Revise spelling of keywords

Jens Gustedt

ISO/IEC JTC 1/SC 22/WG14 **N2457**

November 18, 2019 **v 4**

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 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 revsion 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**	**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
```

```
#define bool          _Bool
#define static_assert _Static_assert
#define thread_local  _Thread_local
```

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

alignas	**enum**	**signed**	~~**_Alignas**~~
alignof	**extern**	**sizeof**	~~**_Alignof**~~
auto	**float**	**static**	**_Atomic**
bool	**for**	**static_assert**	~~**_Bool**~~
break	**goto**	**struct**	**_Complex**
case	**if**	**switch**	**_Decimal128**
char	**inline**	**thread_local**	**_Decimal32**
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**~~

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)]

keyword	alternative spelling
alignas	**_Alignas**
alignof	**_Alignof**
bool	**_Bool**
static_assert	**_Static_assert**
thread_local	**_Thread_local**

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 correspondig feature by means of a predefined macro.

2 An implementation that defines **__STDC_NO_COMPLEX__** shall not define **__STDC_IEC_60559_COMPLEX__** or **__STDC_IEC_559_COMPLEX__**.

6.10.8.4 Optional macros

1 The keywords

alignas **alignof** **bool** **static_assert** **thread_local**

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

6.10.9 Pragma operator

Semantics

1 A unary operator expression of the form:

_Pragma (*string-literal* **)**

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 \" by a double-quote, and replacing each escape sequence \\ 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 *pp-tokens* in a pragma directive. The original four preprocessing tokens in the unary operator expression are removed.

2 **EXAMPLE** A directive of the form:

```
#pragma listing on "..\listing.dir"
```

can also be expressed as:

```
_Pragma ("listing on \"..\\listing.dir\"")
```

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

```
#define LISTING(x) PRAGMA(listing on #x)
#define PRAGMA(x)  _Pragma(#x)

LISTING (..\listing.dir)
```

7.2 Diagnostics <assert.h>

1 The header <assert.h> defines the **assert** ~~and **static_assert** macros~~ macro 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

1
```
#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.[205] 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

```
__alignas_is_defined
```

and

```
__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 .~~

~~expands to **_Bool**.~~

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**.[281)]

[281)] 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

```
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.

4 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.

5 The enumeration constants are

```
mtx_plain
```

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

```
mtx_recursive
```

338) See "future library directions" (7.31.17).

cracosh	**cratanh**	**crexp10**	**crlog1p**	**crrootn**
cracospi	**cratanpi**	**crexp2m1**	**crlog2p1**	**crrsqrt**
cracos	**cratan**	**crexp2**	**crlog2**	**crsinh**
crasinh	**crcompoundn**	**crexpm1**	**crlogp1**	**crsinpi**
crasinpi	**crcosh**	**crexp**	**crlog**	**crsin**
crasin	**crcospi**	**crhypot**	**crpown**	**crtanh**
cratan2pi	**crcos**	**crlog10p1**	**crpowr**	**crtanpi**
cratan2	**crexp10m1**	**crlog10**	**crpow**	**crtan**

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>

1 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>

1 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>

1 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>

1 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>

1 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>

1 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>

1 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>

1 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

1 **NOTE** The notation is described in 6.1.

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**
bool	**for**	**static_assert**	~~**_Bool**~~
break	**goto**	**struct**	**_Complex**
case	**if**	**switch**	**_Decimal128**
char	**inline**	**thread_local**	**_Decimal32**
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*$_{\text{opt}}$ **)**
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**~~ **thread_local**
auto
register
(6.7.2) *type-specifier:*
void
char
short
int
long
float
double
signed
unsigned
~~**_Bool**~~ **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*$_{\text{opt}}$ *identifier*$_{\text{opt}}$ **{** *member-declaration-list* **}**
struct-or-union *attribute-specifier-sequence*$_{\text{opt}}$ *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$_{\text{opt}}$ *specifier-qualifier-list* *member-declarator-list*$_{\text{opt}}$ **;**
static_assert-declaration
(6.7.2.1) *specifier-qualifier-list:*
type-specifier-qualifier *attribute-specifier-sequence*$_{\text{opt}}$
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$_{\text{opt}}$ **:** *constant-expression*
(6.7.2.2) *enum-specifier:*
enum *attribute-specifier-sequence*$_{\text{opt}}$ *identifier*$_{\text{opt}}$ **{** *enumerator-list* **}**
enum *attribute-specifier-sequence*$_{\text{opt}}$ *identifier*$_{\text{opt}}$ **{** *enumerator-list* **,** **}**
enum *identifier*
(6.7.2.2) *enumerator-list:*
enumerator
enumerator-list **,** *enumerator*
(6.7.2.2) *enumerator:*
enumeration-constant *attribute-specifier-sequence*$_{\text{opt}}$
enumeration-constant *attribute-specifier-sequence*$_{\text{opt}}$ **=** *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$_{\text{opt}}$ *direct-declarator*

(6.7.6) *direct-declarator:*
identifier *attribute-specifier-sequence*$_{\text{opt}}$
(*declarator* **)**
array-declarator *attribute-specifier-sequence*$_{\text{opt}}$
function-declarator *attribute-specifier-sequence*$_{\text{opt}}$

(6.7.6) *array-declarator:*
direct-declarator **[** *type-qualifier-list*$_{\text{opt}}$ *assignment-expression*$_{\text{opt}}$ **]**
direct-declarator **[** **static** *type-qualifier-list*$_{\text{opt}}$ *assignment-expression* **]**
direct-declarator **[** *type-qualifier-list* **static** *assignment-expression* **]**
direct-declarator **[** *type-qualifier-list*$_{\text{opt}}$ ***** **]**

(6.7.6) *function-declarator:*
direct-declarator **(** *parameter-type-list*$_{opt}$ **)**

(6.7.6) *pointer:*
***** *attribute-specifier-sequence*$_{\text{opt}}$ *type-qualifier-list*$_{\text{opt}}$
***** *attribute-specifier-sequence*$_{\text{opt}}$ *type-qualifier-list*$_{\text{opt}}$ *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$_{\text{opt}}$ *declaration-specifiers* *declarator*
attribute-specifier-sequence$_{\text{opt}}$ *declaration-specifiers* *abstract-declarator*$_{\text{opt}}$
(6.7.7) *type-name:*
specifier-qualifier-list *abstract-declarator*$_{\text{opt}}$
(6.7.7) *abstract-declarator:*
pointer
pointer$_{\text{opt}}$ *direct-abstract-declarator*
(6.7.7) *direct-abstract-declarator:*
(*abstract-declarator* **)**
array-abstract-declarator *attribute-specifier-sequence*$_{\text{opt}}$
function-abstract-declarator *attribute-specifier-sequence*$_{\text{opt}}$
(6.7.7) *array-abstract-declarator:*
direct-abstract-declarator$_{\text{opt}}$ **[** *type-qualifier-list*$_{\text{opt}}$ *assignment-expression*$_{\text{opt}}$ **]**
direct-abstract-declarator$_{\text{opt}}$ **[static** *type-qualifier-list*$_{\text{opt}}$ *assignment-expression* **]**
direct-abstract-declarator$_{\text{opt}}$ **[** *type-qualifier-list* **static** *assignment-expression* **]**
direct-abstract-declarator$_{\text{opt}}$ **[*]**

(6.7.7) *function-abstract-declarator:*
direct-abstract-declarator$_{\text{opt}}$ **(** *parameter-type-list*$_{\text{opt}}$ **)**
(6.7.8) *typedef-name:*
identifier

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

(6.7.9) *initializer-list:*
designation$_{\text{opt}}$ *initializer*
initializer-list **,** *designation*$_{\text{opt}}$ *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$_{\text{opt}}$ *attribute-specifier*
(6.7.11.1) *attribute-specifier:*
[[*attribute-list* **]]**
(6.7.11.1) *attribute-list:*
attribute$_{\text{opt}}$
attribute-list **,** *attribute*$_{\text{opt}}$
(6.7.11.1) *attribute:*
attribute-token *attribute-argument-clause*$_{\text{opt}}$