Revise spelling of keywords v4
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 been 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 the primary spelling.

This is a revision of papers to N2368 and N2392 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.

**Changes in v3:**

— Remove the requirement for implementations to have these keywords as macro names and adapt title and contents accordingly.
— Update Annex B.

**Changes in v4:**

— Move the changes for `false` and `true` to paper N2458.

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   imaginary   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
alignof, static_assert

The new keywords `false` and `true` also found consensus, but their possible use in the preprocessor needs more provisions than given here. They are thus moved to N2458.

Other proposals will follow that will tackle other parts of N2368 and beyond:

- Handle `false` and `true` and make them of type `bool`.
- Make `noreturn` a keyword or replace it by an attribute.
- Introduce `nullptr` and `nullptr_t`.
- 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:

- Allow for the keywords to also be macros, such that implementations may have an easy transition.
- 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.

- 5 predefined macros (7 when adding `false` and `true`) 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. 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 alignas _Alignas
#define alignof _Alignof
```
At the other end of the spectrum, an implementation that implements all new keywords as first-class constructs and also wants to provide them as macros (though they don’t have to) can simply have definitions that are the token identity:

```
#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
```

4. 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).
— A new subclause to 6.10.8.4 “Optional macros” that lists the new keywords that may also be macros.
— Modifications of the corresponding library clauses (7.2, 7.15, 7.18, and 7.26).
— Mark `<stdalign.h>` and `<stdbool.h>` with the changes in N2458 to be obsolescent inside their specific text and in clause 7.31 “Future library directions”.
— Update Annex A.

Appendix: pages with diffmarks of the proposed changes against the September 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

<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>auto</td>
<td>_Atomic</td>
</tr>
<tr>
<td>bool</td>
<td>_Bool</td>
</tr>
<tr>
<td>break</td>
<td>_Complex</td>
</tr>
<tr>
<td>case</td>
<td>_Decimal128</td>
</tr>
<tr>
<td>char</td>
<td>_Decimal32</td>
</tr>
<tr>
<td>const</td>
<td>_Decimal64</td>
</tr>
<tr>
<td>continue</td>
<td>_Generic</td>
</tr>
<tr>
<td>default</td>
<td>_Imaginary</td>
</tr>
<tr>
<td>do</td>
<td>_Noreturn</td>
</tr>
<tr>
<td>double</td>
<td>_Static_assert</td>
</tr>
<tr>
<td>else</td>
<td>_Thread_local</td>
</tr>
</tbody>
</table>

Constraints

2 The keywords

alignas alignof bool static_assert thread_local

may optionally be predefined macro names (??). None of these shall be the subject of a #define or a #undef preprocessing directive.

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.74

4 The following table provides alternate spellings for certain keywords. These can be used wherever the keyword can.75

<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>

Their spelling inside expressions that are subject to the # and ## preprocessing operators is unspecified.76

6.4.2 Identifiers

6.4.2.1 General

Syntax

1 identifier: identifier-nondigit

identifier identifier-nondigit

identifier digit

---

74 One possible specification for imaginary types appears in Annex G.

75 These alternative keywords are obsolescent features and should not be used for new code.

76 The intent of these specifications is to allow but not to force the implementation of the corresponding feature by means of a predefined macro.
An implementation that defines \texttt{\_\_STDC\_NO\_COMPLEX\_} shall not define \texttt{\_\_STDC\_IEC\_60559\_COMPLEX\_} or \texttt{\_\_STDC\_IEC\_559\_COMPLEX\_}.

6.10.8.4 Optional macros

The keywords

\begin{itemize}
  \item \texttt{alignas}
  \item \texttt{alignof}
  \item \texttt{bool}
  \item \texttt{static\_assert}
  \item \texttt{thread\_local}
\end{itemize}

optionally are also predefined macro names that expand to unspecified tokens.

6.10.9 Pragma operator

Semantics

A unary operator expression of the form:

\begin{verbatim}
  \_Pragma ( string-litera )
\end{verbatim}

is processed as follows: The string literal is destringized by deleting any encoding prefix, deleting the leading and trailing double-quotes, replacing each escape sequence \texttt{"} by a double-quote, and replacing each escape sequence \texttt{\ \} by a single backslash. The resulting sequence of characters is processed through translation phase 3 to produce preprocessing tokens that are executed as if they were the \textit{pp-tokens} in a pragma directive. The original four preprocessing tokens in the unary operator expression are removed.

EXAMPLE A directive of the form:

\begin{verbatim}
  \#pragma listing on "..\listing.dir"
\end{verbatim}

can also be expressed as:

\begin{verbatim}
  \_Pragma ("listing on "..\listing.dir"")
\end{verbatim}

The latter form is processed in the same way whether it appears literally as shown, or results from macro replacement, as in:

\begin{verbatim}
  \#define LISTING(x) PRAGMA(listing on #x)
  \#define PRAGMA(x) \_Pragma(#x)
  LISTING (..\listing.dir)
\end{verbatim}
7.2 Diagnostics <assert.h>

1 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

```
#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

```
#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. It then calls the abort function.

Returns

3 The assert macro returns no value.

Forward references: the abort function (7.22.4.1).

---

205) The message written might be of the form:

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

The header defines four macros:

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

```c
__alignas_is_defined
```

and

```c
__alignof_is_defined
```

which both expand to the integer constant 1.
7.18 Boolean type and values <stdbool.h>

1 The header <stdbool.h> defines four macros that expand to _Bool. These three macros that are suitable for use in #if preprocessing directives. They are:

- `true` which expands to the integer constant 1,
- `false` which expands to the integer constant 0, and
- `__bool_true_false_are_defined` which expands to the integer constant 1.

2 Notwithstanding the provisions of 7.1.3, a program may undefine and perhaps then redefine the macros `bool`, `true`, and `false`.\footnote{See “future library directions” (7.31.11).}
### 7.26 Threads <threads.h>

#### 7.26.1 Introduction

1. The header <threads.h> includes the header <time.h>, defines macros, and declares types, enumeration constants, and functions that support multiple threads of execution.\(^{338}\)

2. Implementations that define the macro `__STDC_NO_THREADS__` need not provide this header nor support any of its facilities.

which expands to the keyword `Thread_local`. The macros are

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ONCE_FLAG_INIT</code></td>
<td>which expands to a value that can be used to initialize an object of type <code>once_flag</code>; and</td>
</tr>
<tr>
<td><code>TSS_DTOR_ITERATIONS</code></td>
<td>which expands to an integer constant expression representing the maximum number of times that destructors will be called when a thread terminates.</td>
</tr>
</tbody>
</table>

The types are

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cnd_t</code></td>
<td>which is a complete object type that holds an identifier for a condition variable;</td>
</tr>
<tr>
<td><code>thrd_t</code></td>
<td>which is a complete object type that holds an identifier for a thread;</td>
</tr>
<tr>
<td><code>tss_t</code></td>
<td>which is a complete object type that holds an identifier for a thread-specific storage pointer;</td>
</tr>
<tr>
<td><code>mtx_t</code></td>
<td>which is a complete object type that holds an identifier for a mutex;</td>
</tr>
<tr>
<td><code>tss_dtor_t</code></td>
<td>which is the function pointer type <code>void (*)(void*)</code>, used for a destructor for a thread-specific storage pointer;</td>
</tr>
<tr>
<td><code>thrd_start_t</code></td>
<td>which is the function pointer type <code>int (*)(void*)</code> that is passed to <code>thrd_create</code> to create a new thread; and</td>
</tr>
<tr>
<td><code>once_flag</code></td>
<td>which is a complete object type that holds a flag for use by <code>call_once</code>.</td>
</tr>
</tbody>
</table>

The enumeration constants are

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mtx_plain</code></td>
<td>which is passed to <code>mtx_init</code> to create a mutex object that does not support timeout;</td>
</tr>
</tbody>
</table>

\(^{338}\) See “future library directions” (7.31.17).
and the same names suffixed with \f, \l, \d32, \d64, or \d128 may be added to the <math.h> header. The \cr prefix is intended to indicate a correctly rounded version of the function.

7.31.9 Signal handling <signal.h>

Macros that begin with either \SIG and an uppercase letter or \SIG_ and an uppercase letter may be added to the macros defined in the <signal.h> header.

7.31.10 Alignment <stdalign.h>

The \_header <stdalign.h> together with its defined macros \_alignas_is_defined and \_alignis_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 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.
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  enum  signed  __Alignas__
alignof  extern sizeof  __Alignof__
auto     float  static  __Atomic
break    for    static_assert __Bool__
case     goto   struct  __Complex
char     inline thread_local __Decimal128
const    int    typedef  __Decimal64
continue long  union  __Generic
default  register unsigned __Imaginary
do      restrict void  __Noreturn
double  return volatile __Static_assert
else    short  while  __Thread_local
```

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
(6.5.1.1) generic-selection:
_Generic ( assignment-expression , generic-assoc-list )

(6.5.1.1) generic-assoc-list:
generic-association
generic-assoc-list , generic-association

(6.5.1.1) generic-association:
type-name : assignment-expression
default : assignment-expression

(6.5.2) postfix-expression:
primary-expression
postfix-expression [ expression ]
postfix-expression ( argument-expression-list_opt )
postfix-expression . identifier
postfix-expression -> identifier
postfix-expression ++
postfix-expression --
( type-name ) { initializer-list }
( type-name ) { initializer-list , }

(6.5.2) argument-expression-list:
assignment-expression
argument-expression-list , assignment-expression

(6.5.3) unary-expression:
postfix-expression
++ unary-expression
-- unary-expression
unary-operator cast-expression
sizeof unary-expression
sizeof ( type-name )
_alignof _Alignof ( type-name )

(6.5.3) unary-operator: one of
& * + - ~ !

(6.5.4) cast-expression:
unary-expression
( type-name ) cast-expression

(6.5.5) multiplicative-expression:
cast-expression
multiplicative-expression * cast-expression
multiplicative-expression / cast-expression
multiplicative-expression % cast-expression

(6.5.6) additive-expression:
multiplicative-expression
additive-expression + multiplicative-expression
additive-expression - multiplicative-expression

(6.5.7) shift-expression:
additive-expression
shift-expression << additive-expression
shift-expression >> additive-expression
(6.7) **init-declarator-list:**
- **init-declarator**
  - **init-declarator-list**, **init-declarator**

(6.7) **init-declarator:**
- **declarator**
  - **declarator = initializer**

(6.7) **attribute-declaration:**
- **attribute-specifier-sequence**

(6.7.1) **storage-class-specifier:**
- **typedef**
- **extern**
- **static**
- **Thread Local**
- **auto**
- **register**

(6.7.2) **type-specifier:**
- **void**
- **char**
- **short**
- **int**
- **long**
- **float**
- **double**
- **signed**
- **unsigned**
- **bool**
- **Complex**
- **Decimal32**
- **Decimal64**
- **Decimal128**
- **atomic-type-specifier**
- **struct-or-union-specifier**
- **enum-specifier**
- **typedef-name**

(6.7.2.1) **struct-or-union-specifier:**
- **struct-or-union attribute-specifier-sequence**, **identifier**

(6.7.2.1) **struct-or-union:**
- **struct**
- **union**

(6.7.2.1) **member-declaration-list:**
- **member-declaration**
  - **member-declaration-list member-declaration**

(6.7.2.1) **member-declaration:**
- **attribute-specifier-sequence**, **specifier-qualifier-list member-declarator-list**
  - **static assert-declaration**

(6.7.2.1) **specifier-qualifier-list:**
- **type-specifier-qualifier attribute-specifier-sequence**
- **type-specifier-qualifier specifier-qualifier-list**

(6.7.2.1) **type-specifier-qualifier:**
- **type-specifier**
  - **type-qualifier**
  - **alignment-specifier**
(6.7.2.1) member-declarator-list:
    member-declarator
    member-declarator-list, member-declarator

(6.7.2.1) member-declarator:
    declarator
    declarator_opt : constant-expression

(6.7.2.2) enum-specifier:
    enum attribute-specifier-sequence_opt identifier_opt { enumerator-list }
    enum attribute-specifier-sequence_opt identifier_opt { enumerator-list, }
    enum identifier

(6.7.2.2) enumerator-list:
    enumerator
    enumerator-list, enumerator

(6.7.2.2) enumerator:
    enumeration-constant attribute-specifier-sequence_opt
    enumeration-constant attribute-specifier-sequence_opt = constant-expression

(6.7.2.4) atomic-type-specifier:
    _Atomic ( type-name )

(6.7.3) type-qualifier:
    const
    restrict
    volatile
    _Atomic

(6.7.4) function-specifier:
    inline
    _Noreturn

(6.7.5) alignment-specifier:
    _Alignas _Alignas ( type-name )
    _Alignas _Alignas ( constant-expression )

(6.7.6) declarator:
    pointer_opt direct-declarator

(6.7.6) direct-declarator:
    identifier attribute-specifier-sequence_opt
    ( declarator )
    array-declarator attribute-specifier-sequence_opt
    function-declarator attribute-specifier-sequence_opt

(6.7.6) array-declarator:
    direct-declarator [ type-qualifier-list_opt assignment-expression_opt ]
    direct-declarator [ static type-qualifier-list_opt assignment-expression ]
    direct-declarator [ type-qualifier-list static assignment-expression ]
    direct-declarator [ type-qualifier-list_opt * ]

(6.7.6) function-declarator:
    direct-declarator ( parameter-type-list_opt )

(6.7.6) pointer:
    * attribute-specifier-sequence_opt type-qualifier-list_opt
    * attribute-specifier-sequence_opt type-qualifier-list_opt pointer

(6.7.6) type-qualifier-list:
    type-qualifier
    type-qualifier-list type-qualifier

(6.7.6) parameter-type-list:
    parameter-list
    parameter-list , ...
(6.7.6) parameter-list:
   parameter-declaration
   parameter-list , parameter-declaration

(6.7.6) parameter-declaration:
   attribute-specifier-sequence opt declaration-specifiers declarator
   attribute-specifier-sequence opt declaration-specifiers abstract-declarator opt

(6.7.7) type-name:
   specifier-qualifier-list abstract-declarator opt

(6.7.7) abstract-declarator:
   pointer
   pointer opt direct-abstract-declarator

(6.7.7) direct-abstract-declarator:
   ( abstract-declarator )
   array-abstract-declarator attribute-specifier-sequence opt
   function-abstract-declarator attribute-specifier-sequence opt

(6.7.7) array-abstract-declarator:
   direct-abstract-declarator opt [ type-qualifier-list opt assignment-expression opt ]
   direct-abstract-declarator opt [ static type-qualifier-list opt assignment-expression ]
   direct-abstract-declarator opt [ type-qualifier-list static assignment-expression ]
   direct-abstract-declarator opt [ * ]

(6.7.7) function-abstract-declarator:
   direct-abstract-declarator opt ( parameter-type-list opt )

(6.7.8) typedef-name:
   identifier

(6.7.9) initializer:
   assignment-expression
   { initializer-list }
   { initializer-list , }

(6.7.9) initializer-list:
   designation opt initializer
   initializer-list , designation opt initializer

(6.7.9) designation:
   designator-list =

(6.7.9) designator-list:
   designator
   designator-list designator

(6.7.9) designator:
   [ constant-expression ]
   . identifier

(6.7.10) static_assert-declaration:
   _Static_assert static_assert ( constant-expression , string-literal ) ;
   _Static_assert static_assert ( constant-expression ) ;

(6.7.11.1) attribute-specifier-sequence:
   attribute-specifier-sequence opt attribute-specifier

(6.7.11.1) attribute-specifier:
   [ [ attribute-list ] ]

(6.7.11.1) attribute-list:
   attribute opt
   attribute-list , attribute opt

(6.7.11.1) attribute:
   attribute-token attribute-argument-clause opt