N3464 - Resolved & discarded

Author: Javier A. Múgica

This document

This document builds upon the "Discarded" paper. The motivations for introducing the term *value-discarded* are exposed in that paper, N3382 and will not be repeated here. Already in that proposal it is noted that the standard abuses the term *to evaluate* to refer to type names, and that a different concept would be needed for type names. Some definitions which seem strange with the current use of "evaluated" turn out natural if a parallel term is defined for type names and used. In particular, the seemingly contradictory fact that part of an expressions which is not evaluated is evaluated (sizes in type names).

We present here a wording bases on the concepts *to evaluate*, for expressions, and *to resolve*, for type names. Also, we changed the introduction of value-discarded subexpression, that are now defined to be *discarded relative to* instead of value-discarded. However, since value-discarded expressions are those which are discarded relative to something, there is little differentce between the two approaches.

An error in the definition of integer and arithmetic constant expressions identified by J. Myers has been corrected. A few redundant “if <*some operand*> is not value-discarded" have been removed. The possibility for implementations to consider other value-discarded expressions has been split to a subproposal.

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, some expressions are retained only for their type, their value and side effects being discarded. These expressions are called *value-discarded*. Vaue-discarded expressions are not evaluated.

4 Type names, both explicit and implicit in declarations, are *resolved*. This entails determining the type that the type name names. The type of an expression is also resolved, and the term may also be applied to the type named by a type name.[1)] Many types are resolved during translation. Other types are only *partly resolved* at translation. What remains to be resolved are the values of certain expressions on which the type depends. Every time the type name or expressions is reached during program execution, and if it is necessary to resolve the type, these expressions are evaluated, or the relevant value retrieved from some previously evaluated expression, whereby the type is *fully resolved* at runtime. Since the value of those expressions may change every time the type name or expression is reached, the resolved type may be different in each occasion. If it is not necessary to fully resolve the type, those evaluations are not performed and the type or type name is said to be *runtime-discarded*. When this term is applied to a type or type name which is resolved during translation, it is devoid of meaning.

6.5 Expressions

6.5.2 Value-discarded

1 The following subclauses identify certain expressions as *discarded relative to* some syntactic construct (expression, type name or paramter declaration). In addition, if an expression `B` is discarded relative

[1)] Thus, for a type name, it may be said that "the type name is resolved" or that "its type is resolved", indiffererntly.

to some syntactic construct `A`, it is also is discarded relative to every construct containing `A`, and any subexpression contained in `B` whose evaluation is not required for the resolution of some type is also discarded relative to `A`.

2 An expression which is discarded relative to some contruct is value-discarded.

3 Unless otherwise stated, the type of a value-discarded expression is runtime-discarded.

6.5.3 Primary expressions

6.5.3.1 Generic selection

Semantics

3 The generic controlling operand is not evaluateddiscarded relative to the generic selection. 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 expressions from the other generic associations of the generic selection are discarded relative to the generic selection.

6.5.4.6 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 compound literal is discarded relative to the parameter declaration of which it is part.

6.5.5 Unary operators

6.5.5.5 The `sizeof`, `_Lengthof` and `alignof` operators

Semantics

2 The **`sizeof`** operator yields the size (in bytes) of its operand, which may be an expression or the parenthesized name of a type. 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 operand is not evaluated and the result is an integer constant expression.If the operand is an expression, then: if it is not a variable length array the expression is discarded relative to the **`sizeof`** expression; otherwise, it is evaluated. If the type of the expression or the type denoted by the type name is not a variable length array type, the type is runtime-discarded.

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

5 The **`_Lengthof`** 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 operand is an expression and the number of elements of the array type is variable, the operand is evaluated; otherwise, the operand is not evaluateddiscarded relative to the **`_Lengthof`** expression. If the operand is the parenthesized name of a type, then: if the number of elements of the array type it denotes is fixed, the type name is runtime-discarded; otherwise, the number of elements of the array is resolved, and if the element type is an array type, this latter is runtime-discarded.

6 EXAMPLE 3 In this example, the size of a variable length array is computed and returned from a function:

```c
#include <stddef.h>

size_t fsize3(int n)
{
      char b[n+3];       // Variable length array
      return sizeof b;   // The type of b is fully resolved at runtime by
                         // retrieveing the value of the expression n+3
}                        // computed previously

int main(void)
{
      size_t size;
      size = fsize3(10); // fsize3 returns 13
      return 0;
}
```

6.5.6 Cast operators

Remove the first paragraph in "Semantics":

Size expressions and typeof operators contained in a type name used with a cast operator are evaluated whenever the cast expression is evaluated.

6.5.15 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 second operand is discarded relative to the logical AND expression.

6.5.16 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 second operand is discarded relative to the logical OR expression.

6.5.17 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 unevaluated operand is discarded relative to the conditional expression. Tthe result is the value of the second or third operand (whichever is evaluated), converted to the type described subsequently in this subclause.[2)]

6.6 Constant expressions

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 relative to the expression.[3)]

[2)] A conditional expression does not yield an lvalue.

[3)] The operand of a typeof (6.7.3.6), **sizeof**, or **alignof** operator is usually not evaluatedvalue-discarded (6.5.4.5)

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.[4)] Otherwise, they designate the same type as the type name with any nested typeof specifier evaluatedresolved.[5)] If the type of the operand is a variably modified type, the operand is evaluated; otherwise, the operand is not evaluateddiscarded relative to the typeof specification.

6.7.7.3 Array declarators

5 If the size is given by an integer constant expression, this size is always resolved during translation; hence, the expression is always evaluated, during translation.

6 If the size is an expression that is not an integer constant expression: if itIf the size is not given by an integer constant expression: if the declarator occurs in a declaration at function prototype scope, itthe size expression discarded relative to the innermost parameter declaration of which it is part and is treated as if it were replaced by $*$; otherwise, each time it is evaluatedthe declarator is reached during execution, if the array type of which it is the size needs to be resolved it is evaluated and it shall have a value greater than zero. The size of each instance of a variable length array type does not change during its lifetime. Where a size expression is part of the operand of a typeof or **sizeof** operator and changing the value of the size expression would not affect the result of the operator, it is unspecified whether or not the size expression is evaluated. Where a size expression is part of the operand of a **_Lengthof** operator and changing the value of the size expression would not affect the result of the operator, the size expression is not evaluated. Where a size expression is part of the operand of an **alignof** operator, that expression is not evaluated.

Addendum to proposal I

Insert the text in blue after the text in black:

2 An expression which is discarded relative to some contruct is value-discarded. An implementation may consider value-discarded other expressions for which it can determine during translation that they will never be evaluated (in the abstract machine) beyond those identified by this document.

The introudction of the precision "in the abstract machine" limits considerably this lee for implementations. This is intended. For example, in

```
3 + 0*n  // n of integer type
```

the implementation is not allowed to consider `n` a value-discarded expression. Hence, the whole expression displayed cannot be considered an integer constant expression.

Here follow examples of what is allowed:

```
static float func(float); // File scope identifier without definition
/* ... */
short n;
float A[1];

n>SHRT_MAX && f(n)
(n>=0 || n<0) || 1/0
n*n >=0 ? 1 : func(0)
(short)(double)n==n ? 1 : A[-1]
```

[4)] When applied to a parameter declared to have array or function type, the typeof operators yield the adjusted (pointer) type (see 6.9.2).

[5)] If the typeof specifier argument is itself a typeof specifier, the operand will be evaluatedresolved before evaluatingresolving the current typeof operator. This happens recursively until a typeof specifier is no longer the operand.

The translator is allowed to consider value-discarded the expressions `f(n)`, `1/0`, `func` and (likely) `A[-1]`. this has the effect of turning constraint violations into well defined behaviors that do not require the issue of a diagnostic. It cannot make integer constant expression an expression that is not otherwise (i.e., without considering value-discarded the subexpression in question). The condition that the expression to be considered value-discarded must never be evaluated in the abstract machine implies that it is an operand of some logical operators (AND, OR, conditional) where the first operand is a tautology or its opposite. If this tautology is expressed by an integer constant expression, the value-discarded expression is so for any implementation; if it is not expressed by an integer constant expression, the value-discarded expression is so because the implementation is "smart", but the whole cannot be considered an integer constant expression because of the non-ICE first operand.

The above clause cannot turn a non-ICE into an ICE

An implementation may extend the consideration of value-discarded to secondary blocks of conditional statements or any other expression that it can determine that will never reached. We don't expect implementations to do so in the short term, but in any case that would just remove some constraint violations for pieces of code that will never be executed. Implementations already remove those pieces of code from the generated program.

Proposal II. Allowing certain constructs in discarded expressions

Integer and arithmetic constant expressions

6.6 Constant expressions

8 An *integer constant expression*[6] shall have integer type and shall only have operands that are discarded relative to it, integer literals, named and compound literal constants of integer type, character literals, **`sizeof`** expressions whose results are integer constants expressions, **`alignof`** expressions, and floating, named or compound literal constants of arithmetic type that are the immediate operands of casts. Cast operators in an integer constant expression which are not discarded relative to it shall only convert arithmetic types to integer types, except as part of an operand to the typeof operators, **`sizeof`** operator, or **`alignof`** operator.

10 An *arithmetic constant expression* shall have arithmetic type and shall only have operands that are discarded relative to it, integer literals, floating literals, named or compound literal constants of arithmetic type, character literals, **`sizeof`** expressions whose results are integer constant expressions, and **`alignof`** expressions. Cast operators in an arithmetic constant expression which are not discarded relative to it shall only convert arithmetic types to arithmetic types, except as part of an operand to the typeof operators, **`sizeof`** operator, or **`alignof`** operator.

(Remove the last footnote 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.

[6] An integer constant expression is required in contexts such as the size of a bit-field member of a structure, the value of an enumeration constant, and the size of a non-variable length array. Further constraints that apply to the integer constant expressions used in conditional-inclusion preprocessing directives are discussed in 6.10.2.

6.7.11 Initialization

5 All the expressionsexpression initializers in an initializer for an object that has static or thread storage duration or is declared with the **`constexpr`** storage-class specifier shall be constant expressions or string literals.

Identifiers missing an external definition

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 value-discarded:

— part of the operand of a **`sizeof`** operator whose result is an integer constant expression;

— part of the operand of an **`alignof`** operator whose result is an integer constant expression;

— 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 value-discarded, somewhere in the entire program there shall be exactly one external definition for the identifier; otherwise, there shall be no more than one.

Preprocessor

The adoption of Proposal 2 enlarges the kinds of expressions considered integer constant expression. For example, `1 || &"abcd"[2]` is now a constant expression. The preprocessor need not handle this kinds of expressions. This is taken care of in the paper "Preprocessor integer expressions".

Decoupling of value-discarded from runtime-discarded

If the operand of sizeof is an expression its value is not needed for anything. Therefore, it makes sense to say that it is value-discarded, irrespective of its type. Since expressions giving the size of variable length arrays need be evaluated, and the only terms used were "evaluated" / "not evaluated", there was no other choice than to say that the expression is evaluated, or to devise a wording specific for that operand, which is possible but was not done. Now we may simply say that the expression is value-discarded and, if the operand is a variable length array, the type is resolved, otherwise it is runtime-discarded.

Examples:

```
float A[3][n*n];
sizeof(A[m++]);
sizeof(n++, A);
p=NULL;
sizeof(*(int(*)[n])(p+5));
```

In the first two examples there is nothing that needs evaluation inside the operand, for the type of the first is that of `A[0]`, which is **`float`** `[`N`]`, where N is the result of the evaluation of `n*n`, which has already taken place, and the type of the second operand is that of A, which again needs only values that have already been computed (N). The third example does need the evaluation of `n`, but skips that of `p+5`, which is not related to the resolution of the type of the expression.

If the operand is an expression, unless its type is given by a cast (or derived form it, as in the third example), there is nothing that will need evaluation in it.

Similarly for _Lengthof. With this change, the sentence "Unless otherwise stated, the type of a value-discarded expression is runtime-discarded" is necessary as written, in the sense that now there are two instances of "otherwise stated". The wording can read

The **`sizeof`** operator yields the size (in bytes) of its operand, which is determined from the type of the operand. If the operand is an expression it is discarded relative to the **`sizeof`** expression. The type of the expression is runtime-discarded unless it is a variable length array.

The **`_Lengthof`** operator yields the number of elements of tis operand, which is determined from the type of the operand. If the operand is an expression it is discarded relative to the **`_Lengthof`** expression. The element type of the operand is runtime-discarded; the number of elements of the operand type is resolved.

We have made some simplifications with respect to the current wording. That the result is an integer is omitted since a paragraph follows specifying that the result has type **`size_t`**. Also, it is not said that the operand may be "an expression or the parenthesized name of a type", since that follows from the syntax and the paragraph on the **`_Lengthof`** operator already omits it. Further, we think that saying that the operator 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 (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 parethesized type name).

The **`_Lengthof`** operator is not yet part of C, so the wording proposed here may be addopted right now. Sizeof operators taking VLA are already present and the proposed wording would introduce semantic changes. They are immediate to diagnostic: any size effect within an operand which has VLA type, other than within its size expressions. For this reason (the semantic change) we prefer to treat this in a separate proposal.

We do suggest to change know the parenthesized name of a type to a type name.