Remove support for function definitions with identifier lists

Lars Gullik Bjønnes, Jens Gustedt

ISO/IEC JTC 1/SC 22/WG14 **N2432**

v 1

September 25, 2019

Remove support for function definitions with identifier lists

proposal for C2x

Lars Gullik Bjønnes and Jens Gustedt
CISCO Inc., Norway and INRIA and ICube, Université de Strasbourg, France

Function definitions with identifier lists (often referred to as K&R functions) are deprecated for a long time. Because they are now clearly phased out everywhere and because function declarations and definitions that provide a prototypes are clearly superior we propose to remove this feature for good.

1. INTRODUCTION

The original syntax for definitions with identifier lists and a subsequent declarator list has the disadvantage that it does not provide a prototype of the defined function, and that the function needs a special convention (called *default argument promotion*) that leaves the verification of type correspondance between caller and callee to the programmer.
As an example take the following declaration and definition:

```
// header declaration, not a prototype
long maxl();

// identifier list definition, not a prototype
long maxl(a, b)
long a, b;
{
  return a > b ? a: b;
}
```

Here a caller of `maxl` would have to ensure that for any call the arguments are of type compatible with **`long`**, automatic conversion to **`long`** for narrower types such as **`int`** is not provided.
In addition, this syntax has caveats when used with parameters that are narrower than **`int`**, since such arguments undergo *default argument promotion*. Narrow integers are (usually) converted to **`int`**, and **`float`** to **`double`**.

```
// header declaration, not a prototype, valid
short maxs();

// prototype declaration, not compatible, invalid
short maxs(short a, short b);

// prototype declaration, compatible, valid
short maxs(int a, int b);

// identifier list definition, undergoing default argument
   promotion
short maxs(a, b)
short a, b;
{
  return a > b ? a: b;
}
```

This function is always called with **int** arguments. But because these **int** arguments then are immediately converted to **short** by the function, there is the additional tacit assumption that the values of these arguments fit into **short**. So calling such a function with value that exceeds **SHRT_MAX**, *e.g*, cannot be be caught by the caller.
The modern, prototyped form does not have these drawbacks. The following declaration and definition ensure that the function always receives correctly converted parameters:

```
// prototype declaration
long maxl(long a, long b);

// prototype definition
long maxl(long a, long b) {
  return a > b ? a: b;
}
```

And the equivalent code for **short** instead of **long** is also valid:

```
// prototype declaration
short maxs(short a, short b);

// prototype definition
short maxs(short a, short b) {
  return a > b ? a: b;
}
```

Therefore, the syntax of function definitions with identifier lists has been deprecated for a long time, and compilers such as gcc and clang issue warnings if they encounter such a construct, or even need specific compiler options to accept it.
There are other issues within the field of function syntax, namely

— declarations (that are not definitions) may omit parameter information completely, and,
— in contrast to pure declarations, definitions that provide a prototype, may not omit the parameter name, even if the parameter is not used in the function body.

For the latter, there is already paper N2381, proposing to remove that restriction. Though it seems that the underlying problem has similar roots as the issues at hand, that proposal is orthogonal to ours and we will not need to handle the issue, here.
The first, the property to provide declarations without prototypes, has also been marked obsolescent in the C standard, but still is relatively widely used. In particular, it seems that the possibility to declare function pointers with unknown parameter list is appreciated in parts of the C community for dealing with function call backs. Therefore, we will not attempt to remove this specific feature here, this would need much more preparation and discussion.
As a consequence, when removing *identifier lists* from function declarations and *declaration lists* from function definitions, we have to ensure that we still have syntax that allows for an empty parameter list.

2. REMOVING FUNCTION DEFINITIONS WITH IDENTIFIER LISTS

At many places the removal of the feature is straight forward, entire sentences or even paragraphs can be completely removed. These are found in 6.5.2.2 p6 (function call), 6.7.6.3 p3 (function declarations), and 6.9.1.5 p5 and p13 (function definitions), as referred in the appendix.

2.1. Syntax changes

The declaration syntax for functions with identifier lists can simply be removed from *direct declarator*:

> *Remove the last line from syntax entry* direct-declarator *in 6.7.6 p1.*

The depending syntax entry, *identifier list*, cannot be removed because it is also used for the syntax of **`#define`**. Therefore we propose to move that syntax entry to section 6.10 (preprocessing):

> *Move syntax entry* identifier-list *from 6.7.6 p1 to 6.10 p1.*

Since we want to keep the possibility of specifying an empty parameter list, we add that feature to the sole remaining syntax for function declarations.

> *Make the term* parameter-type-list *of the term* function-declarator *in 6.7.6 p1 optional.*

The definition syntax for functions with identifier lists can simply be removed:

> *Remove the* declaration-list *term from syntax entry* function-definition *in 6.9.1 p1.*

The depending syntax entry, *declaration-list*, is not needed elsewhere and can therefore be removed, too:

> *Remove the syntax entry* declaration-list *from 6.9.1 p1.*

For the term *function body* see below **??**.

2.2. Function declarators (6.7.6.3)

Here, the paragraphs p13 changes, not only because of the removal of identifier lists, but also because we want to strengthen the case where an empty parameter list is used in a function definition. For that case, we propose to enforce the empty list as to be as if a list of **`void`** had been given. Thus implicitly it is defining a prototype with no parameters. The whole modified paragraph reads:

> For a function declarator without a parameter type list: if it is part of a definition of that function the function has no parameters and the effect is as if it were declared with a parameter type list consisting of the keyword **`void`**; otherwise it specifies that no information about the number or types of the parameters is supplied.[FNT1]
> A function declarator provides a *prototype* for the function if it includes a parameter type list.[FNT2] Otherwise, a function declaration is said to have no prototype.

With two attached footnotes:

> [FNT1] See "future language directions" (6.11.6).
>
> [FNT2] This implies that a function definition without a parameter list provides a prototype, and that subsequent calls to that function in the same translation unit are constrained not to provide any argument to the function call. Thus a definition of a function without parameter list and one that has such a list consisting of the keyword **`void`** are fully equivalent.

Paragraphs p14 also changes, because in the case analysis for the definition of compatible and composite types the cases for declarations with and without prototypes had been in-

tertwined. Therefore the new version is much simpler than the previous, but cannot simply be obtained by text removal.

> For two function types to be compatible, both shall specify compatible return types. Moreover, the parameter type lists, if both are present, shall agree in the number of parameters and in use of the ellipsis terminator; corresponding parameters shall have compatible types. If one type has a parameter type list and the other type has none and is not part of a function definition, the parameter list shall not have an ellipsis terminator. In the determination of type compatibility and of a composite type, each parameter declared with function or array type is taken as having the adjusted type and each parameter declared with qualified type is taken as having the unqualified version of its declared type.

2.3. Function definitions

Since we want to be able to keep a syntax where a function can have an empty parameter list, the removal of the syntax entry with *identifier list* (6.7.6 p1) implies that we have to make the appearance of the *parameter type list* in *function declarator* optional.
This has also the effect of fixing a problem what was voluntarily omitted during the integration of the attribute feature. Namely, as C2x currently stands, a definition that has an attribute after the *function declarator* and where the parameter list is empty, is a constraint violation:

```
1  double myrand() [[deprecated]] {
2    // something here
3  }
```

Whereas the version

```
1  double myrand(void) [[deprecated]] {
2    // something here
3  }
```

is valid. With the above change, both definitions become truly equivalent and are both valid.
Otherwise, the removal of the corresponding descriptive text then has no surprises.

2.4. Question

QUESTION 1. *Shall the feature of function definitions with identifier lists be removed from ISO 9899 as proposed in N2432?*

3. EDITORIAL CHANGES

Along with the above changes we also noticed some possible minor improvements that are merely editorial.

3.1. Transform an abusively long footnote into a note.

The current document has an abusively long footnote 173 that is better expressed as a simple note at the end of the clause.

QUESTION 2. *Shall footnote 173 be tranformed into a new note (6.9.1 p13) as proposed in N2432?*

3.2. Properly introduce the syntax entry function-body

The term *function body* is used in various places of the document, but it is only introduced half-heartedly as

> ... the compound statement that constitutes the body of the definition is executed ...

We propose to clarify this by introducing the new syntax term *function body* (that simply resolves to *compound statement*) and to change the above sentence as

> ... the compound statement ~~that constitutes the body~~ of the ~~definition~~body is executed ...

QUESTION 3. *Shall 6.9.1 introduce the term* function body *as proposed in N2432?*

3.3. Remove the term no-leading-attribute-declaration

During the integration of the attribute feature the the term *no leading attribute declaration* had been introduced to cope with the specific case of *declaration list*. This can now be removed to simplify the corresponding syntax derivation for declarations.

QUESTION 4. *Shall we remove the term* no leading attribute declaration *as proposed in N2432?*

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 If the expression that denotes the called function has a type that does not include a prototype, the integer promotions are performed on each argument, and arguments that have type **float** are promoted to **double**. These are called the *default argument promotions*. If the number of arguments does not equal the number of parameters, the behavior is undefined. If the function is defined with a type that includes a prototype, and either the prototype ends with an ellipsis (`, ...`) or the types of the arguments after promotion are not compatible with the types of the parameters, the behavior is undefined. ~~If the function is defined with a type that does not include a prototype, and the types of the arguments after promotion are not compatible with those of the parameters after promotion, the behavior is undefined, except for the following cases:~~

> ~~one promoted type is a signed integer type, the other promoted type is the corresponding unsigned integer type, and the value is representable in both types; both types are pointers to qualified or unqualified versions of a character type or **void**.~~

7 If the expression that denotes the called function has a type that does include a prototype, the arguments are implicitly converted, as if by assignment, to the types of the corresponding parameters, taking the type of each parameter to be the unqualified version of its declared type. The ellipsis notation in a function prototype declarator causes argument type conversion to stop after the last declared parameter. The default argument promotions are performed on trailing arguments.

8 No other conversions are performed implicitly; in particular, the number and types of arguments are not compared with those of the parameters in a function definition that does not include a function prototype declarator.

9 If the function is defined with a type that is not compatible with the type (of the expression) pointed to by the expression that denotes the called function, the behavior is undefined.

10 There is a sequence point after the evaluations of the function designator and the actual arguments but before the actual call. Every evaluation in the calling function (including other function calls) that is not otherwise specifically sequenced before or after the execution of the body of the called function is indeterminately sequenced with respect to the execution of the called function.[102)]

11 Recursive function calls shall be permitted, both directly and indirectly through any chain of other functions.

12 **EXAMPLE** In the function call

```
(*pf[f1()]) (f2(), f3() + f4())
```

the functions `f1`, `f2`, `f3`, and `f4` can be called in any order. All side effects have to be completed before the function pointed to by `pf[f1()]` is called.

Forward references: function declarators (~~including prototypes) (??~~6.7.6.3), function definitions (6.9.1), the **return** statement (6.8.6.4), simple assignment (6.5.16.1).

6.5.2.3 Structure and union members

Constraints

1 The first operand of the `.` operator shall have an atomic, qualified, or unqualified structure or union type, and the second operand shall name a member of that type.

2 The first operand of the `->` operator shall have type "pointer to atomic, qualified, or unqualified structure" or "pointer to atomic, qualified, or unqualified union", and the second operand shall name a member of the type pointed to.

Semantics

3 A postfix expression followed by the `.` operator and an identifier designates a member of a structure or union object. The value is that of the named member,[103)] and is an lvalue if the first expression is an lvalue. If the first expression has qualified type, the result has the so-qualified version of the type

[102)] In other words, function executions do not "interleave" with each other.

[103)] If the member used to read the contents of a union object is not the same as the member last used to store a value in the object, the appropriate part of the object representation of the value is reinterpreted as an object representation in the new type as described in 6.2.6 (a process sometimes called "type punning"). This might be a trap representation.

6.5.4 Cast operators

Syntax

1 *cast-expression:*
unary-expression
(*type-name* **)** *cast-expression*

Constraints

2 Unless the type name specifies a void type, the type name shall specify atomic, qualified, or unqualified scalar type, and the operand shall have scalar type.

3 Conversions that involve pointers, other than where permitted by the constraints of 6.5.16.1, shall be specified by means of an explicit cast.

4 A pointer type shall not be converted to any floating type. A floating type shall not be converted to any pointer type.

Semantics

5 Preceding an expression by a parenthesized type name converts the value of the expression to the unqualified version of the named type. This construction is called a *cast*.[112] A cast that specifies no conversion has no effect on the type or value of an expression.

6 If the value of the expression is represented with greater range or precision than required by the type named by the cast (6.3.1.8), then the cast specifies a conversion even if the type of the expression is the same as the named type and removes any extra range and precision.

Forward references: equality operators (6.5.9), function declarators (~~including prototypes~~) (~~??~~6.7.6.3), simple assignment (6.5.16.1), type names (6.7.7).

6.5.5 Multiplicative operators

Syntax

1 *multiplicative-expression:*
cast-expression
multiplicative-expression `*` *cast-expression*
multiplicative-expression `/` *cast-expression*
multiplicative-expression `%` *cast-expression*

Constraints

2 Each of the operands shall have arithmetic type. The operands of the `%` operator shall have integer type.

3 If either operand has decimal floating type, the other operand shall not have standard floating type, complex type, or imaginary type.

Semantics

4 The usual arithmetic conversions are performed on the operands.

5 The result of the binary `*` operator is the product of the operands.

6 The result of the `/` operator is the quotient from the division of the first operand by the second; the result of the `%` operator is the remainder. In both operations, if the value of the second operand is zero, the behavior is undefined.

7 When integers are divided, the result of the `/` operator is the algebraic quotient with any fractional part discarded.[113] If the quotient `a/b` is representable, the expression `(a/b)*b + a%b` shall equal `a`;

[112] A cast does not yield an lvalue.
[113] This is often called "truncation toward zero".

6.7 Declarations

Syntax

1 ~~*no-leading-attribute-declaration:*~~ *declaration:*
declaration-specifiers *init-declarator-list*$_{\text{opt}}$ **;**
~~*static_assert-declaration declaration:*~~
~~*no-leading-attribute-declaration*~~
attribute-specifier-sequence declaration-specifiers *init-declarator-list* **;**
static_assert-declaration *attribute-declaration*
declaration-specifiers:
declaration-specifier *attribute-specifier-sequence*$_{\text{opt}}$
declaration-specifier *declaration-specifiers*
declaration-specifier:
storage-class-specifier
type-specifier-qualifier
function-specifier
init-declarator-list:
init-declarator
init-declarator-list **,** *init-declarator*
init-declarator:
declarator
declarator **=** *initializer*
attribute-declaration:
attribute-specifier-sequence **;**

Constraints

2 A declaration other than a static_assert or attribute declaration shall declare at least a declarator (other than the parameters of a function or the members of a structure or union), a tag, or the members of an enumeration.

3 If an identifier has no linkage, there shall be no more than one declaration of the identifier (in a declarator or type specifier) with the same scope and in the same name space, except that:

— a typedef name may be redefined to denote the same type as it currently does, provided that type is not a variably modified type;

— tags may be redeclared as specified in 6.7.2.3.

4 All declarations in the same scope that refer to the same object or function shall specify compatible types.

Semantics

5 A declaration specifies the interpretation and properties of a set of identifiers. A *definition* of an identifier is a declaration for that identifier that:

— for an object, causes storage to be reserved for that object;

— for a function, includes the function body;[126)]

— for an enumeration constant, is the (only) declaration of the identifier;

— for a typedef name, is the first (or only) declaration of the identifier.

6 The declaration specifiers consist of a sequence of specifiers, followed by an optional attribute specifier sequence, that indicate the linkage, storage duration, and part of the type of the entities that the declarators denote. The init declarator list is a comma-separated sequence of declarators, each of

[126)] Function definitions have a different syntax, described in 6.9.1.

which may have additional type information, or an initializer, or both. The declarators contain the identifiers (if any) being declared. The optional attribute specifier sequence appertains to each of the entities declared by the declarators of the init declarator list.

7 If an identifier for an object is declared with no linkage, the type for the object shall be complete by the end of its declarator, or by the end of its init-declarator if it has an initializer; in the case of function parameters~~(including in prototypes)~~, it is the adjusted type (see 6.7.6.3) that is required to be complete.

8 The optional attribute specifier sequence terminating a sequence of declaration specifiers appertains to the type determined by the preceding sequence of declaration specifiers. The attribute specifier sequence affects the type only for the declaration it appears in, not other declarations involving the same type.

9 Except where specified otherwise, the meaning of an attribute declaration is implementation-defined.

10 **EXAMPLE** In the declaration for an entity, attributes appertaining to that entity may appear at the start of the declaration and after the identifier for that declaration.

```
[[deprecated]] void f [[deprecated]] (void); // valid
```

Forward references: declarators (6.7.6), enumeration specifiers (6.7.2.2), initialization (6.7.9), type names (6.7.7), type qualifiers (6.7.3).

6.7.1 Storage-class specifiers

Syntax

1 *storage-class-specifier:*

typedef
extern
static
_Thread_local
auto
register

Constraints

2 At most, one storage-class specifier may be given in the declaration specifiers in a declaration, except that **_Thread_local** may appear with **static** or **extern**.[127)]

3 In the declaration of an object with block scope, if the declaration specifiers include **_Thread_local**, they shall also include either **static** or **extern**. If **_Thread_local** appears in any declaration of an object, it shall be present in every declaration of that object.

4 **_Thread_local** shall not appear in the declaration specifiers of a function declaration.

Semantics

5 The **typedef** specifier is called a "storage-class specifier" for syntactic convenience only; it is discussed in 6.7.8. The meanings of the various linkages and storage durations were discussed in 6.2.2 and 6.2.4.

6 A declaration of an identifier for an object with storage-class specifier **register** suggests that access to the object be as fast as possible. The extent to which such suggestions are effective is implementation-defined.[128)]

7 The declaration of an identifier for a function that has block scope shall have no explicit storage-class specifier other than **extern**.

[127)] See "future language directions" (6.11.5).

[128)] The implementation can treat any **register** declaration simply as an **auto** declaration. However, whether or not addressable storage is actually used, the address of any part of an object declared with storage-class specifier **register** cannot be computed, either explicitly (by use of the unary & operator as discussed in 6.5.3.2) or implicitly (by converting an array name to a pointer as discussed in 6.3.2.1). Thus, the only operator that can be applied to an array declared with storage-class specifier **register** is **sizeof**.

Semantics

6 The first form is equivalent to **_Alignas(_Alignof(** *type-name*)).

7 The alignment requirement of the declared object or member is taken to be the specified alignment. An alignment specification of zero has no effect.[149] When multiple alignment specifiers occur in a declaration, the effective alignment requirement is the strictest specified alignment.

8 If the definition of an object has an alignment specifier, any other declaration of that object shall either specify equivalent alignment or have no alignment specifier. If the definition of an object does not have an alignment specifier, any other declaration of that object shall also have no alignment specifier. If declarations of an object in different translation units have different alignment specifiers, the behavior is undefined.

6.7.6 Declarators

Syntax

1 *declarator:*
pointer$_{\text{opt}}$ *direct-declarator*

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

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*$_{\text{opt}}$ ***** **]**

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

pointer:
***** *attribute-specifier-sequence*$_{\text{opt}}$ *type-qualifier-list*$_{\text{opt}}$
***** *attribute-specifier-sequence*$_{\text{opt}}$ *type-qualifier-list*$_{\text{opt}}$ *pointer*

type-qualifier-list:
type-qualifier
type-qualifier-list *type-qualifier*

parameter-type-list:
parameter-list
parameter-list **, ...**

parameter-list:
parameter-declaration
parameter-list **,** *parameter-declaration*

parameter-declaration:
attribute-specifier-sequence$_{\text{opt}}$ *declaration-specifiers* *declarator*
attribute-specifier-sequence$_{\text{opt}}$ *declaration-specifiers* *abstract-declarator*$_{\text{opt}}$

~~*identifier-list:*~~
~~*identifier*~~
~~*identifier-list* **,** *identifier*~~

[149] An alignment specification of zero also does not affect other alignment specifications in the same declaration.

The first declares x to be a pointer to **int**; the second declares y to be an array of **int** of unspecified size (an incomplete type), the storage for which is defined elsewhere.

9 **EXAMPLE 3** The following declarations demonstrate the compatibility rules for variably modified types.

```
extern int n;
extern int m;

void fcompat(void)
{
      int a[n][6][m];
      int (*p)[4][n+1];
      int c[n][n][6][m];
      int (*r)[n][n][n+1];
      p = a;      // invalid:  not compatible because 4 != 6
      r = c;      // compatible, but defined behavior only if
                  // n == 6 and m == n+1
}
```

10 **EXAMPLE 4** All declarations of variably modified (VM) types have to be at either block scope or function prototype scope. Array objects declared with the **_Thread_local**, **static**, or **extern** storage-class specifier cannot have a variable length array (VLA) type. However, an object declared with the **static** storage-class specifier can have a VM type (that is, a pointer to a VLA type). Finally, all identifiers declared with a VM type have to be ordinary identifiers and cannot, therefore, be members of structures or unions.

```
extern int n;
int A[n];                         // invalid:  file scope VLA
extern int (*p2)[n];              // invalid:  file scope VM
int B[100];                       // valid:  file scope but not VM

void fvla(int m, int C[m][m]);    // valid:  VLA with prototype scope

void fvla(int m, int C[m][m])     // valid:  adjusted to auto pointer to VLA
{
      typedef int VLA[m][m];      // valid:  block scope typedef VLA

      struct tag {
            int (*y)[n];          // invalid:  y not ordinary identifier
            int z[n];             // invalid:  z not ordinary identifier
      };
      int D[m];                   // valid:  auto VLA
      static int E[m];            // invalid:  static block scope VLA
      extern int F[m];            // invalid:  F has linkage and is VLA
      int (*s)[m];                // valid:  auto pointer to VLA
      extern int (*r)[m];         // invalid:  r has linkage and points to VLA
      static int (*q)[m] = &B;    // valid:  q is a static block pointer to VLA
}
```

Forward references: function declarators (6.7.6.3), function definitions (6.9.1), initialization (6.7.9).

6.7.6.3 Function declarators

Constraints

1 A function declarator shall not specify a return type that is a function type or an array type.

2 The only storage-class specifier that shall occur in a parameter declaration is **register**.

3 ~~An identifier list in a function declarator that is not part of a definition of that function shall be empty.~~

After adjustment, the parameters in a parameter type list in a function declarator that is part of a definition of that function shall not have incomplete type.

Semantics

4 If, in the declaration "`T D1`", `D1` has the form

`D (` *parameter-type-list*$_{\text{opt}}$ `)` *attribute-specifier-sequence*$_{\text{opt}}$
~~or `D (` *identifier-list*$_{\text{opt}}$ `)`~~ and the type specified for *ident* in the declaration "`T D`" is "*derived-declarator-type-list T*", then the type specified for *ident* is "*derived-declarator-type-list* function returning the unqualified version of *T*". The optional attribute specifier sequence appertains to the function type.

5 A parameter type list specifies the types of, and may declare identifiers for, the parameters of the function.

6 A declaration of a parameter as "array of *type*" shall be adjusted to "qualified pointer to *type*", where the type qualifiers (if any) are those specified within the `[` and `]` of the array type derivation. If the keyword **`static`** also appears within the `[` and `]` of the array type derivation, then for each call to the function, the value of the corresponding actual argument shall provide access to the first element of an array with at least as many elements as specified by the size expression.

7 A declaration of a parameter as "function returning *type*" shall be adjusted to "pointer to function returning *type*", as in 6.3.2.1.

8 If the list terminates with an ellipsis (`, ...`), no information about the number or types of the parameters after the comma is supplied.[152)]

9 The special case of an unnamed parameter of type **`void`** as the only item in the list specifies that the function has no parameters.

10 If, in a parameter declaration, an identifier can be treated either as a typedef name or as a parameter name, it shall be taken as a typedef name.

11 If the function declarator is not part of a definition of that function, parameters may have incomplete type and may use the `[*]` notation in their sequences of declarator specifiers to specify variable length array types.

12 The storage class specifier in the declaration specifiers for a parameter declaration, if present, is ignored unless the declared parameter is one of the members of the parameter type list for a function definition. The optional attribute specifier sequence in a parameter declaration appertains to the parameter.

13 ~~An identifier list declares only the identifiers of the parameters of the function. An empty list in~~ For a function declarator ~~that~~ without a parameter type list: if it is part of a definition of that function ~~specifies that~~ the function has no parameters ~~. The empty list in a function declarator that is not part of a definition of that function~~ and the effect is as if it were declared with a parameter type list consisting of the keyword **`void`**; otherwise it specifies that no information about the number or types of the parameters is supplied.[153)] A function declarator provides a *prototype* for the function if it includes a parameter type list.[154)] Otherwise, a function declaration is said to have no prototype.

14 For two function types to be compatible, both shall specify compatible return types. Moreover, the parameter type lists, if both are present, shall agree in the number of parameters and in use of the ellipsis terminator; corresponding parameters shall have compatible types. If one type has a parameter type list and the other type ~~is specified by a function declarator that~~ has none and is not part of a function definition~~and that contains an empty identifier list~~, the parameter list shall not have an ellipsis terminator~~and the type of each parameter shall be compatible with the type that results from the application of the default argument promotions. If one type has a parameter type list and the other type is specified by a function definition that contains a (possibly empty) identifier list, both shall agree in the number of parameters, and the type of each prototype parameter shall be compatible with the type that results from the application of the default argument promotions to the type of the corresponding identifier. (~~. In the determination of type compatibility and of a

[152)] The macros defined in the `<stdarg.h>` header (7.16) can be used to access arguments that correspond to the ellipsis.
[153)] See "future language directions" (6.11.6).
[154)] This implies that a function definition without a parameter list provides a prototype, and that subsequent calls to that function in the same translation unit are constrained not to provide any argument to the function call. Thus a definition of a function without parameter list and one that has such a list consisting of the keyword **`void`** are fully equivalent.

composite type, each parameter declared with function or array type is taken as having the adjusted type and each parameter declared with qualified type is taken as having the unqualified version of its declared type.)

15 **EXAMPLE 1** The declaration

```
int f(void), *fip(), (*pfi)();
```

declares a function `f` with no parameters returning an **int**, a function `fip` with no parameter specification returning a pointer to an **int**, and a pointer `pfi` to a function with no parameter specification returning an **int**. It is especially useful to compare the last two. The binding of `*fip()` is `*(fip())`, so that the declaration suggests, and the same construction in an expression requires, the calling of a function `fip`, and then using indirection through the pointer result to yield an **int**. In the declarator `(*pfi)()`, the extra parentheses are necessary to indicate that indirection through a pointer to a function yields a function designator, which is then used to call the function; it returns an **int**.

16 If the declaration occurs outside of any function, the identifiers have file scope and external linkage. If the declaration occurs inside a function, the identifiers of the functions `f` and `fip` have block scope and either internal or external linkage (depending on what file scope declarations for these identifiers are visible), and the identifier of the pointer `pfi` has block scope and no linkage.

17 **EXAMPLE 2** The declaration

```
int (*apfi[3])(int *x, int *y);
```

declares an array `apfi` of three pointers to functions returning **int**. Each of these functions has two parameters that are pointers to **int**. The identifiers `x` and `y` are declared for descriptive purposes only and go out of scope at the end of the declaration of `apfi`.

18 **EXAMPLE 3** The declaration

```
int (*fpfi(int (*)(long), int))(int, ...);
```

declares a function `fpfi` that returns a pointer to a function returning an **int**. The function `fpfi` has two parameters: a pointer to a function returning an **int** (with one parameter of type **long int**), and an **int**. The pointer returned by `fpfi` points to a function that has one **int** parameter and accepts zero or more additional arguments of any type.

19 **EXAMPLE 4** The following prototype has a variably modified parameter.

```
void addscalar(int n, int m,
      double a[n][n*m+300], double x);

int main()
{
      double b[4][308];
      addscalar(4, 2, b, 2.17);
      return 0;
}

void addscalar(int n, int m,
      double a[n][n*m+300], double x)
{
      for (int i = 0; i  <  n; i++)
            for (int j = 0, k = n*m+300; j  <  k; j++)
                  // a is a pointer to a VLA with n*m+300 elements
                  a[i][j] += x;
}
```

20 **EXAMPLE 5** The following are all compatible function prototype declarators.

```
double maximum(int n, int m, double a[n][m]);
double maximum(int n, int m, double a[*][*]);
double maximum(int n, int m, double a[ ][*]);
double maximum(int n, int m, double a[ ][m]);
```

as are:

```
void f(double (* restrict a)[5]);
```

6.9 External definitions

Syntax

1 *translation-unit:*
external-declaration
translation-unit external-declaration

external-declaration:
function-definition
declaration

Constraints

2 The storage-class specifiers **auto** and **register** shall not appear in the declaration specifiers in an external declaration.

3 There shall be no more than one external definition for each identifier declared with internal linkage in a translation unit. Moreover, if an identifier declared with internal linkage is used in an expression (other than as a part of the operand of a **sizeof** or **_Alignof** operator whose result is an integer constant), there shall be exactly one external definition for the identifier in the translation unit.

Semantics

4 As discussed in 5.1.1.1, the unit of program text after preprocessing is a translation unit, which consists of a sequence of external declarations. These are described as "external" because they appear outside any function (and hence have file scope). As discussed in 6.7, a declaration that also causes storage to be reserved for an object or a function named by the identifier is a definition.

5 An *external definition* is an external declaration that is also a definition of a function (other than an inline definition) or an object. If an identifier declared with external linkage is used in an expression (other than as part of the operand of a **sizeof** or **_Alignof** operator whose result is an integer constant), somewhere in the entire program there shall be exactly one external definition for the identifier; otherwise, there shall be no more than one.[172)]

6.9.1 Function definitions

Syntax

1 *function-definition:*
attribute-specifier-sequence$_{\text{opt}}$ *declaration-specifiers declarator*
~~*declaration-list*$_{\text{opt}}$ *compound-statement*~~ *function-body*

~~*declaration-list:*~~
~~*no-leading-attribute-declaration*~~ *function-body:*
~~*declaration-list no-leading-attribute-declaration*~~ *compound-statement*

Constraints

2 The identifier declared in a function definition (which is the name of the function) shall have a function type, as specified by the declarator portion of the function definition.

3 The return type of a function shall be **void** or a complete object type other than array type.

4 The storage-class specifier, if any, in the declaration specifiers shall be either **extern** or **static**.

5 ~~If the declarator includes a parameter type list, the~~ The declaration of each parameter shall include an identifier, except for the special case of a parameter list consisting of a single parameter of type **void**, in which case there shall not be an identifier. ~~No declaration list shall follow.~~

[172)] Thus, if an identifier declared with external linkage is not used in an expression, there need be no external definition for it.

~~If the declarator includes an identifier list, each declaration in the declaration list shall have at least one declarator, those declarators shall declare only identifiers from the identifier list, and every identifier in the identifier list shall be declared. An identifier declared as a typedef name shall not be redeclared as a parameter. The declarations in the declaration list shall contain no storage-class specifier other than **register** and no initializations.~~

Semantics

6 The optional attribute specifier sequence in a function definition appertains to the function.

7 The declarator in a function definition specifies the name of the function being defined and the identifiers ~~of its parameters. If the declarator includes a parameter type list, the list also specifies the~~ and types of all the parameters; ~~such a~~ the declarator also serves as a function prototype for later calls to the same function in the same translation unit. ~~If the declarator includes an identifier list,~~[173] ~~the types of the parameters shall be declared in a following declaration list. In either case, the~~ The type of each parameter is adjusted as described in~~?? for a parameter type list~~ 6.7.6.3; the resulting type shall be a complete object type.

8 If a function that accepts a variable number of arguments is defined without a parameter type list that ends with the ellipsis notation, the behavior is undefined.

9 Each parameter has automatic storage duration; its identifier is an lvalue.[173] The layout of the storage for parameters is unspecified.

10 On entry to the function, the size expressions of each variably modified parameter are evaluated and the value of each argument expression is converted to the type of the corresponding parameter as if by assignment. (Array expressions and function designators as arguments were converted to pointers before the call.)

11 After all parameters have been assigned, the compound statement ~~that constitutes the body~~ of the function ~~definition~~ body is executed.

12 Unless otherwise specified, if the } that terminates ~~a function~~ the function body is reached, and the value of the function call is used by the caller, the behavior is undefined.

13 **NOTE** In a function definition, the type of the function and its prototype cannot be inherited from a typedef:

```
typedef int F(void);                  // type F is “function with no parameters
                                      // returning int”
F f, g;                               // f and g both have type compatible with F
F f { /* ... */ }                     // WRONG: syntax/constraint error
F g() { /* ... */ }                   // WRONG: declares that g returns a function
int f(void) { /* ... */ }             // RIGHT: f has type compatible with F
int g() { /* ... */ }                 // RIGHT: g has type compatible with F
F *e(void) { /* ... */ }              // e returns a pointer to a function
F *((e))(void) { /* ... */ }          // same:  parentheses irrelevant
int (*fp)(void);                      // fp points to a function that has type F
F *Fp;                                // Fp points to a function that has type F
```

14 **EXAMPLE 1** In the following:

```
extern int max(int a, int b)
{
      return a  >  b ? a: b;
}
```

extern is the storage-class specifier and **int** is the type specifier; `max(int a, int b)` is the function declarator; and

```
{ return a  >  b ? a: b; }
```

172) ~~The intent is that the type category in a function definition cannot be inherited from a typedef:~~

173) ~~See “future language directions” (??).~~

173) A parameter identifier cannot be redeclared in the function body except in an enclosed block.

is the function body~~. The following similar definition uses the identifier-list form for the parameter declarations: Here **int** a, b; is the declaration list for the parameters. The difference between these two definitions is that the first form acts as a prototype declaration that forces conversion of the arguments of subsequent calls to the function, whereas the second form does not~~.

15 **EXAMPLE 2** To pass one function to another, one might say

```
int f(void);
/* ... */
g(f);
```

Then the definition of `g` might read

```
void g(int (*funcp)(void))
{
      /* ... */
      (*funcp)(); /* or funcp(); ...*/
}
```

or, equivalently,

```
void g(int func(void))
{
      /* ... */
      func(); /* or (*func)(); ...*/
}
```

6.9.2 External object definitions

Semantics

1 If the declaration of an identifier for an object has file scope and an initializer, the declaration is an external definition for the identifier.

2 A declaration of an identifier for an object that has file scope without an initializer, and without a storage-class specifier or with the storage-class specifier **`static`**, constitutes a *tentative definition*. If a translation unit contains one or more tentative definitions for an identifier, and the translation unit contains no external definition for that identifier, then the behavior is exactly as if the translation unit contains a file scope declaration of that identifier, with the composite type as of the end of the translation unit, with an initializer equal to `{ 0 }`.

3 If the declaration of an identifier for an object is a tentative definition and has internal linkage, the declared type shall not be an incomplete type.

6.10 Preprocessing directives

Syntax

1 *preprocessing-file:*
group$_{\text{opt}}$
group:
group-part
group group-part
group-part:
if-section
control-line
text-line
# *non-directive*
if-section:
if-group elif-groups$_{\text{opt}}$ *else-group*$_{\text{opt}}$ *endif-line*
if-group:
# **if** *constant-expression new-line group*$_{\text{opt}}$
# **ifdef** *identifier new-line group*$_{\text{opt}}$
# **ifndef** *identifier new-line group*$_{\text{opt}}$
elif-groups:
elif-group
elif-groups elif-group
elif-group:
# **elif** *constant-expression new-line group*$_{\text{opt}}$
else-group:
# **else** *new-line group*$_{\text{opt}}$
endif-line:
**endif** *new-line*
control-line:
**include** *pp-tokens new-line*
**define** *identifier replacement-list new-line*
**define** *identifier lparen identifier-list*$_{\text{opt}}$ **)**
replacement-list new-line
**define** *identifier lparen* **...** **)** *replacement-list new-line*
**define** *identifier lparen identifier-list* **,** **...** **)**
replacement-list new-line
**undef** *identifier new-line*
**line** *pp-tokens new-line*
**error** *pp-tokens*$_{\text{opt}}$ *new-line*
**pragma** *pp-tokens*$_{\text{opt}}$ *new-line*
*new-line*
text-line:
pp-tokens$_{\text{opt}}$ *new-line*
non-directive:
pp-tokens new-line
lparen:
a **(** character not immediately preceded by white space
replacement-list:
pp-tokens$_{\text{opt}}$
pp-tokens:
preprocessing-token
pp-tokens preprocessing-token
new-line:
the new-line character
identifier-list:
identifier
identifier-list **,** *identifier*

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 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.

6.11.6 Function declarators

1 The use of function declarators ~~with empty parentheses (not prototype-format parameter type declarators)~~ without prototypes is an obsolescent feature.

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

6.11.7 Pragma directives

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

6.11.8 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.

(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 **(** *type-name* **)**
_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*$_{\text{opt}}$
~~*direct-declarator* **(** *identifier-list*$_{\text{opt}}$ **)**~~

(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*~~ *parameter-type-list*$_{\text{opt}}$ **)**

(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*$_{\text{opt}}$
~~*identifier-list:*~~
~~*identifier*~~
~~*identifier-list* **,** *identifier*~~ (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 (*constant-expression* **,** *string-literal* **) ;**
_Static_assert (*constant-expression* **) ;**
(6.7.11.1) *attribute-specifier-sequence:*
attribute-specifier-sequence$_{\text{opt}}$ *attribute-specifier*

(6.8.4) *selection-statement:*
if (*expression* **)** *statement*
if (*expression* **)** *statement* **else** *statement*
switch (*expression* **)** *statement*

(6.8.5) *iteration-statement:*
while (*expression* **)** *statement*
do *statement* **while (** *expression* **) ;**
for ($expression_{opt}$ **;** $expression_{opt}$ **;** $expression_{opt}$ **)** *statement*
for (*declaration* $expression_{opt}$ **;** $expression_{opt}$ **)** *statement*

(6.8.6) *jump-statement:*
goto *identifier* **;**
continue ;
break ;
return $expression_{opt}$ **;**

A.2.4 External definitions

(6.9) *translation-unit:*
external-declaration
translation-unit external-declaration

(6.9) *external-declaration:*
function-definition
declaration

(6.9.1) *function-definition:*
attribute-specifier-sequence$_{opt}$ *declaration-specifiers declarator* ~~*declaration-list*$_{opt}$ *compound-statement*~~ *function-body*

(6.9.1) ~~*declaration-list:*~~
~~*no-leading-attribute-declaration*~~ *function-body:*
~~*declaration-list no-leading-attribute-declaration*~~ *compound-statement*

A.3 Preprocessing directives

(6.10) *preprocessing-file:*
$group_{opt}$
(6.10) *group:*
group-part
group group-part
(6.10) *group-part:*
if-section
control-line
text-line
*non-directive*
(6.10) *if-section:*
if-group $elif\text{-}groups_{opt}$ $else\text{-}group_{opt}$ *endif-line*
(6.10) *if-group:*
 # if *constant-expression new-line* $group_{opt}$
 # ifdef *identifier new-line* $group_{opt}$
 # ifndef *identifier new-line* $group_{opt}$
(6.10) *elif-groups:*
elif-group
elif-groups elif-group

(6.10) *elif-group:*
**elif** *constant-expression new-line group*$_{\text{opt}}$
(6.10) *else-group:*
**else** *new-line group*$_{\text{opt}}$
(6.10) *endif-line:*
**endif** *new-line*
(6.10) *control-line:*
**include** *pp-tokens new-line*
**define** *identifier replacement-list new-line*
**define** *identifier lparen identifier-list*$_{\text{opt}}$ **)**
replacement-list new-line
**define** *identifier lparen* **...** **)** *replacement-list new-line*
**define** *identifier lparen identifier-list* **,** **...** **)**
replacement-list new-line
**undef** *identifier new-line*
**line** *pp-tokens new-line*
**error** *pp-tokens*$_{\text{opt}}$ *new-line*
**pragma** *pp-tokens*$_{\text{opt}}$ *new-line*
*new-line*
(6.10) *text-line:*
pp-tokens$_{\text{opt}}$ *new-line*
(6.10) *non-directive:*
pp-tokens new-line
(6.10) *lparen:*
a **(** character not immediately preceded by white space
(6.10) *replacement-list:*
pp-tokens$_{\text{opt}}$
(6.10) *pp-tokens:*
preprocessing-token
pp-tokens preprocessing-token
(6.10) *new-line:*
the new-line character
(6.10) *identifier-list:*
identifier
identifier-list **,** *identifier*
(6.10.6) *standard-pragma:*
**pragma** **STDC** **FP_CONTRACT** *on-off-switch*
**pragma** **STDC** **FENV_ACCESS** *on-off-switch*
**pragma** **STDC** **FENV_DEC_ROUND** *direction*
**pragma** **STDC** **FENV_ROUND** *dec-direction*
**pragma** **STDC** **CX_LIMITED_RANGE** *on-off-switch*

(6.10.6) *on-off-switch:* one of
ON **OFF** **DEFAULT**

(6.10.6) *direction:* one of
FE_DOWNWARD **FE_TONEAREST** **FE_TONEARESTFROMZERO**
FE_TOWARDZERO **FE_UPWARD** **FE_DYNAMIC**

(6.10.6) *dec-direction:* one of
FE_DEC_DOWNWARD **FE_DEC_TONEAREST** **FE_DEC_TONEARESTFROMZERO**
FE_DEC_TOWARDZERO **FE_DEC_UPWARD** **FE_DEC_DYNAMIC**

Annex I
(informative)
Common warnings

1 An implementation may generate warnings in many situations, none of which are specified as part of this document. The following are a few of the more common situations.

2
- A new **`struct`** or **`union`** type appears in a function prototype (6.2.1, 6.7.2.3).
- A block with initialization of an object that has automatic storage duration is jumped into (6.2.4).
- An implicit narrowing conversion is encountered, such as the assignment of a **`long int`** or a **`double`** to an **`int`**, or a pointer to **`void`** to a pointer to any type other than a character type (6.3).
- A hexadecimal floating constant cannot be represented exactly in its evaluation format (6.4.4.2).
- An integer character constant includes more than one character or a wide character constant includes more than one multibyte character (6.4.4.4).
- The characters `/*` are found in a comment (6.4.7).
- An "unordered" binary operator (not comma, &&, or ||) contains a side effect to an lvalue in one operand, and a side effect to, or an access to the value of, the identical lvalue in the other operand (6.5).
- A function is called but no prototype has been supplied (6.5.2.2).
- ~~The arguments in a function call do not agree in number and type with those of the parameters in a function definition that is not a prototype (6.5.2.2).~~ An object is defined but not used (6.7).
- A value is given to an object of an enumerated type other than by assignment of an enumeration constant that is a member of that type, or an enumeration object that has the same type, or the value of a function that returns the same enumerated type (6.7.2.2).
- An aggregate has a partly bracketed initialization (6.7.8).
- A statement cannot be reached (6.8).
- A statement with no apparent effect is encountered (6.8).
- A constant expression is used as the controlling expression of a selection statement (6.8.4).
- An incorrectly formed preprocessing group is encountered while skipping a preprocessing group (6.10.1).
- An unrecognized **`#pragma`** directive is encountered (6.10.6).

— An attempt is made to use the value of a void expression, or an implicit or explicit conversion (except to **void**) is applied to a void expression (6.3.2.2).

— Conversion of a pointer to an integer type produces a value outside the range that can be represented (6.3.2.3).

— Conversion between two pointer types produces a result that is incorrectly aligned (6.3.2.3).

— A pointer is used to call a function whose type is not compatible with the referenced type (6.3.2.3).

— An unmatched ' or " character is encountered on a logical source line during tokenization (6.4).

— A reserved keyword token is used in translation phase 7 or 8 for some purpose other than as a keyword (6.4.1).

— A universal character name in an identifier does not designate a character whose encoding falls into one of the specified ranges (6.4.2.1).

— The initial character of an identifier is a universal character name designating a digit (6.4.2.1).

— Two identifiers differ only in nonsignificant characters (6.4.2.1).

— The identifier __func__ is explicitly declared (6.4.2.2).

— The program attempts to modify a string literal (6.4.5).

— The characters ', \, ", //, or /* occur in the sequence between the < and > delimiters, or the characters ', \, //, or /* occur in the sequence between the " delimiters, in a header name preprocessing token (6.4.7).

— A side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object (6.5).

— An exceptional condition occurs during the evaluation of an expression (6.5).

— An object has its stored value accessed other than by an lvalue of an allowable type (6.5).

— For a call to a function without a function prototype in scope, the number of arguments does not equal the number of parameters (6.5.2.2).

— For a call to a function without a function prototype in scope where the function is defined with a function prototype, either the prototype ends with an ellipsis or the types of the arguments after default argument promotion are not compatible with the types of the parameters (6.5.2.2).

— ~~For a call to a function without a function prototype in scope where the function is not defined with a function prototype, the types of the arguments after default argument promotion are not compatible with those of the parameters after promotion (with certain exceptions) (6.5.2.2).~~ A function is defined with a type that is not compatible with the type (of the expression) pointed to by the expression that denotes the called function (6.5.2.2).

— A member of an atomic structure or union is accessed (6.5.2.3).

— The operand of the unary * operator has an invalid value (6.5.3.2).

— A pointer is converted to other than an integer or pointer type (6.5.4).

— The value of the second operand of the / or % operator is zero (6.5.5).

— If the quotient a/b is not representable, the behavior of both a/b and a%b (6.5.5).

— Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object (6.5.6).

— An attempt is made to refer to an object defined with a volatile-qualified type through use of an lvalue with non-volatile-qualified type (6.7.3).

— The specification of a function type includes any type qualifiers (6.7.3).

— Two qualified types that are required to be compatible do not have the identically qualified version of a compatible type (6.7.3).

— An object which has been modified is accessed through a restrict-qualified pointer to a const-qualified type, or through a restrict-qualified pointer and another pointer that are not both based on the same object (6.7.3.1).

— A restrict-qualified pointer is assigned a value based on another restricted pointer whose associated block neither began execution before the block associated with this pointer, nor ended before the assignment (6.7.3.1).

— A function with external linkage is declared with an **inline** function specifier, but is not also defined in the same translation unit (6.7.4).

— A function declared with a **_Noreturn** function specifier returns to its caller (6.7.4).

— The definition of an object has an alignment specifier and another declaration of that object has a different alignment specifier (6.7.5).

— Declarations of an object in different translation units have different alignment specifiers (6.7.5).

— Two pointer types that are required to be compatible are not identically qualified, or are not pointers to compatible types (6.7.6.1).

— The size expression in an array declaration is not a constant expression and evaluates at program execution time to a nonpositive value (6.7.6.2).

— In a context requiring two array types to be compatible, they do not have compatible element types, or their size specifiers evaluate to unequal values (6.7.6.2).

— A declaration of an array parameter includes the keyword **static** within the [and] and the corresponding argument does not provide access to the first element of an array with at least the specified number of elements (6.7.6.3).

— A storage-class specifier or type qualifier modifies the keyword **void** as a function parameter type list (6.7.6.3).

— In a context requiring two function types to be compatible, they do not have compatible return types, or their parameters disagree in use of the ellipsis terminator or the number and type of parameters (after default argument promotion, when there is no parameter type list~~or when one type is specified by a function definition with an identifier list) (**??**)~~ (6.7.6.3).

— The value of an unnamed member of a structure or union is used (6.7.9).

— The initializer for a scalar is neither a single expression nor a single expression enclosed in braces (6.7.9).

— The initializer for a structure or union object that has automatic storage duration is neither an initializer list nor a single expression that has compatible structure or union type (6.7.9).

— The initializer for an aggregate or union, other than an array initialized by a string literal, is not a brace-enclosed list of initializers for its elements or members (6.7.9).

— An identifier with external linkage is used, but in the program there does not exist exactly one external definition for the identifier, or the identifier is not used and there exist multiple external definitions for the identifier (6.9~~). A function definition includes an identifier list, but the types of the parameters are not declared in a following declaration list (6.9.1~~).