N3382 - Discarded - II

Author: Javier A. Múgica

Versions

Initial version: 2024 - oct - 06 N3351
Second version: 2024 - oct - 22

Changes from the previous version **2**

Introduction **2**

Examples **2**
Constant expressions . . . 2
Out-of-bounds access . . . 3
Identifiers without definition . . . 4

Analysis **4**
Value- and wholly discarded code . . . 4
Operands and statements . . . 4
Not executed, not evaluated and discarded . . . 4
Array sizes in type names . . . 5
alignas and [*] . . . 8
Chicken and egg problem . . . 8
Cast operators . . . 9

Splitting of this proposal into parts **10**

Possibilities opened by the "discarded" concept **11**

Proposal I. Terminology **12**
5.2.2.4 Program semantics . . . 12
6.5 Expressions . . . 12
6.7 Declarations . . . 13

Proposal II. Constraint on constant expressions **14**
6.6 Constant expressions . . . 14

Proposal III. Removal of superfluous sentences **14**
Proposal III A . . . 14
Proposal III B . . . 14

Proposal IV. Allowing certain constructs in discarded expressions **15**
Integer and arithmetic constant expressions . . . 15
6.6 Constant expressions . . . 15
6.7.11 Initialization . . . 15
Identifiers missing an external definition . . . 15
6.9 External definitions . . . 15

Statements **16**

Changes from the previous version

This version has been simplified. We removed the discarded statements. With it it goes away the term *regular label* as well as that of *isolated*, applied to a statement. We also removed the proposal to introduce the term *switch label*.

Another important change is that the previous version delegated the identification of subexpressions that cannot be value-discarded within a value-discarded expression to the rest of the standard, by saying that they are the ones that need to be evaluated (at translation time). Now those subexpressions have been identified as integer constant expressions within type names.

A minor change is the application of the term value-discarded to the size expression that can be replaced by $*$.

Introduction

Not being evaluated is a translation- and run-time property which can in many cases be determined at translation time, as for instance the argument to **`alignof`**. Some constructions are allowed in expressions which are not evaluated which are not allowed when the expression is evaluated. If any of these appears in a context that will be evaluated one would like the translator to issue a diagnostic. But that is only possible if it can be known at translation time that the expression will be evaluated. This is in general impossible (it may depend on user input, for example), but to know that certain expressions will *not* be evaluated is not only possible but often immediate, as the argument to **`alignof`** mentioned above.

In these, lets call them translation-time-known unevaluated expressions, there can be latitude as to what they contain. In the extreme, it could be required from them just to be syntactically correct. But to take advantage of this possibility it is first necessary to coin a term and apply it to those expressions. This is the purpose of the present proposal.

This document benefited from suggestions by Jens Gustedt.

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.

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;
}
```

Is here `a++ ? 0 : 0` a constant expression?

According to the letter of the standard it can be, for the expression is not evaluated (hence `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 it seems perfectly valid to consider `a++ ? 0 : 0` a constant expression and replace it by 0, for no increment is being lost since the whole expression is not being evaluated.

But consider the following variation of the example:

```
if(test){
      /* ... */
}else{
      int A[a++ ? 1 : 1];
}
```

and suppose that *test* cannot be decided at translation time. Then `a++ ? 1 : 1` might or might not be valid as constant expression depending on a runtime condition. But being or not a constant expression must be known at translation time!

We can do worse:

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

If `b` happens to be $\neq 0$ at runtime then `b ? 1 : (a++, 1)` could have been taken as a constant expression at translation time, for it can be computed to be 1, independent of the value of `b` and `a`, (if the translator is smart enough) and satisfies the constraint, since neither the increment nor the comma operator are evaluated, even though that could not have been known at translation time. Suppose now that the implementation does not support VLA. Then it may consider `A` to be a fixed length array of length one. If, at runtime, `b` happens to be 0, that choice had not been right. In an implementation not supporting VLA, and expression which is not constant for specifying the length of the array causes undefined behaviour, so treating it as fixed-length array is one valid option. However, the implementation should have produced a diagnostic when translating.

We don't think the wording of this constraint poses any problem in practice, since we expect implementations to consider code like `a++ ? 1 : 1` as not being a constant expression, even if they know it will not be evaluated. But it shows the inappropriateness of having a constraint depend on a runtime feature.

The problem is solved by replacing *not evaluated* with *value-discarded* (the therm we chose for known-at-translation not evaluated) in the constraint, probably in the spirit of the authors of that constraint.

Out-of-bounds access

Consider the code

```
int a[3], b;
b = 8<3 ? a[8] : 0;
```

While nobody would write code like this, it may arise as a result of macro substitution:

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

We would like to make an access to a fixed length array where the index is an integer constant expression exceeding the length of the array a constraint violation. That would break macros like this one. If that constraint were present, using `SAFE_ACCESS` would be 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 as already argued. However, "which is not value-discarded" is possible and would

allow those accesses in discarded expressions, which is were they can appear as a result of macro expansions. This is what motivated this proposal.

Identifiers without definition

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`** 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.

The whole list can be replaced by the term *value-discarded*.

Analysis

Value- and wholly discarded code

Our first approach to "discarded" code was as follows:

Some code is discarded because it is the operand of an operator which just needs to check the operand's type. These are the controlling expression of a generic selection and the operands of **`sizeof`**, **`alignof`** and typeof operators, when they are not evaluated. We will call these ones *value-discarded*. Some other code is discarded because it is skipped by the program flow. Any such code we will call *wholly discarded*.

However, we switched our minds. The translator always needs to check the type of expressions.

Operands and statements

There are two different kinds of (possibly) value-discarded contexts:

1. `||` `&&` `?:` **`_Generic`** **`sizeof`** **`alignof`** **`typeof`**
2. Secondary statements of **`if`**, **`switch`**, **`for`**, **`while`**, and *expression 3* of **`for`**

We will call the first ones *operator discarded*. These are the only ones handled by this proposal.

Not executed, not evaluated and discarded

Take the text we intended at some time for the switch statement, with the original term *wholly discarded*:

*If no converted **`case`** constant expression matches and there is no **`default`** label, no part of the switch body is executed; in this case, if the controlling expression is an integer constant expression and the switch body is isolated, the switch body with the exception of the expressions in each **`case`** label is wholly discarded.*

The immediate reaction to this text might be: "Why not the whole switch statement?" Because the controlling expression and the case expressions had to be evaluated, during translation. Discarded expressions can only be expressions whose value is not needed, even at translation. Only if this

is fulfilled can these expressions be allowed to include constructions not allowed in non-discarded expressions, as an out-of-bounds access.

The last comment in the previous paragraph makes clear what discarded should mean. We have therefore

to evaluate: To perform the actions expressed by an expression in order to determine its value and produce side effects. Exceptionally, the evaluation of the typeof operators yields a type.[1)] Evaluation can take place at translation time or at runtime. In the first case, this can only be for expressions without side effects (or side effects are discarded).

not executed: This is clear. *A fortiori,* the values of those expressions are not needed at runtime.

value-discarded: The value of the expression is not needed, either at runtime or during translation, and this can be determined at translation.

When the standard says of some expression that it is not evaluated, its value is not needed. However, the value of some subexpressions might be needed and these have to be evaluated, at translation time. For example, the subscripts in

```
sizeof(int[3]);
sizeof(int[(4+7)/5]);
sizeof(int[(int)3.999999999999999]);
```

This translates to the concept of value-discarded: some subexpressions of a value-discarded expression must be considered not value-discarded. The first version of this proposal identified the latter as those which "need to be evaluated at translation time". A consideration on which these expressions are concluded that they are the ones within type names: The type of a value-discarded expression is usually needed. In order to determine it, the value of its subexpressions are not needed, only their types, unless these values are part of a type name. We may say, therefore, that "expressions within a type name are not value-discarded". But we define value-discarded when applied to any syntactic construct to mean that the expressions therein are value-discarded. Hence

Type names are never value-discarded

Actually, there are exceptions to the previous rule. So much, that in the end we dropped this rule. This will be analysed in the next section.

Array sizes in type names

On nested `sizeof`

We may say that the operand of a **`sizeof`** operator which is part of another **`sizeof`** operator is value-discarded, even type names within it. For example, in

```
sizeof(sizeof(int[n]))
```

But this will be wrong for an expression like **`sizeof`**`(int[`**`sizeof`**`(int[n])])`.

The precise condition that the **`sizeof`** operator must satisfy is not to be nested, but to be value-discarded. In the latter example, the inner **`sizeof`** is not value-discarded as being part of a type name. For **`alignof`** that condition is not necessary since the value of expressions within type names inside the operand of **`alignof`** do not alter the result of the operator.

The previous rule is wrong if a type within the value-discarded **`sizeof`** can make an expression valid or invalid. We can think of the following example

[1)] If they are considered operands which are evaluated, which can be contentious. Probably that cannot be in a formal sense as the term stands now. However, this interpretation sneaks in naturally. Cf. the following sentence in the semantics of the typeof specifiers: "Otherwise, they designate the same type as the type name with any nested typeof specifier evaluated.", as well as the text of the footnote attached to that sentence. Or the following paragraph in the description of cast operators: "Size expressions and typeof operators contained in a type name used with a cast operator are evaluated whenever the cast expression is evaluated." Either this sentences are rephrased or the definition of *Evaluation* in "Program semantics" should be adjusted.

```
int f(int(*)[3]);
/* ... */
sizeof(sizeof(  f((int(*)[5])NULL)  ))
```

But we still want the expression `E` within `sizeof(sizeof(int[E]))` to be value-discarded. A consideration of these and other examples led us to the rule:

*If a **`sizeof`** operator is value-discarded its operand is value-discarded, including type names, with exception of integer constant expressions within the type name of a cast operator.*

Unfortunately, this is not possible. Whenever `int[5]` appears in the code the compiler needs to know the value within brackets. So the rule may be, for the time being:

*If a **`sizeof`** or **`alignof`** operator is value-discarded its operand is value-discarded, including expressions within type names which are not integer constant expressions.*

Integer constant expressions

Are size expressions which are integer constant expressions ever not evaluated? No.

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

We tested the compilers clang, gcc, msvc, CompCert, icx and nvc at godbolt, and the first three of them offline. msvc and ComCert do not implement variable length arrays and cannot translate the last of the above instructions. We place an asterisk in the last line because clang and icx do not produce a diagnostic and compile seamlessly. An analysis of the assembly code generated reveals that they put the literal -1 in it (i.e., 4294967295), and afterwards carry out the multiplication. Both compilers do raise an error if `n` is replaced by `5`.

Curiously, the behaviour upon

```
sizeof(int[1/0][n]);
sizeof(int[n+1/0]);
```

is reversed. clang and icx emit a warning, and take whatever value happens to be in the register where the result of `1/0` would be placed, while gcc compiles without warning, places the division in the generated code and the execution in the author's computer produces the message "Floating point exception" (?). However, if `1/0` is replaced by `9-2` (say) in gcc, it places a literal `44` in the generated code. So, all three compilers evaluate the expression at translation time. clang and icx decide to omit the operation, while gcc places it in the emitted code as if it were not an ICE (which it is not).

Finally, we compared the behaviour of compilers upon the two instructions

```
sizeof(sizeof(int[2-3]));
sizeof(sizeof(int[1/0]));
```

As we noted, all compilers report an error for the first one. All accept the second one with the exception of CompCert and msvc. The error reported by the latter is "cannot allocate an array of constant size 0", while CompCert complains that the size of the array is not a compile time

constant. Since these two compilers do not implement variable length arrays, they naturally take any size expression in brackets as a mandatory ICE, and report an error for `1/0`. The other compilers seemingly take `1/0` as not an ICE, do not evaluate it and do not resolve the type name **`int`**`[1/0]`, which is allowed by the specification, that implies that the inner **`sizeof`** is not evaluated.

Not integer constant expressions

Array sizes 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 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.*

Here, the last sentence could be changed to *is unspecified whether the size expression is evaluated or is value-discarded*. We did not because discarded expressions shall be ones which are not evaluated, definitely. But, as noted, we want the size expression (if not an ICE) in **`sizeof`**`(`**`sizeof`**`(`**`int`**`[n]))` not to be evaluated. To this end no wording in the lines of the previous sentence is needed. The rules we set up make the type name **`int`**`[n]` there value-discarded and this implies that no non- integer constant expression in it is evaluated.

It is difficult to think of an example that would need that sentence after the concept of value-discarded is introduced and applied as in this proposal. The argument to **`sizeof`** must have, or be the name of, a variable length array type, for otherwise it would be value-discarded. Then, there must be within it a size expression which does not affect the result. But not inside a nested **`sizeof`** or typeof operator, for then the rule would apply to that one. Hence, it must be the type name of a cast. We could think of the following example:

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

Here, it is undefined whether the size expression `m` is evaluated or not, according to that sentence. 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 contradictory 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. Appart from this, we prefer it to be evaluated unconditionally because we do not see why this might not be evaluated there but has to be evaluated in

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

where the value of `m` is equally irrelevant.

Since we cannot think of an example to make sense of the referred to sentence, after the concept of value-discarded is applied, we propose its withdrawal.

Conclusion

We don't like to say that a type name is not value-discarded since that complicates the wording of the **`sizeof`** operator and other places in the standard.

Also, there is no need to restrict the possibility of a size expression which is not an ICE to be value-discarded only to operators that accept operands of any type, viz. **`sizeof`**, **`alignof`** and the typeof and **`alignas`** "operators". Changing the size of a variable length array type cannot change the type of an operand needed only for its type in a way that it is no longer valid as an operand to the operator in question (e.g., +) or changes the value of that operator's expression. The only operator that could cause problems is the conditional operator, where a composite type has to be determined. For this we already have

*If one operand is a pointer to a variably modified type and the other operand is a null pointer constant or has type **`nullptr_t`**, the behavior is undefined if the type depends on an array size expression that is not evaluated.*

So, the rule for the evaluation of expressions inside type names will be either of:

If a type name is value-discarded, expressions within it which are not integer constant expressions are value-discarded; expressions not contained in such expressions are not value-discarded and are evaluated during translation.

If a type name is value-discarded, the integer constant expressions it contains are evaluated during translation. Expressions within it which are not integer constant expressions are value-discarded.

The difference lies in expressions like `alignof(int[n+3/2])`. Since the expression `n+3/2` is going to be value-discarded it seems senseless to require the evaluation of `3/2`. Therefore we keep the first formulation, even if more verbose. But in `sizeof(int[1 ? 2 : 1/0])` the subexpression `1/0` is value discarded, so a different wording is needed:

If a type name is value-discarded, expressions within it which are not integer constant expressions are value-discarded; integer constant expressions not contained in any other expression within the type name are not value-discarded and are evaluated during translation.

alignas and `[*]`

We have not added any text to the subclause on the **alignas** specifier. It would be needed, if needed at all, for the production **alignas**(*type-name*). We believe it is not needed because of the wording

The first form is equivalent to ***alignas****(****alignof****(type-name)).*

The expression within `[]` in a function parameter declaration, if not an integer constant expression, is treated as if it were replaced by `*`, so the standard says. But

```
int g(float *p){return 2;}
int i;
void f(int a[g(&i)]);
```

produces a diagnostic because of the unmatching types. Therefore, the replacement by `*` is not to be taken literally. For this reason we believe it is convenient to state that the expression is value-discarded.

Chicken and egg problem

The fourth part of this proposal intends to allow, in integer constant expressions, essentially anything in its operands that are discarded:

An integer constant expression shall have integer type and shall only have operands that are value-discarded, integer constants ...

Thus, `2 || a+f()` would be an integer constant expression.

This wording does not reflect the intent behind it if the expression appears in a context where all is discarded, as in the following two snippets:

```
sizeof(int[n]);
2 || b + c;
```

In the OR expression the subexpression `b+c` satisfies the requirement above for ICE, since the operands `b` and `c` are discarded, as part of a discarded expression. In the **sizeof** operand, *either*

`n` is not an ICE → **int**`[n]` names a VLA → **int**`[n]` is evaluated → `n` is not an ICE, *or*
`n` is an ICE → **int**`[n]` is not the name of VLA → **int**`[n]` is not evaluated → `n` is an ICE

While we don't expect an implementation to take advantage of the possibility of the second interpretation, we'd rather have a wording which is right. The intent of allowing, in ICE, operands which are value-discarded, is to allow operands that would be value-discarded even if the whole expression were evaluated. Thus, the concept of *relatively discarded* may be introduced, where one subexpression would be said to be relatively discarded with respect to another expression `E` containing it, if the said subexpression would be value-discarded even if `E` were evaluated.

Three possibilities were explored to solve the the-whole-expression-discarded problem:

1. Define the concept of relatively discarded based on the evaluation of `E`.
2. Insert extra wording either in ICE or in **`sizeof`**.
3. Define carefully what discarded operands can be part of an ICE, in a way different from 1.

The first option introduces the concept "if the expression were evaluated" which can be problematic. Consider the following example:

```
constexpr int A[3] = {0,0,0};
constexpr n = 8;
if(n < 3){
      A[n] ? 2 : 1;
}
```

Here, reasoning about "if `A[n] ? 2 : 1` were evaluated" is pointless, since if that were evaluated the behaviour would not be defined.

The rationale of the second solution is that the **`sizeof`** operand is the only case likely to be contentious in practice. But we don't want to leave a half-baked solution. As seen in the first version of this paper, that leaves unfixed situations.

The third solution seeks to identify what operands would be discarded "even if the expression were evaluated", avoiding this expression. In the cases of the `&&`, `||` and `?:` operands, the other operand(s) would be evaluated. In the cases of the **`sizeof`** operand and friends, the unary operator is evaluated. Thus, we may write:

A value-discarded operand is essentially discarded if it is an operand of an operator which is evaluated.

Then, an integer constant expression would be allowed to contain essentially discarded operands.

A little thinking reveals that this definition does not solve the problem; it reverses it. If the whole ICE is value-discarded, and this is when the problem arises, none of its operands would satisfy the definition of an essentially discarded operand, and it turns out that what is allowed in ICE which are evaluated, as of value-discarded operands, would not be allowed in ICE which are not evaluated.

Thinking upon essentially discarded operands or expressions, about which ones we want them to be, it is realized that those are the ones which the different subclauses deem as value-discarded, not their subexpressions. This is the final approach we took for defining that concept, listing all the cases for the sake of unambiguity. Then we introduced the term *discarded relative to* (an expression), in order to make the wording of ICE cleaner, as:

An essentially discarded operand of an expression `E` and its value-discarded sub-operands are discarded relative to *the expression `E` and to any expression of which `E` is a subexpression.*

The reference to "value-discarded" in "its value-discarded sub-operands" is to take account of not value-discarded subexpressions of value-discarded expressions. I.e., `E` ⊃ `F` ⊃ `G`, where `F` is essentially discarded but `G` is not value-discarded. If furthermore `G` ⊃ `H` and `H` is value-discarded, it is discarded relative to `E`.

Then we say that an integer constant expression can contain operands *that are discarded relative to it, ...*

In the final wording we changed *operand* in the definition above to *expression or type name*.

Cast operators

As we saw, integer constant expressions as size expressions within `[]` are always evaluated. Now, the description of cast operators reads:

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

But the opposite is not true; i.e., the cast expression may not be evaluated and the type name be, as in

```
int a[1];
sizeof(  *(char(*)[5])(a-1)  );
sizeof((typeof(signed char))UCHAR_MAX);
```

The first **sizeof** expression evaluates to 5, which means that the size expression is being evaluated (we can have any ICE which evaluates to 5 in place of a literal 5), while the cast expression is not evaluated. The second **sizeof** expression evaluates to 1, not to **sizeof**(**int**), but the cast expression itself is not evaluated.

This can probably be made clearer by simply dropping the sentence. If we understand "to evaluate" to mean "to evaluate expressions therein which are not constant", as seems the case for **sizeof**, etc. when their operand is a type name (though one does not see why that should be the meaning), then the sentence adds nothing, for these expressions are evaluated at run time, an so need to be evaluated every time the cast expression is evaluated. After the introduction of the wording in this proposal to precisely specify when expressions within type names are evaluated, the sentence is completely redundant if right, misleading if wrong or imprecise.

We also think that it should be defined what *to evaluate* means when applied to a type name. The current text in 5.2.2.4 applies only to expressions. Or, better, to use a different term for the determination of a type, expressed by a type name, by the type name implicit in a declaration or the type of an expression. We think that *to resolve* can be a good term for that. We do not include this in the present proposal.

Splitting of this proposal into parts

Proposal I

This proposal is divided into three parts. The first one introduces the term *value-discarded* and other terms, and apply it to what we called operator discarded expressions. This changes nothing in the C language. These are the terms introduced:

value-discarded *essentially discarded* *discarded relative to*

Proposal II

The second part applies the term *value-discarded* to the first constraint in "constant expressions". This is intended as a fix and should have not impact in existing code.

Proposal III

This is the removal of two sentences that now seem superfluous (the one on the evaluation of the type name of a cast operator) or devoid of meaning (the unspecified evaluation of some expressions in **sizeof** and typeof operators).

Proposal IV

The fourth part applies the concept of value-discarded in several places to integer and arithmetic constant expressions and to the use of file-scope identifiers for which there is no definition. Other recent papers propose changes to the text on those types of constant expressions. In the event that this third part be adopted, the wording would have to be coordinated with those papers' wordings.

This part of the proposal makes certain expressions integer constant expressions or arithmetic constant expressions which previously were not, of the kind

```
2 || a + f()
```

which can only arise from macro expansions. We don't expect this change to have any impact on existing code.

It also allows the use of an identifier declared at file scope for which there is no definition also in the discarded operand of the `&&`, `||` and `?:` operators. This is an extension which can have no impact on existing code.

Other

Contrary to the original version, we do not propose here the extension of the concept of *value-discarded* to statements.

Some changes in the text for integer constant expressions have been moved to another proposal (already in the first version), and would have to be incorporated to this one in case they were accepted.

We do not include any proposal to define the term *to evaluate* when referred to a type name, or to introduce a new one. We hinted at *to resolve*.

Possibilities opened by the "discarded" concept

We can see two benefits derived from the introduced terminology. First, constraints can be applied to certain constructions including the proviso that they only apply to expressions which are not value-discarded. For example, with respect to array subscripting when the subscript is an integer constant expression: *If the expression is not value-discarded the subscript shall be less than the length of the array or... ,* which is the original driving force behind this paper. Or to the use of certain identifiers for which there is no definition, as noted, and for which there is already a list of exceptions to the constraint, which provides a first example of simplification of the wording thanks to the introduction of the term *value-discarded*.

The second one is that it opens up the door for an overall relaxation of the requirements for value-discarded secondary blocks in selection and iteration statements, that could become wholly discarded, its contents being checked just for syntax (and the presence of labels), thereby allowing to use the compiler proper in place of the macro language in some places. Instead of:

```
#if condition
      /* some code */
#else
      /* some other code */
#endif
```

write

```
if(condition){ //ICE
      /* some code */
}else{
      /* some other code */
}
```

We don't expect this to be proposed any soon, if ever. However, constructions permitted in expressions which are value-discarded and not elsewhere may also be allowed there. For example, identifiers with file scope for which there is no definition.

Proposal I. Terminology

5.2.2.4 Program semantics

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.

6.5 Expressions

6.5.1 General

Insert the following after paragraph 1, or somewhere else.

2 This document signals some expressions as value-discarded. An implementation may consider value-discarded other expressions for which it can determine during translation that they will never be evaluated, beyond those identified by this document.

3 Type names and typeof specifiers contained in a value-discarded expression are value-discarded.

4 A value-discarded expression or type name is *essentially discarded* if it is:

— the right operand of a && or || operator whose left operand is an integer constant expression with value 0 or unequal to 0 respectively;

— the second or third operand of a conditional operator whose first operand is an integer constant expression with value 0 or unequal to 0 respectively;

— the operand of a **`sizeof`** operator and has a type, or is a type name, which is not a variable length array type;

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

— the operand of a typeof operator and has a type, or is a type name, which is not a variably modified type;

— the controlling expression of a generic selection; or

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

NOTE: These are the constructs explicitly signalled as value-discarded in the following subclauses, but not their subexpressions.

5 An essentially discarded expression or type name `F` contained in an expression `E` and the value-discarded constructs contained in `F` are *discarded relative to* the expression `E`.

6.5.2 Primary expressions

6.5.2.1 Generic selection

Semantics

3 The generic controlling operand is not evaluatedvalue-discarded. If a generic selection has a generic association with a type name that is compatible with the type of 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 value-discarded.

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 compound literal is value-discarded.

6.5.4 Unary operators

6.5.4.5 The `sizeof` 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 **`sizeof`** expression is not value-discarded and the type of the operand is a variable length array type, the operand is evaluated; otherwise, the operand is not evaluatedvalue-discarded and the result is an integer constant expression.

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

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 second operand is value-discarded.

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 second operand is value-discarded.

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

6.7 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.[3)] Otherwise, they designate the same type as the type name with any nested typeof specifier evaluated.[4)] If the typeof specifier is not value-discarded and the type of the operand is a variably modified type, the operand is evaluated; otherwise, the operand is not evaluatedvalue-discarded.

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

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

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

6.7.7.3 Array declarators

5 If the size is an expression that is not an integer constant expression: if it occurs in a declaration at function prototype scope, it is value-discarded and is treated as if it were replaced by `*`; otherwise, each time it is evaluated 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, that operand is not value-discarded 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 an **`alignof`** operator, that expression is ~~not evaluated~~value-discarded.

6.7.7.8 Type names

4 If a type name is value-discarded, expressions within it which are not integer constant expressions are value-discarded. Integer constant expressions not contained in any other expression within the type name are not value-discarded and are evaluated during translation, always, even if the type name is value-discarded. If a type name is not value-discarded, the expressions it contains which are not integer constant expressions are evaluated at runtime when the type name is reached.

5 EXAMPLE In the following expression

```
sizeof(int[1 ? 2 : 1/0]);
```

the operand **`int`**`[1 ? 2 : 4-3]` is value-discarded, the expression `1 ? 2 : 4-3` within it is not value-discarded, and the subexpression `1/0` is.

Proposal II. Constraint on constant expressions

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 evaluated~~discarded relative to the expression.[5)]

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

Proposal III. Removal of superfluous sentences

Proposal III A

Remove the following paragraph in cast operators:

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

Proposal III B

Remove the following sentence in array declarators:

~~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.~~

Proposal IV. 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 value-discarded 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 value-discarded 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.

6.7.11 Initialization

5 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, string literals or value-discarded.

This only allows value-discarded expressions which are part of a constant expression, since the initializer itself is never value-discarded. Hence, "with respect to the initializer" is not needed.

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;

[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.

— 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.

Statements

Unlike in our original version, we do not propose here the extension of the concept value-discarded to apply to statements. However, because wording for it was proposed in the first version, we present here a substantial change to it that would have to be applied if it were proposed. The section “Isolated statements” only apply to discarded statements, so we removed it from this paper. It can be read in the original proposal.

Statements and blocks

6.7.1 General

6 If a declarator is value-discarded, expressions within it are treated as in a value-discarded type name.

7 If an instruction statement is value-discarded its instruction is value-discarded. If a labeled statement is value-discarded, the statement following the label is value-discarded. The term value-discarded applied to a label, a **`static_assert`** declaration or a standard attribute has no effect. The effect on an implementation-specific attribute is implementation-defined.

8 If a declaration is value-discarded its attributes, declarators and initializers are value-discarded, except the initializers of named constants.

9 If a selection or iteration statement is value-discarded its controlling expression and its secondary blocks are value-discarded. In a **`for`** statement, in addition, *clause-1* and *expression-3* are value-discarded.

10 If a compound statement is value-discarded, the block items of which it is composed are value-discarded.