N3549 - Resolved & discarded, IV

Author: Javier A. Múgica

Introduction

This proposal follows the series of proposals on the *discarded* and *resolved* concepts. It constitutes a simplification of the previous one in several respects. In the first place, many points have been extracted to separate proposals. Most notably the clarification of what "to evaluate" a type name means.

In connection with the problem of evaluation of types, this papers takes the assumed interpretation that integer constant expressions as part of type names “do not exist” for the purposes of containing expressions. We find our previous approach more precise, but it was disliked. This interpretation allows one to remove the main remaining complication of the proposal, that had to deal with the fact that subexpressions of a discarded expression might be evaluated, contrary to common sense. It is not that now they are not evaluated, but that they are ignored, in agreement with the common interpretation.

The relative concept *discarded relative to* has been replaced by a “punctual” concept *discarded at*. The old concept is implicit, without formulating it, when saying “an expression discarded at some point within the expression”; this means, in the old terminology, that the first expression is discarded relative to the latter.

The term *value-discarded* has been replaced with *discarded*, for it is how we used to refer to it. Since an expression in C cannot be totally discarded, in the sense of being parsed only to check syntax, the precision *value-* does not add more meaning. This way it also fits better with uses like *discarded at*, *discarded by*, etc.

Finally, we have noted that discarded expressions also have its address discarded; or expressed otherwise, that if an expression is needed only for its address it cannot be discarded. For example, it is not possible to evaluate `&ident` if `ident` is not defined, nor `&A[8]` if `A` has less than eight elements.

Other proposals that originated from this proposals and were separated from it modify parts of the text also modified by this proposal. In particular, there are several modifications proposed to the text on constant expressions. If needed, the author will provide a text integrating all the changes.

Examples

Constant expressions

The text on constant expressions includes the following constraint:

Constant expressions shall not contain assignment, increment, decrement, function-call, or comma operators, except when they are contained within a subexpression that is not evaluated.

Consider then the expression

```
1 ? x : (2, 3)
```

according to the wording above, the expression `(2, 3)` here is an integer constant expression, for its subexpression `2, 3` is not evaluated.

For another example, suppose an implementation accepting the following as constant expression: `a ? 0 : 0`, and consider the following piece of code:

```
if(1){
      /* ... */
}else{
      n = a++ ? 0 : 0;
}
```

Again, according to the letter of the standard `a++ ? 0 : 0` here is a constant expression, since `a++` is not evaluated.

The problem with an increment operator in a constant expression is that either the expression cannot be replaced by a constant with its value and type, for the increment would be skipped, or the translator keeps the increment, in which case the constant expressions would have side effects. In this example the translator can *replace* all the expression by `0` without changing the semantics of the program. But this should not imply that the expression *is* an integer constant expression. The latter should be derived from properties internal to the expression, independent of where the expression is placed in the code.

Here is another example:

```
int A[b ? 1 : (a++, 1)];
```

If he translator can guess that `b` cannot be zero at that point, then `b ? 1 : (a++, 1)` can be taken as a constant expression, for it can be computed to be 1 (if the translator is smart enough) and satisfies the constraint, since neither the increment nor the comma operator are evaluated. Suppose now that the implementation does not support VLA. Then it may consider `A` to be a fixed length array of length one.

Thus, the current wording of the constraint not only can "create" constant expressions depending of its placement in the code, but makes some expressions constant expressions or not depending on runtime conditions (in the example above, that `b` will never be zero).

Allowing constructs in not evaluated expressions

The standard allows the use of an identifier for which there is no definition in some contexts (6.9.1):

[...] there shall be exactly one external definition for the identifier [...], unless it is:

— part of the operand of a **`sizeof`** expression which is an integer constant expression;

— part of the operand of a **`_Countof`** expression which is is an integer constant expression;

— part of the operand of an **`alignof`** operator;

— part of the controlling expression of a generic selection;

— part of the expression in a generic association that is not the result expression of its generic selection;

— or, part of the operand of any typeof operator whose result is not a variably modified type.

The whole list appears twice. Furthermore, it falls short, for it should also include the right operand of an `||` or `&&` operator when the first operand is an integer constant expression with value $\neq 0$ or 0 respectively, the second or third operand of the conditional operator when the first one is an integer constant expressions, and compound literals in function prototypes.

Consider now the following code:

```
#define SAFE_ACCESS(a, x) (((x) < ARRAY_LENGTH(a)) ? a[x] : 0)
int a[3], b;
b = SAFE_ACCESS(a,8);
```

When expanded, the assignment becomes `b = ((8<3) ? a[8] : 0)`. We would like to make an access to an array of known constant length by a subscript which is an integer constant expression

exceeding the length of the array a constraint violation. But it seems uses like the above should be allowed. (If this constraint is introduced, the macro **SAFE_ACCESS** becomes superfluous, and writing `b = a[8]` would raise a diagnostic, which is better than what the macro does. But that constraint does not exist yet and introducing it now could break existing code.) Bounding the constraint by "which is evaluated" (i.e., allowing those out of bound indices in expressions that are not evaluated) is not possible for the same reason as for constant expressions. The allowed places have to be identified otherwise. There are ten such places: the six in the above list plus those related to the operators `||`, `&&` and `? :`, and compound literals with function prototype scope.

Some concept capturing that set of locations is obviously needed.

The concepts introduced

The concept for expressions which are not evaluated during translation and can easily be determined not to be ever evaluated during program execution is *discarded*. This is the concept to be written in place of the above list whenever we want to allow some constructions to appear in those places: Identifiers without definition, subscripts out of bounds, etc.

Discarded expressions are discarded at some point when translating the code. For example, if the first operand of an `||` operator is a nonzero ICE, the second operand *gets* discarded, thereby becoming discarded. Thus, *discarded* has two meanings: A static one, naming a property of an expression, as can be being a constant, being long or being blue. On the other hand, a dynamic property: the translator discards the expression at some point, or the expression is discarded at some point. This is the concept to be used for the recursive definition of integer and arithmetic constant expressions (and we hope, in a short future, for any constant expression).

A type name may also be said to be discarded, with obviuos semantics.

As long as there remain non-recursive definitions for constant expressions, there remains with it the constraint that constant expressions cannot include certain kinds of operators. It is necessary to replace in that constraint the current proviso "except when they are contained within a subexpression that is not evaluated". As as been argued above, this formulation turns into integer constant expressions many expressions that are not. Any piece of definition of constant expressions shall be internal to the expressions; not refer to nor depend on anything containing the expression. The correct formulation is "except when they are discarded at some point within the expression". This expresses, without saying it, the old concept of *discarded relative to*.

Dicarding depending on constant expressions not ICE.

For coherence with ICE, we would like to mandate that `(1.0+1.0) || n` is an arithmetic constant expression. However, since arithmetic constant expressions need not be evaluated during translation, the previous expression would not be known during translation to be or not to be an arithmetic constant expression, and this cannot be. We suggest to allow it as an implementation extension.

"The parenthesized name of a type"

For the sizeof operator, we think that saying that the operand may be "the parenthesized name of a type" can be problematic for any sentence of the standard that may speak about operands supposing they are expressions or type names and, since the **( )** are part of the syntax, we believe it is more correct to say that the operand is a type name, not the parenthesized name of a type. This is the criterion followed by **alignof**. Had the syntax rule been written as **sizeof** *paren-type-name*, with a rule following specifying that *paren-type-name* is **(** *type-name* **)**, then it would be right to say that the operand is a parenthesized type name. Therefore, we have applied the cirterion in **alignof** also to **sizeof**.

Proposal I. Terminology

5.2.2.4 Program semantics

3 *Evaluation* of an expression in general includes both value computations and initiation of side effects. Value computation for an lvalue expression includes determining the identity of the designated object. During translation, the value and side effects of some expressions are discarded, as well as its address if it has one, but not its type. These expressions are called *discarded*. Discarded expressions are not evaluated.

6.5 Expressions

6.5.1 General

Semantics

[...]

4 The grouping of operators and operands is indicated by the syntax.[82)] If an expression is discarded at some point, all its subexpression that were not yet discarded (that is, not discarded by the expression or some of its subexpressions) are also discarded at that point. Except as specified later, side effects and value computations of subexpressions are unsequenced.[83)]

6.5.2 Primary expressions

6.5.2.1 Generic selection

Semantics

3 The generic controlling operand is not evaluated. If a generic selection has a generic association with a type name that is compatible with the controlling type, then the result expression of the generic selection is the expression in that generic association. Otherwise, the result expression of the generic selection is the expression in the **`default`** generic association. None of the expressions from any other generic association of the generic selection is evaluated.The generic selection discards its controlling operand and the expressions from the associations other than the result expression.

6.5.3.6 Compound literals

Semantics

5 For a *compound literal* associated with function prototype scope:

[...]

— if it is not a compound literal constant, neither the compound literal as a whole nor any of the initializers are evaluated.; the parameter declaration of which it is part discards the compound literal.

6.5.4 Unary operators

6.5.4.5 The `sizeof`, `_Countof` and `alignof` operators

Semantics

2 The **`sizeof`** operator yields the size (in bytes) of its operand, which may be an expression or a type name. The size is determined from the type of the operand. The result is an integer. If the type of the operand is a variable length array type, the operand is evaluated; otherwise, the operator discards its operand.

3 The **`alignof`** operator yields the alignment requirement of its operand type. The operand is not evaluated and the expression is an integer constant expression. When applied to an array type, the result is the alignment requirement of the element type. The operator discards its operand.

5 The **`_Countof`** operator yields the number of elements of its operand. The number of elements is determined from the type of the operand. The result is an integer. If the number of elements of the array type is variable, the operand is evaluated; otherwise, the operand is not evaluated and the expression is an integer constant expression.the operator discards its operand.

6.5.14 Logical AND operator

4 Unlike the bitwise binary & operator, the && operator guarantees left-to-right evaluation; if the second operand is evaluated, there is a sequence point between the evaluations of the first and second operands. If the first operand compares equal to 0, the second operand is not evaluated.; if, in addition, the first operand is an integer constant expression, the operator discards its second operand.

6.5.15 Logical OR operator

4 Unlike the bitwise binary | operator, the || operator guarantees left-to-right evaluation; if the second operand is evaluated, there is a sequence point between the evaluations of the first and second operands. If the first operand compares unequal to 0, the second operand is not evaluated.; if, in addition, the first operand is an integer constant expression, the operator discards its second operand.

6.5.16 Conditional operator

5 The first operand is evaluated; there is a sequence point between its evaluation and the evaluation of the second or third operand (whichever is evaluated). The second operand is evaluated only if the first compares unequal to 0; the third operand is evaluated only if the first compares equal to 0;. If the first operand is an integer constant expression, the conditional operator discards its unevaluated operand. Tthe result is the value of the second or third operand (whichever is evaluated), converted to the type described subsequently in this subclause.[111)]

6.6 Constant expressions

6.6.1 General

Constraints

3 Constant expressions shall not contain assignment, increment, decrement, function-call, or comma operators, except when they are contained within a subexpression that is not evaluateddiscarded at some point within the expression.[116)]

6.7 Declarations

6.7.3.6 Typeof specifiers

4 The typeof specifier applies the typeof operators to an *expression* (6.5.1) or a type name. If the typeof operators are applied to an expression, they yield the type of their operand.[149)] Otherwise, they designate the same type as the type name with any nested typeof specifier evaluated.[150)] If the type of the operand is a variably modified type, the operand is evaluated; otherwise, the operand is not evaluatedthe typeof specifer discards it operand.

6.7.6 Alignment specifier

7 The first form is equivalent to **`alignas`**`(`**`alignof`**`(`*type-name*`))`. In particular, the alignment specifier discards the type name.

6.7.8 Type names

4 When a type name is discarded, the expressions it contains that are not integer constant expressions are discarded.

6.9 External definitions

6.9.1 General

Constraints

[...]

3 There shall be no more than one external definition for each identifier declared with internal linkage in a translation unit. Moreover, if an identifier declared with internal linkage is used in an expression there shall be exactly one external definition for the identifier in the translation unit, unless it is discarded.:

[116)] The operand of a typeof (6.7.3.6), **`sizeof`**, or **`alignof`** operator is usually not evaluateddiscarded (6.5.4.5)

— part of the operand of a **`sizeof`** expression which is an integer constant expression;

— part of the operand of a **`_Countof`** expression which is an integer constant expression;

— part of the operand of an **`alignof`** operator;

— part of the controlling expression of a generic selection;

— part of the expression in a generic association that is not the result expression of its generic selection;

— or, part of the operand of any typeof operator whose result is not a variably modified type.

Semantics

[...]

5 An external definition is an external declaration that is also a definition of a function (other than an inline definition) or an object. If an identifier declared with external linkage is used in an expression (other than as part of the operand of a typeof operator whose result is not a variably modified type, part of the controlling expression of a generic selection, part of the expression in a generic association that is not the result expression of its generic selection, or part of a **`sizeof`** or **`alignof`** operator whose result is an integer constant expression)which is not discarded, somewhere in the entire program there shall be exactly one external definition for the identifier; otherwise, there shall be no more than one

Proposal II. Allowing discarded expressions when only the type is needed

Subscript out of bounds

6.5.3.2 Array subscripting

We propose two alternatives. On the left, only “overbounds” are allowed. On the right, we also allow underbounds:

If one of the operands has array type and the subscript is an integer constant expression, the value of the subscript shall not be negative.

If one of the operands has array type and the subscript is an integer constant expression, the value of the subscript shall not be negative; if further the expression is not discarded, the value of the subscript shall be less than the length of the array or equal to it, the latter only if the `[]` operator is followed by zero or more `[]` operators with subscripts equal to zero and the resulting postfix expression is the operand of the unary & operator or is converted to an expression of pointer type as described in 6.3.3.1.

If the expression is not discarded, one of the operands has array type and the subscript is an integer constant expression, the value of the subscript shall not be negative and shall be less than the length of the array or equal to it, the latter only if the `[]` operator is followed by zero or more `[]` operators with subscripts equal to zero and the resulting postfix expression is the operand of the unary & operator or is converted to an expression of pointer type as described in 6.3.3.1.

(Within paragraph 4)

`m` shall not be negative and shall be less than the length of the array or equal to it; it shall only equal the length of the array if the `[]` operator is followed by zero or more `[]` operators with subscripts equal to zero and the resulting postfix expression is the operand of the unary & operator or is converted to an expression with pointer type as described in 6.3.3.1.the latter only in the same situation as expressed in the constraint.

Constant expressions

Here we do not colour the text. All text is new and replaces the paragraphs defining integer and arithmetic constant expressions.

6.6 Constant expressions

6.6.1 General

8 An *integer constant expression* is an expression of integer type, that satisfies the constraints for constant expressions, and which is either

— a literal, a named constant or a compound literal constant;

— a cast expression where the operand is a floating, named or compound literal constant of arithmetic type, possibly enclosed in parentheses;

— a parenthesized integer constant expression, or

— an expression where each of its operands is either discarded by the expression or an integer constant expression.[118)]

10 An *arithmetic constant expression* is an expression of artithmetic type, that satisfies the constraints for constant expressions, and which is either

— a literal, a named constant or a compound literal constant;

— a parenthesized arithmetic constant expression, or

— an expression where each of its operands is either discarded by the expression or an arithmetic constant expression.

(Remove the last footnote in 6.6.1 and add the following example)

18 EXAMPLE In the following code sample

```
int a, *p;
int f(void);
static int i = 2 || 1 / 0;
static int j = 2 || a + f();
static int k = sizeof p[sizeof(int[a])];
```

the three initializers are valid integer constant expressions with values 1, 1 and **sizeof**(**int**) respectively.

[118)] In particular, an expression of integer type that discard all its operands is an integer constant expression. This is the case of **alignof** and, in some cases, **sizeof** and **_Countof**.

Integration with other proposals

The isolated sentence added on the meaning of "discarded" when applied to a type names fits better if the text from *What does to evaluate a type name mean?* is included, putting the sentence after that text. Thust it is also made clear that the exclusion of integer constant expression from the discarded expressions is not because they are to remain, undiscarded, but because they disappear in the process of translation of the type name.

The text proposed here ignores, in integer constant expressions, generic selections where the resulting expressions is a floating expression valid as the operand to an integer cast. This is the current situation. Fixing this is part of the contents of the proposal *Quasi-constants*.

The situation of **_Generic** in other places in integer and arithmetic constant expressions is made clear by the recursive definition and by the sentences saying, essentially, that **_Generic** discards everything in it except the result expression. A technicality remains, though: the recursive rule requires discarded *operands*. What the operands of **_Generic** are, is the subject of another proposal.

The text on integer constant expressions has to be adapted to reflect the proposal *floating literals converted to bool*, in the event that it is accepted. Both proposals jointly, in turn, can be given a better wording if the proposal *phrase bool as bool* is accepted (see the example wording below).

A further proposal that affects the text on integer constant expressions is *array subscripting without decay*. The consideration of **constexpr** arrays as named constants has been split from that proposal but is likely to be presented in a near future.

Two further questions

We suggest here two extensions at the choice of the implementation. If the answer to any of the questions is positive, the corresponding extension will be presented in a future paper.

Does WG14 wish to allow implementations to make the `||`, `&&` *and conditional operators discard the second (or third) operand when the first one is a constant expression other than an integer constant expression?*

These are cases like `1.0+1.0 || E`, where the OR expression may discard the second operand.

Does WG14 wish to allow implementations to make some operators discard an operand when it can be determined that the operand will never be executed in the abstract machine?

These are cases like `unsigned int f(void); (f()>=0) || E;` Or like `int n=1; (n==1) || E;` This cannot create any new integer constant expression, because the operand that is not discarded is not an integer constant expression (if it were, the other operand would have been discarded without the need of this extension).

In both cases, since the second (or third) operand is, already, never evaluated, no observable effect is being lost by discarding it. Discarding it implies that constructions that are only allowed in discarded contexts would be allowed in those places. E.g., if `A` is an array of three elements, to allow `b>=0 || A[4]`. It also implies, in the first case, that in situations like `(1.0+1.0) || n`, if the implementation discards the second operand, the whole expression would become a constant expression. We see no problem in this, but wording can be inserted so that, when an operand is discarded as an implementation extension, the implementation is not required to consider the expression a constant expression. However, since at present all the definitions in "Constant expressions" appertain only to standard constant expressions, leaving complete freedom to the implementation for its extended constant expressions, there is no need to insert any wording. (But that complete freedom and decoupling of extended vs. standard constant expressions seems an *ad hoc* introduction of extended constant expressions, and a better specification is desirable. For instance, currently an implementation is not required to consider the || of two constant expressions a constant expression.)

Example of composed wording

As was pointed in the introduction, the author may provide a wording merging all proposals accepted relating to constant expressions. In that case, it is also better to finish the rewriting of constant expressions so that all kinds of constant expressions are defined recursively based on some atoms. Here is an example of how that wording may look, including a small fix on the specification of address constants (a more important fix for those kinds of constants is still needed). It also fixes a mistake in the wording of the logical AND expressions: "The && operator shall yield 1 if both of its operands compare unequal to zero". No; consider `0 && 1/0`. Here the second operand cannot be compared to zero. When writing the proposal *phrase bool as bool* this mistake went unnoticed and was not corrected. We also make explicit that the operands to those operators (`!`, `&&`, `||` and the first of `? :`) are converted to bool; i.e., that there is an implicit conversion.

6.5.4.4 Unary arithmetic operators

5 The logical negation operator `!` converts its operand to **`bool`**; then negates it (the negation of **`false`** is **`true`**, the negation of **`true`** is **`false`**), then converts the result to **`int`**. The result has type **`int`**.[103)]

6.5.14 Logical AND operator

3 The first operand is converted to **`bool`**. There is a sequence point after this conversion. If the result of the conversion is **`false`** the second operand is not evaluated; otherwise, the second operand is converted to **`bool`**. If any of the conversions result in **`false`**, the result of the AND expression is 0; otherwise, it is 1. The result has type **`int`**. If `(`**`bool`**`)(E1)`, where `E1` is the first operand, is an integer constant expression with value **`false`**, the AND operator discards its second operand.

(And analogously for the logical OR operator. The wording for the conditional operator can be taken from phrase bool as bool without changes)

6.6 Constant expressions

6.6.1 General

Description

1 A constant expression can be evaluated during translation rather than runtime, and accordingly can be used wherever a constant is required. Constant expressions are built from some primitive units (e.g., literals), combining them in expressions together with operands discarded by the expressions.

2 A compound literal with storage-class specifier **`constexpr`** is a *compound literal constant*.

3 An identifier that is:

— an enumeration constant,

— a predefined constant, or

— declared with storage-class specifier **`constexpr`** and has an object type,

is a *named constant*.

4 A postfix expression that applies the `.` member access operator to a named or compound literal constant of structure or union type, or the subscript `[]` operator to a named or compound literal constant of array type where the subscript is an integer constant expression with a value within the range of the array, is a named or compound literal constant, respectively.

5 A *structure or union constant* is a named constant or compound literal constant with structure or union type, respectively.

6 An *immediate address constant* is a unary & expression where the operand is a function designator or an object of static storage duration, such an object of array type that is implicitly converted to a pointer or a function designator that is implicitly converted to a pointer.

[103)] If we represent by ¬ an operator interchanging **`true`** and **`false`**, for an operand `E` of any scalar type the expression `!E` is equivalent to `(`**`int`**`)¬(`**`bool`**`)E`.

7 Literals, named constants, compound literal constants, immediate address constants and the literal **`nullptr`** are collectively called *immediate constant expressions*.

8 A *formal constant expression* is one of the following:

— An immediate constant expression.

— An expression other than a comma expression where each of its operands is either discarded by the expression or a formal constant expression.

9 A *constant expression* is a formal constant expression that evaluates to a constant that is in the range of representable values for its type.

10 All (formal) constant expressions described thus far are *standard (formal) constant expressions*. An implementation may accept other forms of constant expressions, called *extended constant expressions*.

11 EXAMPLE 1

```
void func(int n){
      int i= INT_MAX + INT_MAX + n;
      int j= n + INT_MAX + INT_MAX;
}
```

The expression used to initialize `i` includes the formal constant expression **`INT_MAX + INT_MAX`** that is not a constant expression (and violates a constraint expressed in the next subclause). On the other hand, the initializer for `j` is grouped as `(n+`**`INT_MAX`**`) + `**`INT_MAX`** and this situation does not arise.

Semantics

12 If a floating expression is evaluated in the translation environment, the arithmetic range and precision is at least as great as if the expression were being evaluated in the execution environment.

13 If the member-access operator `.` accesses a member of a union constant, the accessed member shall be the same as the member that is initialized by the union constant's initializer.

14 It is implementation-defined whether extended constant expressions are usable in the same manner as standard constant expressions.

15 The semantic rules for the evaluation of a constant expression are the same as for nonconstant expressions.

6.6.2 Integer, arithmetic and address constant expressions

Description

1 Integer and arithmetic constant expressions are not just constant expressions of integer and arithmetic type respectively. In their recursive structure as constant expressions they are subject to restrictions on the primitive units they are based on, so that integer constant expressions can be evaluated during translation phase 7 and likewise arithmetic constant expressions when the implementation can perform floating point arithmetic during translation.

2 The following paragraphs provide inductive definitions for formal integer constant expressions, formal arithmetic constant expressions, and formal address constants. At the base of the induction stand immediate constant expressions. The three kinds of formal constant expression defined and the corresponding constant expressions are the standard ones. It is implementation defined what extended constant expressions belong to each category, except that any extended integer constant expression is an extended arithmetic constant expression.

3 A *formal integer constant expression* is a formal constant expression of integer type having one of the following forms:

— An immediate constant expression.

— A cast expression with integer type where the operand is, after generic replacement and removal of all surrounding parentheses, an immediate constant expression of arithmetic type.

— A parenthesized formal integer constant expression.

— An expression where each operand is either discarded, or a formal integer constant expression, or where the operand undergoes an implicit conversion to bool and the same operand cast to bool would be a formal integer constant expression.

An *integer constant expression* is a formal integer constant expression that is a constant expression.

4 A *formal arithmetic constant expression* is a formal constant expression of arithmetic type having one of the following forms:

— An immediate constant expression.

— A parenthesized formal arithmetic constant expression.

— An expression where each operand is either discarded or a formal arithmetic constant expression.

An *arithmetic constant expression* is a formal arithmetic constant expression that is a constant expression.

5 A *formal address constant* is a formal constant expression of pointer type having one of the following forms:

— An immediate address constant, optionally plus or minus a formal integer constant expression.

— An expression that, after generic replacement and removal of all surrounding parentheses, is a formal address constant.

An *address constant* is a formal address constant that is a constant expression.

Constraints

6 A formal integer constant expression that is not discarded shall be a constant expression.

7 A format arithmetic constant expression in a place where an arithmetic constant expression would be evaluated during translation phase 7 shall be an arithmetic constant expression.[1)]

Semantics

8 Integer constant expressions are always evaluated during translation phase 7. It is implementation-defined whether an arithmetic constant expression or an address constant is evaluated during translation phase 7 or later.

9 EXAMPLE 1 The expression

```
(int)(1.0+1.0)
```

is a constant expression of integer type but not a standard integer constant expression.

Likewise, the initializer for `x` in

```
static int i;
void func(void){
      float x=(float)(uintptr_t)&i;
}
```

is not a standard arithmetic constant expression.

10 EXAMPLE 2 In the following code sample

```
int a, *p;
int f(void);
static int i = 2 || 1 / 0;
static int j = 2 || a + f();
static int k = sizeof p[sizeof(int[a])];
```

the three initializers are valid integer constant expressions with values 1, 1 and `sizeof(int)` respectively.

[1)] Thus, implementations that evaluate arithmetic constant expressions during translation are required to produce a diagnostic when such an evaluation fails.

Further changes can be performed on top of this wording. First, extended constant expressions should be introduced sooner, at the level of immediate constant expressions, so that, for example, if `A` and `B` are arithmetic constant expressions, `A + B` is an arithmetic constant expressions. Or that if `A` is an integer constant expression with value 0, `A && E` is an integer constant expression (provided `E` is an expression of scalar type). This is currently not mandated. In doing so, and if the possibility is given to implementations of discarding more operands than the minimum the standard mandates, an exception in the last rule for forming formal constant expressions recursively should be added, so that discarding operands does not force the implementation to introduce more constant expressions. For example,

, except that if an operand gets discarded as an implementation extension, it is implementation defined whether the expression is a formal constant expression or not.

and similarly for integer and arithmetic constant expressions. This way, an implementation that discards the second operand in `(1.0+1.0) || n` is allowed but not required to consider it a constant expression. But if it considers `(bool)(1.0+1.0)` an integer constant expression, then it is required to treat `(1.0+1.0) || n` as an integer constant expression, because the discarding of the second operand of `||` when the first one, cast to **`bool`**, is an ICE with value true is not implementation defined. (In the latter example the extension is not in discarding operands in more situations than the minimum, but in considering more integer constant expressions than the standard ones).

In the above example wording we have included the "plus or minus a (formal) integer constant expression" in the definition of (formal) address constants, because there is currently no place in the standard that needs "address constant" as defined now, without the optional integer offset. Further, we believe that in an approach to constant expressions as the one taken here, address constant should be any constant expression of pointer type. Address constants, in any case, need to be fixed to allow, e.g., `&a.arr[E]` only when `E` is an integer constant expression (and within bounds).

The address constants consisting of an immediate address constant $\pm$ an ICE (and recursively) can be subject to a constraint: A formal address constant that is not discarded shall be an address constant. (footnote: While it may not be possible to evaluate these constants during translation phase 7, it is possible to know whether they will evaluate to a valid constant or not.)