N3530 - Resolved & discarded, III

Author: Javier A. Múgica

Introduction

This document proposes the introduction of the concepts *value-discarded* and *discarded relative to*, for expressions, and *to resolve* for types. The latter is a counterpart to the concept *to evaluate,* which applies to expressions. The purpose of these terms is to formulate rigorously the concept expressed in many passages of the standard by *not evaluated,* which most often does not mean what it says; i.e., the intended meaning is not "not evaluated", but something different. To be value-discarded is an absolute property: an expressions either is value-discarded or not. *Discarded relative to,* on the other hand, is a relative property: an expression is discarded relative to some other expression, type name or any other syntactic construction of which it is part.

It has been pointed with respect to the previous version of this proposal, that it replaces a wording which is not clear by another wording which is not clear either. In order to address this criticism, this version has been simplified. The main simplification comes in the analysis, not in the wording itself. The latter has a slight simplification as a result of not modifying the relation *discarded relative to* because of ICE as array lengths, which are always evaluated.

We believe that the proposed wording is clearer than the existing one: it separates the concepts of *to evaluate* and *to resolve*, the latter applied to types and type names. The standard currently defines the evaluation of expressions, but then applies the concept also to type names, and it is not clear what it means there. Also to be noted are the simplifications in the wording for external definitions and for integer and arithmetic constant expressions. Furthermore, the introduction of the term *value-discarded* provides a uniform means to refer to places in the code where some "inevaluable" expressions are allowed, as identifiers without definitions, subscripts out of bounds, etc.

Finally with respect to this point, those who still think that the proposed wording is as complicated as the previous one, should consider the viewpoint that this paper replaces a complicated, wrong wording, by a complicated wording which is (hopefully) right.

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 left 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 expression which 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 *value-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.

Some constructs restrict the kind of subexpressions they may contain (e.g., integer constant expressions). These restrictions can be lifted for subexpressions of it which are not evaluated: why care about the contents of such subexpressions?; the only relevant property is their type. But those subexpressions cannot be referred to by "which are not evaluated", as has been argued above. These subexpressions are the ones that would not be evaluated even if the whole expression were, and the term chosen for this is *discarded relative to* the expression.

Value discarded

This is an absolute concept. It is defined as follows: First, the ten places above are identified as value-discarded expressions. Then, the property is closed “downwards”: If $\mathrm{B} \subset \mathrm{A}$ and A is value-discarded, then B is value-discarded.

Discarded relative to

This is a relative concept. It is defined as follows: First, the ten places above are identified as discarded relative to the larger expression or type name (the latter for typeof). Then, upward and downward closures apply: If $\mathrm{D} \subset \mathrm{C} \subset \mathrm{B} \subset \mathrm{A}$, where D and C are expressions and B and A are some syntactic constructs, and if C is discarded relative to B, then C is discarded relative to A and D is discarded relative to B.

Combined definition of both terms

As we have seen, neither concept need be derived from the other. The definition of both of them as presented above would require wording like the one following, for the `||` operator.

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 value-discarded and is discarded relative to the logical OR expression.*

In order to avoid the duplication “is value-discarded and is discarded relative to”, we omit value-discarded, and at some point in “6.5 Expressions” we say that an expression which is discarded relative to some syntactic construct is value-discarded.

Integer constant expressions in type names

There is an insidious problem for the above definitions: integer constant expressions within type names are evaluated, always:

```
int g(int[2-3]); //Wrong. Negative size
int f(int(*)[3]);
unsigned int n=5;

sizeof( f((int(*)[5])(void*)0) ); //Wrong parameter type
```

```
sizeof(char(*)[2-3]); //Wrong. Negative size
sizeof(sizeof(int[2-3])); //Wrong. Negative size
alignof(int[2-3]); //Wrong. Negative size
```

Therefore, it is not true that any subexpression within a value-discarded expression is not evaluated.

In the previous examples the size is not needed, and the rules for translation of C programs could have been set up so that they were not evaluated. The size is evaluated, not because its value is needed in order to determine the value of some larger expression, but because it is necessary for the resolution of the type. For this reason, in situations like $\mathrm{C} \subset$ Type $\subset \mathrm{B} \subset \mathrm{A}$, where these size expressions are C and B is discarded relative to A, we will still say that C is discarded relative to A.

Therefore, expressions are *evaluated*; types (or type names) are *resolved*. The evaluation of an expression can be omitted (and its side effects), and if so the expression is said to be value-discarded. The resolution of a type is never fully omitted. At most, the part of it that depends on the value of expressions determined at runtime. Thus, in

```
sizeof(sizeof(int[3+1]))
sizeof(sizeof(int[n+1]))
```

the expression `n+1` is never evaluated but the expression `3+1` is, because C insists in evaluating all size expressions when this can be done during translation.

For a type name like that of **`int`**`[n+1]` above, we say that it is not *runtime resolved*.

We cannot say that `3+1` above is value-discarded, if we want to preserve the meaning of this term. But we will still say, as noted above, that it is discarded relative to the outermost sizeof expression and to anything including the latter. This simplifies the wording. It causes no problem: *discarded relative to* expressions appear as subexpressions of some larger expression where some constructs are allowed which are not allowed elsewhere. For instance, an increment operator may appear in an integer constant expression provided it is contained in some subexpression discarded relative to the integer constant expressions. The inclusion of anything of the sort in the size expression will make it not an ICE, whence it becomes value-discarded (because the size of the array type is no longer resolved during translation and, being part of an expression which is discarded relative to something else, it will neither be evaluated during runtime, and becomes value-discarded).

In short, those size expressions that always get evaluated are only ICE which, by definition, can be evaluated during translation and have no side effects. Hence, they can never cause any problem, and the terms are chosen so that the wording is as simple as possible while still being rigorous. The choice is not to modify the “discarded relative to” relation because of them, but we do not say that they are value-discarded, since indeed they are not.

Not integer constant expressions in type names

Expressions in type names which are not integer constant expressions are not evaluated in some cases: function parameters, cast expressions which are not evaluated, operands of **`alignof`** and some operands of **`sizeof`** and typeof operators. With respect to the latter the standard says:

*Where an array length expression is part of the operand of a typeof or **`sizeof`** operator and changing the value of the array length expression would not affect the result of the operator, it is unspecified whether or not the size expression is evaluated.*

With the introduction of the term *to resolve* applied for type names, the distinction between translation-time resolution and runtime resolution, and the clause that says that expressions within a type name which is not runtime resolved are discarded relative to the type name (hence, value-discarded if not ICE's), it is precisely defined when those expressions are evaluated and when not. For example:

```
sizeof(sizeof(int[m]));
sizeof(int[3 + 0*sizeof(int[m])]);
sizeof(int[ n + ((int(*)[m])A)[0][0] ]);
```

First example: The type of the operand to the outer sizeof is not a variable length aray type; hence, it is value-discarded. Hence, the type `int[m]` within it is not runtime resolved. Hence, the expression `m` in the latter is not evaluated.

Second example: The operand to sizeof is an expression with variable length array type; therefore, it is evaluated, and in particular the expression `sizeof(int[m])` within it is evaluated. The operand to this latter sizeof is a type name, which is fully resolved; hence, the expression `m` on which this type depends is evaluated whenever it is reached during execution.

Third example: As the second one: the expression `m` is always evaluated.

With respect to the third example, it is currently undefined whether the size expression `m` is evaluated or not, according to the sentence quoted above. However, there is another sentence which says that the type name of a cast is evaluated whenever the cast is evaluated, so we have two conflicting rules for that type name. Common sense dictates that if a sentence says that it is unspecified whether an expression is evaluated or not and another sentence says that it is evaluated, then the expression has to be evaluated. Apart from this, it doesn't make much sense that the array length might not be evaluated there but has to be evaluated in

```
n = ((int(*)[m])A)[0][0];
```

where the value of `m` is equally irrelevant.

Here we have another instance of an ambiguous specification that will be made precise with the introduction of the terms from this proposal and its application to the relevant operators.

Constant expressions in initializers

The text in **6.7.11 Initialization** includes the following constraint:

All the expressions 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.*

This wording is bogus: In `static int i= 2 || 1/0;` the expression `1/0` is not a constant expression. We propose to change *expressions* to *expression initializers*, which is what the text intends. Writing simply *initializer* is not possible, since an initializer may be a brace-enclosed list.

Proposal I. Terminology and bug fix

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. These expressions are called *value-discarded*. Value-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 equally be applied to the type named by a type name.[1)] Many types are fully resolved during translation. Other types are only partly resolved during translation, what remains for the full resolution being the values of certain expressions on which the type depends. For these types, every time the type name or expression is reached during program execution, and if it is necessary to fully resolve the type, those expressions on which the type depends are evaluated, or the relevant values retrieved from some previously evaluated expressions. Because the value of those expressions may change every time the type name or expression is reached, the resolved type may be different on each occasion (these are the variably-modified types). If it is not necessary to fully resolve the type, those evaluations are not performed and it is said that the type or type name is not *runtime resolved*. When this term is applied to a type or type name that is fully 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, specifier or parameter declaration). In addition, there are cases where a type name which is the operand of a unary operator is not runtime resolved;[2)] in these cases, expressions contained within the type name are discarded relative to the unary expression.

2 If an expression `B` is discarded relative to some syntactic construct `A`, it is also is discarded relative to every construct containing `A`, and any subexpression contained in `B` is discarded relative to `A`.

3 An expression which is discarded relative to some construct is value-discarded, except array length expressions which are integer constant expressions, as well as its evaluated subexpressions.[3)] Type names within a value-discarded expression and the type of the latter are not runtime resolved.

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:

[...]

[1)] Thus, for a type name, it may be said that "the type name is resolved" or that "its type is resolved", indifferently.
[2)] These operators are **`alignof`**, **`sizeof`** and **`_Countof`**.
[3)] Integer constant expressions as array lengths are always evaluated (6.7.7.3).

— 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`, `_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 operand is an expression, then: if it has a known constant size, it is discarded relative to the **`sizeof`** expression; otherwise, it is evaluated. If the operand is a type name, it is fully resolved unless it is a variably modified type of known constant size. In either case, if the type has known constant size, the **`sizeof`** expression is an integer constant expression.

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

5 The **`_Countof`** operator yields the array length (number of elements) of its operand. The array length is determined from the type of the operand. The result is an integer. If the operand is an expression and the array length is variable, the operand is evaluated; otherwise, the operand is discarded relative to the **`_Countof`** expression. If the operand is a type name, the array length is fully resolved; the rest of the type is not runtime resolved. In either case, if the array length is fixed, the **`_Countof`** expression is an integer constant expression.

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
                         // retrieving 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.[4)]

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 relative to the expression.[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, and if the latter is a variably modified type, the operand is evaluated; otherwise, the operand is discarded relative to the typeof specification.[6)] If they are applied to a type name, they designate the same type as the type name with any nested typeof specifier resolved.[7)]

6.7.6 Alignment specifier

7 The first form is equivalent to **`alignas`**`(`**`alignof`**`(`*`type-name`*`))`. In particular, the type name is not runtime resolved and the expressions within it are discarded relative to the alignment specifier.

6.7.7.3 Array declarators

5 If the array length expression is an integer constant expression, this length is resolved during translation; hence, the expression is always evaluated, during translation.

6 If the array length expression is not an integer constant expression: if it occurs in a declaration at function prototype scope, itthe array length expression is 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 length 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 an array length expression is part of the operand of a typeof or **`sizeof`** operator and changing the value of the array length expression would not affect the result of the operator, it is unspecified whether or not the array length expression is evaluated. Where an array length expression is part of the operand of a **`_Countof`** operator and changing the value of the array length expression would not affect the result of the operator, the array length expression is not evaluated. Where an array length expression is part of the operand of an **`alignof`** operator, that expression is not evaluated.

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

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

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

[7)] 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.

6.7.11 Initialization

8 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.

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`** 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 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.

Addendum to proposal I

Insert the text in blue in between the text in black:

2 An expression which is discarded relative to some construct is value-discarded, except array length expressions which are integer constant expressions, as well as its evaluated subexpressions.[xx)] An implementation may consider relatively discarded, hence 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. Type names within a value-discarded expression and the type of the latter are not runtime resolved.

The introduction 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]
```

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 be 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 discarded pieces in constant expressions

6.6 Constant expressions

6.6.1 General

8 An *integer constant expression*[9] 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** and **_Countof** expressions which 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, floating literals, named or compound literal constants of arithmetic type and integer constant 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, **_Countof** operator or **alignof** operator.

(Remove the last footnote and add the following example)

[9] 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.

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.

“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`**.