C++20

- Sprache
- Standard Library

Nicolas Lesser

Wichtig:

Diese Präsentation funktioniert nur, wenn ihr auch mitmacht.

Fragen dürft ihre gerne jederzeit stellen :)

```
std::vector Vec = {1, 2, 3};
std::size_t Index = 0;
for (const auto &Element : Vec) {
  // ...
  ++Index;
}
```

```
std::vector Vec = {1, 2, 3};
for (std::size_t Index = 0;
      const auto &Element : Vec) {
  // ...
  ++Index;
}
```

```
[f](auto&&... args) {
    return
        f(std::forward<decltype(args)>(args)...);
}
```

```
[f]<typename... Ts>(Ts&&... args) {
    return f(std::forward<Ts>(args)...);
}
```

```
std::set<int,
          [](int lhs, int rhs) {
             return lhs > rhs;
          }
> set;
```

```
std::set<int,
            decltype([](int lhs, int rhs) {
                return lhs > rhs;
            })
> set;
```

```
error: lambda expression in an
unevaluated operand
```

```cpp
// file scope
decltype([](){}) foo();
```

Geht nicht! Lambdas haben kein Linkage.

```
static decltype([](){}) foo();
```

Ok.

`[&, this]{};`	✓	(redundant)
`[&, *this]{};`	✓	
`[=, this]{};`	✗	(redundant)
`[=, *this]{};`	✓	

```cpp
template<typename ...Ts>
void foo(Ts &&...Args) {
  [Args...] {}();
}
```

+

```cpp
int Var = 0;
[Moved = std::move(Var)] {
  return Moved;
}();
```

? ? ?

```cpp
template<typename ...Ts>
void foo(Ts &&...Args) {
  [...Moves = std::move(Args)] {}();
}
```

```
struct Vector {
    int x;
    int y;
    int z;
}

First{.x = 1, .y = 3, .z = 2},
Second{.x = 3, .z = 1};
```

```
struct A { int x, y; };
struct B { A a; };

A a = {.y = 1, .x = 2};
int arr[3] = {[1] = 5};
B b = {.a.x = 0};
A a = {.x = 1, 2};
```

```
struct Person {
  int Age : 4;
};
```

+

```
struct Person {
  int Age = 0;
};
```

```
struct Person {
  int Age : 4 = 0;
};
```

error: bit-field member cannot have an in-class initializer

```
struct Foo {
  Foo(Foo&) = default;
};

struct Bar {
  Bar(const Bar&) = default;
  Foo f;
};

void call(Bar) {}
void call2(const Bar&) {}
```

```
Bar b;
call(b);
```

```
struct Foo {
  Foo(Foo&) = default;
};

struct Bar {
  Bar(const Bar&) = default;
  Foo f;
};

void call(Bar) {}
void call2(const Bar&) {}
```

```
Bar b;
call(b);
// const& can't bind to &
```

```
struct Foo {
  Foo(Foo&) = default;
};

struct Bar {
  Bar(const Bar&) = default;
  Foo f;
};

void call(Bar) {}
void call2(const Bar&) {}
```

```
Bar b;
call2(b);
```

```
struct Foo {
  Foo(Foo&) = default;
};

struct Bar {
  Bar(const Bar&) = default;
  Foo f;
};

void call(Bar) {}
void call2(const Bar&) {}
```

```
Bar b;
call2(b);
```

```
namespace Foo {
  struct Bar {};

  void callMe1(Bar) {}
  template<typename T, typename U>
  T callMe2(U) {}

}

callMe1(Foo::Bar());                ✓
callMe2<void>(Foo::Bar());          ✗
```

```cpp
struct Foo {
  template<typename T>
  T get() { return 1; }
};

template<typename T>
int get(T Var) {
  return Var.get<int>();
}

Foo Var;
int Result = get(Var);
```

```
struct Foo {
  template<typename T>
  T get() { return 1; }
};

template<typename T>
int get(T Var) {
  return Var.template get<int>();
}

Foo Var;
int Result = get(Var);
```

```
namespace Foo {
  struct Bar {};

  void callMe1(Bar) {}
  template<typename T, typename U>
  T callMe2(U) {}
}

callMe1(Foo::Bar());                  ✓
callMe2<void>(Foo::Bar());            ✗
```

typename ist manchmal nervig

```
template<typename T>
using Type = typename T::type;
```

🙁

typename ist manchmal nervig

```
template<typename T>
T::type get() {}
```

🙁

typename ist manchmal nervig

```
template<typename T>
void get(T::type) {}
```

🙁

typename ist manchmal nervig

```
template<typename T>
auto get(int Var) {
  return static_cast<T::type>(Var);
}
```

☹

typename ist manchmal nervig

```
template<typename T>
struct Foo : T {};
```

🙂

```
std::tuple{std::tuple{1}};
    => std::tuple<int>

std::vector{std::vector{1}};
    => std::vector<std::vector<int>>
```

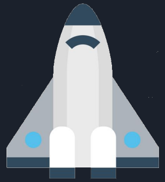
```
struct X { void foo() const&; };

X{}.foo(); // this is okay
(X{}.*&X::foo)(); // this is ill-formed
```

```cpp
#include <sstream>
#include <iterator>

struct X : std::stringstream {};

std::istream_iterator<char> begin(X& x) {
    return std::istream_iterator<char>(x);
}

std::istream_iterator<char> end(X& x) {
    return std::istream_iterator<char>();
}

int main() {
    X x;
    for (auto&& i : x)
        ;// do your magic here
}
```

```cpp
#include <memory>
#include <tuple>
#include <string>

struct X : private std::shared_ptr<int>
{
    std::string fun_payload;
};

template<int N> std::string& get(X& x) {if constexpr(N==0) return x.fun_payload;}

namespace std {
    template<> class tuple_size<X> : public std::integral_constant<int, 1> {};
    template<> class tuple_element<0, X> {public: using type = std::string;};
}

int main()
{
    X x;
    auto& [y] = x; // nein! :(
}
```

```
struct A {
  friend void foo();
private:
  int i;
};

void foo() {
  A a;
  auto x = a.i; // OK
  auto [y] = a; // ill-formed
}
```

operator<=>



```
struct Point {
  int x, y, z;
};
```

```
struct Entity {
  Point Location;
};
```

```
Entity Player, Spaceship{1, 1, 1};
// move, ...

if (Player.Location == Spaceship.Location)
  ; // do something
```

operator<=>

```
struct Point {
  int x, y, z;

  friend bool operator==(const Point &Lhs, const Point &Rhs) {
    return Lhs.x == Rhs.x &&
             Lhs.y == Rhs.y &&
             Lhs.z == Rhs.z;
  }
};
```

operator<=>

```
friend bool operator!=(const Point &Lhs, const Point &Rhs) {
  return !(Lhs == Rhs);
}
```

operator<=>

```
#include <utility>
using namespace std::rel_ops;

template<typename T>
bool operator!=(const T &Lhs, const T &Rhs) {
  return !(Lhs == Rhs);
}
```

operator<=>

```
struct Point {
  int x, y, z;

  friend bool operator==(const Point &Lhs, const Point &Rhs) {
    return Lhs.x == Rhs.x &&
           Lhs.y == Rhs.y &&
           Lhs.z == Rhs.z;
  }
};
```

operator<=>

```
struct Point {
  int x, y, z;

  std::strong_equality operator<=>(const Point &) const = default;
};
```

```
bool operator==(const Point &, const Point &) { /*...*/ }
```

```
bool operator!=(const Point &, const Point &) { /*...*/ }
```

operator<=>

```
std::strong_equality operator<=>(const Point &) const = default;
```

comparison category type

std::weak_equality

EQ, NEQ für jede Variable wenn = default;

operator<=>

```
class CaseInsensitiveString {
  std::string s;
public:
  std::weak_equality operator<=>(const CaseInsensitiveString& b) const {
    return case_insensitive_compare(s.c_str(), b.s.c_str());
  }
};
```

operator<=>

```
std::strong_equality operator<=>(const Point &) const = default;
```

⬆

comparison category type

std::weak_equality	**EQ, NEQ** für jede Variable wenn = default;
std::strong_equality	**EQ, NEQ** für jede Variable wenn = default;

operator<=>

```
struct Point {
  int x, y, z;

  std::strong_equality operator<=>(const Point &) const = default;
};
```

operator<=>

```
std::strong_equality operator<=>(const Point &) const = default;
```

comparison category type

std::weak_ordering

EQ, NEQ, LE(EQ), GT(EQ) für jede Variable wenn = default;

operator<=>

```
class CaseInsensitiveString {
  std::string s;
public:
  std::weak_ordering operator<=>(const CaseInsensitiveString& b) const {
    return case_insensitive_compare(s.c_str(), b.s.c_str());
  }
};
```

operator<=>

```
std::strong_equality operator<=>(const Point &) const = default;
```

comparison category type

std::weak_ordering **EQ, NEQ, LE(EQ), GT(EQ)** für jede Variable wenn = default;

std::strong_ordering **EQ, NEQ, LE(EQ), GT(EQ)** für jede Variable wenn = default;

operator<=>

```cpp
class uint128_t {
  std::uint64_t NumberArray[2];
  // low bits in NumberArray[1]
  // high bits in NumberArray[2]
public:
  std::strong_ordering operator<=>(const uint128_t &Rhs) const {
    if (auto cmp = NumberArray[1] <=> Rhs.NumberArray[1]; cmp != 0)
      return cmp;
    return NumberArray[0] <=> Rhs.NumberArray[0];
  }
};
```

operator<=>

```
std::strong_equality operator<=>(const Point &) const = default;
```

↑

comparison category type

std::partial_ordering

EQ, NEQ, LE(EQ), GT(EQ) für jede Variable wenn = default; wenn zwei Klassen keine Beziehung haben können.

Concepts

```
struct Int {
  std::string to_string() const;
};

template<typename T>
void logError(const T &Value) {
  std::cerr << to_string(Value) << std::endl; // std::to_string or .to_string
}
```

Concepts

```
namespace detail {
  template<typename T, typename = std::string>
  struct has_to_string : std::false_type {};

  template<typename T>
  struct has_to_string<T, std::decay_t<decltype(std::declval<T>().to_string())>>
    : std::true_type {};
}

template<typename T, std::enable_if_t<detail::has_to_string<T>::value>* = nullptr>
std::string to_string(const T &Value) {
  return Value.to_string();
}

template<typename T, std::enable_if_t<!detail::has_to_string<T>::value>* = nullptr>
std::string to_string(const T &Value) {
  return std::to_string(Value);
}
```

Concepts

```
namespace detail {
  template<typename T, typename = std::string>
  struct has_to_string : std::false_type {};

  template<typename T>
  struct has_to_string<T, std::decay_t<decltype(std::declval<T>().to_string())>>
    : std::true_type {};
}

template<typename T>
std::string to_string(const T &Value) {
  if constexpr (detail::has_to_string<T>::value)
    return Value.to_string();
  else
    return std::to_string(Value);
}
```

Concepts

```
template<typename T>
concept HasToString = requires(T Value) {
  { Value.to_string() } -> std::string;
}

template<typename T>
std::string to_string(const T &Value) {
  if constexpr (HasToString<T>)
    return Value.to_string();
  else
    return std::to_string(Value);
}
```

Vorteile

1. einfacher zu schreiben und verstehen
2. bessere Fehlermeldungen
3. weniger Schreibarbeit

Concepts

```
template<typename T>
concept IsIntegral = std::is_integral_v<T>;
```

Muss ein bool sein

```
template<typename T>
concept IsIntegralSmall = std::is_integral_v<T> &&
                          sizeof(T) <= 4;
```

Concepts

```
requires(/*parameter list*/) {
  /*requirements*/
}
```

Expression

```
requires  {
  /*requirements*/
}
```

Concepts

1. **simple requirement**

```
static_assert(requires(int t) {
  t * t;
}); // ok
```

Concepts

2. **type requirement**

```
static_assert(requires {
  typename std::true_type::value_type;
}); // ok
```

Concepts

3. **compound requirement**

```
static_assert(requires(int t) {
  { t + t } -> int;
}); // ok
```

Concepts

4. **nested requirement**

```
static_assert(requires {
  requires(int t) { t + t; };
}); // ok
```

Concepts

```
template<typename T, typename ...Ts>
concept IsConstructible = requires(Ts ...Args)
{
  T(Args...);
};
```

Concepts

```
template<~~typename~~ HasToString T>
std::string get(const T &Value) {
  return Value.to_string();
}
```

Concepts

```
template<typename T>
requires HasToString<T> &&
         IsConstructible<T>
std::string get(const T &Value) {
  return Value.to_string();
}
```

Concepts

```
template<typename T>
std::string get(const T &Value) requires
    HasToString<T> &&
    IsConstructible<T> {
  return Value.to_string();
}
```

Concepts

```
const char *getCpp17()
    requires __cplusplus == 201703L {
  return "C++17!";
}
```

❌ Name mangling?

Concepts

```
[](const auto &Value)
    requires HasToString<decltype(Value)> {
  return Value.to_string();
}
```

Concepts

```
[]<HasToString T>(const T&Value) {
  return Value.to_string();
}
```

Concepts

```
std::string get(const HasToString{} &Value) {
  return Value.to_string();
}
```

Noch nicht in C++20

```
bool isBigEndian() {
  union {
    uint32_t Num;
    char Decompose[4];
  } Hack = {0x01020304};

  return Hack.Decompose[0] == 1;
}
```

```
bool isBigEndian() {
  return std::endian::native == std::endian::big;
}
```

```
enum class endian {
  little = /*unspecified*/,
  big = /*unspecified*/,
  native = /*unspecified*/
};
```

endian::little, endian::big oder ...

```
[[nodiscard]] int foo() { return 0; }
int main() {
  foo();
}
```

ignoring return value of 'int foo()', declared with attribute nodiscard

std::malloc	Nein
std::async	Ja
std::realloc	Nein
C::empty()	Ja
std::launder	Ja
std::printf	Nein
Alloc::allocate()	Ja

```
std::mutex m;
std::scoped_lock l(m, std::adopt_lock);
```

❌

```
std::variant<int, double> v1(3);
std::variant v2 = v1;
```

❌

```
std::mutex m;
std::scoped_lock l(std::adopt_lock, m~~, std::adopt_lock~~);
```

```
std::variant<int, double> v1(3);
std::variant v2 = v1;
```

✓

```
auto UPtr = std::make_unique<int>();          ✓
auto UPtrArr = std::make_unique<int[]>(1);    ✓


auto SPtr = std::make_shared<int>();          ✓
 auto SPtrArr = std::make_shared<int[]>(1);    ✗
```

std::decay → Forwarding references

```
template<typename T>
struct has_to_string<T,
            std::decay_t<decltype(std::declval<T>().to_string())>>
    : std::true_type {};
```

std::remove_cvref

std::basic_*streams sind problematisch weil ______________________________.

std::basic_*streams sind problematisch weil sie keine thread-safety garantieren.

→ **std::basic_osyncstream**

```
std::osyncstream(std::cout) << "Hallo Welt\n";
```

constexpr ALL THE THINGS!
std::find
std::accumulate
std::complex
std::all_of

std::atomic<shared_ptr<T>>

Nur ein Wrapper für thread-safe std::shared_ptr.

Vorher

```
std::atomic_store(&Ptr, std::make_shared<int>(1));
```

Nachher

```
Ptr.store(std::make_shared<int>(1));
```

std::atomic<*floating-point*>

std::move in <algorithms>, z.B. std::accumulate (2x schneller für std::string)

→ acc = binary_op(std::move(acc), *i)

.starts_with und .ends_with für std::string and std::string_view

```
int foo(int Var) {
  [[likely]]
  if (Var <= 10)
    return Var;
  return 100;
}
```

```
int foo(int Var) {
  [[unlikely]]
  if (Var > 10)
    return 100;
  return Var;
}
```

In Schleifen wichtig wenn Profile Guided Optimization fehlschlägt!

<ciso646> → <version>

std::unordered_map == std::unordered_map ignoriert Hash

std::basic_string::reserve reduziert die Kapazität nicht mehr

std::to_address(Obj) anstatt std::addressof(*Obj)

std::span

std::string_view für Arrays

Ptr und Size

BeginPtr und EndPtr

Raw Array

std::array

Container

std::span

std::string_view für Arrays

```
void prettyOutput(const std::vector<Data> &Vector) {
  std::cout << '[';

  for (auto &&Entry : Vector)
    std::cout << Entry << ',';

  std::cout << ']';
}
```

std::span

std::string_view für Arrays

```
template<typename Iter>
void prettyOutput(Iter Begin, Iter End) {
  std::cout << '[';

  for (auto It = Begin; Begin != End; ++It)
    std::cout << *It << ',';

  std::cout << ']';
}
```

std::span

std::string_view für Arrays

```
void prettyOutput(std::span<Data> Array) {
  std::cout << '[';

  for (auto &&Entry : Array)
    std::cout << Entry << ',';

  std::cout << ']';
}
```

Howard Hinnant’s Calendar and Time Zone Library

std::chrono::year → 2018y

std::chrono::month

std::chrono::day

Howard Hinnant’s Calendar and Time Zone Library

```
auto Date = 2018y/april/11;

auto TimePoint = std::chrono::local_days(2018y/april/11);

TimePoint += 19h + 30min;

auto TimeInNY = std::chrono::zoned_time("Asia/New_York", TimePoint);
```

Howard Hinnant’s Calendar and Time Zone Library

```
auto Today = zoned_time(current_zone(), system_clock::now());

auto NextMonth = floor<months>(Today) + 1;


auto NextPres = local_days(wed[2]/NextMonth/2018y)
                + 19h + 30min;
```

ENDE

Quellen

http://www.myiconfinder.com/icon/accept-business-check-checkmark-comleted-done-okey-ok-tick-agree-approved/8806

http://www.myiconfinder.com/icon/cross-close-delete-exit-logout-remove-uncheck-wrong-no-unselect/8798

https://www.iconexperience.com/g_collection/icons/?icon=emoticon_frown

https://www.iconexperience.com/g_collection/icons/?icon=emoticon_smile&style=standard

https://pngtree.com/freepng/flat-color-question-mark_756496.html

https://pngtree.com/freepng/gray-spacecraft_2906419.html

https://github.com/lefticus/constexpr_all_the_things/blob/master/presentation/title.png