Revise spelling of keywords and make them feature tests
proposal for C2x

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

Over time C has integrated some new features as keywords (some genuine, some from C++) but the naming strategy has not be entirely consistent: some were integrated using non-reserved names (const, inline) others were integrated in an underscore-capitalized form. For some of them, the use of the lower-case form then is ensured via a set of library header files. The reason for this complicated mechanism had been backwards compatibility for existing code bases. Since now years or even decades have gone by, we think that it is time to switch and to use to the primary spelling.

This is a follow-up paper to N2368 where we reduce the focus to the list of keywords that found consensus in the WG14 London 2019 meeting. Other papers will build on this for those keywords or features that need more investigation.

1. INTRODUCTION

Several keywords in current C2x have weird spellings as reserved names that have ensured backwards compatibility for existing code bases:

```
_alignas  _Bool     _Decimal32   _Imaginary
_alignof  _Complex  _Decimal64  _Noreturn  _Thread_local
_Atomic   _Decimal128 _Generic   _Static_assert
```

Many of them have alternative spellings that are provided through special library headers:

```
alignas      bool          imaginary       static_assert
alignof      complex       noreturn       thread_local
```

In addition, several important constants or language constructs are provided through headers and have not achieved the status of first class language constructs:

```
NULL         _Imaginary_I   offsetof
_Complex_I   false          true
```

The use of these different keywords make C code often more difficult or unpleasant to read, and always need special care for code that is sought to be included in both languages, C and C++. For all of the features it will be ten years since their introduction when C2x comes out, a time that should be sufficient for all users of the identifiers to have upgraded to a non-conflicting form.

Some of the constructs mentioned above have their own specificities and need more coordination with WG21 and C++. E.g a common mechanism is currently sought for the derived type mechanisms for _Complex and _Atomic, or a keyword like _Noreturn might even be replaced by means of the attribute mechanism that has recently been voted into C2x.

This paper reproposes those keywords of N2368 that found direct consensus in WG14, in the expectation that the thus proposed modifications can be integrated directly into C2x:

```
alignas      bool          thread_local   true
alignof      static_assert false
```

© 2019 by the author(s). Distributed under a Creative Commons Attribution 4.0 International License
Other proposals will follow that will tackle other parts of N2368 and beyond:

— Modify `false` and `true` to be of type `bool`.
— Make `noreturn` a keyword or replace it by an attribute.
— Introduce `nullptr` and deprecate `NULL`.
— Make `complex` and `imaginary` keywords and/or provide `__complex(T)` and `__imaginary(T)` constructs for interoperability with C++.
— Make `atomic` (or `__atomic`) a keyword that resolves to the specifier form of `__Atomic(T)`.
— Replace `_Complex_I` and `_Imaginary_I` by first-class language constructs.
— Make `offsetof` a keyword.
— Make `generic` a keyword that replaces `_Generic`.
— Make `decimal32`, `decimal64` and `decimal128` (or `dec32`, `dec64` and `dec128`) keywords that replace `_Decimal32`, `_Decimal64` and `_Decimal128`.

2. PROPOSED MECHANISM OF INTEGRATION

Many code bases use in fact the underscore-capitalized form of the keywords and not the compatible ones that are provided by the library headers. Therefore we need a mechanism that makes a final transition to the new keywords seamless. We propose the following:

— Require the keywords to be also macros that can be tested.
— Don’t allow user code to change such macros.
— Allow the keywords to result in other spellings when they are expanded in with `#` or `##` operators.
— Keep the alternative spelling with underscore-capitalized identifiers around for a while.

With this in mind, implementing these new keywords is in fact almost trivial for any implementation that is conforming to C17.

— 7 predefined macros have to be added to the startup mechanism of the translator. They should expand to similar tokens as had been defined in the corresponding library headers.
— If some of the macros are distinct to their previous definition, the library headers have to be amended with `ifndef` tests. Otherwise, the equivalent macro definition in a header should not harm.

 Needless to say that on the long run, it would be good if implementations would switch to full support as keywords, but there is no rush, and some implementations that have no need for C++ compatibility might never do this.

3. PREDEFINED CONSTANTS

Predefined constants need a little bit more effort for the integration, because up to now C did not have named constants on the level of the language. We propose to integrate these constants by means of a new syntax term `predefined constant`.

For this proposal we only include `false` and `true`. Other proposals will follow for `nullptr` and maybe `_Complex_I` and `_Imaginary_I`.

3.1. Boolean constants

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`.

Nevertheless, `int` is the type that is currently used for them, so in this consensus paper we propose to stay with this. A follow-up paper will propose to change the type to `bool`.
4. FEATURE TESTS
As additional effect of having the keywords to be macros, too, the macros `bool` and `thread_local` (and eventual future `complex` or `atomic`) can be used as feature tests that are independent of library support and of the inclusion of the corresponding header.

5. 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:

```c
#define alignas _Alignas
#define alignof _Alignof
#define bool _Bool
#define false 0
#define static_assert _Static_assert
#define thread_local _Thread_local
#define true 1
```

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

```c
#define alignas alignas
#define alignof alignof
#define bool bool
#define false false
#define static_assert static_assert
#define thread_local thread_local
#define true true
```

6. MODIFICATIONS TO THE STANDARD TEXT
This proposal implies a large number of trivial modifications in the text, namely simple text processing that replaces the occurrence of one of the deprecated keywords by its new version. These modifications are not by themselves interesting and are not included in the following. WG14 members are invited to inspect them on the VC system, if they want, they are in the branch “keywords”.

The following appendix lists the non-trivial changes:

— Changes to the “Keywords” clause 6.4.1, where we replace the keywords themselves (p1) and add provisions to have the new ones as macro names (p2) and establish the old keywords as alternative spellings (p4).

— Addition of a new clause 6.4.4.5 “Predefined constants” that implement the constants `false` and `true`, and that is anchored in 6.4.4 “Constants”.

— Addition of text to 6.10.8.1 “Mandatory macros” that lists the new keywords.

— Modifications of the corresponding library clauses (7.2, 7.15, 7.18, and 7.26).

— Mark `<stdalign.h>` and `<stdbool.h>` to be obsolescent inside their specific text and in clause 7.13 “Future library directions”.

— Update Annex A.
Appendix: pages with diffmarks of the proposed changes against the May 2019 working draft.
The following page numbers are from the particular snapshot and may vary once the changes are integrated.
6.4.1 Keywords

Syntax

1 keyword: one of

alignas  extern  sizeof       _Alignas
alignof  false   static      _Atomic
auto     float    static_assert _Bool
bool     for      struct      _Complex
break    goto     switch      _Decimal128
case     if       _thread_local _Decimal32
char     inline   true       _Decimal64
cost     int      typedef     _Generic
continue long     union       _Imaginary
default  register unsigned _Noreturn
do       restrict void _Static_assert
double   return   volatile _Thread_local
else      short    while
enum

Constraints

2 The keywords

alignas  bool  static_assert true
alignof  false _thread_local

are also predefined macro names (6.10.8). None of these shall be the subject of a #define or a
#undef preprocessing directive and their spelling inside expressions that are subject to the # and
## preprocessing operators is unspecified.\(^{74}\)

Semantics

3 The above tokens (case sensitive) are reserved (in translation phases 7 and 8) for use as keywords
except in an attribute token, and shall not be used otherwise. The keyword _Imaginary is reserved
for specifying imaginary types.\(^{75}\)

4 The following table provides alternate spellings for certain keywords. These can be used wherever
the keyword can.\(^{76}\)

<table>
<thead>
<tr>
<th>keyword</th>
<th>alternative spelling</th>
</tr>
</thead>
<tbody>
<tr>
<td>alignas</td>
<td>_Alignas</td>
</tr>
<tr>
<td>alignof</td>
<td>_Alignof</td>
</tr>
<tr>
<td>bool</td>
<td>_Bool</td>
</tr>
<tr>
<td>static_assert</td>
<td>_Static_assert</td>
</tr>
<tr>
<td>_thread_local</td>
<td>_Thread_local</td>
</tr>
</tbody>
</table>

6.4.2 Identifiers

6.4.2.1 General

Syntax

1 identifier:

identifier-nondigit
identifier identifier-nondigit
identifier digit

\(^{74}\)The intent of these specifications is to allow but not to force the implementation of the corresponidg feature by means of a predefined macro.

\(^{75}\)One possible specification for imaginary types appears in Annex G.

\(^{76}\)These alternative keywords are obsolescent features and should not be used for new code.
6.4.4 Constants

Syntax

constant:
  integer-constant
  floating-constant
  enumeration-constant
  character-constant
  predefined-constant

Constraints

Each constant shall have a type and the value of a constant shall be in the range of representable values for its type.

Semantics

Each constant has a type, determined by its form and value, as detailed later.

6.4.4.1 Integer constants

Syntax

integer-constant:
  decimal-constant integer-suffix opt
  octal-constant integer-suffix opt
  hexadecimal-constant integer-suffix opt

  decimal-constant:
    nonzero-digit
    decimal-constant digit

  octal-constant:
    0
    octal-constant octal-digit

  hexadecimal-constant:
    hexadecimal-prefix hexadecimal-digit
    hexadecimal-constant hexadecimal-digit

hexadecimal-prefix: one of
  0x 0X

nonzero-digit: one of
  1 2 3 4 5 6 7 8 9

octal-digit: one of
  0 1 2 3 4 5 6 7

hexadecimal-digit: one of
  0 1 2 3 4 5 6 7 8 9
  a b c d e f
  A B C D E F

integer-suffix:
  unsigned-suffix long-suffix opt
  unsigned-suffix long-long-suffix
  long-suffix unsigned-suffix opt
  long-long-suffix unsigned-suffix opt
Forward references: common definitions `<stdbool.h>` (7.19), the `mbtowc` function (7.22.7.2), Unicode utilities `<uchar.h>` (7.28).

6.4.4.5 Predefined constants

Syntax

```
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

The keywords `false` and `true` represent constants of type `int` that are suitable for use as are integer literals. Their values are 0 for `false` and 1 for `true`.\(^{86}\)

6.4.5 String literals

Syntax

```
string-literal:
   encoding-prefixopt " s-char-sequenceopt "

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`.

4 The same considerations apply to each element of the sequence in a string literal as if it were in an integer character constant (for a character or UTF–8 string literal) or a wide character constant (for a wide string literal), except that the single-quote ‘ is representable either by itself or by the escape sequence `\`, but the double-quote ” shall be represented by the escape sequence `\"`.

\(^{86}\)Thus, the keywords `false` and `true` are usable in preprocessor directives.
6.10.8 Predefined macro names

1 The values of the predefined macros listed in the following subclauses⁹¹) (except for \texttt{__FILE__} and \texttt{__LINE__}) remain constant throughout the translation unit.

2 None of these macro names, nor the identifier \texttt{defined}, shall be the subject of a \texttt{#define} or a \texttt{#undef} preprocessing directive. Any other predefined macro names shall begin with a leading underscore followed by a uppercase letter or a second underscore.

3 The implementation shall not redefine the macro \texttt{__cplusplus}, nor shall it define it in any standard header.

Forward references: standard headers (7.1.2).

6.10.8.1 Mandatory macros

The in addition to the keywords

\begin{verbatim}
alignas bool static_assert true
alignof false thread_local
\end{verbatim}

which are object-like macros that expand to unspecified tokens, the following macro names shall be defined by the implementation:

\texttt{__DATE__} The date of translation of the preprocessing translation unit: a character string literal of the form "Mmm dd yyyy", where the names of the months are the same as those generated by the \texttt{asctime} function, and the first character of \texttt{dd} is a space character if the value is less than 10. If the date of translation is not available, an implementation-defined valid date shall be supplied.

\texttt{__FILE__} The presumed name of the current source file (a character string literal).\textsuperscript{192)}

\texttt{__LINE__} The presumed line number (within the current source file) of the current source line (an integer constant).\textsuperscript{192)}

\texttt{__STDC__} The integer constant 1, intended to indicate a conforming implementation.

\texttt{__STDC_ISO_10646__} The integer constant 1 if the implementation is a hosted implementation or the integer constant 0 if it is not.

\texttt{__STDC_VERSION__} The integer constant \texttt{yyymmL}.\textsuperscript{193)}

\texttt{__TIME__} The time of translation of the preprocessing translation unit: a character string literal of the form "hh:mm:ss" as in the time generated by the \texttt{asctime} function. If the time of translation is not available, an implementation-defined valid time shall be supplied.

Forward references: the \texttt{asctime} function (7.27.3.1).

6.10.8.2 Environment macros

The following macro names are conditionally defined by the implementation:

\texttt{__STDC_ISO_10646__} An integer constant of the form \texttt{yyymmL} (for example, 199712L). If this symbol is defined, then every character in the Unicode required set, when stored in an object of type \texttt{wchar_t}, has the same value as the short identifier of that character. The Unicode required set consists of all the characters that are defined by ISO/IEC 10646, along with all amendments and technical corrigenda, as of the specified year and month. If some other encoding is used, the macro shall not be defined and the actual encoding used is implementation-defined.

\textsuperscript{191)}See “future language directions” (6.11.9).

\textsuperscript{192)}The presumed source file name and line number can be changed by the \texttt{#line} directive.

\textsuperscript{193)}See Annex \texttt{M} for the values in previous revisions. The intention is that this will remain an integer constant of type \texttt{long int} that is increased with each revision of this document.
7.2 Diagnostics <assert.h>

The header <assert.h> defines the `assert` and `static_assert` macros and refers to another macro, `NDEBUG` which is not defined by <assert.h>. If `NDEBUG` is defined as a macro name at the point in the source file where <assert.h> is included, the `assert` macro is defined simply as

```c
#define assert(IGNORE) ((void)0)
```

The `assert` macro is redefined according to the current state of `NDEBUG` each time that <assert.h> is included.

2 The `assert` macro shall be implemented as a macro, not as an actual function. If the macro definition is suppressed in order to access an actual function, the behavior is undefined.

The macro expands to `_Static_assert`.

7.2.1 Program diagnostics

7.2.1.1 The assert macro

Synopsis

```c
#include <assert.h>
void assert(scalar expression);
```

Description

2 The `assert` macro puts diagnostic tests into programs; it expands to a void expression. When it is executed, if `expression` (which shall have a scalar type) is false (that is, compares equal to 0), the `assert` macro writes information about the particular call that failed (including the text of the argument, the name of the source file, the source line number, and the name of the enclosing function — the latter are respectively the values of the preprocessing macros `__FILE__` and `__LINE__` and of the identifier `__func__`) on the standard error stream in an implementation-defined format.\(^{207}\) It then calls the `abort` function.

Returns

3 The `assert` macro returns no value.

Forward references: the `abort` function (7.22.4.1).

---

\(^{207}\) The message written might be of the form:

```
Assertion failed: expression, function abc, file xyz, line nnn.
```
7.15 Alignment `<stdlib.h>`

The header defines four macros:

1. The obsolescent header `<stdlib.h>` defines two macros that are suitable for use in `#if` preprocessing directives. They are

   ```
   __alignas_is_defined
   ```

   and

   ```
   __alignof_is_defined
   ```

   which both expand to `true`.  


7.18 Boolean type and values `<stdbool.h>`

The *obsolete* header `<stdbool.h>` defines four macros.

Notwithstanding the provisions of 7.1.3, a program may undefine and perhaps then redefine the macros bool the following macro which is suitable for use in conditional preprocessing directives:

```c
__bool_true_false_are_defined
```

It expands to the constant `true`, `true`, and `false`. 
7.26 Threads <threads.h>

7.26.1 Introduction

The header <threads.h> includes the header <time.h>, defines macros, and declares types, enumeration constants, and functions that support multiple threads of execution. Implementations that define the macro __STDC_NO_THREADS__ need not provide this header nor support any of its facilities.

The header <threads.h> includes the header <time.h>, defines macros, and declares types, enumeration constants, and functions that support multiple threads of execution. 

> which expands to the keyword Thread_local. The macros are

```
ONCE_FLAG_INIT
```

which expands to a value that can be used to initialize an object of type once_flag; and

```
TSS_DTOR_ITERATIONS
```

which expands to an integer constant expression representing the maximum number of times that destructors will be called when a thread terminates.

The types are

```
cnd_t
```

which is a complete object type that holds an identifier for a condition variable;

```
thrd_t
```

which is a complete object type that holds an identifier for a thread;

```
tss_t
```

which is a complete object type that holds an identifier for a thread-specific storage pointer;

```
mtx_t
```

which is a complete object type that holds an identifier for a mutex;

```
tss_dtor_t
```

which is the function pointer type void (*) (void*), used for a destructor for a thread-specific storage pointer;

```
thrd_start_t
```

which is the function pointer type int (*) (void*) that is passed to thrd_create to create a new thread; and

```
once_flag
```

which is a complete object type that holds a flag for use by call_once.

The enumeration constants are

```
mtx_plain
```

which is passed to mtx_init to create a mutex object that does not support timeout;

```
mtx_recursive
```

See “future library directions” (7.31.18).
7.31.10 Alignment `<stdalign.h>`
The header `<stdalign.h>` together with its defined macros `__alignas_is_defined` and `__alignas_is_defined` is an obsolescent feature.

7.31.11 Atomics `<stdatomic.h>`
Macros that begin with `ATOMIC_` and an uppercase letter may be added to the macros defined in the `<stdatomic.h>` header. Typedef names that begin with either `atomic_` or `memory_`, and a lowercase letter may be added to the declarations in the `<stdatomic.h>` header. Enumeration constants that begin with `memory_order_` and a lowercase letter may be added to the definition of the `memory_order` type in the `<stdatomic.h>` header. Function names that begin with `atomic_` and a lowercase letter may be added to the declarations in the `<stdatomic.h>` header.

2 The macro `ATOMIC_VAR_INIT` is an obsolescent feature.

7.31.12 Boolean type and values `<stdbool.h>`
The ability to undefine and perhaps then redefine the macros `bool`, `true`, and `false` header `<stdbool.h>` together with its defined macro `__bool_true_false_are_defined` is an obsolescent feature.

7.31.13 Integer types `<stdint.h>`
Typedef names beginning with `int` or `uint` and ending with `_t` may be added to the types defined in the `<stdint.h>` header. Macro names beginning with `INT` or `UINT` and ending with `_MAX`, `_MIN`, `_WIDTH`, or `_C` may be added to the macros defined in the `<stdint.h>` header.

7.31.14 Input/output `<stdio.h>`
Lowercase letters may be added to the conversion specifiers and length modifiers in `fprintf` and `fscanf`. Other characters may be used in extensions.

2 The use of `ungetc` on a binary stream where the file position indicator is zero prior to the call is an obsolescent feature.

7.31.15 General utilities `<stdlib.h>`
Function names that begin with `str` or `wcs` and a lowercase letter may be added to the declarations in the `<stdlib.h>` header.

2 Invoking `realloc` with a `size` argument equal to zero is an obsolescent feature.

7.31.16 String handling `<string.h>`
Function names that begin with `str`, `mem`, or `wcs` and a lowercase letter may be added to the declarations in the `<string.h>` header.

7.31.17 Date and time `<time.h>`
Macros beginning with `TIME_` and an uppercase letter may be added to the macros in the `<time.h>` header.

7.31.18 Threads `<threads.h>`
Function names, type names, and enumeration constants that begin with either `cnd_`, `mtx_`, `thrd_`, or `tss_`, and a lowercase letter may be added to the declarations in the `<threads.h>` header.

7.31.19 Extended multibyte and wide character utilities `<wchar.h>`
Function names that begin with `wcs` and a lowercase letter may be added to the declarations in the `<wchar.h>` header.

2 Lowercase letters may be added to the conversion specifiers and length modifiers in `fwprintf` and `fwscanf`. Other characters may be used in extensions.
Annex A
(informative)
Language syntax summary

A.1 Lexical grammar

A.1.1 Lexical elements

(6.4) token:
keyword
identifier
constant
string-literal
punctuator

(6.4) preprocessing-token:
header-name
identifier
pp-number
character-constant
string-literal
punctuator
each non-white-space character that cannot be one of the above

A.1.2 Keywords

(6.4.1) keyword: one of
alignas     extern     sizeof     __Alignas__
alignof     false      static     __Atomic__
auto        float      thread_local __Complex__
break       for        true       __Decimal128
case        goto       switch     __Decimal32
char        inline     typedef     __Decimal64
const       int        __Generic__
continue    long       union      __Imaginary
default     register   unsigned   __Noreturn
do          restrict   void       __Static_assert__
double      return     volatile   __Thread_local__
else        short      while
enum

A.1.3 Identifiers

(6.4.2.1) identifier:
identifier-nondigit
identifier  identifier-nondigit
identifier  digit

(6.4.2.1) identifier-nondigit:
nondigit
universal-character-name
other implementation-defined characters

NOTE The notation is described in 6.1.
(6.4.2.1) **nondigit**: one of

\[
\begin{align*}
& a b c d e f g h i j k l m \\
& n o p q r s t u v w x y z \\
& A B C D E F G H I J K L M \\
& N O P Q R S T U V W X Y Z
\end{align*}
\]

(6.4.2.1) **digit**: one of

\[
0 1 2 3 4 5 6 7 8 9
\]

### A.1.4 Universal character names

(6.4.3) **universal-character-name**: 

\[
\textbackslash u \text{ hex-quad} \\
\textbackslash U \text{ hex-quad hex-quad}
\]

(6.4.3) **hex-quad**: 

\[
\text{hexadecimal-digit} \ \text{hexadecimal-digit} \ \text{hexadecimal-digit} \ \text{hexadecimal-digit}
\]

### A.1.5 Constants

(6.4.4) **constant**: 

\[
\text{integer-constant} \\
\text{floating-constant} \\
\text{enumeration-constant} \\
\text{character-constant} \\
\text{predefined-constant}
\]

(6.4.4.1) **integer-constant**: 

\[
\text{decimal-constant} \ \text{integer-suffix}_{\text{opt}} \\
\text{octal-constant} \ \text{integer-suffix}_{\text{opt}} \\
\text{hexadecimal-constant} \ \text{integer-suffix}_{\text{opt}}
\]

(6.4.4.1) **decimal-constant**: 

\[
\text{nonzero-digit} \\
\text{decimal-constant} \ \text{digit}
\]

(6.4.4.1) **octal-constant**: 

\[
0 \\
\text{octal-constant} \ \text{octal-digit}
\]

(6.4.4.1) **hexadecimal-constant**: 

\[
\text{hexadecimal-prefix} \ \text{hexadecimal-digit} \\
\text{hexadecimal-constant} \ \text{hexadecimal-digit}
\]

(6.4.4.1) **hexadecimal-prefix**: one of 

\[
0x \ 0X
\]

(6.4.4.1) **nonzero-digit**: one of 

\[
1 2 3 4 5 6 7 8 9
\]

(6.4.4.1) **octal-digit**: one of 

\[
0 1 2 3 4 5 6 7
\]

(6.4.4.1) **hexadecimal-digit**: one of 

\[
0 1 2 3 4 5 6 7 8 9 \\
a b c d e f \\
A B C D E F
\]
(6.4.4.3) enumeration-constant:
   identifier

(6.4.4.4) character-constant:
   ' c-char-sequence '
   L' c-char-sequence '  
   u' c-char-sequence '  
   U' c-char-sequence '  

(6.4.4.4) c-char-sequence:
   c-char
   c-char-sequence c-char

(6.4.4.4) c-char:
   any member of the source character set except
   the single-quote ', backslash \, or new-line character
   escape-sequence

(6.4.4.4) escape-sequence:
   simple-escape-sequence
   octal-escape-sequence
   hexadecimal-escape-sequence
   universal-character-name

(6.4.4.4) simple-escape-sequence: one of
   \ " \? \\  
   \a \b \f \n \r \t \v

(6.4.4.4) octal-escape-sequence:
   \ octal-digit
   \ octal-digit octal-digit
   \ octal-digit octal-digit octal-digit

(6.4.4.4) hexadecimal-escape-sequence:
   \x hexadecimal-digit
   hexadecimal-escape-sequence hexadecimal-digit

A.1.5.1 Predefined constants

(6.4.4.5) predefined-constant:
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~ false
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~ true

A.1.6 String literals

(6.4.5) string-literal:
   encoding-prefixopt " s-char-sequenceopt "

(6.4.5) encoding-prefix:
   u8
   u
   U
   L

(6.4.5) s-char-sequence:
   s-char
   s-char-sequence s-char