



## Improve type generic programming (slides)

Jens Gustedt

### ► To cite this version:

| Jens Gustedt. Improve type generic programming (slides). 2021. hal-03165732v1

**HAL Id: hal-03165732**

**<https://inria.hal.science/hal-03165732v1>**

Preprint submitted on 10 Mar 2021 (v1), last revised 2 Feb 2022 (v2)

**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 - NonCommercial 4.0 International License

# Improve type generic programming

ISO/IEC JTC 1/SC 22/WG14 **N2638**

WG21 **P2304**

Jens Gustedt

INRIA – Camus

ICube – ICPS

Université de Strasbourg



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



# Table of Contents

- 1 Introduction
- 2 A leveled specification
- 3 Existing type-generic features in C
- 4 Missing features
- 5 Common extensions



# Example

"Five" point tangent evaluation for approximation of a derivative:

$$(-\text{func}(x+2*h) + 8*\text{func}(x+h) - 8*\text{func}(x-h) + \text{func}(x-2*h)) / (12*h)$$

With a macro?

```
// WARNING: multiple argument evaluation
#define TANGENT5(FUNC, X, H) \
    (- FUNC((X)+2*(H)) + 8*FUNC((X)+ (H)) \
    - 8*FUNC((X)- (H)) + FUNC((X)-2*(H))) / (12*(H))
```

How to create an interface that is simple and safe to use?

With a lambda:

```
auto const λ5 = [] (double x, double h, double (*func) (double)) {
    return (-func(x+2*h)+8*func(x+h)-8*func(x-h)+func(x-2*h)) / (12*h);
};
// Can be used in function call or for a function pointer
double (*fp) (double, double, double (*) (double)) = λ5;
```

# Example

or so:

```
// freeze  $\varepsilon$  to  $\delta$  and have the function parameter dependent
auto const  $\lambda 5\varepsilon = [\delta = \varepsilon] (\text{double } x, \text{typeof}(x) \text{ func}(\text{typeof}(x))) \{$ 
    double h =  $\delta * x$ ;
    return (-func(x+2*h)+8*func(x+h)-8*func(x-h)+func(x-2*h)) / (12*h);
};
```

or so:

```
// also freeze a function, and have the parameter dependent
auto const  $\lambda 5\varepsilon\_func = [\delta = \varepsilon, func = f] (\text{typeof}(func(0)) x) \{$ 
    auto h =  $\varepsilon * x$ ;
    return (-func(x+2*h)+8*func(x+h)-8*func(x-h)+func(x-2*h)) / (12*h);
};
```



# 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

- 1 Introduction
- 2 **A leveled specification**
- 3 Existing type-generic features in C
- 4 Missing features
- 5 Common extensions



# A leveled specification: type inference

Type inference from identifiers, value expressions and type expressions

See JeanHeyd's paper [N2619](#) on `typeof`

- keep qualifiers and `_Atomic`

[N2674](#) Type inference for variable definitions and function return  
`auto` feature

- type is inferred from initializer or `return`
- conversion: lvalue, array-to-pointer, function-to-pointer
- lose qualifiers and `_Atomic`



# A leveled specification: lambdas

## N2675 Simple lambdas: function literals and value closures

- primary use: function call expressions
- local variables are captured explicitly and by value
- conversion: no captures => function pointer

## N2634 Type-generic lambdas

- **auto** parameters, types are inferred from
  - function call
  - function pointer conversion

## N2635 Lvalue closures

- lvalue capture: named alias for local variable (pseudo-references)



# Table of Contents

- 1 Introduction
- 2 A leveled specification
- 3 Existing type-generic features in C
- 4 Missing features
- 5 Common extensions



# Existing type-generic features in C

- operators
- promotions and conversions
- macros
- variadic functions
- function pointers
- **void** pointers
- type-generic C library functions
- **\_Generic** primary expressions

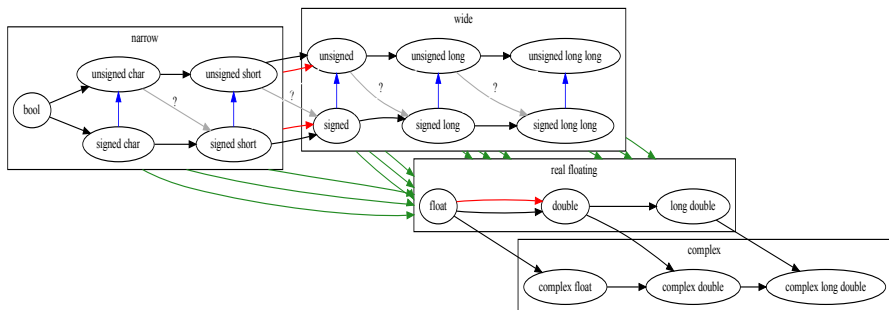


# Operators

- most binary operators, have the same type for both operands
- bit-wise operators are defined for
  - wide integer types
- additionally for multiplicative operators
  - real floating point types
  - complex types
- additionally for additive operators
  - object pointers



# promotions and conversions



- implicit conversion
- promotion and default argument conversion
- default arithmetic conversion



# Macros

- Macros for type-generic expressions (see intro above)
  - no local variables
  - dangerous because of multiple evaluation of arguments
- Macros placeable as statements
  - weird conventions, usually

<code>/* Macro */</code>		<code>/* Type-generic lambda */</code>
<code>#define myfeature(X)</code>	<code>\</code>	<code>#define myfeature</code>
<code>do {</code>	<code>\</code>	<code>[] (auto x) {</code>
<code>    typeGuess x = (X);</code>	<code>\</code>	<code>    /* do something with x */</code>
<code>    /* do something with x */</code>	<code>\</code>	<code>}</code>
<code>} while(false)</code>		
<code>/* Macro usage, conversion? */</code>		<code>/* Lambda usage, type safe */</code>
<code>myfeature(42);</code>		<code>myfeature(42);</code>

- Macros for declarations and definitions



# Type oblivion

## the user has all the burden

- Variadic functions
  - weird default conversions
  - weird library support (`va_list` a reference type?)
  - intrinsically unsafe
  - rarely used for new code
- `void*` pointers
  - unhuman effort has to be made to keep all the types correct
  - not even used by variadic functions
- function pointers
  - used with `void*` parameters for type-genericity (`bsearch`, `qsort`)



# Automatic type deduction

- type-generic C library functions
  - `<tgmath.h>`
  - `<stdatomic.h>`
- **\_Generic** primary expressions
  - difficult to extend
  - mostly restricted to function-like macros
  - not widely used



# Table of Contents

- 1 Introduction
- 2 A leveled specification
- 3 Existing type-generic features in C
- 4 **Missing features**
- 5 Common extensions



# Missing features

- temporary variables
  - temporary objects with a name
- controlled encapsulation
  - don't steal information from the surrounding scopes
  - don't pollute the surrounding scopes
- controlled constant propagation
  - control exactly what information is considered constant



# Missing features

- automatic instantiation of function pointers
  - missing for `<tgmath.h>`
- automatic instantiation of specializations
  - works well with controlled constant propagation
- direct type inference
  - avoid guessing or forcing a type
  - avoid implicit conversions



# Table of Contents

- 1 Introduction
- 2 A leveled specification
- 3 Existing type-generic features in C
- 4 Missing features
- 5 **Common extensions**



# Type inference

in C implementations and in other related programming languages

- **auto** type inference (`__auto_type__`)
- the **typeof** feature
- the **decltype** feature



# Identifiers of surrounding scopes

## use of identifiers distinguishes

- visibility by scope
- access
  - no linkage (same function, relative addressing)
  - internal linkage (same TU, global addressing)
  - external linkage (same program, linktime resolution)



# Identifiers of surrounding scopes

## What does an identifier mean in a local function?

- local function, short lifetime, multiple instances
  - when is the definition evaluated?
- when is an outer identifier evaluated?
  - evaluation of function definition or lambda expression
  - function call
- how much evaluation?
  - remains lvalue
  - lvalue conversion
  - promotion

# Lambdas: design space and terminology

## The design space for captures and closures

access to automatic variables

- evaluate when seeing lambda, *value capture*
  - rvalue (no address)
  - unmutable value (**const** qualified, addressable)
  - mutable value (C++ keyword `mutable`)
- evaluate when calling lambda
  - *lvalue capture*

## Terminology

- |                           |   |
|---------------------------|---|
| • <i>function literal</i> | $\Leftrightarrow$ anonymous function, no captures |
| • <i>closure</i>          | $\Leftrightarrow$ any capture                     |
| • <i>value closure</i>    | $\Leftrightarrow$ only value captures             |
| • <i>lvalue closure</i>   | $\Leftrightarrow$ at least one lvalue capture     |

# Lambdas: existing extensions

- Objective C's blocks
  - value capture per default
  - managed memory for lvalue captures
- Statement expressions
  - weird specification of the effective value
  - all captures are lvalue captures
- Nested functions
  - all captures are lvalue captures
  - separation of definition and call
- C++ lambdas
  - capture model chosen by user
    - [un]mutable value captures
    - lvalue captures

