Lvalue closures

Jens Gustedt

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

N2737 v3
P2307R2

2021-5-15

Lvalue closures
proposal for C23

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

For the lambda expressions that were introduced in N2736, we propose the addition of lvalue captures that can read and modify objects of enclosing blocks.

Changes:

v3/R2. this document, add a note for a minor syntax ambiguity and update the diffmark text
v2/R1. integrating feedback from different sources
— Make the property of being an lvalue closure recursive.
— Enforce the `register` storage class if necessary.
— Clarify that all access to thread local or automatic storage of another thread is implementation-defined.
— Note that the same rules apply for `longjmp` and lvalue captures as would for other local variables.

I. MOTIVATION

In N2734 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 extension to lvalue captures for lambdas as a stand-alone addition to C without full type-genericity.

There are certainly some situations were an lvalue capture can be useful, for example when a functional unit collects statistical data in a sideline of its principal task. But our main motivation to introduce the feature is portability. Current extensions to C, by gcc in particular, already use local functions or compound expressions that allow access to automatic variables of an enclosing block. For code bases that use these features and that want to move to lambdas (to be better portable, for example) should be offered migration path to do that easily. With this proposal a nested function that is not recursive and not converted to a function pointer in the sequel

```
1  ...
2  double alpha(double x) {
3    ...
4  }
5  ...
```

can be relatively easily transformed to an initialization of a variable of lvalue closure type

```
1  ...
2  auto const α = [&](double x) {
3    ...
4  };
5  ...
```

Similarly, a gcc compound expression

({ *block-item-list expression-statement* })

can easily be formulated as an adhoc call to an lvalue closure

[&](**void**){ *block-item-list* **return** *expression-statement* }()

namely, a closure expression with capture default `&`, no parameters and that is immediately called in a function call with no arguments.

II. DESIGN CHOICES

We chose to follow C++ syntax as close a possible. There is one feature that we did not include in this proposal, though, namely the possible syntax

`&` *identifier* `=` *unary-expression*

which in C++ terms would be an initialization of a reference variable with an lvalue, that is, that would establish *identifier* as a named alias to an object representation that is referred to by the unary expression.[1]

We are not aware of proposals that would add that reference type category to C, and we think that the introduction of lvalue closures should not introduce it silently. If WG14 decides at some point to include such a type category in the future, the syntax for lambdas can easily be amended.

We made one design choice about variables with **register** storage class that does not seem unanymous. The address of such variables would still be inaccessible in lvalue closures, and taking the address of such an lvalue closure would in some sense undermine the contract that the user seeks for the variable.

Listing 1. Example for a breach of a **register** contract by using a wide function pointer feature.

```
1  void leakage(math_wfp fun);    // different TU
2  void do_something(double y);   // different TU
3
4  int main(void) {
5    register double x = 0;
6    // implements a simple data exchange with x
7    auto λ0 = [&x](double val){
8      auto prev = x;
9      x = val;
10     return prev;
11   };
12   // does this store the _Wide pointer fun in global memory?
13   leakage(&λ0);
14
15   x = 1;
16   // does this use the leaked _Wide pointer?
17   do_something(5.0);
18   // what value for x can we expect?
19   printf("now␣x␣is␣%g\n"",␣x);
```

[1] Unary expression is the lowest binding syntax construct in C that can result in an lvalue.

```
20 }
```

This problem is not as significant for the simple lambdas we are proposing up to now, but more relevant for implementations or future extensions that provide some form of extended (wide) function pointers by which access to lambdas is possible from other translation units.

To prevent such a break of contract we propose a very simple remedy: all lambda objects that directly or indirectly access a **register** variable have to be themselves declared with **register** storage class, see the last sentence of 6.5.2.6 p5. Thereby the address of such a lambda can never be taken; it cannot be copied by address and no address to it can be passed on to a different translation unit.

III. SYNTAX AND TERMINOLOGY

For all proposed wording see Section VII.

Because C does not have the concept of references (in the sense of identifiers that act as aliases for other objects), the terminology of a "reference capture" would not be adequate. Therefore we use the term *lvalue capture* to indicate that these are captures that are seen not as evaluated expressions but as lvalues. Other terms that are added in 6.5.2.6 p8, are *value closure* and *lvalue closure*, because these have quite distinct properties and in particular different possible extend of validity.

For the syntax, within the range of the proposed semantics, we follow C++ as close as possible, see 6.5.2.6 p1.

— We add a new capture default, a & token, to indicate that by default captures are lvalue captures.
— We add a new capture category, *lvalue capture*, with a syntax of

& *identifier*

— In the case that the first element in the capture list is a capture default, only one category of captures (value or lvalue) is permitted for the rest of the capture list. Although it would have been possible to write this property up in pure syntax, this would have been relatively complicated and so we chose to formulate this as a syntax constraint, only, see 6.5.2.6 p3 last sentence. But if WG14 prefers to have this a pure syntax concept, this could be moved in that direction.

Note that the syntax &*identifier* adds another lexical ambiguity to those that are listed in N2736. A token sequence

```
[ & identifier ]
```

that appears in an initializer now can be read as a designator or as the capture clause of a lambda. This ambiguity should be easy to detect and act upon by compilers because &*identifier* is never a valid integer constant expression as it would be required for a designator.

Another important feature is to describe which outer variables are used as lvalue captures. This may not only be variables that a lambda uses directly (either as an explicit lvalue capture or by the default rule) but also lvalue captures that are used by another lambda that is passed itself as a captures, 6.5.2.6 p12.

```
1 double context = 0.0;
```

```
2
3   auto comp = [&context](double const* a, double const* b){
4     int ret = 0;
5     // ... use context to determine an order
6     ret = ...
7     return ret;
8   }
9
10  auto search = [comp](size_t n, double A[n], double val) {
11    size_t pos = 0;
12    // ... use comp for a binary search
13    pos = ...
14    return pos;
15  }
```

Here, **search** itself only has a value capture comp, but the lambda comp that is captures has an lvalue capture context and is an lvalue closure. Therefore context is considered as an *indirect* lvalue capture of **search** which therefore is an lvalue closure.

Because lvalue closures can also be executed in a multi-threaded context or in contexts with signals or long jumps, 6.5.2.6 p12 also precises which side effects are visible within a lambda and after a possible modification of an lvalue capture. To avoid misunderstandings, we also sharpen the implementation-defined properties if thread local (6.4.2 p4) or automatic (p5) variables are accessible by other threads or not.

IV. SEMANTICS

The principal semantic of lvalue captures are described in a newly inserted paragraph 6.2.5.6 p12. It has two parts. First, if gives a simple model for the access to the underlying object, namely that the access to a lvalue capture is just seen as an access to the variable itself. Then, it is clarified that the access to such objects follows the usual rules for the visibility of side effects.

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.

The most important requirement for lvalue captures is that they should not be used in a function call where any of their lvalue captures is dead. This could just be done by making such a call undefined (implicitly or explicitly), but lambda values that cannot be used in function calls are a bit pointless, anyhow. So we prefer to formulate a constraint that forbids the return of lambda values into contexts where any of their lvalue captures are dead.

This is done by inserting a sentence in 6.5.2.6 p5 that stipulates that a lambda value can only be returned into a context where any of its lvalue captures is defined with the same name that is referring to the same object.

With all of this we have the following properties of lvalue closures:

— Lvalue closures cannot occur in file scope, because there are no automatic variables to which an lvalue capture could refer.
— Functions can never return lvalue closures, because none of their automatic variables are defined in the calling context.
— Since closures have unspecified types without declaration syntax, lvalue closures (seen as objects of lambda type) cannot be copied out of the block in which they live.
— Lvalue closures can only escape a block via a **`return`** statement of another lambda, which is restricted by the constraint.

VI. QUESTIONS FOR WG14

(1) Does WG14 want the lvalue capture feature for C23 along the lines of N2737?
(2) Does WG14 want to force lambda objects that have lvalue closures with **`register`** storage class to also be of that storage class as specified in N2737?
(3) Does WG14 want to integrate the changes as specified in N2737 into C23?

References

Jens Gustedt. 2021a. *Function literals and value closures*. Technical Report N2736. ISO. available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2736.pdf.

Jens Gustedt. 2021b. *Improve type generic programming*. Technical Report N2734. ISO. available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2734.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 N2738. ISO. available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2738.pdf.

Jens Gustedt. 2021e. *Type inference for variable definitions and function return*. Technical Report N2735. ISO. available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2735.pdf.

VII. PROPOSED WORDING

The proposed text is given as diff against N2736.

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

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

8 **NOTE** Internal and external linkage is used to access objects or functions that have a lifetime of the whole program execution. It is therefore usually determined before the execution of a program starts. For variables with a lifetime that is not the whole program execution and that are accessed from lambda expressions an additional mechanism called capture is available that dynamically provides the access to the current instance of such a variable within the active function call that defines it.

Forward references: storage durations of objects (6.2.4), declarations (6.7), expressions (6.5), external definitions (6.9), statements (6.8).

6.2.3 Name spaces of identifiers

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

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

— the *tags* of structures, unions, and enumerations (disambiguated by following any[35] of the keywords **`struct`**, **`union`**, or **`enum`**);

— the *members* of structures or unions; each structure or union has a separate name space for its members (disambiguated by the type of the expression used to access the member via the `.` or `->` operator);

— all other identifiers, called *ordinary identifiers* (declared in ordinary declarators or as enumeration constants).

Forward references: enumeration specifiers (6.7.2.2), labeled statements (6.8.1), structure and union specifiers (6.7.2.1), structure and union members (6.5.2.3), tags (6.7.2.3), the **`goto`** statement (6.8.6.1).

6.2.4 Storage durations of objects

1 An object has a *storage duration* that determines its lifetime. There are four storage durations: static, thread, automatic, and allocated. Allocated storage is described in 7.22.3.

2 The *lifetime* of an object is the portion of program execution during which storage is guaranteed to be reserved for it. An object exists, has a constant address,[36] and retains its last-stored value throughout its lifetime.[37] If an object is referred to outside of its lifetime, the behavior is undefined. The value of a pointer becomes indeterminate when the object it points to (or just past) reaches the end of its lifetime.

3 An object whose identifier is declared without the storage-class specifier **`_Thread_local`**, and either with external or internal linkage or with the storage-class specifier **`static`**, has *static storage duration*. Its lifetime is the entire execution of the program and its stored value is initialized only once, prior to program startup.

4 An object whose identifier is declared with the storage-class specifier **`_Thread_local`** has *thread storage duration*. Its lifetime is the entire execution of the thread for which it is created, and its stored value is initialized when the thread is started. There is a distinct object per thread, and use of the declared name in an expression refers to the object associated with the thread evaluating the expression. The result of attempting to ~~indirectly~~ access an object with thread storage duration from a thread other than the one with which the object is associated is implementation-defined.

5 An object whose identifier is declared with no linkage and without the storage-class specifier **`static`** has *automatic storage duration*, as do some compound literals. The result of attempting to ~~indirectly~~ access an object with automatic storage duration from a thread other than the one with which the object is associated is implementation-defined.

---

[35] There is only one name space for tags even though three are possible.

[36] The term "constant address" means that two pointers to the object constructed at possibly different times will compare equal. The address can be different during two different executions of the same program.

[37] In the case of a volatile object, the last store need not be explicit in the program.

The first always has static storage duration and has type array of **char**, but need not be modifiable; the last two have automatic storage duration when they occur within the body of a function, and the first of these two is modifiable.

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 (6.7.10).

6.5.2.6 Lambda expressions

Syntax

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

capture-clause:
[*capture-list*$_{\text{opt}}$ **]**
[*default-capture* **]**

capture-list:
capture-list-element
default-capture
capture-list **,** *capture-list-element*

default-capture:
=
&

capture-list-element:
value-capture
lvalue-capture

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

lvalue-capture:
& *capture*

capture:
identifier

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

Constraints

2 A capture that is listed in the capture list is an *explicit capture*. If the ~~capture clause has~~ first element in the capture list is a default capture, `id` is the name of an object with automatic storage duration ~~of~~ in an enclosing block that is not an array, `id` is used within the function body of the lambda without redeclaration ~~, and~~ `id` is not an explicit capture ~~, and it is not~~ or a parameter, the effect is as if ~~a capture list had been specified with~~ `id` ~~as a member~~ `id` were a value capture (for an = token) or an lvalue capture (for an & token). Such a capture is an *implicit capture*. If the first element in the capture list is an = token, all other elements shall be lvalue captures; if it is an & token, all shall be value captures.

3 ~~Captures~~ Value captures without assignment expression or lvalue captures shall be names of complete objects with automatic storage duration of a block enclosing the lambda expression that ~~do not have array type and that~~ are visible and accessible at the point of evaluation of the lambda expression. Additionally, value captures shall not have an array type. An identifier shall appear at most once; either as an explicit capture or as a parameter name in the parameter list.

4 Within the lambda expression, identifiers (including explicit and implicit 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 lvalue captures:

— Objects or type definitions with VM type shall not be used.

— Objects with automatic storage duration shall not be evaluated.[112)]

5 A variable `id` is an *indirect lvalue capture* of a lambda expression and of the corresponding lambda value if it is one of the implicit or explicit lvalue captures, or recursively, if it is the indirect lvalue capture of one of the captures of lambda type. A lambda value that has an indirect lvalue capture `id` shall not be used as the expression of a **`return`** statement, unless that **`return`** statement is itself associated to another lambda expression for which `id` is an indirect lvalue capture that refers to the same object.[113)] If a lambda value with an indirect lvalue capture `id` is used to initialize an object λ of lambda type and `id` has been defined with the **`register`** storage class, λ shall also have the **`register`** storage class.

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

[112)] 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.

[113)] Since each closure expression may have a unique type, it is generally not possible to assign it to an object with lambda value or to a function pointer that is declared outside of its defining scope or to use it, even indirectly, through a pointer to lambda value. Therefore the present constraint inhibits the use of an lvalue closure outside of the widest enclosing scope of its defining closure expression in which all its lvalue captures are visible.

Semantics

7 The optional attribute specifier sequence in a lambda expression appertains to the resulting lambda value. 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*. A closure that has an indirect lvalue capture is called an *lvalue closure* , otherwise it is a *value closure* .

8 Similar to a function definition, a lambda expression forms a single block that comprises all of its parts. 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 addresses of value captures are not necessarily unique.

9 If a value 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 enclosing block that exists according to the constraints.[114)]

10 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,[115)] and that address is used to modify the underlying object, the behavior is undefined.

11 The evaluation of the explicit or implicit assignment expressions of value captures takes place during each evaluation of the lambda expression. The evaluation of assignment expressions for explicit value captures is sequenced in order of declaration; an earlier capture may occur within an assignment expression of a later one. The objects of automatic storage duration corresponding to implicit value captures are evaluated unsequenced among each other. The evaluation of a lambda expression is sequenced before any use of the resulting lambda value. For each call to a lambda value, explicit 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. Explicit value captures and earlier parameters may occur within the declaration of a later one.

12 The object of automatic storage duration `id` of the surrounding scope that corresponds to an lvalue 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. 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.[116)]

13 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. 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 **NOTE 1** A direct function call to a function literal expression can be modeled by first performing a conversion of the function

[114)] The evaluation rules in the next paragraph then stipulate 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.

[115)] 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.

[116)] That is, evaluation of `id` 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.

literal to a function pointer and then calling that function pointer.

15 **NOTE 2** A direct function call to a closure expression without default capture and with parameters

```
[ captures-no-default ] (decl1, ..., decln) {
  block-item-list
}(arg1, ..., argn)
```

can be modeled with a such a call to a closure expression without parameters

```
[ _ArgCap1 = arg1, ..., _ArgCapn = argn, captures-no-default ] (void) {
  decl1 = _ArgCap1;
  ...
  decln = _ArgCapn;
  block-item-list
}()
```

where $\_ArgCap_1, \ldots, \_ArgCap_n$ are new identifiers that are unique for the translation unit. This equivalence uses the fact that the evaluation of the argument expressions $arg_1, \ldots, arg_n$ and the original closure expression as a whole can be evaluated without sequencing constraints before the actual function call operation. In particular, side effects that occur during the evaluation of any of the arguments or the capture list will not effect one another. This not withstanding, side effects that have an influence about the evaluation of captures in the specified capture list or that determine the type of parameters occur sequenced as specified in the original closure expression.

Analogously, a closure expression with lvalue default capture

```
[&, capture-list] (decl1, ..., decln) {
  block-item-list
}(arg1, ..., argn)
```

can be modeled as follows

```
[&, _ArgCap1 = arg1, ..., _ArgCapn = argn, capture-list ] (void) {
  decl1 = _ArgCap1;
  ...
  decln = _ArgCapn;
  block-item-list
}()
```

Recommended practice

16 Implementations are encouraged to diagnose any attempt to modify a lambda type object other than by assignment.

17 To avoid lexical conflicts with the attribute feature (**??**) the appearance two consecutive [tokens in the translation unit that do not start an attribute specifier results in undefined behavior (**??**). It is recommended that implementations that do not accept such a construct issue a diagnosis. For applications, the portable use of a call to a lambda expression as an array subscript, for example, is possible by surrounding it with a pair of parenthesis.

18 **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
  auto const λ = [var](void){ printf("%d\n", var); };    // freeze var
  [&var](void){ var = 7; printf("%d\n", var); }();       // valid, prints 7
  λ();                                                   // valid, prints 5
```

```
  [ var](void){ printf("%d\n", var); }();                  // valid, prints 7
  [    ](void){ printf("%d\n", var); }();                  // invalid
  [var](void){ printf("%zu\n", sizeof var); }();           // valid, prints sizeof(int)
  [ 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

}
```

19 **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.

20 **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
```