

# Addenda to the Undefined Behavior and IFNDR Annexes

Document #: P4284R1  
Date: 2026-10-23  
Project: Programming Language C++  
Audience: CWG  
Reply-to: Joshua Berne <[jberne4@bloomberg.net](mailto:jberne4@bloomberg.net)>

## Abstract

[[P3596R3](#)] introduced a pair of new annexes to the C++ Standard containing full catalogues of undefined behavior and ill-formed, no-diagnostic required constructs. That paper was current and complete when it was approved to be merged into the draft Standard in Brno, but it was also instantly out of date due to other papers that landed concurrently at the same meeting. In addition, new eyes on the contents of the annexes have brought up some minor issues and suggestions for improved readability and formatting. This paper captures a round of changes to the annexes that will bring the Standard to a state where ongoing maintenance of the annexes can be reliably expected to happen in individual papers that touch on the relevant items directly.

## Contents

|          |                        |                           |
|----------|------------------------|---------------------------|
| <b>1</b> | <b>Introduction</b>    | <b><a href="#">2</a></b>  |
| <b>2</b> | <b>Wording Changes</b> | <b><a href="#">5</a></b>  |
| <b>3</b> | <b>Conclusion</b>      | <b><a href="#">36</a></b> |

# Revision History

## Revision 1

- Added 1 new UB entry and 12 new IFNDR entries.
- Updated the modifications 12 UB entries and 2 IFNDR entries.
- Made new modifications to 15 UB entries and 1 IFNDR entry.
- Rebased on more recent draft

## Revision 0

- Original version of the paper

## 1 Introduction

A few items have been brought up directly during the final merging of [P3596R3] into the C++ draft or since that merge completed:

- Shafik Yaghmour pointed out during the review ([P3596R3] PR Comment) that two cases of undefined behavior introduced by contracts had been missed from [class.base.init].

To handle this we have added examples of both of these cases to [class.base.init.mem.fun].

- A. Jiang pointed out ([P3596R3] PR Comment) that the example in [ifnldr:expr.prim.req.always.sub.fail] references a case where an earlier core issue ([CWG2392]) made this case no longer IFNDR.

To address this, we have changed the example to use a different dependent expression that is always invalid (`new decltype((void)T{})`).

- A. Jiang pointed out ([P3596R3] PR Comment) that the example in [ifnldr:basic.def.odr.definition.matches] is now well-formed thanks to [CWG2300].

To address this, we have changed the example to use an inline function with a lambda in a default argument (the example from [basic.def.odr]).

- Thomas Köppe, during the merging of [P3596R3], suggested that many of the examples in the annexes relied too heavily on putting expository information in comments in the code examples. To improve the readability of the examples, we have adjusted a number of entries to put more content in the prose around the example and less in the comments.
- Jan Schultke pointed out (Proposed CWG issue) that the paper [P3899R3] made non-arithmetic expressions such as pointer difference with arithmetic results that are not representable into defined behavior (with no corresponding definition). This was a normative bug, but does mean that the non-arithmetic results of pointer difference are a new category of UB distinct from [ub:expr.expr.eval].

To address this we have foreshadowed the normative reinstatement of the conversion by making what was previously a note about conversion to `std::ptrdiff_t` into a normative statement and adding a corresponding ub entry in [ub:expr.sub.pointers.rep].

This change is also covered by core issue [CWG3232], whose normative updates are included here and whose annex entry is here with minor edits.

- Jan Schultke noted that a number of cases were using `int`, `float`, or `double` and then showing examples that were only relevant on platforms where those types had very specific properties (such as being 32-bit). These examples have been updated to use various fixed-width types from `<stdint>` and `<stdfloat>`, such as `std::int32_t`, `std::float32_t`, and `std::float64_t`.
- A bug in `[ifndr:basic.def.odr.definition.matches]` was noted by Lénárd Szolnoki (see [Editorial Issue 9238](#)), where the IFNDR case was being qualified to a definable item that is *not* “an inline or templated function”. The correct qualification is the opposite — the definable item must be an inline or templated function or variable, or something that is not a function or variable.
- An editorial issues submission by Yin Xinyu (see [Editorial Issue 9284](#)) noted that there was no annex entry for the undefined behavior in `[conv.array]`. We have added that missing entry (`[ub:conv.array.unsimilar.type]`).
- Two missing IFNDR entries about function-contract assertion matching on redeclarations were noted in another editorial issue by Yin Xinyu ([Editorial Issue 9345](#)). We have added the two missing entries:

- `[ifndr:dcl.contract.func.mismatched.redeclaration]`
- `[ifndr:dcl.contract.func.mismatched.lambda]`

The same search resulted in two additional missing IFNDR entries being uncovered:

- `[ifndr:module.unit.conflicting.module.partitions]`
- `[ifndr:except.spec.mismatched.explicit.instantiation]`

- Yin Xinyu also noted that `[class.dtor]/16` seems to introduce an undefined behavior that is not tracked in the annex. This sentence is, however, elaborating the implications of applying “the usual rules for member functions” and is not itself introducing a new form of undefined behavior.
- Yin Xinyu pointed out a lack of an IFNDR entry for `[temp.res.general]/6` which we have added as one entry for each bullet of that paragraph:
  - `[ifndr:temp.res.general.no.valid.specialization]`
  - `[ifndr:temp.res.general.no.valid.expansion]`
  - `[ifndr:temp.res.general.no.valid.default.arg]`
  - `[ifndr:temp.res.general.no.valid.alias]`
  - `[ifndr:temp.res.general.no.valid.constraint.check]`
  - `[ifndr:temp.res.general.variadic.empty.pack]`
  - `[ifndr:temp.res.general.hypothetical.ill.formed]`
  - `[ifndr:temp.res.general.hypothetical.different.meaning]`

In addition, a number of papers landed in Brno, concurrently with [P3596R3], that added, removed, or modified the parts of the standard containing UB and IFNDR entries.

- CWG Motion 4 — [P3899R3] (Clarify the behavior of floating-point overflow) deletes the blanket rule in [expr.pre] that made any result not mathematically defined or not representable undefined, replacing it with a narrow rule scoped to arithmetic expressions — so an overflow yielding a representable infinity or NaN is now well-defined, and division by zero becomes the only arithmetic operation with undefined behavior.

To cover the change in this wording we have expanded the examples to include cases of UB for a `float` type that does not have NaN or infinities, all in [ub.expr.expr.eval].

- CWG Motion 7 — [P2434R5] (Nondeterministic pointer provenance) reworks pointer provenance in the core language. It adds a new undefined behavior in [basic.compound] — an evaluation that produces, or gives an object, a pointer value to or past an object *O* and that happens before the beginning of the storage duration for *O* — and it makes the integer-to-pointer conversion in [expr.reinterpret.cast] conditionally undefined when no representable value would give the program defined behavior.

To address these changes, we added the following ub entries:

- [ub:basic.compound.pointer.before.storage.duration]
- [ub:expr.reinterpret.cast.invalid.pointer.value]

One change in [cstdio.syn] does not lead to a new annex entry because it is in the library, not the core wording.

- CWG Motion 8 — [P3347R6] (Invalid Pointer Operations) rewrites [basic.compound] paragraph 4 so that indirection through an invalid pointer remains undefined while most other operations become implementation-defined or well-defined, and it deletes [conv.lval] paragraph 3.3. The set of undefined behavior is essentially unchanged, but the wording must be considered for impact on the annex entry.

This motion affected the wording around [ub:basic.compound.invalid.pointer]. This annex entry previously also covered freeing memory that has already been freed, but that is not the result of this particular wording so we have made the example slightly more specific.

- CWG Motion 10 — [P3950R1] (`return_value` & `return_void` Are Not Mutually Exclusive) retains the undefined behavior of flowing off the end of a coroutine but rewords its trigger in [stmt.return.coroutine] in terms of whether overload resolution for `p.return_void()` succeeds.

The cases where [ub:stmt.return.coroutine.flow.off] might be triggered are changed very slightly because a promise type might have a `return_void` that is constrained and not usable. We have clarified the wording to better match the criteria (`p.return_void()` is valid as opposed to talking about a coroutine that can return void).

- CWG Motion 14 — [P3424R2] (Deallocation Functions with Throwing Exception Specification Are Ill-formed) makes it ill-formed to declare a deallocation function with a potentially-throwing exception specification removing the possibility of such a function throwing.

[ub:basic.stc.alloc.dealloc.throw] was removed entirely when this paper was merged.

- CWG Motion 17 — [P4101R1] (Consteval-only Values for C++26) replaces the type-based model of consteval-only values with a value-based one. It adds a new IFNDR case in [expr.const.const] — an expression that odr-uses a variable declaring or referring to an immediate object outside an immediate-function context, with a diagnostic required only under a reachability condition.

To address this paper, we added the following IFNDR entry:

- [ifnldr:expr.const.const.immediate.var.def]

- CWG Motion 18 — [P2414R12] (Pointer lifetime-end zap proposed solutions) adds a new undefined behavior in [conv.lval] for an lvalue-to-rvalue conversion when the bits of the value representation are not valid for the object’s type, and it removes the undefined behavior previously reachable through volatile reads and volatile assignments of a lifetime-end pointer value, which now yield a well-defined prospective pointer value.

Though this paper changes a paragraph that defines some undefined behavior, the UB itself is not changed.

- Minor improvements and defect fixes were also made in the wording of the following undefined behavior entries:

- [ub:basic.align.object.alignment]
- [ub:lifetime.outside.pointer.delete]
- [ub:lifetime.outside.pointer.virtual]
- [ub:basic.stc.alloc.dealloc.constraint]
- [ub:conv.double.out.of.range]
- [ub:conv.fpint.float.not.represented]
- [ub:conv.fpint.float.not.represented]
- [ub:conv.ptr.virtual.base]
- [ub:expr.dynamic.cast.pointer.lifetime]
- [ub:expr.dynamic.cast.glvalue.lifetime]
- [ub:expr.mptr.oper.not.contain.member]
- [ub:stmt.return.coroutine.flow.off]

All of the above changes are represented in the wording changes below.

## 2 Wording Changes

These wording changes are relative to the C++29 draft sources with git hash [47974a2f](#), last modified on Wed, 14 Oct 2026 19:31:54 +0200.

## Modified Section Contents

|                 |                                                       |                                                            |           |
|-----------------|-------------------------------------------------------|------------------------------------------------------------|-----------|
| <b>6</b>        | <b>Basics</b>                                         | <b>[basic]</b>                                             | <b>8</b>  |
| 6.9             | Types                                                 | [basic.types]                                              | 8         |
| 6.9.5           | Compound types                                        | [basic.compound]                                           | 8         |
| <b>7</b>        | <b>Expressions</b>                                    | <b>[expr]</b>                                              | <b>8</b>  |
| 7.3             | Standard conversions                                  | [conv]                                                     | 8         |
| 7.3.3           | Array-to-pointer conversion                           | [conv.array]                                               | 8         |
| 7.6             | Compound expressions                                  | [expr.compound]                                            | 9         |
| 7.6.1           | Postfix expressions                                   | [expr.post]                                                | 9         |
| 7.6.1.10        | Reinterpret cast                                      | [expr.reinterpret.cast]                                    | 9         |
| 7.6.6           | Additive operators                                    | [expr.add]                                                 | 9         |
| 7.7             | Constant evaluation                                   | [expr.const]                                               | 9         |
| 7.7.3           | Constant expressions                                  | [expr.const.const]                                         | 9         |
| <b>9</b>        | <b>Declarations</b>                                   | <b>[dcl]</b>                                               | <b>10</b> |
| 9.4             | Function contract specifiers                          | [dcl.contract]                                             | 10        |
| 9.4.1           | General                                               | [dcl.contract.func]                                        | 10        |
| <b>10</b>       | <b>Modules</b>                                        | <b>[module]</b>                                            | <b>11</b> |
| 10.1            | Module units and purviews                             | [module.unit]                                              | 11        |
| <b>13</b>       | <b>Templates</b>                                      | <b>[temp]</b>                                              | <b>11</b> |
| 13.8            | Name resolution                                       | [temp.res]                                                 | 11        |
| 13.8.1          | General                                               | [temp.res.general]                                         | 11        |
| <b>14</b>       | <b>Exception handling</b>                             | <b>[except]</b>                                            | <b>13</b> |
| 14.5            | Exception specifications                              | [except.spec]                                              | 13        |
| <b>F</b>        | <b>Core undefined behavior</b>                        | <b>[ub]</b>                                                | <b>13</b> |
| F.2             | Clause 6[basic]: Basics                               | [ub.basic]                                                 | 13        |
| F.2.3           | basic.align.object.alignment                          | [ub:basic.align.object.alignment]                          | 14        |
| F.2.4           | lifetime.outside.pointer.delete                       | [ub:lifetime.outside.pointer.delete]                       | 14        |
| F.2.6           | lifetime.outside.pointer.virtual                      | [ub:lifetime.outside.pointer.virtual]                      | 14        |
| F.2.15          | basic.stc.alloc.dealloc.constraint                    | [ub:basic.stc.alloc.dealloc.constraint]                    | 15        |
| <u>F.2.16+a</u> | <u>basic.compound.pointer.before.storage.duration</u> | <u>[ub:basic.compound.pointer.before.storage.duration]</u> | 15        |
| F.2.17          | basic.compound.invalid.pointer                        | [ub:basic.compound.invalid.pointer]                        | 16        |
| F.2.19          | intro.races.data                                      | [ub:intro.races.data]                                      | 16        |
| F.2.20          | intro.progress.stops                                  | [ub:intro.progress.stops]                                  | 16        |
| F.2.21          | basic.start.main.exit.during.destruction              | [ub:basic.start.main.exit.during.destruction]              | 17        |
| F.2.22          | basic.start.term.use.after.destruction                | [ub:basic.start.term.use.after.destruction]                | 17        |
| F.3             | Clause 7[expr]: Expressions                           | [ub.expr]                                                  | 18        |
| F.3.1           | expr.expr.eval                                        | [ub:expr.expr.eval]                                        | 18        |
| <u>F.3.5+a</u>  | <u>conv.array.unsimilar.type</u>                      | <u>[ub:conv.array.unsimilar.type]</u>                      | 18        |

|          |                                                             |                                                           |           |
|----------|-------------------------------------------------------------|-----------------------------------------------------------|-----------|
| F.3.6    | conv.double.out.of.range                                    | [ub:conv.double.out.of.range]                             | 19        |
| F.3.7    | conv.fpint.float.not.represented                            | [ub:conv.fpint.float.not.represented]                     | 19        |
| F.3.9    | conv.ptr.virtual.base                                       | [ub:conv.ptr.virtual.base]                                | 20        |
| F.3.13   | expr.dynamic.cast.pointer.lifetime                          | [ub:expr.dynamic.cast.pointer.lifetime]                   | 20        |
| F.3.14   | expr.dynamic.cast.glvalue.lifetime                          | [ub:expr.dynamic.cast.glvalue.lifetime]                   | 20        |
|          | <u>F.3.19+a expr.reinterpret.cast.invalid.pointer.value</u> | <u>[ub:expr.reinterpret.cast.invalid.pointer.value]</u>   | 21        |
| F.3.22   | expr.delete.mismatch                                        | [ub:expr.delete.mismatch]                                 | 21        |
| F.3.23   | expr.delete.array.mismatch                                  | [ub:expr.delete.array.mismatch]                           | 21        |
| F.3.26   | expr.mptr.oper.not.contain.member                           | [ub:expr.mptr.oper.not.contain.member]                    | 22        |
| F.3.28   | expr.mul.div.by.zero                                        | [ub:expr.mul.div.by.zero]                                 | 22        |
| F.3.29   | expr.mul.representable.type.result                          | [ub:expr.mul.representable.type.result]                   | 22        |
| F.3.30   | expr.add.out.of.bounds                                      | [ub:expr.add.out.of.bounds]                               | 23        |
|          | <u>F.3.30+a expr.sub.pointers.rep</u>                       | <u>[ub:expr.sub.pointers.rep]</u>                         | 23        |
| F.3.33   | expr.shift.neg.and.width                                    | [ub:expr.shift.neg.and.width]                             | 24        |
| F.4      | Clause 8[stmt]: Statements                                  | [ub.stmt]                                                 | 24        |
| F.4.2    | stmt.return.coroutine.flow.off                              | [ub:stmt.return.coroutine.flow.off]                       | 24        |
| F.5      | Clause 9[dcl]: Declarations                                 | [ub.dcl]                                                  | 25        |
| F.5.5    | dcl.ref.uninitialized.reference                             | [ub:dcl.ref.uninitialized.reference]                      | 25        |
| F.5.6    | dcl.fct.def.coroutine.resume.not.suspended                  | [ub:dcl.fct.def.coroutine.resume.not.suspended]           | 25        |
| F.5.7    | dcl.fct.def.coroutine.destroy.not.suspended                 | [ub:dcl.fct.def.coroutine.destroy.not.suspended]          | 26        |
| F.6      | Clause 11[class]: Classes                                   | [ub.class]                                                | 27        |
| F.6.3    | class.base.init.mem.fun                                     | [ub:class.base.init.mem.fun]                              | 27        |
| F.6.6    | class.cctor.convert.pointer                                 | [ub:class.cctor.convert.pointer]                          | 28        |
| <b>G</b> | <b>Ill-formed, no diagnostic required</b>                   | <b>[ifndr]</b>                                            | <b>29</b> |
| G.3      | Clause 6[basic]: Basics                                     | [ifndr.basic]                                             | 29        |
| G.3.4    | basic.def.odr.maximum.one.def                               | [ifndr:basic.def.odr.maximum.one.def]                     | 29        |
| G.3.5    | basic.def.odr.definition.matches                            | [ifndr:basic.def.odr.definition.matches]                  | 29        |
| G.4      | Clause 7[expr]: Expressions                                 | [ifndr.expr]                                              | 30        |
| G.4.1    | expr.prim.req.always.sub.fail                               | [ifndr:expr.prim.req.always.sub.fail]                     | 30        |
|          | <u>G.4.1+a expr.const.const.immediate.var.def</u>           | <u>[ifndr:expr.const.const.immediate.var.def]</u>         | 30        |
| G.6      | Clause 9[dcl]: Declarations                                 | [ifndr.dcl]                                               | 31        |
|          | <u>G.6.3+a dcl.contract.func.mismatched.redeclaration</u>   | <u>[ifndr:dcl.contract.func.mismatched.redeclaration]</u> | 31        |
|          | <u>G.6.4+a dcl.contract.func.mismatched.lambda</u>          | <u>[ifndr:dcl.contract.func.mismatched.lambda]</u>        | 31        |
| G.7      | Clause 10[module]: Modules                                  | [ifndr.module]                                            | 32        |
|          | <u>G.7.2+a module.unit.conflicting.module.partitions</u>    | <u>[ifndr:module.unit.conflicting.module.partitions]</u>  | 32        |
| G.10     | Clause 13[temp]: Templates                                  | [ifndr.temp]                                              | 32        |
|          | <u>G.10.8+a temp.res.general.no.valid.specialization</u>    | <u>[ifndr:temp.res.general.no.valid.specialization]</u>   | 32        |

|                          |                                                                 |                                                                         |    |
|--------------------------|-----------------------------------------------------------------|-------------------------------------------------------------------------|----|
| <a href="#">G.10.8+b</a> | <a href="#">temp.res.general.no.valid.expansion</a>             | <a href="#">[ifndr:temp.res.general.no.valid.expansion]</a>             | 32 |
| <a href="#">G.10.8+c</a> | <a href="#">temp.res.general.no.valid.default.arg</a>           | <a href="#">[ifndr:temp.res.general.no.valid.default.arg]</a>           | 33 |
| <a href="#">G.10.8+d</a> | <a href="#">temp.res.general.no.valid.alias</a>                 | <a href="#">[ifndr:temp.res.general.no.valid.alias]</a>                 | 33 |
| <a href="#">G.10.8+e</a> | <a href="#">temp.res.general.no.valid.constraint.check</a>      | <a href="#">[ifndr:temp.res.general.no.valid.constraint.check]</a>      | 34 |
| <a href="#">G.10.8+f</a> | <a href="#">temp.res.general.variadic.empty.pack</a>            | <a href="#">[ifndr:temp.res.general.variadic.empty.pack]</a>            | 34 |
| <a href="#">G.10.8+g</a> | <a href="#">temp.res.general.hypothetical.ill.formed</a>        | <a href="#">[ifndr:temp.res.general.hypothetical.ill.formed]</a>        | 34 |
| <a href="#">G.10.8+h</a> | <a href="#">temp.res.general.hypothetical.different.meaning</a> | <a href="#">[ifndr:temp.res.general.hypothetical.different.meaning]</a> | 35 |
| <a href="#">G.10+a</a>   | <a href="#">Clause 14[except]: Exception handling</a>           | <a href="#">[ifndr.except]</a>                                          | 35 |
| <a href="#">G.10+a.1</a> | <a href="#">except.spec.mismatched.explicit.instantiation</a>   | <a href="#">[ifndr:except.spec.mismatched.explicit.instantiation]</a>   | 35 |

## Modifications

**6 Basics** [basic]

**6.9 Types** [basic.types]

**6.9.5 Compound types** [basic.compound]

Modify section 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.2.16+a\[ubx:basic.compound.pointer.before.storage.duration\]](#)).

[*Note 5*: 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*]

**7 Expressions** [expr]

**7.3 Standard conversions** [conv]

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

Modify section 7.3.3[conv.array], paragraph 1:

- <sup>1</sup> 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.3.5+a\[ubx:conv.array.unsimilar.type\]](#)).

## 7.6 Compound expressions

[expr.compound]

### 7.6.1 Postfix expressions

[expr.post]

#### 7.6.1.10 Reinterpret cast

[expr.reinterpret.cast]

Modify section 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.3.19+a[ubx:expr.reinterpret.cast.invalid.pointer.value]).

[Note 4: 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 5: It can be an invalid pointer value. — end note]

## 7.6.6 Additive operators

[expr.add]

Modify 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*. ~~[Note 2: If the that value *i* - *j* is not in the range of representable values of type `std::ptrdiff_t`, the behavior is undefined (7.1[expr.pre]) (F.3.30+a[ubx:expr.sub.pointers.rep]). — end note]~~

(5.3) — Otherwise, the behavior is undefined (F.3.31[ubx:expr.add.sub.diff.pointers]).

## 7.7 Constant evaluation

[expr.const]

### 7.7.3 Constant expressions

[expr.const.const]

Modify section 7.7.3[expr.const.const], paragraph 3:

- 3 Every immediate object shall be

(3.1) — the object associated with a `constexpr` variable or a subobject thereof,

(3.2) — a template parameter object (13.2[temp.param]) or a subobject thereof, or

(3.3) — an object whose lifetime begins and ends during the evaluation of a core constant expression.

[Example 1:

```
constexpr int plus1(int x) { return x + 1; }
```

```

template <auto V> struct C {};

auto a = plus1;           // error: immediate object not associated with constexpr variable
constexpr auto b = plus1; // OK
auto c = C<plus1>();      // OK

auto d = ~int;            // error: immediate object not associated with constexpr variable
auto e = C<~char>();      // OK

```

— *end example*]

Each expression  $E$  that odr-uses a variable that declares or refers to an immediate object shall be in an immediate function context; a diagnostic is required only if either

(3.1) — the innermost declaration that contains  $E$  or

(3.2) — the defining declaration of the variable

is reachable from the other (G.4.1+a[ifndrx:expr.const.const.immediate.var.def]).

## 9 Declarations

[dcl]

### 9.4 Function contract specifiers

[dcl.contract]

#### 9.4.1 General

[dcl.contract.func]

Modify section 9.4.1[dcl.contract.func], paragraphs 4-5:

4 A declaration  $D$  of a function or function template  $f$  that is not a first declaration shall have either no *function-contract-specifier-seq* or the same *function-contract-specifier-seq* (see below) as any first declaration  $F$  reachable from  $D$ . If  $D$  and  $F$  are in different translation units, a diagnostic is required only if  $D$  is attached to a named module (G.6.3+a[ifndrx:dcl.contract.func.mismatched.redeclaration]). If a declaration  $F_1$  is a first declaration of  $f$  in one translation unit and a declaration  $F_2$  is a first declaration of  $f$  in another translation unit,  $F_1$  and  $F_2$  shall specify the same *function-contract-specifier-seq*, no diagnostic required (G.6.4[ifndrx:dcl.contract.func.mismatched.contract.specifiers]).

5 A *function-contract-specifier-seq*  $S_1$  is the same as a *function-contract-specifier-seq*  $S_2$  if  $S_1$  and  $S_2$  consist of the same *function-contract-specifiers* in the same order. A *function-contract-specifier*  $C_1$  on a function declaration  $D_1$  is the same as a *function-contract-specifier*  $C_2$  on a function declaration  $D_2$  if

(5.1) — their predicates  $P_1$  and  $P_2$  would satisfy the one-definition rule (6.3[basic.def.odr]) if placed in function definitions on the declarations  $D_1$  and  $D_2$ , respectively, except for

- renaming of the parameters of  $f$ ,
- renaming of template parameters of a template enclosing  $f$ , and
- renaming of the result binding (9.4.2[dcl.contract.res]), if any,

and, if  $D_1$  and  $D_2$  are in different translation units, corresponding entities defined within each predicate behave as if there is a single entity with a single definition, and

(5.2) — both  $C_1$  and  $C_2$  specify a *result-name-introducer* or neither do.

If this condition is not met solely due to the comparison of two *lambda-expressions* that are contained within  $P_1$  and  $P_2$ , no diagnostic is required ([G.6.4+a\[ifndrx:dcl.contract.func.mismatched.lambda\]](#)).

[*Note 3*: Equivalent *function-contract-specifier-seqs* apply to all uses and definitions of a function across all translation units. — *end note*]

[*Example 1*:

```
bool b1, b2;

void f() pre (b1) pre ([]{ return b2; }());
void f();           // OK, function-contract-specifiers omitted
void f() pre (b1) pre ([]{ return b2; }()); // ill-formed, no diagnostic required: closures have
                                           // different types
void f() pre (b1);   // error: function-contract-specifiers only partially repeated

int g() post(r : b1);
int g() post(b1);    // error: mismatched result-name-introducer presence

namespace N {
    void h() pre (b1);
    bool b1;
    void h() pre (b1); // error: function-contract-specifiers differ according to
                       // the one-definition rule (6.3[basic.def.odr]).
}
```

— *end example*]

## 10 Modules

[**module**]

### 10.1 Module units and purviews

[**module.unit**]

Modify subclause 10.1[**module.unit**], paragraph 3:

- <sup>3</sup> A *module partition* is a module unit whose *module-declaration* contains a *module-partition*. A named module shall not contain multiple module partitions with the same *module-partition*. All module partitions of a module that are module interface units shall be directly or indirectly exported by the primary module interface unit (10.3[**module.import**]). No diagnostic is required for a violation of these rules ([G.7.2+a\[ifndrx:module.unit.conflicting.module.partitions\]](#)) (G.7.3[ifndrx:module.unit.unexported.module.partition]).

[*Note 1*: Module partitions can be imported only by other module units in the same module. The division of a module into module units is not visible outside the module. — *end note*]

## 13 Templates

[**temp**]

### 13.8 Name resolution

[**temp.res**]

#### 13.8.1 General

[**temp.res.general**]

Modify section 13.8.1[**temp.res.general**], paragraph 6:

- <sup>6</sup> The validity of a templated entity may be checked prior to any instantiation.

[*Note 3*: Knowing which names are type names allows the syntax of every template to be checked in this way. — *end note*]

The program is ill-formed, no diagnostic required, if

- (6.1) — no valid specialization, ignoring *static\_assert-declarations* that fail (9.1[dcl.pre]), can be generated for a templated entity or a substatement of a `constexpr` if statement (8.5.2[stmt.if]) within a templated entity and the innermost enclosing template is not instantiated ([G.10.8+a\[ifndrx:temp.res.general.no.valid.specialization\]](#)), or
- (6.2) — no valid specialization, ignoring *static\_assert-declarations* that fail, can be generated for the *compound-statement* of an *expansion-statement* and there is no instantiation of it ([G.10.8+b\[ifndrx:temp.res.general.no.valid.expansion\]](#)), or
- (6.3) — no valid specialization, ignoring *static\_assert-declarations* that fail, can be generated for a default *template-argument* and the default *template-argument* is not used in any instantiation ([G.10.8+c\[ifndrx:temp.res.general.no.valid.default.arg\]](#)), or
- (6.4) — no specialization of an alias template (13.7.8[temp.alias]) is valid and no specialization of the alias template is named in the program ([G.10.8+d\[ifndrx:temp.res.general.no.valid.alias\]](#)), or
- (6.5) — any *constraint-expression* in the program, introduced or otherwise, has (in its normal form) an atomic constraint *A* where no satisfaction check of *A* could be well-formed and no satisfaction check of *A* is performed ([G.10.8+e\[ifndrx:temp.res.general.no.valid.constraint.check\]](#)), or
- (6.6) — every valid specialization of a variadic template requires an empty template parameter pack ([G.10.8+f\[ifndrx:temp.res.general.variadic.empty.pack\]](#)), or
- (6.7) — a hypothetical instantiation of a templated entity immediately following its definition would be ill-formed due to a construct (other than a *static\_assert-declaration* that fails) that does not depend on a template parameter ([G.10.8+g\[ifndrx:temp.res.general.hypothetical.ill.formed\]](#)), or
- (6.8) — the interpretation of such a construct in the hypothetical instantiation is different from the interpretation of the corresponding construct in any actual instantiation of the templated entity ([G.10.8+h\[ifndrx:temp.res.general.hypothetical.different.meaning\]](#)).

[Note 4: This can happen in situations including the following:

- (6.1) — a type used in a non-dependent name is incomplete at the point at which a template is defined but is complete at the point at which an instantiation is performed, or
- (6.2) — lookup for a name in the template definition found a *using-declaration*, but the lookup in the corresponding scope in the instantiation does not find any declarations because the *using-declaration* was a pack expansion and the corresponding pack is empty, or
- (6.3) — an instantiation uses a default argument or default template argument that had not been defined at the point at which the template was defined, or
- (6.4) — constant expression evaluation (7.7.2[expr.const.core]) within the template instantiation uses
  - the value of a `const` object of integral or unscoped enumeration type or
  - the value of a `constexpr` object or
  - the value of a reference or
  - the definition of a `constexpr` function,
 and that entity was not defined when the template was defined, or

- (6.5) — a class template specialization or variable template specialization that is specified by a non-dependent *simple-template-id* is used by the template, and either it is instantiated from a partial specialization that was not defined when the template was defined or it names an explicit specialization that was not declared when the template was defined.

— *end note*]

[*Note 5*: If a template is instantiated, errors will be diagnosed according to the other rules in this document. Exactly when these errors are diagnosed is a quality of implementation issue. — *end note*]

[*Example 7*:

```
int j;
template<class T> class X {
    void f(T t, int i, char* p) {
        t = i;           // diagnosed if X::f is instantiated, and the assignment to t is an error
        p = i;           // may be diagnosed even if X::f is not instantiated
        p = j;           // may be diagnosed even if X::f is not instantiated
        X<T>::g(t);       // OK
        X<T>::h();        // may be diagnosed even if X::f is not instantiated
    }
    void g(T t) {
        +;               // may be diagnosed even if X::g is not instantiated
    }
};

template<class... T> struct A {
    void operator++(int, T... t);           // error: too many parameters
};
template<class... T> union X : T... { };    // error: union with base class
template<class... T> struct A : T..., T... { }; // error: duplicate base class
```

— *end example*]

## 14 Exception handling

[**except**]

### 14.5 Exception specifications

[**except.spec**]

Modify subclause 14.5[**except.spec**], paragraph 4:

- 4 If a declaration of a function does not have a *noexcept-specifier*, the declaration has a potentially throwing exception specification unless it is a destructor or a deallocation function or is defaulted on its first declaration, in which cases the exception specification is as specified below and no other declaration for that function shall have a *noexcept-specifier*. In an explicit instantiation (13.9.3[temp.explicit]) a *noexcept-specifier* may be specified, but is not required. If a *noexcept-specifier* is specified in an explicit instantiation, the exception specification shall be the same as the exception specification of all other declarations of that function. A diagnostic is required only if the exception specifications are not the same within a single translation unit ([G.10+a.1\[ifndrx:except.spec.mismatched.explicit.instantiation\]](#)).

## F Core undefined behavior

[**ub**]

### F.2 Clause 6[**basic**]: Basics

[**ub.basic**]

### F.2.3 basic.align.object.alignment

[ub:basic.align.object.alignment]

Modify UB description F.2.3[ub:basic.align.object.alignment], paragraph 2:

#### F.2.3

[ub:basic.align.object.alignment]

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

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&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]

### F.2.4 lifetime.outside.pointer.delete

[ub:lifetime.outside.pointer.delete]

Modify UB description F.2.4[ub:lifetime.outside.pointer.delete], paragraphs 1-2:

#### F.2.4

[ub:lifetime.outside.pointer.delete]

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

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

2

[Example 1:

```
struct S {  
    float f = 0;  
    ~S() {}  
};  
  
void f() {  
    S* p = new S;  
    p->~S();  
    delete p; // undefined behavior, operand of delete, lifetime has ended and S  
    // has a non-trivial destructor  
    ←  
}
```

— end example]

### F.2.6 lifetime.outside.pointer.virtual

[ub:lifetime.outside.pointer.virtual]

Modify UB description F.2.6[ub:lifetime.outside.pointer.virtual], paragraphs 1-2:

#### F.2.6

[ub:lifetime.outside.pointer.virtual]

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

For a pointer pointing to an object outside of its lifetime, behavior is undefined if the pointer is ~~implicitly~~ converted (7.3.12[conv.ptr], 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* bb1 = p; undefined behavior, implicit conversion
    B* b2 = static_cast<B*>(p); undefined behavior, explicit conversion
}

```

— end example]

## F.2.15 basic.stc.alloc.dealloc.constraint

[ub:basic.stc.alloc.dealloc.constraint]

Modify UB description F.2.15[ub:basic.stc.alloc.dealloc.constraint], paragraph 2:

### F.2.15

[ub:basic.stc.alloc.dealloc.constraint]

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

2

[Example 1:

```

#include <new>
void* operator new(std::size_t sz) {
    if (sz == 0)
        return nullptr; // undefined behavior, should return non-null pointer

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

```

— end example]

## F.2.16 + a basic.compound.pointer.before.storage.duration

[ub:basic.compound.pointer.before.storage.duration]

Add a new UB description [F.2.16+a](#)[ub:basic.compound.pointer.before.storage.duration] after F.2.16[ub:basic.stc.alloc.zero.dereference]

### F.2.16+a

[ub:basic.compound.pointer.before.storage.duration]

Specified in: 6.9.5[ubx:basic.compound.pointer.before.storage.duration]

1

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 <cstdlib>
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]

```

## F.2.17 basic.compound.invalid.pointer

[ub:basic.compound.invalid.pointer]

Modify UB description F.2.17[ub:basic.compound.invalid.pointer], paragraphs 1-2:

### F.2.17

[ub:basic.compound.invalid.pointer]

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

Indirection ~~or the invocation of a deallocation function~~ 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
    delete x; // undefined behavior
}

```

— end example]

## F.2.19 intro.races.data

[ub:intro.races.data]

Modify UB description F.2.19[ub:intro.races.data], paragraph 2:

### F.2.19

[ub:intro.races.data]

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

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

```

int count = 0;
auto f = [&] { ++count++; };
std::thread t1{f}, t2{f}, t3{f};
// undefined behavior; t1, t2 and t3 have a data race on access of variable count

```

— end example]

## F.2.20 intro.progress.stops

[ub:intro.progress.stops]

Modify UB description F.2.20[ub:intro.progress.stops], paragraph 2:

### F.2.20

[ub:intro.progress.stops]

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

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; thread makes no progress but the loop

```

```

} ————— // is not trivial because stop() is not a constant expression
←
←
} ←

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

— end example]

```

## F.2.21 basic.start.main.exit.during.destruction

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

Modify UB description F.2.21[ub:basic.start.main.exit.during.destruction], paragraph 2:

**F.2.21** [ub:basic.start.main.exit.during.destruction]  
**Specified in:** 6.10.3.1[ubx:basic.start.main.exit.during.destruction]

2 [Example 1:

```

#include <cstdlib>

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

Exiter ex;

int main() {}
// undefined behavior when destructor of static variable ex is called it will call std::exit

```

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

## F.2.22 basic.start.term.use.after.destruction [ub:basic.start.term.use.after.destruction]

Modify UB description F.2.22[ub:basic.start.term.use.after.destruction], paragraph 2:

**F.2.22** [ub:basic.start.term.use.after.destruction]  
**Specified in:** 6.10.3.4[ubx:basic.start.term.use.after.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; static objects are destructed in reverse order; in this case a then b and
// finally c. When the destructor of c is called, it calls f() which passes through the definition of
// previously destroyed block-scope object

```

```
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*

### F.3 Clause 7[expr]: Expressions

[ub:expr]

#### F.3.1 `expr.expr.eval`

[ub:expr.expr.eval]

Add a paragraph to and modify UB description F.3.1[ub:expr.expr.eval], paragraph 2:

##### F.3.1

[ub:expr.expr.eval]

Specified in: 7.1[ubx:expr.expr.eval]

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() {
  // Assuming 32-bit int, the range of values is: -2,147,483,648 to 2,147,483,647.
  int x1 = std::numeric_limits<int>::max() + 1;
  // undefined behavior, 2,147,483,647 + 1 is not representable as an int
  int x2 = std::numeric_limits<int>::min() - 1;
  // undefined behavior, -2,147,483,648 - 1 is not representable as an int
  std::int32_t x1 = std::numeric_limits<std::int32_t>::max() + 1;
  // undefined behavior, 2,147,483,647 + 1 is not representable as a std::int32_t
  std::int32_t x2 = std::numeric_limits<std::int32_t>::min() / -1;
  // undefined behavior, -2,147,483,648 / -1 is not representable as a std::int32_t
  std::int32_t x3 = std::numeric_limits<std::int32_t>::min() * -1;
  // undefined behavior, -2,147,483,648 * -1 is not representable as a std::int32_t
}
```

— *end example*

2+a

[*Example 1+a:* Assuming a `float` type that does not have NaN or infinities, operations that result in either have undefined behavior because their results are not in the range of `float`.

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

— *end example*

#### F.3.5 + a `conv.array.unsimilar.type`

[ub:conv.array.unsimilar.type]

Add a new UB description F.3.5+a[ub:conv.array.unsimilar.type] after F.3.5[ub:conv.lval.valid.representation]

##### F.3.5+a

[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*>(&arr);    // undefined behavior, arr is
                                         // an array of a different size
}
— end example]

```

### F.3.6 conv.double.out.of.range

[ub:conv.double.out.of.range]

Modify UB description F.3.6[ub:conv.double.out.of.range], paragraphs 1-2:

#### F.3.6

[ub:conv.double.out.of.range]

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

Converting a floating-point value to a type that cannot represent ~~the~~ that value or values surrounding that value has undefined behavior.

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() {
    // Assuming 32-bit int, 32-bit float and 64-bit double.
    double d2 = std::numeric_limits<double>::max();
    float f = d2; // undefined behavior on systems where the range of representable values
                // of float is [-max,+max]; on systems where the range of representable
                // values is [-inf,+inf] this would not be undefined behavior
    int_ifloat f = d2; // undefined behavior, the max value of doubled2 is not representable as int greater than
                    // the largest representable float
}
— end example]

```

### F.3.7 conv.fpint.float.not.represented

[ub:conv.fpint.float.not.represented]

Modify UB description F.3.7[ub:conv.fpint.float.not.represented], paragraphs 1-2:

#### F.3.7

[ub:conv.fpint.float.not.represented]

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

When converting a floating-point value to an integer type, if the truncated value is not representable in the destination type, the behavior is undefined.

2

[Example 1: In the following example, assume that std::float64\_t, std::int32\_t, and std::uint32\_t are supported.

```

#include <cstdint>
#include <stdfloat>
#include <limits>

int main() {
    // Assuming 32-bit int, the range of values is: -2,147,483,648 to
    // 2,147,483,647. Assuming 32-bit float and 64-bit double.
    double d = (double)std::numeric_limits<int>::max() + 1;
    int x1 = d; // undefined behavior, 2,147,483,647 + 1 is not representable as int
    std::float64_t d = static_cast<std::float64_t>(std::numeric_limits<std::int32_t>::max()) + 1;
    std::int32_t x1 = d; // undefined behavior, 2,147,483,647 + 1 is not representable as std::int32_t
    std::int32_t x2 = std::numeric_limits<std::float64_t>::infinity();
    // undefined behavior, infinity is not representable as std::int32_t
    std::int32_t x3 = -std::numeric_limits<std::float64_t>::infinity();
    // undefined behavior, negative infinity is not representable as std::int32_t
    std::int32_t x4 = std::numeric_limits<std::float64_t>::quiet_NaN();
    // undefined behavior, NaN is not representable as std::int32_t
}

```

```

    std::uint32_t x5 = -0.5f; // OK, truncated value is 0
}
— end example]

```

### F.3.9 conv.ptr.virtual.base

[ub:conv.ptr.virtual.base]

Modify UB description F.3.9[ub:conv.ptr.virtual.base], paragraph 2:

#### F.3.9

[ub:conv.ptr.virtual.base]

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

2

[Example 1:

```

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

```

— end example]

### F.3.13 expr.dynamic.cast.pointer.lifetime

[ub:expr.dynamic.cast.pointer.lifetime]

Modify UB description F.3.13[ub:expr.dynamic.cast.pointer.lifetime], paragraph 2:

#### F.3.13

[ub:expr.dynamic.cast.pointer.lifetime]

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

2

[Example 1:

```

struct B { virtual ~B(); };
struct D : B {}; ←
void f() {
    B* bsf1bp = new B;
    bp->~B();
    D* dp = dynamic_cast<B*>(bsf1D*)(bp); uuu // undefined behavior
}

```

— end example]

### F.3.14 expr.dynamic.cast.glvalue.lifetime

[ub:expr.dynamic.cast.glvalue.lifetime]

Modify UB description F.3.14[ub:expr.dynamic.cast.glvalue.lifetime], paragraph 2:

#### F.3.14

[ub:expr.dynamic.cast.glvalue.lifetime]

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

2

[Example 1:

```

struct B { virtual ~B(); };
struct D : B {}; ←
void f() {
    B* bsf1bp = new B;
    B& br = *bp;
    bp->~B();
    D& dr = dynamic_cast<B&>(bsf1D&)(br); uuu // undefined behavior
}

```

— end example]

**F.3.19 + a** **expr.reinterpret.cast.invalid.pointer.value**

[ub:expr.reinterpret.cast.invalid.pointer.value]

Add a new UB description **F.3.19+a**[ub:expr.reinterpret.cast.invalid.pointer.value] after F.3.19[ub:expr.static.cast.does.not.contain.original.member]

**F.3.19+a** [ub:expr.reinterpret.cast.invalid.pointer.value]  
**Specified in:** 7.6.1.10[ubx:expr.reinterpret.cast.invalid.pointer.value]

- 1 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 <cstdlib>
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]

**F.3.22** **expr.delete.mismatch**

[ub:expr.delete.mismatch]

Modify UB description F.3.22[ub:expr.delete.mismatch], paragraphs 1-2:

**F.3.22** [ub:expr.delete.mismatch]  
**Specified in:** 7.6.2.9[ubx:expr.delete.mismatch]  
Using ~~array~~single object delete on the result of ~~a single object~~an array new expression has undefined behavior.

- 2 [Example 1:

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

— end example]

**F.3.23** **expr.delete.array.mismatch**

[ub:expr.delete.array.mismatch]

Modify UB description F.3.23[ub:expr.delete.array.mismatch], paragraphs 1-2:

**F.3.23** [ub:expr.delete.array.mismatch]  
**Specified in:** 7.6.2.9[ubx:expr.delete.array.mismatch]  
Using ~~single object~~array delete on the result of ~~an array~~a single object new expression

has undefined behavior.

2

[Example 1:

```
int* x = new int[40];
delete[] x; // undefined behavior, allocated using array single object new expression
— end example]
```

### F.3.26 `expr.mptr.oper.not.contain.member` [ub:expr.mptr.oper.not.contain.member]

Modify UB description F.3.26[ub:expr.mptr.oper.not.contain.member], paragraphs 1-2:

**F.3.26** [ub:expr.mptr.oper.not.contain.member]

**Specified in:** 7.6.4[ubx:expr.mptr.oper.not.contain.member]

~~Abbreviating~~ In a pm-expression, \*cast-expression as of the form E1.\*E2, if E1 is called  
~~therefers to an object expression. If whose type is not similar to the dynamic 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() {
— B *b = new B;
— D *d = static_cast<D*>(b);
— int D::*p = &D::x;
( *d ).*p = 1; // undefined behavior, dynamic type B does not contain
int B::*pb = static_cast<int B::*>(&D::x);
B b;
b.*pb = 1; // undefined behavior, B does not contain x
}
— end example]
```

### F.3.28 `expr.mul.div.by.zero`

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

Modify UB description F.3.28[ub:expr.mul.div.by.zero], paragraph 2:

**F.3.28** [ub:expr.mul.div.by.zero]

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

2

[Example 1:

```
int main() {
    int x = 1 / 0; // undefined behavior, division by zero
    double d = 1.0 / 0.0; // undefined behavior on systems where the range of  

representable values of double is [-max, max], on systems where  

representable values is [-inf, inf] this would not be undefined behavior
}
— end example]
```

### F.3.29 `expr.mul.representable.type.result`

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

Modify UB description F.3.29[ub:expr.mul.representable.type.result], paragraph 2:

**F.3.29****[ub:expr.mul.representable.type.result]****Specified in:** 7.6.5[ubx:expr.mul.representable.type.result]

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() {
    intstd::int32_t x = std::numeric_limits<intstd::int32_t>::min() / -1; ↔
    // Assuming LP64 -2,147,483,648 which when divided by -1 undefined behavior
    std::int32_t y = std::numeric_limits<std::int32_t>::min() % -1; // gives us 2,147,483,648 which is not
    representable by int undefined behavior
}
```

— end example]

**F.3.30 expr.add.out.of.bounds****[ub:expr.add.out.of.bounds]**

Modify UB description F.3.30[ub:expr.add.out.of.bounds], paragraph 2:

**F.3.30****[ub:expr.add.out.of.bounds]****Specified in:** 7.6.6[ubx:expr.add.out.of.bounds]

2

[Example 1:

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

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

— end example]

[Example 2:

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

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

— end example]

**F.3.30 + a expr.sub.pointers.rep****[ub:expr.sub.pointers.rep]**Add a new UB description **F.3.30+a**[ub:expr.sub.pointers.rep] after F.3.30[ub:expr.add.out.of.bounds]**F.3.30+a****[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.

2

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

```

#include <cstddef>
#include <cstdint>
char big[static_cast<std::size_t>(PTRDIFF_MAX) + 1];
std::ptrdiff_t d = big + sizeof(big) - big;    // undefined behavior

— end example]

```

### F.3.33 `expr.shift.neg.and.width`

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

Modify UB description F.3.33[ub:expr.shift.neg.and.width], paragraph 2:

#### F.3.33

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

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

2

[Example 1:

```

#include <limits>
unsigned int yy1 = 1 << -1;          // undefined behavior, shift is negative
←
static_assert(sizeof(unsigned int) == 4 && CHAR_BIT == 8, "y2 = 1 << std::numeric_limits<unsigned> 8); ←
int::digits; ←
y1 = 1 << 32;                       // undefined behavior, shift is equal to the bit width of int
unsigned int y2y3 = 1 >> 32std::numeric_limits<unsigned int>::digits; ←
// undefined behavior, shift is equal to the bit width of int

```

— end example]

## F.4 Clause 8[stmt]: Statements

[ub.stmt]

### F.4.2 `stmt.return.coroutine.flow.off`

[ub:stmt.return.coroutine.flow.off]

Modify UB description F.4.2[ub:stmt.return.coroutine.flow.off], paragraphs 1-2:

#### F.4.2

[ub:stmt.return.coroutine.flow.off]

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

~~Flowing off the end of~~In a coroutine ~~function body~~whose promise object is p that does not return voidand 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(coroutine_handle handle) : handle_(handle) { assert(handle); }
    resumable(resumable&) = delete;
    resumable(resumable&&) = delete;
    bool resume() {
        if (not handle_.done())
            handle_.resume();
        return not handle_.done();
    }
    ~resumable() { handle_.destroy(); }
    const char* return_val();

private:
    coro_handle handle_;

```

```

};

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

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

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

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

— end example]

```

## F.5 Clause 9[dcl]: Declarations

[ub:dcl]

### F.5.5 dcl.ref.uninitialized.reference

[ub:dcl.ref.uninitialized.reference]

Modify UB description F.5.5[ub:dcl.ref.uninitialized.reference], paragraph 2:

**F.5.5** [ub:dcl.ref.uninitialized.reference]  
**Specified in:** 9.3.4.3[ubx:dcl.ref.uninitialized.reference]

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 = i3dynamicValue(); // not constant-initialized

```

— end example]

### F.5.6 dcl.fct.def.coroutine.resume.not.suspended

[ub:dcl.fct.def.coroutine.resume.not.suspended]

Modify UB description F.5.6[ub:dcl.fct.def.coroutine.resume.not.suspended], paragraph 2:

**F.5.6** [ub:dcl.fct.def.coroutine.resume.not.suspended]  
**Specified in:** 9.6.4[ubx:dcl.fct.def.coroutine.resume.not.suspended]

2 [Example 1:

```

#include <coroutine>

struct minigtask {

```

```

struct promise_type {
    int val;
    minig_get_return_object() { return { *this }; }
    constexpr suspend_always initial_suspend() noexcept { return {}; }
    constexpr suspend_always final_suspend() noexcept { return {}; }
    constexpr void return_void() noexcept {}
    [[noreturn]] void unhandled_exception() noexcept { throw; }
    suspend_always yield_value(int v) noexcept {
        val = v;
        return {};
    }
    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 {}
};

using HDL = coroutine_handle<promise_type>;
HDL coro;
minig(promise_type& p) : coro(HDL::from_promise(p)) {}
minig() { coro.destroy(); }
bool move_next() {
    coro.resume();
    return !coro.done();
}
int current_value() { return coro.promise().val; }

static minig f(int n) {
    for (int i = 0; i < n; ++i)
        co_yield i;
}

int main() {
    auto g = f(10);
    int sum = 0;
    while (g.move_next())
        sum += g.current_value();

    g.move_next(); // undefined behavior, will call coro.resume() but final_suspend
                  // has already returned, even though it returned suspend_always
    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; }
    };

    return sum;
}

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

— end example]

```

### F.5.7 dcl.fct.def.coroutine.destroy.not.suspended

[ub:dcl.fct.def.coroutine.destroy.not.suspended]

Modify UB description F.5.7[ub:dcl.fct.def.coroutine.destroy.not.suspended], paragraph 2:

#### F.5.7

[ub:dcl.fct.def.coroutine.destroy.not.suspended]

Specified in: 9.6.4[ubx:dcl.fct.def.coroutine.destroy.not.suspended]

2

[Example 1:

```
#include <coroutine>
```

```

struct minigtask {
    struct promise_type {
        int val;
        minig_get_return_object() { return { *this }; }
        constexpr suspend_always initial_suspend() noexcept { return {}; }
        constexpr suspend_always final_suspend() noexcept { return {}; }
        constexpr void return_void() noexcept {}
        [[noreturn]] void unhandled_exception() { throw; }
        suspend_always yield_value(int v) noexcept {
            val = v;
            return {};
        }
        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 {}
    };
    using HDL = coroutine_handle<promise_type>;
    HDL coro;
    minig(promise_type& p) : coro(HDL::from_promise(p)) {}
    ~minig() { coro.destroy(); }
    bool move_next() {
        coro.resume();
        return !coro.done();
    }
    int current_value() {
        // this coroutine is not suspended therefore a call to destroy has undefined behavior
        coro.destroy();
        return coro.promise().val;
    }
};

static minig f(int n) {
    for (int i = 0; i < n; ++i)
        co_yield i;
}

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; }
};

int main() {
    auto g = f(10);
    int sum = 0;
    while (g.move_next())
        sum += g.current_value();

    return sum;

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

```

— end example]

## F.6 Clause 11[class]: Classes

[ub.class]

### F.6.3 class.base.init.mem.fun

[ub:class.base.init.mem.fun]

Modify UB description F.6.3[ub:class.base.init.mem.fun], paragraphs 1-2:

#### 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()
        pre(this->f()) // undefined behavior, calls member function in constructor precondition
        : A(f()),      // undefined behavior, calls member function but base A not yet initialized
          j(f()) {}     // defined, bases are all initialized
    ~B()
        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]

## F.6.6 class.ctor.convert.pointer

[ub:class.ctor.convert.pointer]

Modify UB description F.6.6[ub:class.ctor.convert.pointer], paragraph 2:

### F.6.6

[ub:class.ctor.convert.pointer]

Specified in: 11.9.5[ubx:class.ctor.convert.pointer]

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, 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
```

```

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

G Ill-formed, no diagnostic required

[ifndr]

G.3 Clause 6[**basic**]: Basics

[ifndr.basic]

G.3.4 **basic.def.odr.maximum.one.def**

[ifndr:basic.def.odr.maximum.one.def]

Modify IFNDR description G.3.4[ifndr:basic.def.odr.maximum.one.def], paragraph 1:

**G.3.4** [ifndr:basic.def.odr.maximum.one.def]

**Specified in:** 6.3[ifndrx:basic.def.odr.maximum.one.def]

~~If there are definitions in different translation units~~ For a definable item D ~~of that~~

- (1.a) — is a non-inline non-templated function or variable, ~~that are~~
- (1.b) — is not defined by an injected declaration (7.7.6[expr.const.reflect]),
- (1.c) — has definitions in different translation units, and
- (1.d) — is not attached to a named module or has definitions that are not reachable from one another,

the program is ill-formed, no diagnostic required.

G.3.5 **basic.def.odr.definition.matches**

[ifndr:basic.def.odr.definition.matches]

Modify IFNDR description G.3.5[ifndr:basic.def.odr.definition.matches], paragraphs 1-2:

**G.3.5** [ifndr:basic.def.odr.definition.matches]

**Specified in:** 6.3[ifndrx:basic.def.odr.definition.matches]

~~If there are definitions in different translation units of~~ For a definable item D ~~where that~~

- (1.1) — ~~D is not defined by an injected declaration~~ (7.7.6[expr.const.reflect]) a non-inline non-templated function or variable,
- (1.2) — ~~D is not an inline or templated function or variable~~ defined by an injected declaration (7.7.6[expr.const.reflect]), ~~and~~
- (1.2+a) — has definitions in different translation units that do not satisfy the matching rules described in 6.3[**basic.def.odr**], and
- (1.3) — ~~D is not attached to a named module or the declarations are not~~ has no definitions reachable from ~~one~~ another,

~~that do not satisfy the matching rules described in 6.3[**basic.def.odr**], the~~ program is ill-formed, no diagnostic required.

2

[Example 1:

Translation unit #1:

```

inline void f() {}           // #1
inline void g() {}           // #2
inline void h(bool cond, void (*p)() f= [](){};) // #3{
    if (cond) h(false);
} // #3←
namespace { int i = 0; }
inline void j() {++i;}       // #4

```

Translation unit #2:

```

inline void f() {}           // OK, same as #1
inline void g() {}           // IFNDR, different sequence of tokens than #2
inline void h(bool cond, void (*p)() f= [](){};) // IFNDR, closure has different type than #3{
    if (cond) h(false);
} // IFNDR, closure in default argument has different type than #3←
namespace { int i = 0; }
inline void j() {++i;}       // IFNDR, i refers to different entity than in #4

```

— end example]

## G.4 Clause 7[expr]: Expressions

[ifndr.expr]

G.4.1 `expr.prim.req.always.sub.fail`

[ifndr:expr.prim.req.always.sub.fail]

Modify IFNDR description G.4.1[ifndr:expr.prim.req.always.sub.fail], paragraph 2:

## G.4.1

[ifndr:expr.prim.req.always.sub.fail]

Specified in: 7.5.8.1[ifndrx:expr.prim.req.always.sub.fail]

2

[Example 1:

```

template <typename T> concept C = requires {
    new int[decltype(int)sizeof(void)T{}];;;; // IFNDR, the sizeinvalid use of the allocation is required to be
greater
----- // than zero but can never bevoid in all instantiations
};

```

— end example]

G.4.1 + a `expr.const.const.immediate.var.def` [ifndr:expr.const.const.immediate.var.def]Add a new IFNDR description G.4.1+a[ifndr:expr.const.const.immediate.var.def] after G.4.1[ifndr:expr.prim.req.always.sub.fail]

## G.4.1+a

[ifndr:expr.const.const.immediate.var.def]

Specified in: 7.7.3[ifndrx:expr.const.const.immediate.var.def]

1

A variable that declares or refers to an immediate object must be odr-used only from immediate function contexts. No diagnostic is required if the odr-use cannot reach the defining declaration of the variable and the defining declaration cannot reach the odr-use.

2

[Example 1:

Translation unit #1:

```

struct S { decltype(??::) d = ??::; };
extern constexpr S s{};           // #1

```

Translation unit #2:

```
struct S;  
extern const S s;  
const void* p = &s;           // IFNDR, cannot reach #1 that is an immediate object
```

— end example]

## G.6 Clause 9[dcl]: Declarations

[ifndr.dcl]

### G.6.3 + a dcl.contract.func.mismatched.redeclaration

[ifndr:dcl.contract.func.mismatched.redeclaration]

Add a new IFNDR description [G.6.3+a](#)[ifndr:dcl.contract.func.mismatched.redeclaration] after G.6.3[ifndr:dcl.fct.default.inline.same.defaults]

**G.6.3+a** [ifndr:dcl.contract.func.mismatched.redeclaration]  
**Specified in:** 9.4.1[ifndrx:dcl.contract.func.mismatched.redeclaration]

- 1 If a function that is not attached to a named module has a redeclaration in a different translation unit and that redeclaration declares mismatched function-contract specifiers, the program is ill-formed, no diagnostic required.

- 2 [Example 1:

Translation unit #1:

```
export module M;  
export extern "C++" int f(int x) pre(x >= 0); // #1, attached to the global module
```

Translation unit #2:

```
import M;  
int f(int x) pre(x != 0); // IFNDR, pre differs from #1, which is reachable
```

— end example]

### G.6.4 + a dcl.contract.func.mismatched.lambda

[ifndr:dcl.contract.func.mismatched.lambda]

Add a new IFNDR description [G.6.4+a](#)[ifndr:dcl.contract.func.mismatched.lambda] after G.6.4[ifndr:dcl.contract.func.mismatched.contract.specifiers]

**G.6.4+a** [ifndr:dcl.contract.func.mismatched.lambda]  
**Specified in:** 9.4.1[ifndrx:dcl.contract.func.mismatched.lambda]

- 1 If the function-contract specifiers must compare equal and they do not match solely due to the comparison of *lambda-expressions* nested within them then no diagnostic is required.

- 2 [Example 1:

```
void f() pre([]{ return true; }());  
void f() pre([]{ return true; }()); // IFNDR, closures have different types
```

— end example]

## G.7 Clause 10[module]: Modules

[ifndr.module]

### G.7.2 + a module.unit.conflicting.module.partitions

[ifndr:module.unit.conflicting.module.partitions]

Add a new IFNDR description [G.7.2+a](#)[ifndr:module.unit.conflicting.module.partitions] after G.7.2[ifndr:module.unit.named.module.no.partition]

**G.7.2+a** [ifndr:module.unit.conflicting.module.partitions]

**Specified in:** 10.1[ifndrx:module.unit.conflicting.module.partitions]

1 If more than one module partition has the same *module-partition*, the program is ill-formed, no diagnostic required.

2 [Example 1:

Translation unit #1:

```
module M:oops; // IFNDR, multiple module partitions named :oops
```

Translation unit #2:

```
module M:oops; // IFNDR, multiple module partitions named :oops
```

— end example]

## G.10 Clause 13[temp]: Templates

[ifndr.temp]

### G.10.8 + a temp.res.general.no.valid.specialization

[ifndr:temp.res.general.no.valid.specialization]

Add a new IFNDR description [G.10.8+a](#)[ifndr:temp.res.general.no.valid.specialization] after G.10.8[ifndr:temp.res.general.default.but.not.found]

**G.10.8+a** [ifndr:temp.res.general.no.valid.specialization]

**Specified in:** 13.8.1[ifndrx:temp.res.general.no.valid.specialization]

1 If no valid specialization, ignoring failing *static\_assert-declarations*, can be generated for a templated entity, or for a substatement of a constexpr if statement within one, and the innermost enclosing template is not instantiated, the program is ill-formed, no diagnostic required.

2 [Example 1:

```
template<class T> void f() {  
    T& r[1]; // IFNDR, an array of references is never valid  
}
```

— end example]

### G.10.8 + b temp.res.general.no.valid.expansion

[ifndr:temp.res.general.no.valid.expansion]

Add a new IFNDR description [G.10.8+b](#)[ifndr:temp.res.general.no.valid.expansion] after [G.10.8+a](#)[ifndr:temp.res.general.no.valid.specialization]

**G.10.8+b** [ifndr:temp.res.general.no.valid.expansion]  
**Specified in:** 13.8.1[ifndrx:temp.res.general.no.valid.expansion]

1 If no valid specialization, ignoring failing *static\_assert-declarations*, can be generated for the body of an *expansion-statement* that is never instantiated, the program is ill-formed, no diagnostic required.

2 [Example 1:

```
void f() {  
    template for (auto x : {}) {  
        decltype(x)& r[1]; // IFNDR, never valid and never instantiated  
    }  
}
```

— end example]

**G.10.8 + c temp.res.general.no.valid.default.arg** [ifndr:temp.res.general.no.valid.default.arg]

Add a new IFNDR description **G.10.8+c**[ifndr:temp.res.general.no.valid.default.arg] after **G.10.8+b**[ifndr:temp.res.general.no.valid.expansion]

**G.10.8+c** [ifndr:temp.res.general.no.valid.default.arg]  
**Specified in:** 13.8.1[ifndrx:temp.res.general.no.valid.default.arg]

1 If no valid specialization, ignoring failing *static\_assert-declarations*, can be generated for a default *template-argument* that is never used, the program is ill-formed, no diagnostic required.

2 [Example 1:

```
template<class T, class U = T&[1]> struct S {}; // IFNDR, T&[1] is never valid  
S<int, int> s;
```

— end example]

**G.10.8 + d temp.res.general.no.valid.alias** [ifndr:temp.res.general.no.valid.alias]

Add a new IFNDR description **G.10.8+d**[ifndr:temp.res.general.no.valid.alias] after **G.10.8+c**[ifndr:temp.res.general.no.valid.default.arg]

**G.10.8+d** [ifndr:temp.res.general.no.valid.alias]  
**Specified in:** 13.8.1[ifndrx:temp.res.general.no.valid.alias]

1 If no specialization of an alias template is valid and none is named in the program, the program is ill-formed, no diagnostic required.

2 [Example 1:

```
template<class T> using A = T&[1]; // IFNDR, no specialization of A is valid
```

— end example]

**G.10.8 + e temp.res.general.no.valid.constraint.check**  
**[ifndr:temp.res.general.no.valid.constraint.check]**

Add a new IFNDR description **G.10.8+e**[ifndr:temp.res.general.no.valid.constraint.check] after **G.10.8+d**[ifndr:temp.res.general.no.valid.alias]

**G.10.8+e** **[ifndr:temp.res.general.no.valid.constraint.check]**  
**Specified in:** 13.8.1[ifndrx:temp.res.general.no.valid.constraint.check]

1 If an atomic constraint can never be checked without being ill-formed and is never checked, the program is ill-formed, no diagnostic required.

2 [Example 1:

```
template<class T>
    requires (sizeof(T) > 1) && 1    // IFNDR, 1 is not of type bool
void f(T);
```

— end example]

**G.10.8 + f temp.res.general.variadic.empty.pack**  
**[ifndr:temp.res.general.variadic.empty.pack]**

Add a new IFNDR description **G.10.8+f**[ifndr:temp.res.general.variadic.empty.pack] after **G.10.8+e**[ifndr:temp.res.general.no.valid.constraint.check]

**G.10.8+f** **[ifndr:temp.res.general.variadic.empty.pack]**  
**Specified in:** 13.8.1[ifndrx:temp.res.general.variadic.empty.pack]

1 If every valid specialization of a variadic template requires an empty template parameter pack, the program is ill-formed, no diagnostic required.

2 [Example 1:

```
template<class... T> struct S {
    void operator++(int, T...);    // IFNDR, valid only if T is empty
};
```

— end example]

**G.10.8 + g temp.res.general.hypothetical.ill.formed**  
**[ifndr:temp.res.general.hypothetical.ill.formed]**

Add a new IFNDR description **G.10.8+g**[ifndr:temp.res.general.hypothetical.ill.formed] after **G.10.8+f**[ifndr:temp.res.general.variadic.empty.pack]

**G.10.8+g** **[ifndr:temp.res.general.hypothetical.ill.formed]**  
**Specified in:** 13.8.1[ifndrx:temp.res.general.hypothetical.ill.formed]

1 If a templated entity would be ill-formed because of a construct that does not depend on a template parameter (other than a failing *static\_assert-declaration*), the program is ill-formed, no diagnostic required.

2 [Example 1:

```
template<class T> void f() {
    int* p = 1;    // IFNDR, ill-formed for every T
}
```

— end example]

**G.10.8 + h temp.res.general.hypothetical.different.meaning**

[ifndr:temp.res.general.hypothetical.different.meaning]

Add a new IFNDR description [G.10.8+h](#)[ifndr:temp.res.general.hypothetical.different.meaning] after [G.10.8+g](#)[ifndr:temp.res.general.hypothetical.ill.formed]

**G.10.8+h** [ifndr:temp.res.general.hypothetical.different.meaning]

**Specified in:** 13.8.1[ifndrx:temp.res.general.hypothetical.different.meaning]

- 1 If a construct that does not depend on a template parameter would be interpreted differently immediately following the definition of a templated entity than in an instantiation of that entity, the program is ill-formed, no diagnostic required.

- 2 [Example 1:

```
struct B {};
struct D;
void h(void*);
void h(B*);
template<class T> void f(D* p) {
    h(p);    // IFNDR, h(void*) here, h(B*) in f<int>
}
struct D : B {};
template void f<int>(D*);
```

— end example]

**G.10 + a Clause 14[except]: Exception handling**

[ifndr.except]

Add a new subclause [G.10+a](#)[ifndr.except] after G.10.14[ifndr:temp.deduct.general.diff.order]

**G.10+a Clause 14[except]: Exception handling**

[ifndr.except]

**G.10 + a.1 except.spec.mismatched.explicit.instantiation**

[ifndr:except.spec.mismatched.explicit.instantiation]

Add a new IFNDR description [G.10+a.1](#)[ifndr:except.spec.mismatched.explicit.instantiation] under [G.10+a](#)[ifndr.except]

**G.10+a.1** [ifndr:except.spec.mismatched.explicit.instantiation]

**Specified in:** 14.5[ifndrx:except.spec.mismatched.explicit.instantiation]

- 1 If an explicit instantiation in one translation unit contains a *noexcept-specifier* that is not the same as that of a declaration of the same function in another translation unit, the program is ill-formed, no diagnostic required.

- 2 [Example 1:

Translation unit #1:

```
void g(int) noexcept;
template<class T> void f(T t) noexcept(noexcept(g(t))) {}
template void f(int) noexcept;           // #1, g(int) is noexcept
```

Translation unit #2:

```
void g(long);
template<class T> void f(T t) noexcept(noexcept(g(t)));
extern template void f(int) noexcept(false); // IFNDR, g(long) is not noexcept,
                                              // differs from #1
```

— end example]

### 3 Conclusion

These changes are being published for the record, and, given the editorial nature of the annex, may be simply merged as editorial issues. Our intent with this paper is to be a companion piece for [P3596R3] to reference the initial contents of the new annexes.

### Acknowledgments

Thank you to Jens Maurer, Thomas Köppe, Shafik Yaghmour, A. Jiang, Dan Katz, Davis Herring, Timur Doumler, Jan Schultke, Lénárd Szolnoki, and Yin Xinyu for contributing to, reviewing, authoring, or co-authoring these updates.

Claude (Anthropic) was used for editorial assistance during the preparation of this paper.

### Bibliography

- [CWG2300] Robert Haberlach, “Lambdas in multiple definitions”  
<https://wg21.link/cwg2300>
- [CWG2392] Tam S. B, “new-expression size check and constant evaluation”  
<https://wg21.link/cwg2392>
- [CWG3232] Jan Schultke, “Accidental removal of undefined behavior in pointer subtraction”  
<https://wg21.link/cwg3232>
- [P2414R12] Paul E. McKenney, Maged Michael, Jens Maurer, Peter Sewell, Martin Uecker, Hans Boehm, Hubert Tong, Niall Douglas, Thomas Rodgers, Will Deacon, Michael Wong, David Goldblatt, Kostya Serebryany, Anthony Williams, Tom Scogland, JF Bastien, Daniel Krügler, and David Tenty, “Pointer lifetime-end zap proposed solutions”, 2026  
<http://wg21.link/P2414R12>
- [P2434R5] S. Davis Herring, “Nondeterministic pointer provenance”, 2026  
<http://wg21.link/P2434R5>
- [P3347R6] Paul E. McKenney, Maged Michael, Jens Maurer, Peter Sewell, Martin Uecker, Hans Boehm, Hubert Tong, Niall Douglas, Thomas Rodgers, Will Deacon, Michael Wong,

- David Goldblatt, Kostya Serebryany, Anthony Williams, Tom Scogland, JF Bastien, Jason McGuinness, and David Tenty, “Invalid/Prospective Pointer Operations”, 2026  
<http://wg21.link/P3347R6>
- [P3424R2] Alisdair Meredith, “Define Delete With Throwing Exception Specification”, 2026  
<http://wg21.link/P3424R2>
- [P3596R3] Joshua Berne, Timur Doumler, Jens Maurer, and Shafik Yaghmour, “Undefined Behavior and IFNDR Annexes”, 2026  
<http://wg21.link/P3596R3>
- [P3899R3] Jan Schultke and Matthias Kretz, “Clarify the behavior of floating-point overflow”, 2026  
<http://wg21.link/P3899R3>
- [P3950R1] Robert Leahy, “`return_value` & `return_void` Are Not Mutually Exclusive”, 2026  
<http://wg21.link/P3950R1>
- [P4101R1] Barry Revzin, Peter Dimov, Daveed Vandevoorde, and Dan Katz, “Consteval-only Values for C++26”, 2026  
<http://wg21.link/P4101R1>