Lvalue closures

Jens Gustedt

Lvalue closures

ISO/IEC JTC 1/SC 22/WG14 N2737
WG21 P2307

Jens Gustedt

INRIA – Camus

ICube – ICPS
Université de Strasbourg

https://modernc.gforge.inria.fr/

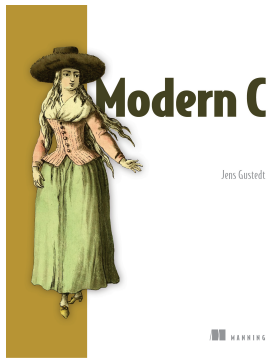

Table of Contents

Policy

- *extend* the standard
  - valid code remains valid
  - new feature integrates syntaxtically and semantically
- fix as much requirements as possible through constraints
  - specific syntax
  - explicit constraints
- avoid new undefined behavior
  - only, if property is not (or hardly) detectable at translation time
  - or we leave design space to implementations
- don't mess with ABI
  - no changes
  - no extensions

Table of Contents

Example: simple lvalue capture

```
auto const λ0 = [     ](void){ printf("%d\n", var); }; // invalid
auto const λ1 = [ var](void){ printf("%d\n", var); }; // freeze var
auto const λ2 = [&var](void){ printf("%d\n", var); }; // current
```

- It is a conscient user decision if and when ‘var‘ is evaluated
- these policies can even be combined:

```
auto const λ3 = [var0 = var, &var](void){
    printf("var␣previously:␣%d,␣now:␣%d\n", var0, var);
};
```

Example: access a lambda from another one

```
void matmult(size_t k0, size_t l0, size_t m0,
             double A0[k0][l0], double B0[l0][m0], double C0[k0][m0]) {
  // dot product, stride of m0 for B0, constant prop of l0 and m0
  auto const λ1 =
    [l0, m0](double  v1[l0], double B1[l0][m0], size_t m1) {
      double ret = 0.0;
      for (size_t i = 0; i < l0; ++i) ret += v1[i]*B1[i][m1];
      return ret;
    };
  // vector matrix product, ensure accessibility of λ1
  auto const λ2 =
     [l0, m0, &λ1](double v2[l0], double B2[l0][m0], double r[m0]) {
       for (size_t m = 0; m < m0; ++m) r[m] = λ1(v2, B2, m);
     };
  for (size_t k = 0; k < k0; ++k) λ2(A[k], B, C[k]);
}
```

- λ_1 is accessed by λ_2, but the address is not taken
- dimensions of B_1 and B_2 are fixed, once λ_1 and λ_2 are evaluated

Table of Contents

Terminology

C does not have C++' references

- *lvalue capture* and *lvalue closure*

Syntax

- a capture with a leading & is an lvalue capture

```
& identifier
```

- object is accessible (fresh and modifiable) throughout lambda body
- no support for C++ aliasing syntax

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

- support for default capture mechanisms
  - [=] or [=, &id, ...]
    - default is value capture, id lvalue capture
  - [&] or [&, id, ...]
    - default is lvalue capture, id value capture

Semantics

- lvalue capture <=> access to variable of an outerscope
- lvalue closure λ_1 can be returned from another lvalue closure λ_2

```
size_t n;
double R[10];
... // after a long decision process n is now fixed
auto const λ2 = [&R, n](void) {
    size_t m = really_complicated_computation(n);
    auto const λ1 = [&R, m](void) {
        // returns a double*
        return &R[m];
    };
    // returns a lambda value
    return λ1;
};
auto const Funλ = λ2();
for (i = 0; i < k; ++i)  *Funλ() += i;
```

- only if lvalue closures of λ_1 are a subset of those for λ_2
- safe access of outer scope

Semantics

- lvalue capture is not taking the address
- `register` variables possible
  (take some precautions)
- it is up to the implementation how to implement this
  (function, `goto`, `setjmp`, . . .)
- potential access conflict with another thread
  (when we have wide function pointers)

Table of Contents

Existing extensions

C++ and widely used gcc extensions

	language	value capture	lvalue capture
nested function	gcc' C	no	always
statement expression	gcc' C	no	always
blocks	objective C gcc' C and C++	default	object property
lambda	C++	explicit	explicit

possible mechanical transformation of existing code in many cases

New C23 feature `defer` ?

In other programming languages `defer` allows to freeze values.

Use with lambda

If `defer` sees a lambda value

```
defer [ var](void){ printf("%d\n", var); }; // freeze var
defer [&var](void){ printf("%d\n", var); }; // current
```

when leaving the current block for whatever reason:

execute the lambda

New C23 feature `defer` ?

Use without lambda

A `defer` with just an expression or compound statement

```
double* array = malloc(sizeof(double[n]));
defer   free(array);        // or equivalent
defer { free(array); }
```

is equivalent to

```
double* array = malloc(sizeof(double[n]));
defer [&](void){ free(array); };
```

is equivalent to

```
double* array = malloc(sizeof(double[n]));
defer [&array](void){ free(array); };
```