N3724 - Discarded, V

Author: Javier A. Múgica

Introduction

This proposal follows the series of proposals on the *discarded* concept. The proposal “Resolved & Discarded IV” was simpler than previous ones. Several points not central to the proposal were left out, and it took a different, simpler, approach to the relative concept, thanks mostly to the review by M. Uecker. This was welcomed better that the former approach, and the vote at the Brno meeting manifested a clear support for the direction taken. The present proposal updates the wording to be based on the latest working draft and corrects an important mistake that was present in the wording of the sizeof, _Countof and alignof operators: As a consequence of the reelaborations of the wording, the sentence pointing that these expressions are integer constant expressions, when they are, had been removed. This was an oversight; the removal can take place only when the wording for ICE is changed to define them in terms of the discarded concept, and this is not included in the proposal.

In this version we have added the remark that a size expression that is treated as * at function-prototype scope is discarded. Currently, that the expression is not evaluated is not mentioned, so it might not be necessary to say that it is discarded, but we feel that its omission makes ambiguous whether these expressions are considered discarded or not. Probably the right solution for function parameters in function prototypes is to discard all of it, instead of special casing compound literals (6.5.3.6) or these size expressions, as was pointed by J. Myers. This is not an issue introduced by the “discarded” concept: the special casing of compound literals is present already in the standard, using the term “not evaluated”. Therefore, we do not change anything in this respect. This is better handled by a separate proposal.

The motivation was rewritten to better focus on the different, incompatible uses of "not evaluated", and one example is added to clarify the local concept.

Not Evaluated

The expression “not evaluated” can be used in three different senses. In the first place, it can mean what it says: “if `n` is 0, the expression `1/n` will have undefined behaviour whenever it is evaluated”. Thus, if the expression is evaluated it features undefined behaviour; if it is not evaluated, it does not.

In can also mean not evaluated because of its place in the code, in specific places. For example: “C allows an identifier without definition provided it is placed where it is not evaluated”. This is an informal statement. The exact list of places 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 an expression in a generic association that is not the result expression of its generic selection.

— part of an array length expression that is treated as $*$, in function-prototype scope or in the type name of a generic association.

— part of the second operand of a logial AND expression where the first operand is a integer constant expression with value zero.

— part of the second operand of a logical OR expression where the first operand is a integer constant expression with a value different from zero.

— part of the second or third operand of a conditional expression where the first operand is an integer constant expression with value zero or different from zero respectively.

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

— part of a compound literal with function-prototype scope.

This is the lists of places where the operand is *discarded.*

Finally, it can be used as in the description of constant expressions: “Constant expressions do not contain assignment, increment, decrement, function call and comma operators, except when they are contained within a subexpression that is not evaluated." What is intended here is not “not evaluated”, but “discarded” and, further, discarded within the expression itself, not because the whole expression is discarded, as in the following example:

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

According to the wording above, the expression `1+(2, 3)` here is an integer constant expression, for its subexpression `2, 3` is not evaluated. That is not what is intended.

This shows the problem of using “not evaluated” for two different concepts (the first, correct, and the third, wrong).

The absolute discarded concept

Or simply, *discarded.* This is the second of the uses above. The standard does not make the mistake of using “not evaluated” with this meaning. The workaround is awkward: when specifying the places where a file-scope identifier without definition is allowed (6.9.1), it needs to list the above set of places. The whole list appears twice. Furthermore, it misses the cases of the AND, OR and conditional operators, function-prototype compound literals and array lengths and generic association types.

The lack of the “discarded” concept implies that either the whole list is repeated whenever something is allowed only in discarded contexts, or that whatever was intended is not included in the standard. The second possibility is what happened with subscripts out of bounds in arrays, when all quantities involved are fixed (ICE for the subscript and fixed length array):

```
#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 uses in discarded contexts, as the example above, should be allowed. In the end, the committee didn’t consider the possibility of listing again the above eleven-item list (nobody even proposed it) and the constraint was not included. In contrast, because negative indices seem always wrong, a constraint was added precluding them. The outcome was that, because of the lack of the “discarded” concept, the split between what is constrained and what is not was based on negative versus too large, when it should have been based on discarded vs. not discarded.

Another constraint for which the “discarded” concept is needed is the oft-proposed integer division by zero. The wording for this is plain: “There shall not be an integer division by an integer constant expression of value zero unless the division expression is discarded.” The exception is needed, otherwise it would turn currently valid programs into invalid ones. But it cannot be formulated without “discarded”.

The local concept

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, a dynamic concept: an expression *discards* some of its operands, results in the static, absolute concept: an expression *is discarded*. The dynamic property is also local: the action of discarding takes place *at* a specific point. For example:

```
sizeof(
    2 || i    // i discarded here (at the OR expression)
);

sizeof(    // i discarded here (at the sizeof expression)
    i || 2
);
```

In the second example, `i` is discarded as part of `i||2`, that is discarded by the sizeof expression. This discarding of discarded subexpressions is expressed in the proposed wording by:

(1) 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.

This completes the identification of what expressions are discarded and at what point in the code they get discarded. Without that sentence, we would get that `i||2` is discarded (as the operand, with type of known fixed size, of a sizeof operator), but not that the expression `i` itself is discarded. Consider now the example **`sizeof`**`(1/0 || 2)`, and the constraint proposed above for integer division by constant zero. We do not want to mandate a diagnosis in this example. Hence, we need `1/0` there to be discarded.

Why not simply say that "when an expression is discarded its subexpressions also become discarded"? Yes, this is right. But this is precisely what (1) says, with the important precision that this only applies to subexpressions that had not already been discarded, so as to have only one, precise, point at which every discarded expression gets discarded.

Discarding a type name

A type name may also be said to be discarded: "When a type name is discarded, the expressions it contains that are not integer constant expressions are discarded". The exception for integer constant expressions is needed because they are always evaluated, during translation, as part of the determination of the type, as in **`_BitInt`**`(2*3)`.

"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 **`typeof`**. 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 criterion in **`typeof`** also to **`sizeof`**.

Proposed wording

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, an expression may have its value and side effects 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)]

5 EXAMPLE The following code includes operators that discard operands. The comments note the expressions that are discarded at each point.

```
sizeof(    //Discarded at the sizeof expression: 2||i, 2
  2 || i   //Discarded at the OR expression: i
)
sizeof(    //Discarded at the sizeof expression: i||2, i, 2
  i || 2
)
```

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, the type names from all the associations, 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 can 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 does not have a known constant size, the operand is evaluated; otherwise, the operand is not evaluatedthe operator discards its operand and the expression is an integer constant expression.

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 and the expression is an integer constant expression.

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 evaluatedthe operator discards it operand and the expression is an integer constant expression.

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

6 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

4 Constant expressions do 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.[117)]

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.[150)] Otherwise, they designate the same type as the type name with any nested typeof specifier evaluated.[151)] If the type of the operand is a variably modified type, the operand is evaluated; otherwise, the operand is not evaluatedthe typeof specifier discards it operand.

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

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.

6.7.7 Declarators

6.7.7.3 Array declarators

5 If the array length expression is not an integer constant expression: if it occurs in a declaration at function prototype scope or in a type name of a generic association (as described above), it is treated as if it were replaced by *, and in the first case it is discarded by the parameter declaration;[152)] otherwise [...]

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

[...]

4 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.:

- 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

[...]

6 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**, **_Countof** or **alignof** operator that 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.[201)]

[152)] In the second case it is discarded by the generic selection (6.5.2.1).

Note

If the proposal N3721 on the operands of a generic selection is approved, the sentence added for this operator here (the sentence in blue in 6.5.2.1) should be replaced by the simplified version: The generic selection discards all its operands except the result expression.