Make false and true first-class language features

Jens Gustedt

ISO/IEC JTC 1/SC 22/WG14 **N 2393**

v 1

June 8, 2019

Make false and true first-class language features

proposal for C2x

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. This is only a first step to make these constants first-class language features and to achieve a full compatibility with C++. Therefore we need also to change their type, namely to change them from **int** to **bool**, and a deprecation of their use in contexts which are non-logical.

1. INTRODUCTION

The Boolean constants **false** and **true** are a bit ambivalent because in C17 they expand to integer constants `0` and `1` that have type **int** and not **_Bool**. This is unfortunate when they are used as arguments to type-generic macros, because there they could trigger an unexpected expansion, namely for **int** instead of **_Bool**. Since for C++, these constants are of type **bool**, we propose to do it the same.

The integration of these constants as proper language constructs, also allows to provide a better feedback to programmers, where such constants seem to be used erroneously. In particular, diagnostics may be provided when they are used in arithmetic or used contrary to the intent, *e.g* as null pointer constants. Such diagnostics could be enforced by adding constraints to the corresponding text that defines the constants, but this would require that implementations provide proper implementations for the constants already for C2x. To give them more time to adapt, we thus only recommend diagnostics in case of misuse, and add a depreciation as a future language direction.

2. IMPACT

The change should not have a big impact on user code. In most contexts where these constants are used (assignment, arithmetic, comparison, non-prototyped function call), **bool** values will be promoted to **int**, anyhow. So in these "regular" contexts the result after promotion would be exactly the same, namely **int** values 0 and 1, respectively. As arguments to function calls that provide a prototype, there is no change either, since the values 0 and 1 are valid for any arithmetic type and so the value and type received by the function are exactly the same.

The change can have marginal impact on existing code, when the constants are used in **sizeof**, **alignas** or **_Generic** expressions. All should be relatively rare. The latter, **_Generic**, is a sought effect of this change, because we think that choosing **bool** for these constants is a much more natural choice and will surprise less. In any case, these usages are compile-time detectable and we expect that quality implementations can provide diagnostics during the transition phase to C2x.

For implementations that do not want to integrate these constants in their parser infrastructure, yet, this can easily be achieved by definitions similar as specified below. By preprocesor magic, these expand to values `0` and `1` when used in preprocessor conditionals, and to constant expressions of type **bool** in other places.

3. REFERENCE IMPLEMENTATION

To add minimal support for the proposed changes, an implementation would have to add definitions that are equivalent to the following lines to their startup code:

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

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

At the other end of the spectrum, an implementation that implements these new keywords as first-class constructs can simply have definitions that are the token identity:

```
#define false         false
#define true          true
```

4. CHANGES

On top of the "keywords" paper, the changes for this feature are quite minimal.

(1) We propose the replacement of the token **int** by **bool** in one place.
(2) We add a footnote that spells out the trick from above for macros that are usable in preprocessor conditionals.
(3) We add "recommended practice" to discourage the use of these constants in contexts that are not "logical".
(4) We make the unintended use cases obsolescent by adding a clause to "Future language directions".

Forward references: common definitions <stddef.h> (7.19), the **mbtowc** function (7.22.7.2), Unicode utilities <uchar.h> (7.28).

6.4.4.5 Predefined constants

Syntax

1 *predefined-constant:*
false
true

Description

Some keywords represent constants of a specific value and type.

6.4.4.5.1 The **false** and **true** constants

Description

1 The keywords **false** and **true** represent constants of type ~~**int**~~ **bool** that are suitable for use as are integer literals. Their values are 0 for **false** and 1 for **true**.[86)]

Recommended practice

2 The intended use of these constants is as logical values. In particular, they should only be used for initialization or assignment of **int** or **bool** objects, as function arguments for parameters of these types, as controlling expressions and as operands of logical expressions. Implementations are encouraged to diagnose violations of this intent.

6.4.5 String literals

Syntax

1 *string-literal:*
encoding-prefix$_{\text{opt}}$ " *s-char-sequence*$_{\text{opt}}$ "
encoding-prefix:
u8
u
U
L

s-char-sequence:
s-char
s-char-sequence s-char

s-char:
any member of the source character set except
the double-quote ", backslash \, or new-line character
escape-sequence

Constraints

2 A sequence of adjacent string literal tokens shall not include both a wide string literal and a UTF–8 string literal.

Description

3 A *character string literal* is a sequence of zero or more multibyte characters enclosed in double-quotes, as in "xyz". A UTF–*8 string literal* is the same, except prefixed by **u8**. A *wide string literal* is the same, except prefixed by the letter **L**, **u**, or **U**.

[86)] Thus, the keywords **false** and **true** are usable in preprocessor directives. Suitable implementations for **false** and **true** are predefined macros that expand to ((**bool**)+0) and ((**bool**)+1), respectively.

6.11 Future language directions

6.11.1 Floating types

1 Future standardization may include additional floating-point types, including those with greater range, precision, or both than **long double**.

6.11.2 Linkages of identifiers

1 Declaring an identifier with internal linkage at file scope without the **static** storage-class specifier is an obsolescent feature.

6.11.3 External names

1 Restriction of the significance of an external name to fewer than 255 characters (considering each universal character name or extended source character as a single character) is an obsolescent feature that is a concession to existing implementations.

6.11.4 Character escape sequences

1 Lowercase letters as escape sequences are reserved for future standardization. Other characters may be used in extensions.

6.11.5 Predefined constants

1 The following are obsolescent features:

— the use of **false** and **true** as operands to operators that are non-logical, assignment or equality testing;

— the promotion or implicit conversion of **false** and **true** to **int** other than for use as operands to these operations, as an initializer of an integer object or as a controlling expression;

— the use of **false** as a null pointer constant.

6.11.6 Storage-class specifiers

1 The placement of a storage-class specifier other than at the beginning of the declaration specifiers in a declaration is an obsolescent feature.

6.11.7 Function declarators

1 The use of function declarators with empty parentheses (not prototype-format parameter type declarators) is an obsolescent feature.

6.11.8 Function definitions

1 The use of function definitions with separate parameter identifier and declaration lists (not prototype-format parameter type and identifier declarators) is an obsolescent feature.

6.11.9 Pragma directives

1 Pragmas whose first preprocessing token is **STDC** are reserved for future standardization.

6.11.10 Predefined macro names

1 Macro names beginning with **__STDC_** are reserved for future standardization.

2 Uses of the **__STDC_IEC_559__** and **__STDC_IEC_559_COMPLEX__** macros are obsolescent features.