

Alla scoperta di C++23

Jim Carrey

Marco Arena

Thank you! ❤



Why this meetup?

- ❖ We haven't met in Milan for 2 years
- ❖ I've just moved to Milan. What about meeting up more often?

A large, round pizza is the central focus, resting on a blue and white patterned plate. The pizza is topped with melted cheese, sliced meat, and fresh basil leaves. The background is slightly blurred, showing a dark surface.

Why this meetup?

C++23 is not a major release

- ❖ Informally called "the pandemic edition"
- ❖ Motivates adoption of C++20 by adding some missing pieces
- ❖ Introduces some additional bits (we are not going to see everything)
- ❖ By the way, 23 in Rome means “luck” (let’s say...)
- ❖ [Table with current compiler support](#)

In this talk

❖ Language additions

- ❖ Explicit object parameter
- ❖ `auto(x)`
- ❖ Multidimensional subscript operator
- ❖ Range-based for loop soap opera
- ❖ `[[assume]]`
- ❖ `if consteval`

❖ Library additions

- ❖ `import std & import std.compat`
- ❖ `flat_{your associative container here}`
- ❖ `expected`
- ❖ `generator`
- ❖ `msdspan`
- ❖ `<stacktrace>`
- ❖ `<print>`
- ❖ Formatting ranges
- ❖ Ranges additions
- ❖ Other small bits

Language Additions

Explicit object parameter

- ❖ When writing generic code (e.g. libraries), supporting const and non-const member functions commonly leads to **code duplication**

```
struct Wrapper
{
    T& operator[](std::size_t idx) { return m_data[idx]; }
    const T& operator[](std::size_t idx) const { return m_data[idx]; }
    // and what about T&& overloads?

    ...
};
```

Explicit object parameter

- ❖ You probably know a popular solution proposed by **Scott Meyers** in the old times

```
struct Foo {  
    const T& foo() const {  
        // do some work  
        return bar;  
    }  
  
    T& foo() {  
        return const_cast<T&>(const_cast<const Foo*>(this)->foo());  
    }  
};
```

Explicit object parameter

- ❖ C++23 introduces a new way of specifying non-static member functions
- ❖ A great talk about this feature [here](#)

```
struct X {  
    void foo(this X const& self); // same as void foo() const;  
// void foo() const; // Error: already declared  
};
```

Explicit object parameter

You are **not** allowed to:

Add any trailing implicit parameter specifiers

```
void foo(this X const& self, int i) const; // Error!  
void foo(this X const& self, int i) &&; // Error!
```

Refer implicitly nor explicitly to **this**

```
struct X {  
    int x;  
    void foo(this X const& self, int i) {  
        cout << x; // nope (implicitly)  
        cout << this->x; // nope (explicitly)  
        cout << self.x; // ok  
    }  
};
```

Override virtual functions

```
struct X {  
    virtual void foo();  
};  
  
struct Bar : X {  
    void foo(this const Bar&) override; // Error  
    void foo(this const Bar&); // it's not overriding  
    void foo(this const X&); // it's not overriding  
};
```

Explicit object parameter

A pointer to member function with EOP is an ordinary function

```
struct Foo {  
    int f(int, int) const&;  
    int g(this Y const&, int, int);  
};  
  
auto pf = &Foo::f;  
pf(y, 1, 2);          // error: pointers to member functions are not callable  
(y.*pf)(1, 2);      // ok  
std::invoke(pf, y, 1, 2); // ok  
  
auto pg = &Foo::g;  
pg(y, 3, 4);          // ok  
(y.*pg)(3, 4);      // error: pg is not a pointer to member function  
std::invoke(pg, y, 3, 4); // ok
```

Explicit object parameter + templates?



Explicit object parameter

- ❖ For template member functions, this feature allows deduction of type and value category
- ❖ Based on C++ template deduction rules
- ❖ Informally called **deducing this**

```
struct X {  
    template<typename Self>  
    auto&& foo(this Self&& self, int i) // auto&& foo(this auto&& self, int i)  
    { }  
  
X x;  
x.foo(1);           // Self = X&          (decltype(self) = X& && = X&)  
move(x).foo(1);    // Self = X&&         (decltype(self) = X&& && = X&&)  
const X xc;  
xc.foo(1);         // Self = const X& (decltype(self) = const X& && = const X&)
```

Explicit object parameter

- ❖ **Use Case 1:** avoid code duplication

```
struct Wrapper
{
    T& operator[](std::size_t idx) { return m_data[idx]; }
    const T& operator[](std::size_t idx) const { return m_data[idx]; }

    // covers both
    decltype(auto) operator[](this auto& self, std::size_t idx) { return m_data[idx]; }
};
```

Explicit object parameter

- ❖ **Use Case 2:** CRTP simplification

```
struct base {  
    template <class Self>  
    void f(this Self&& self);  
};  
  
struct derived : base {};  
  
derived my_derived;  
my_derived.f(); // in f, Self = derived& (this is not a dynamic dispatch!)
```

Explicit object parameter

- #### ❖ Use Case 2: CRTP simplification

Explicit object parameter

```
// C++23 CRTP
struct printer {
    auto& print(this auto const& self, const char* message)  {
        std::cout << message << "\n";
        return self;
    }
};

struct better_printer : printer {
    auto& endl() const {
        std::cout << std::endl;
        return *this;
    }
};
```

Explicit object parameter

- ❖ **Use Case 3:** Recursive lambdas

```
auto fact = [](this auto&& self, int n) {  
    if (n == 0)  
        return 1;  
    return n * self(n - 1);  
};  
  
cout << fact(5);
```

Explicit object parameter

- ❖ **Use Case 4: Pass this by value** (reasonable for small objects)

```
struct Foo
{
    int x;
    void bar(this auto self);
};

Foo foo {10};
foo.bar(); // the compiler can avoid allocating foo
```

Decay copy - auto(x)

- ❖ A motivating example:

```
vector<int> x = {1, 1, 2, 3, 4, 2, 2, 3};  
erase(x, x.front()); // ?
```

<https://wandbox.org/permlink/5AjlbtGIVE772hc>

Decay copy - auto(x)

- ❖ A motivating example:

```
vector<int> x = {1, 1, 2, 3, 4, 2, 2, 3};  
erase(x, x.front()); // ?
```

```
auto tmp = x.front(); // name + semantically wrong value category  
erase(x, tmp);
```

<https://wandbox.org/permlink/5AjlbtGIVE772hc>

Decay copy - auto(x)

- ❖ A motivating example:

```
vector<int> x = {1, 1, 2, 3, 4, 2, 2, 3};  
erase(x, x.front()); // ?
```

```
auto tmp = x.front(); // name + semantically wrong value category  
erase(x, tmp);
```

```
erase(x, int{x.front()}); // not-generic (need to know 'int')
```

<https://wandbox.org/permlink/5Ajlb1G1IVE772hc>

Decay copy - auto(x)

- ❖ A motivating example:

```
vector<int> x = {1, 1, 2, 3, 4, 2, 2, 3};  
erase(x, x.front()); // ?
```

```
auto tmp = x.front(); // name + semantically wrong value category  
erase(x, tmp);  
  
erase(x, auto{x.front()});
```

<https://wandbox.org/permlink/5Ajlb1G1IVE772hc>



Decay copy - `auto(x)`

- ❖ In other words, syntactic sugar for obtaining **a copy of a value** (actually, a **prvalue**)

```
void foo(const int& i)
{
    std::cout << &i;
}
```

```
int i = 10;
foo(auto{i});
```

Multidimensional subscript operator

- ❖ Subscript operator can now take an arbitrary number of subscripts (even 0)
- ❖ Cannot be non-member (as before)

```
struct Matrix
{
    decltype(auto) operator[](size_t x, size_t y) const
    {
        return m_data[x*m_cols + y];
    }
};

...
```

```
Matrix m = ...
std::cout << m[3, 1];
```

Multidimensional subscript operator

- ❖ 0-argument [] is possible:

```
struct Foo {  
    auto operator[]() const;  
};  
  
struct Bar {  
    auto operator[](auto...) const;  
};
```

```
Foo foo; foo[]; // ok  
Bar bar; bar[]; // ok
```

Multidimensional subscript operator

- ❖ By the way, **subscript** and **call operators** can be static in C++23:

```
struct Foo
{
    static auto operator[](auto...) const;
    static bool operator()(auto x);
};

auto is_even = [](int x) static { return x%2==0; };
```

Range-based for loop soap opera

```
{  
    auto && __range = range-expression ;  
    for (auto __begin = begin-expr, __end = end-expr; __begin != __end; ++__begin)  
    {  
        range-declaration = *__begin;  
        loop-statement  
    }  
}  
  
{  
    auto && __range = range-expression ;  
    auto __begin = begin-expr ;  
    auto __end = end-expr ;  
    for ( ; __begin != __end; ++__begin)  
    {  
        range-declaration = *__begin;  
        loop-statement  
    }  
}  
  
{  
    init-statement  
    auto && __range = range-expression ;  
    auto __begin = begin-expr ;  
    auto __end = end-expr ;  
    for ( ; __begin != __end; ++__begin)  
    {  
        range-declaration = *__begin;  
        loop-statement  
    }  
}
```

Range-based for loop soap opera

[N. Josuttis et. al.: R2644R1: Final Fix of Broken Range-based for Loop](#)

Project: ISO JTC1/SC22/WG21: Programming Language C++

Doc No: WG21 P2644R1

Date: 2022-11-11

Reply to: Nicolai Josuttis (nico@josuttis.de)

Co-authors: Herb Sutter, Titus Winter, Ha

Bryce Adelstein Lelbach, Pe

Audience: EWG, CWG

Issues: [cwg900](#), [cwg1498](#), [ewg120](#)

Previous: <http://wg21.link/P2012>

**Final Fix of Broken
Rev 1**



Range-based for loop soap opera

[N. Josuttis et. al.: R2644R1: Final Fix of Broken Range-based for Loop](#)

Project: ISO JTC1/SC22/WG21: Programming Language C++
Doc No: WG21 P2644R1
Date: 2022-11-11
Reply to: Nicolai Josuttis (nico@josuttis.de)
Co-authors: Herb Sutter, Titus Winter, Hana Dusíková, Fabio Fracassi, Victor Zverovich,
Bryce Adelstein Lelbach, Peter Sommerlad
Audience: EWG, CWG
Issues: [cwg900](#), [cwg1498](#), [ewg120](#)
Previous: <http://wg21.link/P2012>

**Final Fix of Broken Range-based for Loop,
Rev 1**



Range-based for loop soap opera

- ❖ TL;DR: Lifetimes of all temporaries within **range-expression** are extended

```
// until C++23 undefined behavior if foo() returns by value
for (auto& x : foo().items()) { // extended until the end of the loop
    std::cout << x;
} // destroyed here
```

Range-based for loop soap opera

- ❖ TL;DR: Lifetimes of all temporaries within **range-expression** are extended

```
// until C++23 undefined behavior if foo() returns by value
for (auto& x : foo().items()) { // extended until the end of the loop
    std::cout << x;
} // destroyed here
```

- ❖ BTW, since C++23 we can also introduce alias declarations as init-statements:

```
for (using elem_t = decltype(v)::value_type; elem_t i : v)
```

[[assume]]

- ❖ Introduced support for **assumptions**
- ❖ Specifies that an expression will always evaluate to true at a given point
- ❖ The expression is not evaluated (aka: it's not a **contract**)
- ❖ If an assumption does not hold, the behavior is undefined (allowing optimizations)

```
[[assume(x > 0)]]; // the compiler may assume x is positive
g(x / 2); // more efficient code possibly generated
x = 3; auto z = x;
[[assume((h(), x == z))]]; // the compiler may assume x==z after h()
h();
g(x); // compiler may replace this with g(3)!
```

std::unreachable

- ❖ A standard way to invoke **undefined behavior**
- ❖ Compilers may use such information to optimize things away (e.g. impossible branches)
- ❖ It's, basically, `[[assume(false)]]`;

```
string to_string(MyEnum e)
{
    switch (e) {
        case MyEnum::One: return "One";
        case MyEnum::Two: return "Two";
        ...
        default:
            unreachable();
    }
}
```

if consteval

- ❖ Tests whether a `constexpr` function is invoked from a constant-evaluated context
- ❖ Enables to call `consteval` functions from `constexpr` functions

```
consteval int foo(int i){ ... }  
int bla() { ... }
```

```
constexpr int bar(int i)  
{  
    if consteval {  
        return foo(i);  
    }  
    return bla();  
}
```

if consteval

- ❖ Useful also the other way around, to add "side effects" to constexpr functions invoked at run-time

```
constexpr int add(int a, int b)
{
    if not consteval {
        log("this is a side effect");
    }
    return a + b;
}
```

Library Additions

import std & import std.compat

- ❖ It's faster to bring in the entire standard library with `import std` (or `std.compat`) than it's to include a single header file (e.g. `<vector>`)
- ❖ Named macros (e.g. `assert`) are not available since modules don't expose macros
- ❖ Importing `std` brings in declarations and names in `std`, and also contents of C wrapper headers such as `<cstdio>` which provide functions like `std::printf`
- ❖ Importing `std.compat` brings in everything in `std` and adds the C runtime global namespaces such as `::printf`, `::fopen`. In other words, this makes easier to work with codebases that refer to many C runtime functions/types in the global namespace

```
import std;

int main() {
    std::cout << std::string{"hello"};
}
```

flat_map & friends

- ❖ Drop-in replacements for ordered non-flat types (`map`, `set`, `multi_map`, `multi_set`)
- ❖ **Different space and time complexity guarantees** than non-flat (aka: suitable for different use cases)
- ❖ Container adapters on **2 sequence containers** supporting **random access iterators** (keys, values)
- ❖ Still **ordered**
- ❖ Underneath, **contiguity is not required** (the API is a bit more flexible)
- ❖ Compared to non-flat types:
 -  **Insertions** and **removals** are **more expensive** (linear + possibly reallocation) and invalidate iterators
 -  **Exception safety** is **weaker** because of data movement
 -  Can't store **scoped types** (non-copyable & non-movable)
-  Faster **iteration** & cache-friendliness, random access, smaller memory usage (modulo backend)
-  Faster allocation and deallocation (modulo backend)

flat_map & friends

```
flat_map<int, std::string> flat { {1, "one"}, {2, "two"}, {3, "three"} };
cout << flat.lower_bound(1)->second << "\n";

for (auto [k, v] : flat) {
    std::cout << k << ":" << v << '\n';
}

// real use case
flat_map<std::string, std::vector<double>> parameters;
auto keys = SomeApi_GetKeys();
auto vals = SomeApi_GetVals();
parameters.replace(move(keys), move(vals));
// a bunch of queries on parameters (not changing anymore)
```

expected

- ❖ **Vocabulary type** to store either an expected value of type T or an error of type E
- ❖ Never valueless (e.g. `expected<T,E>{}` contains a `T{}`)
- ❖ No dynamic memory involved
- ❖ Supports some monadic operations (C++23 adds monadic operations to `std::optional` too)
- ❖ You know Rust's Result?

```
expected<double, error_code> parse(string_view sv) {
    // ... if parse error ...
    return unexpected(make_error_code(errc::invalid_argument));
}

auto result = parse(user_input);
if (result.has_value()) {
    // process result...
}
```

expected

- ❖ More interesting when used "functionally":

```
expected<coupon_info, coupon_error> find_coupon(string_view url);
expected<coupon_info, coupon_error> check_active(const coupon_info& ci);
expected<coupon_info, coupon_error> check_expired(const coupon_info& ci);

auto out = find_coupon("MILAN23")
    .and_then(check_active)
    .and_then([&](const auto& ci) {
        return check_expired(ci, sys_days{ January / 1 / 2027 });
    }).transform([](const auto& ci) {
        return ci.amount;
});
```

generator

- ❖ A view of the elements yielded by the evaluation of a coroutine (models `view` and `input_range` concepts)
- ❖ We have support for nested generators through [elements of](#)
- ❖ Check out [Alberto Barbati's awesome 30-min talk](#), a reference implementation is [here](#)

```
generator<string> files() {
    for (auto entry : filesystem::directory_iterator(".")) {
        co_yield entry.path().string();
    }
}

for (auto s : files() | views::take(3)) {
    std::cout << s << "\n";
}
```

mdspan

- ❖ Multi-dimensional **span** (non-owning view into a contiguous sequence of data)
- ❖ Provides minimal access operations (not slicing that should be merged into C++26)
- ❖ Supports:
 - ❖ dynamic and static **extents** (can be mixed together)
 - ❖ **layout** policies (e.g. Fortran-style's `layout_left`)
 - ❖ **accessor** policies to map indices to values

```
template<  
    class T,                                     // data type  
    class Extents,                                // shape (dimension)  
    class LayoutPolicy = layout_right,           // [i,j...] -> offset  
    class AccessorPolicy = default_accessor<T> // offset -> T& (ownership)  
> class mdspan;
```

mdspan

- ❖ `std::extents` allows to embed the **shape** (possibly mixing compile-time and run-time sizes)

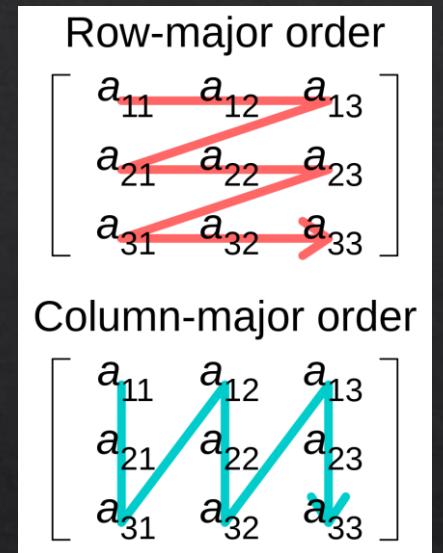
```
int buffer[6] = ...;
auto s1 = m/span(buffer, 3, 2); // all run-time extents

extents<int, 2, 3> shape;
auto s2 = m/span(buffer, shape); // all compile-time extents
static_assert(s.extent(0)==2, "Rows are two");
static_assert(s.extent(1)==3, "Cols are three");

// mixing compile-time and run-time extents
using shape_t = extents<int, dynamic_extent, 3, 320, 380>;
auto tensor = m/span<float, shape_t>(imageReshaped, 25);
```

mdspan

- ❖ `LayoutPolicy` defines how to **map** a multi-dimensional **index** into a one-dimensional **offset** (by default, `layout_right` that is C-style row-major layout)
- ❖ In addition to `layout_right` and `layout_left` (column-major), the standard provides `layout_stride` for regularly strided dimensions
- ❖ `AccessorPolicy` specifies how to **obtain a reference from an offset**
- ❖ Handles memory access patterns such as remote and atomic access, compression, non-aliasing semantics



Play with `mdspan` [here](#) (production-quality implementation + some doc [here](#))

<stacktrace>

- ❖ Minimal and handy abstraction on **invocation sequences** (e.g. main→foo→bar)
- ❖ Functions and constructors are **lazy** (information brought to you on demand)
- ❖ Frames are stored in **dynamic memory** (support for custom allocators)
- ❖ Represents a **snapshot** of the **stacktrace** as a **SequenceContainer** & **ReversibleContainer**
- ❖ Do not expect [Boost.Stacktrace](#)'s completeness

```
auto trace = stacktrace::current(); // all the entries
cout << trace << "\n"; // default pretty printing
cout << trace.size(); << "\n"; // number of entries (depth of the stack)
cout << trace[1].source_file()
```

<stacktrace>

```
int main() {
    set_terminate([]() {
        if (current_exception())
        {
            cout << "Something bad happened...\n\n";
            cout << stacktrace::current() << "\n";
        }
        abort();
    });
    ...
}
```

<stacktrace>

```
void api_facade()
{
    for (const auto& s : views::reverse(stacktrace::current()))
    {
        log(s.to_string());
    }
}
```

<print>

- ❖ **Formats** (`std::format`) a string and prints the result to a (`FILE*`) stream
- ❖ If the stream is omitted, the standard C output stream (`stdout`) is used
- ❖ Better than `std::cout` because – for example – the latter ignores encoding

```
#include <print>

print("hello {}\n", 10);
println("hello {}", 10);

if (FILE* stream = fopen(c:/temp/tmp.txt, "w"))
{
    print(stream, "File: {}", some_content);
    fclose(stream);
}
```

<https://wandbox.org/permlink/H5MgM63fu4tV5RDc>

Ranges additions

- ❖ A bunch of new views:
 - ❖ `repeat`
 - ❖ `cartesian_product`
 - ❖ `enumerate`
 - ❖ `zip`, `zip_transform`
 - ❖ `join_with`
 - ❖ `slide`, `stride`, `chunk`, `chunk_by`, `adjacent`, `adjacent_transform`, `pairwise`, `pairwise_transform`
 - ❖ `as_const`, `as_rvalue`
- ❖ A bunch of new algorithms:
 - ❖ `find_last`, `find_last_if`, `find_last_not_if`
 - ❖ `contains`, `contains_subrange`
 - ❖ `starts_with`, `ends_with`
 - ❖ `shift_left`, `shift_right`
 - ❖ `iota`
 - ❖ `fold_left`, `fold_right`, `fold_left_first`, `fold_right_last`, `fold_left_with_iter`, `fold_left_first_with_iter`
- ❖ Also, `ranges::to` which constructs a non-view (aka: container) from a range

Formatting ranges

- ❖ `std::format` supports ranges (can't change separators, in case you are wondering)

```
vector v = {1,2,3};  
cout << format("{}", v); // [1, 2, 3]
```

```
queue<int> q; v.push(1); v.push(2);  
print("{}", q); // [1, 2]
```

```
map<string, vector<int>> cache = {{"1", {1,2,3}}, {"5", {10,20}}};  
print("{}", cache); // {"1": [1, 2, 3], "5": [10, 20]}
```

Other Library Additions

Fixed width floating-point types

- ❖ New types added:

`std::float16_t`

`std::float32_t`

`std::float64_t`

`std::float128_t`

`std::bfloat16_t`

`string::resize_and_overwrite`

- ❖ Avoids zero-initializing a `std::string` when intended to be used as a char array to be filled (e.g. by a C API)

```
std::string str(1024, '\0'); // might be expensive
str.resize(External_Api(s1.data(), s1.size()));

// using this feature:
std::string str; // no initialization
str.resize_and_overwrite(1024, [](char* raw, size_t n){
    return External_Api(raw, n);
});
```

`string::resize_and_overwrite`

```
std::string str; // no initialization  
  
str.resize_and_overwrite(1024, [](char* raw, size_t n){  
    return External_Api(raw, n);  
});
```

str

string::resize_and_overwrite

```
std::string str; // no initialization  
str.resize_and_overwrite(1024, [](char* raw, size_t n){  
    return External_Api(raw, n);  
});
```

str	Gdòf/^^TYLHçDJFM...
-----	---------------------

string::resize_and_overwrite

```
std::string str; // no initialization  
str.resize_and_overwrite(1024, [](char* raw, size_t n){  
    return External_Api(raw, n);  
});
```

str

Hello\0YLMGH?^\$%

string::resize_and_overwrite

```
std::string str; // no initialization  
str.resize_and_overwrite(1024, [](char* raw, size_t n){  
    return External_Api(raw, n);  
});
```

str	Hello\0
-----	---------

spanstream

- ❖ Added support for fixed character buffer I/O operations

```
char* data_from_external_api = api_get_data(); // "10 20"
ispanstream stream{ span{data_from_external_api} };
int i, j;
stream >> i >> j;
cout << i << " " << j; // 10 20

char* destination[10]{};
ospanstream oss{ span{out} };
oss << "too many characters..." << "\n"; // won't allocate more memory
```

move_only_function

- ❖ Like `std::function` but it's not copyable and **supports move-only types**
- ❖ Invoking an empty `move_only_function` is UB (differently from `std::function` which throws)

```
move_only_function<int()> fun = [p = make_unique<int>(1)] {
    return *p;
});
```

- ❖ Note that this code is now ill-formed *:

```
function<const int&()> F([] { return 42; }); // Error since C++23
move_only_function<const int&()> F([] { return 42; }); // ditto
int x = F(); // until C++23 this was UB
```

*) C++23 introduces traits to detect reference binding to temporary
`reference_constructs_from_temporary` and `reference Converts_from_temporary`

to_underlying

- ❖ Converts an enumeration to its underlying type (e.g. int)
- ❖ Shortcut for `static_cast<underlying_type_t<Enum>>(e)`

```
enum class Pippo {A=20, B, C};  
cout << to_underlying(Pippo::A); // 20
```

Associative containers heterogeneous erasure

- ❖ You remember heterogenous lookup for associative containers (added to ordered containers in C++14 and to unordered in C++20)?
- ❖ Well, the capability has been extended to **erasure** as well

```
map<std::string, int, less<>> m = {...};  
const char* key = "some_key";  
m.erase(key); // no implicit conversion to string
```

Takeaway

- ❖ C++23 is not a major release and **mostly** introduces new **library features**
- ❖ **Explicit Object Parameter** is probably the most important language addition
- ❖ We are now **aware of things** that can spring up like mushrooms in codebases
- ❖ To stay relevant, **learn** how these things **work** and their **impact**
- ❖ As usual: **adding enables removing**
- ❖ Code and slides on [github](#)

Questions?