Author: Javier A. Múgica

Purpose: Precise definition of concepts

Date: 2025 - september - 22

This paper intends to make clear the status of generic selections and parentheses surrounding expressions of a certain kind, when at some points the standard mandates an expression of that kind.

This is N3605 with some editorial changes applied and a better analysis of attribute messages.

Analysis

A literal is a kind of token at translation phase 7, endowed with a type and a value. Its definition is very precise, so that when a literal is mandated it does not follow that a parenthesised literal is allowed, neither a generic selection where the selected expression is a literal. However, it is tacitly agreed that these are allowed, unless the mandate comes from a syntax rule. Here follow cases where one or the other happens.

Cases

Literals in constant expressions

Integer constant expressions allow literals of arithmetic type as operands to casts. The following code is accepted by compilers without warnings:

```
enum A{a=(int)(_Generic(0, int: (2.5)))};
```

Here, parentheses surround the literal `2.5` and the generic selection.

More generally, constant expressions of different kinds allow only operands of certain types, but any operand in _Generic which is not the selected expressions is not considered for this:

```
constexpr int a= _Generic((float*)0, default:0);
```

String literals as initializers

The definition for this is "a string literal, optionally enclosed in braces, or ..."

```
const char s[]= ("word");
const char s[]= _Generic(0, default: "word");
```

Gcc warns on "Array initialized from parenthesis", while the other compilers tested give no warning (clang, MSVC and ICX). No compiler warns because of **`_Generic`**.

Null pointer constant

The standard currently defines

> An integer constant expression with the value 0, such an expression cast to type **`void*`**, or the predefined constant **`nullptr`** is called a *null pointer constant*.

The following is accepted by compilers with no warning:

```
float (*f)(void) = (_Generic(1, default: (void*)0));
```

static_assert

Here, neither parentheses nor _Generic are allowed:

```
static_assert(1, ("Error message")); // Error
```

and similarly for **_Generic**.

attribute messages

```
[[nodiscard(
    _Generic(1, default: "Do not discard this")
)]]
int important_func(void);
```

ICX accepts it (invoking it with the option `-std=c2x`). Gcc and Clang reject it.

```
[[nodiscard(("Do not discard this"))]] int important_func(void); // Error
```

ICX and Clang accept it; gcc rejects it. Gcc accepts it if the code is compiled as C++.

The divergence in the attribute case speaks for the need to make explicit (here and elsewhere) whether parentheses and generic selections are allowed or not.

In both attributes and static_assert the string literal is part of the syntax of the feature. We understand therefore that gcc is right in rejecting the parenthesised string in place of the attribute message, and the behaviour of the other two compilers can be understood as an extension (thought maybe it is an unintended mistake, or a known "mistake" they don't intend to fix).

Solutions considered

Quasi-literal

Our first choice was the definition of *quasi-literal,* to be placed in the section on primary expressions:

> The following are *quasi-literals:*
>
> — A literal.
>
> — A generic selection where the selected expression is a quasi-literal.
>
> — A parenthesized quasi-literal.
>
> A quasi-literal is of the same kind as the literal on which it is based: integer quasi-literal, string quasi-literal, etc.

And use quasi-literal instead of literal in a few places.

Generic replacement

The term quasi-literal is of no use for null pointer constants. It is for this reason that the concept *generic replacement* was conceived:

> *Generic replacement* refers to the process of replacing a generic selection by its result expression, enclosed in parentheses if the expression is not a primary expression. (Therefore, the result of generic replacement is always a primary expression).

Using this term, the definition of null pointer constant would become

> A *null pointer constant* is an expression that, after generic replacement and removal of all surrounding parentheses, is an integer constant expression with the value 0 or such an expression cast to type **`void`** `*`, or the predefined constant **`nullptr`**.

Immediate constant

While the term *generic replacement* serves well for null pointer constants and string literals as initializers, the wording for floating operands in integer constant expressions remains very verbose, in part because it is not only literals that are allowed, but "floating, named, or compound literal constants of arithmetic type". All this must be subject to generic replacement and parentheses removal, resulting in the wording seen in the proposal.

To simplify that wording, the use of the new term *immediate constant* is proposed. Its definition is not placed under "primary expressions", where we intended that of quasi-literals to be, but in the section for constant expressions, because it is only applied there. We propose it separate from the main re-wording proposal. N3605 discusses the different names considered for the term. The term *immediate* is only used by the standard in the construction *immediate operand*, and only twice: once in an example and the other one precisely here, in *immediate operands of casts*.

Not addressed

The wording we propose for integer constant expressions takes care of generic selections and surrounding parentheses for the literals and constants of arithmetic type that are allowed as operands to casts. We do not take care of any operand that may be in that situation, for integers or for arithmetic constant expressions, as for example

```
_Generic(sqrt(2.0), default: sizeof(float))
```

In order to handle this in the wording, an "**after generic replacement**" would have to be inserted preceding the enumeration of all possible operands.

We do not do that because we believe that this is better achieved by a deeper change of the wording for these two kinds of expressions, that would list the atomic ones and then a point saying that an expression is an ICE if its operands are either discarded by the expression or ICEs, and similarly for ACE. That change is the subject of another proposal.

This notwithstanding, the introduction of the term *immediate constant*, in addition of simplifying the wording, handles generic selections for all cases when the result expression is a constant or literal. If the complete rewriting of integer and arithmetic constant expressions based on the discarded concept takes place, the term *immediate constant* could be put to other use, but that is no reason for not adopting it now: if it need be changed in the future, be it so.

Wording

Gray text is text to be removed; blue text it new text; green text is changed text.

Main proposal

Add at the end of the semantics of generic selections, 6.5.2.1, the following:

Generic replacement is the process of replacing a generic selection by its result expression, enclosed in parentheses if the expression is not a primary expression (therefore, the result of generic replacement is always a primary expression). When the term is applied to an expression, it means the repeated application of generic replacement to the generic selections within it until no generic selections remain.

Change also, in "6.3.3.3 Pointers":

An integer constant expression with the value 0, such an expression cast to type `void *`, or the predefined constant `nullptr` is called a *null pointer constant*. A *null pointer constant* is an expression that, after generic replacement and removal of all surrounding parentheses, is an integer constant expression with the value 0 or such an expression cast to type **`void *`**, or the predefined constant **`nullptr`**.

And maybe add a forward reference to generic selection.

In 6.6 Constant expressions,

7 An *integer constant expression* has integer type and only has operands that are integer literals, named and compound literal constants of integer type, character literals, **`sizeof`** or **`_Countof`** expressions which are integer constant expressions, **`alignof`** expressions, and floating, named, or compound literal constants of arithmetic type that are the immediate operands of casts.; when these operands have floating type, they only appear in cast expressions of integer type where the operand, after generic replacement, is such an operand optionally enclosed in an arbitrary number of parentheses. Cast operators in an integer constant expression only convert arithmetic types to integer types, except as part of an operand to the typeof operators, **`sizeof`** operator, **`_Countof`** operator, or **`alignof`** operator.

In 6.7.11 Initialization,

7 The initializer for an array shall be either a string literal, optionally enclosed in braces,an expression that after generic replacement and removal of all surrounding parentheses is a string literal, such an expression enclosed in braces, or a brace-enclosed list of initializers for the elements.

Immediate constant

(Only if the previous proposal is accepted).
With the introduction of a term here, the text on integer and arithmetic constant expressions reduces to the following:

7 An *immediate constant* is an expression that, after generic replacement and removal of surrounding parentheses, is a literal, a compound literal constant or a named constant.

8 An *integer constant expression* has integer type and only has operands that are immediate constants of integer type, **`sizeof`** or **`_Countof`** expressions which are integer constant expressions, **`alignof`** expressions, and immediate constants of arithmetic type that are the immediate operands of casts. Cast operators in an integer constant expression only convert arithmetic types to integer types, except as part of an operand to the typeof operators, **`sizeof`** operator, **`_Countof`** operator, or **`alignof`** operator.

10 An *arithmetic constant expression* has arithmetic type and only has operands that are immediate constants of arithmetic type and integer constant expressions. Cast operators in an arithmetic constant expression only convert arithmetic types to arithmetic types, except as part of an operand to the typeof operators, **sizeof** operator, **_Countof** operator, or **alignof** operator.