Basic lambdas for C (slides)

Jens Gustedt

Basic lambdas for C

ISO/IEC JTC 1/SC 22/WG14 N2892

WG21 P2303

Jens Gustedt

INRIA – Camus

ICube – ICPS
Université de Strasbourg

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

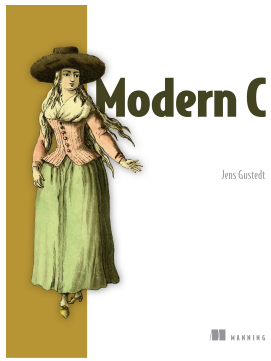

Table of Contents

1 Policy

2 Motivation

3 Design choices

4 Syntax

5 Existing extensions

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

1 Policy

2 Motivation

3 Design choices

4 Syntax

5 Existing extensions

Example: simple function literal

```
char const* stringArray[NUMEL];
...
qsort(stringArray, NUMEL, sizeof(stringArray[0]),
      [](void const* a, void const* b){
        char const*const* A = a;
        char const*const* B = b;
        return strcmp(*A, *B);
      });
```

- function literal appears close to its use
- return type is inferred as the same as **strcmp**
- function literal is converted to function pointer

```
int (*)(void const* a, void const* b)
```

- that function pointer is passed into **qsort**

Example: iterated function literal

```
auto strSort = [](size_t n, char const* stringArray[n]) {
  qsort(stringArray, n, sizeof(stringArray[0]),
        [](void const* a, void const* b){
          typeof(stringArray[0]) const* A = a;
          typeof(stringArray[0]) const* B = b;
          return strcmp(*A, *B);
        });
};
```

- a lambda expression that sorts string pointers
- the outer lambda has no **return** => return type is **void**
- use of **sizeof** and **typeof** makes it type safe (no evaluation!)
- no function with **void*** parameters is exported
- may be used directly in a function call
- may be converted to a function pointer

```
void (*)(size_t n, char const* stringArray[n])
```

Example: value closure

- value captures are evaluated when the lambda *expression* is met

```
// freeze ε to δ and have the function parameter dependent
auto const λ5ε = [δ = ε](double x, typeof(x) func(typeof(x))) {
  double h = δ * x;
  return (-func(x+2*h)+8*func(x+h)-8*func(x-h)+func(x-2*h))/(12*h);
};
```

- even if the capture is itself a lambda *value*

```
// also freeze a function, and have the parameter dependent
auto const λ5ε_func = [δ = ε, func = f](typeof(func(0)) x) {
  auto h = ε * x;
  return (-func(x+2*h)+8*func(x+h)-8*func(x-h)+func(x-2*h))/(12*h);
};
```

- this works if `f` is a function, function pointer or lambda

Table of Contents

1 Policy

2 Motivation

3 Design choices

4 Syntax

5 Existing extensions

Design choice, expression

Function definitions versus expressions

- function definition
  - naming is a burden
  - function use is distant from definition
  - nested functions may access all outer variables (no control mechanism)
- expression
  - anonymous
  - use where defined
  - lambda expression:
    - dedicated syntax to control access to outer variables

Design choice, capture

Capture model

- *visibility* of (almost) all identifiers as usual
- *access* is different
  - identifiers with linkage => business as usual
  - static identifiers without linkage => ok if not VM
  - automatic variables: default is *no capture* ([] , function literal)
  - value capture by explicit request ([hui = fui])
  - identifier capture by explicit request ([&hui])
  - options N2893
    - shadow capture by explicit request ([bla])
    - *capture all* as identifier capture ([&])
    - *capture all* as shadow capture ([=])

Design choices, function call

Call sequence

- lambda values are only visible in the same TU
- escape from TU only if function literal and converted to function pointer
- closure => addressless `static` function with *no escape* guarantee
- no ABI change

Design choices

Interoperability

- all is fixed at *translation time*
- no linker dependency

Not available

recursion	only if function literal via function pointer
stdarg	no variable argument list allowed
VM types	would be evaluated

Table of Contents

1 Policy

2 Motivation

3 Design choices

4 Syntax

5 Existing extensions

Deviation from C++ – maybe added later

No return type specification

- C++ has the syntax "–> *return-type*" as in

```
[](double x, double h, double (*func)(double)) -> double {
  return (-func(x+2*h)+8*func(x+h)-8*func(x-h)+func(x-2*h))/(12*h);
}
```

Make captures mutable

- C++ has the keyword mutable

```
[δ = ε](double x, double (*func)(double)) mutable {
  if (condition) δ += 1.E-7;
  double h = δ * x;
  return (-func(x+2*h)+8*func(x+h)-8*func(x-h)+func(x-2*h))/(12*h);
}
```

Table of Contents

1 Policy

2 Motivation

3 Design choices

4 Syntax

5 Existing extensions

Existing extensions

C++ and widely used gcc extensions

	language	value	identifier	shadow
nested function	gcc' C	no	always	no
statement expression	gcc' C et al.	no	always	no
blocks	objective C gcc' C and C++	no	property	default
lambda	C++	explicit	explicit	explicit