Author: Javier A. Múgica

Purpose: Integration of existing features

Date: 2025 - june - 22

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

It modifies N3546 in that it states explicitly that the operands of `!`, AND and OR are converted to bool and in that it keeps the example in the "if" statement that N3546 proposed to remove. It also adds some wording for conversions to and from bool for the preprocessor.

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 included).

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 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

The text in the standard for conversion of arithmetic types to **`bool`** specifies that any value other than zero converts to **true**. For the new wording to match the current behaviour, it is needed that NaNs compare unequal to zero. The standard already imposes, with respect to the `==` and `!=` operators, that “For any pair of operands, exactly one of the relations is true”. Althought 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. (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). The more since even the expression `x == x` is false if `x` is a NaN (in ISO/IEC 60559).

Even if one forced the standard to read that NaN == 0 can be true, the present wording would change an unspecified behaviour by a specified one (but we don't think that reading is acceptable).

Side effects

There is an improper use of “shall be” in “The `||` operator shall yield 1 if... ”, and analogously for the && operator. In the new wording we use “is”.

We also fix a mistake in the wording of the logical AND expressions: “The && operator shall yield 1 if both of its operands compare unequal to zero”. Consider `0 && 1/0`. Here the second operand cannot be compared to zero.

Wording

Unary arithmetic operators (6.5.4.4):

5 The logical negation operator `!` converts its operand to **`bool`**; then negates it (the negation of **`false`** is **`true`**, the negation of **`true`** is **`false`**), then converts the result to **`int`**. The result has type **`int`**.[†]

Logical AND operator (6.5.14):

Semantics

3 The first operand is converted to **`bool`**. There is a sequence point after this conversion. If the result of the conversion is **`false`** the second operand is not evaluated; otherwise, the second operand is converted to **`bool`**. If any of the conversions result in **`false`**, the result of the AND expression is 0; otherwise, it is 1. The result has type **`int`**.

[†] If we represent by ¬ an operator interchanging **`true`** and **`false`**, the expression `!E` is equivalent to `(`**`int`**`)¬(`**`bool`**`)E`.

Logical OR operator (6.5.15):

Semantics

3 The first operand is converted to **bool**. There is a sequence point after this conversion. If the result of the conversion is **true** the second operand is not evaluated; otherwise, the second operand is converted to **bool**. If any of the conversions result in **true**, the result of the OR expression is 1; otherwise, it is 0. The result has type **int**.

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 The conversion to **bool** of the controlling expression may have side effects:

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

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).

Preprocessor conditional inclusion (6.10.2):

Add a new paragraph:

15 The operators !, &&, || and the conditional operator convert their operands, or some of them, to **bool** (6.5.4.4, 6.5.14, 6.5.15 and 6.5.16). Any nonzero value is converted to **true** and zero is converted to **false**. Of these, the ! operator converts its result back to a signed integer. The value **true** is converted to 1 and the value **false** is converted to 0.

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`**, [...]