# Undefined Behavior and IFNDR Annexes


| | |
|---|---|
| Document #: | P3596R0 |
| Date: | 2026-02-20 |
| Project: | Programming Language C++ |
| Audience: | CWG |


| | |
|---|---|
| Reply-to: | Joshua Berne <jberne4@bloomberg.net> |
| | Timur Doumler <papers@timur.audio> |
| | Jens Maurer <jens.maurer@gmx.net> |
| | Shafik Yaghmour <shafik.yaghmour@intel.com> |



**Abstract**

Programs that are ill-formed, no diagnostic required (IFNDR) or that exhibit Undefined Behavior are always potentially problematic, and so it is good to be aware of the full scope of potential places where these issues might arise. This paper includes wording for a new pair of Annexes to the C++ Standard that catalog all instances of both kinds of behavior.


## Contents

## Revision History

Revision 0

- Original version of the paper

# 1 Introduction

Cataloging all undefined and IFNDR behavior has been identified as a useful task for quite a while. An earlier attempt at gathering this information was made in [P1705R1]. Calls to formalize this data into an annex within the C++ Standard itself was made in [P2234R0] and later [P3075R0].

The wording in this annex has been developed collaboratively as a branch of the C++ working draft, which can be found here. It is now ready for review by CWG so that we can incorporate this work into the Standard and maintain it going forward.

As part of this work, tentative stable names for each potential issue have been chosen. These names are used internally within the source of the Standard in order to produce links from the main source to the new annexes and back, and have also been used in [P3100R5] that is cataloging potential approaches to mitigating each instance of undefined behavior. We hope that these stable names will be fruitful to use as a common vocabulary for any efforts to modify or mitigate specific classes of problems.

We have atempted to follow some basic guidance on what is and is not appropriate to include in this annex:

- In general the annex examples must be correct and provide an example of the specific undefined behavior or IFNDR construct that is being described, and not a piece of code that is incorrect for some other reason.
- Annex entries should be thorough as well, but should not be a complete reproduction of the core wording where the issue is defined. The text of the annex is not normative, but will likely be used as a reference.

# 2 Wording Changes

Note that there are some differences between the wording as rendered in this paper and how it will be rendered in the final Standard document:

— As with all wording instructions, green is used to mark text that will be added to the draft while ~~red~~ is for text that will be removed. In general, for any paragraphs that contain modifications the entire paragraph's contents is included for context.

— Cross-references within this wording all appear using the stable name of the section they are referencing within square brackets. In the rendered Standard that same cross-reference will show up as the section number being referenced (or, in the case of top level clauses, as the word "Clause" or "Annex" followed by the clause number or annex letter).

| In this paper | In the Standard |
|---|---|
| [lex] | Clause 5 |
| [implimits] | Annex B |
| [basic.def.odr] | 6.3 |

— Within this paper, we are assigning and displaying new (somewhat) stable identifiers for each kind of IFNDR and undefined behavior. For the moment, these will be used within the LaTeXsource of the standard to correlate the point in the main body of the Standard where the issue is introduced and its corresponding explanation in one of the annexes.

| In this paper | In the Standard |
|---|---|
| ([intro.object.implicit.create]) | (E.2.1) |
| Specified in: [intro.object.implicit.create] | Specified in: 6.8.2 |

In most pdf viewers the section and annex numbers will be clickable links that take you to the corresponding location.

— Clause, section, and paragraph numbers shown within this document strive to match those of the current draft whenever possible. New clauses, sections, and paragraph numbers are shown with a letter added to an existing number. For example, we are adding two new annexes after annex D to the standard, and they are shown below as annexes (D+a) and (D+b). In the final rendered standard these will be annexes E and F. Annexes that come after will then have their values increased, and so the current Annex E (Conformance with UAX #31) will become Annex G.

— Font sizes and margines are slightly different than the Standard, and consistently reproducing that is not viable, so be aware that word wrapping and pagination will not be precisely what gets rendered in the final Standard. Some attempt has been made to size code blocks to match what the standard will do as comments need to be manually wrapped, but various factors make that imprecise.

— Various reviewers have had open questions about both the general approach and specific entries. When the appropriate fix to the wording has not been apparent, those comments have been retained.

> JMB: Such comments will appear in the text in a box lke this one, and must be resolved prior to the completion of wording review.

— The sections in the undefined behavior annex (what will be Annex E) are all one level below a clause-specific subsection of the annex. The sections in the ifndr annex (what will be Annex F) all mirror the structure of the main document completely, with the documented behaviors described in a parallel subsection to the main section — one level deeper in the hierarchy than the corresponding main section. The flatter structure of the UB annex gives us the option to categorize behaviors and involves less effectively empty intermediate sections. A parallel structure helps us avoid having to produce new section names and organizations, and might improve navigation and discoverability. Both options provide different forms of maintenance burdens, and clearly the original producers of these documents have not so far managed to remain consistent in a choice of approach to annex structure.

JMB: Decide which approach we want for the sub-sectioning of the annexes and make them both follow that approach.

The wording changes in this paper are relative to the C++29 working draft with git hash 71dd1cd5, last modified on Tue Jan 6 21:35.

**Modified Section Contents**

## Modifications

### 5 **Lexical conventions** [lex]

### 5.11 **Identifiers** [lex.name]

Modify [lex.name], paragraph 3:

3 In addition, some identifiers appearing as a *token* or *preprocessing-token* are reserved for use by C++ implementations and shall not be used otherwise; no diagnostic is required ([lex.name.reserved]).

— Each identifier that contains a double underscore `__` or begins with an underscore followed by an uppercase letter, other than those specified in this document (for example, `__cplusplus`([cpp.predefined])), is reserved to the implementation for any use.

— Each identifier that begins with an underscore is reserved to the implementation for use as a name in the global namespace.

### 6 **Basics** [basic]

### 6.3 **One-definition rule** [basic.def.odr]

Modify [basic.def.odr], paragraph 12:

12 Every program shall contain at least one definition of every function or variable that is odr-used in that program outside of a discarded statement([stmt.if]); no diagnostic required ([basic.def.odr.exact.one.def]). The definition can appear explicitly in the program, it can be found in the standard or a user-defined library, or (when appropriate) it is implicitly defined (see [class.default.ctor], [class.copy.ctor], [class.dtor], and [class.copy.assign]).

[*Example*:

```
auto f() {
  struct A {};
  return A{};
}
decltype(f()) g();
auto x = g();
```

A program containing this translation unit is ill-formed because `g` is odr-used but not defined, and cannot be defined in any other translation unit because the local class `A` cannot be named outside this translation unit. — *end example*]

Modify [basic.def.odr], paragraph 20:

20 If, at any point in the program, there is more than one reachable unnamed enumeration definition in the same scope that have the same first enumerator name and do not have typedef names for linkage purposes([dcl.enum]), those unnamed enumeration types shall be the same; no diagnostic required([basic.def.odr.unnamed.enum.same.type]).

### 6.5 **Name lookup** [basic.lookup]

#### 6.5.2 **Member name lookup** [class.member.lookup]

Modify [class.member.lookup], paragraph 6:

6 The result of the search is the declaration set of $S(M, T)$. If it is an invalid set, the program is ill-formed. If it differs from the result of a search in $T$ for $M$ in a complete-class context([class.mem]) of $T$, the program is ill-formed, no diagnostic required([class.member.lookup.name.refers.diff.decl]).

[*Example*:

```
struct A { int x; };                    // S(x,A) = { { A::x }, { A } }
struct B { float x; };                  // S(x,B) = { { B::x }, { B } }
struct C: public A, public B { };       // S(x,C) = { invalid, { A in C, B in C } }
struct D: public virtual C { };         // S(x,D) = S(x,C)
struct E: public virtual C { char x; }; // S(x,E) = { { E::x }, { E } }
struct F: public D, public E { };       // S(x,F) = S(x,E)
int main() {
  F f;
  f.x = 0;                              // OK, lookup finds E::x
}
```

$S(\texttt{x}, \texttt{F})$ is unambiguous because the `A` and `B` base class subobjects of `D` are also base class subobjects of `E`, so $S(\texttt{x}, \texttt{D})$ is discarded in the first merge step. — *end example*]

### 6.7 **Program and linkage** [basic.link]

Modify [basic.link], paragraph 10:

10 If two declarations of an entity are attached to different modules, the program is ill-formed; no diagnostic is required if neither is reachable from the other ([basic.link.entity.same.module]).

[*Example*:

`"decls.h"`

```
int f();            // #1, attached to the global module
int g();            // #2, attached to the global module
```

Module interface of `M`

```
module;
#include "decls.h"
export module M;
export using ::f;   // OK, does not declare an entity, exports #1
int g();            // error: matches #2, but attached to M
export int h();     // #3
export int k();     // #4
```

Other translation unit

```
import M;
static int h();     // error: matches #3
int k();            // error: matches #4
```

— *end example*]

As a consequence of these rules, all declarations of an entity are attached to the same module; the entity is said to be *attached* to that module.

Modify [basic.link], paragraph 11:

11 For any two declarations of an entity $E$:

— If one declares $E$ to be a variable or function, the other shall declare $E$ as one of the same type.

— If one declares $E$ to be an enumerator, the other shall do so.

— If one declares $E$ to be a namespace, the other shall do so.

— If one declares $E$ to be a type, the other shall declare $E$ to be a type of the same kind([dcl.type.elab]).

— If one declares $E$ to be a class template, the other shall do so with the same kind and an equivalent *template-head*([temp.over.link]).

[*Note*: The declarations can supply different default template arguments. — *end note*]

— If one declares $E$ to be a function template or a (partial specialization of a) variable template, the other shall declare $E$ to be one with an equivalent *template-head* and type.

— If one declares $E$ to be an alias template, the other shall declare $E$ to be one with an equivalent *template-head* and *defining-type-id*.

— If one declares $E$ to be a concept, the other shall do so.

Types are compared after all adjustments of types (during which type aliases([dcl.typedef]) are replaced by the types they denote); declarations for an array object can specify array types that differ by the presence or absence of a major array bound([dcl.array]). No diagnostic is required if neither declaration is reachable from the other([basic.link.consistent.types]).

[*Example*:

```
int f(int x, int x);    // error: different entities for x
void g();               // #1
void g(int);            // OK, different entity from #1
int g();                // error: same entity as #1 with different type
void h();               // #2
namespace h {}          // error: same entity as #2, but not a function
```

— *end example*]

## 6.8 Memory and objects [basic.memobj]

### 6.8.2 Object model [intro.object]

Modify [intro.object], paragraph 12:

12 Some operations are described as *implicitly creating objects* within a specified region of storage. For each operation that is specified as implicitly creating objects, that operation implicitly creates and starts the lifetime of zero or more objects of implicit-lifetime types([term.implicit.lifetime.type]) in its specified region of storage if doing so would result in the program having defined behavior. If no such set of objects would give the program defined behavior([intro.object.implicit.create]), the behavior of the program is

undefined. If multiple such sets of objects would give the program defined behavior, it is unspecified which such set of objects is created.

[*Note*: Such operations do not start the lifetimes of subobjects of such objects that are not themselves of implicit-lifetime types. — *end note*]

Modify [intro.object], paragraph 13:

13 Further, after implicitly creating objects within a specified region of storage, some operations are described as producing a pointer to a *suitable created object*. These operations select one of the implicitly-created objects whose address is the address of the start of the region of storage, and produce a pointer value that points to that object, if that value would result in the program having defined behavior. If no such pointer value would give the program defined behavior, the behavior of the program is undefined([intro.object.implicit.pointer]). If multiple such pointer values would give the program defined behavior, it is unspecified which such pointer value is produced.

### 6.8.3 **Alignment** [**basic.align**]

Modify [basic.align], paragraph 1:

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

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

Modify [basic.life], paragraph 7:

7 Before the lifetime of an object has started but after the storage which the object will occupy has been allocated[1] or after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any pointer that represents the address of the storage location where the object will be or was located may be used but only in limited ways. For an object under construction or destruction, see [class.cdtor]. Otherwise, such a pointer refers to allocated storage([basic.stc.dynamic.allocation]), and using the pointer as if the pointer were of type `void*` is well-defined. Indirection through such a pointer is permitted but the resulting lvalue may only be used in limited ways, as described below. The program has undefined behavior if:

- the pointer is used as the operand of a *delete-expression* ([lifetime.outside.pointer.delete]),
- the pointer is used to access a non-static data member or call a non-static member function of the object([lifetime.outside.pointer.member]), or

— the pointer is converted([conv.ptr,expr.static.cast]) to a pointer to a virtual base class([lifetime.outside.pointer.virtual]) or to a base class thereof, or

— the pointer is used as the operand of a `dynamic_cast`([expr.dynamic.cast]) ([lifetime.outside.pointer.dynamic.cast]).

[*Example*:

```
#include <cstdlib>

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

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

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

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

— *end example*]

## Foonotes

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

Modify [basic.life], paragraph 8:

8 Similarly, before the lifetime of an object has started but after the storage which the object will occupy has been allocated or after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any glvalue that refers to the original object may be used but only in limited ways. For an object under construction or destruction, see [class.cdtor]. Otherwise, such a glvalue refers to allocated storage([basic.stc.dynamic.allocation]), and using the properties of the glvalue that do not depend on its value is well-defined. The program has undefined behavior if:

— the glvalue is used to access the object([lifetime.outside.glvalue.access]), or

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

— the glvalue is bound to a reference to a virtual base class([dcl.init.ref]) ([lifetime.outside.glvalue.virtual]), or

— the glvalue is used as the operand of a `dynamic_cast`([expr.dynamic.cast]) or as the operand of `typeid`([lifetime.outside.glvalue.dynamic.cast]).

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

Modify [basic.life], paragraph 11:

11 If a program ends the lifetime of an object of type `T` with static([basic.stc.static]), thread([basic.stc.thread]), or automatic([basic.stc.auto]) storage duration and if `T` has a non-trivial destructor,[2] and another object of the original type does not occupy that same storage location when the implicit destructor call takes place, the behavior of the program is undefined([original.type.implicit.destructor]). This is true even if the block is exited with an exception.

[*Example*:

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

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

— *end example*]

## Foonotes

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

Modify [basic.life], paragraph 12:

12 Creating a new object within the storage that a const, complete object with static, thread, or automatic storage duration occupies, or within the storage that such a const object used to occupy before its lifetime ended, results in undefined behavior([creating.within.const.complete.obj]).

[*Example*:

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

const B b;

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

— *end example*]

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

Modify [basic.indet], paragraph 2:

2 If any operand of a built-in operator that produces a prvalue is evaluated, is not a discarded-value expression([expr.context]), and produces an erroneous value, then the value produced by that operator is erroneous. Except in the following cases, if an indeterminate value is produced by an evaluation, the behavior is undefined([basic.indet.value]), and if an erroneous value is produced by an evaluation, the behavior is erroneous and the result of the evaluation is that erroneous value:

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

  then the result of the operation is an indeterminate value or that erroneous value, respectively.
- If an indeterminate or erroneous value of unsigned ordinary character type or `std::byte` type is produced by the evaluation of the right operand of a simple assignment operator([expr.assign]) whose first operand is an lvalue of unsigned ordinary character type or `std::byte` type, an indeterminate value or that erroneous value, respectively, replaces the value of the object referred to by the left operand.
- If an indeterminate or erroneous value of unsigned ordinary character type is produced by the evaluation of the initialization expression when initializing an object of unsigned ordinary character type, that object is initialized to an indeterminate value or that erroneous value, respectively.
- If an indeterminate value of unsigned ordinary character type or `std::byte` type is produced by the evaluation of the initialization expression when initializing an object of `std::byte` type, that object is initialized to an indeterminate value or that erroneous value, respectively.

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

[*Example*:

```
int f(bool b) {
  unsigned char *c = new unsigned char;
  unsigned char d = *c;         // OK, d has an indeterminate value
  int e = d;                    // undefined behavior
```

```
  return b ? d : 0;             // undefined behavior if b is true
}

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

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

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

void h() {
  int d1, d2;

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

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

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

— *end example*]

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

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

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

Modify [basic.stc.dynamic.general], paragraph 3:

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

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

Modify [basic.stc.dynamic.allocation], paragraph 2:

2 An allocation function attempts to allocate the requested amount of storage. If it is successful, it returns the address of the start of a block of storage whose length in bytes is at least as large as the requested size. The order, contiguity, and initial value of storage allocated by successive calls to an allocation function are unspecified. Even if the size of the space requested is zero, the request can fail. If the request succeeds, the value returned by a replaceable allocation function is a non-null pointer value([basic.compound]) `p0` different from any previously returned value `p1`, unless that value `p1` was subsequently passed to a replaceable deallocation function. Furthermore, for the library allocation functions in [new.delete.single] and [new.delete.array], `p0` represents the address of a block of storage disjoint from the storage for any other object accessible to the caller. The effect of indirecting through a pointer returned from a request for zero size is undefined([basic.stc.alloc.zero.dereference]).[3]

## Foonotes

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

#### 6.8.6.5.3 **Deallocation functions** **[basic.stc.dynamic.deallocation]**

Modify [basic.stc.dynamic.deallocation], paragraph 4:

4 If a deallocation function terminates by throwing an exception, the behavior is undefined ([basic.stc.alloc.dealloc.throw]). The value of the first argument supplied to a deallocation function may be a null pointer value; if so, and if the deallocation function is one supplied in the standard library, the call has no effect.

### 6.9 **Types** **[basic.types]**

#### 6.9.4 **Compound types** **[basic.compound]**

Modify [basic.compound], paragraph 4:

4 A pointer value $P$ is *valid in the context of* an evaluation $E$ if $P$ is a pointer to function or a null pointer value, or if it is a pointer to or past the end of an object $O$ and $E$ happens before the end of the duration of the region of storage for $O$. If a pointer value $P$ is used in an evaluation $E$ and $P$ is not valid in the context of $E$, then the behavior is undefined if $E$ is an indirection([expr.unary.op]) or an invocation of a deallocation function([basic.stc.dynamic.deallocation])([basic.compound.invalid.pointer]), and implementation-defined otherwise.[4]

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

## Foonotes

4. Some implementations might define that copying such a pointer value causes a system-generated runtime fault.

### 6.10 **Program execution** **[basic.exec]**

#### 6.10.1 **Sequential execution** **[intro.execution]**

Modify [intro.execution], paragraph 10:

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

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

The value computations of the operands of an operator are sequenced before the value computation of the result of the operator. The behavior is undefined([intro.multithread]) ([intro.execution.unsequenced.modification]) if

— a side effect on a memory location([intro.memory]) or

— starting or ending the lifetime of an object in a memory location

is unsequenced relative to

— another side effect on the same memory location,

— starting or ending the lifetime of an object occupying storage that overlaps with the memory location, or

— a value computation using the value of any object in the same memory location,

and the two evaluations are not potentially concurrent([intro.multithread]).

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

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

[*Example*:

```
void g(int i) {
  i = 7, i++, i++;              // i becomes 9

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

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

— *end example*]

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

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

Modify [intro.races], paragraph 17:

17 Two actions are *potentially concurrent* if

— they are performed by different threads, or

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

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

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

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

Modify [intro.progress], paragraph 1:

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

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

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

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

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

Modify [basic.start.main], paragraph 4:

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

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

Modify [basic.start.term], paragraph 5:

5 If a function contains a block variable of static or thread storage duration that has been destroyed and the function is called during the destruction of an object with static or thread storage duration, the program has undefined behavior

([basic.start.term.use.after.destruction]) if the flow of control passes through the definition of the previously destroyed block variable.

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

Modify [basic.start.term], paragraph 7:

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

[*Note*: If there is a use of an object with static storage duration that does not happen before the object's destruction, the program has undefined behavior. Terminating every thread before a call to `std::exit` or the exit from `main` is sufficient, but not necessary, to satisfy these requirements. These requirements permit thread managers as static-storage-duration objects. — *end note*]

## 6.11 Contract assertions [basic.contract]

### 6.11.1 General [basic.contract.general]

Modify [basic.contract.general], paragraph 3:

3 An invocation of the macro `va_start`([cstdarg.syn]) shall not be a subexpression of the predicate of a contract assertion, no diagnostic required ([basic.contract.vastart.contract.predicate]).

### 6.11.3 Contract-violation handler [basic.contract.handler]

Modify [basic.contract.handler], paragraph 3:

3 It is implementation-defined whether the contract-violation handler is replaceable([term.replaceable.function]). If the contract-violation handler is not replaceable, a declaration of a replacement function for the contract-violation handler is ill-formed, no diagnostic required.([basic.contract.handler.replacing.nonreplaceable])

# 7 Expressions [expr]

## 7.1 Preamble [expr.pre]

Modify [expr.pre], paragraph 4:

4 If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined([expr.expr.eval]).

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

## 7.2 **Properties of expressions** [expr.prop]

### 7.2.1 **Value category** [basic.lval]

Modify [basic.lval], paragraph 11:

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

— $\texttt{T}_{\text{obj}}$,

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

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

If a program attempts to access([defns.access]) the stored value of an object through a glvalue through which it is not type-accessible, the behavior is undefined ([expr.basic.lvalue.strict.aliasing.violation]).[5] If a program invokes a defaulted copy/-move constructor or copy/move assignment operator for a union of type `U` with a glvalue argument that does not denote an object of type *cv* `U` within its lifetime, the behavior is undefined([expr.basic.lvalue.union.initialization]).

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

## Foonotes

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

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

Modify [expr.type], paragraph 1:

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

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

## 7.3 **Standard conversions** [conv]

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

Modify [conv.lval], paragraph 3:

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

— If `T` is *cv* `std::nullptr_t`, the result is a null pointer constant([conv.ptr]).

[*Note*: Since the conversion does not access the object to which the glvalue refers, there is no side effect even if `T` is volatile-qualified([intro.execution]), and the glvalue can refer to an inactive member of a union([class.union]). — *end note*]

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

— Otherwise, if the object to which the glvalue refers contains an invalid pointer value([basic.compound]), the behavior is implementation-defined.

— Otherwise, if the bits in the value representation of the object to which the glvalue refers are not valid for the object's type, the behavior is undefined ([conv.lval.valid.representation]).

[*Example*:

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

— *end example*]

— Otherwise, the object indicated by the glvalue is read([defns.access]). Let `V` be the value contained in the object. If `T` is an integer type, the prvalue result is the value of type `T` congruent([basic.fundamental]) to `V`, and `V` otherwise.

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

Modify [conv.double], paragraph 2:

2 If the source value can be exactly represented in the destination type, the result of the conversion is that exact representation. If the source value is between two adjacent destination values, the result of the conversion is an implementation-defined choice of either of those values. Otherwise, the behavior is undefined([conv.double.out.of.range]).

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

Modify [conv.fpint], paragraph 1:

1 A prvalue of a floating-point type can be converted to a prvalue of an integer type. The conversion truncates; that is, the fractional part is discarded. The behavior is undefined([conv.fpint.float.not.represented]) if the truncated value cannot be represented in the destination type.

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

Modify [conv.fpint], paragraph 2:

2 A prvalue of an integer type or of an unscoped enumeration type can be converted to a prvalue of a floating-point type. The result is exact if possible. If the value being converted is in the range of values that can be represented but the value cannot be

represented exactly, it is an implementation-defined choice of either the next lower or higher representable value.

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

If the value being converted is outside the range of values that can be represented, the behavior is undefined([conv.fpint.int.not.represented]). If the source type is `bool`, the value `false` is converted to zero and the value `true` is converted to one.

### 7.3.12 **Pointer conversions** **[conv.ptr]**

Modify [conv.ptr], paragraph 3:

3 A prvalue `v` of type "pointer to *cv* `D`", where `D` is a complete class type, can be converted to a prvalue of type "pointer to *cv* `B`", where `B` is a base class([class.derived]) of `D`. If `B` is an inaccessible([class.access]) or ambiguous([class.member.lookup]) base class of `D`, a program that necessitates this conversion is ill-formed. If `v` is a null pointer value, the result is a null pointer value. Otherwise, if `B` is a virtual base class of `D` or is a base class of a virtual base class of `D` and `v` does not point to an object whose type is similar([conv.qual]) to `D` and that is within its lifetime or within its period of construction or destruction([class.cdtor]), the behavior is undefined([conv.ptr.virtual.base]). Otherwise, the result is a pointer to the base class subobject of the derived class object.

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

Modify [conv.mem], paragraph 2:

2 A prvalue of type "pointer to member of `B` of type *cv* `T`", where `B` is a class type, can be converted to a prvalue of type "pointer to member of `D` of type *cv* `T`", where `D` is a complete class derived([class.derived]) from `B`. If `B` is an inaccessible([class.access]), ambiguous([class.member.lookup]), or virtual([class.mi]) base class of `D`, or a base class of a virtual base class of `D`, a program that necessitates this conversion is ill-formed. If class `D` does not contain the original member and is not a base class of the class containing the original member, the behavior is undefined([conv.member.missing.member]). Otherwise, the result of the conversion refers to the same member as the pointer to member before the conversion took place, but it refers to the base class member as if it were a member of the derived class. The result refers to the member in `D`'s instance of `B`. Since the result has type "pointer to member of `D` of type *cv* `T`", indirection through it with a `D` object is valid. The result is the same as if indirecting through the pointer to member of `B` with the `B` subobject of `D`. The null member pointer value is converted to the null member pointer value of the destination type.[6]

## Foonotes

6. The rule for conversion of pointers to members (from pointer to member of base to pointer to member of derived) appears inverted compared to the rule for pointers to objects (from pointer to derived to pointer to base)([conv.ptr,class.derived]). This inversion is necessary to ensure type safety. Note that a pointer to

member is not an object pointer or a function pointer and the rules for conversions of such pointers do not apply to pointers to members. In particular, a pointer to member cannot be converted to a `void*`.

## 7.5 Primary expressions [expr.prim]

### 7.5.8 Requires expressions [expr.prim.req]

#### 7.5.8.1 General [expr.prim.req.general]

Modify [expr.prim.req.general], paragraph 5:

5 The substitution of template arguments into a *requires-expression* can result in the formation of invalid types or expressions in the immediate context of its *requirement*s([temp.deduct.general]) or the violation of the semantic constraints of those *requirement*s. In such cases, the *requires-expression* evaluates to `false`; it does not cause the program to be ill-formed. The substitution and semantic constraint checking proceeds in lexical order and stops when a condition that determines the result of the *requires-expression* is encountered. If substitution (if any) and semantic constraint checking succeed, the *requires-expression* evaluates to `true`.

[*Note*: If a *requires-expression* contains invalid types or expressions in its *requirement*s, and it does not appear within the declaration of a templated entity, then the program is ill-formed. — *end note*]

If the substitution of template arguments into a *requirement* would always result in a substitution failure, the program is ill-formed; no diagnostic required ([expr.prim.req.always.sub.fail]).

[*Example*:

```
template<typename T> concept C =
requires {
  new decltype((void)T{});      // ill-formed, no diagnostic required
};
```

— *end example*]

## 7.6 Compound expressions [expr.compound]

### 7.6.1 Postfix expressions [expr.post]

#### 7.6.1.3 Function call [expr.call]

Modify [expr.call], paragraph 5:

5 A type $\texttt{T}_{\text{call}}$ is *call-compatible* with a function type $\texttt{T}_{\text{func}}$ if $\texttt{T}_{\text{call}}$ is the same type as $\texttt{T}_{\text{func}}$ or if the type "pointer to $\texttt{T}_{\text{func}}$" can be converted to type "pointer to $\texttt{T}_{\text{call}}$" via a function pointer conversion([conv.fctptr]). Calling a function through an expression whose function type is not call-compatible with the type of the called function's definition results in undefined behavior([expr.call.different.type]).

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

#### 7.6.1.5 **Class member access** [expr.ref]

Modify [expr.ref], paragraph 10:

10 If E2 designates a non-static member (possibly after overload resolution) or direct base class relationship and the result of E1 is an object whose type is not similar([conv.qual]) to the type of E1, the behavior is undefined([expr.ref.member.not.similar]).

[*Example*:

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

— *end example*]

#### 7.6.1.7 **Dynamic cast** [expr.dynamic.cast]

Modify [expr.dynamic.cast], paragraph 7:

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

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

Modify [expr.static.cast], paragraph 2:

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

[*Example*:

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

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

— *end example*]

Modify [expr.static.cast], paragraph 8:

8 A value of integral or enumeration type can be explicitly converted to a complete enumeration type. If the enumeration type has a fixed underlying type, the value is first converted to that type by integral promotion([conv.prom]) or integral conversion([conv.integral]), if necessary, and then to the enumeration type. If the enumeration type does not have a fixed underlying type, the value is unchanged if the original value is within the range of the enumeration values([dcl.enum]), and otherwise, the behavior is undefined([expr.static.cast.enum.outside.range]). A value of floating-point type can also be explicitly converted to an enumeration type. The resulting value is the same as converting the original value to the underlying type of the enumeration([conv.fpint]), and subsequently to the enumeration type.

Modify [expr.static.cast], paragraph 9:

9 A prvalue of floating-point type can be explicitly converted to any other floating-point type. If the source value can be exactly represented in the destination type, the result of the conversion has that exact representation. If the source value is between two adjacent destination values, the result of the conversion is an implementation-defined choice of either of those values. Otherwise, the behavior is undefined([expr.static.cast.fp.outside.range]).

Modify [expr.static.cast], paragraph 10:

10 A prvalue of type "pointer to *cv1* `B`", where `B` is a class type, can be converted to a prvalue of type "pointer to *cv2* `D`", where `D` is a complete class derived([class.derived]) from `B`, if *cv2* is the same cv-qualification as, or greater cv-qualification than, *cv1*. If `B` is a virtual base class of `D` or a base class of a virtual base class of `D`, or if no valid standard conversion from "pointer to `D`" to "pointer to `B`" exists([conv.ptr]), the program is ill-formed. The null pointer value([basic.compound]) is converted to the null pointer value of the destination type. If the prvalue of type "pointer to *cv1* `B`" points to a `B` that is actually a base class subobject of an object of type `D`, the resulting pointer points to the enclosing object of type `D`. Otherwise, the behavior is undefined([expr.static.cast.downcast.wrong.derived.type]).

Modify [expr.static.cast], paragraph 11:

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

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

If no valid standard conversion from "pointer to member of `B` of type `T`" to "pointer to member of `D` of type `T`" exists([conv.mem]), the program is ill-formed. The null member pointer value([conv.mem]) is converted to the null member pointer value of the destination type. If class `B` contains the original member, or is a base class of the class containing the original member, the resulting pointer to member points to the original member. Otherwise, the behavior is undefined([expr.static.cast.does.not.contain.orignal.member]).

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

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

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

Modify [expr.unary.op], paragraph 1:

1 The unary `*` operator performs *indirection*. Its operand shall be a prvalue of type "pointer to `T`", where `T` is an object or function type. The operator yields an lvalue of type `T`. If the operand points to an object or function, the result denotes that object or function; otherwise, the behavior is undefined except as specified in [expr.typeid] ([expr.unary.dereference]).

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

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

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

Modify [expr.new], paragraph 22:

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

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

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

Modify [expr.delete], paragraph 2:

2 In a single-object delete expression, the value of the operand of `delete` may be a null pointer value, a pointer value that resulted from a previous non-array *new-expression*, or a pointer to a base class subobject of an object created by such a *new-expression*. If not, the behavior is undefined([expr.delete.mismatch]). In an array delete expression, the value of the operand of `delete` may be a null pointer value or a pointer value that resulted from a previous array *new-expression* whose allocation function was not a non-allocating form([new.delete.placement]).[7] If not, the behavior is undefined([expr.delete.array.mismatch]).

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

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

## Foonotes

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

Modify [expr.delete], paragraph 3:

3 In a single-object delete expression, if the static type of the object to be deleted is not similar([conv.qual]) to its dynamic type and the selected deallocation function (see below) is not a destroying operator delete, the static type shall be a base class of the dynamic type of the object to be deleted and the static type shall have a virtual destructor or the behavior is undefined([expr.delete.dynamic.type.differ]). In an array delete expression, if the dynamic type of the object to be deleted is not similar to its static type, the behavior is undefined([expr.delete.dynamic.array.dynamic.type.differ]).

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

Modify [expr.mptr.oper], paragraph 4:

4 Abbreviating *pm-expression*`.*`*cast-expression* as `E1.*E2`, `E1` is called the *object expression*. If the result of `E1` is an object whose type is not similar to the type of `E1`, or whose most derived object does not contain the member to which `E2` refers, the behavior is undefined([expr.mptr.oper.not.contain.member]). The expression `E1` is sequenced before the expression `E2`.

Modify [expr.mptr.oper], paragraph 6:

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

[*Example*:

```
(ptr_to_obj->*ptr_to_mfct)(10);
```

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

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

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

Modify [expr.mul], paragraph 4:

4 The binary `/` operator yields the quotient, and the binary `%` operator yields the remainder from the division of the first expression by the second. If the second operand of `/` or `%` is zero, the behavior is undefined([expr.mul.div.by.zero]). For integral operands, the `/` operator yields the algebraic quotient with any fractional part discarded;[8] if the quotient `a/b` is representable in the type of the result, `(a/b)*b + a%b` is equal to `a`; otherwise, the behavior of both `a/b` and `a%b` is undefined([expr.mul.representable.type.result]).

## Foonotes

8. This is often called truncation towards zero.

### 7.6.6 **Additive operators** **[expr.add]**

Modify [expr.add], paragraph 4:

4 When an expression `J` that has integral type is added to or subtracted from an expression `P` of pointer type, the result has the type of `P`.

— If `P` evaluates to a null pointer value and `J` evaluates to 0, the result is a null pointer value.

— Otherwise, if `P` points to a (possibly-hypothetical) array element $i$ of an array object `x` with $n$ elements([dcl.array]),[9] the expressions `P + J` and `J + P` (where `J` has the value $j$) point to the (possibly-hypothetical) array element $i+j$ of `x` if $0 \leq i+j \leq n$ and the expression `P - J` points to the (possibly-hypothetical) array element $i-j$ of `x` if $0 \leq i-j \leq n$.

— Otherwise, the behavior is undefined([expr.add.out.of.bounds]).

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

## Foonotes

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

Modify [expr.add], paragraph 5:

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

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

— 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*: If the value $i - j$ is not in the range of representable values of type `std::ptrdiff_t`, the behavior is undefined([expr.pre]). — *end note*]

— Otherwise, the behavior is undefined([expr.add.sub.diff.pointers]).

Modify [expr.add], paragraph 6:

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

[*Example*:

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

— *end example*]

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

Modify [expr.shift], paragraph 1:

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

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

The operands shall be prvalues of integral or unscoped enumeration type and integral promotions are performed. The type of the result is that of the promoted left operand. The behavior is undefined([expr.shift.neg.and.width]) if the right operand is negative, or greater than or equal to the width of the promoted left operand.

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

Modify [expr.assign], paragraph 7:

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

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

## 8 **Statements** [stmt]

### 8.8 **Jump statements** [stmt.jump]

#### 8.8.4 **The `return` statement** [stmt.return]

Modify [stmt.return], paragraph 4:

4 Flowing off the end of a constructor, a destructor, or a non-coroutine function with a *cv* `void` return type is equivalent to a `return` with no operand. Otherwise, flowing off the end of a function that is neither `main`([basic.start.main]) nor a coroutine([dcl.fct.def.coroutine]) results in undefined behavior([stmt.return.flow.off]).

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

Modify [stmt.return.coroutine], paragraph 3:

3 If a search for the name `return_void` in the scope of the promise type finds any declarations, flowing off the end of a coroutine's *function-body* is equivalent to a `co_return` with no operand; otherwise flowing off the end of a coroutine's *function-body* results in undefined behavior([stmt.return.coroutine.flow.off]).

## 8.10 Declaration statement [stmt.dcl]

Modify [stmt.dcl], paragraph 3:

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

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

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

[*Example*:

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

— *end example*]

## 8.11 Ambiguity resolution [stmt.ambig]

Modify [stmt.ambig], paragraph 3:

3 The disambiguation is purely syntactic; that is, the meaning of the names occurring in such a statement, beyond whether they are *type-name*s or not, is not generally used in or changed by the disambiguation. Class templates are instantiated as necessary to determine if a qualified name is a *type-name*. Disambiguation precedes parsing, and a statement disambiguated as a declaration may be an ill-formed declaration. If, during parsing, lookup finds that a name in a template argument is bound to (part of) the declaration being parsed, the program is ill-formed. No diagnostic is required([stmt.ambig.bound.diff.parse]).

[*Example*:

```
struct T1 {
  T1 operator()(int x) { return T1(x); }
  int operator=(int x) { return x; }
  T1(int) { }
};
struct T2 { T2(int) { } };
int a, (*(*b)(T2))(int), c, d;

void f() {
  // disambiguation requires this to be parsed as a declaration:
  T1(a) = 3,
  T2(4),                        // T2 will be declared as a variable of type T1, but this will not
  (*(*b)(T2(c)))(int(d));       // allow the last part of the declaration to parse properly,
                                // since it depends on T2 being a type-name
}
```

— *end example*]

## 9 **Declarations** [dcl]

### 9.2 **Specifiers** [dcl.spec]

#### 9.2.7 **The `constinit` specifier** [dcl.constinit]

Modify [dcl.constinit], paragraph 1:

1 The `constinit` specifier shall be applied only to a declaration of a variable with static or thread storage duration or to a structured binding declaration([dcl.struct.bind]).

[*Note*: A structured binding declaration introduces a uniquely named variable, to which the `constinit` specifier applies. — *end note*]

If the specifier is applied to any declaration of a variable, it shall be applied to the initializing declaration. No diagnostic is required if no `constinit` declaration is reachable at the point of the initializing declaration([dcl.constinit.specifier.not.reachable]).

#### 9.2.8 **The `inline` specifier** [dcl.inline]

Modify [dcl.inline], paragraph 4:

4 If a definition of a function or variable is reachable at the point of its first declaration as inline, the program is ill-formed. If a function or variable with external or module linkage is declared inline in one definition domain, an inline declaration of it shall be reachable from the end of every definition domain in which it is declared; no diagnostic is required([dcl.inline.missing.on.definition]).

[*Note*: A call to an inline function or a use of an inline variable can be encountered before its definition becomes reachable in a translation unit. — *end note*]

#### 9.2.9 **Type specifiers** [dcl.type]

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

Modify [dcl.type.cv], paragraph 4:

4 Any attempt to modify([expr.assign,expr.post.incr,expr.pre.incr]) a const object([basic.type.qualifier]) during its lifetime([basic.life]) results in undefined behavior([dcl.type.cv.modify.const.obj]).

[*Example*:

```
const int ci = 3;                       // cv-qualified (initialized as required)
ci = 4;                                 // error: attempt to modify const

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

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

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

For another example,

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

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

— *end example*]

Modify [dcl.type.cv], paragraph 5:

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

## 9.3 **Declarators** [dcl.decl]

### 9.3.4 **Meaning of declarators** [dcl.meaning]

#### 9.3.4.3 **References** [dcl.ref]

Modify [dcl.ref], paragraph 6:

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

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

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

[*Example*:

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

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

— *end example*]

### 9.3.4.7 **Default arguments** **[dcl.fct.default]**

Modify [dcl.fct.default], paragraph 4:

4 For non-template functions, default arguments can be added in later declarations of a function that have the same host scope. Declarations that have different host scopes have completely distinct sets of default arguments. That is, declarations in inner scopes do not acquire default arguments from declarations in outer scopes, and vice versa. In a given function declaration, each parameter subsequent to a parameter with a default argument shall have a default argument supplied in this or a previous declaration, unless the parameter was expanded from a parameter pack, or shall be a function parameter pack.

[*Note*: A default argument cannot be redefined by a later declaration (not even to the same value)([basic.def.odr]). — *end note*]

[*Example*:

```
void g(int = 0, ...);           // OK, ellipsis is not a parameter so it can follow
                                // a parameter with a default argument
void f(int, int);
void f(int, int = 7);
void h() {
  f(3);                         // OK, calls f(3, 7)
  void f(int = 1, int);         // error: does not use default from surrounding scope
}
void m() {
  void f(int, int);             // has no defaults
  f(4);                         // error: wrong number of arguments
  void f(int, int = 5);         // OK
  f(4);                         // OK, calls f(4, 5);
  void f(int, int = 5);         // error: cannot redefine, even to same value
}
void n() {
  f(6);                         // OK, calls f(6, 7)
}
template<class ... T> struct C {
```

```
  void f(int n = 0, T...);
};
C<int> c;                       // OK, instantiates declaration void C::f(int n = 0, int)
```

— *end example*]

For a given inline function defined in different translation units, the accumulated sets of default arguments at the end of the translation units shall be the same; no diagnostic is required([dcl.fct.default.inline.same.defaults]). If a friend declaration $D$ specifies a default argument expression, that declaration shall be a definition and there shall be no other declaration of the function or function template which is reachable from $D$ or from which $D$ is reachable.

## 9.4 **Function contract specifiers** [dcl.contract]

### 9.4.1 **General** [dcl.contract.func]

Modify [dcl.contract.func], paragraph 4:

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. 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([dcl.contract.func.mismatched.contract.specifiers]).

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

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

Modify [dcl.fct.def.coroutine], paragraph 9:

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

Modify [dcl.fct.def.coroutine], paragraph 12:

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

### 9.6.5 **Replaceable function definitions** **[dcl.fct.def.replace]**

Modify [dcl.fct.def.replace], paragraph 1:

1 Certain functions for which a definition is supplied by the implementation are *replaceable*. A C++ program may provide a definition with the signature of a replaceable function, called a *replacement function*. The replacement function is used instead of the default version supplied by the implementation. Such replacement occurs prior to program startup([basic.def.odr,basic.start]). A declaration of the replacement function

- — shall not be inline,
- — shall be attached to the global module,
- — shall have C++ language linkage,
- — shall have the same return type as the replaceable function, and
- — if the function is declared in a standard library header, shall be such that it would be valid as a redeclaration of the declaration in that header;

no diagnostic is required([dcl.fct.def.replace.bad.replacement]).

[*Note*: The one-definition rule([basic.def.odr]) applies to the definitions of a replaceable function provided by the program. The implementation-supplied function definition is an otherwise-unnamed function with no linkage. — *end note*]

## 9.12 **Linkage specifications** **[dcl.link]**

Modify [dcl.link], paragraph 6:

6 If two declarations of an entity give it different language linkages, the program is ill-formed; no diagnostic is required if neither declaration is reachable from the other ([dcl.link.mismatched.language.linkage]). A redeclaration of an entity without a linkage specification inherits the language linkage of the entity and (if applicable) its type.

## 9.13 **Attributes** **[dcl.attr]**

### 9.13.2 **Alignment specifier** **[dcl.align]**

Modify [dcl.align], paragraph 6:

6 If the defining declaration of an entity has an *alignment-specifier*, any non-defining declaration of that entity shall either specify equivalent alignment or have no *alignment-specifier*. Conversely, if any declaration of an entity has an *alignment-specifier*, every defining declaration of that entity shall specify an equivalent alignment. No diagnostic is required([dcl.align.diff.translation.units]) if declarations of an entity have different *alignment-specifier*s in different translation units.

[*Example*:

```
// Translation unit #1:
struct S { int x; } s, *p = &s;

// Translation unit #2:
```

```
struct alignas(16) S;          // ill-formed, no diagnostic required: definition of S lacks alignment
extern S* p;
```

— *end example*]

### 9.13.3 **Assumption attribute** [dcl.attr.assume]

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

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

( *conditional-expression* )

The expression is contextually converted to `bool`([conv.general]). The expression is not evaluated. If the converted expression would evaluate to `true` at the point where the assumption appears, the assumption has no effect. Otherwise, evaluation of the assumption has runtime-undefined behavior([dcl.attr.assume.false]).

### 9.13.6 **Indeterminate storage** [dcl.attr.indet]

Modify [dcl.attr.indet], paragraph 2:

2 If a function parameter is declared with the `indeterminate` attribute, it shall be so declared in the first declaration of its function. If a function parameter is declared with the `indeterminate` attribute in the first declaration of its function in one translation unit and the same function is declared without the `indeterminate` attribute on the same parameter in its first declaration in another translation unit, the program is ill-formed, no diagnostic required([dcl.attr.indet.mismatched.declarations]).

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

Modify [dcl.attr.noreturn], paragraph 1:

1 The *attribute-token* `noreturn` specifies that a function does not return. No *attribute-argument-clause* shall be present. The attribute may be applied to a function or a lambda call operator. The first declaration of a function shall specify the `noreturn` attribute if any declaration of that function specifies the `noreturn` attribute. If a function is declared with the `noreturn` attribute in one translation unit and the same function is declared without the `noreturn` attribute in another translation unit, the program is ill-formed, no diagnostic required([dcl.attr.noreturn.trans.unit.mismatch]).

Modify [dcl.attr.noreturn], paragraph 2:

2 If a function `f` is invoked where `f` was previously declared with the `noreturn` attribute and that invocation eventually returns, the behavior is runtime-undefined ([dcl.attr.noreturn.eventually.returns]).

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

## 10 **Modules** [module]

### 10.1 **Module units and purviews** [module.unit]

Modify [module.unit], paragraph 1:

1 A *module unit* is a translation unit that contains a *module-declaration*. A *named module* is the collection of module units with the same *module-name*. The identifiers `module` and `import` shall not appear as *identifier*s in a *module-name* or *module-partition*. All *module-name*s either beginning with an *identifier* consisting of `std` followed by zero or more *digit*s or containing a reserved identifier([lex.name]) are reserved and shall not be specified in a *module-declaration*; no diagnostic is required([module.unit.reserved.identifiers]). If any *identifier* in a reserved *module-name* is a reserved identifier, the module name is reserved for use by C++ implementations; otherwise it is reserved for future standardization. The optional *attribute-specifier-seq* appertains to the *module-declaration*.

Modify [module.unit], paragraph 2:

2 A *module interface unit* is a module unit whose *module-declaration* starts with *export-keyword*; any other module unit is a *module implementation unit*. A named module shall contain exactly one module interface unit with no *module-partition*, known as the *primary module interface unit* of the module; no diagnostic is required([module.unit.named.module.no.partition]).

Modify [module.unit], paragraph 3:

3 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([module.import]). No diagnostic is required for a violation of these rules([module.unit.unexported.module.partition]).

[*Note*: 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*]

### 10.5 **Private module fragment** [module.private.frag]

Modify [module.private.frag], paragraph 1:

1 A *private-module-fragment* shall appear only in a primary module interface unit([module.unit]). A module unit with a *private-module-fragment* shall be the only module unit of its module; no diagnostic is required([module.private.frag.other.module.units]).

## 11 **Classes** [class]

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

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

Modify [class.dtor], paragraph 18:

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

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

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

### 11.7.3 **Virtual functions** [**class.virtual**]

Modify [class.virtual], paragraph 12:

12 A virtual function declared in a class shall be defined, or declared pure([class.abstract]) in that class, or both; no diagnostic is required([basic.def.odr]) ([class.virtual.pure.or.defined]).

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

Modify [class.abstract], paragraph 6:

6 Member functions can be called from a constructor (or destructor) of an abstract class; the effect of making a virtual call([class.virtual]) to a pure virtual function directly or indirectly for the object being created (or destroyed) from such a constructor (or destructor) is undefined([class.abstract.pure.virtual]).

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

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

Modify [class.base.init], paragraph 6:

6 A *mem-initializer-list* can delegate to another constructor of the constructor's class using any *class-or-decltype* that denotes the constructor's class itself. If a *mem-initializer-id* designates the constructor's class, it shall be the only *mem-initializer*; the constructor is a *delegating constructor*, and the constructor selected by the *mem-initializer* is the *target constructor*. The target constructor is selected by overload resolution. Once the target constructor returns, the body of the delegating constructor is executed. If a constructor delegates to itself directly or indirectly, the program is ill-formed, no diagnostic required([class.base.init.delegate.itself]).

[*Example*:

```
struct C {
  C( int ) { }                 // #1: non-delegating constructor
  C(): C(42) { }               // #2: delegates to #1
  C( char c ) : C(42.0) { }    // #3: ill-formed due to recursion with #4
  C( double d ) : C('a') { }   // #4: ill-formed due to recursion with #3
};
```

— *end example*]

Modify [class.base.init], paragraph 16:

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

— a *ctor-initializer* (or in a function called directly or indirectly from a *ctor-initializer*) before all the *mem-initializer*s for base classes have completed,

— a precondition assertion of a constructor, or

— a postcondition assertion of a destructor([dcl.contract.func]),

the program has undefined behavior([class.base.init.mem.fun]).

[*Example*:

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

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

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

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

— *end example*]

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

Modify [class.cdtor], paragraph 1:

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

[*Example*:

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

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

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

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

For another example,

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

— *end example*]

Modify [class.cdtor], paragraph 3:

3 To explicitly or implicitly convert a pointer (a glvalue) referring to an object of class `X` to a pointer (reference) to a direct or indirect base class `B` of `X`, the construction of `X` and the construction of all of its direct or indirect bases that directly or indirectly derive from `B` shall have started and the destruction of these classes shall not have completed, otherwise the conversion results in undefined behavior([class.cdtor.convert.pointer]). To form a pointer to (or access the value of) a direct non-static member of an object `obj`, the construction of `obj` shall have started and its destruction shall not have completed, otherwise the computation of the pointer value (or accessing the member value) results in undefined behavior([class.cdtor.form.pointer]).

[*Example*:

```
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*]

Modify [class.cdtor], paragraph 4:

4 Member functions, including virtual functions([class.virtual]), can be called during construction or destruction([class.base.init]). When a virtual function is called directly or indirectly from a constructor or from a destructor, including during the construction or destruction of the class's non-static data members, or during the evaluation of a postcondition

assertion of a constructor or a precondition assertion of a destructor([dcl.contract.func]), and the object to which the call applies is the object (call it `x`) under construction or destruction, the function called is the final overrider in the constructor's or destructor's class and not one overriding it in a more-derived class. If the virtual function call uses an explicit class member access([expr.ref]) and the object expression refers to the complete object of `x` or one of that object's base class subobjects but not `x` or one of its base class subobjects, the behavior is undefined([class.cdtor.virtual.not.x]).

[*Example*:

```
struct V {
  virtual void f();
  virtual void g();
};

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

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

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

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

— *end example*]

Modify [class.cdtor], paragraph 5:

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

Modify [class.cdtor], paragraph 6:

6 `dynamic_casts`([expr.dynamic.cast]) can be used during construction or destruction([class.base.init]). When a `dynamic_cast` is used in a constructor (including the *mem-initializer* or default member initializer for a non-static data member) or in a destructor, or used in a function called (directly or indirectly) from a constructor or destructor, if the operand of the `dynamic_cast` refers to the object under construction or destruction, this object is considered to be a most derived object that has the type of the constructor

or destructor's class. If the operand of the `dynamic_cast` refers to the object under construction or destruction and the static type of the operand is not a pointer to or object of the constructor or destructor's own class or one of its bases, the `dynamic_cast` results in undefined behavior([class.cdtor.dynamic.cast]).

[*Example*:

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

struct A : virtual V { };

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

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

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

— *end example*]

## 12 **Overloading** [over]

### 12.6 **User-defined literals** [over.literal]

Modify [over.literal], paragraph 1:

1 The *user-defined-string-literal* in a *literal-operator-id* shall have no *encoding-prefix*. The *unevaluated-string* or *user-defined-string-literal* shall be empty. The *ud-suffix* of the *user-defined-string-literal* or the *identifier* in a *literal-operator-id* is called a *literal suffix identifier*. The first form of *literal-operator-id* is deprecated([depr.lit]). Some literal suffix identifiers are reserved for future standardization; see [usrlit.suffix]. A declaration whose *literal-operator-id* uses such a literal suffix identifier is ill-formed, no diagnostic required([over.literal.reserved]).

## 13 **Templates** [temp]

### 13.1 **Preamble** [temp.pre]

Modify [temp.pre], paragraph 10:

10 A definition of a function template, member function of a class template, variable template, or static data member of a class template shall be reachable from the end of every definition domain([basic.def.odr]) in which it is implicitly instantiated([temp.inst]) unless the corresponding specialization is explicitly instantiated([temp.explicit]) in some translation unit; no diagnostic is required([temp.pre.reach.def]).

## 13.4 Template arguments [temp.arg]

### 13.4.4 Template template arguments [temp.arg.template]

Modify [temp.arg.template], paragraph 2:

2 Any partial specializations([temp.spec.partial]) associated with the primary template are considered when a specialization based on the template template parameter is instantiated. If a specialization is not reachable from the point of instantiation, and it would have been selected had it been reachable, the program is ill-formed, no diagnostic required([temp.arg.template.sat.constraints]).

[*Example*:

```
template<class T> class A {      // primary template
  int x;
};
template<class T> class A<T*> { // partial specialization
  long x;
};
template<template<class U> class V> class C {
  V<int>  y;
  V<int*> z;
};
C<A> c;             // V<int> within C<A> uses the primary template, so c.y.x has type int
                    // V<int*> within C<A> uses the partial specialization, so c.z.x has type long
```

— *end example*]

## 13.5 Template constraints [temp.constr]

### 13.5.2 Constraints [temp.constr.constr]

#### 13.5.2.3 Atomic constraints [temp.constr.atomic]

Modify [temp.constr.atomic], paragraph 2:

2 Two atomic constraints, $e_1$ and $e_2$, are *identical* if they are formed from the same appearance of the same *expression* and if, given a hypothetical template $A$ whose *template-parameter-list* consists of *template-parameter*s corresponding and equivalent([temp.over.link]) to those mapped by the parameter mappings of the expression, a *template-id* naming $A$ whose *template-argument*s are the targets of the parameter mapping of $e_1$ is the same([temp.type]) as a *template-id* naming $A$ whose *template-argument*s are the targets of the parameter mapping of $e_2$.

[*Note*: The comparison of parameter mappings of atomic constraints operates in a manner similar to that of declaration matching with alias template substitution([temp.alias]).

[*Example*:

```
template <unsigned N> constexpr bool Atomic = true;
template <unsigned N> concept C = Atomic<N>;
template <unsigned N> concept Add1 = C<N + 1>;
template <unsigned N> concept AddOne = C<N + 1>;
template <unsigned M> void f()
  requires Add1<2 * M>;
template <unsigned M> int f()
  requires AddOne<2 * M> && true;

int x = f<0>();     // OK, the atomic constraints from concept C in both fs are Atomic<N>
                    // with mapping similar to N ↦ 2 * M + 1
```

```
template <unsigned N> struct WrapN;
template <unsigned N> using Add1Ty = WrapN<N + 1>;
template <unsigned N> using AddOneTy = WrapN<N + 1>;
template <unsigned M> void g(Add1Ty<2 * M> *);
template <unsigned M> void g(AddOneTy<2 * M> *);

void h() {
  g<0>(nullptr);    // OK, there is only one g
}
```

— *end example*]

As specified in [temp.over.link], if the validity or meaning of the program depends on whether two constructs are equivalent, and they are functionally equivalent but not equivalent, the program is ill-formed, no diagnostic required ([temp.constr.atomic.equiv.but.not.equiv]).

[*Example*:

```
template <unsigned N> void f2()
  requires Add1<2 * N>;
template <unsigned N> int f2()
  requires Add1<N * 2> && true;
void h2() {
  f2<0>();          // ill-formed, no diagnostic required:
                    // requires determination of subsumption between atomic constraints that are
                    // functionally equivalent but not equivalent
}
```

— *end example*]

— *end note*]

Modify [temp.constr.atomic], paragraph 3:

3 To determine if an atomic constraint is *satisfied*, the parameter mapping and template arguments are first substituted into its expression. If substitution results in an invalid type or expression in the immediate context of the atomic constraint([temp.deduct.general]), the constraint is not satisfied. Otherwise, the lvalue-to-rvalue conversion([conv.lval]) is performed if necessary, and `E` shall be a constant expression of type `bool`. The constraint is satisfied if and only if evaluation of `E` results in `true`. If, at different points in the program, the satisfaction result is different for identical atomic constraints and template arguments, the program is ill-formed, no diagnostic required([temp.constr.atomic.sat.result.diff]).

[*Example*:

```
template<typename T> concept C =
  sizeof(T) == 4 && !true;      // requires atomic constraints sizeof(T) == 4 and !true

template<typename T> struct S {
  constexpr operator bool() const { return true; }
};

template<typename T> requires (S<T>{})
void f(T);                      // #1
void f(int);                    // #2

void g() {
  f(0);                         // error: expression S<int>{} does not have type bool
}                               // while checking satisfaction of deduced arguments of #1;
                                // call is ill-formed even though #2 is a better match
```

— *end example*]

### 13.5.4 **Constraint normalization** **[temp.constr.normal]**

Modify [temp.constr.normal], paragraph 1:

1 The *normal form* of an *expression* `E` is a constraint([temp.constr.constr]) that is defined as follows:

- The normal form of an expression `( E )` is the normal form of `E`.
- The normal form of an expression `E1 || E2` is the disjunction([temp.constr.op]) of the normal forms of `E1` and `E2`.
- The normal form of an expression `E1 && E2` is the conjunction of the normal forms of `E1` and `E2`.
- For a concept-id `C<`$A_1$`, `$A_2$`, ..., `$A_n$`>` termed `CI`:
  - If `C` names a dependent concept, the normal form of `CI` is a concept-dependent constraint whose concept-id is `CI` and whose parameter mapping is the identity mapping.
  - Otherwise, to form `CE`, any non-dependent concept template argument $A_i$ is substituted into the *constraint-expression* of `C`. If any such substitution results in an invalid concept-id, the program is ill-formed; no diagnostic is required. The normal form of `CI` is the result of substituting, in the normal form `N` of `CE`, appearances of `C`'s template parameters in the parameter mappings of the atomic constraints in `N` with their respective arguments from `C`. If any such substitution results in an invalid type or expression, the program is ill-formed; no diagnostic is required.

  [*Example*:

  ```
  template<typename T> concept A = T::value || true;
  template<typename U> concept B = A<U*>;
  template<typename V> concept C = B<V&>;
  ```

  Normalization of `B`'s *constraint-expression* is valid and results in `T::value` (with the mapping $\texttt{T} \mapsto \texttt{U*}$) $\vee$ `true` (with an empty mapping), despite the expression `T::value` being ill-formed for a pointer type `T`. Normalization of `C`'s *constraint-expression* results in the program being ill-formed, because it would form the invalid type `V&*` in the parameter mapping. — *end example*]

- For a *fold-operator* `Op`([expr.prim.fold]) that is either `&&` or `||`:
  - The normal form of an expression `( ... Op E )` is the normal form of `( E Op ... )`.
  - The normal form of an expression `( E1 Op ... Op E2 )` is the normal form of
    - `( E1 Op ... ) Op E2` if `E1` contains an unexpanded pack, or
    - `E1 Op ( E2 Op ... )` otherwise.
  - The normal form of an expression `F` of the form `( E Op ... )` is as follows: If `E` contains an unexpanded concept template parameter pack, it shall not

contain an unexpanded template parameter pack of another kind. Let `E′` be the normal form of `E`.

— If `E` contains an unexpanded concept template parameter pack $\texttt{P}_k$ that has corresponding template arguments in the parameter mapping of any atomic constraint (including concept-dependent constraints) of `E′`, the number of arguments specified for all such $\texttt{P}_k$ shall be the same number $N$. The normal form of `F` is the normal form of $\texttt{E}_0$ `Op` $\cdots$ `Op` $\texttt{E}_{N-1}$ after substituting in $\texttt{E}_i$ the respective $i^{\text{th}}$ concept argument of each $\texttt{P}_k$. If any such substitution results in an invalid type or expression, the program is ill-formed; no diagnostic is required([temp.constr.normal.invalid]).

— Otherwise, the normal form of `F` is a fold expanded constraint([temp.constr.fold]) whose constraint is `E′` and whose *fold-operator* is `Op`.

— The normal form of any other expression `E` is the atomic constraint whose expression is `E` and whose parameter mapping is the identity mapping.

## 13.7 Template declarations [temp.decls]

### 13.7.6 Partial specialization [temp.spec.partial]

#### 13.7.6.1 General [temp.spec.partial.general]

Modify [temp.spec.partial.general], paragraph 1:

1 A partial specialization of a template provides an alternative definition of the template that is used instead of the primary definition when the arguments in a specialization match those given in the partial specialization([temp.spec.partial.match]). A declaration of the primary template shall precede any partial specialization of that template. A partial specialization shall be reachable from any use of a template specialization that would make use of the partial specialization as the result of an implicit or explicit instantiation; no diagnostic is required([temp.spec.partial.general.partial.reachable]).

### 13.7.7 Function templates [temp.fct]

#### 13.7.7.2 Function template overloading [temp.over.link]

Modify [temp.over.link], paragraph 7:

7 If the validity or meaning of the program depends on whether two constructs are equivalent, and they are functionally equivalent but not equivalent, the program is ill-formed, no diagnostic required([temp.over.link.equiv.not.equiv]). Furthermore, if two declarations $A$ and $B$ of function templates

— introduce the same name,

— have corresponding signatures([basic.scope.scope]),

— would declare the same entity, when considering $A$ and $B$ to correspond in that determination([basic.link]), and

— accept and are satisfied by the same set of template argument lists,

but do not correspond, the program is ill-formed, no diagnostic required.

## 13.8 Name resolution [temp.res]

### 13.8.1 General [temp.res.general]

Modify [temp.res.general], paragraph 2:

2 If the validity or meaning of the program would be changed by considering a default argument or default template argument introduced in a declaration that is reachable from the point of instantiation of a specialization([temp.point]) but is not found by lookup for the specialization, the program is ill-formed, no diagnostic required([temp.res.general.default.but.not.found]).

*typename-specifier*:
`typename` *nested-name-specifier identifier*
`typename` *nested-name-specifier* $_{opt}$`template` *simple-template-id*

### 13.8.4 Dependent name resolution [temp.dep.res]

#### 13.8.4.1 Point of instantiation [temp.point]

Modify [temp.point], paragraph 7:

7 A specialization for a function template, a member function template, or of a member function or static data member of a class template may have multiple points of instantiations within a translation unit, and in addition to the points of instantiation described above,

— for any such specialization that has a point of instantiation within the *declaration-seq* of the *translation-unit*, prior to the *private-module-fragment* (if any), the point after the *declaration-seq* of the *translation-unit* is also considered a point of instantiation, and

— for any such specialization that has a point of instantiation within the *private-module-fragment*, the end of the translation unit is also considered a point of instantiation.

A specialization for a class template has at most one point of instantiation within a translation unit. A specialization for any template may have points of instantiation in multiple translation units. If two different points of instantiation give a template specialization different meanings according to the one-definition rule([basic.def.odr]), the program is ill-formed, no diagnostic required([temp.point.diff.pt.diff.meaning]).

#### 13.8.4.2 Candidate functions [temp.dep.candidate]

Modify [temp.dep.candidate], paragraph 1:

1 If a dependent call([temp.dep]) would be ill-formed or would find a better match had the lookup for its dependent name considered all the function declarations with external linkage introduced in the associated namespaces in all translation units, not just

considering those declarations found in the template definition and template instantiation contexts([basic.lookup.argdep]), then the program is ill-formed, no diagnostic required([temp.dep.candidate.different.lookup.different]).

## 13.9 Template instantiation and specialization [temp.spec]

### 13.9.2 Implicit instantiation [temp.inst]

Modify [temp.inst], paragraph 16:

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

[*Example*:

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

— *end example*]

### 13.9.3 Explicit instantiation [temp.explicit]

Modify [temp.explicit], paragraph 12:

12 If an entity is the subject of both an explicit instantiation declaration and an explicit instantiation definition in the same translation unit, the definition shall follow the declaration. An entity that is the subject of an explicit instantiation declaration and that is also used in a way that would otherwise cause an implicit instantiation([temp.inst]) in the translation unit shall be the subject of an explicit instantiation definition somewhere in the program; otherwise the program is ill-formed, no diagnostic ~~required~~requiredi([temp.explicit.decl.implicit.inst]).

[*Note*: This rule does apply to inline functions even though an explicit instantiation declaration of such an entity has no other normative effect. This is needed to ensure that if the address of an inline function is taken in a translation unit in which the implementation chose to suppress the out-of-line body, another translation unit will supply the body. — *end note*]

An explicit instantiation declaration shall not name a specialization of a template with internal linkage.

### 13.9.4 Explicit specialization [temp.expl.spec]

Modify [temp.expl.spec], paragraph 7:

7 If a template, a member template or a member of a class template is explicitly specialized, a declaration of that specialization shall be reachable from every use of that specialization that would cause an implicit instantiation to take place, in every translation unit in which such a use occurs; no diagnostic is required([temp.expl.spec.unreachable.declaration]). If the program does not provide a definition for an explicit specialization and either the

specialization is used in a way that would cause an implicit instantiation to take place or the member is a virtual member function, the program is ill-formed, no diagnostic required([temp.expl.spec.missing.definition]). An implicit instantiation is never generated for an explicit specialization that is declared but not defined.

[*Example*:

```
class String { };
template<class T> class Array { /* ... */ };
template<class T> void sort(Array<T>& v) { /* ... */ }

void f(Array<String>& v) {
  sort(v);          // use primary template sort(Array<T>&), T is String
}

template<> void sort<String>(Array<String>& v);     // error: specialization after use of primary template
template<> void sort<>(Array<char*>& v);            // OK, sort<char*> not yet used
template<class T> struct A {
  enum E : T;
  enum class S : T;
};
template<> enum A<int>::E : int { eint };           // OK
template<> enum class A<int>::S : int { sint };     // OK
template<class T> enum A<T>::E : T { eT };
template<class T> enum class A<T>::S : T { sT };
template<> enum A<char>::E : char { echar };        // error: A<char>::E was instantiated
                                                    // when A<char> was instantiated
template<> enum class A<char>::S : char { schar }; // OK
```

— *end example*]

## 13.10 **Function template specializations** [temp.fct.spec]

### 13.10.3 **Template argument deduction** [temp.deduct]

#### 13.10.3.1 **General** [temp.deduct.general]

Modify [temp.deduct.general], paragraph 7:

7 The *deduction substitution loci* are

— the function type outside of the *noexcept-specifier*,

— the *explicit-specifier*,

— the template parameter declarations, and

— the template argument list of a partial specialization([temp.spec.partial.general]).

The substitution occurs in all types and expressions that are used in the deduction substitution loci. The expressions include not only constant expressions such as those that appear in array bounds or as constant template arguments but also general expressions (i.e., non-constant expressions) inside `sizeof`, `decltype`, and other contexts that allow non-constant expressions. The substitution proceeds in lexical order and stops when a condition that causes deduction to fail is encountered. If substitution into different declarations of the same function template would cause template instantiations to occur in a different order or not at all, the program is ill-formed; no diagnostic required ([temp.deduct.general.diff.order]).

[*Note*: The equivalent substitution in exception specifications([except.spec]) and function contract assertions([dcl.contract.func]) is done only when the *noexcept-specifier* or *function-contract-specifier*, respectively, is instantiated, at which point a program is ill-formed if the substitution results in an invalid type or expression. — *end note*]

[*Example*:

```
template <class T> struct A { using X = typename T::X; };
template <class T> typename T::X f(typename A<T>::X);
template <class T> void f(...) { }
template <class T> auto g(typename A<T>::X) -> typename T::X;
template <class T> void g(...) { }
template <class T> typename T::X h(typename A<T>::X);
template <class T> auto h(typename A<T>::X) -> typename T::X;   // redeclaration
template <class T> void h(...) { }

void x() {
  f<int>(0);        // OK, substituting return type causes deduction to fail
  g<int>(0);        // error, substituting parameter type instantiates A<int>
  h<int>(0);        // ill-formed, no diagnostic required
}
```

— *end example*]

## 14 **Exception handling** [except]

### 14.4 **Handling an exception** [except.handle]

Modify [except.handle], paragraph 11:

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

## 15 **Preprocessing directives** [cpp]

### 15.2 **Conditional inclusion** [cpp.cond]

Modify [cpp.cond], paragraph 10:

10 Prior to evaluation, macro invocations in the list of preprocessing tokens that will become the controlling constant expression are replaced (except for those macro names modified by the `defined` unary operator), just as in normal text. If replacement of macros in the preprocessing tokens following the sequence `__has_embed (` and before a matching `)` (possibly produced by macro expansion) encounters a preprocessing token that is one of the *identifier*s `limit`, `prefix`, `suffix`, or `if_empty` and that *identifier* is defined as a macro([cpp.replace.general]), the program is ill-formed. If the preprocessing token `defined` is generated as a result of this replacement process([cpp.cond.defined.after.macro]) or use of the `defined` unary operator does not match one of the two specified forms prior to macro replacement, the program is ill-formed, no diagnostic required([cpp.cond.defined.malformed]).

### 15.3 **Source file inclusion** [cpp.include]

Modify [cpp.include], paragraph 7:

7 A preprocessing directive of the form

`# include` *pp-tokens new-line*

(that does not match the previous form) is permitted. The preprocessing tokens after `include` in the directive are processed just as in normal text (i.e., each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens). The resulting sequence of preprocessing tokens shall be of the form

*header-name-tokens*

An attempt is then made to form a *header-name* preprocessing token([lex.header]) from the whitespace and the characters of the spellings of the *header-name-tokens*; the treatment of whitespace is implementation-defined. If the attempt succeeds, the directive with the so-formed *header-name* is processed as specified for the previous form. Otherwise, the program is ill-formed, no diagnostic required([cpp.include.malformed.headername]).

[*Note*: Adjacent *string-literal*s are not concatenated into a single *string-literal* (see the translation phases in [lex.phases]); thus, an expansion that results in two *string-literal*s is an invalid directive. — *end note*]

### ($D + a$) **Enumeration of Core Undefined Behavior** **[ub]**

Add a new clause (D+a) ([ub]) after [depr.atomics.order]

**Annex (D+a)**
**(informative)**
**Enumeration of Core Undefined Behavior** **[ub]**

#### ($D + a$).1 **General** **[ub.general]**

**(D+a).1 General** **[ub.general]**

This Annex documents undefined behavior explicitly called out in the main standard text using the following phrases: the behavior of the program is undefined, has undefined behavior, results in undefined behavior, the behavior is undefined, have undefined behavior, is undefined, result has undefined behavior. Undefined behavior that is implicit is not covered by this annex. Each entry contains a title, a numeric cross reference to the main standard text, a summary of the issue and a code example demonstrating the issue. The code examples are there to clarify the undefined behavior and will not exhaustively cover all possible ways of invoking that case.

#### ($D + a$).2 **[basic]: Basics** **[ub.basic]**

**(D+a).2 [basic]: Basics** **[ub.basic]**

#### ($D + a$).2.1 **Implicitly creating object and undefined behavior** **[ub.intro.object]**

**(D+a).2.1 Implicitly creating object and undefined behavior** **[ub.intro.object]**

1 **Specified in:** [intro.object.implicit.create]
For each operation that is specified as implicitly creating objects, that operation implicitly creates and starts the lifetime of zero or more objects of implicit-lifetime types([basic.types]) in its specified region of storage if doing so would result in the program having defined behavior. If no such set of objects would give the program defined behavior, the behavior of the program is undefined.

2 [*Example*:

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

— *end example*]

3 **Specified in:** [intro.object.implicit.pointer]
After implicitly creating objects within a specified region of storage, some operations are described as producing a pointer to a suitable created object([basic.types]). These operations select one of the implicitly-created objects whose address is the address of the start of the region of storage, and produce a pointer value that points to that object, if that value would result in the program having defined behavior. If no such pointer value would give the program defined behavior, the behavior of the program is undefined.

4 [*Example*:

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

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

— *end example*]

[*Example*:

```
#include <cstdlib>

struct X {
  int a;
};

struct Y {
  int x;
};

*make_y() {
  X* p1 = (X*)std::malloc(sizeof(struct X));
  p1->a = 1;
  Y* p2 = (Y*)p1;
  p2->x = 2;    // undefined behavior, lifetime of p1 was started but not
                // ended and lifetime of p2 was not started.
  return p2;
}
```

— *end example*]

SY: These comments feel too loose. Shouldn't we be saying something along the lines of p1 points to an object of type X whose lifetime was started but not ended .... p2 points to an object of type Y but its lifetime was not started ..."

### ($D + a$).2.2 **Object alignment** **[ub.basic.align]**

### (D+a).2.2 Object alignment [ub.basic.align]

1 **Specified in:** [basic.align.object.alignment]
All instances of a type must be created in storage that meets the alignment requirement of that type.

2 [*Example*:

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

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

— *end example*]

### ($D + a$).2.3 **Object lifetime** **[ub.basic.life]**

### (D+a).2.3 Object lifetime [ub.basic.life]

1 **Specified in:** [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*:

```
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*]

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

4 [*Example*:

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

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

— *end example*]

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

6 [*Example*:

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

— *end example*]

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

8 [*Example*:

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

— *end example*]

9 **Specified in:** [lifetime.outside.glvalue.access]
Behavior is undefined if a glvalue referring to an object outside of its lifetime is used to access the object.

10 [*Example*:

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

— *end example*]

11 **Specified in:** [lifetime.outside.glvalue.member]
Behavior is undefined if a glvalue referring to an object outside of its lifetime is used to call a non-static member function of the object.

12 [*Example*:

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

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

— *end example*]

13 **Specified in:** [lifetime.outside.glvalue.virtual]
Behavior is undefined if a glvalue referring to an object outside of its lifetime is bound to a reference to a virtual base class.

14 [*Example*:

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

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

— *end example*]

15 **Specified in:** [lifetime.outside.glvalue.dynamic.cast]
Behavior is undefined if a glvalue referring to an object outside of its lifetime is used as the operand of a `dynamic_cast` or as the operand of `typeid`.

16 [*Example*:

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

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

— *end example*]

17 **Specified in:** [original.type.implicit.destructor]
The behavior is undefined if a non-trivial implicit destructor call occurs when the type of the object inhabiting the associated storage is not the original type associated with that storage.

18 [*Example*:

```
class T {};

struct B {
```

```
  ~B();
};

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

— *end example*]

19 **Specified in:** [creating.within.const.complete.obj]
Creating a new object within the storage that a const complete object with static, thread, or automatic storage duration occupies, or within the storage that such a const object used to occupy before its lifetime ended, results in undefined behavior

20 [*Example*:

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

const B b;

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

— *end example*]

### ($D + a$).2.4 **Indeterminate and erroneous values** **[ub.basic.indet]**

### (D+a).2.4 Indeterminate and erroneous values [ub.basic.indet]

1 **Specified in:** [basic.indet.value]
When the result of an evaluation is an indeterminate value (but not just an erroneous value) the behavior is undefined.

2 [*Example*:

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

— *end example*]

### ($D + a$).2.5 **Dynamic storage Duration** **[ub.basic.stc.dynamic]**

### (D+a).2.5 Dynamic storage Duration [ub.basic.stc.dynamic]

1 **Specified in:** [basic.stc.alloc.dealloc.constraint]
If the behavior of an allocation or deallocation function does not satisfy the semantic constraints specified in [basic.stc.dynamic.allocation] and [basic.stc.dynamic.deallocation]. the behavior is undefined.

2 [*Example*:

```
#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*]

3 **Specified in:** [basic.stc.alloc.dealloc.throw]
If a call to a deallocation function terminates by throwing an exception the behavior is undefined.

4 [*Example*:

```
struct X {
  void operator delete(void*) { throw "oops"; }
};
void f()
{
  X* x = new X();
  delete x;     // undefined behavior
}
```

— *end example*]

### ($D + a$).2.6 **Zero-sized allocation dereference** [**ub.basic.stc.alloc.zero.dereference**]

### (D+a).2.6 Zero-sized allocation dereference[ub.basic.stc.alloc.zero.dereference]

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

2 [*Example*:

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

— *end example*]

### ($D + a$).2.7 **Compound types** [**ub.basic.compound**]

### (D+a).2.7 Compound types [ub.basic.compound]

1 **Specified in:** [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 implemention-defined behavior.)

2 [*Example*:

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

```
}
```

— *end example*]

### $(D + a)$.2.8 **Sequential execution** **[ub.intro.execution]**

### (D+a).2.8 Sequential execution [ub.intro.execution]

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

2 [*Example*:

```
void g(int i) {
  i = 7, i++, i++;  // i becomes 9

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

— *end example*]

### $(D + a)$.2.9 **Data races** **[ub.intro.races]**

### (D+a).2.9 Data races [ub.intro.races]

1 **Specified in:** [intro.races.data]
The execution of a program contains a data race if it contains two potentially concurrent conflicting actions, at least one of which is not atomic, and neither happens before the other, except for the special case for signal handlers described in [intro.races]. Any such data race results in undefined behavior.

2 [*Example*:

```
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*]

### $(D + a)$.2.10 **Forward progress** **[ub.intro.progress]**

### (D+a).2.10 Forward progress [ub.intro.progress]

1 **Specified in:** [intro.progress.stops]
The behavior is undefined if a thread of execution that has not terminated stops making execution steps.

2 [*Example*:

```
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*]

### ($D + a$).2.11 **main function** **[ub.basic.start.main]**

### (D+a).2.11 main function [ub.basic.start.main]

1 **Specified in:** [basic.start.main.exit.during.destruction]
If `std::exit` is called to end a program during the destruction of an object with static or thread storage duration, the program has undefined behavior.

2 [*Example*:

```
#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*]

### ($D + a$).2.12 **Termination** **[ub.basic.start.term]**

### (D+a).2.12 Termination [ub.basic.start.term]

1 **Specified in:** [basic.start.term.use.after.destruction]
If a function contains a block-scope object of static or thread storage duration that has been destroyed and the function is called during the destruction of an object with static or thread storage duration, the program has undefined behavior if the flow of control passes through the definition of the previously destroyed block-scope object. Likewise, the behavior is undefined if the block-scope object is used indirectly (i.e., through a pointer) after its destruction.

2 [*Example*:

```
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
```

— *end example*]

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

4 [*Example*:

— *end example*]

JMB/TD: This is really a general precondition imposed on the Standard Library, not a piece of core language undefined behavior. It is also currently missing an example. Should we retain this UB? Should we have a core issue to move this wording into the library section somewhere?

SY: If we keep this, we should have an example.

### ($D+a$).3 **[expr]: Expressions** **[ub.expr]**

### **(D+a).3 [expr]: Expressions** **[ub.expr]**

#### ($D+a$).3.1 **Result of Expression not Mathematically Defined/out of Range** **[ub.expr.eval]**

#### **(D+a).3.1 Result of Expression not Mathematically Defined/out of Range [ub.expr.eval]**

1 **Specified in:** [expr.expr.eval]
If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined.

2 [*Example*:

```
#include <limits>
int main() {
  // Assuming 32-bit int the range of values are: −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
}
```

— *end example*]

### ($D+a$).3.2 **Value category** **[ub.basic.lval]**

### **(D+a).3.2 Value category** **[ub.basic.lval]**

1 **Specified in:** [expr.basic.lvalue.strict.aliasing.violation]
If a program attempts to access([defns.access]) the stored value of an object whose dynamic type is $T$ through a glvalue whose type is not similar([conv.rval]) to $T$ (or its corresponding signed or unsigned types) the behavior is undefined.

2 [*Example*:

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

  return *i;
}

int main() {
  int x = 0;

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

— *end example*]

3 **Specified in:** [expr.basic.lvalue.union.initialization]
If a program invokes a defaulted copy/move constructor or defaulted copy/move assignment operator of a union with an argument that is not an object of a similar type within its lifetime, the behavior is undefined.

4 [*Example*:

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

— *end example*]

### $(D + a)$.3.3 **Type** **[ub.expr.type]**

### **(D+a).3.3 Type** **[ub.expr.type]**

1 **Specified in:** [expr.type.reference.lifetime]
Evaluating a reference when an equivalent use of a pointer denoting the same object would be invalid has undefined behavior.

2 [*Example*:

```
void g()
{
  int* ip = new int(5);
  int& i = *ip;
  delete ip;
  i;    // undefined behavior
}
```

— *end example*]

### $(D + a)$.3.4 **Lvalue-to-rvalue conversion** **[ub.conv.lval]**

### **(D+a).3.4 Lvalue-to-rvalue conversion** **[ub.conv.lval]**

1 **Specified in:** [conv.lval.valid.representation]
Performing an lvalue-to-rvalue conversion on an object whose value representation is not valid for its type has undefined behavior.

2 [*Example*:

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

— *end example*]

### ($D + a$).3.5 **Floating-point conversions** **[ub.conv.double]**

### (D+a).3.5 Floating-point conversions [ub.conv.double]

1 **Specified in:** [conv.double.out.of.range]
Converting a floating point value to a type that cannot represent the value is undefined behavior.

2 [*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 system where
                // representable values are [-inf,+inf] this would not be UB
  int i = d2;   // undefined behavior, the max value of double is not representable as int
}
```

— *end example*]

### ($D + a$).3.6 **Floating-integral conversions** **[ub.conv.fpint]**

### (D+a).3.6 Floating-integral conversions [ub.conv.fpint]

1 **Specified in:** [conv.fpint.int.not.represented]
When converting a floating-point value to an integer type and vice versa if the value is not representable in the destination type it is undefined behavior.

2 [*Example*:

```
#include <limits>

int main() {
  // Assuming 32-bit int the range of values are: −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
}
```

— *end example*]

3 **Specified in:** [conv.fpint.float.not.represented]
When converting a value of integer or unscoped enumeration type to a floating-point type, if the value is not representable in the destination type it is undefined behavior.

4 [*Example*:

```
int main() {
  __uint128_t x2 = -1;
  float f = x2; // undefined behavior on systems where the range of
  // representable values of float is [-max,+max] on system where
  // represetable values are [-inf,+inf] this would not be UB
}
```

— *end example*]

### $(D + a)$.3.7 **Pointer conversions** **[ub.conv.ptr]**

### (D+a).3.7 Pointer conversions [ub.conv.ptr]

1 **Specified in:** [conv.ptr.virtual.base]
Converting a pointer to a derived class `D` to a pointer to a virtual base class `B` that does not point to a valid object within its lifetime has undefined behavior.

2 [*Example*:

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

— *end example*]

### $(D + a)$.3.8 **Pointer-to-member conversions** **[ub.conv.mem]**

### (D+a).3.8 Pointer-to-member conversions [ub.conv.mem]

1 **Specified in:** [conv.member.missing.member]
The conversion of a pointer to a member of a base class to a pointer to member of a derived class that could not contain that member has undefined behavior.

2 [*Example*:

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

— *end example*]

### $(D + a)$.3.9 **Function call** **[ub.expr.call]**

### (D+a).3.9 Function call [ub.expr.call]

1 **Specified in:** [expr.call.different.type]
Calling a function through an expression whose function type is not call-compatible with the function type of the called function's definition results in undefined behavior.

2 [*Example*:

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

int f1(float) { return 10; }

int f2(int) { return 20; }

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

— *end example*]

### $(D + a)$.3.10 **Class member access** **[ub.expr.ref]**

### (D+a).3.10 Class member access [ub.expr.ref]

1 **Specified in:** [expr.ref.member.not.similar]
If `E2` is a non-static member and the result of `E1` is an object whose type is not similar([conv.qual]) to the type of `E1`, the behavior is undefined.

2 [*Example*:

```
struct A { int i; };
struct B { int j; };
struct D : A, B {};
void f() {
  D d;
  reinterpret_cast<B&>(d).j;    // undefined behavior
}
```

— *end example*]

SY: The wording is not great, I don't have a good suggestion but we should get some opinions

### $(D + a)$.3.11 **Dynamic cast** **[ub.expr.dynamic.cast]**

### (D+a).3.11 Dynamic cast [ub.expr.dynamic.cast]

1 **Specified in:** [expr.dynamic.cast.pointer.lifetime]
Evaluating a `dynamic_cast` on a non-null pointer that points to an object (of polymorphic type) of the wrong type or to an object not within its lifetime has undefined behavior.

2 [*Example*:

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

— *end example*]

3 **Specified in:** [expr.dynamic.cast.glvalue.lifetime]
Evaluating a `dynamic_cast` on a reference that denotes an object (of polymorphic type) of the wrong type or an object not within its lifetime has undefined behavior.

4 [*Example*:

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

— *end example*]

## $(D + a)$.3.12 **Static cast** [**ub.expr.static.cast**]

## (D+a).3.12 Static cast [ub.expr.static.cast]

1 **Specified in:** [expr.static.cast.base.class]
We can cast a base class B to a reference to derived class D (with certain restrictions wrt to cv qualifiers) as long B is a base class subobject of type D, otherwise the behavior is undefined.

2 [*Example*:

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

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

— *end example*]

3 **Specified in:** [expr.static.cast.enum.outside.range]
If the enumeration type does not have a fixed underlying type, the value is unchanged if the original value is within the range of the enumeration values([dcl.enum]), and otherwise, the behavior is undefined.

4 [*Example*:

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

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

— *end example*]

5 **Specified in:** [expr.static.cast.fp.outside.range]
An explicit conversion of a floating-point value that is outside the range of the target type has undefined behavior.

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

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

— *end example*]

7 **Specified in:** [expr.static.cast.downcast.wrong.derived.type]
Casting from a pointer to a base class to a pointer to a derived class when there is no enclosing object of that derived class at the specified location has undefined behavior.

8 [*Example*:

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

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

— *end example*]

9 **Specified in:** [expr.static.cast.does.not.contain.orignal.member]
A pointer to member of derived class D can be cast to a pointer to member of base class B (with certain restrictions on cv qualifiers) as long as B contains the original member, or is a base or derived class of the class containing the original member; otherwise the behavior is undefined.

10 [*Example*:

```
struct A {
  char c;
};

struct B {
  int i;
};

struct D : A, B {};

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

— *end example*]

### $(D + a)$.3.13 **Unary operators** **[ub.expr.unary.op]**

### **(D+a).3.13 Unary operators** **[ub.expr.unary.op]**

1 **Specified in:** [expr.unary.dereference]
Dereferencing a pointer that does not point to an object or function has undefined behavior.

2 [*Example*:

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

— *end example*]

### $(D + a)$.3.14 **New** **[ub.expr.new]**

### **(D+a).3.14 New** **[ub.expr.new]**

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

2 [*Example*:

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

— *end example*]

[*Example*:

```
#include <new>

struct A {
  int x;
};

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

— *end example*]

### $(D + a)$.3.15 **Delete** **[ub.expr.delete]**

### (D+a).3.15 Delete [ub.expr.delete]

1 **Specified in:** [expr.delete.mismatch]
Using array delete on the result of a single object new expression is undefined behavior.

2 [*Example*:

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

— *end example*]

3 **Specified in:** [expr.delete.array.mismatch]
Using single object delete on the result of an array new expression is undefined behavior.

4 [*Example*:

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

— *end example*]

5 **Specified in:** [expr.delete.dynamic.type.differ]
If the static type of the object to be deleted is different from its dynamic type and the selected deallocation function is not a destroying operator delete, the static type shall be a base class of the dynamic type of the object to be deleted and the static type shall have a virtual destructor or the behavior is undefined.

6 [*Example*:

```
struct B {
  int a;
};
```

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

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

— *end example*]

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

8 [*Example*:

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

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

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

— *end example*]

## $(D + a)$.3.16 **Pointer-to-member operators** [ub.expr.mptr.oper]

## (D+a).3.16 Pointer-to-member operators [ub.expr.mptr.oper]

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

2 [*Example*:

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

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

— *end example*]

3 **Specified in:** [expr.mptr.oper.member.func.null]
If the second operand in a `.*` expression is the null member pointer value([conv.mem]), the behavior is undefined.

4 [*Example*:

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

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

— *end example*]

### ($D+a$).3.17 **Multiplicative operators** **[ub.expr.mul]**

### (D+a).3.17 Multiplicative operators [ub.expr.mul]

1 **Specified in:** [expr.mul.div.by.zero]
Division by zero is undefined behavior.

2 [*Example*:

```
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 float is [-max,+max] on system where
                        // represetable values are [-inf,+inf] this would not be UB
}
```

— *end example*]

3 **Specified in:** [expr.mul.representable.type.result]
If the quotient a/b is representable in the type of the result, (a/b)*b + a%b is equal to a; otherwise, the behavior of both a/b and a%b is undefined.

4 [*Example*:

```
#include <limits>

int main() {
  int x = std::numeric_limits<int>::min() / -1;
        // Assuming LP64 −2,147,483,648 which when divided by −1
        // gives us 2,147,483,648 which is not representable by int
}
```

— *end example*]

### ($D+a$).3.18 **Additive operators** **[ub.expr.add]**

### (D+a).3.18 Additive operators [ub.expr.add]

1 **Specified in:** [expr.add.out.of.bounds]
Creating an out of bounds pointer is undefined behavior.

2 [*Example*:

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

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

— *end example*]

[*Example*:

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

int main() {
  const int(*y)[10] = arrs + 11;        // undefined behavior, creating an out of bounds pointer.
                                        // We can't treat arrs as-if it was a pointer to 100 int,
}
```

— *end example*]

3 **Specified in:** [expr.add.sub.diff.pointers]
Subtracting pointers that are not part of the same array is undefined behavior.

4 [*Example*:

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

— *end example*]

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

6 [*Example*:

```
struct S {
  int i;
};

struct T : S {
  double d;
};

void f(const S* s, std::size_t count) {
  for (const S* end = s + count; s != end; ++s) {
    ...
  }
}

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

— *end example*]

### ($D + a$).3.19 **Shift operators** **[ub.expr.shift]**

### (D+a).3.19 Shift operators [ub.expr.shift]

1 **Specified in:** [expr.shift.neg.and.width]
Shifting by a negative amount or equal or greater than the bit-width of a type is undefined behavior.

2 [*Example*:

```
int y = 1 << -1;        // undefined behavior, shift is negative

static_assert(sizeof(int) == 4 && CHAR_BIT == 8);
int y1 = 1 << 32;       // undefined behavior, shift is equal to the bit width of int
int y2 = 1 >> 32;       // undefined behavior, shift is equal to the bit width of int
```

— *end example*]

### ($D + a$).3.20 **Assignment and compound assignment operators** [ub.expr.assign]

### (D+a).3.20 Assignment and compound assignment operators[ub.expr.assign]

1 **Specified in:** [expr.assign.overlap]
Overlap in the storage between the source and destination may result in undefined behavior.

2 [*Example*:

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

— *end example*]

### ($D + a$).4 **[stmt]: Statements** [ub.stmt.stmt]

### (D+a).4 [stmt]: Statements [ub.stmt.stmt]

### ($D + a$).4.1 **The return statement** [ub.stmt.return]

### (D+a).4.1 The return statement [ub.stmt.return]

1 **Specified in:** [stmt.return.flow.off]
Flowing off the end of a function other than main or a coroutine results in undefined behavior if the return type is not *cv* `void`.

2 [*Example*:

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

void b() {
  int x = f(0); // undefined behavior, using 0 as an argument will cause f(...) to flow off the end
                // with a return statement
}
```

— *end example*]

### ($D + a$).4.2 **The co_return statement** [ub.return.coroutine]

### (D+a).4.2 The co_return statement [ub.return.coroutine]

1 **Specified in:** [stmt.return.coroutine.flow.off]
Flowing off the end of a coroutine function body that does not return void has undefined behavior.

2 [*Example*:

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

class resumable {
 public:
  struct promise_type;
  using coro_handle = std::coroutine_handle<promise_type>;
  resumable(coro_handle handle) : handle_(handle) { assert(handle); }
  resumable(resumable&) = delete;
  resumable(resumable&&) = delete;
  bool resume() {
    if (not handle_.done())
      handle_.resume();
    return not handle_.done();
  }
  ~resumable() { handle_.destroy(); }
  const char* return_val();

 private:
  coro_handle handle_;
};

struct resumable::promise_type {
  using coro_handle = std::experimental::coroutine_handle<promise_type>;
  const char* string_;
  auto get_return_object() { return coro_handle::from_promise(*this); }
  auto initial_suspend() { return std::experimental::suspend_always(); }
  auto final_suspend() noexcept { return std::experimental::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::experimental::suspend_always();
  // undefined behavior, falling off the end of coroutine that does not return void
}

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

— *end example*]

### $(D + a).4.3$ **Declaration statement** **[ub.stmt.dcl]**

### (D+a).4.3 Declaration statement [ub.stmt.dcl]

1 **Specified in:** [stmt.dcl.local.static.init.recursive]
If control re-enters the declaration recursively while the variable is being initialized, the behavior is undefined.

2 [*Example*:

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

— *end example*]

### $(D + a)$.5 **[dcl]: Declarations** **[ub.dcl.dcl]**

### (D+a).5 [dcl]: Declarations [ub.dcl.dcl]

### $(D + a)$.5.1 **The cv-qualifiers** **[ub.dcl.type.cv]**

### (D+a).5.1 The cv-qualifiers [ub.dcl.type.cv]

1 **Specified in:** [dcl.type.cv.modify.const.obj]
Any attempt to modify a const object during its lifetime results in undefined behavior.

2 [*Example*:

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

— *end example*]

3 **Specified in:** [dcl.type.cv.access.volatile]
If an attempt is made to access an object defined with a volatile-qualified type through the use of a non-volatile glvalue, the behavior is undefined

4 [*Example*:

```
volatile int x = 0;
int& y = const_cast<int&>(x);
std::cout << y;         // undefined behavior, accessing volatile through non-volatile glvalue
```

— *end example*]

### $(D + a)$.5.2 **References** **[ub.dcl.ref]**

### (D+a).5.2 References [ub.dcl.ref]

1 **Specified in:** [dcl.ref.incompatible.function]
Initializing a reference to a function with a value that is a function that is not call-compatible([expr.call]) with the type of the reference has undefined behavior.

2 [*Example*:

```
void f(float x);
void (&g)(int) = reinterpret_cast<void (&)(int)>(f);    // undefined behavior
```

— *end example*]

3 **Specified in:** [dcl.ref.incompatible.type]
Initializing a reference to an object with a value that is not type-accessible([basic.lval]) through the type of the reference has undefined behavior.

4 [*Example*:

```
float g;
int& i = reinterpret_cast<int&>(g);     // undefined behavior
```

— *end example*]

5 **Specified in:** [dcl.ref.uninitialized.reference]
Evaluating a reference prior to initializing that reference has undefined behavior.

6 [*Example*:

```
extern int &ir1;
int i2 = ir1;   // undefined behavior, ir1 not yet initialized
int i3 = 17;
int &ir1 = i3;
```

— *end example*]

### $(D + a)$.5.3 Coroutine definitions [ub.dcl.fct.def.coroutine]

### (D+a).5.3 Coroutine definitions [ub.dcl.fct.def.coroutine]

1 **Specified in:** [dcl.fct.def.coroutine.resume.not.suspended]
Invoking a resumption member function for a coroutine that is not suspended results in undefined behavior.

2 [*Example*:

```
#include <experimental/coroutine>

using namespace std::experimental;

struct minig {
  struct promise_type {
    int val;
    minig get_return_object() { return {*this}; }
    constexpr suspend_always initial_suspend() noexcept { return {}; }
    constexpr suspend_always final_suspend() noexcept { return {}; }
    constexpr void return_void() noexcept {}
    [[noreturn]] void unhandled_exception() noexcept { throw; }
    suspend_always yield_value(int v) noexcept {
      val = v;
      return {};
    }
  };
  using HDL = coroutine_handle<promise_type>;
  HDL coro;
  minig(promise_type& p) : coro(HDL::from_promise(p)) {}
  ~minig() { coro.destroy(); }
  bool move_next() {
    coro.resume();
    return !coro.done();
  }
  int current_value() { return coro.promise().val; }
};

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

int main() {
  auto g = f(10);
  int sum = 0;
  while (g.move_next())
    sum += g.current_value();
#if 0
  // undefined behavior to call move_next(), because coro.resume() is
  // UB after final_suspend returns, even when it returns
  // suspend_always
  g.move_next();
#endif
  return sum;
}
```

— *end example*]

3 **Specified in:** [dcl.fct.def.coroutine.destroy.not.suspended]
Invoking destroy() on a coroutine that is not suspended is undefined behavior.

4 [*Example*:

```
#include <experimental/coroutine>

using namespace std::experimental;

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

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

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

  return sum;
}
```

— *end example*]

### $(D + a)$.5.4 **Assumption attribute** [ub.dcl.attr.assume]

### (D+a).5.4 Assumption attribute [ub.dcl.attr.assume]

1 **Specified in:** [dcl.attr.assume.false]
If an assumption expression would not evaluate to true at the point where it appears the behavior is undefined.

2 [*Example*:

```
int g(int x) {
  [[assume(x >= 0)]];
```

```
  return x/32;
}

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

— *end example*]

### $(D + a)$.5.5 **Noreturn attribute** **[ub.dcl.attr.noreturn]**

### (D+a).5.5 Noreturn attribute [ub.dcl.attr.noreturn]

1 **Specified in:** [dcl.attr.noreturn.eventually.returns]
If a function f is called where f was previously declared with the `noreturn` attribute and f eventually returns, the behavior is undefined.

2 [*Example*:

```
[[noreturn]] void f(int i) {
  if (i > 0)
    throw "positive";
}

int main() {
  f(0);         // undefined behavior, returns from a [[noreturn]] function
}
```

— *end example*]

### $(D + a)$.6 **[class]: Classes** **[ub.class]**

### (D+a).6 [class]: Classes [ub.class]

### $(D + a)$.6.1 **Destructors** **[ub.class.dtor]**

### (D+a).6.1 Destructors [ub.class.dtor]

1 **Specified in:** [class.dtor.no.longer.exists]
Once a destructor is invoked for an object, the object's lifetime has ended; the behavior is undefined if the destructor is invoked for an object whose lifetime has ended.

2 [*Example*:

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

int main() {
  A a;
  a.~A();
}   // undefined behavior, lifetime of a already ended before implicit destructor
```

— *end example*]

### $(D + a)$.6.2 **Abstract classes** **[ub.class.abstract]**

### (D+a).6.2 Abstract classes [ub.class.abstract]

1 **Specified in:** [class.abstract.pure.virtual]
Calling a pure virtual function from a constructor or destructor in an abstract class is undefined behavior.

2 [*Example*:

```
struct B {
  virtual void f() = 0;
  B() {
    f();        // undefined behavior, f is pure virtual and we are calling from the constructor
  }
};

struct D : B {
  void f() override;
};
```

— *end example*]

### ($D + a$).6.3 **Initializing bases and members** **[ub.class.base.init]**

### (D+a).6.3 Initializing bases and members [ub.class.base.init]

1 **Specified in:** [class.base.init.mem.fun]
It is undefined behavior to call a member function before all the *mem-initializer*s for base classes have completed.

2 [*Example*:

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

class B : public A {
  int j;

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

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

class D : public B, C {
  int i;

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

— *end example*]

### ($D + a$).6.4 **Construction and destruction** **[ub.class.cdtor]**

### (D+a).6.4 Construction and destruction [ub.class.cdtor]

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

2 [*Example*:

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

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

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

extern X xobj;
int *p3 = &xobj.i;      // OK, X is a trivial class
X xobj;

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

— *end example*]

CM: Can this example be shortened?

3 **Specified in:** [class.cdtor.after.dtor]
For an object with a non-trivial destructor, referring to any non-static member or base class of the object after the destructor finishes execution has undefined behavior.

4 [*Example*:

```
struct X {
  int i;
  ~X();     // non-trivial
};
X& g()
{
  static X x;
  return x;
}
void f()
{
  X* px = &g();
  px->~X();
  int j = px->i;    // undefined behavior
```

```
}
```

— *end example*]

5 **Specified in:** [class.cdtor.convert.pointer]
When converting a pointer to a base class of an object, construction must have started and destruction must not have finished otherwise this is undefined.

6 [*Example*:

```
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*]

7 **Specified in:** [class.cdtor.form.pointer]
When forming a pointer to a direct non-static member of a class, construction must have started and destruction must not have finished otherwise the behavior is undefined.

8 [*Example*:

```
struct A {
  int i = 0;
};
struct B {
  int *p;
  A a;
  B() : p(&a.i) {}  // undefined behavior
};
```

— *end example*]

9 **Specified in:** [class.cdtor.virtual.not.x]
When a virtual function is called directly or indirectly from a constructor or from a destructor, including during the construction or destruction of the class's non-static data members, and the object to which the call applies is the object (call it `x`) under construction or destruction, the function called is the final overrider in the constructor's or destructor's class and not one overriding it in a more-derived class. If the virtual function call uses an explicit class member access([expr.ref]) and the object expression refers to the complete object of `x` or one of that object's base class subobjects but not `x` or one of its base class subobjects, the behavior is undefined.

10 [*Example*:

```
struct V {
  virtual void f();
  virtual void g();
};

struct A : virtual V {
```

```
  virtual void f();
};

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

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

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

— *end example*]

11 **Specified in:** [class.cdtor.typeid]
If the operand of `typeid` refers to the object under construction or destruction and the static type of the operand is neither the constructor or destructor's class nor one of its bases, the behavior is undefined.

12 [*Example*:

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

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

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

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

— *end example*]

13 **Specified in:** [class.cdtor.dynamic.cast]
If the operand of the `dynamic_cast` refers to the object under construction or destruction and the static type of the operand is not a pointer to or object of the constructor or destructor's own class or one of its bases, the `dynamic_cast` results in undefined behavior.

14 [*Example*:

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

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

struct D : A, B {
```

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

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

— *end example*]

### ($D + a$).7 [temp]: Templates [ub.temp]

### (D+a).7 [temp]: Templates [ub.temp]

#### ($D + a$).7.1 Implicit instantiation [ub.temp.inst]

#### (D+a).7.1 Implicit instantiation [ub.temp.inst]

1 **Specified in:** [temp.inst.inf.recursion]
The result of an infinite recursion in template instantiation is undefined.

2 [*Example*:

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

int main() {
  X<int> x;     // undefined behavior, instantiation will kick off infinite recursion
}
```

— *end example*]

### ($D + a$).8 [except]: Exception handling [ub.except]

### (D+a).8 [except]: Exception handling [ub.except]

#### ($D + a$).8.1 Handling an exception [ub.except.handle]

#### (D+a).8.1 Handling an exception [ub.except.handle]

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

2 [*Example*:

```
#include <iostream>

struct A {
  A() try : x(0 ? 1 : throw 1) {
  } catch (int) {
    std::cout << "y: " << y << std::endl;   // undefined behavior, referring to non-static member y in
                                            // the handler of function-try-block
  }
  int x;
  int y = 42;
```

```
};
```

— *end example*]

## (*D* + *b*) Enumeration of Ill-formed, No Diagnostic Required [ifndr]

## Annex (D+b)
## (informative)
## Enumeration of Ill-formed, No Diagnostic Required [ifndr]

### (*D* + *b*).1 General [ifndr.general]

### (D+b).1 General [ifndr.general]

This Annex documents ill-formed no diagnostic required behavior called out in the main standard text by the following phrases: no diagnostic is required, no diagnostic required and no diagnostic shall be issued. Each entry contains a title, a numeric cross reference to the main standard text, a summary of the issue and a code example demonstrating the issue. The code examples are there to clarify the ill-formed no diagnostic required cases and will not exhaustively cover all possible ways of invoking that case.

### (*D* + *b*).2 [lex]: Lexical conventions [ifndr.lex]

### (D+b).2 [lex]: Lexical conventions [ifndr.lex]

### (*D* + *b*).2.1 Identifiers [ifndr.lex.name]

### (D+b).2.1 Identifiers [ifndr.lex.name]

1 **Specified in:** [lex.name.reserved]
Some identifiers are reserved for use by C++ implementations and shall not be used otherwise; no diagnostic is required.

2 [*Example*:

```
int _z; // no diagnostic required, _z is reserved because it starts with _ at global scope

int main() {
    int __x;    // no diagnostic required, __x is reserved because it starts with __
    int _Y;     // no diagnostic required, _Y is reserved because it starts with _ followed by a capital letter
    int x__y;   // no diagnostic required, x__y is reserved because it contains __
}
```

— *end example*]

### (*D* + *b*).3 [basic]: Basics [ifndr.basic]

### (D+b).3 [basic]: Basics [ifndr.basic]

### (*D* + *b*).3.1 Program and linkage [ifndr.basic.link]

### (D+b).3.1 Program and linkage [ifndr.basic.link]

1 **Specified in:** [basic.link.entity.same.module]
If an entity has multiple declarations attached to different modules where neither is reachable from the other the program is ill-formed, no diagnostic required.

2 [*Example*:

`"decls.h"`

```
int h();            // #1, attached to the global module
```

Module interface of `M`

```
module;
export moodule M;
export int h();     // #1, attached to M
```

Other translation unit

```
#include "decls.h"
import M;
    // ill-formed, no diagnostic required, the declarations of ‘h‘ cannot
    // reach one another
```

— *end example*]

3 **Specified in:** [basic.link.consistent.types]
Multiple declarations of an entity must be consistent, no diagnostic is required if neither declaration is reachable from the other.

4 [*Example*:

Module interface of `M`

```
void g();       // #1
void h();       // #2
template <typename T> int j;    // #3
```

Module interface of `N`

```
int g();        // same entity as #1, different type
namespace h {}  // same entity as #2, not both namespaces
template <int X> int j;         // same entity as #3, non-equivalent template heads
```

Other translation unit

```
import M;
import N;
    // ill-formed, no diagnostic required due to the mismatched pairs above
```

— *end example*]

## $(D + b)$.3.2 **One-definition rule** **[ifndr.basic.def.odr]**

## (D+b).3.2 One-definition rule [ifndr.basic.def.odr]

1 **Specified in:** [basic.def.odr.exact.one.def]
Every program shall contain exactly one definition of every non-inline function or variable that is odr-used in that program outside of a discarded statement([stmt.if]); no diagnostic required.

2 [*Example*:

```
auto f() {
  struct A {};
  return A{};
}
decltype(f()) g();
auto x = g();   // ill-formed, no diagnostic required, function g is used but not defined
                // in this translation unit, and cannot be defined in any other translation unit
                // because its type does not have linkage
```

— *end example*]

3 **Specified in:** [basic.def.odr.unnamed.enum.same.type]
If, at any point in the program, there is more than one reachable unnamed enumeration definition in the same scope that have the same first enumerator name and do not have typedef names for linkage purposes([dcl.enum]), those unnamed enumeration types shall be the same; no diagnostic required.

4 [*Example*:

```
// a.h
enum { a };

// b.h
enum { a, b };

// main.cpp
import "a.h";
import "b.h";
auto n = decltype(a)::b;        // ill-formed no diagnostic required, more than one unnammed enum
                                // definition reachable at this point but their types are not the same
```

— *end example*]

### ($D + b$).3.3 **Contract assertions** **[ifndr.basic.contract]**

### **(D+b).3.3 Contract assertions** **[ifndr.basic.contract]**

#### ($D + b$).3.3.1 **General** **[ifndr.basic.contract.general]**

#### **(D+b).3.3.1 General** **[ifndr.basic.contract.general]**

1 **Specified in:** [basic.contract.vastart.contract.predicate]
The use of `va_start`([cstdarg.syn]) within the predicate of a contract assertion is ill-formed, no diagnostic required;

2 [*Example*:

```
void f(...)
{
  va_list args;
  contract_assert((va_start(const_cast<decltype(args)>(args)), true))   // ill-formed, no diagnostic required
}
```

— *end example*]

#### ($D + b$).3.3.2 **Contract-violation handler** **[ifndr.basic.contract.handler]**

#### **(D+b).3.3.2 Contract-violation handler** **[ifndr.basic.contract.handler]**

1 **Specified in:** [basic.contract.handler.replacing.nonreplaceable]
On platforms where the contract-violation handler is not replaceable([term.replaceable.function]) a function definition which could be such a replacement function is ill-formed, no diagnostic required.

2 [*Example*:

```
#include <contracts>
void handle_contract_violation(const std::contract_violation& violation) {}
    // ill-formed, no diagnostic required if violation-handler is not replaceable
```

— *end example*]

### ($D+b$).3.4 **Member name lookup** **[ifndr.class.member.lookup]**

### (D+b).3.4 Member name lookup [ifndr.class.member.lookup]

1 **Specified in:** [class.member.lookup.name.refers.diff.decl]
A name N used in a class S shall refer to the same declaration in its context and when re-evaluated in the completed scope of S.

2 [*Example*:

```
struct foo {};

struct bar {
  foo *m_foo;

  foo *foo() {
    return m_foo;
  } // IFNDR, foo now refers to member function foo() while previously referred to struct foo
};
```

— *end example*]

[*Example*:

```
struct B {
  static int f();
};

struct D : public B {
  using B::f;
  int g(decltype(f()) x) {
    return 0;
  } // Ill-formed no diagnostic required,
    // decltype(f()) will refer to B::f() here but if
    // moved to the end of D would refer to D::f()
  static float f();
};

int main() {
  D d;

  return d.g(0);
}
```

— *end example*]

### ($D+b$).4 **[expr]: Expressions** **[ifndr.expr]**

### (D+b).4 [expr]: Expressions [ifndr.expr]

### $(D + b)$.4.1 **Requires expressions** **[ifndr.expr.prim.req]**

### (D+b).4.1 Requires expressions [ifndr.expr.prim.req]

1 **Specified in:** [expr.prim.req.always.sub.fail]
If the substitution of template arguments into a *requirement* would always result in a substitution failure, the program is ill-formed; no diagnostic required.

2 [*Example*:

```
template <typename T> concept C = requires {
  new int[-(int)sizeof(T)]; // ill-formed, no diagnostic required, the size of the allocation is
                            // required to be greater than zero but can never be
};
```

— *end example*]

### $(D + b)$.5 **[stmt]: Statements** **[ifndr.stmt.stmt]**

### (D+b).5 [stmt]: Statements [ifndr.stmt.stmt]

### $(D + b)$.5.1 **Ambiguity resolution** **[ifndr.stmt.ambig]**

### (D+b).5.1 Ambiguity resolution [ifndr.stmt.ambig]

1 **Specified in:** [stmt.ambig.bound.diff.parse]
If, during parsing, a name in a template parameter is bound differently than it would be bound during a trial parse, the program is ill-formed. No diagnostic is required.

2 [*Example*:

```
template <int N> struct A { const static int a = 20; };

template <> struct A<100> { using a = char; };

const int x = 10;

int main() {
  using T = const int;
  T(x)
  (100), (y)(A<x>::a);  // ill-formed no diagnostic required, during trial parse the template
                        // parameter x is bound to the global x later during parsing the template
                        // parameter x is bound to the local x declared on the same line
}
```

— *end example*]

### $(D + b)$.6 **[dcl]: Declarations** **[ifndr.dcl.dcl]**

### (D+b).6 [dcl]: Declarations [ifndr.dcl.dcl]

### $(D + b)$.6.1 **Specifiers** **[ifndr.dcl.spec]**

### (D+b).6.1 Specifiers [ifndr.dcl.spec]

### $(D + b)$.6.1.1 **The `constinit` specifier** **[ifndr.dcl.constinit]**

### (D+b).6.1.1 The `constinit` specifier [ifndr.dcl.constinit]

1 **Specified in:** [dcl.constinit.specifier.not.reachable]
If the initializing declaration of a variable that has the `constinit` specifier applied to declarations not reachable from that initializing declaration the program is ill-formed, no diagnostic required.

2 [*Example*:

— *end example*]

JMB: produce an example

### ($D + b$).6.1.2 **The `inline` specifier** **[ifndr.dcl.inline]**

### **(D+b).6.1.2 The `inline` specifier** **[ifndr.dcl.inline]**

1 **Specified in:** [dcl.inline.missing.on.definition]
If a function or variable with external or module linkage is declared inline but there is no inline declaration reachable from the end of some definition domain the program is ill-formed, no diagnostic required.

2 [*Example*:

— *end example*]

JMB: produce an example

### ($D + b$).6.2 **Functions** **[ifndr.dcl.fct]**

### **(D+b).6.2 Functions** **[ifndr.dcl.fct]**

### ($D + b$).6.2.1 **Default arguments** **[ifndr.dcl.fct.default]**

### **(D+b).6.2.1 Default arguments** **[ifndr.dcl.fct.default]**

1 **Specified in:** [dcl.fct.default.inline.same.defaults]
If the accumulated set of default arguments for a given inline function with definitions in multiple translation units is different at the end of different translation units, the program is ill-formed, no diagnostic required.

2 [*Example*:

— *end example*]

JMB: produce an example

### ($D + b$).6.3 **Function contract specifiers** **[ifndr.dcl.contract]**

### **(D+b).6.3 Function contract specifiers** **[ifndr.dcl.contract]**

### ($D + b$).6.3.1 **General** **[ifndr.dcl.contract.func]**

### **(D+b).6.3.1 General** **[ifndr.dcl.contract.func]**

1 **Specified in:** [dcl.contract.func.mismatched.contract.specifiers]
If two different first declarations of a function (which must therefore not be reachable from one another) do not have equivalent function contract specifiers the program is ill-formed, no diagnostic required.

2 [*Example*:

— *end example*]

JMB: produce an example

### ($D + b$).6.4 **Function definitions** **[ifndr.dcl.fct.def]**

### **(D+b).6.4 Function definitions** **[ifndr.dcl.fct.def]**

### ($D + b$).6.4.1 **Replaceable function definitions** **[ifndr.dcl.fct.def.replace]**

### **(D+b).6.4.1 Replaceable function definitions** **[ifndr.dcl.fct.def.replace]**

1 **Specified in:** [dcl.fct.def.replace.bad.replacement]
A declaration of a replaceable function that is inline, not attached to the global module, does not have C++ language linkage, does not have the required return type, or is not a valid redeclaration of the corresponding declaration in a standard library header (if there is one) then the program is ill-formed, no diagnostic required.

2 [*Example*:

— *end example*]

JMB: produce an example

### ($D + b$).6.5 **Linkage specifications** **[ifndr.dcl.link]**

### **(D+b).6.5 Linkage specifications** **[ifndr.dcl.link]**

1 **Specified in:** [dcl.link.mismatched.language.linkage]
If two declarations of an entity do not have the same language linkage and neither is reachable from the other the program is ill-formed, no diagnostic required.

2 [*Example*:

— *end example*]

### ($D + b$).6.6 **Attributes** **[ifndr.dcl.attr]**

### **(D+b).6.6 Attributes** **[ifndr.dcl.attr]**

### ($D + b$).6.6.1 **Alignment specifier** **[ifndr.dcl.align]**

**(D+b).6.6.1 Alignment specifier** **[ifndr.dcl.align]**

1 **Specified in:** [dcl.align.diff.translation.units]
No diagnostic is required if declarations of an entity have different *alignment-specifier*s in different translation units.

2 [*Example*:

Translation unit #1

```
struct S { int x; } s, *p = &s;
```

Translation unit #2

```
struct alignas(16) S;    // ill-formed, no diagnostic required, definition of S lacks alignment
extern S* p;
```

— *end example*]

### ($D + b$).6.6.2 **Indeterminate storage** **[ifndr.dcl.attr.indet]**

**(D+b).6.6.2 Indeterminate storage** **[ifndr.dcl.attr.indet]**

1 **Specified in:** [dcl.attr.indet.mismatched.declarations]
If two first declarations of a function declare a function parameter with mismatched uses of the `indeterminate` attribute, the program is ill-formed, no diagnostic required.

2 [*Example*:

— *end example*]

JMB: produce an example

### ($D + b$).6.6.3 **Noreturn attribute** **[ifndr.dcl.attr.noreturn]**

**(D+b).6.6.3 Noreturn attribute** **[ifndr.dcl.attr.noreturn]**

1 **Specified in:** [dcl.attr.noreturn.trans.unit.mismatch]
No diagnostic is requried if a function is declared in one translation unit with the `noreturn` attribute but has declarations in other translation units without the attribute.

2 [*Example*:

Translation unit #1

```
[[noreturn]] void f() {}
```

Translation unit #2

```
void f(int i);          // ill-formed no diagnostic required, declared without noreturn
```

— *end example*]

### ($D + b$).7 **[module]: Modules** **[ifndr.module]**

### (D+b).7 [module]: Modules [ifndr.module]

### ($D + b$).7.1 **Module units and purviews** **[ifndr.module.unit]**

### (D+b).7.1 Module units and purviews [ifndr.module.unit]

1 **Specified in:** [module.unit.reserved.identifiers]
All *module-name*s either beginning with an identifier consisting of std followed by zero or more digits or containing a reserved identifier([lex.token]) are reserved and shall not be specified in a *module-declaration*; no diagnostic is required.

2 [*Example*:

```
module std;             // ill-formed no diagnostic required, std is not allowed at the beginning
module module;          // ill-formed no diagnostic required, module is a reserved identifier
module std0;            // ill-formed no diagnostic required, std followed by digits is not allowed at the beginning
export module _Test;    // ill-formed no diagnostic required, _Test is a reserved identifier
export module te__st;   // ill-formed no diagnostic required, te__st is a reserved identifier
```

— *end example*]

3 **Specified in:** [module.unit.named.module.no.partition]
A named module shall contain exactly one module interface unit with no module-partition, known as the primary module interface unit of the module; no diagnostic is required.

4 [*Example*:

```
module A;
export import :Internals;       // ill-formed no diagnostic required, module partition not allowed
```

— *end example*]

5 **Specified in:** [module.unit.unexported.module.partition]
If a module partition of a module that is a module interface unit but is not directly or indirectly exported by the primary module interface unit([module.import]), the program is ill-formed, no diagnostic required.

6 [*Example*:

— *end example*]

JMB: produce an example

### ($D + b$).7.2 **Private module fragment** **[ifndr.module.private.frag]**

### (D+b).7.2 Private module fragment [ifndr.module.private.frag]

1 **Specified in:** [module.private.frag.other.module.units]
If a module has a private module fragment and there is another module unit of that module, the program is ill-formed, no diagnostic required.

2 [*Example*:

— *end example*]

JMB: produce an example

### ($D + b$).8 [class]: Classes [ifndr.class]

**(D+b).8 [class]: Classes [ifndr.class]**

#### ($D + b$).8.1 Initializing bases and members [ifndr.class.base.init]

**(D+b).8.1 Initializing bases and members [ifndr.class.base.init]**

1 **Specified in:** [class.base.init.delegate.itself]
If a constructor delegates to itself directly or indirectly, the program is ill-formed, no diagnostic required

2 [*Example*:

```
struct C {
  C( int ) { }                  // #1: non-delegating constructor
  C(): C(42) { }                // #2: delegates to #1
  C( char c ) : C(42.0) { }     // #3: ill-formed no diagnostic required due to recursion with #4
  C( double d ) : C('a') { }    // #4: ill-formed no diagnostic required due to recursion with #3
};
```

— *end example*]

#### ($D + b$).8.2 Virtual functions [ifndr.class.virtual]

**(D+b).8.2 Virtual functions [ifndr.class.virtual]**

1 **Specified in:** [class.virtual.pure.or.defined]
A virtual function must be declared pure or defined, no diagnostic required. A virtual function declared pure can be defined out of line

2 [*Example*:

```
class A {
  virtual void f();
};

int main() {
  A a;      // ill-formed no diagnostic required, virtual function that is not pure but has not definition
}
```

— *end example*]

### ($D + b$).9 [over]: Overloading [ifndr.over]

**(D+b).9 [over]: Overloading [ifndr.over]**

#### ($D + b$).9.1 User-defined literals [ifndr.over.literal]

**(D+b).9.1 User-defined literals [ifndr.over.literal]**

1 **Specified in:** [over.literal.reserved]
Some literal suffix identifiers are reserved for future standardization. A declaration whose literal-operator-id uses such a literal suffix identifier is ill-formed, no diagnostic required.

2 [*Example*:

```
float operator ""E(const char*);        // ill-formed, no diagnostic required, reserved literal suffix
double operator"" _Bq(long double);     // ill-formed, no diagnostic required, // uses the reserved identifier _Bq
```

— *end example*]

### ($D + b$).10 **[temp]: Templates** **[ifndr.temp]**

### **(D+b).10 [temp]: Templates** **[ifndr.temp]**

#### ($D + b$).10.1 **Preamble** **[ifndr.temp.pre]**

#### **(D+b).10.1 Preamble** **[ifndr.temp.pre]**

1 **Specified in:** [temp.pre.reach.def]
A definition of a function template, member function of a class template, variable template, or static data member of a class template shall be reachable from the end of every definition domain([basic.def.odr]) in which it is implicitly instantiated([temp.inst]) unless the corresponding specialization is explicitly instantiated([temp.explicit]) in some translation unit; no diagnostic is required.

2 [*Example*:

```
// a.h
template <typename T>
void f();

// a.cpp
#include "a.h"
int main() {
    f<int>();   // ill-formed no diagnostic required, function template implicity
                // instantiated but not reachable definition
}
```

— *end example*]

#### ($D + b$).10.2 **Template template arguments** **[ifndr.temp.arg.template]**

#### **(D+b).10.2 Template template arguments** **[ifndr.temp.arg.template]**

1 **Specified in:** [temp.arg.template.sat.constraints]
Any partial specializations([temp.spec.partial]) associated with the primary template are considered when a specialization based on the template template-parameter is instantiated. If a specialization is not reachable from the point of instantiation, and it would have been selected had it been reachable, the program is ill-formed, no diagnostic required.

2 [*Example*:

```
template<class T> struct A {
  int x;
};

template<template<class U> class V> struct C {
  V<int> y;
  V<int*> z;
};

C<A> c;

// Ill-formed no diagnostic required, specialization is not reachable from point
```

```
// of instantiation above and it would have been selected if it had
template<class T> struct A<T*> {
  long x;
};
```

— *end example*]

### ($D + b$).10.3 **Atomic constraints** **[ifndr.constr.atomic]**

### (D+b).10.3 Atomic constraints [ifndr.constr.atomic]

1 **Specified in:** [temp.constr.atomic.equiv.but.not.equiv]
If the validity or meaning of the program depends on whether two atomic constraints are equivalent, and they are functionally equivalent but not equivalent, the program is ill-formed, no diagnostic required.

2 [*Example*:

```
template <unsigned N> void f2()
requires Add1<2 * N>;
template <unsigned N> int f2()
requires Add1<N * 2> && true;
void h2() {
f2<0>();        // ill-formed, no diagnostic required,
                // requires determination of subsumption between atomic constraints that are
                // functionally equivalent but not equivalent
```

— *end example*]

3 **Specified in:** [temp.constr.atomic.sat.result.diff]
If, at different points in the program, the satisfaction result is different for identical atomic constraints and template arguments, the program is ill-formed, no diagnostic required.

4 [*Example*:

```
template<class T>
concept Complete = sizeof(T) == sizeof(T);

struct A;
static_assert(!Complete<A>);    // #1
struct A {};
static_assert(Complete<A>);     // ill-formed no diagnostic required, satisfaction
                                // result differs from point #1
```

— *end example*]

### ($D + b$).10.4 **Constraint normalization** **[ifndr.temp.constr.normal]**

### (D+b).10.4 Constraint normalization [ifndr.temp.constr.normal]

1 **Specified in:** [temp.constr.normal.invalid]
If during constraint normalization any such substitution results in an invalid type or expression, the program is ill-formed; no diagnostic is required

2 [*Example*:

```
template<typename T> concept A = T::value || true;
template<typename U> concept B = A<U*>;
template<typename V> concept C = B<V&>; // Ill-formed no diagnostic required, it
                                        // would form the invalid type V&* in the parameter mapping
```

— *end example*]

There are two earlier "no diagnostic requireds" in this section, i am not sure if they a really are worth calling distinct cases of ifndr.

### ($D+b$).10.5 **Partial specialization** **[ifndr.temp.spec.partial]**

### (D+b).10.5 Partial specialization [ifndr.temp.spec.partial]

1 **Specified in:** [temp.spec.partial.general.partial.reachable]
A partial specialization shall be reachable from any use of a template specialization that would make use of the partial specialization as the result of an implicit or explicit instantiation; no diagnostic is required.

2 [*Example*:

```
template<typename T> class X{
public:
  void foo(){};
};

template class X<void *>;       // ill-formed no diagnostic required, explicit instantiation
                                // and partial specialization is not reachable

template<typename T> class X<T*>{
public:
  void baz();
};
```

— *end example*]

### ($D+b$).10.6 **Names of template specializations** **[ifndr.temp.names]**

### (D+b).10.6 Names of template specializations [ifndr.temp.names]

### ($D+b$).10.7 **Function templates** **[ifndr.temp.fct]**

### (D+b).10.7 Function templates [ifndr.temp.fct]

### ($D+b$).10.7.1 **Function template overloading** **[ifndr.temp.over.link]**

### (D+b).10.7.1 Function template overloading [ifndr.temp.over.link]

1 **Specified in:** [temp.over.link.equiv.not.equiv]
If the validity or meaning of the program depends on whether two constructs are equivalent, and they are functionally equivalent but not equivalent, the program is ill-formed, no diagnostic required.

2 [*Example*:

```
template <int I>
struct A{};

// ill-formed, no diagnostic required, the following declarations are functionally equivalent but not equivalent
template <int I> void f(A<I>, A<I+10>);
template <int I> void f(A<I>, A<I+1+2+3+4>);
```

— *end example*]

### $(D + b)$.10.8 **Name resolution** **[ifndr.temp.res]**

**(D+b).10.8 Name resolution** **[ifndr.temp.res]**

### $(D + b)$.10.8.1 **General** **[ifndr.temp.res.general]**

**(D+b).10.8.1 General** **[ifndr.temp.res.general]**

1 **Specified in:** [temp.res.general.default.but.not.found]
If the validity or meaning of the program would be changed by considering a default argument or default template argument introduced in a declaration that is reachable from the point of instantiation of a specialization([temp.point]) but is not found by lookup for the specialization, the program is ill-formed, no diagnostic required.

2 [*Example*:

— *end example*]

SY, JMB: We need to produce an example for this case.

### $(D + b)$.10.8.2 **Dependent name resolution** **[ifndr.temp.dep.res]**

**(D+b).10.8.2 Dependent name resolution** **[ifndr.temp.dep.res]**

### $(D + b)$.10.8.2.1 **Point of instantiation** **[ifndr.temp.point]**

**(D+b).10.8.2.1 Point of instantiation** **[ifndr.temp.point]**

1 **Specified in:** [temp.point.diff.pt.diff.meaning]
A specialization for a class template has at most one point of instantiation within a translation unit. A specialization for any template may have points of instantiation in multiple translation units. If two different points of instantiation give a template specialization different meanings according to the one-definition rule (6.3), the program is ill-formed, no diagnostic required.

2 [*Example*:

```
// a.h
#include <type_traits>

template <typename T, typename Enabler = void>
struct is_complete : std::false_type {};

template <typename T>
struct is_complete<T, std::void_t<decltype(sizeof(T) != 0)>> : std::true_type {};

// a.cpp
#include "a.h"
struct X;
static_assert(!is_complete<X>::value);

// b.cpp
#include "a.h"
struct X { };
static_assert(is_complete<X>::value);
```

— *end example*]

### $(D + b)$.10.8.2.2 **Candidate functions** **[ifndr.temp.dep.candidate]**

### (D+b).10.8.2.2 Candidate functions [ifndr.temp.dep.candidate]

1 **Specified in:** [temp.dep.candidate.different.lookup.different]
If considering all function declarations with external linkage in the associated namespaces in all translations would make a dependent call([temp.dep]) ill-formed or find a better match, the program is ill-formed, no diagnostic required.

2 [*Example*:

— *end example*]

JMB: produce an example

### $(D + b)$.10.8.3 **Explicit instantiation** **[ifndr.temp.explicit]**

### (D+b).10.8.3 Explicit instantiation [ifndr.temp.explicit]

1 **Specified in:** [temp.explicit.decl.implicit.inst]
An entity that is the subject of an explicit instantiation declaration and that is also used in a way that would otherwise cause an implicit instantiation([temp.inst]) in the translation unit shall be the subject of an explicit instantiation definition somewhere in the program; otherwise the program is ill-formed, no diagnostic required.

2 [*Example*:

```
// Explicit instantiation declaration
extern template class std::vector<int>;

int main() {
  std::cout << std::vector<int>().size();   // ill-formed no diagnostic required, implicit instantiation
                                            // but no explicit instantiation definition
}
```

— *end example*]

### $(D + b)$.10.8.4 **Explicit specialization** **[ifndr.temp.expl.spec]**

### (D+b).10.8.4 Explicit specialization [ifndr.temp.expl.spec]

1 **Specified in:** [temp.expl.spec.unreachable.declaration]
If an implicit instantiation of a template would occur and there is an unreachable explicit specialization that would have matched, the program is ill-formed, no diagnostic required.

2 [*Example*:

— *end example*]

JMB: produce an example

3 **Specified in:** [temp.expl.spec.missing.definition]
If an explicit specialization of a template is declared but there is no definition provided for that specialization, the program is ill-formed, no diagnostic required.

4 [*Example*:

— *end example*]

JMB: produce an example

### ($D + b$).10.9 **Template argument deduction** **[ifndr.temp.deduct]**

**(D+b).10.9 Template argument deduction** **[ifndr.temp.deduct]**

### ($D + b$).10.9.1 **General** **[ifndr.temp.deduct.general]**

**(D+b).10.9.1 General** **[ifndr.temp.deduct.general]**

1 **Specified in:** [temp.deduct.general.diff.order]
If substitution into different declarations of the same function template would cause template instantiations to occur in a different order or not at all, the program is ill-formed; no diagnostic required.

2 [*Example*:

```
template <class T> struct A { using X = typename T::X; };
template <class T> typename T::X h(typename A<T>::X);           // #1
template <class T> auto h(typename A<T>::X) -> typename T::X;   // redeclaration #2
template <class T> void h(...) { }

void x() {
  h<int>(0);    // ill-formed, no diagnostic required
                // #1 fails to find T::X and instantiates nothing
                // #2 instantiates A<T>
}
```

— *end example*]

JMB: Someone should confirm that the comments correctly describe why this example is ill-formed.

### ($D + b$).11 **[cpp]: Preprocessing directives** **[ifndr.cpp]**

**(D+b).11 [cpp]: Preprocessing directives** **[ifndr.cpp]**

### ($D + b$).11.1 **Conditional inclusion** **[ifndr.cpp.cond]**

**(D+b).11.1 Conditional inclusion** **[ifndr.cpp.cond]**

1 **Specified in:** [cpp.cond.defined.after.macro]
If the expansion of a macro produces the preprocessing token `defined` the program is ill-formed, no diagnostic required.

2 [*Example*:

```
#define A defined
#if A   // Ill-formed no diagnostic required, defined is generated by macro replacement
        // in controlling expression
#endif
```

— *end example*]

3 **Specified in:** [cpp.cond.defined.malformed]
If the `defined` unary operator is used when it does not match one of the specified grammatical forms, the program is ill-formed, no diagnostic required.

4 [*Example*:

```
#define A
#define B A)
#if defined ( B // Ill-formed no diagnostic required, unary operator defined did not match
                // valid form before replacement
#endif
```

— *end example*]

($D + b$).11.2 **Source file inclusion** **[ifndr.cpp.include]**

**(D+b).11.2 Source file inclusion** **[ifndr.cpp.include]**

1 **Specified in:** [cpp.include.malformed.headername]
If the *header-name-tokens* after an `include` directive cannot be formed into a *header-name* (with implementation-defined treatment of whitespace), the program is ill-formed, no diagnostic required.

2 [*Example*:

```
#include ``      // Ill-formed no diagnoatic required, does not match one of the two allowable forms
```

— *end example*]

# 3 Conclusion

That was an annex.

# Acknowledgments

Thanks to all of those who have worked to make and improve this annex.

# Bibliography

[P1705R1] Shafik Yaghmour, “Enumerating Core Undefined Behavior”, 2019
http://wg21.link/P1705R1

[P2234R0] Scott Schurr, “Consider a UB and IF-NDR Audit”, 2020
http://wg21.link/P2234R0

[P3075R0] Shafik Yaghmour, “Adding an Undefined Behavior and IFNDR Annex”, 2023
http://wg21.link/P3075R0

[P3100R5] Timur Doumler and Joshua Berne, “Implicit contract assertions”, 2025
http://wg21.link/P3100R5