ISO/IEC JTC 1/SC 22/WG14 **N2935 v8**
WG 21, SG 22 **P2311R2**

2022-2-15

Make false and true first-class language features
proposal for C23

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

In its London 2019 meeting, WG14 has found consensus to elevate **false** and **true** to proper keywords.
Changes in v2: WG14 was not sympathetic to force these keywords also to be macros, so we remove the text corresponding to this idea. WG14 also was not in favor of the parts that proposed to introduce recommended practice and to add future language directions, so these are also removed.
Changes in v3: It was then observed in a discussion on the reflector, that the possible use of these predefined constants in the preprocessor needs some more precautions.
Changes in v4: Now that the type change has been integrated into C23, it remains to integrate the new keywords properly into all translation phases.
Changes in v5:
— Make it clear that the constants count as integer constant expressions.
— Synchronize the handling in the preprocessor with C++.
— Explicitly mark the macro **__bool_true_false_are_defined** as obsolescent and keep it as last remaining content in `<stdbool.h>`.
Changes in v6:
— Simplify the approach that makes them integer constant expressions.
— Synchronize the possible definition as predefined macro with N2934.
— Use the change to the **bool** type that previously was an alternate form. WG14 chose that one.
— Move the special promotion rules for the constants where they belong, namely to the definition of integer promotion.
— Make an optional proposal for a change for integer promotions of type **bool**.
Changes in v7:
— After some discussion on the WG14 reflector is was found that making the text for preprocessing similar to C++ would introduce more problems than it solves. In C, all relational operators have type **int**, so the question how **bool** expressions during preprocessing convert does never occur. So we don't need to introduce the concept of **bool**, there. Consequently for preprocessor conditionals we fall back to a simple replacement of the keywords by `0` and `1`, respectively.
— Add an option to force the width of **bool** to 1.
Changes in v8: Wording as decided by WG14
— Don't make the header obsolescent.

1. INTRODUCTION

The integration of Boolean constants **false** and **true** as proper language constructs, is meant to provide a better feedback to programmers for the use of these constants by the translantor or from debuggers. In particular, diagnostics will hopefully be provided when they are used in arithmetic or used contrary to the intent, *e.g* as null pointer constants.

2. IMPACT

A possible impact of changing **false** and **true** to keywords could be the use of these constants in preprocessing conditional expressions. Currently preprocessing arithmetic sees the existing macros from `<stdbool.h>` as signed values, and thus the result of expressions is merely consistent between the preprocessor and the rest of the language. When changing to keywords we should ensure that **false** and **true** may still be used in the preprocessor with the same semantics as before. This is done by enforcing that in preprocessor conditionals **true** is replaced by `1`; **false** is replaced as any other identifier that remains in such a conditional by `0`. This ensures that preprocessor arithmetic uses signed values for these constants and that results of such arithmetic remain the same between C17 and C23.

© 2022 by the author(s). Distributed under a Creative Commons Attribution 4.0 International License

3. REFERENCE IMPLEMENTATION

To add minimal support for the proposed changes, an implementation that does not yet want to implement **false** and **true** as full-featured keywords would have to add definitions that are equivalent to the following lines to their startup code:

```
#define false          ((bool)+0)
#define true           ((bool)+1)
```

Notice that these do not use the literals `0U` or `1U` because with that arithmetic with these constants in the preprocessor would be performed as unsigned integers. This would have the consequence that something like **-true** would result to **UINTMAX_MAX** in the preprocessor and `-1` otherwise.

4. CHANGES

We assume that the non-optional part of N2934 has been integrated into C23, otherwise the present paper is obsolete. Predefined constants need a little bit more effort for the integration, than the other keywords in N2934, because up to now C did not have named constants on the level of the language.

4.1. Syntax

We propose to integrate these constants by means of a new syntax term `predefined constant`. The text itself is then integrated as a specific clause.

CHANGE 1. *Add* **false** *and* **true** *into the alphabetic order of 6.4.1.*

CHANGE 2. *Add a new syntax item* predefined-constants *to the end of 6.4.4 p1, Constants.*

CHANGE 3. *Add a new clause 6.4.4.5 as follows.*

6.4.4.5 Predefined constants

Syntax

1 *predefined-constant:*
    false
    true

Description

2 Some keywords represent constants of a specific value and type.

3 The keywords **false** and **true** are constants of type **bool** with value 0 for **false** and 1 for **true**.

FOOTNOTE[The constants **false** and **true** promote to type **int**, see 6.3.1.1. When used for arithmetic in translation phase 4, they are signed values and the result of such arithmetic is consistent with results of later translation phases.]

Also, the predefined constants should be constants of the right kind.

CHANGE 4. *Add to* `6.6 p6`*:*

6 An integer constant expression*[127] shall have integer type and shall only have operands that are integer constants, enumeration constants, character constants,* predefined constants, **sizeof** *expressions whose results are integer constants,*

> ***alignof** expressions, and floating constants that are the immediate operands of casts. Cast operators in an integer constant expression shall only convert arithmetic types to integer types, except as part of an operand to the **sizeof** or **alignof** operator.*

Change 5. *Add to* `6.6 p8`*:*

> *8 An* arithmetic constant expression *shall have arithmetic type and shall only have operands that are integer constants, floating constants, enumeration constants, character constants,* predefined constants, **sizeof** *expressions whose results are integer constants, and* **alignof** *expressions. Cast operators in an arithmetic constant expression shall only convert arithmetic types to arithmetic types, except as part of an operand to a* **sizeof** *or* **alignof** *operator.*

4.2. Interaction with legacy code

There is still some code in the field that redefines these keywords. When compiler versions for C23 come out, it would be important that there is no silent redefinition of types or values depending on which headers are included and in which order.

Change 6. *Add the following to 6.10.8 p2:*

> *None of these macro names, nor the identifiers* **defined** *or* `__has_c_attribute`*, shall be the subject of a* **#define** *or a* **#undef** *preprocessing directive. Any other predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore or shall be any of the identifiers* **alignas**, **alignof**, **bool**, **false**, ~~or~~ **static_assert**, or **true** .

4.3. The bool type

Definitions of the **bool** type should now directly refer to the constants and make no fuzz about zero or non-zero values anymore.

Change 7. *In 6.2.5 (Types) make the following change to p2:*

> *An object declared as type* **bool** *is large enough to store the values* ~~0~~**false** *and* ~~1~~**true**.

The current state of conversion to the type **bool** makes several implicit references back and forth between conversions and the equality operator.[1] The changes proposed here, give an opportunity to improve that situation and WG14 has seen this favorably.

Change 8. *In 6.3.1.2 (Boolean type) make the following change to p1 and remove the corresponding footnote:*

> *When any scalar value is converted to* **bool**, *the result is* ~~0~~**false** *if the value* ~~compares equal to 0~~is a zero (for arithmetic types) or null (for pointer types)*;*~~(FNT)~~ *otherwise, the result is* ~~1~~**true**.

[1]The process of converting a **long** to **bool** is *e.g* as follows: `1L` $\Longrightarrow$ `(1L == 0)` $\Longrightarrow$ `(1L == 0L)` $\Longrightarrow$ **false**.

4.4. Preprocessing

The token **true** needs a specific exception during preprocessing, such that constructs such as the following do not have surprising results.

```
#if true
...
#endif
```

In contrast to that, **false** needs no special treatment, since identifiers that remain in preprocessor conditionals after macro replacement are replaced with `0`, anyhow. But to make that behavior clear, we add **false** as an example for those identifiers that produce `0`.

CHANGE 9. *In 6.10.1 p7, amend the following partial phrase:*

> *... all remaining identifiers* other than **true** *(including those lexically identical to keywords* such as **false***) are replaced with the pp-number 0,* **true** is replaced with the pp-number 1, *...*

Because transitionally these new keywords might still have predefined macro definitions, we also add them to the list for which the spelling after preprocessing is unspecified.

CHANGE 10. *In 6.4.1 p2' (as of N2934) make the following changes:*

> *The spelling of these keywords*~~, and~~ *their alternate forms*, and of **false** and **true** *inside expressions that are subject to the* **#** *and* **##** *preprocessing operators is unspecified.*

4.5. Changes to library clauses

Clause 7.18 `<stdbool.h>`

CHANGE 11. *Replace the content of clause 7.18 by*

> The header `<stdbool.h>` provides the obsolescent macro **__bool_true_false_are_defined** which expands to the integer constant 1.

Update the corresponding entry for future library directions, the macros to which this referred do no longer exist:

CHANGE 12. *Replace the content of clause 7.31.11 "Boolean type and values* `<stdbool.h>`*" by*

> The macro **__bool_true_false_are_defined** is an obsolescent feature.

Clause 7.26 `<threads.h>`

This header has several functions or macros that return **`bool`** values.

CHANGE 13. *In 7.17.5.1, 7.17.7.4 and 7.17.8.1 change the specification of return values to the keywords* **`false`** *and* **`true`** *where appropriate.*

4.6. Integer promotions and width of `bool`.

Since the beginning, there has been an inconsistency in C that on some special architectures the **`bool`** type is promoted to **`unsigned int`** instead of **`int`**, whereas **`bool`** bit-fields of width 1 and the symbolic constants **`false`** and **`true`** are always promoted to **`int`**. This is the case for architectures where the types **`bool`**, **`unsigned char`** and **`unsigned short`** not only have the same size as **`int`** but also the same width. On these architectures the representation of a **`bool`** object could be manipulated to represent a value as large as **`UINT_MAX`**.
As an explicit choice by WG14, this paper changes this status quo. We make that change for C23, because here we newly introduce the macros **`BOOL_WIDTH`** and **`BOOL_MAX`**. Users can reasonably expect these to be stable over different versions of the C standard.
This is a normative change for those rare architectures described above that removes an incompatibility with C++.

CHANGE 14. *Change the final sentence of 6.2.6.2 p1 (Integer types)*

The type **`bool`** shall have one value bit and **`sizeof(bool)*CHAR_BIT - 1`** padding bits. Otherwise, t~~T~~*here need not be any padding bits;* **`unsigned char`** *shall not have any padding bits.*

CHANGE 15. *and add a footnote to the entry for* **`BOOL_WIDTH`** *in 5.2.4.2.1 p1*

FOOTNOTE) This value is exact.

5. QUESTION FOR WG14

QUESTION. *Does WG14 want to integrate the changes as proposed in N2935 into C23?*

Acknowledgement

We thank Joseph Myers, JeanHeyd Meneide and Aaron Ballmann as well as the C/C++ liaison study group for feedback and discussions.