
Яндекс

Яндекс

C++17

Antony Polukhin
Полухин Антон

Boost libraries maintainer (DLL, LexicalCast, Any, TypeIndex, Conversion)
+ Boost.CircularBuffer, Boost.Variant
Представитель РГ21, national body

Оглавление

Оглавление

Автоматическое определение шаблонных параметров классов

Auto deduct class templates

```cpp
// C++14
std::pair<int, double> p14_0(17, 42.0);
auto p14_1 = std::pair<int, double> (17, 42.0);
```

Auto deduct class templates

```cpp
// C++14
std::pair<int, double> p14_0(17, 42.0);
auto p14_1 = std::pair<int, double> (17, 42.0);

// C++17
std::pair p17_0(17, 42.0);
auto p17_1 = std::pair(17, 42.0);
```

Auto deduct class templates

```
std::vector<int> v;


// C++14
typedef std::vector<int>::iterator v_iter;
std::pair<v_iter, v_iter>(v.begin(), v.end());
```

Auto deduct class templates

```
std::vector<int> v;


// C++14
typedef std::vector<int>::iterator v_iter;
std::pair<v_iter, v_iter>(v.begin(), v.end());


// C++17
std::pair p(v.begin(), v.end());
```

Almost-auto deduct class templates

```
template <typename EF> struct scope_exit;

auto foo() { /* очень много кода с return foo1(); */ };


// C++14

auto guard14 = foo();
```

Almost-auto deduct class templates

```
template <typename EF> struct scope_exit;

auto foo() { /* очень много кода с return foo1(); */ };


// C++14

auto guard14 = foo();


// C++17

scope_exit guard17 = foo();
```

Auto deduct class templates

```
namespace std {


    template <class T1, class T2>
    struct pair {
        // ...
        constexpr pair(const T1& x, const T2& y);
        // ...
    };


}
```

Auto deduct class templates

```
namespace std {


    template <class T1, class T2>
    constexpr pair(const T1& x, const T2& y) -> pair<T1, T2>;


}
```

Auto deduct class templates

```
std::vector<int> v;
```

```
std::pair p(v.begin(), v.end());
```

Auto deduct class templates

```
std::vector<int> v;


std::pair p(v.begin(), v.end());


// auto p = std::pair(v.begin(), v.end());
```

Auto deduct class templates

```
std::vector<int> v;


std::pair p(v.begin(), v.end());


// auto p = std::pair(v.begin(), v.end());
// template <class T1, class T2> pair(const T1& x, const T2& y) -> pair<T1, T2>;
```

Auto deduct class templates

```
std::vector<int> v;


std::pair p(v.begin(), v.end());


// auto p = std::pair(v.begin(), v.end());
// template <class T1, class T2> pair(const T1& x, const T2& y) -> pair<T1, T2>;
// T1 == std::vector<int>::iterator
// T2 == std::vector<int>::iterator
```

Almost-auto deduct class templates

```
// C++14
std::array<char, ???> a2{"Hello word"};
```

Almost-auto deduct class templates

```
// C++17
namespace std {
    // deduction guide
    template <class T, size_t N> array(const T (&array)[N]) -> array<T, N>;
}

std::array a2{"Hello word"};     // deduces the type `std::array<char, 11>`
```

std::to_chars и std::from_chars

std::to_chars и std::from_chars

```cpp
#include <sstream>                  //   :-(


template <class T>
T to_number_14(const std::string& s) {
    T res{};
    std::ostringstream oss(s); //   :-(
    oss >> res;
    return res;
}
```

std::to_chars и std::from_chars

```
template<typename _Facet>
locale::locale(const locale& __other, _Facet* __f) {
    _M_impl = new _Impl(*__other._M_impl, 1);
    __try { _M_impl->_M_install_facet(&_Facet::id, __f); }
    __catch(...) {
        _M_impl->_M_remove_reference();
        __throw_exception_again;
      }
    delete [] _M_impl->_M_names[0];
    _M_impl->_M_names[0] = 0;    // Unnamed.
}
```

std::to_chars и std::from_chars

```cpp
#include <utility>


template <class T>
T to_number_17(const std::string& s) {
    T res{};
    std::from_chars(s.data(), s.data() + s.size(), res); //   :-)
    return res;
}
```

std::*_v<T...>

std::*_v<T...>

```
// In <type_traits>
// template<class T, class U> struct is_same;

// C++14
std::cout << std::is_same<T1, T2>::value;
```

std::*_v<T...>

```
// In <type_traits>
// template<class T, class U> struct is_same;

// C++14
std::cout << std::is_same<T1, T2>::value;

// C++17
std::cout << std::is_same_v<T1, T2>;
```

std::*_v<T...>

```
template<class T, class U>
constexpr bool is_same_v = is_same<T, U>::value;
```

std::variant<T...>

std::variant<T...>

```cpp
#include <variant>


std::variant<int, std::string, double> v;
```

std::variant<T...>

```cpp
#include <variant>

std::variant<int, std::string, double> v;

union {
    int __a;
    std::string __b;
    double __c;
};
```

std::variant<T...>

```cpp
#include <variant>

std::variant<int, std::string, double> v;

union {
    int __a;
    std::string __b;
    double __c;
};

boost::variant<int, std::string, double> v;
```

std::variant<T...>

- std::variant не аллоцирует память для собственных нужд

std::variant<T...>

```cpp
#include <variant>


std::variant<int, std::string, double> v;
```

std::variant<T...>

```cpp
#include <variant>


std::variant<int, std::string, double> v;


v = 10;
assert(std::get<int>(v) == 10);
assert(std::get<0>(v) == 10);
```

std::variant<T...>

```cpp
#include <variant>

std::variant<int, std::string, double> v;

v = 10;
assert(std::get<int>(v) == 10);
assert(std::get<0>(v) == 10);

v = "Hello";
assert(std::get<std::string>(v) == "Hello");
assert(std::get<1>(v) == "Hello");
```

std::variant<T...>

```
std::variant<int, double> int_or_double{ 42.0 };
```

std::variant<T...>

```
std::variant<int, double> int_or_double{ 42.0 };


// std::get<std::string>(int_or_double);
```

std::variant<T...>

```
std::variant<int, double> int_or_double{ 42.0 };

// std::get<std::string>(int_or_double);

assert(std::get_if<int>(&int_or_double) == nullptr);
```

std::variant<T...>

```
std::variant<int, double> int_or_double{ 42.0 };

// std::get<std::string>(int_or_double);

assert(std::get_if<int>(&int_or_double) == nullptr);

int_or_double = 17;
assert(*std::get_if<int>(&int_or_double) == 17);
```

std::variant<T...>

```
constexpr std::variant<int, float> int_or_float{17};
```

std::variant<T...>

```
constexpr std::variant<int, float> int_or_float{17}; // noexcept
```

std::variant<T...>

```cpp
constexpr std::variant<int, float> int_or_float{17}; // noexcept
```

std::variant<T...>

```
constexpr std::variant<int, float> int_or_float{17}; // noexcept


Deleter d{ /* ... */ };
std::variant<int, std::shared_ptr<int> > v{std::in_place<1>, nullptr, std::move(d)};
```

std::variant<T...>

```cpp
constexpr std::variant<int, float> int_or_float{17}; // noexcept
```

```cpp
Deleter d{ /* ... */ };
std::variant<int, std::shared_ptr<int> > v{std::in_place<1>, nullptr, std::move(d)};
```

std::in_place

std::in_place<T> and std::in_place<N>

```
std::variant<std::string, int> vari(in_place<0>, v.begin(), v.end());

std::optional<std::string>     opti(in_place, v.begin(), v.end());

std::any                       anys(in_place<std::string>, v.begin(), v.end());
```

std::in_place<T> and std::in_place<N>

```
struct in_place_tag {};

template <class T>  in_place_tag in_place(unspecified1<T>) { return {}; };
template <int I>    in_place_tag in_place(unspecified2<I>) { return {}; };
                    in_place_tag in_place(unspecified3) { return {}; };
```

std::in_place<T> and std::in_place<N>

```
template <class T> struct unspecified1{};
template <std::size_t I> struct unspecified2{};
struct unspecified3{};

struct in_place_tag {};

template <class T>  in_place_tag in_place(unspecified1<T>) { return {}; };
template <int I>    in_place_tag in_place(unspecified2<I>) { return {}; };
                    in_place_tag in_place(unspecified3) { return {}; };
```

std::in_place<T> and std::in_place<N>

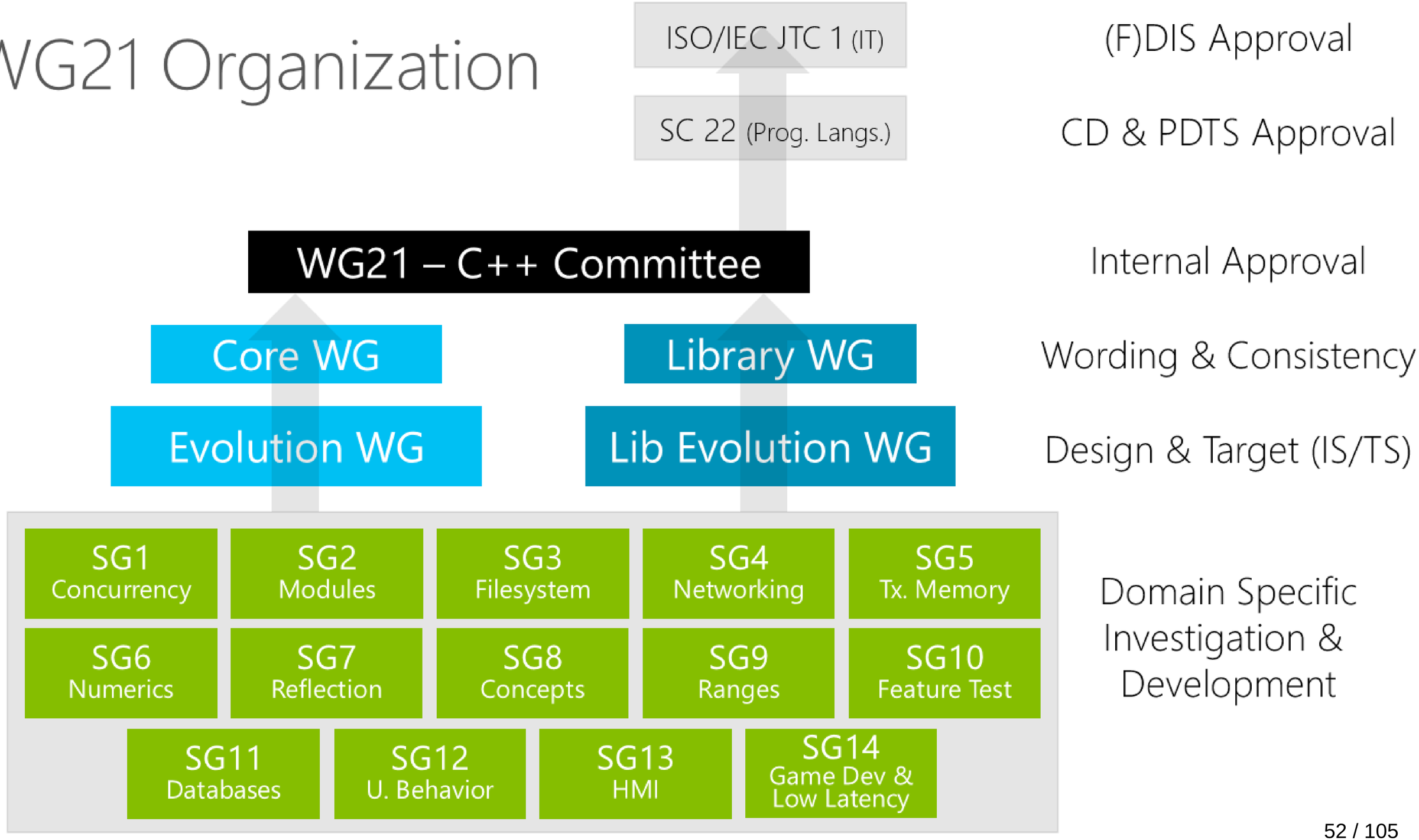
```
template <class T>
using in_place_type_t = in_place_tag(&)(unspecified1<T>);

template <int I>
using in_place_index_t = in_place_tag(&)(unspecified2<I>);

using in_place_t = in_place_tag(&)(unspecified3);
```

WG21

WG21 Organization



Инициализация в условных выражениях

cond-word (init-statement; condition)

```
if (auto state = get_state(); is_good(state)) {
    do_something(state);
} else {
    std::cerr << "Bad state:" << state;
}
```

cond-word (init-statement; condition)

```
if (auto state = get_state(); is_good(state)) {
    switch (std::lock_guard lk(m); state) {
    case ONE: /* ... */ break;
    case TWO: /* ... */ break;
    }

    do_something(state);
} else {
    std::cerr << "Bad state:" << state;
}
```

cond-word (init-statement; condition)

```
if (auto state = get_state(); is_good(state)) {
    switch (std::lock_guard lk(m); state) {
    case ONE: /* ... */ break;
    case TWO: /* ... */ break;
    }

    do_something(state);
} else {
    std::cerr << "Bad state:" << state;
}
```

cond-word (init-statement; condition)

```
if (auto state = get_state(); is_good(state)) {
    switch (std::lock_guard lk(m); state) {
    case ONE: /* ... */ break;
    case TWO: /* ... */ break;
    }

    do_something(state);
} else {
    std::cerr << "Bad state:" << state;
}
```

cond-word (init-statement; condition)

```
if (auto state = get_state(); is_good(state)) {
    switch (std::lock_guard lk(m); state) {
    case ONE: /* ... */ break;
    case TWO: /* ... */ break;
    }

    do_something(state);
} else {
    std::cerr << "Bad state:" << state;
}
```

std::size

std::size

```
int a[] = { -5, 10, 15 };


// ...
for (size_t i = 0; i < std::size(a); ++i)
    std::cout << a[i] << ',';
```

std::size

```
template <class T, std::size_t N>
constexpr std::size_t size(const T (&)[N]) noexcept {
    return N;
}
```

Многопоточные алгоритмы

MT algorithms

```
std::vector<int> v;

v.reserve(100500 * 1024);

some_function_that_fills_vector(v);


// Многопоточная сортировка данных

std::sort(std::execution::par, v.begin(), v.end());
```

MT algorithms

```
std::vector<int> v;

v.reserve(100500 * 1024);

some_function_that_fills_vector(v);


// Многопоточная сортировка данных

std::sort(std::execution::par, v.begin(), v.end());
```

MT algorithms

```
    std::vector<int> v;
    v.reserve(100500 * 1024);
    some_function_that_fills_vector(v);

    // Многопоточная сортировка данных
    std::sort(std::execution::par, v.begin(), v.end());

// In <execution>:
//    std::execution::seq
//    std::execution::par
//    std::execution::par_unseq
```

MT algorithms

```
std::sort(std::execution::par, v.begin(), v.end(), [](auto left, auto right) {
    if (!left || !right)
        throw std::logic_error("Zero values are not expected"); // std::terminate()

    return left < right;
});
```

MT algorithms

```
const bool not_ok = std::any_of(
    std::execution::par, v.cbegin(), v.cend(), [](auto v) noexcept { return !v; }
);

if (not_ok)
    throw std::logic_error("Zero values are not expected");

std::sort(std::execution::par, v.begin(), v.end(), [](auto left, auto right) noexcept {
    return left < right;
});
```

Структурное связывание

Structured binding

```
auto safe_info_14() {
    auto d = get_device_info();
    if (!d.first)                   // first? Что в нём хранится?
        throw safe_info_exception();
    return d.second;                // second?
}
```

Structured binding

```
using device_info
    = std::array<char, 1024 * 640>; // 640КБ должно быть достаточно для каждого :)

std::pair<bool, device_info> get_device_info() noexcept;

auto safe_info_14() {
    auto d = get_device_info();
    if (!d.first)                   // first? Что в нём хранится?
        throw safe_info_exception();
    return d.second;                // second?
}
```

Structured binding

```
auto safe_info_17() {
    auto [ok, info] = get_device_info();
    if (!ok)
        throw safe_info_exception();
    return info;
}
```

Structured binding

```
struct point {
    point() = delete;
    long double dimensions[3];
};
point& get_point_of_interest();
```

Structured binding

```
struct point {
    point() = delete;
    long double dimensions[3];
};
point& get_point_of_interest();

// ...
auto& [x, y, z] = get_point_of_interest();
x += 42.0;
y += 17.0;
z += 3.14;
```

Structured binding

```
std::map<int, short> m;
// ...

for (auto& [client_id, port]: m) {
    port = ::open_port_for(client_id);
}
```

Constexpr lambda

Constexpr lambda

```
template <class... T>
constexpr bool to_bool(const std::variant<T...>& var);
```

Constexpr lambda

```
template <class... T>
constexpr bool to_bool(const std::variant<T...>& var) {
    if (var.valueless_by_exception())
        return false;
```

Constexpr lambda

```
template <class... T>
constexpr bool to_bool(const std::variant<T...>& var) {
    if (var.valueless_by_exception())
        return false;

    return std::visit([](const auto& v) -> bool {
        return v;
    }, var);
}
```

Constexpr lambda

```
template <class... T>
constexpr bool to_bool(const std::variant<T...>& var) {
    if (var.valueless_by_exception())
        return false;

    return std::visit([](const auto& v) -> bool {
        return v;
    }, var);
}
```

Constexpr lambda

```
template <class... T>
constexpr bool to_bool(const std::variant<T...>& var) {
    if (var.valueless_by_exception())
        return false;

    return std::visit([](const auto& v) -> bool {
        return v;
    }, var);
}
```

if constexpr

if constexpr

```
template <class ...T>
auto vectorize(const T&... args) {
    constexpr std::size_t vector_length = 3u;
    constexpr std::size_t count = sizeof...(args);

    // ...
```

if constexpr

```
template <class ...T>
auto vectorize(const T&... args) {
    constexpr std::size_t vector_length = 3u;
    constexpr std::size_t count = sizeof...(args);

    if constexpr (count % vector_length != 0) {
        return vectorize(args..., 0);
    } else {
        return compute(args...);
    }
}
```

Splicing Maps and Sets

Splicing Maps and Sets

```
struct user {
    std::string                     bio;
    std::string                     address;
    std::vector<unsigned char>      photo;
    std::array<unsigned char, 128>  key;

    // ...
};
```

Splicing Maps and Sets

```
class user_registry {
    std::unordered_map<std::string, user> data_;

public:
    void update(const std::string& old_name, std::string new_name);
};
```

Splicing Maps and Sets

```
// C++14
void user_registry::update(const std::string& old_name, std::string new_name) {
    auto it = data_.find(old_name);
    if (it == data_.cend())
        return;

    user user_copy = std::move(it->second);
    data_.erase(it);
    data_.emplace(std::move(new_name), std::move(user_copy));
}
```

Splicing Maps and Sets

```
// C++17
void user_registry::update(const std::string& old_name, std::string new_name) {
    auto node = data_.extract(old_name);
    if (!node)
        return;

    node.key() = std::move(new_name);
    data_.insert(std::move(node));
}
```

std::string_view

std::string_view

```
// C++14
void foo(const std::string& value);
```

std::string_view

```
// C++14
void foo(const std::string& value);
void foo(const char* value);
```

std::string_view

```
// C++14
void foo(const std::string& value);
void foo(const char* value);
void foo(const char* value, std::size_t length);
```

std::string_view

```
// C++14
void foo(const std::string& value);
void foo(const char* value);
void foo(const char* value, std::size_t length);

// C++17
void foo(std::string_view value);
```

std::string_view

```
// C++14
template <class CharT>
void foo(const std::basic_string<CharT>& value);

template <class CharT>
void foo(const CharT* value);

template <class CharT>
void foo(const CharT* value, std::size_t length);
```

std::string_view

```
// C++17
template <class CharT>
void foo(std::basic_string_view<CharT> value);
```

std::filesystem

std::filesystem

```cpp
#include <filesystem>
#include <iostream>

int main() {
    std::filesystem::directory_iterator it("./");
    std::filesystem::directory_iterator end;

    for (; it != end; ++it) {
        std::filesystem::file_status fs = it->status();
        // ...
```

std::filesystem

```
std::filesystem::file_status fs = it->status();
switch (fs.type()) {
case std::filesystem::file_type::regular:
    std::cout << "FILE       ";
    break;
case std::filesystem::file_type::symlink:
    std::cout << "SYMLINK    ";
    break;
case std::filesystem::file_type::directory:
    std::cout << "DIRECTORY  ";
    break;
```

std::filesystem

```
        if (fs.permissions() & std::filesystem::owner_write) {
            std::cout << "W ";
        } else {
            std::cout << "  ";
        }

        std::cout << it->path() << '\n';
    } /*for*/
} /*main*/
```

std::filesystem

```
using namespace std::filesystem;


path read_symlink(const path& p);

path read_symlink(const path& p, std::error_code& ec);
```

std::filesystem

```
using namespace std::filesystem;


path read_symlink(const path& p);

path read_symlink(const path& p, std::error_code& ec);
```

std::filesystem

```
using namespace std::filesystem;


path read_symlink(const path& p);
path read_symlink(const path& p, std::error_code& ec);
```

Прочее...

Прочее...

- memory_order_consume
- std::function's allocators
- std::iterator/std::get_temporary_buffer/std::is_literal_type
- template <auto V> struct ...
- std::any
- std::optional
- [*this](){ /* ... */ }
- Math special functions
- Inline variables
- namespace foo::bar::example { /* ... */ }

Спасибо! Вопросы?