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

N2633 v1
D2303R0

2021-1-20

Function literals and value closures v.1

proposal for C23

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

We propose the inclusion of simple lambda expressions into the C standard. We build on a slightly restricted syntax of that feature in C++. In particular, they only have immutable value captures, fully specified parameter types, and, based on N2632, the return type is inferred from **return** statements. This is part of a series of papers for the improvement of type-generic programming in C that has been introduced in N2638. Follow-up papers N2634 and N2635 will extend this feature with **auto** parameter types and lvalue captures, respectively.

I. MOTIVATION

In N2638 it is argued that the features presented in this paper are useful in a more general context, namely for the improvement of type-generic programming in C. We will not repeat this argumentation here, but try to motivate the introduction of lambdas as a stand-alone addition to C.

When programming in C we are often confronted with the need of specifying small functional units that

— are to be reused in several places
— are to be passed as argument to another function
— need a fine control of data in- and outflow.

The smallest unit currently is the specification of a function, that is a top-level named entity with identified parameters for input and output. Current C provides several mechanisms to ease the specification of such small functions:

— The possibility to distinguish internal and external linkage via a specification with **static** (or not).
— The possibility to add function definitions to header files and thus to make the definitions and not only the interface declaration available across translation units via the **inline** mechanism.
— The possibility to add additional properties to functions via the attribute mechanism.

All these mechanisms are relatively rigid:

(1) They require a naming convention for the function.
(2) They require a specification far away and ahead of the first use.
(3) They treat all information that is passed from the caller to the function as equally important.

As an example, take the task of specifying a comparison function for strings to **qsort**. There is already such a function, **strcmp**, in the C library that is almost fit for the task, only that its prototype is missing an indirection. The semantically correct comparison function could look something like this:

```
  int strComp(char* const* a, char* const* b) {
    return strcmp(*a, *b);
  }
```

Although probably for most existing ABI its call interface could be used as such (if **char*** **const*** and **void const*** have the same representation) the use of it in the following call is a constraint violation:

```
#define NUMEL 256
char* stringArray[NUMEL] = { "hei", "you", ... };

  ...
  qsort(stringArray, NUMEL, sizeof(char*),
        strComp); // mismatch, constraint violation
  ...
```

The reflex of some C programmers will perhaps be to paint over this by using a cast:

```
  ...
  qsort(stringArray, NUMEL, sizeof(char*),
        (int(*)(void const*, void const*))strComp); // UB
  ...
```

This does not only make the code barely readable, but also just introduces undefined behavior instead of a constraint violation. On the other hand, on many platforms the behavior of this code may indeed be well defined, because finally the ABI of strComp is the right one. Unfortunately there is no way for the programmer to know that for all possible target platforms.

So the “official” strategy in C is to invent yet another wrapper:

```
int strCompV(void const* a, void const* b) {
  return strComp(a, b);
}

  ...
  qsort(stringArray, NUMEL, sizeof(char*),
        strCompV); // OK
  ...
```

This strategy has the disadvantages (1) and (2), but on most platforms it will also miss optimization opportunities:

— Since strCompV is specified as a function its address must be unique. The caller cannot inspect **qsort**, it cannot know if strCompV and strComp must have different addresses. Thus we are forcing the creation of a function that only consists of code duplication.
— If the two functions are found in two different translation units, strCompV will just consist of a tail call to strComp and thus create a useless indirection for every call within **qsort**.

C++'s lambda feature that we propose to integrate into C allows the following simple specification:

```
...
qsort(stringArray, NUMEL, sizeof(char*),
      [](void const* a, void const* b){
        return strComp(a, b);
      });
...
```

By such a specification of a lambda we do not only avoid (1) and (2), but we also leave it to the discretion of the implementation if this produces the a new function with a different address or if the tail call is optimized at the call site and the address of `strComp` is used instead.

Altogether, the improvements that we want to gain with this feature are:

— Similar to compound literals, avoid useless naming conventions for functions with a local scope (anonymous functions).
— Avoid to declare and define small functions far from their use.
— Allow the compiler to reuse functions that have the same functionality and ABI.
— Split interface specifications for such small functions into an invariant part (captures) and into a variable part (parameters).
— Strictly control the in- and outflow of data into specific functional units.
— Provide more optimization opportunities to the compiler, for example better tail call elimination or JIT compilation of code snippets for fixed run-time values.

II. DESIGN CHOICES

II.1. Expression versus function definition

Currently, the `C` standard imposes to use named callbacks for small functional units that would be used by `C` library functions such as **`atexit`** or **`qsort`**. Where inventing a name is already an unnecessary burden to the programming of small one-use functionalities, the distance between definition and use is a real design problem and can make it difficult to enforce consistency between a callback and a call. Already for the `C` library itself this is a real problem, because function arguments are even reinterpreted (transiting through **`void const`**`*`) by a callback to **`qsort`**, for example. The situation is even worse, if input data for the function is only implicitly provided by access to global variables as for **`atexit`**.

Nested functions improve that situation only marginally: definition and use are still dissociated, and access to variables from surrounding scopes can still be used within the local function. In many cases the situation can even be worse than for normal functions, because variables from outside that are accessed by nested functions may have automatic storage duration. Thus, nested functions may access objects that are already dead when they are called, making the behavior of the execution undefined.

For these reasons we opted for an expression syntax referred to as *lambda*. This particular choice not withstanding we think that it should still be *possible* to name a local functionality if need be, and to reuse it in several places of the same program. Therefore, lambdas still allow to manipulate *lambda values*, the results of a lambda expresssion, and in particular that these values are assigned to objects of lambda type.

II.2. Capture model

For the possible visibility of types and objects inside the body of a lambda, the simplest is to apply the existing scope model. This is what is chosen here (consistently with `C++`) for all use of types and objects that do not need an evaluation.

— All visible types can be used, if otherwise permitted, as type name in within **`alignof`**, **`alignas`** or **`sizeof`** expressions, type definitions, generic choice expressions, casts or compound literals, as long as they do not lead to an evaluation of a variably modified type.
— All visible objects can be used within the controlling expression of **`_Generic`**, within **`alignof`** expressions, and, if they do not have a variably modified type, within **`sizeof`** expressions.

In contrast to that and as we have discussed in N2638, there are four possible design choices for the *access* of automatic variables that are visible at the point of the evaluation of a lambda expression. We don't think that there is any "natural" choice among these, but that for a given lambda the choice has to depend on several criteria, some of which are general (such as personal preferences or coding styles) and some of which are special (such as a subsequent modification of the object or optimization questions).

As a consequence, we favor a solution that leaves the principal decision if a capture is a value capture or an lvalue capture to the programmer of the lambda; it is only they who can appreciate the different criteria. For this particular paper, we put the question on how lvalue captures should be be handled aside and only introduce value captures.[1] Nevertheless we think that the choice of explicit specification of value captures as provided by `C++` lambdas is preferable to the implicit use of value captures for all automatic variables as in Objective C's blocks, or of lvalue captures as for `gcc`'s compound expression or nested functions.[2]

II.3. Call sequence

As for all papers in this series, we intend not to impose ABI changes to implementations. We chose a specification for a call sequence for lambdas that either uses an existing function call ABI or encapsulates all calls to lambdas within a given translation unit.

For function literals, that is lambdas that have no captures, we impose that they should be convertible to function pointers with the same prototype. It is easy to see that such a lambda can be rewritten to a static function with an auxiliary name which then is used in place of the lambda expression itself.

For closures, that is lambdas with captures, the situation is a bit more complicated. Where some implementations, building for example upon `gcc`'s nested functions, may prefer to use the same calling sequence as for functions, others may want to evaluate captures directly in place and use an extended ABI to call a derived function interface or pass information for the captures implicitly in a special register.

Therefore, our proposal just adds lambda values to the possibilities of the postfix expression (LHS) of a function call, and imposes no further restrictions how this feature is to be implemented.

[1]Lvalue captures will be proposed in N2635.
[2]These different possibilities have been discussed in N2638.

II.4. Interoperability

The fact that objects with lambda type can be defined and may have external linkage, could imply that such lambda objects are made visible between different translation units. If that would be possible, implementations would be forced to extend ABIs with the specification of lambda types, and platforms that have several interoperable implementations would have to agree on such ABI.

To require such an ABI specifiction would have several disadvantages:

— A cross-implementation effort of for an ABI extension would incur a certain burden for implementations.
— Many different ABI are possible, in particular special cases have a certain of potential for optimization. Fixing an ABI too early, forces implementations to give stability guarantees for the interface.

For our proposal here, we expect that most lambda expressions that appear in file scope will be function literals. Since function literals can be converted to function pointers, no special syntax is needed to make their functionalities available to other translation units.

Because there are no objects with automatic storage duration in file scope, the only captures that can be formed in file scope are those that are derived from expressions, and these expression must have a value that can be determined at translation time. We think that it should be possible to define most such captures as lambda-local unmutable objects with static storage duration, and thus, in general such lambdas are better formulated as function literals.

To be accessible in another translation unit a closure expression that is evaluated in block scope, would have to be assigned to a global variable of lambda type. We inhibit this by not specifying a declaration syntax for lambdas. Thereby the only possibility to declare an object of lambda type is to use **auto**, and thus each such declaration must also be a definition such that the full specification of the lambda expression is visible. But then, no translation unit can declare an object of lambda value with external linkage that is not already a definition.

II.5. Invariability

Since lambdas will often concern small functional units, our intent is that implementations use all the means available to optimize them, as long as the security of the execution can be guaranteed. Therefore we will enforce that lambda values, once they are stored in an object, will be known to never change. This will inhibit, e.g, that implementation specific functions or jump targets will change between calls to the same lambda value, or that any lambda value can escape to a context where its originating lambda expression is not known.

II.6. Recursion

Since there is no syntax to forward-declare a lambda and they cannot be assigned, a lambda cannot refer to itself (same lambda value and type), neither directly nor indirectly by calling other functions or lambdas. The only possibility is for function literals, when they are converted and assigned to function pointers. Such a function pointer can then be used directly or indirectly as any other function pointer, also by the function literal expression that gave rise to its conversion.

```
// file scope definition
```

```
static int (*comp)(void const*, void const*) = 0;
...
int main(void) {
  ...
  comp = [](void const* A, void const* B){
    if (something) {
      return 0;
    } else {
      return comp(B, A);
    }
  }
  ...
}
```

Such examples for function literals are a bit contrived, and will probably not be very common.

In contrast to that, closures cannot be called recursively because they don't even convert to function pointers. This is a conscious decision for this paper, because we don't want to constrain implementations in the way(s) they reserve the storage that is necessary to hold captures, and how they implement captures in general. For example, closures that return **`void`** can be implement relatively simple as-if by adding some small state, an entry label, one return label per call, and some switched or computed **`goto`** statements.

As a consequence, the maximum storage that is needed for the captures of a given closure can be computed at translation time, and no additional mechanism to handle dynamic storage is necessary.

III. SYNTAX AND TERMINOLOGY

For all proposed wording see Section VII.

III.1. Lambda expressions

Since it is the most flexible and expressive, we propose to adopt `C++` syntax for lambdas, `6.5.2.6 p1`, as a new form of postfix expression (`6.5.2 p1`) introducing the terms *lambda expression*, *capture clause*, *capture list*, *capture default*, *value capture*, *capture* and *parameter clause*.

We make some exceptions from that `C++` syntax for the scope of this paper:

(1) We omit the possibility to specify the return type of a lambda. The corresponding `C++` syntax

```
    -> return-type
```

reuses the **`->`** token in an unexpected way, and is not strictly necessary if we have **`auto`** return. If WG14 wishes so, this feature could be added easily in the future as a general function return type syntax.

(2) We omit the possibility to specify all value captures as mutable. The `C++` syntax introduces a keyword, `mutable`, that would be new to `C`. We don't see enough gain that would justify the introduction of a new keyword.

(3) For the simplicity of this proposal we omit lvalue captures and lvalue aliases. A follow-up paper, N2635, takes care of lvalue captures. The introduction of lvalue aliases (`C++`'s references) is not currently planned.

As this syntax leaves the parameter clause as optional, `6.5.2.6 p7` fixes the semantics for this case to be equivalent to an empty parameter list, and also introduces the terminology of *function literal* (no captures) and *closure* (any capture).

Also, `6.5.2.6 p3` introduces a distinction between *explicit captures*, that are captures that are explicitly listed in the capture list, and *implicit captures*, that are automatic variables of a surrounding scope that are caught because the capture clause is **`[=]`**.

The terminology for *lambda values* and *lambda types* and their *prototype* is introduced with the other type categories in `6.2.5 p20`, and then later specified in the clause for lambda expressions, `6.5.2.6 p11`.

III.2. Adjustments to other constructs

With the introduction of lambda expressions, functions bodies can now be nested and several standard constructs become ambiguous. Therefore it is necessary to adjust the definitions of these constructs and relate them to the nearest other constructs to which they could refer. This ensures that their use remains unique and well defined, and that no jumps across boundaries of function bodies are introduced.

— For labels we enforce that they are anchored within the nearest function body in which they appear:
  — Function scope as the scope for labels must only extend to the innermost function body in which a label is found and such function scopes are *not* nested (`6.2.1 p3`).
  — Case labels must be found within a corresponding **`switch`** statement of their innermost function body (`6.8.1 p2`).

— **`continue`** and **`break`** statements must match to a loop or **`switch`** construct that is found in the innermost function body that contains them (`6.8.6.2 p1` and `6.8.6.3 p1`).

— A **`return`** statement also has to be associated to the innermost function body. It has to return a value, if any, according to the type of that function body. Also, if its function body is associated to a lambda, it only has to terminate the corresponding call to the lambda, and not the surrounding function (`6.8.6.4 p3`).

IV. SEMANTICS

The principal semantic of lambda expressions themselves is described in `6.2.5.6 p8`. Namely, it describes how lambda expressions are similar to functions concerning the scope of visibility and the lifetime of captures and parameters.

Captures are handled in two paragraphs, but the main feature is the description of the evaluation that provides values for value captures, `6.5.2.6 p10`. It stipulates that their values are determined at the point of evaluation of the lambda expression (basically in order of declaration), that the value undergoes lvalue, array-to-pointer or function-to-pointer conversion if necessary, and that the type of the capture then is the type of the expression *after* that conversion, that is without any qualification or atomic derivation, and, that it gains a **`const`** qualification. Additionally, we insist that the so-determined value of a value capture will and cannot not change by any means and is the same during all evaluations during all calls to the same lambda.

Another paragraph, `6.5.2.6 p9`, describes how the two forms of value captures relate, namely that the form without assignment expression is really a short form that evaluates an automatic variable of the surrounding scope of the same name.

The other specifications for lambda expressions are then their use in different contexts.

— Function literals may be converted to function pointers, `6.3.2.1 p5`. For these this is easily possible because they have exactly the same functionality as functions: all additional caller information is transferred by arguments to the call. Thus the existing function ABI can be used to call a function literal, and the translator has in fact all information to provide such a call interface.
— As postfix expression within function calls they can take the place that previously only had function pointers. If we would not provide the possibility of captures, the corresponding function literals could all first be converted to function literals (see above) and called then. But since we don't want to impose how lambda-specific capture information is transferred during a call and to guarantee the properties specified in II.3 above, we just add lambdas to the possibilities of the postfix expression that describes the called function.[3]

V. CONSTRAINTS AND REQUIREMENTS

As a general policy, we try to fix as much requirements as possible through constraints, either with specific syntax or explicit constraints. Only if a requirement is not (or hardly) detectable at translation time, or if we want to leave design space to implementations, we formulate it as an imperative, indicating that the behavior then is undefined by the `C` standard.

— Captures are introduced to handle objects of automatic storage duration, all other categories of objects and functions are to use other mechanisms of access within lambdas. Therefore, we constrain captures to names of objects of automatic storage duration (`6.5.2.6 p4`) and limit the evaluation of all such objects from a surrounding scope to the initialization of captures (`6.5.2.6 p5`). All such evaluations thus take place during the evaluation of the lambda expression itself, not during a subsequent call to the lambda value.
— Since an automatic object of array type would evaluate to a pointer type, it would give rise to a capture of a different type than in the surrounding scope. Therefore in `6.5.2.6 p4` we also add a constraint that forbids array types for captures (explicit or implicit) without assignment expression. It is possible to overwrite that constraint by explicitly specifying a capture of the form `id = id`, even if `id` has an array type; within the lambda expression `id` then has pointer type and retrieving the size of the underlying array is not possible.[4]
— Calling a closure needs additional information, namely the transfer of lambda-specific values for captures. In `6.3.2.1 p5` we explicitly call out the fact that converting closures to function pointers is not defined by the text. This would also follow as implicit undefined behavior from the following text, but we found it important to point this out and thereby guide the expectations of programmers.
— A **`switch`** label should not enable control flow that jumps from the controlling expression of the **`switch`** into a lambda. The corresponding property is syntactic and can be checked at translation time. Therefore we formulate this as a constraint in `6.8.1 p2`.
— Labels should not be used to bypass the calling sequence (capture and parameter instanciation) and jump into a lambda. Therefore we constrain the visibility scope of labels to

[3]A similar addition for function designators could also be made, see [Gustedt 2016].
[4]Arrays themselves can be accessed as lvalue captures that will be introduced in N2635.

the surrounding function body, `6.2.1 p3`. With these constraints, no **`goto`** statement can be formed that jumps into or out of a lambda or into a different function.

— Similarly, all jump statements other than **`return`** should never attempt to jump into or out of the nearest enclosing function body. To ensure this we add an explicit constraint as `6.8.6 p2`, and in `6.8.6.2 p1` and `6.8.6.3 p1`.
— According to II.5 we don't want lambda values to be modified. If they were specified from scratch, this would probably be reflected in both, a constraint and a requirement. But since we want to be able to leave the possibility that lambda values are implemented as function pointers (in particular for function literals) we cannot make this a requirement. Therefore, we only introduce a requirement (`6.5.2.6 p11` last sentence) and recommended practice for applications to use a **`const`** qualification and for implementations to diagnose modifications when possible (`6.5.2.6 p12`).
— There is no direct syntax to declare lambda types, and so objects of lambda type can only be declared (and defined) through type inference. The necessary adjustments to that feature are integrated to the constraints of `6.7.10 p4`.

VI. QUESTIONS FOR WG14

(1) Does WG14 want the lambda feature for C23 along the lines of N2633?
(2) Does WG14 want to integrate the changes as specified in N2633 into C23?

References

Jens Gustedt. 2016. *The register overhaul*. Technical Report N2067. ISO. available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2067.pdf.

Jens Gustedt. 2021a. *Function literals and value closures*. Technical Report N2633. ISO. available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2633.pdf.

Jens Gustedt. 2021b. *Improve type generic programming*. Technical Report N2638. ISO. available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2638.pdf.

Jens Gustedt. 2021c. *Lvalue closures*. Technical Report N2635. ISO. available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2635.pdf.

Jens Gustedt. 2021d. *Type-generic lambdas*. Technical Report N2634. ISO. available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2634.pdf.

Jens Gustedt. 2021e. *Type inference for variable definitions and function return*. Technical Report N2632. ISO. available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2632.pdf.

VII. PROPOSED WORDING

The proposed text is given as diff against N2632.

— 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 body in which it appears, and is declared implicitly by its syntactic appearance (followed by a : and a statement). Each function body has a function scope that is separate from the function scope of any other function body. In particular, a label is visible in exactly one function scope (the innermost function body in which it appears) and distinct function bodies may use the same identifier to designate different labels.[29)]

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.[30)] 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.

[29)] As a consequence, it is not possible to specify a **`goto`** statement that jumps into or out of a lambda or into another function.

[30)] Identifiers that are defined in the parameter list of a lambda expression do not have prototype scope, but a scope that comprises the whole body of the lambda.

— A *structure type* describes a sequentially allocated nonempty set of member objects (and, in certain circumstances, an incomplete array), each of which has an optionally specified name and possibly distinct type.

— A *union type* describes an overlapping nonempty set of member objects, each of which has an optionally specified name and possibly distinct type.

— A *function type* describes a function with specified return type. A function type is characterized by its return type and the number and types of its parameters. A function type is said to be derived from its return type, and if its return type is *T*, the function type is sometimes called "function returning *T*". The construction of a function type from a return type is called "function type derivation".

— A *lambda type* is a complete object type that describes the value of a lambda expression. A lambda type is characterized but not determined by a return type that is inferred from the function body of the lambda expression, and by the number, order, and type of parameters that are expected for function calls. The function type that has the same return type and list of parameter types as the lambda is called the *prototype* of the lambda.

— A *pointer type* may be derived from a function type or an object type, called the *referenced type*. A pointer type describes an object whose value provides a reference to an entity of the referenced type. A pointer type derived from the referenced type *T* is sometimes called "pointer to *T*". The construction of a pointer type from a referenced type is called "pointer type derivation". A pointer type is a complete object type.

— An *atomic type* describes the type designated by the construct **`_Atomic`**(*type-name*). (Atomic types are a conditional feature that implementations need not support; see 6.10.8.3.)

These methods of constructing derived types can be applied recursively.

21 Arithmetic types and pointer types are collectively called *scalar types*. Array and structure types are collectively called *aggregate types*.[50)]

22 An array type of unknown size is an incomplete type. It is completed, for an identifier of that type, by specifying the size in a later declaration (with internal or external linkage). A structure or union type of unknown content (as described in 6.7.2.3) is an incomplete type. It is completed, for all declarations of that type, by declaring the same structure or union tag with its defining content later in the same scope.

23 A type has *known constant size* if the type is not incomplete and is not a variable length array type.

24 Array, function, and pointer types are collectively called *derived declarator types*. A *declarator type derivation* from a type *T* is the construction of a derived declarator type from *T* by the application of an array-type, a function-type, or a pointer-type derivation to *T*.

25 A type is characterized by its *type category*, which is either the outermost derivation of a derived type (as noted above in the construction of derived types), or the type itself if the type consists of no derived types.

26 Any type so far mentioned is an *unqualified type*. Each unqualified type has several *qualified versions* of its type,[51)] corresponding to the combinations of one, two, or all three of the **`const`**, **`volatile`**, and **`restrict`** qualifiers. The qualified or unqualified versions of a type are distinct types that belong to the same type category and have the same representation and alignment requirements.[52)] A derived type is not qualified by the qualifiers (if any) of the type from which it is derived.

27 Further, there is the **`_Atomic`** qualifier. The presence of the **`_Atomic`** qualifier designates an atomic type. The size, representation, and alignment of an atomic type need not be the same as those of the corresponding unqualified type. Therefore, this document explicitly uses the phrase "atomic,

[50)]Note that aggregate type does not include union type because an object with union type can only contain one member at a time.

[51)]See 6.7.3 regarding qualified array and function types.

[52)]The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.

3 Except when it is the operand of the **sizeof** operator, or the unary & operator, or is a string literal used to initialize an array, an expression that has type "array of *type*" is converted to an expression with type "pointer to *type*" that points to the initial element of the array object and is not an lvalue. If the array object has register storage class, the behavior is undefined.

4 A *function designator* is an expression that has function type. Except when it is the operand of the **sizeof** operator,[70] or the unary & operator, a function designator with type "function returning *type*" is converted to an expression that has type "pointer to function returning *type*".

5 Closures shall not be converted to any other object type. A function literal with a type "lambda with prototype *type*" can be converted implicitly or explicitly to an expression that has type "pointer to *type*".[71] The function pointer value behaves as if a function with internal linkage with the appropriate prototype, a unique name, and the same function body as for λ had been specified in the translation unit and the function pointer had been formed by function-to-pointer conversion of that function. The only difference is that the function pointer needs not necessarily to be distinct from any other compatible function pointer that provides the same observable behavior.

Forward references: lambda expressions (6.5.2.6) address and indirection operators (6.5.3.2), assignment operators (6.5.16), common definitions `<stddef.h>` (7.19), initialization (6.7.9), postfix increment and decrement operators (6.5.2.4), prefix increment and decrement operators (6.5.3.1), the **sizeof** and **_Alignof** operators (6.5.3.4), structure and union members (6.5.2.3).

6.3.2.2 void

1 The (nonexistent) value of a *void expression* (an expression that has type **void**) shall not be used in any way, and implicit or explicit conversions (except to **void**) shall not be applied to such an expression. If an expression of any other type is evaluated as a void expression, its value or designator is discarded. (A void expression is evaluated for its side effects.)

6.3.2.3 Pointers

1 A pointer to **void** may be converted to or from a pointer to any object type. A pointer to any object type may be converted to a pointer to **void** and back again; the result shall compare equal to the original pointer.

2 For any qualifier *q*, a pointer to a non-*q*-qualified type may be converted to a pointer to the *q*-qualified version of the type; the values stored in the original and converted pointers shall compare equal.

3 An integer constant expression with the value 0, or such an expression cast to type **void** *, is called a *null pointer constant*.[72] If a null pointer constant is converted to a pointer type, the resulting pointer, called a *null pointer*, is guaranteed to compare unequal to a pointer to any object or function.

4 Conversion of a null pointer to another pointer type yields a null pointer of that type. Any two null pointers shall compare equal.

5 An integer may be converted to any pointer type. Except as previously specified, the result is implementation-defined, might not be correctly aligned, might not point to an entity of the referenced type, and might be a trap representation.[73]

6 Any pointer type may be converted to an integer type. Except as previously specified, the result is implementation-defined. If the result cannot be represented in the integer type, the behavior is undefined. The result need not be in the range of values of any integer type.

7 A pointer to an object type may be converted to a pointer to a different object type. If the resulting pointer is not correctly aligned[74] for the referenced type, the behavior is undefined. Otherwise,

[70] Because this conversion does not occur, the operand of the **sizeof** operator remains a function designator and violates the constraints in 6.5.3.4.

[71] It follows that lambdas of different type cannot be assigned to each other. Thus, in the conversion of a function literal to a function pointer, the prototype of the originating lambda expression can be assumed to be known, and a diagnostic can be issued if the prototypes do not aggree.

[72] The macro **NULL** is defined in `<stddef.h>` (and other headers) as a null pointer constant; see 7.19.

[73] The mapping functions for converting a pointer to an integer or an integer to a pointer are intended to be consistent with the addressing structure of the execution environment.

[74] In general, the concept "correctly aligned" is transitive: if a pointer to type A is correctly aligned for a pointer to type B, which in turn is correctly aligned for a pointer to type C, then a pointer to type A is correctly aligned for a pointer to type C.

`default` **:** *assignment-expression*

Constraints

2 A generic selection shall have no more than one **`default`** generic association. The type name in a generic association shall specify a complete object type other than a variably modified type. No two generic associations in the same generic selection shall specify compatible types. The type of the controlling expression is the type of the expression as if it had undergone an lvalue conversion,[99] array to pointer conversion, or function to pointer conversion. That type shall be compatible with at most one of the types named in the generic association list. If a generic selection has no **`default`** generic association, its controlling expression shall have type compatible with exactly one of the types named in its generic association list.

Semantics

3 The controlling expression of a generic selection is not evaluated. If a generic selection has a generic association with a type name that is compatible with the type of the controlling expression, then the result expression of the generic selection is the expression in that generic association. Otherwise, the result expression of the generic selection is the expression in the **`default`** generic association. None of the expressions from any other generic association of the generic selection is evaluated.

4 The type and value of a generic selection are identical to those of its result expression. It is an lvalue, a function designator, or a void expression if its result expression is, respectively, an lvalue, a function designator, or a void expression.

5 **EXAMPLE** The **`cbrt`** type-generic macro could be implemented as follows:

```
      #define cbrt(X) _Generic((X),                  \
                              long double: cbrtl,     \
                              default: cbrt,          \
                              float: cbrtf            \
                              )(X)
```

6.5.2 Postfix operators

Syntax

1 *postfix-expression:*
primary-expression
postfix-expression **[** *expression* **]**
postfix-expression **(** *argument-expression-list*$_{opt}$ **)**
postfix-expression **.** *identifier*
postfix-expression **->** *identifier*
postfix-expression **++**
postfix-expression **-**
(*type-name* **)** **{** *initializer-list* **}**
(*type-name* **)** **{** *initializer-list* **,** **}**
lambda-expression

argument-expression-list:
assignment-expression
argument-expression-list **,** *assignment-expression*

6.5.2.1 Array subscripting

Constraints

1 One of the expressions shall have type "pointer to complete object *type*", the other expression shall have integer type, and the result has type "*type*".

[99] An lvalue conversion drops type qualifiers.

Semantics

2 A postfix expression followed by an expression in square brackets `[]` is a subscripted designation of an element of an array object. The definition of the subscript operator `[]` is that `E1[E2]` is identical to `(*((E1)+(E2)))`. Because of the conversion rules that apply to the binary `+` operator, if `E1` is an array object (equivalently, a pointer to the initial element of an array object) and `E2` is an integer, `E1[E2]` designates the `E2` -th element of `E1` (counting from zero).

3 Successive subscript operators designate an element of a multidimensional array object. If `E` is an n-dimensional array ($n \geq 2$) with dimensions $i \times j \times \cdots \times k$, then `E` (used as other than an lvalue) is converted to a pointer to an $(n-1)$-dimensional array with dimensions $j \times \cdots \times k$. If the unary $*$ operator is applied to this pointer explicitly, or implicitly as a result of subscripting, the result is the referenced $(n-1)$-dimensional array, which itself is converted into a pointer if used as other than an lvalue. It follows from this that arrays are stored in row-major order (last subscript varies fastest).

4 **EXAMPLE** Consider the array object defined by the declaration

```
      int x[3][5];
```

Here `x`

is a 3×5 array of

`int` s; more precisely, `x` is an array of three element objects, each of which is an array of five **`int`** s. In the expression `x[i]`, which is equivalent to `(*((x)+(i)))`, `x` is first converted to a pointer to the initial array of five **`int`** s. Then `i` is adjusted according to the type of `x`, which conceptually entails multiplying `i` by the size of the object to which the pointer points, namely an array of five **`int`** objects. The results are added and indirection is applied to yield an array of five **`int`** s. When used in the expression `x[i][j]`, that array is in turn converted to a pointer to the first of the **`int`** s, so `x[i][j]` yields an **`int`**.

Forward references: additive operators (6.5.6), address and indirection operators (6.5.3.2), array declarators (6.7.6.2).

6.5.2.2 Function calls

Constraints

1 The ~~expression that denotes the called function~~postfix expression[100] shall have ~~type~~ lambda type or pointer to function type, returning **`void`** or returning a complete object type other than an array type.

2 If the ~~expression that denotes the called function has a type that~~ postfix expression is a lambda or if the type of the function includes a prototype, the number of arguments shall agree with the number of parameters of the function or lambda type. Each argument shall have a type such that its value may be assigned to an object with the unqualified version of the type of its corresponding parameter.

Semantics

3 A postfix expression followed by parentheses `()` containing a possibly empty, comma-separated list of expressions is a function call. The postfix expression denotes the called function or lambda. The list of expressions specifies the arguments to the function or lambda.

4 An argument may be an expression of any complete object type. In preparing for the call to a function, the arguments are evaluated, and each parameter is assigned the value of the corresponding argument.[101]

5 If the expression that denotes the called function has lambda type or type pointer to function returning an object type, the function call expression has the same type as that object type, and has the value determined as specified in 6.8.6.4. Otherwise, the function call has type **`void`**.

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

[100] Most often, this is the result of converting an identifier that is a function designator.

[101] A function or lambda can change the values of its parameters, but these changes cannot affect the values of the arguments. On the other hand, it is possible to pass a pointer to an object, and the function or lambda can then change the value of the object pointed to. A parameter declared to have array or function type is adjusted to have a pointer type as described in 6.9.1.

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 is a lambda or is a 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 lambda or 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 lambda or 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 or lambda 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 or lambdas.

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 of the designated member.

[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.

13 **EXAMPLE 6** Like string literals, const-qualified compound literals can be placed into read-only memory and can even be shared. For example,

```
        (const char [])(“abc”} == "abc"
```

might yield 1 if the literals' storage is shared.

14 **EXAMPLE 7** Since compound literals are unnamed, a single compound literal cannot specify a circularly linked object. For example, there is no way to write a self-referential compound literal that could be used as the function argument in place of the named object `endless_zeros` below:

```
        struct int_list { int car; struct int_list *cdr; };
        struct int_list endless_zeros = {0, &endless_zeros};
        eval(endless_zeros);
```

15 **EXAMPLE 8** Each compound literal creates only a single object in a given scope:

```
        struct s { int i; };

        int f (void)
        {
              struct s *p = 0, *q;
              int j = 0;

        again:
              q = p, p = &((struct s){ j++ });
              if (j  <  2) goto again;

              return p == q && q->i == 1;
        }
```

The function `f()` always returns the value 1.

16 Note that if an iteration statement were used instead of an explicit **goto** and a labeled statement, the lifetime of the unnamed object would be the body of the loop only, and on entry next time around `p` would have an indeterminate value, which would result in undefined behavior.

Forward references: type names (6.7.7), initialization (6.7.9).

6.5.2.6 Lambda expressions

Syntax

1 *lambda-expression:*
  capture-clause parameter-clause$_{opt}$ *attribute-specifier-sequence*$_{opt}$ *function-body*

capture-clause:
  [ ]
  [*capture-list* **]**
  [*capture-default* **]**

capture-list:
  value-capture
  capture-list **,** *value-capture*

capture-default:
  =

value-capture:
  capture
  capture **=** *assignment-expression*

capture:
  identifier

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

Constraints

2 A lambda expression shall not be operand of the unary & operator.[110)]

3 A capture that is listed in the capture list is an *explicit capture* . If the capture clause is `[=]`, `id` is the name of an object with automatic storage duration in a surrounding scope, `id` is used within the function body of the lambda without redeclaration and `id` is not a parameter, the effect is as if `id` had been used in a capture list. Such a capture is an *implicit capture* .

4 Captures without assignment expression shall be names of complete objects with automatic storage duration in a scope surrounding the lambda expression that do not have array type and that are visible at the point of evaluation of the lambda expression. An identifier shall appear at most once; either as an explicit capture or as a parameter name in the parameter type list.

5 Within the function body, identifiers (including explicit and implicit captures, and parameters of the lambda) shall be used according to the usual scoping rules, but identifiers of a scope that includes the lambda expression and that are declared with automatic storage duration shall only be evaluated within the assignment expression of a value capture.[111)]

6 The function body shall be such that a return type *type* according to the rules in 6.8.6.4 can be inferred.

Semantics

7 If the parameter clause is omitted, a clause of the form `()` is assumed. A lambda expression without capture list is called a *function literal expression* , otherwise it is called a *closure expression* . A lambda value originating from a function literal expression is called a *function literal* , otherwise it is called a *closure* .

8 Similar to a function definition, a lambda expression forms a single block scope that comprises its capture clause, its parameter clause and its function body. Each explicit capture and parameter has a scope of visibility that starts immediately after its definition is completed and extends to the end of the function body. The scope of visibility of implicit captures is the function body. In particular, captures and parameters are visible throughout the whole function body, unless they are redeclared in a depending block within that function body. Captures and parameters have automatic storage duration; in each function call to the formed lambda value, a new instance of each capture and parameter is created and initialized in order of declaration and has a lifetime until the end of the call, only that the address of captures is not necessarily unique.

9 If a capture `id` is defined without an assignment expression, the assignment expression is assumed to be `id` itself, referring to the object of automatic storage duration of the surrounding scope that exists according to the constraints.[112)]

10 The implicit or explicit assignment expression `E` in the definition of a value capture determines a value $\mathtt{E}_0$ with type $\mathtt{T}_0$, which is `E` after possible lvalue, array-to-pointer or function-to-pointer conversion. The type of the capture is $\mathtt{T}_0$ **`const`** and its value is $\mathtt{E}_0$ for all evaluations in all function calls to the lambda value. If, within the function body, the address of the capture `id` or one of its members is taken, either explicitly by applying a unary & operator or by an array to pointer conversion,[113)] and that address is used to modify the underlying object, the behavior is undefined.

[110)]Objects with lambda type that can be operand of the unary & operator can be formed by type inference and initialization with a lambda value.

[111)]Identifiers of visible automatic objects that are not captures, may still be used if they are not evaluated, for example in **`sizeof`** expressions (if they are not VM types) or as controlling expression of a generic primary expression.

[112)]The evaluation in rules in the next paragraph then stipulates that it is evaluated at the point of evaluation of the lambda expression, and that within the body of the lambda an unmutable **`auto`** object of the same name, value and type is made accesssible.

[113)]The capture does not have array type, but if it has a union or structure type, one of its members may have such a type.

The evaluation of E takes place during the evaluation of the lambda expression; for an explicit capture when the value capture is met and for an implicit capture at the beginning of the evaluation of the function body.

11 For each lambda expression, the return type *type* is inferred as indicated in the constraints. A lambda expression λ has an unspecified lambda type L that is the same for every evaluation of λ. If λ appears in a context that is not a function call, a value of type L is formed that identifies λ and the specific set of values of the identifiers in the capture clause for the evaluation, if any. This is called a *lambda value* . It is unspecified, whether two lambda expressions λ and κ share the same lambda type even if they are lexically equal but appear at different points of the program. Objects of lambda type shall not be modified.

Recommended practice

12 To avoid their accidental modification, it is recommended that declarations of lambda type objects are **const** qualified. Whenever possible, implementations are encouraged to diagnose any attempt to modify a lambda type object.

13 **EXAMPLE 1** The usual scoping rules extend to lambda expressions; the concept of captures only restricts which identifiers may be evaluated or not.

```
#include <stdio.h>
static long var;
int main(void) {
  [   ](void){ printf("%ld\n", var); }();                // valid, prints 0
  [var](void){ printf("%ld\n", var); }();                // invalid, var is static

  int var = 5;

  [var](void){ printf("%d\n", var); }();                 // valid, prints 5
  [   ](void){ printf("%d\n", var); }();                 // invalid
  [var](void){ printf("%zu\n", sizeof var); }();         // valid, prints sizeof(int)
  [   ](void){ printf("%zu\n", sizeof var); }();         // valid, prints sizeof(int)
  [   ](void){ extern long var; printf("%ld\n", var; }(); // valid, prints 0

}
```

14 **EXAMPLE 2** The following uses a function literal as a comparison function argument for **qsort**.

```
#define SORTFUNC(TYPE) [](size_t nmemb, TYPE A[nmemb]) {                  \
  qsort(A, nmemb, sizeof(A[0]),                                           \
        [](void const* x, void const* y){          /* comparison lambda  */ \
          TYPE X = *(TYPE const*)x;                                       \
          TYPE Y = *(TYPE const*)y;                                       \
          return (X < Y) ? -1 : ((X > Y) ? 1 : 0); /* return of type int */ \
        }                                                                 \
        );                                                                \
  return A;                                                               \
  }
  ...
  long C[5] = { 4, 3, 2, 1, 0, };
  SORTFUNC(long)(5, C);                        // lambda → (pointer →) function call
  ...
  auto const sortDouble = SORTFUNC(double);    // lambda value → lambda object
  double* (*sF)(size_t nmemb, double[nmemb]) = sortDouble;    // conversion
  ...
  double* ap = sortDouble(4, (double[]){ 5, 8.9, 0.1, 99, });
  double B[27] = { /* some values ... */ };
  sF(27, B);                                  // reuses the same function
  ...
  double* (*sG)(size_t nmemb, double[nmemb]) = SORTFUNC(double); // conversion
```

This code evaluates the macro SORTFUNC twice, therefore in total four lambda expressions are formed.

The function literals of the "comparison lambdas" are not operands of a function call expression, and so by conversion a pointer to function is formed and passed to the corresponding call of **qsort**. Since the respective captures are empty, the effect is as if to define two comparison functions, that could equally well be implemented as **static** functions with auxiliary names and these names could be used to pass the function pointers to **qsort**.

The outer lambdas are again without capture. In the first case, for **long**, the lambda value is subject to a function call, and it is unspecified if the function call uses a specific lambda type or directly uses a function pointer. For the second, a copy of the lambda value is stored in the variable sortDouble and then converted to a function pointer sF. Other than for the difference in the function arguments, the effect of calling the lambda value (for the compound literal) or the function pointer (for array B) is the same.

For optimization purposes, an implementation may fold lambda values that are expanded at different points of the program such that effectively only one function is generated. For example here the function pointers sF and sG may or may not be equal.

15 **EXAMPLE 3**

```
void matmult(size_t k, size_t l, size_t m,
             double const A[k][l], double const B[l][m], double const C[k][m]) {
  // dot product with stride of m for B
  // ensure constant propagation of l and m
  auto const λδ = [l,m](double const v[l], double const B[l][m], size_t m0) {
    double ret = 0.0;
    for (size_t i = 0; i < l; ++i) {
      ret += v[i]*B[i][m0];
    }
    return ret;
  };
  // vector matrix product
  // ensure constant propagation of l and m, and accessibility of λδ
  auto const λμ = [l, m, λδ](double const v[l], double const B[l][m], double res[m]) {
    for (size_t m0 = 0; m0 < m; ++m0) {
      res[m0] = λδ(v, B, m0);
    }
  };
  for (size_t k0 = 0; k0 < k; ++k0) {
    double const (*Ap)[l] = A[k0];
    double (*Cp)[m] = C[k0];
    λμ(*Ap, B, *Cp);
  }
}
```

This function evaluates two closures; $\lambda\delta$ has a return type of **double**, $\lambda\mu$ of **void**. Both lambda values serve repeatedly as first operand to function evaluation but the evaluation of the captures is only done once for each of the closures. For the purpose of optimization, an implementation could generate copies of the underlying functions for each evaluation of such a closure such that the values of the captures l and m are replaced on a machine instruction level.

6.5.3 Unary operators

Syntax

1 *unary-expression:*

postfix-expression
++ *unary-expression*
- *unary-expression*
unary-operator cast-expression
sizeof *unary-expression*
sizeof (*type-name* **)**
_Alignof (*type-name* **)**

unary-operator: one of
& * + - ˜ !

```
    struct T x = {.l = 43, .k = 42, };

    void f(void)
    {
          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).

6.7.10 Type inference

Constraints

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

2 For an underspecified declaration of a function that is also a definition, the return type shall be completed as of 6.9.1. For an underspecified declaration of a function that is not a definition a prior definition of the declared function shall be visible.

3 An underspecified declaration of an object that is also a definition and that is not the declaration of a parameter shall be of one of the forms

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

such that the declarator does not declare an array.

4 For an underspecified declaration such that the assignment expression does not have lambda type there shall be a type specifier *type* that can be inserted in the declaration immediately after the last storage class specifier that makes the adjusted declaration a valid declaration and such that the assignment expression, after possible lvalue, array-to-pointer or function-to-pointer conversion, has the non-atomic, unqualified type of the declared object.[164] if the assignment expression has lambda type, the declarator shall only consist of storage class specifiers, qualifiers and the identifier that is to be declared. A function declaration that is not a definition shall have a type that is compatible with the type of the corresponding definition.

Description

5 ~~Provided~~ Although there is no syntax derivation to form declarators of lambda type, values of lambda type can be used as assignment expression and the inferred type is that lambda type, possibly qualified. Otherwise, provided the constraints above are respected, in an underspecified declaration the type of the declared identifiers is the type after the declaration has been adjusted by *type*. The type of each identifier that declares an object is incomplete until the end of the assignment expression that initializes it.

6 **NOTE** 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.

[164] For most assignment expressions of integer or floating point type, there are several types *type* that would make such a declaration valid. The second part of the constraint ensures that among these a unique type is determined that does not need further conversion to be a valid initializer for the object.

6.8 Statements and blocks

Syntax

1 *statement:*

> *labeled-statement*
> *compound-statement*
> *expression-statement*
> *selection-statement*
> *iteration-statement*
> *jump-statement*

Semantics

2 A *statement* specifies an action to be performed. Except as indicated, statements are executed in sequence.

3 A *block* allows a set of declarations and statements to be grouped into one syntactic unit. The initializers of objects that have automatic storage duration, and the variable length array declarators of ordinary identifiers with block scope, are evaluated and the values are stored in the objects (including storing an indeterminate value in objects without an initializer) each time the declaration is reached in the order of execution, as if it were a statement, and within each declaration in the order that declarators appear.

4 A *full expression* is an expression that is not part of another expression, nor part of a declarator or abstract declarator. There is also an implicit full expression in which the non-constant size expressions for a variably modified type are evaluated; within that full expression, the evaluation of different size expressions are unsequenced with respect to one another. There is a sequence point between the evaluation of a full expression and the evaluation of the next full expression to be evaluated.

5 **NOTE** Each of the following is a full expression:

— a full declarator for a variably modified type,
— an initializer that is not part of a compound literal,
— the expression in an expression statement,
— the controlling expression of a selection statement (**`if`** or **`switch`**),
— the controlling expression of a **`while`** or **`do`** statement,
— each of the (optional) expressions of a **`for`** statement,
— the (optional) expression in a **`return`** statement.

While a constant expression satisfies the definition of a full expression, evaluating it does not depend on nor produce any side effects, so the sequencing implications of being a full expression are not relevant to a constant expression.

Forward references: expression and null statements (6.8.3), selection statements (6.8.4), iteration statements (6.8.5), the **`return`** statement (6.8.6.4).

6.8.1 Labeled statements

Syntax

1 *labeled-statement:*

> *identifier* **`:`** *statement*
> **`case`** *constant-expression* **`:`** *statement*
> **`default`** **`:`** *statement*

Constraints

2 A **`case`** or **`default`** label shall appear only in a **`switch`** statement ~~.~~ that is associated with the same function body as the statement to which the label is attached.[165] Further constraints on such labels are discussed under the **`switch`** statement.

[165] Thus, a label that appears within a lambda expression may only be associated to a switch statement within the body of the lambda.

6.8.5.3 The `for` statement

1 The statement

```
    for (clause-1; expression-2; expression-3) statement
```

behaves as follows: The expression *expression-2* is the controlling expression that is evaluated before each execution of the loop body. The expression *expression-3* is evaluated as a void expression after each execution of the loop body. If *clause-1* is a declaration, the scope of any identifiers it declares is the remainder of the declaration and the entire loop, including the other two expressions; it is reached in the order of execution before the first evaluation of the controlling expression. If *clause-1* is an expression, it is evaluated as a void expression before the first evaluation of the controlling expression.[171)]

2 Both *clause-1* and *expression-3* can be omitted. An omitted *expression-2* is replaced by a nonzero constant.

6.8.6 Jump statements

Syntax

1 *jump-statement:*

```
        goto identifier ;
        continue ;
        break ;
        return expression_opt ;
```

Constraints

2 No jump statement other than **`return`** shall have a target that is found in another function body.[172)]

Semantics

3 A jump statement causes an unconditional jump to another place.

6.8.6.1 The `goto` statement

Constraints

1 The identifier in a **`goto`** statement shall name a label located somewhere in the enclosing function body. A **`goto`** statement shall not jump from outside the scope of an identifier having a variably modified type to inside the scope of that identifier.[173)]

Semantics

2 A **`goto`** statement causes an unconditional jump to the statement prefixed by the named label in the enclosing function.

3 **EXAMPLE 1** It is sometimes convenient to jump into the middle of a complicated set of statements. The following outline presents one possible approach to a problem based on these three assumptions:

1. The general initialization code accesses objects only visible to the current function.
2. The general initialization code is too large to warrant duplication.
3. The code to determine the next operation is at the head of the loop. (To allow it to be reached by **`continue`** statements, for example.)

```
    /* ... */
    goto first_time;
    for (;;) {
```

[171)] Thus, *clause-1* specifies initialization for the loop, possibly declaring one or more variables for use in the loop; the controlling expression, *expression-2*, specifies an evaluation made before each iteration, such that execution of the loop continues until the expression compares equal to 0; and *expression-3* specifies an operation (such as incrementing) that is performed after each iteration.

[172)] Thus jump statements other than **`return`** may not jump between different functions or cross the boundaries of a lambda expression, that is, they may not jump into or out of the function body of a lambda. Other features such as signals (7.14) and long jumps (7.13) may delegate control to points of the program that do not fall under these constraints.

[173)] The visibility of labels is restricted such that a **`goto`** statement that jumps into or out of a different function body, even if it is nested within a lambda, is a constraint violation.

```
                // determine next operation
                /* ... */
                if (need to reinitialize) {
                        // reinitialize-only code
                        /* ... */
                first_time:
                        // general initialization code
                        /* ... */
                        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 ~~.~~ that is associated to the same function body.[174]

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`;.[175]

6.8.6.3 The **break** statement

Constraints

1 A **break** statement shall appear only in or as a switch body or loop body ~~.~~ that is associated to the same function body.[176]

[174] Thus a **continue** statement by itself may not be used to terminate the execution of the body of a lambda expresssion.
[175] Following the `contin`: label is a null statement.
[176] Thus a **break** statement by itself may not be used to terminate the execution of the body of a lambda expresssion.

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 body whose return type is **void**. A **return** statement without an expression shall only appear in a function body whose return type is **void**.

2 For a function body that has an underspecified return type, 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 function-to-pointer conversion) with the same type; otherwise all **return** expressions shall have no expression.

Semantics

3 A **return** statement is associated to the innermost function body in which appears. It evaluates the expression, if any, terminates the execution of ~~the~~ that function body and returns control to ~~the caller~~its caller; if it has an expression, the value of the expression is returned to the caller as the value of the function call expression. 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.[177)]

5 For a lambda or a function that has an underspecified return type, 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.

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();
```

there is no undefined behavior, although there would be if the assignment were done directly (without using a function call to fetch the value).

177)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.