# Type-generic lambdas

Jens Gustedt

# Type-generic lambdas

## ISO/IEC JTC 1/SC 22/WG14 N2924

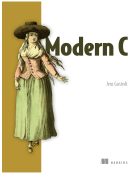
## WG21 P2306

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 syntactically 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

# Current state

## Proposals for C23 voted favorably by WG14

- type inference
  - `typeof`, from (type) expressions
  - `auto`, for definitions of
    - objects
    - functions
- lambdas
  - function literals
  - closures
- `defer`

# Current state

## Interrelationship

- `auto` needs `typeof`
- lambda needs
  - `auto` for the return type
  - `auto` for value captures
  - `auto` for variables with lambda type

# Current state

## Function literal (lambda without capture)

- expression
- small code snippet
- most outer identifiers are visible (compile time information)
  - types, enumeration constants, ...
  - exception: VM types
- outer identifiers are accessible
  - identifiers with static or thread storage duration (independent of linkage)
  - automatic variables are not accessible (only **`sizeof`**, **`typeof`**)
- can be converted to a function pointer

# Current state

## Closure

- *identifier capture*: access an outer variable
- *value capture*: frozen value of an expression
- *shadow capture*: frozen value of an automatic variable
- *cannot* be converted to a function pointer (needs extra state)

# Table of Contents

# Example 1: type generic lambda by using captures

```
#define MAXIMUM0(X, Y)                                       \
[MAXIMUM_A = (X), MAXIMUM_B = (Y)](void){                    \
      auto const a = MAXIMUM_A;                              \
      auto const b = MAXIMUM_B;                              \
      return (a < 0)                                         \
             ? (b < 0) ? (a < b) ? b : a : b                 \
             : (0 < b) ? (b < a) ? a : b : a;                \
    }()
```

- directly used in a function call
- no conversion to function pointer (captures!, call!)
- auxiliar names for captures
- redefinition as local variables
  - no interference with existing variables
  - respect evaluation order

# Example 1: rewrite with auto parameters

```
#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))
```

- directly used in a function call
- no conversion to function pointer (call!)
- easier to read
- less errorprone (no auxiliary variables)

# Example 2: function literal with explicit 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);
```

- conversion to function pointer, no captures
- function-like macro, but with the types as arguments
- unnatural in the context of a call

```
auto maxy = MAXY(signed, unsigned)(-1, +1);
```

- errorprone for mixed arguments

```
auto maxy = MAXY(signed, unsigned)(+1, -1); // error → UINT_MAX
```

# Example 2: rewrite with auto parameters

```
#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;
```

- conversion to function pointer, no captures
- object-like macro (similar to function name)
- natural in the context of a call

```
auto maxy = MAXIMUM(-1, +1);
```

- works well for mixed arguments

```
auto maxy = MAXIMUM(+1, -1); // same
```

# Table of Contents

# Design choices

## Permitted contexts

- function call
- conversion to function type

## Forbidden context

- variable of lambda type
  - type generic lambdas have incomplete type

## Rewrite of parameters

according to the usual rules of parameters

- array to pointer
- function to pointer

# Table of Contents

# Function pointers

## provide pointers to the user

```
#define COSINUS [](auto x){ ... iteration in x ... return x; }

double (*myfunc)(double) = COSINUS;
```

## future: function definitions by initialization

Several function *definitions* with different base types.

```
float       cosf(float)       = COSINUS;
double      cos (double)      = COSINUS;
long double cosl(long double) = COSINUS;
```