Improve type generic programming (slides)

Jens Gustedt

To cite this version:

Jens Gustedt. Improve type generic programming (slides). 2022. hal-03165732v2

HAL Id: hal-03165732
https://inria.hal.science/hal-03165732v2

Preprint submitted on 2 Feb 2022

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 N2890

WG21 P2304

Jens Gustedt

INRIA – Camus

ICube – ICPS
Université de Strasbourg

https://gustedt.gitlabpages.inria.fr/modern-c/

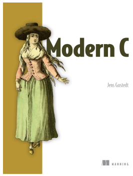


Table of Contents

Example

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

```
(-func(x+2*h) + 8*func(x+h) - 8*func(x-h) + 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 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);
};
```

or so:

```
// 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);
};
```

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

A leveled specification: type inference

Type inference from identifiers, value expressions and type expressions

See JeanHeyd's paper N2899 on `typeof`

`typeof`:	keep qualifiers and `_Atomic`
`remove_quals`:	remove qualifiers

N2891 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`
- as-if `auto` were replaced by a typeof expression
- no new types in an `auto` declaration!

A leveled specification: lambdas

N2892 Basic lambdas for C

- primary use: function call expressions
- no default captures, all captures are explicit
- local *identifiers* are captured by `&id` notation
- *values* are captured by `id=expr`
- modest syntax ambiguity
- conversion: no captures $\longrightarrow$ funtion pointer

N2893 Options for lambdas

- `&` for default identifier captures — migration path for gcc
- `=` for default *shadow* captures — clang
- `id` for explicit shadow captures
- more syntax ambiguity

A leveled specification: usage patters for lambdas

N2894 Type-generic lambdas

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

N2862 Function Pointer Types for Pairing Code and Data

Lead by Martin Uecker

- main feature is indepedent of lambdas:
  - provide API for existing ABI
    additional context pointer to function calls
- option to integrate lambdas

N2895 A simple defer feature for C

- use lambdas as syntax to describe `defer`

Table of Contents

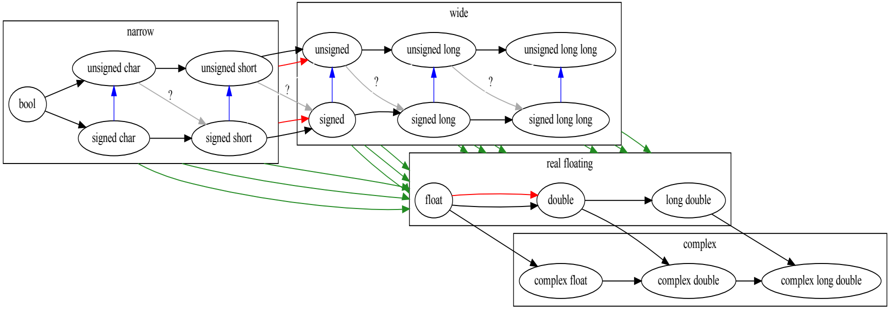
Existing type-generic features in C

- 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

```c
/* Macro */
#define myfeature(X)                 \
do {                                 \
   typeGuess x = (X);                \
   /* do something with x */         \
} while(false)

/* Macro usage, conversion? */
myfeature(42);
```

```c
/* Type-generic lambda */
#define myfeature                    \
[] (auto x) {                        \
   /* do something with x */         \
}

/* Lambda usage, type safe */
myfeature(42);
```

- 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?)
  - intrinsicly 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

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 inferrence
    - avoid guessing or forcing a type
    - avoid implicit conversions

Table of Contents

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 evalutated?
- when is an outer identifier evalutated?
  - 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 *expression* when seeing lambda, *value capture*
  - rvalue (no address)
  - unmutable value (**`const`** qualified, addressable)
  - mutable value (C++ keyword `mutable`)
- evaluate *variable* when seeing lambda, *shadow capture*
  - same possible differentiation as above
- evaluate when calling lambda
  - *identifier capture*

Terminology

function literal	<=> anonymous function, no captures
closure	<=> any capture

Table of Contents

Type inferrence

in C implementations and in other related programming languages

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

Lambdas: existing extensions

- Objective C's blocks — clang
  - shadow capture per default
  - managed memory for identifier captures
- Statement expressions — gcc, clang, intel, ibm xl, open64
  - weird specification of the effective value
  - all captures are identifier captures
- Nested functions — gcc
  - all captures are identifier captures
  - separation of definition and call
- C++ lambdas — since C++11
  - capture model chosen by user
    - [un]mutable value or shadow captures
    - identifier and *alias* captures
    - defaults & and =