Type inference for variable definitions and function returns

Jens Gustedt

ISO/IEC JTC 1/SC 22/WG14
WG 21, SG 22

N2923 v6
P2305R5

2022-1-30

Type inference for variable definitions and function returns
proposal for C23

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

We propose the inclusion of the so-called **auto** feature for variable definitions and function types into C. This feature allows to infer types from expressions that are used in initializers or **return** statements. This is part of a series of papers for the improvement of type-generic programming in C that has been introduced in N2890.

Changes: v6/R5
— move handling of auto parameters to N2924
— make type declarations in underspecified declarations undefined behavior
— add Joseph Myers's wording for the typeof specifier, but make it its own subphrase
— address an ambiguity concerning the scope of auto variables
— make clear that shadowing by auto only concerns ordinary identifiers
— minor text adaptions

v5/R4 straighten the proposed text and move discussion of all special cases (bit-fields, compatible types, VM) into non-normative text such as notes and examples

v4/R3 some word smithing

v3/R2 remove a requirement on qualifiers and atomic derivation that was too restrictive

v2/R1 changes the rules for type inference to use a **typeof** specifier. This simplifies the rules and brings the defined semantics in line with C++.

I. MOTIVATION

In N2890 it is argued that the features presented in this paper are useful in a more general context, namely for the combination with lambdas. We will not repeat this argumentation here, but try to motivate the introduction of the **auto** feature as a stand-alone addition to C.

In accordance with C's syntax for declarations and in extension of its semantics, C++ has a feature that allows to infer the type of a variable from its initializer expression.

```
auto y = cos(x);
```

This eases the use of type-generic functions because now the return value **and** type can be captured in an auxiliary variable, without necessarily having the type of the argument, here x, at hand. That feature is not only interesting because of the obvious convenience for programmers who are perhaps too lazy to lookup the type of x. It can help to avoid code maintenance problems: if x is a function parameter for which potentially the type may be adjusted during the lifecycle of the program (say from **float** to **double**), all dependent auxiliary variables within the function are automatically updated to the new type.

This can even be used if the return type of a type-generic function is just an aggregation of several values for which the type itself is just an uninteresting artefact:

```
1 #define div(X, Y)            \
2   _Generix((X)+(Y),          \
```

```
            int: div,          \
            long: ldiv,        \
            long long: lldiv) \
            ((X), (Y))

  // int, long or long long?
  auto res = div(38484848448, 448484844);
  auto a = b * res.quot + res.rem;
```

An important restriction for the coding of type-generic macros in current C is the impossibility to declare local variables of a type that is dependent on the type(s) of the macro argument(s). Therefore, such macros often need arguments that provide the types for which the macro was evaluated. This not only inconvenient for the user of such macros but also an important source of errors. If the user chooses the wrong type, implicit conversions can impede on the correctness of the macro call.

For type-generic macros that declare local variables, **auto** can easily remove the need for the specification of the base types of the macro arguments:

```
#define dataCondStoreTG(P, E, D)                \
  do {                                          \
    auto* _pr_p = (P);                          \
    auto _pr_expected = (E);                    \
    auto _pr_desired = (D);                     \
    bool _pr_c;                                 \
    do {                                        \
      mtx_lock(&_pr_p->mtx);                    \
      _pr_c = (_pr_p->data == _pr_expected);    \
      if (_pr_c) _pr_p->data = _pr_desired;     \
      mtx_unlock(&_pr_p->mtx);                  \
    }  while(!_pr_c);                           \
  } while (false)
```

C's declaration syntax currently already allows to omit the type in a variable definition, as long as the variable is initialized and a storage initializer (such as **auto** or **static**) disambiguates the construct from an assignment. In previous versions of C the interpretation of such a definition had been **int**; since C11 this is a constraint violation. We will propose to align C with C++, here, and to change this such the type of the variable is inferred the type from the initializer expression.

In a second alignment with C++ we propose to also extend this notion of **auto** type inference to function return types, namely such that such a return type can be deduced from **return** statements or be **void** if there is none. Having that possibility can also ease portable coding with types that, depending on the platform, may resolve to different base types.

A good example for such a type in the C standard itself is **time_t**, which is just known to be an implementation-defined real type. Consider the following function that computes the maximum value of two parameters that have types **time_t** and **long**.

```
inline auto max(time_t a, long b){
  return (a < 0)
         ? ((b <  0) ? ((a < b) ? b : a) : b)
         : ((b >= 0) ? ((a < b) ? b : a) : a);
```

```
}
```

The **return** expression performs default arithmetic conversion to determine a type that can hold the maximum value. The function definition is adjusted to that return type. This property holds regardless if **time_t** is a floating point or integer type and, if it is an integer type, if it is a signed or unsigned type.

As another example, consider the following function that computes the sum over an array of integers of a platform-dependent integer type strength and returns the value as the promoted type of strength.

```
inline auto sum(size_t n, strength A[n]){
  switch(n) {
    case 0: return +((strength)0); // return the promoted type
    case 1: return +A[0];          // return the promoted type
    default: return sum(n/2, A) + sum(n - n/2, &A[n/2]);
  }
}
```

If instead sum would have been defined with a prototype as follows

```
strength sum(size_t n, strength A[n]);
```

for a narrow type strength such as **unsigned char**, the return type and result would be different from the previous. In particular, the result of the addition would have been converted back from the promoted type to strength before each **return**, possibly leading to a surprising overall results. On the other hand, using the promoted type explicitly

```
strength_promoted sum(size_t n, strength A[n]);
```

forces the user to determine that promoted type in a possibly complicated cascade of compile-time conditionals for which the result heavily depends on properties of the execution platform.

It makes not much sense to have **auto** forward declarations of indentifiers since they could not be used easily before their definition. Most functions that use the **auto** feature will probably be restricted to one TU (and thus best declared with **static**) or be declared to be **inline**. For the latter, it will still be important to be able to emit the function symbol in a chosen TU, and so declaration syntax for **auto** still may have its use *after* a definition has been met. Consider the following declarations for the **max** function from above:

```
extern auto max(time_t, long); // forces symbol emission for TU
auto max(time_t, long);        // same
auto max();                    // same
auto max;                      // same
```

The **extern** declaration and the equivalent ones are considered to be valid, if they follow the definition and thus the inferred return type is already known.

II. PROPOSED ADDTIONS

In the following we will explain our proposed additions and argue the design choices. The full text of the proposed additions is given as a diff against C17 in the appendix.

II.1. Syntax

Type inference for definitions of objects and functions could be added to the standard with a minimal effort by just allowing the omission of a type specifier in all places where this is unambiguous. Unfortunately this is not the path that the current extensions have chosen.

Gcc and related compilers provide the feature by adding an **__auto_type** keyword. The use of that keyword disambiguates between declarations and assignments, but is also used when a declaration that has a storage specifier infers the type from an intializer.

C++ reuses the **auto** keyword for the same purpose. That is, **auto** can be added to any declaration, even already having another storage specifier, to indicate that the type of the declared identifier is inferred from an initialization (for object declarations) or from **return** statements (for function declarations).

To achieve maximum compatibility with C++, we propose to follow their lead and to relax the rules for the **auto** keyword as indicated. Details of the necessary relaxation of syntax constraints and semantics can be found in clauses 6.7.1 (storage class specifiers) and 6.9.1 (function definitions).

II.2. Semantics

The addition to the semantics is anchored in clause 6.7.2 (type specifiers) where the constraint that a type specifier has to appear in a declaration is removed, and the term *underspecified declaration* is introduced to describe declarations that have no such specifier.

For underspecified declarations, a new clause 6.7.11 is added. It refers to the necessary adjustments for functions (see II.4 below), and then specifies and exemplifies the new rules for objects.

II.3. Type inference for objects

An underspecified declaration of a object has to be a definition that additionally has an initializer, namely an initializer with an *assignment expression* E of type T_0. The type of that assignment expression that enters into the adjusted type of the declared object is the type T_1 of that expression after possible lvalue, array-to-pointer or function-to-pointer conversion. The type T_1 is unique and determined at translation time. There is a unique type T_2 that can replace the **auto** specifier, such that the such adjusted declaration becomes a complete object declaration of type T_3 that can be initialized with the given initializer. To accomodate complicated type expressions, T_2 is supposed to be given as a typeof operator.

T_0	type of assignment expression E
T_1	type of E after lvalue etc conversion
T_2	type specifier to adjust the declaration
T_3	type of the adjusted the declaration

For example with

```
long A[5] = { 0 };
auto * ap = A;
```

we have

T_0 **long**[5]
T_1 **long***
T_2 **typeof**(**long**)
T_3 **long***

In this example, T_2 could equally have been specified as **long**, but there are cases that could otherwise be specified in a closed form that replaces **auto** if a **typedef** of the corresponding type would be available. With the above

```
1 auto const pA = &A;
2 typedef long (*const pAtype)[5];
3 pAtype pA = &A;                       // same
4 typeof(long(*)[5]) const pA = &A; // same, fulfills requirements
5 long (*const pA)[5]) = &A;            // same
```

only the second of the equivalent definitions fulfills the requirements without referring to an additional type definition.

II.3.1. Bit-fields. The defintion of bit-fields in C is underspecified, in that their types are only known if an lvalue expression of a bit-field additionaly undergoes integer promotion. If no such promotion is performed, for example in a **_Generic** or comma expression, implementations diverge in their interpretation of the standard. Some always produce one of the types **bool**, **signed int** or **unsigned int**, others produce some implementation defined types of that reflect the width of the bit-field. The latter are not integer types in the sense of the standard because they only have to convert under promotion and need not to have any other property of integers, and, usually don't have documented declaration syntax.

It is not the place of this proposal here to sort out this inconsistency between different interpretations of the standard. Therefore we simply add a remark to a note, NOTE 1, that describes the possible difficulties.

II.3.2. Combined definitions and compatible types. The semantics of underspecified declarations become complicated if they contain definitions for several objects where the inferred type has to be consistent. Here the choice is, that inferred types have to be the same, only having compatible types is not sufficient. This is particularly important for integer types, where mixing different enumeration types would have an ambiguity which type is chosen.

To emphasize on this we have added an example, EXAMPLE 2, to the proposed text.

II.3.3. Combined definitions and variably modified types. These types add a difficulty to combined definitions because the inferred array bounds of different pointers to VLA may depend on runtime values and need not to be consistent between different objects that are defined in the same declaration. The consistency of such definitions can in general not be checked at compile time, and thus a program that exhibits this may run into undefined behavior.

To emphasize on this we have added an example, EXAMPLE 3, to the proposed text.

II.4. Type inference for functions

The semantics for underspecified functions are mainly defined in two places: clauses 6.8.6.4 (the return statement) and 6.9.1 (function definitions).

Important requirements for the semantics are to ensure that multiple **return** statements provide consistent return types and that underspecified functions still can be used recursively. This is ensured by using the (lvalue converted) type of the lexicographic first return expression, and by constraining possible other return expressions to have the same type.

Using the first expression, has the advantage that the function prototype is then known thereafter, and that the scope of the identifier can start at the end of that first **return** statement. For example in the function `sum` above, the first **return** statement (for **case** `0`) determines that the return type is the promoted type of `strength` and the identifier `sum` can then be used in recursive calls for the **default** case.

II.5. Permitted types for the return of functions

A return statement in an underspecified function could *a priori* have a type that is only locally defined within the function. Using such a type would make it visible to code outside of the definition of the function, and thus defy the usually scoping rules for type definitions. Therefore, clause `6.9.1, p.7` constrains types that may be used as an inferred type to be visible at the location of the **auto** keyword, and stipulates that the type must be complete at that point.

II.6. Ambiguities with type definitions

Since identifiers may be redeclared in inner scopes, ambiguities with identifiers that are type definitions could occur. We resolve that ambiguity by reviving a rule that solved the same problem when C still had the implicit **int** rule. This is done in `6.7.8 p3` (Type definitions) by adding the following phrase:

> *If the identifier is redeclared in an inner scope the inner declaration shall not be underspecified.*

III. QUESTIONS FOR WG14

In the March 2021 session, WG14 has already voted in favor of integrating the **auto** feature into C23 along the lines as described here.

(1) Does WG14 want to integrate the changes as specified in N2923 into C23?

Acknowledgements

Thanks to JeanHeyd Meneide, Joseph Myers, Richard Smith and Martin Uecker for comments and suggestions.

IV. PROPOSED WORDING

The proposed text is given as diff against C17 with some traces of a possible addition of **typeof**.

— Additions to the text are marked as shown.
— Deletions of text are marked as ~~shown~~.

6. Language

6.1 Notation

1 In the syntax notation used in this clause, syntactic categories (nonterminals) are indicated by *italic type*, and literal words and character set members (terminals) by **bold type**. A colon (:) following a nonterminal introduces its definition. Alternative definitions are listed on separate lines, except when prefaced by the words "one of". An optional symbol is indicated by the subscript "opt", so that

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

indicates an optional expression enclosed in braces.

2 When syntactic categories are referred to in the main text, they are not italicized and words are separated by spaces instead of hyphens.

3 A summary of the language syntax is given in Annex A.

6.2 Concepts

6.2.1 Scopes of identifiers

1 An identifier can denote an object; a function; a tag or a member of a structure, union, or enumeration; a typedef name; a label name; a macro name; or a macro parameter. The same identifier can denote different entities at different points in the program. A member of an enumeration is called an *enumeration constant*. Macro names and macro parameters are not considered further here, because prior to the semantic phase of program translation any occurrences of macro names in the source file are replaced by the preprocessing token sequences that constitute their macro definitions.

2 For each different entity that an identifier designates, the identifier is *visible* (i.e., can be used) only within a region of program text called its *scope*. Different entities designated by the same identifier either have different scopes, or are in different name spaces. There are four kinds of scopes: function, file, block, and function prototype. (A *function prototype* is a declaration of a function that declares the types of its parameters.)

3 A label name is the only kind of identifier that has *function scope*. It can be used (in a **`goto`** statement) anywhere in the function in which it appears, and is declared implicitly by its syntactic appearance (followed by a `:` and a statement).

4 Every other identifier has scope determined by the placement of its declaration (in a declarator or type specifier). If the declarator or type specifier that declares the identifier appears outside of any block or list of parameters, the identifier has *file scope*, which terminates at the end of the translation unit. If the declarator or type specifier that declares the identifier appears inside a block or within the list of parameter declarations in a function definition, the identifier has *block scope*, which terminates at the end of the associated block. If the declarator or type specifier that declares the identifier appears within the list of parameter declarations in a function prototype (not part of a function definition), the identifier has *function prototype scope*, which terminates at the end of the function declarator. If an identifier designates two different entities in the same name space, the scopes might overlap. If so, the scope of one entity (the *inner scope*) will end strictly before the scope of the other entity (the *outer scope*). Within the inner scope, the identifier designates the entity declared in the inner scope; the entity declared in the outer scope is *hidden* (and not visible) within the inner scope.

5 Unless explicitly stated otherwise, where this document uses the term "identifier" to refer to some entity (as opposed to the syntactic construct), it refers to the entity in the relevant name space whose declaration is visible at the point the identifier occurs.

6 Two identifiers have the *same scope* if and only if their scopes terminate at the same point.

7 Structure, union, and enumeration tags have scope that begins just after the appearance of the tag in a type specifier that declares the tag. Each enumeration constant has scope that begins just after the appearance of its defining enumerator in an enumerator list. An ordinary identifier that

has an underspecified definition and that designates an object, has a scope that starts at the end of its initializer and from that point extends to the whole translation unit (for file scope identifiers) or to the whole block (for block scope identifiers); if the same ordinary identifier declares another entity with a scope that encloses the current block, that declaration is hidden as soon as the inner declarator is completed.[29] An identifier that designates a function with an underspecified definition has a scope that starts after the lexically first **`return`** statement in its function body or at the end of the function body if there is no such **`return`**, and from that point extends to the whole translation unit. Any other identifier has scope that begins just after the completion of its declarator.

8 As a special case, a type name (which is not a declaration of an identifier) is considered to have a scope that begins just after the place within the type name where the omitted identifier would appear were it not omitted.

Forward references: declarations (6.7), function calls (6.5.2.2), function definitions (6.9.1), identifiers (6.4.2), macro replacement (6.10.3), name spaces of identifiers (6.2.3), source file inclusion (6.10.2), statements and blocks (6.8).

6.2.2 Linkages of identifiers

1 An identifier declared in different scopes or in the same scope more than once can be made to refer to the same object or function by a process called *linkage*.[30] There are three kinds of linkage: external, internal, and none.

2 In the set of translation units and libraries that constitutes an entire program, each declaration of a particular identifier with *external linkage* denotes the same object or function. Within one translation unit, each declaration of an identifier with *internal linkage* denotes the same object or function. Each declaration of an identifier with *no linkage* denotes a unique entity.

3 If the declaration of a file scope identifier for an object or a function contains the storage-class specifier **`static`**, the identifier has internal linkage.[31]

4 For an identifier declared with the storage-class specifier **`extern`** in a scope in which a prior declaration of that identifier is visible,[32] if the prior declaration specifies internal or external linkage, the linkage of the identifier at the later declaration is the same as the linkage specified at the prior declaration. If no prior declaration is visible, or if the prior declaration specifies no linkage, then the identifier has external linkage.

5 If the declaration of an identifier for a function has no storage-class specifier, its linkage is determined exactly as if it were declared with the storage-class specifier **`extern`**. If the declaration of an identifier for an object has file scope and no storage-class specifier or only the specifier **`auto`** , its linkage is external.

6 The following identifiers have no linkage: an identifier declared to be anything other than an object or a function; an identifier declared to be a function parameter; a block scope identifier for an object declared without the storage-class specifier **`extern`**.

7 If, within a translation unit, the same identifier appears with both internal and external linkage, the behavior is undefined.

Forward references: declarations (6.7), expressions (6.5), external definitions (6.9), statements (6.8).

6.2.3 Name spaces of identifiers

1 If more than one declaration of a particular identifier is visible at any point in a translation unit, the syntactic context disambiguates uses that refer to different entities. Thus, there are separate *name spaces* for various categories of identifiers, as follows:

— *label names* (disambiguated by the syntax of the label declaration and use);

[29] That means, that the outer declaration is not visible for the initializer.
[30] There is no linkage between different identifiers.
[31] A function declaration can contain the storage-class specifier **`static`** only if it is at file scope; see 6.7.1.
[32] As specified in 6.2.1, the later declaration might hide the prior declaration.

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** , and that **auto** may appear with all others but with **typedef** .[124)]

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. **auto** shall only appear in the declaration specifiers of an identifier with file scope if the declaration is also a definition or if a definition of that identifier is already visible.

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.[125)]

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

8 If an aggregate or union object is declared with a storage-class specifier other than **typedef**, the properties resulting from the storage-class specifier, except with respect to linkage, also apply to the members of the object, and so on recursively for any aggregate or union member objects.

9 If **auto** appears with another storage-class specifier, or if it appears in a declaration at file scope it is ignored for the purpose of determining a storage class or linkage. It then only indicates that the declared type may be inferred from an initializer (for objects see 6.7.11), or from the function body (for functions see 6.8.6.4).

Forward references: type definitions (6.7.8), type inference (6.7.11), function definitions (6.9.1).

6.7.2 Type specifiers

Syntax

1 *type-specifier:*

void
char
short
int

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

[125)] 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**.

long
float
double
signed
unsigned
_Bool
_Complex
atomic-type-specifier
struct-or-union-specifier
enum-specifier
typedef-name
typeof-specifier

Constraints

2 ~~At~~ Except where the type is inferred (6.7.11, 6.8.6.4, 6.9.1), at least one type specifier shall be given in the declaration specifiers in each declaration, and in the specifier-qualifier list in each struct declaration and type name. Each list of type specifiers shall be one of the following multisets (delimited by commas, when there is more than one multiset per item); the type specifiers may occur in any order, possibly intermixed with the other declaration specifiers.

— **void**
— **char**
— **signed char**
— **unsigned char**
— **short**, **signed short**, **short int**, or **signed short int**
— **unsigned short**, or **unsigned short int**
— **int**, **signed**, or **signed int**
— **unsigned**, or **unsigned int**
— **long**, **signed long**, **long int**, or **signed long int**
— **unsigned long**, or **unsigned long int**
— **long long**, **signed long long**, **long long int**, or **signed long long int**
— **unsigned long long**, or **unsigned long long int**
— **float**
— **double**
— **long double**
— **_Bool**
— **float _Complex**
— **double _Complex**
— **long double _Complex**
— atomic type specifier
— ~~struct or union~~ **struct** or **union** specifier
— ~~enum~~ **enum** specifier
— ~~typedef name~~ **typedef** name
— **typeof** specifier.

3 The type specifier **_Complex** shall not be used if the implementation does not support complex types (see 6.10.8.3).

Semantics

4 Specifiers for structures, unions, enumerations, and atomic types are discussed in 6.7.2.1 through 6.7.2.4. Declarations of typedef names are discussed in 6.7.8. The characteristics of the other types are discussed in 6.2.5. Declarations for which the type specifiers are inferred from initializers are discussed in 6.7.11.

5 Each of the comma-separated multisets designates the same type, except that for bit-fields, it is implementation-defined whether the specifier **`int`** designates the same type as **`signed int`** or the same type as **`unsigned int`**.

6 A declaration that contains no type specifier is said to be *underspecified*. Identifiers that are such declared have incomplete type. Their type can be completed by type inference from an initialization (for objects) or from **`return`** statements in a function body (for return types of functions). If such an initialization or **`return`** statement uses a type that has not been declared previous to the underspecified declaration, the behavior is undefined.

Forward references: the **`return`** statement (6.8.6.4), atomic type specifiers (6.7.2.4), enumeration specifiers (6.7.2.2), structure and union specifiers (6.7.2.1), tags (6.7.2.3), type definitions (6.7.8), type inference (6.7.11).

6.7.2.1 Structure and union specifiers

Syntax

1 *struct-or-union-specifier:*
struct-or-union *identifier*$_{\text{opt}}$ **`{`** *struct-declaration-list* **`}`**
struct-or-union *identifier*

struct-or-union:
`struct`
`union`

struct-declaration-list:
struct-declaration
struct-declaration-list *struct-declaration*

struct-declaration:
specifier-qualifier-list *struct-declarator-list*$_{\text{opt}}$ **`;`**
static_assert-declaration

specifier-qualifier-list:
type-specifier *specifier-qualifier-list*$_{\text{opt}}$
type-qualifier *specifier-qualifier-list*$_{\text{opt}}$
alignment-specifier *specifier-qualifier-list*$_{\text{opt}}$

struct-declarator-list:
struct-declarator
struct-declarator-list **`,`** *struct-declarator*

struct-declarator:
declarator
declarator$_{\text{opt}}$ **`:`** *constant-expression*

Constraints

2 A struct-declaration that does not declare an anonymous structure or anonymous union shall contain a struct-declarator-list.

3 A structure or union shall not contain a member with incomplete or function type (hence, a structure shall not contain an instance of itself, but may contain a pointer to an instance of itself), except that the last member of a structure with more than one named member may have incomplete array type; such a structure (and any union containing, possibly recursively, a member that is such a structure) shall not be a member of a structure or an element of an array.

4 The expression that specifies the width of a bit-field shall be an integer constant expression with a nonnegative value that does not exceed the width of an object of the type that would be specified

6.7.8 Type definitions

Syntax

1 *typedef-name:*
identifier

Constraints

2 If a typedef name specifies a variably modified type then it shall have block scope.

Semantics

3 In a declaration whose storage-class specifier is **typedef**, each declarator defines an identifier to be a typedef name that denotes the type specified for the identifier in the way described in 6.7.6. Any array size expressions associated with variable length array declarators are evaluated each time the declaration of the typedef name is reached in the order of execution. A **typedef** declaration does not introduce a new type, only a synonym for the type so specified. That is, in the following declarations:

```
typedef T type_ident;
type_ident D;
```

`type_ident` is defined as a typedef name with the type specified by the declaration specifiers in `T` (known as *T*), and the identifier in `D` has the type "*derived-declarator-type-list T*" where the *derived-declarator-type-list* is specified by the declarators of `D`. A typedef name shares the same name space as other identifiers declared in ordinary declarators. If the identifier is redeclared in an enclosed block the inner declaration shall not be underspecified.

4 **EXAMPLE 1** After

```
typedef int MILES, KLICKSP();
typedef struct { double hi, lo; } range;
```

the constructions

```
MILES distance;
extern KLICKSP *metricp;
range x;
range z, *zp;
```

are all valid declarations. The type of `distance` is **int**, that of `metricp` is "pointer to function with no parameter specification returning **int**", and that of `x` and `z` is the specified structure; `zp` is a pointer to such a structure. The object `distance` has a type compatible with any other **int** object.

5 **EXAMPLE 2** After the declarations

```
typedef struct s1 { int x; } t1, *tp1;
typedef struct s2 { int x; } t2, *tp2;
```

type `t1` and the type pointed to by `tp1` are compatible. Type `t1` is also compatible with type **struct** `s1`, but not compatible with the types **struct** `s2`, `t2`, the type pointed to by `tp2`, or **int**.

6 **EXAMPLE 3** The following obscure constructions

```
typedef signed int t;
typedef int plain;
struct tag {
      unsigned t:4;
      const t:5;
      plain r:5;
};
```

```
{
      struct S l = { 1, .t = x, .t.l = 41, };
}
```

The value of `l.t.k` is 42, because implicit initialization does not override explicit initialization.

37 **EXAMPLE 13** Space can be "allocated" from both ends of an array by using a single designator:

```
int a[MAX] = {
      1, 3, 5, 7, 9, [MAX-5] = 8, 6, 4, 2, 0
};
```

38 In the above, if `MAX` is greater than ten, there will be some zero-valued elements in the middle; if it is less than ten, some of the values provided by the first five initializers will be overridden by the second five.

39 **EXAMPLE 14** Any member of a union can be initialized:

```
union { /* ... */ } u = {.any_member = 42 };
```

Forward references: common definitions <stddef.h> (7.19).

6.7.11 Type inference

Constraints

1 An underspecified declaration shall contain the storage class specifier **auto**.

2 For an identifier that is declared but not defined by an underspecified declaration, a prior definition shall be be visible. For an underspecified declaration an init-declarator corresponding to the definition of an object shall have of one of the forms

declarator **=** *assignment-expression*
declarator **= {** *assignment-expression* **}**
declarator **= {** *assignment-expression* **, }**

such that the declarator does not declare an array.[158)]

3 Prior to an underspecified declaration there shall exist a typeof specifier *type* (that is, a type specifier that is a typeof operator applied to an expression or type name) that if used to replace the **auto** specifier makes the adjusted declaration a valid declaration;[159)] *type* shall not declare a tag or the contents of a structure, union or enumeration (including at function prototype scope). If it is also the definition of a function the return type shall be determined from **return** statements (or the lack thereof) as specified in 6.9.1. Otherwise, *type* shall be such that for all defined objects the assignment expression in the corresponding init-declarator, after possible lvalue, array-to-pointer or function-to-pointer conversion, has the non-atomic, unqualified type of the declared object.

Description

4 In an underspecified declaration the type of the declared identifiers is the type after the declaration would have been adjusted by a choice for *type* as described in the constraints. If the declaration is also an object definition, each assignment expressions that is used to determine the type and initial value of an object is evaluated exactly once each time the declaration is met.

5 **NOTE 1** Because of the relatively complex syntax and semantics of type specifiers, the requirements for *type* use a typeof specifier. For an underspecified declaration

```
auto x = v;
```

in many situations a non-atomic unqualified type *type* as required can be found by using **remove_quals** and the adjusted definition as follows would be valid:

[158)] The scope rules as described in 6.2.1 also prohibit the use of the identifier of the declarator within the assignment expression.

[159)] The qualification of the type of an lvalue that is the assignment expression, or the fact that it is atomic, can never be used to infer such a property of the type of the defined object.

```
    remove_quals(v) x = v;
```

This is for example the case if the identifier or tag name of the type of the initializer expression `v` in the initializer of `x` is shadowed. In contrast to that, if `v` is a bit-field member to which the implementation assigns a type that has no declaration syntax outside a structure or union declaration, no typeof specifier exists and the underspecified declaration as above is invalid. The indicated adjustment with a typeof operator doesn't imply that `v` is evaluated twice, even if it has a variably modified type.

6 **NOTE 2** If an underspecified declaration that also defines a structure or union type is valid or not depends on a prior visibility of a type with the same tag.

```
    auto p = (struct s { int a; } *)0;
```

Here a replacement of **`auto`** by **`typeof`** without duplication of the definition would be as follows.

```
    typeof(struct s *) p = (struct s { int a; } *)0;
```

Such a declaration is valid if prior to the declaration a forward declaration of **`struct`** `s` introduces the new structure type to the current scope. If there is no such forward declaration in the current scope the behavior is undefined, in particular if even a different type with the tag `s` in an surrounding scope is visible. A direct use of the structure definition as the type specifier ensures the validity of the declaration and should be preferred if possible.

```
    struct s { int a; } * p = 0;
```

7 **NOTE 3** For most assignment expressions of integer or floating point type, there are several choices for *type* that would make such a declaration valid. The second part of the constraint ensures that among these either a unique type is determined such that none of the initializers needs further conversion, or, that the declaration is invalid because no typeof specifier exists that has exactly the required type.

8 **NOTE 4** For the correspondence of the declared type of an object and the type of its initializer, having compatible types is not sufficient. For example integer types of the same rank and signedness but that are nevertheless different types are not considered. Thus, the validity of an underspecified declaration that declares several identifiers and that uses initializer expressions for which this standard does not specify a unique type for all implementations, such as integer literals or bit-field members, depends on the concrete choice of these types by the implementation. Where possible, it is recommended to split such an underspecified declaration into several to achieve full portability.

9 **EXAMPLE 1** Consider the following file scope definitions:

```
static auto a = 3.5;
auto * p = &a;
```

They are interpreted as if they had been written as:

```
static double a = 3.5;
double * p = &a;
```

So effectively `a` is a **`double`** and `p` is a **`double`***.

Both identifiers can later be redeclared as long as such a declaration is consistent with the previous ones. For example declarations as the following

```
    extern auto a;
    extern auto p;
```

may be used inside a block where the file scope declarations are shadowed by declarations in an enclosing block.

10 **EXAMPLE 2** Declarations that are the definition of several objects, may make type inferrence difficult and not portable.

```
    enum A { aVal, } aObj = aVal;
    enum B { bVal, } bObj = bVal;
```

```
    int  au = aObj, bu = bObj;  // valid, values have type convertible to int
    auto ax = aObj, bx = bObj;  // invalid, same rank but different types
    auto ay = aObj;             // valid, ay has type enum A
    auto by = bObj;             // valid, by has type enum B
    auto az = aVal, bz = bVal;  // valid, az and bz have type int
    struct set { int bits:6; } X = { .bits = 37, };
    auto k = 37, m = X.bits;    // possibly valid or invalid
```

Here, the definitions of `ax` and `bx` cannot be satified with the same typeof specifier as a replacement for **`auto`**; any fixed choice would require the conversion of at least one of the initializer expressions to the other type. For `k` and `m` the difficulty is that `X.bits` may have a signed or an unsigned type, and that even it is signed it is not specified that the type then is necessarily **`int`**.

11 **EXAMPLE 3** If *type* is a variably modified type, the variable array bounds that are determined by *type* have to be consistent for all objects that are defined by the same underspecified declaration. This consistency can in general not be verified at translation time.

```
    size_t r, s;
    ...
    double aVM[r];
    double bVM[s];
    double cVM[3];
    double dVM[r];
    auto vmPa = &aVM, vmPb = &bVM;    // undefined if r != s
    auto vmPa = &aVM, vmPc = &cVM;    // invalid, even if for some executions r is 3
    auto vmPa = &aVM, vmPd = &dVM;    // valid, same array sizes in all executions
```

12 **EXAMPLE 4** The scope of the identifier for which the type is inferred only starts after the end of the initializer (6.2.1), so the assignment expression cannot use the identifier to refer to the object or function that is declared, for example to take its address. Any use of the identifier in the initializer is invalid, even if an entity with the same name exists in an outer scope.

```
    {
      double a = 7;
      double b = 9;
      {
        double b = b * b;    // undefined, uses uninitialized variable without address
        printf("%g\n", a);   // valid, uses "a" from outer scope, prints 7
        auto a   = a * a;    // invalid, "a" from outer scope is already shadowed
      }
      {
        auto b   = a * a;    // valid, uses "a" from outer scope
        auto a   = b;        // valid, shadows "a" from outer scope
        ...
        printf("%g\n", a);   // valid, uses "a" from inner scope, prints 49
      }
      ...
    }
```

13 **EXAMPLE 5** In the following, `pA` is valid because the type of `A` after array-to-pointer conversion is a pointer type, and `qA` and `rA` are valid because they do not declare arrays but pointers to array.

```
double A[3]   = { 0 };
auto * pA     = A;
auto (*qA)[3] = &A;
auto * rA     = &A;
```

14 **EXAMPLE 6** Type inference can be used to capture the type of a call to a type-generic function and can be used to ensure that the same type as the argument `x` is used.

```
#include <tgmath.h>
auto y = cos(x);
```

If instead the type of `y` is explicitly specified to a different type than `x`, a diagnosis of the mismatch is not enforced.

15 **EXAMPLE 7** A type-generic macro that generalizes the **`div`** functions (7.22.6.2) is defined and used as follows.

```
#define div(X, Y) _Generic((X)+(Y), int: div, long: ldiv, long long: lldiv)((X), (Y))
auto z = div(x, y);
auto q = z.quot;
auto r = z.rem;
```

16 **EXAMPLE 8** Underspecified definitions of objects may occur in all contexts that allow the initializer syntax as described in the constraints. In particular they can be used to ensure type safety of **`for`**-loop controlling expressions.

```
   for (auto i = j; i < 2*j; ++i) {
      ...
   }
```

Here, regardless of the integer rank or signedness of the type of `j`, `i` will have the non-atomic unqualified type of `j`. So, after lvalue conversion and possible promotion, the two operands of the `<` operator in the controlling expression are guaranteed to have the same type, and, in particular, the same signedness.

6.7.12 Static assertions

Syntax

1 *static_assert-declaration:*

`_Static_assert` **`(`** *constant-expression* **`,`** *string-literal* **`)`** **`;`**

Constraints

2 The constant expression shall compare unequal to 0.

Semantics

3 The constant expression shall be an integer constant expression. If the value of the constant expression compares unequal to 0, the declaration has no effect. Otherwise, the constraint is violated and the implementation shall produce a diagnostic message that includes the text of the string literal, except that characters not in the basic source character set are not required to appear in the message.

Forward references: diagnostics (7.2).

```
                /* ... */
                continue;
        }
        // handle other operations
        /* ... */
}
```

4 **EXAMPLE 2** A **goto** statement is not allowed to jump past any declarations of objects with variably modified types. A jump within the scope, however, is permitted.

```
goto lab3;              // invalid:  going INTO scope of VLA.
{
        double a[n];
        a[j] = 4.4;
lab3:
        a[j] = 3.3;
        goto lab4;      // valid:  going WITHIN scope of VLA.
        a[j] = 5.5;
lab4:
        a[j] = 6.6;
}
goto lab4;              // invalid:  going INTO scope of VLA.
```

6.8.6.2 The **continue** statement

Constraints

1 A **continue** statement shall appear only in or as a loop body.

Semantics

2 A **continue** statement causes a jump to the loop-continuation portion of the smallest enclosing iteration statement; that is, to the end of the loop body. More precisely, in each of the statements

```
while (/* ...  */) {
   /* ...  */
   continue;
   /* ...  */
contin:;
}
```

```
do {
   /* ...  */
   continue;
   /* ...  */
contin:;
} while (/* ...  */);
```

```
for (/* ...  */) {
   /* ...  */
   continue;
   /* ...  */
contin:;
}
```

unless the **continue** statement shown is in an enclosed iteration statement (in which case it is interpreted within that statement), it is equivalent to **goto** `contin`;.[166)]

6.8.6.3 The **break** statement

Constraints

1 A **break** statement shall appear only in or as a switch body or loop body.

Semantics

2 A **break** statement terminates execution of the smallest enclosing **switch** or iteration statement.

6.8.6.4 The **return** statement

Constraints

1 A **return** statement with an expression shall not appear in a function whose return type is **void**. A **return** statement without an expression shall only appear in a function whose return type is **void**.

2 For a function that has an underspecified definition, all **return** statements shall provide expressions with a consistent type or none at all. That is, if any **return** statement has an expression, all **return** statements shall have an expression (after lvalue, array-to-pointer or

[166)] Following the `contin`: label is a null statement.

function-to-pointer conversion) with the same type; otherwise all **return** expressions shall have no expression.

Semantics

3 A **return** statement ~~terminates~~ evaluates the expression, if any, terminates the execution of the ~~current function~~ function body and returns control to ~~its caller. A function~~ the caller. A function body may have any number of **return** statements.

4 If a **return** statement with an expression is executed, the value of the expression is returned to the caller as the value of the function call expression. If the expression has a type different from the return type of the function in which it appears, the value is converted as if by assignment to an object having the return type of the function.[167)]

5 For a function that has an underspecified definition, the return type is determined by the lexically first **return** statement, if any, that is associated to the function body and is specified as the type of that expression, if any, after lvalue, array-to-pointer, function-to-pointer conversion, or as **void** if there is no expression or no **return** statement.

6 **EXAMPLE** In:

```
struct s { double i; } f(void);
union {
      struct {
            int f1;
            struct s f2;
      } u1;
      struct {
            struct s f3;
            int f4;
      } u2;
} g;

struct s f(void)
{
      return g.u1.f2;
}

/* ... */
g.u2.f3 = f();
```

even though members `f2` and `f3` may overlap there is no undefined behavior, although there would be if the assignment were done directly (without using a function call to fetch the value).

[167)] The **return** statement is not an assignment. The overlap restriction of 6.5.16.1 does not apply to the case of function return. The representation of floating-point values can have wider range or precision than implied by the type; a cast can be used to remove this extra range and precision.

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~~ specifier **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.[168)]

6.9.1 Function definitions

Syntax

1 *function-definition:*
declaration-specifiers declarator declaration-list$_{\text{opt}}$ *compound-statement*

declaration-list:
declaration
declaration-list declaration

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.[169)]

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**, possibly combined with **auto**.

5 If the declarator includes a parameter type list, 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.

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

6 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.

7 An underspecified function definition shall contain an **`auto`** storage class specifier. The return type for such a function is determined as described for the **`return`** statement (6.8.6.4) and shall be visible prior to the function definition.

Semantics

8 If **`auto`** appears as a storage-class specifier it is ignored for the purpose of determining a storage class or linkage of the function. It then only indicates that the return type of the function may be inferred from **`return`** statements or the lack thereof, see 6.8.6.4.

9 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 types of all the parameters; such a declarator (possibly adjusted by an inferred type specifier) 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,[170] the types of the parameters shall be declared in a following declaration list. In either case, the type of each parameter is adjusted as described in 6.7.6.3 for a parameter type list; the resulting type shall be a complete object type.

10 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.

11 Each parameter has automatic storage duration; its identifier is an lvalue.[171] The layout of the storage for parameters is unspecified.

12 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.)

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

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

15 Provided the constraints above are respected, the return type of an underspecified function definition is adjusted as if the corresponding type specifier had been inserted in the definition. The type of such a function is incomplete within the function body until the lexically first **`return`** statement that it contains, if any, or until the end of the function body, otherwise.[172]

16 **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
```

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

170) See “future language directions” (6.11.7).

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

172) This means that such a function cannot be used for direct recursion before or within the first return statement.

17 **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; }
```

is the function body. The following similar definition uses the identifier-list form for the parameter declarations:

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

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.

18 **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)(); ...*/
}
```

19 **EXAMPLE 3** Consider the following function that computes the maximum value of two parameters that have integer types T and S.

```
   inline auto max(T, S); // invalid: no definition visible
   ...
   inline auto max(T a, S b){
     return (a < 0)
            ? ((b <  0) ? ((a < b) ? b : a) : b)
            : ((b >= 0) ? ((a < b) ? b : a) : a);
   }
   ...
   // valid: definition visible
   extern auto max(T, S); // forces definition to be external
   auto max(T, S);        // same
   auto max();            // same
```

The **return** expression performs default arithmetic conversion to determine a type that can hold the maximum value and is at least as wide as **int**. The function definition is adjusted to that return type. This property holds regardless if types T and S have the same or different signedness.

The first forward declaration of the function is invalid, because an **auto** type function declaration that is not a definition is only valid if the definition of the function is visible. In contrast to that, the **extern** declaration and the two following equivalent ones are valid because they follow the definition and thus the inferred return type is known. Thereby in is ensured that the translation unit provides an external definition of the function.

20 **EXAMPLE 4** The following function computes the sum over an array of integers of type T and returns the value as the promoted type of T.

```
inline
auto sum(size_t n, T A[n]){
  switch(n) {
    case 0:
      return +((T)0);                          // return the promoted type
    case 1:
      return +A[0];                            // return the promoted type
    default:
      return sum(n/2, A) + sum(n - n/2, &A[n/2]);  // valid recursion
  }
}
```

If instead sum would have bee defined with a prototype as follows

```
T sum(size_t n, T A[n]);
```

for a narrow type T such as **unsigned char**, the return type and result would be different from the previous. In particular, the result of the addition would have been converted back from the promoted type to T before each **return**, possibly leading to a surprising overall results. Also, specifying the promoted type of a narrow type T explicitly can be tedious because that type depends on properties of the execution platform.

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.

4 **EXAMPLE 1**

```
int i1 = 1;         // definition, external linkage
static int i2 = 2;  // definition, internal linkage
extern int i3 = 3;  // definition, external linkage
int i4;             // tentative definition, external linkage
static int i5;      // tentative definition, internal linkage

int i1;             // valid tentative definition, refers to previous
int i2;             // 6.2.2 renders undefined, linkage disagreement
int i3;             // valid tentative definition, refers to previous
int i4;             // valid tentative definition, refers to previous
int i5;             // 6.2.2 renders undefined, linkage disagreement

extern int i1;      // refers to previous, whose linkage is external
```