ubx:except.handle.handler.ctor.dtor]).

## 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 | ✗         | ✓      | P3850  | P3850         | P3850   | P3850   | P3850            | P3850            | —         | ✓      | P3850  | P3850         | P3850   | P3850   | P3850            | P3850            |                   |
| ↪ ...ended                      | ✓         | ✓      | P3850  | P3850         | P3850   | P3850   | P3850            | P3850            | —         | ✓      | P3850  | P3850         | P3850   | P3850   | P3850            | P3850            | CE                |
| ↪ ...initialized                | ✗         | ✓      | P3850  | P3850         | P3850   | P3850   | P3850            | P3850            | —         | ✓      | P3850  | P3850         | P3850   | P3850   | P3850            | P3850            | CE                |
| ↪ ...reference                  | ✗         | ✓      | P3850  | P3850         | P3850   | P3850   | P3850            | P3850            | —         | ✓      | P3850  | P3850         | P3850   | P3850   | P3850            | P3850            | 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 supports `constexpr` exceptions, and its constant evaluator rejects a handler's access to a member or base that was never initialized, or to a class-type member or base already destroyed (as an uninitialized or out-of-lifetime access), but accepts a read of a scalar member initialized before the throw and any reference that is not an access, such as `&y` (not yet reported upstream). Clang does not yet support `constexpr` exceptions, so its constant-evaluation cells are ---.

When the member referred to in the handler was never initialized (the exception was thrown before its initializer ran), MemorySanitizer (`-fsanitize=memory`, Clang only) catches, heuristically, a use of the value read from it – it reports where the value influences behavior, not at the reference in the handler itself, and a member that was already initialized is not caught this way. MSan provides *noexcept-enforce* by default, *noexcept-observe* with `-fsanitize-recover=memory`, and, integrated on the P3850 Clang branch, *quick-enforce*.

### *P3850 Implementation*

To enable this check in either compiler, `-fcontracts-p3100` must be specified and the group `ub:except.handle.handler.ctor.dtor` given a semantic other than *assume*.

Both prototypes check lexically and unconditionally: a non-static data member access (of the class or of a base, address-taking included) or a member function call (named or an operator) on `*this` inside the handler of a constructor's or destructor's *function-try-block* is a contract violation wherever it is reached. Only direct forms are found – not a reference through a lambda's captured `this`, nor through a pointer or reference to the object saved before the handler – and a bare conversion of `this` to a base is not checked. Because the check cannot know which members were initialized before the exception, it reports the group `ub:except.handle.handler.ctor.dtor` itself rather than `.ended`, `.initialized` or `.reference`, so a configuration must name that group (or one above it) to select it. There is no replacement: execution continuing past a violation performs the reference as written.

**GCC:** The front end inserts the check where it builds the member access or the member function call.

**Clang:** Code generation inserts the check when it emits the member access or the member function call.

## 2.4 Excluded: Compile-Time Template Recursion

The wording of undefined behaviors in this section is likely to be removed or changed to no longer classify the situation as undefined behavior. Infinite template-instantiation recursion is a compile-time issue, not runtime undefined behavior; [P3100R9] treats it as a wording bug being addressed by [CWG3034].

**[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
}
```

In practice, compilers reject the above code with a fatal error once the implementation-defined maximum instantiation depth ([implimits]) is exceeded.

**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 [P3100R9] 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*, *ignore*, *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 [P3100R9] 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 every category of undefined behavior and its 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. It records what that compiler’s constant evaluator does when the undefined behavior is forced into a `static_assert` – independent of `-fcontracts-p3100`, `-fsanitize=`, or any of the evaluation semantics – and always holds one of four marks:

- ✓ – rejected (the expression is not a core constant expression because of this undefined behavior) by both stock trunk and our P3850 branch;
- ✗<sub>P3850</sub> – rejected only by our P3850 branch; stock trunk accepts it;
- ✗ – a *constexpr* trigger exists, but the compiler accepts it; the category’s prose links the upstream bug report where one exists;
- — – no *constexpr* trigger can exist (for example, the undefined behavior requires threads, a `reinterpret_cast`, or a construct that is never *constexpr*); the category’s prose says why.

### 3 IMPLEMENTATION STATUS

A category whose variations are caught differently is split into sub-groups, so that each row is, for each compiler and each detection method, either caught, caught heuristically, or not caught.

Every row except an N/A row (N/A in the Compiler Explorer column) 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 [P3100R9]; an indented row such as  $\hookrightarrow$  `known.subscript` 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 <sup>†</sup> |         |         | MSan <sup>†</sup> | MSan <sup>†</sup> |                   |
| ↪ ...attr.indeterminate          | ✓         | ✓      |        |                   |         |         |                   |                   | ✓         | ✓      |        | MSan <sup>†</sup> |         |         | MSan <sup>†</sup> | MSan <sup>†</sup> | CE                |
| ↪ ...object.padding              | ✓         | ✓      |        |                   |         |         |                   |                   | ✓         | ✓      |        | MSan <sup>†</sup> |         |         | MSan <sup>†</sup> | MSan <sup>†</sup> | CE                |
| II. Bounds                       |           |        |        |                   |         |         |                   |                   |           |        |        |                   |         |         |                   |                   |                   |
| basic.stc.alloc.zero.dereference | ✗         | ✓      |        |                   |         |         |                   |                   | ✓         | ✓      |        |                   |         |         |                   |                   |                   |
| ↪ ...access                      | ✓         | ✓      |        | ASan <sup>†</sup> |         |         | ASan <sup>†</sup> | ASan <sup>†</sup> | ✓         | ✓      |        | ASan <sup>†</sup> |         |         | ASan <sup>†</sup> | ASan <sup>†</sup> | CE                |
| ↪ ...indirection                 | ✗         | ✓      |        |                   |         |         |                   |                   | ✓         | ✓      |        |                   |         |         |                   |                   | CE                |
| expr.delete.mismatch             | ✓         | ✓      |        | ASan              |         |         | ASan              | ASan              | ✓         | ✓      |        | ASan              |         |         | ASan              | ASan              | CE                |
| expr.delete.array.mismatch       | ✓         | ✓      |        | ASan              |         |         | ASan              | ASan              | ✓         | ✓      |        | ASan              |         |         | ASan              | ASan              | CE                |
| expr.add.out.of.bounds           | ✗         | ✓      |        |                   |         |         |                   |                   | ✓         | ✓      |        |                   |         |         |                   |                   |                   |
| ↪ ...known.subscript             | ✓         | ✓      | P3850  | P3850             | P3850   | P3850   | P3850             | P3850             | ✓         | ✓      | P3850  | P3850             | P3850   | P3850   | P3850             | P3850             | CE                |
|                                  |           |        |        | UBSan             |         |         | UBSan             | UBSan             |           |        |        | UBSan             |         |         | UBSan             | UBSan             |                   |
|                                  |           |        |        | UBSan             |         |         | UBSan             | UBSan             |           |        |        | UBSan             |         |         | UBSan             | UBSan             |                   |
| ↪ ...known.arithmetic            | ✗         | ✓      | P3850  | P3850             | P3850   | P3850   | P3850             | P3850             | ✓         | ✓      | P3850  | P3850             | P3850   | P3850   | P3850             | P3850             | CE                |
|                                  |           |        |        |                   |         |         |                   |                   |           |        |        | UBSan             |         |         | UBSan             | UBSan             |                   |
|                                  |           |        |        |                   |         |         |                   |                   |           |        |        | UBSan             |         |         | UBSan             | UBSan             |                   |
| ↪ ...unknown                     | ✗         | ✓      |        | ASan <sup>†</sup> |         |         | ASan <sup>†</sup> | ASan <sup>†</sup> | ✓         | ✓      |        | ASan <sup>†</sup> |         |         | ASan <sup>†</sup> | ASan <sup>†</sup> | CE                |
| ↪ ...pointer.arithmetic          | —         | ✓      | P3850  | P3850             | P3850   | P3850   | P3850             | P3850             | —         | ✓      | P3850  | P3850             | P3850   | P3850   | P3850             | P3850             | CE                |
|                                  |           |        |        | UBSan             |         |         | UBSan             | UBSan             |           |        |        | UBSan             |         |         | UBSan             | UBSan             |                   |
|                                  |           |        |        | UBSan             |         |         | UBSan             | UBSan             |           |        |        | UBSan             |         |         | UBSan             | UBSan             |                   |
| expr.add.sub.diff.pointers       | ✗         | ✓      |        |                   |         |         |                   |                   | ✓         | ✓      |        |                   |         |         |                   |                   |                   |
| ↪ ...distinct.objects            | ✓         | ✓      |        | ASan              |         |         | ASan              | ASan              | ✓         | ✓      |        | ASan              |         |         | ASan              | ASan              | CE                |
|                                  |           |        |        |                   |         |         |                   |                   |           |        |        |                   |         |         |                   |                   |                   |
| ↪ ...same.object                 | ✗         | ✓      |        |                   |         |         |                   |                   | ✓         | ✓      |        |                   |         |         |                   |                   | CE                |
| III. Type and Lifetime           |           |        |        |                   |         |         |                   |                   |           |        |        |                   |         |         |                   |                   |                   |
| intro.object.implicit.create     | —         | ✓      |        |                   |         |         |                   |                   | —         | ✓      |        |                   |         |         |                   |                   | CE                |

### 3 IMPLEMENTATION STATUS

| 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                |
| basic.align.object.alignment                   | —          | ✓      | P3850  | P3850              |         |         | P3850                                    | P3850              | —          | ✓      | P3850  | P3850<br>UBSan    | P3850   | P3850   | P3850<br>UBSan                         | P3850<br>UBSan                         |                   |
| ↪ ...store                                     | —          | ✓      | P3850  | P3850<br>UBSan     |         |         | P3850<br>UBSan                           | P3850<br>UBSan     | —          | ✓      | P3850  | P3850<br>UBSan    | P3850   | P3850   | P3850<br>UBSan                         | P3850<br>UBSan                         | CE                |
| ↪ ...nostream                                  | —          | ✓      | P3850  | P3850              | P3850   | P3850   | P3850                                    | P3850              | —          | ✓      | P3850  | P3850<br>UBSan    | P3850   | P3850   | P3850<br>UBSan                         | P3850<br>UBSan                         | CE                |
| lifetime.outside.pointer.delete                | ✗<br>P3850 | ✓      |        |                    |         |         |                                          |                    | ✓          | ✓      |        |                   |         |         |                                        |                                        | CE                |
| lifetime.outside.pointer.member                | ✗          | ✓      |        |                    |         |         |                                          |                    | ✓          | ✓      |        |                   |         |         |                                        |                                        |                   |
| ↪ ...data                                      | ✓          | ✓      |        |                    |         |         |                                          |                    | ✓          | ✓      |        |                   |         |         |                                        |                                        | CE                |
| ↪ ...call                                      | ✗          | ✓      |        |                    |         |         |                                          |                    | ✓          | ✓      |        |                   |         |         |                                        |                                        | CE                |
| lifetime.outside.pointer.virtual               | ✓          | ✓      |        | UBSan <sup>†</sup> |         |         | UBSan <sup>†</sup><br>UBSan <sup>†</sup> | UBSan <sup>†</sup> | ✗<br>P3850 | ✓      |        | MSan <sup>†</sup> |         |         | MSan <sup>†</sup><br>MSan <sup>†</sup> | MSan <sup>†</sup><br>MSan <sup>†</sup> | CE                |
| lifetime.outside.pointer.dynamic.cast          | ✗          | ✓      |        |                    |         |         |                                          |                    | ✓          | ✓      |        |                   |         |         |                                        |                                        |                   |
| ↪ ...automatic                                 | ✓          | ✓      |        |                    |         |         |                                          |                    | ✓          | ✓      |        |                   |         |         |                                        |                                        | CE                |
| ↪ ...dynamic                                   | ✗          | ✓      |        |                    |         |         |                                          |                    | ✓          | ✓      |        |                   |         |         |                                        |                                        | CE                |
| lifetime.outside.glvalue.access                | ✗<br>P3850 | ✓      |        |                    |         |         |                                          |                    | ✓          | ✓      |        |                   |         |         |                                        |                                        | CE                |
| lifetime.outside.glvalue.member                | ✗          | ✓      |        |                    |         |         |                                          |                    | ✓          | ✓      |        |                   |         |         |                                        |                                        | CE                |
| lifetime.outside.glvalue.virtual               | ✗          | ✓      |        |                    |         |         |                                          |                    | ✗<br>P3850 | ✓      |        | MSan <sup>†</sup> |         |         | MSan <sup>†</sup><br>MSan <sup>†</sup> | MSan <sup>†</sup><br>MSan <sup>†</sup> |                   |
| ↪ ...named                                     | ✗          | ✓      |        |                    |         |         |                                          |                    | ✗<br>P3850 | ✓      |        | MSan <sup>†</sup> |         |         | MSan <sup>†</sup><br>MSan <sup>†</sup> | MSan <sup>†</sup><br>MSan <sup>†</sup> | CE                |
| ↪ ...unnamed                                   | ✓          | ✓      |        | UBSan <sup>†</sup> |         |         | UBSan <sup>†</sup><br>UBSan <sup>†</sup> | UBSan <sup>†</sup> | ✗<br>P3850 | ✓      |        | MSan <sup>†</sup> |         |         | MSan <sup>†</sup><br>MSan <sup>†</sup> | MSan <sup>†</sup><br>MSan <sup>†</sup> | CE                |
| lifetime.outside.glvalue.dynamic.cast          | ✗          | ✓      |        |                    |         |         |                                          |                    | ✓          | ✓      |        |                   |         |         |                                        |                                        |                   |
| ↪ ...dynamic.cast                              | ✓          | ✓      |        |                    |         |         |                                          |                    | ✓          | ✓      |        |                   |         |         |                                        |                                        | CE                |
| ↪ ...typeid                                    | ✗          | ✓      |        |                    |         |         |                                          |                    | ✓          | ✓      |        | MSan <sup>†</sup> |         |         | MSan <sup>†</sup><br>MSan <sup>†</sup> | MSan <sup>†</sup><br>MSan <sup>†</sup> | CE                |
| original.type.implicit.destructor              | —          | ✓      |        |                    |         |         |                                          |                    | —          | ✓      |        |                   |         |         |                                        |                                        | CE                |
| creating.within.const.complete.obj             | ✗          | ✓      |        |                    |         |         |                                          |                    | ✓          | ✓      |        |                   |         |         |                                        |                                        |                   |
| ↪ ...live                                      | ✗          | ✓      |        |                    |         |         |                                          |                    | ✓          | ✓      |        |                   |         |         |                                        |                                        | CE                |
| ↪ ...reuse                                     | ✗          | ✓      |        |                    |         |         |                                          |                    | ✓          | ✓      |        |                   |         |         |                                        |                                        | CE                |
| basic.compound.pointer.before.storage.duration | —          | ✓      |        |                    |         |         |                                          |                    | —          | ✓      |        |                   |         |         |                                        |                                        | CE                |

### 3 IMPLEMENTATION STATUS

| 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              | ✗         | ✓      |        |                                         |         |         |                                                                                    |                                          | ✓         | ✓      |        |                         |         |         |                                          |                         |                   |
| ↪ ...access                                 | ✓         | ✓      |        | ASan <sup>†</sup>                       |         |         | ASan <sup>†</sup>                                                                  | ASan <sup>†</sup>                        | ✓         | ✓      |        | ASan <sup>†</sup>       |         |         | ASan <sup>†</sup>                        | ASan <sup>†</sup>       | CE                |
| ↪ ...indirection                            | ✗         | ✓      |        |                                         |         |         |                                                                                    |                                          | ✓         | ✓      |        |                         |         |         |                                          |                         | 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<br>UBSan<br>UBSan                 |         |         | P3850<br>UBSan<br>UBSan                                                            | P3850<br>UBSan<br>UBSan                  | —         | ✓      | P3850  | P3850<br>UBSan<br>UBSan | P3850   | P3850   | P3850<br>UBSan<br>UBSan                  | P3850<br>UBSan<br>UBSan |                   |
| ↪ ...bool                                   | —         | ✓      | P3850  | P3850<br>UBSan<br>UBSan                 |         |         | P3850<br>UBSan<br>UBSan                                                            | P3850<br>UBSan<br>UBSan                  | —         | ✓      | P3850  | P3850<br>UBSan<br>UBSan | P3850   | P3850   | P3850<br>UBSan<br>UBSan                  | P3850<br>UBSan<br>UBSan | CE                |
| ↪ ...enum                                   | —         | ✓      | P3850  | P3850<br>UBSan<br>UBSan                 |         |         | P3850<br>UBSan<br>UBSan                                                            | P3850<br>UBSan<br>UBSan                  | —         | ✓      | P3850  | P3850<br>UBSan<br>UBSan | P3850   | P3850   | P3850<br>UBSan<br>UBSan                  | P3850<br>UBSan<br>UBSan | CE                |
| ↪ ...int                                    | —         |        |        |                                         |         |         |                                                                                    |                                          | —         |        |        |                         |         |         |                                          |                         | N/A               |
| ↪ ...float                                  | —         |        |        |                                         |         |         |                                                                                    |                                          | —         |        |        |                         |         |         |                                          |                         | N/A               |
| conv.array.unsimilar.type                   | —         | ✓      |        |                                         |         |         |                                                                                    |                                          | —         | ✓      |        |                         |         |         |                                          |                         | CE                |
| conv.ptr.virtual.base                       | ✗         | ✓      |        |                                         |         |         |                                                                                    |                                          | ✗         | ✓      |        |                         |         |         |                                          |                         |                   |
| ↪ ...past.end                               | ✓         | ✓      |        | UBSan <sup>†</sup><br>ASan <sup>†</sup> |         |         | UBSan <sup>†</sup><br>ASan <sup>†</sup><br>UBSan <sup>†</sup><br>ASan <sup>†</sup> | UBSan <sup>†</sup><br>ASan <sup>†</sup>  | ✓         | ✓      |        | ASan <sup>†</sup>       |         |         | ASan <sup>†</sup><br>ASan <sup>†</sup>   | ASan <sup>†</sup>       | CE                |
| ↪ ...lifetime                               | ✓         | ✓      |        | UBSan <sup>†</sup>                      |         |         | UBSan <sup>†</sup><br>UBSan <sup>†</sup>                                           | UBSan <sup>†</sup>                       | ✗         | ✓      |        | UBSan <sup>†</sup>      |         |         | UBSan <sup>†</sup><br>UBSan <sup>†</sup> | UBSan <sup>†</sup>      | CE                |
| ↪ ...union                                  | ✗         | ✓      |        |                                         |         |         |                                                                                    |                                          | ✗         | ✓      |        | UBSan <sup>†</sup>      |         |         | UBSan <sup>†</sup><br>UBSan <sup>†</sup> | UBSan <sup>†</sup>      | CE                |
| conv.member.missing.member                  | ✗         | ✓      |        |                                         |         |         |                                                                                    |                                          | ✓         | ✓      |        |                         |         |         |                                          |                         | CE                |
| expr.call.different.type                    | —         | ✓      |        |                                         |         |         |                                                                                    |                                          | —         | ✓      |        | UBSan<br>UBSan          |         |         | UBSan<br>UBSan                           | UBSan                   | CE                |
| expr.ref.member.not.similar                 | —         | ✓      |        |                                         |         |         |                                                                                    |                                          | —         | ✓      |        |                         |         |         |                                          |                         |                   |
| ↪ ...polymorphic                            | —         | ✓      |        | UBSan <sup>†</sup>                      |         |         | UBSan <sup>†</sup><br>UBSan <sup>†</sup>                                           | UBSan <sup>†</sup><br>UBSan <sup>†</sup> | —         | ✓      |        |                         |         |         |                                          |                         | CE                |
| ↪ ...nonpolymorphic                         | —         | ✓      |        |                                         |         |         |                                                                                    |                                          | —         | ✓      |        |                         |         |         |                                          |                         | CE                |
| expr.dynamic.cast.pointer.lifetime          | ✗         | ✓      |        |                                         |         |         |                                                                                    |                                          | ✓         | ✓      |        | UBSan <sup>†</sup>      |         |         | UBSan <sup>†</sup><br>UBSan <sup>†</sup> | UBSan <sup>†</sup>      | CE                |
| expr.dynamic.cast.glvalue.lifetime          | ✓         | ✓      |        |                                         |         |         |                                                                                    |                                          | ✓         | ✓      |        |                         |         |         | UBSan <sup>†</sup><br>UBSan <sup>†</sup> | UBSan <sup>†</sup>      | CE                |

### 3 IMPLEMENTATION STATUS

| 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             | ✓         | ✓      |        | 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.access                               | ✓          | ✓      | P3850  | P3850             |         |         | P3850             | P3850             | ✓         | ✓      | P3850  | P3850             | P3850   | P3850   | P3850             | P3850             | CE                |
|                                                   |            |        |        | UBSan             |         |         | UBSan             | UBSan             |           |        |        | UBSan             |         |         | UBSan             | UBSan             |                   |
|                                                   |            |        |        | UBSan             |         |         | UBSan             | UBSan             |           |        |        | UBSan             |         |         | UBSan             | UBSan             |                   |
| ↪ ...nullptr.indirection                          | ✗          | ✓      | P3850  | P3850             |         |         | P3850             | P3850             | ✗         | ✓      | P3850  | P3850             | P3850   | P3850   | P3850             | P3850             | CE                |
| ↪ ...invalid.access                               | ✓          | ✓      |        | ASan <sup>†</sup> |         |         | ASan <sup>†</sup> | ASan <sup>†</sup> | ✓         | ✓      |        | ASan <sup>†</sup> |         |         | ASan <sup>†</sup> | ASan <sup>†</sup> | CE                |
|                                                   |            |        |        |                   |         |         | ASan <sup>†</sup> | ASan <sup>†</sup> |           |        |        |                   |         |         | ASan <sup>†</sup> | ASan <sup>†</sup> |                   |
| ↪ ...invalid.indirection                          | ✗          | ✓      |        |                   |         |         |                   |                   | ✗         | ✓      |        |                   |         |         |                   |                   | CE                |
| expr.delete.dynamic.type.differ                   | ✗<br>P3850 | ✓      |        | ASan <sup>†</sup> |         |         | ASan <sup>†</sup> | ASan <sup>†</sup> | ✓         | ✓      |        | ASan <sup>†</sup> |         |         | ASan <sup>†</sup> | ASan <sup>†</sup> | CE                |
| expr.delete.dynamic.array.dynamic.type.differ     | ✗          | ✓      |        | ASan <sup>†</sup> |         |         | ASan <sup>†</sup> | ASan <sup>†</sup> | ✓         | ✓      |        | ASan <sup>†</sup> |         |         | ASan <sup>†</sup> | ASan <sup>†</sup> | CE                |
| expr.mpPtr.oper.not.contain.member                | ✗          | ✓      |        | ASan <sup>†</sup> |         |         | ASan <sup>†</sup> | ASan <sup>†</sup> | ✓         | ✓      |        | ASan <sup>†</sup> |         |         | ASan <sup>†</sup> | ASan <sup>†</sup> | CE                |
| expr.mpPtr.oper.member.func.null                  | ✗          | ✓      | P3850  | P3850             | P3850   | P3850   | P3850             | P3850             | ✓         | ✓      | P3850  | P3850             | P3850   | P3850   | P3850             | P3850             |                   |
| ↪ ...function                                     | ✓          | ✓      | P3850  | P3850             | P3850   | P3850   | P3850             | P3850             | ✓         | ✓      | P3850  | P3850             | P3850   | P3850   | P3850             | P3850             | CE                |
| ↪ ...data                                         | ✗          | ✓      | P3850  | P3850             | P3850   | P3850   | P3850             | P3850             | ✓         | ✓      | P3850  | P3850             | P3850   | P3850   | P3850             | P3850             | CE                |
| expr.add.not.similar                              | —          | ✓      |        |                   |         |         |                   |                   | —         | ✓      |        |                   |         |         |                   |                   | CE                |
| expr.assign.overlap                               | ✗          | ✓      | P3850  | P3850             | P3850   | P3850   | P3850             | P3850             | ✗         | ✓      | P3850  | P3850             | P3850   | P3850   | P3850             | P3850             | CE                |
| dcl.type.cv.modify.const.obj                      | ✗          | ✓      |        |                   |         |         |                   |                   | ✓         | ✓      |        |                   |         |         |                   |                   |                   |
| ↪ ...static.automatic                             | ✓          | ✓      |        |                   |         |         |                   |                   | ✓         | ✓      |        |                   |         |         |                   |                   | CE                |
| ↪ ...dynamic                                      | ✗          | ✓      |        |                   |         |         |                   |                   | ✓         | ✓      |        |                   |         |         |                   |                   | CE                |
| dcl.type.cv.access.volatile                       | ✗          | ✓      |        |                   |         |         |                   |                   | ✓         | ✓      |        |                   |         |         |                   |                   | CE                |
| dcl.ref.incompatible.function                     | —          | ✓      |        |                   |         |         |                   |                   | —         | ✓      |        | UBSan             |         |         | UBSan             | UBSan             | CE                |
|                                                   |            |        |        |                   |         |         |                   |                   |           |        |        | UBSan             |         |         | UBSan             | UBSan             |                   |
| dcl.ref.incompatible.type                         | —          | ✓      |        |                   |         |         |                   |                   | —         | ✓      |        |                   |         |         |                   |                   | CE                |

### 3 IMPLEMENTATION STATUS

| 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.<br>reference  | ✗          | ✓      |        |                    |         |         |                    |                    | ✓          | ✓      |        |                    |         |         |                    |                    |                   |
| ↪ ...self                            | ✗          | ✓      |        |                    |         |         |                    |                    | ✓          | ✓      |        | MSan <sup>†</sup>  |         |         | MSan <sup>†</sup>  | MSan <sup>†</sup>  | CE                |
| ↪ ...member                          | ✓          | ✓      |        |                    |         |         |                    |                    | ✓          | ✓      |        | MSan <sup>†</sup>  |         |         | MSan <sup>†</sup>  | MSan <sup>†</sup>  | CE                |
| ↪ ...static                          | —          | ✓      |        | UBSan <sup>†</sup> |         |         | UBSan <sup>†</sup> | UBSan <sup>†</sup> | —          | ✓      |        |                    |         |         |                    |                    | CE                |
| class.dtor.no.longer.<br>exists      | ✗          | ✓      |        |                    |         |         |                    |                    | ✓          | ✓      |        |                    |         |         |                    |                    |                   |
| ↪ ...polymorphic                     | ✓          | ✓      |        | UBSan <sup>†</sup> |         |         | UBSan <sup>†</sup> | UBSan <sup>†</sup> | ✓          | ✓      |        | MSan <sup>†</sup>  |         |         | MSan <sup>†</sup>  | MSan <sup>†</sup>  | CE                |
| ↪ ...nonpolymorphic.<br>dynamic      | ✗<br>P3850 | ✓      |        |                    |         |         |                    |                    | ✓          | ✓      |        |                    |         |         |                    |                    | CE                |
| ↪ ...nonpolymorphic.<br>automatic    | ✓          | ✓      |        |                    |         |         |                    |                    | ✓          | ✓      |        |                    |         |         |                    |                    | CE                |
| ↪ ...nonpolymorphic.empty            | ✗          | ✓      |        |                    |         |         |                    |                    | ✓          | ✓      |        |                    |         |         |                    |                    | CE                |
| class.abstract.pure.<br>virtual      | ✗          | ✓      |        | P3850              | P3850   | P3850   | P3850              | P3850              | ✓          | ✓      |        | P3850              | P3850   | P3850   | P3850              | P3850              |                   |
| ↪ ...indirect                        | ✓          | ✓      |        | P3850              | P3850   | P3850   | P3850              | P3850              | ✓          | ✓      |        | P3850              | P3850   | P3850   | P3850              | P3850              | CE                |
| ↪ ...direct                          | ✗          | ✓      |        | P3850              | P3850   | P3850   | P3850              | P3850              | ✓          | ✓      |        | P3850              | P3850   | P3850   | P3850              | P3850              | CE                |
| class.base.init.mem.fun              | ✗          | ✓      | P3850  | P3850              | P3850   | P3850   | P3850              | P3850              | ✗          | ✓      | P3850  | P3850              | P3850   | P3850   | P3850              | P3850              |                   |
| ↪ ...polymorphic.virtual.<br>call    | ✗          | ✓      | P3850  | P3850              | P3850   | P3850   | P3850              | P3850              | ✓          | ✓      | P3850  | P3850              | P3850   | P3850   | P3850              | P3850              | CE                |
| ↪ ...polymorphic.typeid              | ✗          | ✓      | P3850  | P3850              | P3850   | P3850   | P3850              | P3850              | ✓          | ✓      | P3850  | P3850              | P3850   | P3850   | P3850              | P3850              | CE                |
| ↪ ...polymorphic.dynamic.<br>cast    | ✓          | ✓      | P3850  | P3850              | P3850   | P3850   | P3850              | P3850              | ✓          | ✓      | P3850  | P3850              | P3850   | P3850   | P3850              | P3850              | CE                |
| ↪ ...nonpolymorphic.mem.<br>init     | ✗          | ✓      | P3850  | P3850              | P3850   | P3850   | P3850              | P3850              | ✗          | ✓      | P3850  | P3850              | P3850   | P3850   | P3850              | P3850              | CE                |
| ↪ ...nonpolymorphic.<br>contract     | ✗          | ✓      | P3850  | P3850              | P3850   | P3850   | P3850              | P3850              | ✗<br>P3850 | ✓      | P3850  | P3850              | P3850   | P3850   | P3850              | P3850              | CE                |
| class.cdtor.before.ctor              | ✗          | ✓      |        |                    |         |         |                    |                    | ✗          | ✓      |        |                    |         |         |                    |                    |                   |
| ↪ ...polymorphic                     | ✓          | ✓      |        | UBSan <sup>†</sup> |         |         | UBSan <sup>†</sup> | UBSan <sup>†</sup> | ✓          | ✓      |        | UBSan <sup>†</sup> |         |         | UBSan <sup>†</sup> | UBSan <sup>†</sup> | CE                |
| ↪ ...nonpolymorphic.<br>virtual.base | ✗          | ✓      |        |                    |         |         |                    |                    | ✗<br>P3850 | ✓      |        | MSan <sup>†</sup>  |         |         | MSan <sup>†</sup>  | MSan <sup>†</sup>  | CE                |
| ↪ ...nonpolymorphic.<br>nonvirtual   | ✗          | ✓      |        |                    |         |         |                    |                    | ✗          | ✓      |        |                    |         |         |                    |                    | CE                |
| class.cdtor.after.dtor               | ✗          | ✓      |        |                    |         |         |                    |                    | ✗          | ✓      |        |                    |         |         |                    |                    |                   |
| ↪ ...polymorphic                     | ✓          | ✓      |        | UBSan <sup>†</sup> |         |         | UBSan <sup>†</sup> | UBSan <sup>†</sup> | ✓          | ✓      |        | MSan <sup>†</sup>  |         |         | MSan <sup>†</sup>  | MSan <sup>†</sup>  | CE                |
| ↪ ...nonpolymorphic.access           | ✓          | ✓      |        |                    |         |         |                    |                    | ✓          | ✓      |        | MSan <sup>†</sup>  |         |         | MSan <sup>†</sup>  | MSan <sup>†</sup>  | CE                |
| ↪ ...nonpolymorphic.<br>address      | ✗          | ✓      |        |                    |         |         |                    |                    | ✗          | ✓      |        |                    |         |         |                    |                    | CE                |
| class.cdtor.convert.<br>pointer      | ✗          | ✓      |        |                    |         |         |                    |                    | ✗          | ✓      |        |                    |         |         |                    |                    | CE                |

### 3 IMPLEMENTATION STATUS

| 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.ctor.form.pointer         | ✗         | ✓      |        |               |         |         |                  |                  | ✗         | ✓      |        |                    |         |         |                    |                    |                   |
| ↪ ...polymorphic                | ✗         | ✓      |        |               |         |         |                  |                  | ✗         | ✓      |        | UBSan <sup>†</sup> |         |         | UBSan <sup>†</sup> | UBSan <sup>†</sup> | CE                |
| ↪ ...nonpolymorphic             | ✗         | ✓      |        |               |         |         |                  |                  | ✗         | ✓      |        |                    |         |         |                    |                    | CE                |
| class.ctor.virtual.not.x        | ✗         | ✓      |        |               |         |         |                  |                  | ✗         | ✓      |        |                    |         |         |                    |                    | CE                |
| class.ctor.typeid               | ✗         | ✓      |        |               |         |         |                  |                  | ✗         | ✓      |        |                    |         |         |                    |                    | CE                |
| class.ctor.dynamic.cast         | ✗         | ✓      |        |               |         |         |                  |                  | ✗         | ✓      |        |                    |         |         |                    |                    | CE                |
| except.handle.handler.ctor.dtor | ✗         | ✓      | P3850  | P3850         | P3850   | P3850   | P3850            | P3850            | —         | ✓      | P3850  | P3850              | P3850   | P3850   | P3850              | P3850              |                   |
| ↪ ...ended                      | ✓         | ✓      | P3850  | P3850         | P3850   | P3850   | P3850            | P3850            | —         | ✓      | P3850  | P3850              | P3850   | P3850   | P3850              | P3850              | CE                |
|                                 |           |        |        |               |         |         |                  |                  |           |        |        | MSan <sup>†</sup>  |         |         | MSan <sup>†</sup>  | MSan <sup>†</sup>  |                   |
| ↪ ...initialized                | ✗         | ✓      | P3850  | P3850         | P3850   | P3850   | P3850            | P3850            | —         | ✓      | P3850  | P3850              | P3850   | P3850   | P3850              | P3850              | CE                |
| ↪ ...reference                  | ✗         | ✓      | P3850  | P3850         | P3850   | P3850   | P3850            | P3850            | —         | ✓      | P3850  | P3850              | P3850   | P3850   | P3850              | P3850              | CE                |

#### IV. Arithmetic

|                                     |   |   |       |       |       |       |       |       |   |   |       |       |       |       |       |       |     |
|-------------------------------------|---|---|-------|-------|-------|-------|-------|-------|---|---|-------|-------|-------|-------|-------|-------|-----|
| expr.expr.eval                      | ✓ | ✓ | P3850 | P3850 |       |       | P3850 | P3850 | ✓ | ✓ | P3850 | P3850 | P3850 | P3850 | P3850 | P3850 |     |
|                                     |   |   | UBSan | UBSan |       |       | UBSan | UBSan |   |   |       | UBSan |       |       | UBSan | UBSan |     |
| ↪ ...signed.integer.add             | ✓ | ✓ | P3850 | P3850 |       |       | P3850 | P3850 | ✓ | ✓ | P3850 | P3850 | P3850 | P3850 | P3850 | P3850 | CE  |
|                                     |   |   | UBSan | UBSan |       |       | UBSan | UBSan |   |   |       | UBSan |       |       | UBSan | UBSan |     |
| ↪ ...signed.integer.sub             | ✓ | ✓ | P3850 | P3850 |       |       | P3850 | P3850 | ✓ | ✓ | P3850 | P3850 | P3850 | P3850 | P3850 | P3850 | CE  |
|                                     |   |   | UBSan | UBSan |       |       | UBSan | UBSan |   |   |       | UBSan |       |       | UBSan | UBSan |     |
| ↪ ...signed.integer.mul             | ✓ | ✓ | P3850 | P3850 |       |       | P3850 | P3850 | ✓ | ✓ | P3850 | P3850 | P3850 | P3850 | P3850 | P3850 | CE  |
|                                     |   |   | UBSan | UBSan |       |       | UBSan | UBSan |   |   |       | UBSan |       |       | UBSan | UBSan |     |
| ↪ ...signed.integer.negate          | ✓ | ✓ | P3850 | P3850 |       |       | P3850 | P3850 | ✓ | ✓ | P3850 | P3850 | P3850 | P3850 | P3850 | P3850 | CE  |
|                                     |   |   | UBSan | UBSan |       |       | UBSan | UBSan |   |   |       | UBSan |       |       | UBSan | UBSan |     |
| conv.double.out.of.range            | — |   |       |       |       |       |       |       | — |   |       |       |       |       |       |       | N/A |
| conv.fpint.float.not.represented    | ✓ | ✓ | P3850 | P3850 | P3850 | P3850 | P3850 | P3850 | ✓ | ✓ | P3850 | P3850 | P3850 | P3850 | P3850 | P3850 | CE  |
|                                     |   |   | UBSan | UBSan |       |       | UBSan | UBSan |   |   |       | UBSan |       |       | UBSan | UBSan |     |
| conv.fpint.int.not.represented      | — |   |       |       |       |       |       |       | — |   |       |       |       |       |       |       | N/A |
| expr.static.cast.enum.outside.range | ✗ | ✓ | P3850 | P3850 | P3850 | P3850 | P3850 | P3850 | ✗ | ✓ | P3850 | P3850 | P3850 | P3850 | P3850 | P3850 |     |
|                                     |   |   |       |       |       |       |       |       |   |   |       | UBSan |       |       | UBSan | UBSan |     |
| ↪ ...integral                       | ✗ | ✓ | P3850 | P3850 | P3850 | P3850 | P3850 | P3850 | ✓ | ✓ | P3850 | P3850 | P3850 | P3850 | P3850 | P3850 | CE  |
|                                     |   |   |       |       |       |       |       |       |   |   |       | UBSan |       |       | UBSan | UBSan |     |
| ↪ ...floating                       | ✓ | ✓ | P3850 | P3850 | P3850 | P3850 | P3850 | P3850 | ✗ | ✓ | P3850 | P3850 | P3850 | P3850 | P3850 | P3850 | CE  |
|                                     |   |   |       |       |       |       |       |       |   |   |       | UBSan |       |       | UBSan | UBSan |     |
| expr.static.cast.fp.outside.range   | — |   |       |       |       |       |       |       | — |   |       |       |       |       |       |       | N/A |

### 3 IMPLEMENTATION STATUS

| 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                   | ✓         | ✓      | P3850  | P3850<br>UBSan<br>UBSan | P3850   | P3850   | P3850<br>UBSan<br>UBSan | P3850<br>UBSan<br>UBSan | ✓         | ✓      | P3850  | P3850<br>UBSan<br>UBSan | P3850   | P3850   | P3850<br>UBSan<br>UBSan | P3850<br>UBSan<br>UBSan |                   |
| ↪ ...int.div                           | ✓         | ✓      | P3850  | P3850<br>UBSan<br>UBSan | P3850   | P3850   | P3850<br>UBSan<br>UBSan | P3850<br>UBSan<br>UBSan | ✓         | ✓      | P3850  | P3850<br>UBSan<br>UBSan | P3850   | P3850   | P3850<br>UBSan<br>UBSan | P3850<br>UBSan<br>UBSan | CE                |
| ↪ ...int.mod                           | ✓         | ✓      | P3850  | P3850<br>UBSan<br>UBSan | P3850   | P3850   | P3850<br>UBSan<br>UBSan | P3850<br>UBSan<br>UBSan | ✓         | ✓      | P3850  | P3850<br>UBSan<br>UBSan | P3850   | P3850   | P3850<br>UBSan<br>UBSan | P3850<br>UBSan<br>UBSan | CE                |
| ↪ ...fp.div                            | ✓         | ✓      | P3850  | P3850<br>UBSan<br>UBSan | P3850   | P3850   | P3850<br>UBSan<br>UBSan | P3850<br>UBSan<br>UBSan | ✓         | ✓      | P3850  | P3850<br>UBSan<br>UBSan | P3850   | P3850   | P3850<br>UBSan<br>UBSan | P3850<br>UBSan<br>UBSan | CE                |
| expr.mul.representable.<br>type.result | ✓         | ✓      | P3850  | P3850<br>UBSan<br>UBSan | P3850   | P3850   | P3850<br>UBSan<br>UBSan | P3850<br>UBSan<br>UBSan | ✓         | ✓      | P3850  | P3850<br>UBSan<br>UBSan | P3850   | P3850   | P3850<br>UBSan<br>UBSan | P3850<br>UBSan<br>UBSan |                   |
| ↪ ...int.div                           | ✓         | ✓      | P3850  | P3850<br>UBSan<br>UBSan | P3850   | P3850   | P3850<br>UBSan<br>UBSan | P3850<br>UBSan<br>UBSan | ✓         | ✓      | P3850  | P3850<br>UBSan<br>UBSan | P3850   | P3850   | P3850<br>UBSan<br>UBSan | P3850<br>UBSan<br>UBSan | CE                |
| ↪ ...int.mod                           | ✓         | ✓      | P3850  | P3850<br>UBSan<br>UBSan | P3850   | P3850   | P3850<br>UBSan<br>UBSan | P3850<br>UBSan<br>UBSan | ✓         | ✓      | P3850  | P3850<br>UBSan<br>UBSan | P3850   | P3850   | P3850<br>UBSan<br>UBSan | P3850<br>UBSan<br>UBSan | CE                |
| expr.sub.pointers.rep                  | —         | ✓      |        |                         |         |         |                         |                         | —         | ✓      |        |                         |         |         |                         |                         | CE                |
| expr.shift.neg.and.width               | ✓         | ✓      | P3850  | P3850<br>UBSan<br>UBSan | P3850   | P3850   | P3850<br>UBSan<br>UBSan | P3850<br>UBSan<br>UBSan | ✓         | ✓      | P3850  | P3850<br>UBSan<br>UBSan | P3850   | P3850   | P3850<br>UBSan<br>UBSan | P3850<br>UBSan<br>UBSan |                   |
| ↪ ...shl                               | ✓         | ✓      | P3850  | P3850<br>UBSan<br>UBSan | P3850   | P3850   | P3850<br>UBSan<br>UBSan | P3850<br>UBSan<br>UBSan | ✓         | ✓      | P3850  | P3850<br>UBSan<br>UBSan | P3850   | P3850   | P3850<br>UBSan<br>UBSan | P3850<br>UBSan<br>UBSan | CE                |
| ↪ ...shr                               | ✓         | ✓      | P3850  | P3850<br>UBSan<br>UBSan | P3850   | P3850   | P3850<br>UBSan<br>UBSan | P3850<br>UBSan<br>UBSan | ✓         | ✓      | P3850  | P3850<br>UBSan<br>UBSan | P3850   | P3850   | P3850<br>UBSan<br>UBSan | P3850<br>UBSan<br>UBSan | CE                |

#### V. Threading

|                      |   |   |  |                   |  |  |                   |                   |   |   |  |                   |  |  |                   |                   |    |
|----------------------|---|---|--|-------------------|--|--|-------------------|-------------------|---|---|--|-------------------|--|--|-------------------|-------------------|----|
| intro.races.data     | — | ✓ |  | TSan <sup>†</sup> |  |  | TSan <sup>†</sup> | TSan <sup>†</sup> | — | ✓ |  | TSan <sup>†</sup> |  |  | TSan <sup>†</sup> | TSan <sup>†</sup> | CE |
| intro.progress.stops | — | ✓ |  |                   |  |  |                   |                   | — | ✓ |  |                   |  |  |                   |                   |    |

#### VI. Sequencing

|                                              |   |   |  |  |  |  |  |  |   |   |  |  |  |  |  |  |    |
|----------------------------------------------|---|---|--|--|--|--|--|--|---|---|--|--|--|--|--|--|----|
| intro.execution.<br>unsequenced.modification | ✗ | ✓ |  |  |  |  |  |  | ✗ | ✓ |  |  |  |  |  |  | CE |
|----------------------------------------------|---|---|--|--|--|--|--|--|---|---|--|--|--|--|--|--|----|

#### VII. Assumptions

|                           |   |   |       |                         |       |       |                         |                         |   |   |       |                         |       |       |                         |                         |    |
|---------------------------|---|---|-------|-------------------------|-------|-------|-------------------------|-------------------------|---|---|-------|-------------------------|-------|-------|-------------------------|-------------------------|----|
| dcl.attr.assume.false     | ✗ | ✓ | P3850 |                         |       |       |                         |                         | ✗ | ✓ | P3850 |                         |       |       |                         |                         |    |
| ↪ ...pure                 | ✓ | ✓ | P3850 | P3850<br>UBSan<br>UBSan | P3850 | P3850 | P3850<br>UBSan<br>UBSan | P3850<br>UBSan<br>UBSan | ✓ | ✓ | P3850 | P3850<br>UBSan<br>UBSan | P3850 | P3850 | P3850<br>UBSan<br>UBSan | P3850<br>UBSan<br>UBSan | CE |
| ↪ ...nonpure.opaque       | ✓ | ✓ | P3850 |                         |       |       |                         |                         | ✗ | ✓ | P3850 |                         |       |       |                         |                         | CE |
| ↪ ...nonpure.side.effects | ✗ | ✓ | P3850 |                         |       |       |                         |                         | ✗ | ✓ | P3850 |                         |       |       |                         |                         | CE |

#### VIII. Control Flow

|                                              |   |   |  |  |  |  |  |  |   |   |  |  |  |  |  |  |    |
|----------------------------------------------|---|---|--|--|--|--|--|--|---|---|--|--|--|--|--|--|----|
| basic.start.main.exit.<br>during.destruction | — | ✓ |  |  |  |  |  |  | — | ✓ |  |  |  |  |  |  | CE |
|----------------------------------------------|---|---|--|--|--|--|--|--|---|---|--|--|--|--|--|--|----|

### 3 IMPLEMENTATION STATUS

| 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                |
| stmt.return.flow.off                   | ✓         | ✓      | P3850          | P3850<br>UBSan | P3850   | P3850   | P3850<br>UBSan   | P3850<br>UBSan   | ✓         | ✓      | P3850  | P3850<br>UBSan | P3850   | P3850   | P3850<br>UBSan   | P3850<br>UBSan   | CE                |
| stmt.dcl.local.static.init.recursive   | —         | ✓      |                |                |         |         |                  |                  | —         | ✓      |        |                |         |         |                  |                  | CE                |
| dcl.attr.noreturn.eventually.returns   | ✓         | ✓      | P3850<br>UBSan | P3850<br>UBSan | P3850   | P3850   | P3850<br>UBSan   | P3850<br>UBSan   | ✗         | ✓      | P3850  | P3850<br>UBSan | P3850   | P3850   | P3850<br>UBSan   | P3850            |                   |
| ↪ ...direct                            | ✓         | ✓      | P3850<br>UBSan | P3850<br>UBSan | P3850   | P3850   | P3850<br>UBSan   | P3850<br>UBSan   | ✗         | ✓      | P3850  | P3850<br>UBSan | P3850   | P3850   | P3850<br>UBSan   | P3850<br>UBSan   | CE                |
| ↪ ...indirect                          | ✓         | ✓      | P3850<br>UBSan | P3850<br>UBSan | P3850   | P3850   | P3850<br>UBSan   | P3850<br>UBSan   | ✗         | ✓      | P3850  | P3850<br>UBSan | P3850   | P3850   | P3850<br>UBSan   | P3850<br>UBSan   | CE                |

#### IX. Replacement Functions

|                                    |   |   |                |                |       |       |                |                |   |   |       |                |       |       |                |                |    |
|------------------------------------|---|---|----------------|----------------|-------|-------|----------------|----------------|---|---|-------|----------------|-------|-------|----------------|----------------|----|
| basic.stc.alloc.dealloc.constraint | — | ✓ |                |                |       |       |                |                | — | ✓ |       | UBSan          |       |       | UBSan          | UBSan          | CE |
| expr.new.non.allocating.null       | ✓ | ✓ | P3850<br>UBSan | P3850<br>UBSan | P3850 | P3850 | P3850<br>UBSan | P3850<br>UBSan | ✓ | ✓ | P3850 | P3850<br>UBSan | P3850 | P3850 | P3850<br>UBSan | P3850<br>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  | — | ✓ |       | ASan <sup>†</sup> |       |       | ASan <sup>†</sup><br>ASan <sup>†</sup> | ASan <sup>†</sup> | — | ✓ |       | ASan <sup>†</sup> |       |       | ASan <sup>†</sup><br>ASan <sup>†</sup> | ASan <sup>†</sup> | CE |
| dcl.fct.def.coroutine.destroy.not.suspended | — | ✓ |       | ASan <sup>†</sup> |       |       | ASan <sup>†</sup><br>ASan <sup>†</sup> | ASan <sup>†</sup> | — | ✓ |       | ASan <sup>†</sup> |       |       | ASan <sup>†</sup><br>ASan <sup>†</sup> | ASan <sup>†</sup> | CE |

### 3.1 Implementation Coverage Charts

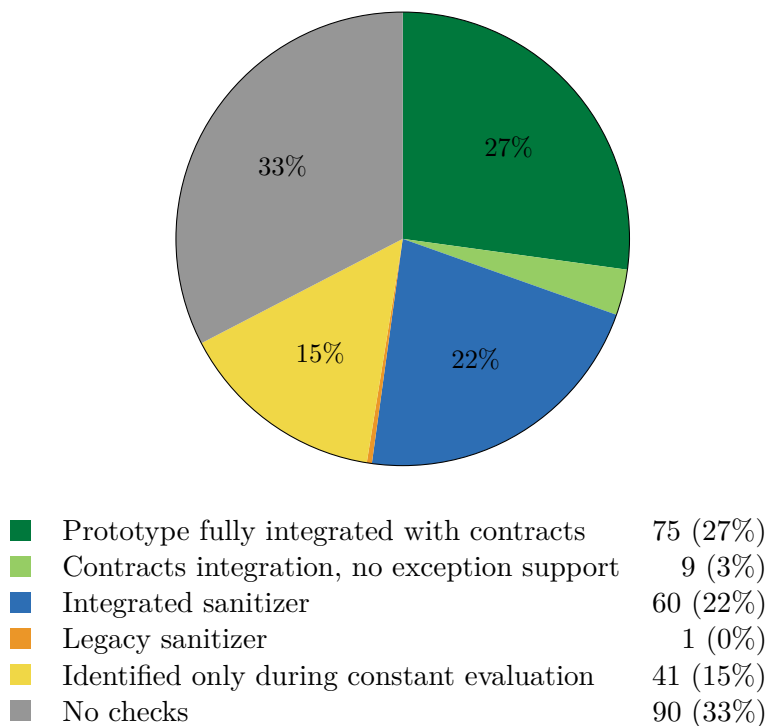
Now, of course, nothing is more fun than charts. It’s important to note that the data underlying these charts is, inherently, not weighted in a manner that makes it particularly applicable to reality. Real exploitable problems in production systems are far, far more likely to be the result of integer 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 through the contract-violation handler, as [P3100R9] proposes.

In the below charts, all subcategories are weighted equally and both open-source compilers are weighted equally.

#### UB Categories by Checking Support

GCC and Clang counted separately (138 leaf categories and subgroups  $\times$  2 compilers)



### 4 Foundational Wording Changes

The primary proposal of [P3100R9] 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 Before All Undefined Behavior

The currently proposed wording strategy for introducing an implicit contract assertion before all undefined behavior is to apply a guarding contract assertion, as an implicit precondition assertion, to every evaluation that has undefined behavior. The wording changes needed for this approach are presented in this section.

### 3 Terms and definitions

[intro.defs]

#### 3.66 undefined behavior

[defns.undefined]

Modify definition 3.66[defns.undefined]:

3.66 [defns.undefined]  
**undefined behavior**  
 behavior for which this document imposes no requirements

[*Note to entry 1*: Undefined behavior may be expected when this document omits any explicit definition of behavior or when a program uses an incorrect construct or invalid data. Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a *diagnostic message* (defns.diagnostic)), to terminating a translation or execution (with the issuance of a diagnostic message). Many incorrect program constructs do not engender undefined behavior; they are required to be diagnosed. Evaluation of a constant expression (7.7.2[expr.const.core]) never exhibits behavior explicitly specified as undefined in Clause 4[intro] through Clause 15[cpp]. — *end note*]

[*Note to entry 1+a*: There is a guarding contract assertion (6.11.2+a[basic.contract.implicit]) sequenced before any occurrence of undefined behavior. — *end note*]

### 6.11 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:

<sup>1</sup> ~~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-specifiers*, *postcondition-specifiers* (9.4.1[dcl.contract.func]), and *assertion-statements* (8.9[stmt.contract.assert]). Implicit contract assertions are introduced as parts of other language constructs.

<sup>2</sup> 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*]~~

<sup>2+a</sup> 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:

<sup>1</sup> 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*.

<sup>2</sup> ~~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]

<sup>3</sup> *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.

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

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

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

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

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

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

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

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

[*Example 1*:

```
struct S {
 mutable int g = 5;
} s;
```

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

— end example]

— end note]

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

7 A *contract violation* occurs when

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

(7.1) — *B* is false,

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

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

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

### 6.11.2 + a Implicit contract assertions

[[basic.contract.implicit](#)]

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

#### 6.11.2+a Implicit contract assertions

[[basic.contract.implicit](#)]

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

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

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

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

#### Note

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

## 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]

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                               |
|------------------------------|---------------------------------------|
| <code>pre</code>             | A precondition assertion              |
| <code>post</code>            | A postcondition assertion             |
| <code>assert</code>          | An <i>assertion-statement</i>         |
| <u><code>implicit</code></u> | <u>An implicit contract assertion</u> |

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

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

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

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

| Name                       | Meaning                           |
|----------------------------|-----------------------------------|
| <code>ignore</code>        | Ignore evaluation semantic        |
| <code>observe</code>       | Observe evaluation semantic       |
| <code>enforce</code>       | Enforce evaluation semantic       |
| <code>quick_enforce</code> | Quick-enforce evaluation semantic |
| <code>assume</code>        | Assume evaluation semantic        |

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

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

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

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

| Name                              | Meaning                                                                                                                                                         |
|-----------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>predicate_false</code>      | The predicate of the contract assertion evaluated to false <del>or</del> <sub>2</sub> would have evaluated to false, <u>or its condition is not satisfied</u> . |
| <code>evaluation_exception</code> | An uncaught exception occurred during evaluation of the contract assertion.                                                                                     |

## 4.2 Implicit Assertions as Part of Undefined Behavior

This alternative instead makes the implicit contract assertion part of the definition of undefined behavior, separating undefined behavior into a guarding contract assertion followed by unconstrained behavior.

This was the original strategy proposed in [P3100R7]; [P3100R9] replaced it with the approach in Section 4.

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

3 Terms and definitions [intro.defs]

3.65 + *a* unconstrained behavior [defs.unconstrained]

Add a new definition [3.65+a](#)[defs.unconstrained] after 3.65[defs.unblock]

[3.65 + a](#) [defs.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* ([defs.diagnostic](#))), to terminating a translation or execution (with the issuance of a diagnostic message). — *end note*]

### 3.66 undefined behavior

[[defs.undefined](#)]

Modify definition 3.66[[defs.undefined](#)]:

3.66

[[defs.undefined](#)]

#### undefined behavior

behavior ~~for which this document~~that begins with an implicit contract assertion imposes no requirementsthat 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* ([defs.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 unconstrained 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<sup>3</sup> 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.

---

<sup>3</sup> “Correct execution” can include undefined behavior, unconstrained behavior, and erroneous behavior, depending on the data being processed; see Clause 3[intro.defs] and 4.1.2[intro.abstract].

### 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 (6.11.2+a[basic.contract.implicit]), which is an implicit contract assertion that they do not occur. If control continues normally after a guarding contract assertion, the behavior is unconstrained.

<sup>4</sup> 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 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-specifiers*, *postcondition-specifiers* (9.4.1[`dcl.contract.func`]), and *assertion-statements* (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*]

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

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

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

[*Example 1*:

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

— *end example*]

— *end note*]

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

7 A *contract violation* occurs when

- (7.a) — the contract assertion has a condition that is not satisfied,
- (7.1) — *B* is false,
- (7.2) — the evaluation of the predicate exits via an exception, or
- (7.3) — the evaluation of the predicate is performed in a context that is manifestly constant-evaluated (7.7.7[*expr.const.defns*]) and the predicate is not a core constant expression.

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

### 6.11.2 + *a* Implicit contract assertions

[**basic.contract.implicit**]

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

#### 6.11.2+*a* Implicit contract assertions

[**basic.contract.implicit**]

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

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

3 [*Note 1*: 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 behaves as specified. — *end note*]

#### Note

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

## 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 (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]

### 4.3 Wording Changes Common to Both Strategies

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]

17.3 Implementation properties [support.limits]

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

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

2

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

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

```
#define __cpp_lib_adaptor_iterator_pair_constructor 202106L // also in <stack>, <queue>
#define __cpp_lib_addressof_constexpr 201603L // freestanding, also in <memory>
#define __cpp_lib_algorithm_default_value_type 202603L
// also in <algorithm>, <ranges>, <string>, <deque>, <list>, <forward_list>, <vector>, <memory>
#define __cpp_lib_algorithm_iterator_requirements 202207L
// also in <algorithm>, <numeric>, <memory>
#define __cpp_lib_aligned_accessor 202411L // freestanding, also in <mdspan>
#define __cpp_lib_allocate_at_least 202302L // also in <memory>
#define __cpp_lib_allocator_traits_is_always_equal 201411L
// freestanding, also in <memory>, <scoped_allocator>, <string>, <deque>, <forward_list>, <list>,
// <vector>, <map>, <set>, <unordered_map>, <unordered_set>
#define __cpp_lib_any 201606L // also in <any>
#define __cpp_lib_apply 202603L
// freestanding, also in <tuple>, <type_traits>, <meta>
#define __cpp_lib_array_constexpr 201811L
// freestanding, also in <iterator>, <array>
#define __cpp_lib_as_const 201510L // freestanding, also in <utility>
#define __cpp_lib_associative_heterogeneous_erasure 202110L
// also in <map>, <set>, <unordered_map>, <unordered_set>
#define __cpp_lib_associative_heterogeneous_insertion 202306L
// also in <map>, <set>, <unordered_map>, <unordered_set>
#define __cpp_lib_assume_aligned 201811L // freestanding, also in <memory>
#define __cpp_lib_atomic_flag_test 201907L // freestanding, also in <atomic>
#define __cpp_lib_atomic_float 201711L // freestanding, also in <atomic>
#define __cpp_lib_atomic_is_always_lock_free 201603L // freestanding, also in <atomic>
#define __cpp_lib_atomic_lock_free_type_aliases 201907L // also in <atomic>
#define __cpp_lib_atomic_min_max 202506L // freestanding, also in <atomic>
#define __cpp_lib_atomic_reductions 202506L // freestanding, also in <atomic>
#define __cpp_lib_atomic_ref 202603L // freestanding, also in <atomic>
#define __cpp_lib_atomic_shared_ptr 201711L // also in <memory>
#define __cpp_lib_atomic_value_initialization 201911L // freestanding, also in <atomic>, <memory>
#define __cpp_lib_atomic_wait 201907L // freestanding, also in <atomic>
#define __cpp_lib_barrier 202302L // also in <barrier>
#define __cpp_lib_bind_back 202306L // freestanding, also in <functional>
#define __cpp_lib_bind_front 202306L // freestanding, also in <functional>
#define __cpp_lib_bit_cast 201806L // freestanding, also in <bit>
#define __cpp_lib_bitops 202607L // freestanding, also in <bit>
#define __cpp_lib_bitset 202306L // also in <bitset>
#define __cpp_lib_bool_constant 201505L // freestanding, also in <type_traits>
#define __cpp_lib_bounded_array_traits 201902L // freestanding, also in <type_traits>
#define __cpp_lib_boyer_moore_searcher 201603L // also in <functional>
#define __cpp_lib_byte 201603L // freestanding, also in <cstdint>
#define __cpp_lib_byteswap 202110L // freestanding, also in <bit>
#define __cpp_lib_char8_t 201907L
// freestanding, also in <atomic>, <filesystem>, <iosfwd>, <istream>, <limits>, <locale>, <ostream>,
// <string>, <string_view>
#define __cpp_lib_chrono 202306L // also in <chrono>
```

## Wording Changes Common to Both Strategies

```
#define __cpp_lib_chrono_udls 201304L // also in <chrono>
#define __cpp_lib_clamp 201603L // freestanding, also in <algorithm>
#define __cpp_lib_common_reference 202302L // freestanding, also in <type_traits>
#define __cpp_lib_common_reference_wrapper 202302L // freestanding, also in <functional>
#define __cpp_lib_complex_udls 201309L // also in <complex>
#define __cpp_lib_concepts 202207L
// freestanding, also in <concepts>, <compare>
#define __cpp_lib_constant_wrapper 202606L // freestanding, also in <utility>
#define __cpp_lib_constexpr_algorithms 202306L // also in <algorithm>, <utility>
#define __cpp_lib_constexpr_atomic 202411L // also in <atomic>
#define __cpp_lib_constexpr_bitset 202207L // also in <bitset>
#define __cpp_lib_constexpr_charconv 202207L // freestanding, also in <charconv>
#define __cpp_lib_constexpr_cmath 202306L // also in <cmath>, <cstdlib>
#define __cpp_lib_constexpr_complex 202306L // also in <complex>
#define __cpp_lib_constexpr_deque 202502L // also in <deque>
#define __cpp_lib_constexpr_dynamic_alloc 201907L // also in <memory>
#define __cpp_lib_constexpr_exceptions 202502L
// also in <exception>, <stdexcept>, <expected>, <optional>, <variant>, and <format>
#define __cpp_lib_constexpr_flat_map 202502L // also in <flat_map>
#define __cpp_lib_constexpr_flat_set 202502L // also in <flat_set>
#define __cpp_lib_constexpr_format 202511L // also in <format>
#define __cpp_lib_constexpr_forward_list 202502L // also in <forward_list>
#define __cpp_lib_constexpr_functional 201907L // freestanding, also in <functional>
#define __cpp_lib_constexpr_inplace_vector 202502L // also in <inplace_vector>
#define __cpp_lib_constexpr_iterator 201811L // freestanding, also in <iterator>
#define __cpp_lib_constexpr_list 202502L // also in <list>
#define __cpp_lib_constexpr_map 202502L // also in <map>
#define __cpp_lib_constexpr_memory 202506L // freestanding, also in <memory>
#define __cpp_lib_constexpr_new 202406L // freestanding, also in <new>
#define __cpp_lib_constexpr_numeric 201911L // freestanding, also in <numeric>
#define __cpp_lib_constexpr_queue 202502L // also in <queue>
#define __cpp_lib_constexpr_set 202502L // also in <set>
#define __cpp_lib_constexpr_stack 202502L // also in <stack>
#define __cpp_lib_constexpr_string 202511L // also in <string>
#define __cpp_lib_constexpr_string_view 201811L // also in <string_view>
#define __cpp_lib_constexpr_tuple 201811L // freestanding, also in <tuple>
#define __cpp_lib_constexpr_typeinfo 202106L // freestanding, also in <typeinfo>
#define __cpp_lib_constexpr_unordered_map 202502L // also in <unordered_map>
#define __cpp_lib_constexpr_unordered_set 202502L // also in <unordered_set>
#define __cpp_lib_constexpr_utility 201811L // freestanding, also in <utility>
#define __cpp_lib_constexpr_vector 201907L // also in <vector>
#define __cpp_lib_constrained_equality 202411L // freestanding,
// also in <utility>, <tuple>, <optional>, <variant>, <expected>
#define __cpp_lib_containers_ranges 202202L
// also in <vector>, <list>, <forward_list>, <map>, <set>, <unordered_map>, <unordered_set>,
// <deque>, <queue>, <stack>, <string>
#define __cpp_lib_contracts 202502L // freestanding, also in <contracts>
#define __cpp_lib_contracts_implicit XXXXXXL // freestanding, also in <contracts>←
#define __cpp_lib_copiable_function 202306L // also in <functional>
#define __cpp_lib_coroutine 201902L // freestanding, also in <coroutine>
#define __cpp_lib_counting_scope 202506L // also in <execution>
#define __cpp_lib_debugging 202403L // freestanding, also in <debugging>
#define __cpp_lib_define_static 202506L // freestanding, also in <meta>
#define __cpp_lib_destroying_delete 201806L // freestanding, also in <new>
#define __cpp_lib_enable_shared_from_this 201603L // also in <memory>
#define __cpp_lib_endian 201907L // freestanding, also in <bit>
#define __cpp_lib_erase_if 202002L
// also in <string>, <deque>, <forward_list>, <list>, <vector>, <map>, <set>, <unordered_map>,
// <unordered_set>
#define __cpp_lib_exception_ptr_cast 202603L // also in <exception>
#define __cpp_lib_exchange_function 201304L // freestanding, also in <utility>
#define __cpp_lib_execution 201902L // also in <execution>
#define __cpp_lib_expected 202606L // also in <expected>
#define __cpp_lib_filesystem 201703L // also in <filesystem>
#define __cpp_lib_flat_map 202511L // also in <flat_map>
#define __cpp_lib_flat_set 202511L // also in <flat_set>
#define __cpp_lib_format 202603L // also in <format>
#define __cpp_lib_format_path 202506L // also in <filesystem>
#define __cpp_lib_format_ranges 202207L // also in <format>
#define __cpp_lib_format_uchar 202311L // also in <format>
#define __cpp_lib_formatters 202302L // also in <stacktrace>, <thread>
```

## Wording Changes Common to Both Strategies

```
#define __cpp_lib_forward_like 202207L // freestanding, also in <utility>
#define __cpp_lib_freestanding_algorithm 202502L // freestanding, also in <algorithm>
#define __cpp_lib_freestanding_array 202311L // freestanding, also in <array>
#define __cpp_lib_freestanding_char_traits 202306L // freestanding, also in <string>
#define __cpp_lib_freestanding_charconv 202306L // freestanding, also in <charconv>
#define __cpp_lib_freestanding_cstdlib 202306L // freestanding, also in <cstdlib>, <cmath>
#define __cpp_lib_freestanding_cstring 202311L // freestanding, also in <cstring>
#define __cpp_lib_freestanding_cwchar 202306L // freestanding, also in <cwchar>
#define __cpp_lib_freestanding_errc 202306L
 // freestanding, also in <cerrno>, <system_error>
#define __cpp_lib_freestanding_execution 202502L // freestanding, also in <execution>
#define __cpp_lib_freestanding_expected 202311L // freestanding, also in <expected>
#define __cpp_lib_freestanding_feature_test_macros 202306L // freestanding
#define __cpp_lib_freestanding_functional 202306L // freestanding, also in <functional>
#define __cpp_lib_freestanding_iterator 202306L // freestanding, also in <iterator>
#define __cpp_lib_freestanding_mdspan 202311L // freestanding, also in <mdspan>
#define __cpp_lib_freestanding_memory 202502L // freestanding, also in <memory>
#define __cpp_lib_freestanding_numeric 202502L // freestanding, also in <numeric>
#define __cpp_lib_freestanding_operator_new see below // freestanding, also in <new>
#define __cpp_lib_freestanding_optional 202506L // freestanding, also in <optional>
#define __cpp_lib_freestanding_random 202502L // freestanding, also in <random>
#define __cpp_lib_freestanding_ranges 202306L // freestanding, also in <ranges>
#define __cpp_lib_freestanding_ratio 202306L // freestanding, also in <ratio>
#define __cpp_lib_freestanding_string_view 202311L // freestanding, also in <string_view>
#define __cpp_lib_freestanding_tuple 202306L // freestanding, also in <tuple>
#define __cpp_lib_freestanding_utility 202306L // freestanding, also in <utility>
#define __cpp_lib_freestanding_variant 202311L // freestanding, also in <variant>
#define __cpp_lib_fstream_native_handle 202306L // also in <fstream>
#define __cpp_lib_function_ref 202604L // freestanding, also in <functional>
#define __cpp_lib_gcd_lcm 201606L // freestanding, also in <numeric>
#define __cpp_lib_generator 202207L // also in <generator>
#define __cpp_lib_generic_associative_lookup 201304L // also in <map>, <set>
#define __cpp_lib_generic_unordered_lookup 201811L
 // also in <unordered_map>, <unordered_set>
#define __cpp_lib_hardware_interference_size 201703L // freestanding, also in <new>
#define __cpp_lib_has_unique_object_representations 201606L // freestanding, also in <type_traits>
#define __cpp_lib_hazard_pointer 202606L // also in <hazard_pointer>
#define __cpp_lib_hive 202502L // also in <hive>
#define __cpp_lib_hypot 201603L // also in <cmath>
#define __cpp_lib_incomplete_container_elements 201505L
 // also in <forward_list>, <list>, <vector>
#define __cpp_lib_indirect 202502L // also in <memory>
#define __cpp_lib_initializer_list 202511L
 // freestanding, also in <initializer_list>
#define __cpp_lib_inplace_vector 202603L // also in <inplace_vector>
#define __cpp_lib_int_pow2 202002L // freestanding, also in <bit>
#define __cpp_lib_integer_comparison_functions 202002L // freestanding, also in <utility>
#define __cpp_lib_integer_sequence 202511L // freestanding, also in <utility>
#define __cpp_lib_integral_constant_callable 201304L // freestanding, also in <type_traits>
#define __cpp_lib_interpolate 201902L // also in <cmath>, <numeric>
#define __cpp_lib_invoke 201411L // freestanding, also in <functional>
#define __cpp_lib_invoke_r 202106L // freestanding, also in <functional>
#define __cpp_lib_ios_noreplace 202207L // also in <ios>
#define __cpp_lib_is_aggregate 201703L // freestanding, also in <type_traits>
#define __cpp_lib_is_constant_evaluated 201811L // freestanding, also in <type_traits>
#define __cpp_lib_is_final 201402L // freestanding, also in <type_traits>
#define __cpp_lib_is_implicit_lifetime 202302L // freestanding, also in <type_traits>
#define __cpp_lib_is_invocable 201703L // freestanding, also in <type_traits>
#define __cpp_lib_is_layout_compatible 201907L // freestanding, also in <type_traits>
#define __cpp_lib_is_nothrow_convertible 201806L // freestanding, also in <type_traits>
#define __cpp_lib_is_null_pointer 201309L // freestanding, also in <type_traits>
#define __cpp_lib_is_pointer_interconvertible 201907L // freestanding, also in <type_traits>
#define __cpp_lib_is_scoped_enum 202011L // freestanding, also in <type_traits>
#define __cpp_lib_is_structural 202603L
 // freestanding, also in <meta> and <type_traits>
#define __cpp_lib_is_sufficiently_aligned 202411L // freestanding, also in <memory>
#define __cpp_lib_is_swappable 201603L // freestanding, also in <type_traits>
#define __cpp_lib_is_virtual_base_of 202406L // freestanding, also in <type_traits>
#define __cpp_lib_is_within_lifetime 202603L // freestanding, also in <type_traits>
#define __cpp_lib_jthread 201911L // also in <stop_token>, <thread>
#define __cpp_lib_latch 201907L // also in <latch>
```

## Wording Changes Common to Both Strategies

```
#define __cpp_lib_laundry 201606L // freestanding, also in <new>
#define __cpp_lib_linalg 202511L // also in <linalg>
#define __cpp_lib_list_remove_return_type 201806L // also in <forward_list>, <list>
#define __cpp_lib_logical_traits 201510L // freestanding, also in <type_traits>
#define __cpp_lib_make_from_tuple 201606L // freestanding, also in <tuple>
#define __cpp_lib_make_reverse_iterator 201402L // freestanding, also in <iterator>
#define __cpp_lib_make_unique 201304L // also in <memory>
#define __cpp_lib_map_lookup 202606L
// also in <map>, <unordered_map>, <flat_map>
#define __cpp_lib_map_try_emplace 201411L // also in <map>
#define __cpp_lib_math_constants 201907L // also in <numbers>
#define __cpp_lib_math_special_functions 201603L // also in <cmath>
#define __cpp_lib_mdspan 202406L // freestanding, also in <mdspan>
#define __cpp_lib_mdspan_copy 202606L // also in <mdspan>
#define __cpp_lib_memory_resource 201603L // also in <memory_resource>
#define __cpp_lib_modules 202207L // freestanding
#define __cpp_lib_move_iterator_concept 202207L // freestanding, also in <iterator>
#define __cpp_lib_move_only_function 202110L // also in <functional>
#define __cpp_lib_node_extract 201606L
// also in <map>, <set>, <unordered_map>, <unordered_set>
#define __cpp_lib_nonmember_container_access 201411L
// freestanding, also in <array>, <deque>, <forward_list>, <iterator>, <list>, <map>, <regex>, <set>,
// <string>, <unordered_map>, <unordered_set>, <vector>
#define __cpp_lib_not_fn 202306L // freestanding, also in <functional>
#define __cpp_lib_null_iterators 201304L // freestanding, also in <iterator>
#define __cpp_lib_observable_checkpoint 202506L // freestanding, also in <utility>
#define __cpp_lib_optional 202506L // also in <optional>
#define __cpp_lib_optional_range_support 202406L // freestanding, also in <optional>
#define __cpp_lib_out_ptr 202311L // freestanding, also in <memory>
#define __cpp_lib_parallel_algorithm 202506L
// also in <algorithm>, <numeric>, <memory>
#define __cpp_lib_parallel_scheduler 202506L // also in <execution>
#define __cpp_lib_philox_engine 202406L // also in <random>
#define __cpp_lib_pointer_tag_pair 202606L // freestanding, also in <memory>
#define __cpp_lib_polymorphic 202502L // also in <memory>
#define __cpp_lib_polymorphic_allocator 201902L // also in <memory_resource>
#define __cpp_lib_print 202406L // also in <print>, <ostream>
#define __cpp_lib_quoted_string_io 201304L // also in <iomanip>
#define __cpp_lib_ranges 202406L
// also in <algorithm>, <functional>, <iterator>, <memory>, <ranges>
#define __cpp_lib_ranges_as_const 202311L // freestanding, also in <ranges>
#define __cpp_lib_ranges_as_input 202502L // freestanding, also in <ranges>
#define __cpp_lib_ranges_as_rvalue 202207L // freestanding, also in <ranges>
#define __cpp_lib_ranges_cache_latest 202411L // freestanding, also in <ranges>
#define __cpp_lib_ranges_cartesian_product 202207L // freestanding, also in <ranges>
#define __cpp_lib_ranges_chunk 202202L // freestanding, also in <ranges>
#define __cpp_lib_ranges_chunk_by 202202L // freestanding, also in <ranges>
#define __cpp_lib_ranges_concat 202403L // freestanding, also in <ranges>
#define __cpp_lib_ranges_contains 202207L // freestanding, also in <algorithm>
#define __cpp_lib_ranges_enumerate 202302L // freestanding, also in <ranges>
#define __cpp_lib_ranges_filter 202603L // freestanding, also in <ranges>
#define __cpp_lib_ranges_find_last 202207L // freestanding, also in <algorithm>
#define __cpp_lib_ranges_fold 202207L // freestanding, also in <algorithm>
#define __cpp_lib_ranges_generate_random 202403L // also in <random>
#define __cpp_lib_ranges_indices 202506L // also in <ranges>
#define __cpp_lib_ranges_iota 202202L // freestanding, also in <numeric>
#define __cpp_lib_ranges_join_with 202202L // freestanding, also in <ranges>
#define __cpp_lib_ranges_repeat 202207L // freestanding, also in <ranges>
#define __cpp_lib_ranges_reserve_hint 202502L // also in <ranges>
#define __cpp_lib_ranges_slide 202202L // freestanding, also in <ranges>
#define __cpp_lib_ranges_starts_ends_with 202106L // freestanding, also in <algorithm>
#define __cpp_lib_ranges_stride 202207L // freestanding, also in <ranges>
#define __cpp_lib_ranges_to_container 202202L // freestanding, also in <ranges>
#define __cpp_lib_ranges_zip 202110L
// freestanding, also in <ranges>, <tuple>, <utility>
#define __cpp_lib_ratio 202306L // freestanding, also in <ratio>
#define __cpp_lib_raw_memory_algorithms 202411L // also in <memory>
#define __cpp_lib_rcu 202306L // also in <rcu>
#define __cpp_lib_reference_from_temporary 202202L // freestanding, also in <type_traits>
#define __cpp_lib_reference_wrapper 202403L // freestanding, also in <functional>
#define __cpp_lib_reflection 202603L // also in <meta>
```

## Wording Changes Common to Both Strategies

```
#define __cpp_lib_remove_cvref 201711L // freestanding, also in <type_traits>
#define __cpp_lib_replaceable_contract_violation_handler see below
// freestanding, also in <contracts>
#define __cpp_lib_result_of_sfinae 201210L
// freestanding, also in <functional>, <type_traits>
#define __cpp_lib_robust_nonmodifying_seq_ops 201304L // freestanding, also in <algorithm>
#define __cpp_lib_sample 201603L // freestanding, also in <algorithm>
#define __cpp_lib_saturation_arithmetic 202603L // freestanding, also in <numeric>
#define __cpp_lib_scoped_lock 201703L // also in <mutex>
#define __cpp_lib_semaphore 201907L // also in <semaphore>
#define __cpp_lib_senders 202506L // also in <execution>
#define __cpp_lib_shared_mutex 201505L // also in <shared_mutex>
#define __cpp_lib_shared_ptr_arrays 201707L // also in <memory>
#define __cpp_lib_shared_ptr_weak_type 201606L // also in <memory>
#define __cpp_lib_shared_timed_mutex 201402L // also in <shared_mutex>
#define __cpp_lib_shift 202202L // also in <algorithm>
#define __cpp_lib_simd 202606L // also in <simd>
#define __cpp_lib_simd_bitops 202607L // also in <simd>
#define __cpp_lib_simd_complex 202502L // also in <simd>
#define __cpp_lib_simd_permutations 202506L // also in <simd>
#define __cpp_lib_smart_ptr_for_overwrite 202002L // also in <memory>
#define __cpp_lib_smart_ptr_owner_equality 202306L // also in <memory>
#define __cpp_lib_source_location 201907L // freestanding, also in <source_location>
#define __cpp_lib_span 202311L // freestanding, also in
#define __cpp_lib_spanstream 202106L // also in <iosfwd>, <spanstream>
#define __cpp_lib_ssize 201902L // freestanding, also in <iterator>
#define __cpp_lib_sstream_from_string_view 202306L // also in <sstream>
#define __cpp_lib_stacktrace 202011L // also in <stacktrace>
#define __cpp_lib_start_lifetime 202603L // freestanding, also in <memory>
#define __cpp_lib_start_lifetime_as 202207L // freestanding, also in <memory>
#define __cpp_lib_starts_ends_with 201711L // also in <string>, <string_view>
#define __cpp_lib_stdatomic_h 202011L // also in <stdatomic.h>
#define __cpp_lib_stdbit_h 202603L // also in <stdbit.h>
#define __cpp_lib_stdckdint_h 202603L // also in <stdckdint.h>
#define __cpp_lib_string_contains 202011L // also in <string>, <string_view>
#define __cpp_lib_string_resize_and_overwrite 202110L // also in <string>
#define __cpp_lib_string_subview 202506L // also in <string>, <string_view>
#define __cpp_lib_string_udls 201304L // also in <string>
#define __cpp_lib_string_view 202403L // also in <string>, <string_view>
#define __cpp_lib_submdspan 202603L // freestanding, also in <mdspan>
#define __cpp_lib_syncbuf 201803L // also in <iosfwd>, <syncstream>
#define __cpp_lib_task 202506L // also in <execution>
#define __cpp_lib_text_encoding 202306L // also in <text_encoding>
#define __cpp_lib_thread_attributes 202606L // also in <thread>
#define __cpp_lib_three_way_comparison 201907L // freestanding, also in <compare>
#define __cpp_lib_to_address 201711L // freestanding, also in <memory>
#define __cpp_lib_to_array 201907L // freestanding, also in <array>
#define __cpp_lib_to_chars 202606L // also in <charconv>
#define __cpp_lib_to_string 202306L // also in <string>
#define __cpp_lib_to_underlying 202102L // freestanding, also in <utility>
#define __cpp_lib_transformation_trait_aliases 201304L // freestanding, also in <type_traits>
#define __cpp_lib_transparent_operators 201510L
// freestanding, also in <memory>, <functional>
#define __cpp_lib_tuple_element_t 201402L // freestanding, also in <tuple>
#define __cpp_lib_tuple_like 202311L
// also in <utility>, <tuple>, <map>, <unordered_map>
#define __cpp_lib_tuples_by_type 201304L // freestanding, also in <utility>, <tuple>
#define __cpp_lib_type_identity 201806L // freestanding, also in <type_traits>
#define __cpp_lib_type_order 202506L // also in <compare>
#define __cpp_lib_type_trait_variable_templates 201510L // freestanding, also in <type_traits>
#define __cpp_lib_uncaught_exceptions 201411L // freestanding, also in <exception>
#define __cpp_lib_unordered_map_try_emplace 201411L // also in <unordered_map>
#define __cpp_lib_unreachable 202202L // freestanding, also in <utility>
#define __cpp_lib_unwrap_ref 201811L // freestanding, also in <type_traits>
#define __cpp_lib_valarray 202511L // also in <valarray>
#define __cpp_lib_variant 202306L // also in <variant>
#define __cpp_lib_view_interface 202606L // also in <ranges>
#define __cpp_lib_void_t 201411L // freestanding, also in <type_traits>
```

## 17.10 Contract-violation handling

[support.contract]

## 17.10.1 Header &lt;contracts&gt; synopsis

[contracts.syn]

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

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

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

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

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

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

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

 see below ~contract_violation();

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

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

## 17.10.2 Enumerations

[support.contract.enum]

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

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

### 17.10.3 Class `contract_violation`

[support.contract.violation]

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

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

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

## F Core undefined behavior

[ub]

### F.1 General

[ub.general]

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

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

- (1.1) — is undefined
- (1.2) — the behavior is undefined
- (1.3) — the behavior of the program is undefined
- (1.4) — has undefined behavior
- (1.5) — have undefined behavior
- (1.6) — result has undefined behavior
- (1.7) — results in undefined behavior
- (1.7+a) — an implicit precondition assertion (6.11.2+a[basic.contract.implicit])

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

## 5 Conclusion

EWG asked for a line-item review of [P3100R9]. There have also been significant questions about the implementability and meaning of the changes proposed by [P3100R9].

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 [P3100R9].

## 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>
- [P3100R9] Timur Doumler and Joshua Berne, “A framework for systematically addressing undefined behaviour in the C++ Standard”, 2026  
<http://wg21.link/P3100R9>
- [P3400R5] Joshua Berne, “Controlling Contract-Assertion Properties”, 2026  
<http://wg21.link/P3400R5>
- [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>
- [P4284R1] Joshua Berne, “Addenda to the Undefined Behavior and IFNDR Annexes”, 2026  
<http://wg21.link/P4284R1>
- [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>