Author: Javier A. Múgica

Purpose: Clarification

Date: 2025 - may - 8

This paper proposes a rewording of the description of the value of for certain operands, replacing comparison to zero by the more natural conversion to bool.

Analysis

Take for example the specification of the logical OR operator:

> The `||` operator shall yield 1 if either of its operands compare unequal to 0; otherwise, it yields 0. The result has type **`int`**.

This text was written when only numbers (integers and floating-point) and pointers could be its operands. What it meant was that "is not zero (for arithmetic types) or not null (for pointer types)". Instead of plainly saying this, it takes the roundabout of expressing it via comparison against zero. Thereby the description is shorter, and back then comparison of pointers against a literal zero was more common than is today.

Years have passed; comparison of a pointer against zero has become questioned and, although that possibility may never be removed, phrasing nowadays "the value is not null" as "compares unequal to 0" does not seem a good choice.

More important, the types bool and nullptr_t have been added, making the description via comparison against zero more complicated than it was when it was originally written. (Thrice as complicated, we may say, since besides numbers, the other types do not compare naturally to a number, zero inlcluded).

Curiously, the description of the assert macro uses the word "false", but insists in comparing to zero: "[...] is false (that is, compares equal to 0)". This clearly predates the introduction of the constant **`false`**.

We may compare those wordings with the wording for conversion to bool, which is newer:

> When any scalar value is converted to **`bool`**, the result is **`false`** if the value is a zero (for arithmetic types), null (for pointer types), or the scalar has type **`nullptr_t`**; otherwise, the result is **`true`**.

This is the natural way to express it. Further, comparison of **`nullptr`** against zero hangs on a very slender thread: In contrast to pointers, that can be naturally represented as an integer, and compared to any integer if one is cast to the type of the other, as in `p == (void*)1` or `(uintptr_t)p == 1`, whereby the lack of need of a cast for the constant 0 is just a shortcut, **`nullptr`** cannot. It can be compared against zero not because **`nullptr`** can be transformed to an integer or vice-versa, but because comparison to a null *pointer* is allowed for it.

Instead of repeating the words used for describing conversion to bool, we can take advantage of their existence there, to phrase the semantics of those operators based on it, on the conversion to bool of their operands. This is how these operators are universally described: "the result is true if any of the operands is true", for example, for the OR operator.

We have also taken advantage of the change in the wording to suppress the "shall be" in the semantics of the OR and AND operators, where it is used with the meaning "is".

We have not modified the wording for static_assert, where it should plainly say "with a nonzero value" (the expression has integer type), because there is already a proposal fixing that *(static_assert without UB)*.

NaN

For implementations that conform to annex F, or at least carry out comparisons as specified by F.9.4, the following holds: "The expression $x = x$ is false if x is a NaN". Therefore, a NaN compares unequal to zero (previous wording) and gets converted to **true** (new wording).

For any implementations whatsoever, the standard already imposes, with respect to the == and != operators, that "**For any pair of operands, exactly one of the relations is true**". Although it does not mention which one of the two is true when one of the operands is a NaN, given that a NaN is a value different from zero, we understand the wording as implying that NaN == 0 is false, in any implementation. (Just as it is implied that 1 != 0 without any need to an explicit mention that for finite values the relation == is true if the values are the same).

For conversion to bool it is specified that any value other than zero converts to **true**. The new wording imposes the obvious choice where previously a forced interpretation of the standard could imply that the result was unspecified.

Wording

Unary arithmetic operators (6.5.4.4):

5 The result of the logical negation `!` operator is 0 if its operand becomes **`true`** when converted to **`bool`**; otherwise it is 1. The result has type **`int`**. The expression `!E` is equivalent to `(0==E)`.

The last sentence may be removed. It is still true, but may not be pertinent any more. This would be a further change, that we leave up to the committee.

Logical AND operator (6.5.14):

Semantics

3 The result of the logical AND expression is 1 if both of its operands are **`true`** when converted to **`bool`**; otherwise, it is 0. The result has type **`int`**.

4 Unlike the bitwise binary `&` operator, the `&&` operator guarantees left-to-right evaluation. If the first operand is **`false`** when converted to **`bool`**, the second operand is not evaluated. If the second operand is evaluated, there is a sequence point between the evaluations of the first and second operands.

Logical OR operator (6.5.15):

Semantics

3 The result of the logical OR expression is 0 if both of its operands are **`false`** when converted to **`bool`**; otherwise, it is 1. The result has type **`int`**.

4 Unlike the bitwise binary `|` operator, the `||` operator guarantees left-to-right evaluation. If the first operand is **`true`** when converted to **`bool`**, the second operand is not evaluated. If the second operand is evaluated, there is a sequence point between the evaluations of the first and second operands.

Conditional operator (6.5.16):

Semantics

5 The first operand is evaluated and its value converted to **bool**. There is a sequence point after this conversion. If the boolean value is **true**, the second operand is evaluated; otherwise, the third operand is evaluated. The result is the value of the second or third operand (whichever is evaluated), converted to the type described subsequently in this subclause.

The if statement (6.8.5.2):

Semantics

2 The controlling expression is evaluated and converted to **bool**. In both forms, the first secondary block is executed if the result of the conversion is **true**. In the **else** form, the second secondary block is executed if the result is **false**. If the first secondary block is reached via a label, the second secondary block is not executed.

Example 2 for the if statement is

EXAMPLE 2 The controlling expression of any **if** statement is always implicitly compared to 0 by the statement itself:

```
double x = DBL_SNAN;
if (x) {
      // fetestexcept (FE_INVALID) is nonzero because of the comparison
}
```

It can be adapted to the new wording. But we propose its removal. Now the semantics says that the controlling expression *is* converted to bool, whereas the previous wording said "if the controlling expression compares unequal to 0", thereby making a clarification appropriate: "is always implicitly compared to 0".

Iteration statements (6.8.6):

Semantics

3 An iteration statement causes a secondary block called the loop body to be executed repeatedly until the value of the controlling expression is **false** when converted to **bool**. The repetition occurs regardless of whether the loop body is entered from the iteration statement or by a jump.

The for statement (6.8.6.4):

2 Both *clause-1* and *expression-3* can be omitted An omitted *expression-2* is replaced by **true**.

Also, in the text of the footnote immediately preceeding this paragraph, replace "compares equal to 0" by "is false" (no fixed-wdith font).

The assert macro (7.2.2.1):

Description

2 The **assert** macro puts diagnostic tests into programs; it expands to a void expression. When it is executed, if the value of `expression` (which shall have a scalar type) is **false** (when converted to **bool**), [...]