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

N2924 v5
P2306R4

2022-1-30

Type-generic lambdas
proposal for C23

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

For the lambda expressions that were introduced in N2892, 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).

Changes:

v5/R4. move handling of auto parameters from N2923
v4/R3. rebase relative to the changes in the other papers in this series
v3/R2. provide a proper motivation for type-generic macros
v2/R1.
— make primary expressions transparent for lambda expression operands
— force types to be the same when a tg lambda is converted
— specify which syntax verifications are necessary for lamda expressions in void expressions

I. MOTIVATION

Compared to the basic lambda feature introduced in N2892 the introduction of **auto** parameters the interest to make them type-generic is threefold. The first two are operational, namely to provide function calls and function pointer conversions in a user-friendly way. The third has to do with the notorious problem of exposing commas that stem from declaration syntax in function-like macro calls.

I.1. Comfortable type-generic function calls

The first can be seen by replacing a "simple" macro-ized lambda that uses value captures instead of parameters for type-genericity:

```
1 #define MAXIMUM0(X, Y)                                  \
2 [MAXIMUM_A = X, MAXIMUM_B = Y](void){                   \
3       auto const a = MAXIMUM_A;                         \
4       auto const b = MAXIMUM_B;                         \
5       return (a < 0)                                    \
6              ? (b < 0) ? (a < b) ? b : a : b            \
7              : (0 < b) ? (b < a) ? a : b : a;           \
8     }()
```

Here, the fact of using captures `MAXIMUM_A` and `MAXIMUM_B` with inferred types ensures that the lambda depends on the type of the macro arguments, and that the return type is inferred as the common super-type of the two argument types.

So this technique already provides the possibility to have a fully generic definition that would work with any pair of argument types that allow a comparison against each other and against **int** (for the comparison to `0`). Unfortunately it still has two of the drawbacks of macro programming. First, the captures must have names that are guaranteed not to

collide with any of the identifiers that might be used in expressions used as arguments. For example for the following choices of `a` and `b`

```
  double a = 34.0;
  int b = 77;
  auto c = MAXIMUM0(a+b, a-b);
```

the expansion of the capture clause as given would be

```
[MAXIMUM_A = (a+b), MAXIMUM_B = (a-b)]
```

If we had used `a` and `b` directly as capture names, it would have read instead

```
[a = (a+b), b = (a-b)]
```

where in the evaluation of `(a-b)` `a` would already refer to the capture `a`. Capture `a` would be `111.0` as expected, but capture `b` would surprisingly not be `-43.0` but `34.0`.

Additionally, we need two declarations for the identifiers `a` and `b` that we really want to use in the body of the lambda, and that might, if we need that, not be **`const`** qualified.

Compared to that, a type-generic lambda as proposed by this paper is simpler to program and easier to understand

```
#define MAXIMUM2(X, Y)                                  \
    [](auto a, auto b){                                 \
      return (a < 0)                                    \
             ? (b < 0) ? (a < b) ? b : a : b            \
             : (0 < b) ? (b < a) ? a : b : a;           \
    }(X , Y)
```

It avoids the invention of arbitrary capture names and their redefinition to local variables. But other than that it has nothing fancy and can, within such a direct call, just be rewritten to code as above. The only difference is that we delegate this rewriting to the compiler.

I.2. Multiple instantiations as function pointers

The second main feature that we propose is the conversion of type-generic function literals to function pointers. To see that, consider the following lambda that uses the **`typeof`** operator and then produces two function pointers with inferred types.

```
#define MAXY(X, Y)                                  \
    [](typeof(X) a, typeof(Y) b){                   \
      return (a < 0)                                \
             ? (b < 0) ? (a < b) ? b : a : b        \
             : (0 < b) ? (b < a) ? a : b : a;       \
    }

double   (*maxdd)(double, double)   = MAXY(double, double);
unsigned (*maxsu)(signed, unsigned) = MAXY(signed, unsigned);
```

Using that macro directly in a call would look quite unconventional and would also be errorprone because of the repetition or replacement of the arguments.

```
  auto c = MAXY(a+b, a-b)(a+b, a-b);
  auto d = MAXY(unsigned, unsigned)(a+b, a-b);
```

In contrast to that, again, a type-generic lambda is simple to read and write:

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

double   (*maxdd)(double, double)   = MAXIMUM;
unsigned (*maxsu)(signed, unsigned) = MAXIMUM;
```

The two expansions of MAXIMUM read

```
double   (*maxdd)(double, double)   = [](auto a, auto b){ ... };
unsigned (*maxsu)(signed, unsigned) = [](auto a, auto b){ ... };
```

They are internally rewritten by the compiler as if they had been specified as

```
double   (*maxdd)(double, double)   = [](double a, double b){
   ... };
unsigned (*maxsu)(signed, unsigned) = [](signed a, unsigned b){
   ... };
```

Still, a call to the type-generic lambda would be similar to a call to a conventional function.

```
  auto c = MAXIMUM(a+b, a-b);
```

I.3. Composing declaration syntax and function-like macro calls

The technique that lead to macro MAXIMUM0 above is not only tedious to write, it also lacks an important property that one would expect from a basic feature such as the computation of a maximum value, namely *composability*. This is a problem that already occurs with current function-like macros when their arguments are compound literals. Namely that a comma in a compound literal can be exposed to the preprocessor as if it were a separator for the arguments of the macro. This problem becomes much more severe when introducing lambdas because of the unprotected comma separated list of the captures. In general, C's declaration syntax does not well behave in expressions that are processed by function-like macros.

For the sake of the argument we extend the example as follows such that our feature operates on pointers to values instead of the values themselves. First, we design a comparison macro that operates on pointers to arithmetic objects and adds the convention that a null pointer compares "less" than any pointer to a value.

```
#define ISPOINTLESS(X, Y)                        \
[ISPOINTLESS_A = X, ISPOINTLESS_B = Y](void){    \
```

```
  auto const*const a = ISPOINTLESS_A;          \
  auto const*const b = ISPOINTLESS_B;          \
  if (!b) return false;                        \
  else if (!a && b) return true;               \
  else return (*a < *b);                       \
}()
```

Then we encapsulate our maximum computation such that it uses that new comparison and for which the result is the pointer to the maximum. (This only works if both pointers have the same base type).

```
#define MAXAPOINTING(X, Y)                                    \
[MAXAPOINTING_A = X, MAXAPOINTING_B = Y](void){               \
  auto const*const a = MAXAPOINTING_A;                        \
  auto const*const b = MAXAPOINTING_B;                        \
  static int const zero = 0;                                  \
  return ISLESS(a, &zero)                                     \
    ? ISPOINTLESS(b, &zero) ? ISPOINTLESS(a, b) ? b : a : b   \
    : ISPOINTLESS(&zero, b) ? ISPOINTLESS(b, a) ? a : b : a;  \
}()
```

In the following, the first line should compile fine, but the second is not valid and contains a constraint violation.

```
  auto* c = MAXAPOINTING(&x, &y);
  auto* d = MAXAPOINTING(MAXAPOINTING(&x, &y), &z);
```

This is because after the expansion of the inner macro call the commas in the capture lists of the expanded `ISPOINTLESS` macros are exposed to the scan for arguments of the outer call.

This problem can be avoided by a cascade of parenthesis around macro arguments **and** lambda expressions. But in general the syntax of such constructs is quite fragile, and type-generic macros provide a better alternative that is more user-friendly because it does not need function-like macros or captures.

```
#define ispointless                              \
[ ](auto const* a, auto const* b) {              \
   if (!b) return false;                         \
   else if (!a && b) return true;                \
   else return (*a < *b);                        \
}
#define maxapointing                                          \
[ ](auto const* a, auto const* b) {                           \
  static int const zero = 0;                                  \
  return ispointless(a, &zero)                                \
    ? ispointless(b, &zero) ? ispointless(a, b) ? b : a : b   \
    : ispointless(&zero, b) ? ispointless(b, a) ? a : b : a;  \
}
```

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 N2924?
(2) Does WG14 want to integrate the changes as specified in N2924 into C23?

References

Jens Gustedt. 2021a. *Function literals and value closures*. Technical Report N2892. ISO. available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2892.pdf.

Jens Gustedt. 2021b. *Improve type generic programming*. Technical Report N2890. ISO. available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2890.pdf.

Jens Gustedt. 2021c. *Lvalue closures*. Technical Report N2737. ISO. available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2737.pdf.

Jens Gustedt. 2021d. *Type-generic lambdas*. Technical Report N2924. ISO. available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2924.pdf.

Jens Gustedt. 2021e. *Type inference for variable definitions and function return*. Technical Report N2923. ISO. available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2923.pdf.

VII. PROPOSED WORDING

The proposed text is given as diff against N2892.

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

20 Any number of *derived types* can be constructed from the object and function types, as follows:

— An *array type* describes a contiguously allocated nonempty set of objects with a particular member object type, called the *element type*. The element type shall be complete whenever the array type is specified. Array types are characterized by their element type and by the number of elements in the array. An array type is said to be derived from its element type, and if its element type is *T*, the array type is sometimes called "array of *T*". The construction of an array type from an element type is called "array type derivation".

— 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, and by the lexical position of the lambda expressions in the program~~. ;~~ ~~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 type has no syntax derivation[50] and the lexical position of the originating lambda expression determines its scope of visibility. Objects of such a type shall only be defined as a capture (of another lambda expression) or by an underspecified declaration for which the lambda type is inferred.[51] An object of lambda type shall only be modified by simple assignment (6.5.16.1). A lambda expression that has underspecified parameters has an incomplete lambda type that can be completed by function call arguments, or, if it has no captures, in a conversion to a function pointer.

— 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*.[52]

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.

[50] Not even a **typeof** type specifier with lambda type can be formed. So there is no syntax to make a lambda type a choice in a generic selection other than **default**

[51] Another possibility to create an object that has an effective lambda type is to copy a lambda value into allocated storage via simple assignment.

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

the object. A *modifiable lvalue* is an lvalue that does not have array type, does not have an incomplete type, does not have a const-qualified type, and if it is a structure or union, does not have any member (including, recursively, any member or element of all contained aggregates or unions) with a const-qualified type.

2 Except when it is the operand of the **typeof** specifier, the **sizeof** operator, the unary & operator, the `++` operator, the `--` operator, or the left operand of the `.` operator or an assignment operator, an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue); this is called *lvalue conversion*. If the lvalue has qualified type, the value has the unqualified version of the type of the lvalue; additionally, if the lvalue has atomic type, the value has the non-atomic 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 **typeof** specifier, the unary **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 **typeof** specifier, the **sizeof** operator,[72] 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 ~~A~~ Other than specified in the following, lambda types shall not be converted to any other object type. A complete function literal with a type "lambda with prototype `P`" can be converted implicitly or explicitly to an expression that has type "pointer to `Q`", where `Q` is a function type that is compatible with `P`.[73] For a type-generic function literal expression, 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 of a unique type as described in 6.7.11 such that the adjusted parameter type is the same as the adjusted parameter type of the target function type; after that, the prototype `P` of the thus completed lambda expression shall be the target prototype `Q`.[74] The function pointer value behaves as if a function `F` of type `P` with internal linkage, a unique name, and the same ~~parameter list and~~ function body as for λ, where uses of identifiers from enclosing blocks in expressions that are not evaluated are replaced by proper types or values, had been defined 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[75] 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), **typeof** specifier 6.7.9, initialization (6.7.10), 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 (6.7.11).

[72] 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.

[73] 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.

[74] Thus a specification of the target function pointer type in a conversion from a type-generic function literal expression that uses the `[*]` syntax for VM types is invalid.

[75] 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.

capture:
identifier

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

Constraints

2 An identifier shall appear at most once; either as a capture or as a parameter name in the parameter list. The identifier of an identifier capture shall designate an object of automatic storage duration that is defined in a scope that surrounds the lambda expression.

3 Within the lambda expression, identifiers (including captures and parameters of the lambda) shall be used according to the usual scoping rules, but outside the assignment expression of a value capture the following exceptions apply to identifiers that are declared in a block that strictly encloses the lambda expression and that are not identifier captures:

— Objects or type definitions with variably modified type shall not be used.

— Objects with automatic storage duration shall not be evaluated.[114)]

4 ~~The~~ After determining the type of all captures and parameters, either directly or because a type-generic lambda appears in a function-call or conversion to function pointer, 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; if additionally the lambda is type-generic, the return type shall be the same as the specified return type.

5 When a lambda expression with an underspecified parameter is evaluated as a void expression, the capture clause shall fulfill the constraints as specified above. The parenthesized parameter list shall provide a valid list of declarations of parameters, only that one or more of these may have an underspecified type. After that shall follow a `{` token, a balanced token sequence (**??**), and a `}` token.[115)]

Semantics

6 The optional attribute specifier sequence in a lambda expression appertains to the resulting lambda type and to its function prototype. If the parameter clause is omitted, a clause of the form `()` is assumed. A lambda expression without any capture 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*.

7 Similar to a function definition, a lambda expression forms a single block that comprises all of its parts. Each 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. In particular, captures and parameters are visible throughout the whole function body, unless they are redeclared in a depending block within that function body. 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 addresses of value captures are not necessarily unique.

8 A lambda expression for which at least one parameter declaration in the parameter list has no type

[114)] Identifiers of visible automatic objects that are not captures and that do not have a VM type, may still be used if they are not evaluated, for example in **`sizeof`** expressions, in **`typeof`** specifiers (if they are not lambdas themselves) or as controlling expression of a generic primary expression.

[115)] That means, besides the validity of the capture clause and the parameter list, an implementation is only required to parse the function body as a token sequence but is not required to diagnose additional constraints, such as the validity of the use of keywords or identifiers within the function body if these are possibly restricted through a syntax derivation or additional constraints.

specifier is a *type-generic lambda* with an imcomplete lambda type. It shall only be evaluated as a void expression, be the postfix expression of a function call or, if the capture clause is empty, be the operand of a conversion to a pointer to function with fully specified parameter types, see 6.3.2.1. For a void expression, it has only the side effects that result from the evaluation of the capture clause and shall be syntactically correct as indicated in the constraints; the translation may fail, if the function body is such that no possible function call arguments or target types for a conversion could successfully complete the lambda type; the lambda expression shall otherwise be ignored.

9 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 6.7.11, 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.

10 The 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 or one of its members is taken, either explicitly by applying a unary & operator or by an array to pointer conversion,[116)] and that address is used to modify the underlying object, the behavior is undefined.

11 The evaluation of the assignment expressions of value captures takes place during each evaluation of the lambda expression. The evaluations for the value captures are sequenced in order of declaration; an earlier capture may occur within an assignment expression of a later one. The evaluation of a lambda expression is sequenced before any use of the resulting lambda value. For each call to a lambda value, value captures (with type and value as determined during the evaluation of the lambda expression) and then parameter types and values are determined in order of declaration. Value captures and earlier parameters may occur within the declaration of a later one.

12 The object of automatic storage duration of the surrounding scope that corresponds to an identifier capture shall be visible within the function body according to the usual scoping rules and shall be accessible within the function body throughout each call to the lambda. If the definition of the object uses the **register** storage class, the behavior is undefined. Access to the object within a call to the lambda follows the happens-before relation, in particular modifications to the object that happen before the call are visible within the call, and modifications to the object within the call are visible for all evaluations that happen after the call.[117)]

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. ~~As~~ ; as a result of the expression, 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 other then by simple assignment.

14 A lambda expression λ that is generic has an incomplete lambda type that is completed when the expression is used directly in a function call expression or converted to a function pointer. When used in a function call, the parameter types are inferred in order of declaration, but after the evaluation of the assignment expressions of the explicit value captures, after which the return type of the lambda is inferred from the function body. The so completed lambda value is then used in the function call which is sequenced after the evaluation of the lambda expression.

15 **NOTE 1** A direct function call to a function literal expression can be modeled by first performing a conversion of the function literal to a function pointer and then calling that function pointer.

16 **NOTE 2** A direct function call to a closure expression with parameters (possibly type-generic)

[116)] 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.

[117)] That is, evaluation of the identifier results in the same lvalue with the same type and address as for the scope surrounding the lambda. In particular, it is possible that the value of such an object becomes indeterminate after a call to **longjmp**, see 7.13.2.1.

```
...
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.

20 **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.

21 **EXAMPLE 4**

```
void matmult(size_t k, size_t n, size_t m,
             double const A[k][n], double const B[n][m], double const C[k][m]) {
  // dot product with stride of m for B
  // ensure constant propagation of n and m
  auto const λ0 = [ν=n, μ=m](double const x[ν], double const B[ν][μ], size_t m0) {
    double ret = 0.0;
    for (size_t i = 0; i < ν; ++i) {
      ret += x[i]*B[i][m0];
    }
```

```
    }
```

Forward references: function declarators (6.7.6.3), function definitions (6.9.1), initialization (6.7.10).

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*$_{\text{opt}}$ `)`

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

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.[159)]

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

[159)] 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).

6.7.11 Type inference

Constraints

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

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

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

such that the declarator does not declare an array.[169] If the assignment expression has lambda type that type shall be complete, the declaration shall only declare one object and shall only consist of storage class specifiers, qualifiers, the identifier that is to be declared, and the initializer.

3 Unless it is the definition of an object with an assignment expression of lambda type as above, prior to an underspecified declaration there shall exist a typeof specifier *type* (that is, a type specifier that is a typeof operator applied to an expression or type name) that if used to replace the **auto** specifier makes the adjusted declaration a valid declaration;[170] *type* shall not declare a tag or the contents of a structure, union or enumeration (including at function prototype scope). If it is also the definition of a function the return type shall be determined from **return** statements (or the lack thereof) as specified in 6.9.1. Otherwise, *type* shall be such that for all defined objects the assignment expression in the corresponding init-declarator, after possible lvalue, array-to-pointer or function-to-pointer conversion, has the non-atomic, unqualified type of the declared object.

Description

4 Although there is no syntax derivation to form declarators of lambda type, a value λ of lambda type `L` can be used as assignment expression to initialize an underspecified object declaration and as

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

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