Type-generic lambdas v.1
proposal for C23

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

For the lambda expressions that were introduced in N2633, we propose the addition of **auto** parameters that can be completed by the arguments (in a function call) or by the parameter types of target function pointer (in a conversion).

I. MOTIVATION

This paper is fully motivated in N2638, namely for the improvement of type-generic programming in C. For a simple motivation of the feature compared to simple lambdas see for example the `MAXIMUM` macro in the proposed text, `6.5.2.6 p17`.

II. DESIGN CHOICES

We chose to follow C++ syntax and semantic as close a possible.

II.1. Permissible contexts for type-generic lambdas

It is the intent of this paper, to allow a value of a type-generic lambda type only in a context where it will be completed, either by the arguments of a function call or by the parameter types of a target function pointer to which a type-generic function literal is converted. This is to ensure that compilers that implement this feature have to do no lookahead or pre-compilation of code snippets with a lot of unknown types.

This is achieved by integrating types of type-generic lambdas into the terminology of the standard as being incomplete types. Thereby it is not possible to define objects of such a type. Because lambdas can only be declared in definitions by type inference, effectively such lambdas cannot even be declared.

By these properties, the only possibility to specify a type-generic lambda that is re-usable at different places of a source is *textual*, in particular by defining function-like macros. This restriction is a deliberate choice for this proposal, here. If in a later phase (probably C26) WG14 would also want to add objects of type-generic lambda type to the language or adopt C++'s `template` functions, this could easily be achieved on top of what is done here.

II.2. Parameter type inference

Parameter type inference only leaves a design choice for array and function parameters. To be in line with traditional function declarations, we extend the possibility of type inference to such types and specify that these are to be re-written to pointers to form a valid function prototype.

III. SYNTAX AND TERMINOLOGY

For all proposed wording see Section VII.

Syntax considerations for this feature are straight forward; we just have to allow the **auto** feature to extend to the parameters of lambdas, `6.7.6.3`.

In terms of terminology, we introduce the terms *incomplete lambda type* (`6.2.5 p20`) and *type-generic lambda* (`6.5.2.6 p9`).

IV. SEMANTICS

The principal semantics of type-generic lambdas are described within three paragraphs.

— Paragraph `6.2.5.6 p9` specifies the possible use of type-generic lambdas.
— Paragraph `6.2.5.6 p10` provides the rules for the completion of such a lambda in a function call.
— An insertion into `6.3.2.1 p5` describes the mechanism for conversions of type-generic function literals to function pointers.

V. CONSTRAINTS AND REQUIREMENTS

This proposal constrains the possible uses of type-generic lambdas even further than for simple lambdas, namely essentially to function calls and conversions to pointer-types. Even though it would have been possible to formulate such a requirement as a constraint, we chose not to do so because this might be an area for implementations to extend the `C` standard and to implement some template feature for lambda values. Forcing them to diagnose such constructs would be counter-productive and hinder progress in that area.

The only constraint that this proposal includes is in `6.5.2.6 p6`, namely that a type-generic lambda that is used in a conversion to a function pointer must have a return type that is compatible to the one of the target function pointer type.

VI. QUESTIONS FOR WG14

(1) Does WG14 want type-generic lambdas for C23 along the lines of N2634?
(2) Does WG14 want to integrate the changes as specified in N2634 into C23?

References

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 N2633.

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

— 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~~ an object type that describes the value of a lambda expression. A complete 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~~ 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 lambda expression that has underspecified parameters has an incomplete lambda type that can be completed by function call arguments.

— 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.

[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.

version of the type of the lvalue; otherwise, the value has the type of the lvalue. If the lvalue has an incomplete type and does not have array type, the behavior is undefined. If the lvalue designates an object of automatic storage duration that could have been declared with the **register** storage class (never had its address taken), and that object is uninitialized (not declared with an initializer and no assignment to it has been performed prior to use), the behavior is undefined.

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~~ Other than specified in the following, lambda types 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*".For a type-generic lambda, types of underspecified parameters shall first be completed according to the parameters of the target prototype; that is, for each underspecified parameter there shall be a type specifier as described in 6.7.10 such that the adjusted parameter type is compatible with the parameter type of the target function type. After that, the inferred return type of the thus completed lambda shall be compatible with the return type of the target prototype.[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~~ differences are that, if λ is not type-generic, the resulting function pointer is the same for the whole program execution whenever a conversion of λ is met[72] and 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)~~.~~, type inference (6.7.10).

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*.[73] 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.

[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]Thus a function literal that is not type-generic has properties that are similar to a function declared with **static** and **inline**. A possible implementation of the lambda type is to be the the function pointer type to which they convert.

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

Constraints

2 A lambda expression shall not be operand of the unary & operator.[111)]

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.[112)]

6 ~~The~~ After determining the type of all captures and parameters the function body shall be such that a return type *type* according to the rules in 6.8.6.4 can be inferred. If the lambda occurs in a conversion to a function pointer, the inferred return type shall be compatible to the specified return type of the function pointer.

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~~ Value captures and parameters have automatic storage duration; in each function call to the formed lambda value, a new instance of each value 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 A lambda expression for which at least one parameter declaration in the parameter list has no type specifier is a *type-generic lambda* with an imcomplete lambda type. It shall only occur in a void expression, as the postfix expression of a function call or, if the capture clause is empty, in a conversion to a pointer to function with fully specified parameter types, see 6.3.2.1. For a void expression, it has no side effects and shall be ignored.

10 For a function call, the type of an argument (after lvalue, array-to-pointer or function-to-pointer conversion) to an underspecified parameter shall be such that it can be used to complete the type of that parameter analogous to 6.7.10, only that the inferred type for an parameter of array or function type is adjusted analogously to function declarators (6.7.6.3) to a possibly qualified object pointer type (for an array) or to a function pointer type (for a function) to match type of the argument. For a conversion of any arguments, the parameter types shall be those of the function type.

11 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

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

[112)]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.

exists according to the constraints.[113)]

12 The implicit or explicit assignment expression `E` in the definition of a value capture determines a value E_0 with type T_0, which is `E` after possible lvalue, array-to-pointer or function-to-pointer conversion. The type of the capture is T_0 **`const`** and its value is 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,[114)] and that address is used to modify the underlying object, the behavior is undefined. 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.

13 For each lambda expression, the return type *type* is inferred as indicated in the constraints. A lambda expression λ that is not type-generic 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

14 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.

15 **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

}
```

16 **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, };
```

[113)]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.

[114)]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.

```
  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.

17 **EXAMPLE 3** Consider the following type-generic function literal that computes the maximum value of two parameters X and Y.

```
#define MAXIMUM(X, Y)                                          \
    [](auto a, auto b){                                        \
      return (a < 0)                                           \
             ? ((b <  0) ? ((a < b) ? b : a) : b)              \
             : ((b >= 0) ? ((a < b) ? b : a) : a);             \
    }(X , Y)

    auto R = MAXIMUM(-1, -1U);
    auto S = MAXIMUM(-1U, -1L);
```

After preprocessing, the definition of R becomes

```
auto R = [](auto a, auto b){
  return (a < 0)
     ? ((b <  0) ? ((a < b) ? b : a) : b)
     : ((b >= 0) ? ((a < b) ? b : a) : a);
  }(-1, -1U);
```

To determine type and value of R, first the type of the parameters in the function call are inferred to be **signed int** and **unsigned int**, respectively. With this information, the type of the **return** expression becomes the common arithmetic type of the two, which is **unsigned int**. Thus the return type of the lambda is that type. The resulting lambda value is the first operand to the function call operator (). So R has the type **unsigned int** and a value of **UINT_MAX**.

For S, a similar deduction shows that the value still is **UINT_MAX** but the type could be **unsigned int** (if **int** and **long** have the same width) or **long** (if **long** is wider than **int**).

As long as they are integers, regardless of the specific type of the arguments, the type of the expression is always such that the mathematical maximum of the values fits. So MAXIMUM implements a type-generic maximum macro that is suitable for any combination of integer types.

18 **EXAMPLE 4**

```
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) {
    }
```

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

6.7.6.3 Function declarators (including prototypes)

Constraints

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

2 The only storage-class ~~specifier~~ specifiers that shall occur in a parameter declaration ~~is~~ are **`auto`** and **`register`**.

3 An identifier list in a function declarator that is not part of a definition of that function shall be empty. A parameter declaration without type specifier shall not be formed, unless it includes the storage class specifier **`auto`** and unless it appears in the parameter list of a lambda expression.

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

Semantics

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

`D (` *parameter-type-list* `)`

or

`D (` *identifier-list*$_{opt}$ `)`

and the type specified for *ident* in the declaration "`T D`" is "*derived-declarator-type-list T*", then the type specified for *ident* is "*derived-declarator-type-list* function returning the unqualified version of *T*".

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

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

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

9 If the list terminates with an ellipsis (`, ...`), no information about the number or types of the parameters after the comma is supplied.[156)]

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

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

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

13 The storage-class specifier in the declaration specifiers for a parameter declaration, if present, is ignored unless the declared parameter is one of the members of the parameter type list for a function definition.

14 An identifier list declares only the identifiers of the parameters of the function. An empty list in a function declarator that is part of a definition of that function specifies that the function has no parameters. The empty list in a function declarator that is not part of a definition of that function

156) The macros defined in the `<stdarg.h>` header (7.16) can be used to access arguments that correspond to the ellipsis.

```
    struct S {
          int i;
          struct T t;
    };

    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.[165] if the assignment expression has lambda type, the lambda type shall be complete and 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 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

[165]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.