HAL
archives-ouvertes.fr

Lvalue closures

Jens Gustedt

▶ **To cite this version:**

Jens Gustedt. Lvalue closures: proposal for C23. [Research Report] N2635, ISO JCT1/SC22/WG14. 2021, pp.8. hal-03106930

HAL Id: hal-03106930
https://hal.inria.fr/hal-03106930

Submitted on 12 Jan 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution| 4.0 International License

N2635

Lvalue closures 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 lvalue captures that can read and modify objects of surrounding scopes,

I. MOTIVATION

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

© 2021 by the author(s). Distributed under a Creative Commons Attribution 4.0 International License

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.

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.

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.

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

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 new paragraph 6.5.2.6 p6 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 scope in which they live.
— Lvalue closures can only escape a scope 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 N2635?
(2) Does WG14 want to integrate the changes as specified in N2635 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~~.

might yield 1 if the literals' storage is shared.

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

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

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

```
struct s { int i; };

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

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

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

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

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

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

6.5.2.6 Lambda expressions

Syntax

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

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

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

capture-default:
=
&

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

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

lvalue-capture:
& *capture*

capture:
identifier

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

Constraints

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

3 A capture that is listed in the capture list is an *explicit capture*. If the ~~capture clause is [=]~~ first element in the capture list is a capture default, `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 an explicit capture or a parameter, the effect is as if `id` ~~had been used in a capture list~~were a value capture (for an = token) or a reference 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.

4 ~~Captures~~ Value captures without assignment expression or lvalue captures 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. 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 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, that are not lvalue captures and that are declared with automatic storage duration shall only be evaluated within the assignment expression of a value capture.[111)]

6 A closure that has an explicit or implicit lvalue capture `id` shall not be used as the expression of a **`return`** statement, unless that **`return`** statement is itself associated to another closure for which `id` is an lvalue capture that refers to the same object.[112)]

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

Semantics

8 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*. A closure that has an lvalue capture is called an *lvalue closure* , otherwise it is a *value closure* .

9 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

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

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

[112)] 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.

until the end of the call, only that the address of value captures is not necessarily unique.

10 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 surrounding scope that exists according to the constraints.[113)]

11 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.

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.[115)]

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. 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
  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)
```

[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.

[115)] That is, lvalue conversion of `id` results in the same lvalue with the same type and address as for the scope surrounding the lambda.

```
  [    ](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, };
  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**

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