# Addenda to the Undefined Behavior and IFNDR Annexes


| | |
|---|---|
| Document #: | P4284R0 |
| Date: | 2026-08-13 |
| Project: | Programming Language C++ |
| Audience: | CWG |


Reply-to: Joshua Berne <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was current and complete when it was approved to be merged into the draft Standard in Brno, but it was also instantly out of date due to other papers that landed concurrently at the same meeting.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

## Revision History

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 [ifndr: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 [ifndr:basic.def.odr.-definition.matches] is now well-formed thanks to [CWG2300].

  To address this, we have changed the example to use a local function declaration that has a lambda for a default argument, which we then invoke.

- 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 [expr.sub.pointers.representable].

- 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-sized types from `<cstdint>` and `<cstdfloat>`, such as `std::int32_t`, `std::float32_t`, and `std::float64_t`.

- A bug in [basic.def.odr.definition.matches] was noted by Lénárd Szolnoki (see Editorial Issue 9238), where the IFNDR case was being qualified to a definiable item that is *not* "an inline or

templated function". The correct qualification is the opposite — the definitable item must be inline or templated.

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. This does not seem to add to understanding the undefined behavior itself, which is more about general control flow in coroutines, and so we have not updated the wording for the annex entry.

- 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.

  [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 entries:

  – [ifndr: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.

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 ec1038c8, last modified on Sat, 1 Aug 2026 19:40:39 +0200.

**Modified Section Contents**

## 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:

5 If an evaluation produces or causes an object to have (6.9.2[basic.types.trivial]) a pointer value to or past the end of an object $O$ and happens before the beginning of the duration of the region of storage for $O$, the behavior is undefined (F.2.16+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]

### 7.6 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 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.representable]). — *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 subject thereof, or

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

[*Example 1*:

```
consteval 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]).

## *F* Core undefined behavior [ub]

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

### *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 storage duration of an object cannot be a pointer to that object.

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

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

— *end example*]

### *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]

~~[*Example 1*:~~

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

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

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

— *end example*]

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
// undefined behavior
```

— *end example*]

Objects with static storage duration are destroyed in reverse order of the completion of their construction. In the above example, first `c` then `a` and finally `b` completes construction. When

the destructor of `c` is called after first destroying `b` and then `a`, it calls `f()` which passes through the definition of the previously destroyed block-scope object `a`. — *end example*]

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

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

Modify UB description F.3.1[ub:expr.expr.eval], paragraph 2:

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

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

2 [*Example 1*:

```
#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
}
```

— *end example*]

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

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

— *end example*]

### *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], paragraph 2:

**F.3.6** **[ub:conv.double.out.of.range]**

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

~~[*Example 1*:~~

2 [*Example a*: Assume a 32-bit `int`, 32-bit `float`, and a 64-bit `double` 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 i = d2;   // undefined behavior, the max value of double is not representable as int
}
```

— *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], paragraph 2:

**F.3.7** **[ub:conv.fpint.float.not.represented]**
**Specified in:** 7.3.11[ubx:conv.fpint.float.not.represented]

2 [*Example 1*:

```
#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 std::float64_t d␣ = static_cast<std::float64_t>(double)std::numeric_limits<intstd::int32_t>::max()) + 1;
  int std::int32_t  x1 = d;␣   // undefined behavior, 2,147,483,647 + 1 is not representable as intstd::int32_t
}
```

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

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

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

— *end example*]

*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*: Dividing the minimum `std::int32_t` value of $-2,147,483,648$ by $-1$ yields $2,147,483,648$ which is not representable by `std::int32_t` on such platforms.

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

int main() {
  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 (F.3.1[ub:expr.expr.eval])
  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;␣␣        // 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.representable** [ub:expr.sub.pointers.representable]

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

**F.3.30+a** [ub:expr.sub.pointers.representable]
**Specified in:** 7.6.6[ubx:expr.sub.pointers.representable]

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

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

```
#include <limits>
#include <iterator>

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

— *end example*]

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

```
~~int~~#include <cstdint>
std::uint32_t y = std::uint32_t(1) << -1;□□□□□   // undefined behavior, shift is negative
~~static_assert(sizeof(int) == 4 && CHAR_BIT == 8);~~
~~int~~std::uint32_t y1 = std::uint32_t(1) << 32;□□□□□  // undefined behavior, shift is equal to ~~the~~ bit width ~~of int~~
~~int~~std::uint32_t y2 = std::uint32_t(1) >> 32;□□□□□  // undefined behavior, shift is equal to ~~the~~ bit width ~~of int~~
```

— *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-initializer*s for base classes have completed, or after base classes have been destroyed, has undefined behavior. This includes precondition assertions of the constructor and postcondition assertions of the destructor.

2 [*Example 1*:

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

class B : public A {
  int j;

public:
  int f();
  B()
      pre( this->f() )   // undefined behavior, calls member function in constructor precondition↩
      : A(f()),          // undefined behavior, calls member function but base A not yet initialized
        j(f()) {}        // defined, bases are all initialized
↩
  B(int x)
      pre( this->f() )   // undefined behavior, calls member function in delegating constructor precondition
      : B() {}

  ~B()
      post( this->f() ) // undefined behavior, calls member function in destructor postcondition
      {}↩
};

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

class D : public B, C {
  int i;

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

— *end example*]

### *F*.6.6 **class.cdtor.convert.pointer** **[ub:class.cdtor.convert.pointer]**

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

**F.6.6** **[ub:class.cdtor.convert.pointer]**

**Specified in:** 11.9.5[ubx:class.cdtor.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↩~~
~~↩~~
↩
  X(this) {}        // defined, upon construction of X, C/B/D/A sublattice is fully constructed
};
```

~~— *end example*~~]

In the above constructor for `E`, upcast from `E*` to `A*` might use the path `E*` → `D*` → `A*`, but the `D` subobject has not yet been constructed. If the member initializer was instead "`D((C*)this)`" then that would be defined, because `E*` → `C*` is defined (`E()` has started) and `C*` → `A*` is defined (the `C` subobject is fully constructed). — *end example*]

## *G* Ill-formed, no diagnostic required [ifndr]

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

#### *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 a definable item `D` where

(1.1) — `D` is not defined by an injected declaration (7.7.6[expr.const.reflect]),

(1.2) — `D` is ~~not~~ an inline or templated function or variable, and

(1.3) — `D` is not attached to a named module or the declarations are not 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~~f() {~~[]{}();~}                                    // #~~3~~1
inline void g() {}                                                  // #2
inline void h(bool cond, void (*p)() = []{}) {
  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)() ~~{~~= []{}()~~;}~~  // ~~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 or the defining declaration cannot reach the odr-use.

2 [*Example 1*:

Translation unit #1:

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

Translation unit #2:

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

— *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, and Lénárd Szolnoki, 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

[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 McGuiness, and David Tenty, "Invalid/Prospective Pointer Operations", 2026
http://wg21.link/P3347R6

[P3424R2] Alisdair Meredith, "Deallocation Functions with Throwing Exception Specification Are Ill-formed", 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