N3351 - Discarded

Author: Javier A. Múgica

Versions

Initial version: 2024 - oct - 06

Contents

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.

Splitting of this proposal into parts

This proposal is divided into three parts. The first one introduces the term *value-discarded* and other terms, and apply it to certain expressions known not to be evaluated. This changes nothing in the C language. These are the terms introduced:

value-discarded *essentially discarded* *discarded relative to* *isolated* *regular label*

At the end It includes a subproposal to introduce, in addition, the term *switch label*. In the opposite direction, the term *regular label* could be omitted.

The second part applies the term *value-discarded* to the first constraint in “constant expressions”.

The third 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.

Discussion

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.

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*.

At present there is no difference between one an the other. But this distinction opens up the possibility of permitting more latitude for wholly discarded expressions. Consider for example the following piece of code:

```
#define compute_all(x) \
_Generic((x), \
      double: compute_one(x), \
      double*: compute_several(x, sizeof(x)/sizeof(double)) \
)
```

where the functions operate on a **double** value or on an array of **double**. (Here we would have written **double**[ ] in place of **double***, but the former is not (yet) standard.)

This code will not compile because the argument passed to one of the two functions does not match the declared parameter: **double** vs. **double**[ ] or vice versa. If wholly discarded code where required to just be syntactically correct, this code would become valid.

Irrespective of the previous example we believe that establishing a distinction between value-discarded and wholly discarded code is convenient.

However, we switched our minds. The reasons for this will be explained in the section «Not executed, not evaluated and discarded».

Compound statements

The "then" or "else" part of an **if** statement can in some cases be discarded if the controlling expression is an ICE, as well as the secondary block of other flow control constructions. It cannot be discarded if the program can jump into the block from outside. For referring to blocks that can be discarded we coined the term *isolated*. Our first definition of an isolated statement was:

*It does not include **setjmp**, no goto instructions jumps into it and, if it contains any **case** or **default** label, it contains the whole switch body to which it is associated.*

The reference to **setjmp** can be removed. That function can serve as entry point to the block only if it first executed, so cannot be by itself a means of entering the block.

As for the goto instructions, if the block contains some label, the definition above requires the translator to inspect all the function to see if the label is referenced by any goto instruction. This, in turn, may be required to decide whether to issue or not a diagnostic depending on whether the code is considered discarded or not. We prefer that the constraints associated to a piece of code be decidable with the information the compiler has so far and the piece of code itself, so we strengthened the condition of no goto jumping to the label to no labels being present, not without hesitation.

It should be kept in mind the goal of qualifying some instructions as value-discarded. This is to allow there some constructions which would not be allowed elsewhere, such as a translation-time-known out-of-bounds access or, eventually, function calls not matching the function prototype. These can only appear as a result of macro expansions. There is no intent in permitting the programmer to directly write invalid code. Therefore, it is relevant that simple constructions that can arise because of macro expansions be deemed value-discarded if they are known not to be evaluated, but it is less so for larger blocks with labeled statements. Furthermore, since discarded instructions are (or will be) allowed to contain constructions which are not allowed in no discarded instructions, expanding the range of instructions which is considered to be discarded is always possible, while restricting it in the future can make code invalid which was previously valid. For this reason we preferred a criterion easy to follow by a compiler than a finer, more complex one. This rationale prevented as from qualifying as discarded the instructions preceding the first label in a compound statement (when

that compound statement would have been deemed discarded were not for the labels it contains) and outside any iteration statement included in the compound statement and containing the label.

Finally, we must require also that the statement itself is not labeled. So our final wording for isolated statements is:

A statement is isolated *if it is not labeled, does not contain regular labels and, if it contains any* **`case`** *or* **`default`** *label, it contains the whole switch body to which it is associated.*

One may argue, then, that if the property of being value-discarded is only relevant for small pieces of code, compound statements may never be considered value-discarded. This is outright wrong, for it would make the semantics of the following two lines different:

```
if(8<3) a[8];
if(8<3){a[8];}
```

Similarly, considering value-discardable only instruction statements and compound statements with just one instruction within does not solve this problem, for then the semantics would change if a single instruction inside a compound statement were split in several ones.

As to the requirement that "if it contains any **`case`** or **`default`** label, it contains the whole switch body to which it is associated": The natural condition is to mandate it contain the whole switch statement, not just the switch body. We chose the latter so that the switch body itself satisfies the definition. This is necessary when we want to mandate that if the controlling expression of a switch statement is an integer constant expression and no **`case`** label matches it, there is no **`default`** label and the switch body is isolated, the later is value-discarded.

Regular labels

The definition of *isolated* includes the term *regular label*. We found ourselves referring to "label other than **`case`** or **`default`** labels", or similarly. This is cumbersome, so we introduced the said term to refer to those labels.

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 expression without side effects (or side effects are discarded), and the evaluation of the typeof operands happens necessarily during translation.

[1)] If they are considered operands which are evaluated, which can be contentious. Probably that cannot be in a formal specification of the language. However, this interpretation sneaks in naturally. See the second footnote in the semantics of the typeof specifiers: "If the typeof specifier argument is itself a typeof specifier, the operand will be evaluated before evaluating the current typeof operator. [...] ".

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

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 indices in

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

Therefore, it is probably not completely right to say that the operand to **`sizeof`**, etc. is not evaluated.

It is not the intent of this paper to make the uses of "not evaluated" more precise. This discussion is just to show that the concept of *discarded* cannot be accommodated by *not evaluated*, much less by *not executed.*

The withdrawal of the term wholly discarded

Our first version of this paper contained the following assessment:

There is at present no construct that is permitted in wholly discarded code which is not permitted in value-discarded one. This notwithstanding, we prefer to keep the two terms. Wholly discarded code is not executed because of a logical branching (an ICE evaluating to 0 or $\neq$0), that could be completely discarded. Value-discarded code, on the other side, is needed for its type.

This text was written before we realised two facts:

First, that code that is excluded because of an ICE evaluating to 0 or $\neq$0 in the operators `?:`, `||` and `&&` is also needed for its type. For the conditional operator the type is needed in order to determine the common type for the second and third operands. For the operators `&&` and `||` the type is not needed in the sense of being relevant for the semantics of the program, but is needed in order to see if it is a valid type for the operand of those operators. This left as wholly discarded just the secondary blocks of selection or iteration statements, and the third expression in a for statement, when they are discarded.

Second, and more important, that some subexpressions of discarded expressions must be considered not discarded. This is acceptable for value-discarded expressions, but does not seem so for code deemed wholly discarded. While that code *could* be checked just for syntax, the reality is that it is translated as any other code, including the evaluation of some expressions at translation time.

Therefore, we changed all our "wholly discarded" into "value-discarded" and the term *discarded* disappears.

On not evaluated code

Some uses in the standard of "not evaluated" are not precise, but the uses of "value-discarded" need be precise, for some constructions might be allowed there which are not allowed elsewhere. Therefore, precise wording was inserted to take into account the possibility of value-discarded expressions containing subexpressions which are not value-discarded, which themselves may contain expressions which are value-discarded. This wording provides a model on which to base the wording for "not evaluated", if it be wanted to improve it.

As we said, it is not the purpose of this paper to tweak the uses of that term. However, here is an idea of how it could be done: The proposed paragraph

If an operand in the following subclauses is said to be value-discarded it is also implied that it is not evaluated.

would disappear and instead wording would be inserted for each of the operands. For **`alignof`** it could be The operand is value-discarded and not evaluated, while for **`sizeof`** it would be

Otherwise, the **sizeof** expression is an integer constant expression and the operand is value-discarded; if the **sizeof** expression is not part of an essentially discarded operand, or if being part of one such operand it needs to be evaluated according to the specifications of all those operands of which it is part, it is evaluated during translation and the parts of its operand are evaluated only inasmuch as is needed to determine the type of the operand, except that it is unspecified whether the sizes of variable length arrays are evaluated or not.

V.L.A.s may be part of a **sizeof** expression which is an i.c.e.

Some reference to "if evaluated" applying to the operator, **sizeof**, as a precondition to evaluate part of its operand is necessary, in some or other form, since in

```
typeof(sizeof(/*...*/)) a;
a = 2 || sizeof(/*...*/);
```

neither **sizeof** is evaluated, so nothing in their operands is evaluated. However, “if it is evaluated” by itself is not possible since, again, that can be a runtime condition. For example, in

```
a = sizeof(int[3+2]);
```

the implementation may very well defer to runtime the evaluation of the operator and translate it as `4*(3+2)`, say, in which case the **sizeof** gets evaluated only if the assignment expression of which it is part is executed. The expression “if evaluated, it is evaluated during translation”, which could be thought of as a way of preventing the deferring to run time of its evaluation, forces the compiler to evaluate it at translation time unless it can know for sure that its evaluation will not be need. This would leave some ambivalent cases:

```
if(0){
      a = sizeof(int[2-3]);
}
```

which may trigger a diagnostic or not according as to whether the translator decides to evaluate it during translation or leave it unevaluated. The expression “if the **sizeof** operator is not part of an essentially discarded operand [...] it is evaluated during translation” solves all problems. It mandates evaluation in contexts as the piece of code just shown, conforming to current practice, and as for appearances in operands which are essentially discarded, each of these would carry a wording like the one we have shown for example in **alignof** and **sizeof**, specifying what inside it is evaluated or not.

The text preceding the Otherwise, if... sentence in the standard: “If the type of the operand is a variable length array type, the operand is evaluated”, could have “at runtime” added to it. No wording is needed to say that the operator is evaluated at runtime only if the program flow reaches it, for it is a tautology.

Subclauses with insertions not needed or doubtful

alignas

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))*.

Array declarators

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.*

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. If instead of *undefined* the text said *implementation defined* we would have made the change, just as, e.g., implementation defined integer constant expressions can be used for fixed array sizes.

Chicken and egg problem

The third 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]);
if(0){
      a = b + c;
}
```

In the assignment instruction the expression `b+c` satisfies the requirement above for ICE, since the operands `b` and `c` are discarded, as part of a discarded statement. 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. In the following:

```
int a, b;
if(0){
      int A[a + b];
}
```

`A` would be a fixed length array, if only **`sizeof`** is fixed.

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 operand 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 or expression which are part of an essentially discarded operand which itself is a subexpression of the ICE.

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 will 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, is not 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, ...*

The definition we gave of essentially discarded does not cover expressions in discarded blocks. So we extended it to

A value-discarded expression is *essentially discarded* *if it is [...] or if it is not a subexpression of a vavlue-discarded expression.*

but in the end we removed the extension, for it would make essentially discarded some expressions which are not value-discarded, as in

```
if(0){
      int[E] a;
}
```

where `E` stands for an integer constant expression. This expression has to be evaluated at translation time, hence cannot be value-discarded, yet it is not part of any other expression (discarded or not). And also removed it because at present, in the language, more is allowed in the (other) expressions we call essentially discarded than in expressions which belong to statements that we call value-discarded, of which nothing in particular is said in the standard.

Finally, we considered changing the term of art from *essentially discarded* to *essentially discarded operand* to accommodate there, by definition, the expressions of the rejected associations in a generic selection, which are not operands of an operator. It would be very cumbersome to refer to *essentially discarded operand or expression in a generic association*, where furthermore is not clear that *essentially discarded* applies also to *expression in a generic association*; the alternative is to write *essentially discarded expression or type name*. But we didn't like the use of "operand" in *essentially discarded operand* to include the rejected associations of **`_Generic`**, so we finally stayed with *essentially discarded* and write *essentially discarded expression or type name*.

Identifiers without definitions

The standard contains the following text referring to external declarations:

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:

and a list follows identifying just what we called *essentially discarded* expressions or type names, except that it does not include those of the `&&`, `||` and `?:` operators, since value-discarding them is a new feature of this proposal.

Therefore, there is already a construct allowed in value-discarded code not allowed elsewhere. It is limited to essentially discarded "operators", excluding the discarded statements and instruction (the third of the **`for`**) of selection and iteration statements. In the third part of this proposal we propose to extend the permission to the above listed operators, but not to the code discarded because of statements with controlling ICE, though we'd like to see the use of those identifiers without a definition allowed in those contexts too.

Also, note that the current list of exceptions includes

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

where *whose result is an integer constant expression* is already redundant.

The reason we do not extend the permitted places to code discarded because of statements with controlling ICE is that in this proposal we do not want to change in any form the way these statements are translated. Thus, the proposed wording is *unless it is part of an essentially discarded expression or type name* instead of the simpler *unless it is value-discarded*.

Impact on existing and future code

Existing code

The first and second parts of the proposals should have no impact on existing code. The first one just introduces terminology.

All expressions which are evaluated during translation continue to be evaluated after this proposal

Including those in value-discarded compound statements. The second one takes advantage of that terminology to fix a constraint on constant expressions.

The third 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.

Future code

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,* 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. This provides a first example of simplification of the wording thanks to the introduction of the terms *value-discarded* and *essentially discarded*.

The second one is that it opens up the door for an overall relaxation of the requirements for value-discarded code, and even more for wholly discarded code, if discarded secondary blocks

where to become so, 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 */
}
```

The ultimate relaxation of the conditions imposed on the discarded block would be to check its syntax inasmuch as is needed to identify the presence of labels so that it can be decided whether the block can be discarded or not. Without going to this extreme, all its syntax might be checked. Once it is decided that the block can be discarded, the whole **if** statement might be mandated to be equivalent to the selected block; i.e., the test is resolved at translation time and the "true" branch is retained, the other one discarded, literally. We don't expect this to be proposed any soon, if ever. However, partial incorrect constructions, i.e., pieces of code that would be incorrect were the block not discarded, might be little by little allowed. For example, identifiers with file scope for which there is no definition.

Proposal I. Terminology

6.5 Expressions

6.5.1 General

Insert the following after paragraph 1, or somewhere else.

2 Some expressions are not evaluated during translation nor at program execution. In some cases it can be determined during translation that an expression will never be evaluated. Some of these expressions will be called *value-discarded*. An implementation may consider value-discarded other expressions for which it can determine at translation that they will never be evaluated beyond those so qualified by this document. The term may be applied to a construct other than an expression, with semantics defined in the following paragraph.

3 If an expression or type name in the following subclauses is said to be value-discarded it is also implied that it is not evaluated. If a construct is value-discarded, the expressions therein are not value-discarded or are according as to whether they need to be evaluated at translation time or not;[2)] if it is unspecified whether the expression is evaluated or not, it is not value-discarded.

4 EXAMPLE In the following expression

```
      sizeof(int[1 ? 2 : 4-3]);
```

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 `4-3` is.

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

[2)] Expressions within declarations are evaluated or not during translation independently of whether they are contained in a value-discarded statement (or are themselves value-discarded) or not. **static_assert** declarations take effect also inside discarded statements.

— 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 the name of a type, 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 the name of a type, 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 nor the ones deemed value-discarded in the "statements" subclause.

6 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.

[...]

Forward references: the **sizeof** and **alignof** operators (6.5.4.5), logical OR operator (6.5.15), the **FP_CONTRACT** pragma (7.12.3), copying functions (7.26.2).

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 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.[3)]

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 evaluated.[5)] If the type of the operand is a variably modified type, the operand is evaluated; otherwise, the operand is not evaluatedvalue-discarded.

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 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 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 evaluatedvalue-discarded.

6.8 Statements and blocks

6.8.1 General

Semantics

6 A statement is *isolated* if it is not labeled, does not contain regular labels and if it contains any **`case`** or **`default`** label it contains the whole switch body to which it is associated.

6.8.2 Labeled statements

Syntax

1 *label:*

regular-label
attribute-specifier-sequence$_{\text{opt}}$ **`case`** *constant-expression* **`:`**
attribute-specifier-sequence$_{\text{opt}}$ **`default:`**

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

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

[5)] 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.

regular-label:
attribute-specifier-sequence$_{\text{opt}}$ *identifier* **:**

labeled-statement:
label statement

6.8.5 Selection statements

6.8.5.2 The `if` statement

Semantics

2 In both forms, the first substatementsecondary block is executed if the expression compares unequal to 0. In the **`else`** form, the second substatementsecondary block is executed if the expression compares equal to 0. If the first substatementsecondary block is reached via a label, the second substatementsecondary block is not executed. If the controlling expression is an integer constant expression and a secondary block which is not executed is isolated, that block is value-discarded.

6.8.5.3 The `switch` statement

5 The integer promotions are performed on the controlling expression. The constant expression in each **`case`** label is converted to the promoted type of the controlling expression. If a converted value matches that of the promoted controlling expression, control jumps to the statement or declaration following the matched **`case`** label. Otherwise, if there is a **`default`** label, control jumps to the statement or declaration following the **`default`** label. 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, that body is value-discarded.[6)]

6.8.6 Iteration statements

6.8.1.1 The `while` statement

1 The evaluation of the controlling expression takes place before each execution of the loop body. If the controlling expression is an integer constant expression with value 0 and the loop body is isolated, that body is value-discarded.

6.8.1.2 The `for` statement

2 Both *clause-1* and *expression-3* can be omitted. An omitted *expression-2* is replaced by a nonzero constant. If the controlling expression is an integer constant expression with value 0 and the loop body is isolated, this body and *expression-3* are value-discarded.

6.8.2 Jump statements

Semantics

3 A jump statement causes an unconditional jump to another place. Let S be the innermost statement enclosing the jump statement or equal to it which is the secondary block of a selection statement or an iteration statement, or is a function body. If the block items inside S following the jump statement, in case they were enclosed in braces, would constitute an isolated statement as defined in 6.8.1, those items are value-discarded.[7)]

[6)] The expressions in each **`case`** label are not value-discarded (6.5.1, par. 3).

[7)] Between the jump statement and the end of block S there may be closing braces corresponding to opening braces that appear before the jump statement. These braces are not block-items and are excluded from the previous construction.

Proposal Ib. Switch label

In addition, introduce the term *switch label.*

6.7.13.6 The `fallthrough` attribute

Constraints

1 The attribute token **`fallthrough`** shall only appear in an attribute declaration (6.7); such a declaration is a *fallthrough declaration*. No attribute argument clause shall be present. A fallthrough declaration may only appear within an enclosing **`switch`** statement (6.8.5.3). The next block item (6.8.3) that would be encountered after a fallthrough declaration shall be a **`case`** label or **`default`**switch label associated with the innermost enclosing **`switch`** statement and, if the fallthrough declaration is contained in an iteration statement, the next statement shall be part of the same execution of the secondary block of the innermost enclosing iteration statement.

Semantics

2 The **`__has_c_attribute`** conditional inclusion expression (6.10.2) shall return the value `202311L` when given **`fallthrough`** as the pp-tokens operand if the implementation supports the attribute.

Recommended practice

3 The use of a fallthrough declaration is intended to suppress a diagnostic that an implementation can otherwise issue for a **`case`** or **`default`**switch label that is reachable from another **`case`** or **`default`**switch label along some path of execution. Implementations are encouraged to issue a diagnostic if a fallthrough declaration is not dynamically reachable.

6.8 Statements and blocks

6.8.1 General

Semantics

6 A statement is *isolated* if it is not labeled, does not contain regular labels and if it contains any **`case`** or **`default`**switch label it contains the whole switch body to which it is associated.

6.8.2 Labeled statements

Syntax

1 *label:*
regular-label
switch-label

regular-label:
attribute-specifier-sequence$_{\text{opt}}$ *identifier* **`:`**

switch-label:
attribute-specifier-sequence$_{\text{opt}}$ **`case`** *constant-expression* **`:`**
attribute-specifier-sequence$_{\text{opt}}$ **`default:`**

labeled-statement:
label statement

Constraints

2 A **`case`** or **`default`**switch label shall appear only in a **`switch`** statement. Further constraints on such labels are discussed under the **`switch`** statement.

6.8.2.3 The `switch` statement

Constraints

1 The controlling expression of a **`switch`** statement shall have integer type.

2 If a **`switch`** statement has an associated **`case`** or **`default`**switch label within the scope of an identifier with a variably modified type, the entire **`switch`** statement shall be within the scope of that identifier.[8)]

Semantics

4 A **switch** statement causes control to jump to, into, or past the statement that is the *switch body*, depending on the value of a controlling expression, and on the presence of a **default** label and the values of any **case** labels on or in the switch body. A **case** or **default**switch label is accessible only within the closest enclosing **switch** statement.

[8)] That is, the declaration either precedes the **switch** statement, or it follows the last **case** or **default** label associated with the **switch** that is in the block containing the declaration.

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 evaluateddiscarded relative to the expression.[9)]

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

Proposal III. Allowing certain constructs in discarded expressions

Integer and arithmetic constant expressions

6.6 Constant expressions

8 An *integer constant expression*[10)] 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.

[10)] 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 **??**.

6.7.11 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 part of an essentially discarded expression or type name.:

— 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)other than as part of an essentially discarded expression or type name, somewhere in the entire program there shall be exactly one external definition for the identifier; otherwise, there shall be no more than one.