C++20

by

Dr. Günter Kolousek

Neuerungen – ”Die großen 4”

- Konzepte
  - eher für Leute, die Libraries entwickeln
- Module
  - wird noch eine Weile dauern bis sich diese etablieren
    - weil der gesamte Buildprozess sich ändern wird
- Koroutinen
  - werden im täglichen Leben nicht so verwendet werden
    - zumindest nicht bis es entsprechende Bibliotheken gibt
- Ranges
  - wird sich im Alltag etablieren…

Neuerungen – Interessantes?

- vorzeichen behaftete `ints`: 2er Komplement
  - endlich…
- designated initializers
  - explizit und besser zu lesen
  - default-mäßige Members können weggelassen werden
- erweitertes range-basiertes `for`
  - praktisch und einheitlich
- spaceship operator
  - einfachere Definition von Vergleichsoperatoren
- `auto` außerhalb von Lambdas
  - praktisch, reduziert die Notwendigkeit von `template`
- Lambdas mit Template-Parameter
  - wenn `auto` mit Lambdas nicht reicht…
- `constinit` und `consteval`
  - zwingende Initialisierung und Auswertung zu Übersetzungszeit
- `span`
  - nicht besitzende Sicht auf eine Sequenz von Ojekten
    - ownership!

Neuerungen – Interessantes? - 2

…aber NYI in g++:

- `constexpr` Unterstützung für `string` & `vector`
- `using enum` innerhalb von `switch`
  - einfachere Verwendung von `enum class` in `switch`
- Formatierte Ein- und Ausgabe
  - aber vorhanden in Bibliothek `fmt`
    https://fmt.dev/latest/index.html
- Datumserweiterungen
  - aber vorhanden in Bibliothek `date`
    https://howardhinnant.github.io/date/date.html

Neuerungen – Interessantes? - 3

… für Anwendungen basierend auf Threads:

- joining threads: `jthread`
- `latch`, `barrier`, `counting_semaphore`, `binary_sempahore`
  - `binary_semaphore` ≡ `counting_semaphore<1>`
  - NYI in `g++`
- synchronisierte Ausgabe mittesl `osyncstream`
  - NYI in `g++`
- atomare Smart Pointer
  - NYI in `g++`

Designated Initializers

```cpp
#include <iostream>  // designated_initializers.cpp

struct Point3D {
    int x;
    int y{1};
    int z;
};

using namespace std;

int main() {
    Point3D p1{.x=1, .z=3};  // order matters
    cout << p1.x << ", " << p1.y << ", "
         << p1.z << endl;  // 1, 1, 3

    Point3D p2{.y=2, .z=3};  // x will be initialized!
    cout << p2.x << ", " << p2.y << ", "
         << p2.z << endl;  // 0, 2, 3
}
```

Erweitertes range-basiertes for

```cpp
#include <iostream>  // variables_for.cpp
#include <vector>
using namespace std;

int main() {
    vector<int> v{1, 1, 2, 3, 5, 8, 13};
    for (int sum{0}; auto value : v) {
        sum += value;
        cout << sum << endl;
    }
    // cout << sum << endl;  // sum not declared...
}
```

Vergleichsoperatoren

bisher: alle 6+ Operatoren schreiben (mühsam und meist trivial)!

```cpp
#include <iostream>  // comparision_operators.cpp
struct Int {
    int v;
    Int(int v=0) : v{v} {}

    bool operator<(const Int& other) const {
        return v < other.v; }
    bool operator==(const Int& other) const {
        return v == other.v; }
    bool operator!=(const Int& other) const {
        return !(*this == other); }
    bool operator<=(const Int& other) const {
        return *this == other || *this < other; }
    bool operator>(const Int& other) const {
        return other < *this; }
    bool operator>=(const Int& other) const {
        return *this == other || *this > other;
    } };
```

Vergleichsoperatoren – 2

```
using namespace std;

// then it could be used like this:
int main() {
    Int i1;
    Int i2{2};
    cout << (i1 == i2) << endl;  // 0
    cout << (i1 != i2) << endl;  // 1
    cout << (i1 < i2) << endl;  // 1
    cout << (i1 <= i2) << endl;  // 1
    cout << (i1 > i2) << endl;  // 0
    cout << (i1 >= i2) << endl;  // 0

    // wrong order in C++17:
    //cout << (0 == i1) << endl;
    // -> additional operators and friends have
    //    to be defined
}
```

Vergleichsoperatoren – 3

```cpp
#include <iostream> // comparision_operators2.cpp
struct Pair {  // comparing lexicographically!
    int a{1};  int b{2};

    bool operator<(const Pair& other) const {
        return a < other.a || a == other.a &&
          b < other.b; }
    bool operator==(const Pair& other) const {
        return a == other.a && b == other.b; }
    bool operator!=(const Pair& other) const {
        return !(*this == other); }
    bool operator<=(const Pair& other) const {
        return *this == other || *this < other; }
    bool operator>(const Pair& other) const {
        return other < *this; }
    bool operator>=(const Pair& other) const {
        return *this == other || *this > other;
    }
};
```

Vergleichsoperatoren – 4

```
using namespace std;

int main() {
    Pair p1;
    Pair p2{.b=3};
    cout << (p1 == p2) << endl;  // 0
    cout << (p1 != p2) << endl;  // 1
    cout << (p1 < p2) << endl;  // 1
    cout << (p1 <= p2) << endl;  // 1
    cout << (p1 > p2) << endl;  // 0
    cout << (p1 >= p2) << endl;  // 0
}
```

Spaceship Operator

```cpp
#include <iostream>  // spaceship_operator.cpp
using namespace std;

struct Pair {
    int a{1};  int b{2};
    auto operator<=>(const Pair& other) const {
       if (auto compare{a <=> other.a}; compare != 0)
           return compare;
       return b <=> other.b;
    }  // generates operators for: <, <=, >, >=
    bool operator==(const Pair& other) const {
        return (*this <=> other) == 0;
    }  // has to be defined separately!
    // since C++20: operator!= has not to be defined
    //   anymore...
};
```

Spaceship Operator – 2

```
int main() {
    Pair p1;  Pair p2{.b=3};
    cout << (p1 == p2) << endl;  // 0
    cout << (p1 != p2) << endl;  // 1
    cout << (p1 < p2) << endl;  // 1
    cout << (p1 <= p2) << endl;  // 1
    cout << (p1 > p2) << endl;  // 0
    cout << (p1 >= p2) << endl;  // 0
    // will be implicitely converted and
    // order does not matter:
    cout << ({1, 2} > p2) << endl;

    // may be compared against 0!
    //   (though it is not a number!)
    cout << (p1 <=> p2 < 0) << endl;  // 1
    cout << (p1 <=> p2 == 0) << endl;  // 0
    cout << (p1 <=> p2 > 0) << endl;  // 0
}
```

Spaceship Operator – 3

```cpp
#include <iostream>  // spaceship_operator2.cpp
using namespace std;

struct Pair {
    int a{1};  int b{2};
    // also, generates operators for: ==, !=
    auto operator<=>(const Pair&) const=default; };

int main() {
    Pair p1;  Pair p2{.b=3};
    cout << (p1 == p2) << endl;  // 0
    cout << (p1 != p2) << endl;  // 1
    cout << (p1 < p2) << endl;  // 1
    cout << (p1 <= p2) << endl;  // 1
    cout << (p1 > p2) << endl;  // 0
    cout << (p1 >= p2) << endl;  // 0
    cout << (p1 <=> p2 < 0) << endl;  // 1
    cout << (p1 <=> p2 == 0) << endl;  // 0
    cout << (p1 <=> p2 > 0) << endl;  // 0
}
```

Spaceship Operator – 4

- Vergleichbarkeit von Werten
  - Sind alle Werte miteinander vergleichbar?
  - Bsp.: 2 Mengen stehen mittels $\subseteq$ nicht notwendigerweise in Relation!
    - d.h. es gilt u.U. weder $A \subseteq B$ noch $B \subseteq A$
    - d.h. Halbordnung… (siehe `sets.pdf`)
  - Bsp.: <= über `double` ist ebenfalls eine Halbordnung, da `NaN` mit keinem anderen Wert vergleichbar ist (auch nicht mit sich selbst!)
- Unterscheidbarkeit äquivalenter Werte
  - Sind äquivalente Werte voneinander unterscheidbar?
    - allgemein: nicht unterscheidbar, wenn: $a \equiv b \rightarrow f(a) \equiv f(b)$ (f ist Funktion, die nur die allgemein zugänglichen Attribute der Objekte heranzieht, also so etwas wie den ”Wert” des Objektes ausmacht)
  - Bsp.: case-insensitives Vergleichen zweier Strings
    - ”`abc`” $\equiv$ ”`ABc`”, sind äquivalent, aber eben nicht gleich

Spaceship Operator – 5

- Welche Arten gibt es?
  - `partial_ordering`
    - eine Halbordnung (siehe `sets.pdf`)
    - äquivalente Werte sind unterscheidbar
    - nicht vergleichbare Werte sind erlaubt
  - `weak_ordering`
    - eine Totalordnung (siehe `sets.pdf`)
    - nicht vergleichbare Werte sind nicht erlaubt
    - äquivalente Werte sind unterscheidbar
  - `strong_ordering`
    - eine starke Totalordnung (siehe `sets.pdf`)
    - nicht vergleichbare Werte sind nicht erlaubt
    - äquivalente Werte sind nicht unterscheidbar
- Was wird bei `auto` zurückgeliefert?
  - etwas, das einem der drei Arten entspricht…

Spaceship Operator – 6

- für ein besseres Verständnis, die Werte der einzelnen Arten…
  - `partial_ordering`
    - `std::partial_ordering::less`
    - `std::partial_ordering::greater`
    - `std::partial_ordering::equivalent`
    - `std::partial_ordering::unordered`
  - `weak_ordering`
    - `std::weak_ordering::less`
    - `std::weak_ordering::greater`
    - `std::weak_ordering::equivalent`
  - `strong_ordering`
    - `std::strong_ordering::less`
    - `std::strong_ordering::greater`
    - `std::strong_ordering::equivalent`, gleich wie `equal`
    - `std::strong_ordering::equal`, gleich wie `equivalent`

Spaceship Operator – 7

```cpp
#include <iostream> // spaceship_operator3.cpp
#include <unordered_set>
#include <algorithm>
using namespace std;
struct Set {
  unordered_set<int> v;
  partial_ordering operator<=>(const Set& o) const {
      if (v == o.v)
          return partial_ordering::equivalent;
      else if (includes(v.begin(), v.end(),
                  o.v.begin(), o.v.end()))
          return partial_ordering::less;
      else if (includes(o.v.begin(), o.v.end(),
                  v.begin(), v.end()))
          return partial_ordering::greater;
      return partial_ordering::unordered;
  }
  bool operator==(const Set& o) const {
      return (*this <=> o) == 0;
  } };
```

Spaceship Operator – 8

```
int main() {
    Set s1{{1, 2, 3}};
    Set s2{{2, 3, 4}};
    cout << (s1 == s2) << endl;  // 0
    cout << (s1 != s2) << endl;  // 1
    cout << (s1 < s2) << endl;  // 0
    cout << (s1 <= s2) << endl;  // 0
    cout << (s1 > s2) << endl;  // 0
    cout << (s1 >= s2) << endl;  // 0
}
```

Spaceship Operator – 9

```cpp
#include <iostream>  // spaceship_operator4.cpp
#include <algorithm>
using namespace std;
struct CaseInsensitiveString;
using CIStr = CaseInsensitiveString;

struct CaseInsensitiveString {
  string v;
  weak_ordering operator<=>(const CIStr& other) const {
    string s1{v};  string s2{other.v};
    transform(s1.begin(),s1.end(),s1.begin(),::tolower);
    transform(s2.begin(),s2.end(),s2.begin(),::tolower);
    return s1 <=> s2;
  }
  bool operator==(const CIStr& other) const {
    string s1{v};  string s2{other.v};
    transform(s1.begin(),s1.end(),s1.begin(),::tolower);
    transform(s2.begin(),s2.end(),s2.begin(),::tolower);
    return s1 == s2;
  } };
```

Spaceship Operator – 10

```
int main() {
    CIStr s1{"abc"};
    CIStr s2{"aBC"};
    cout << (s1 == s2) << endl;  // 1
    cout << (s1 != s2) << endl;  // 0
    cout << (s1 < s2) << endl;  // 0
    cout << (s1 <= s2) << endl;  // 1
    cout << (s1 > s2) << endl;  // 0
    cout << (s1 >= s2) << endl;  // 1
}
```

Spaceship Operator – 11

```cpp
#include <iostream>  // spaceship_operator5.cpp
using namespace std;
struct Int {
    int v;
    Int(int v=0) : v{v} {}
    strong_ordering operator<=>(const Int& other) const
        return v <=> other.v; }
    bool operator==(const Int& other) const {
        return (v == other.v); } };
int main() {
    Int i1{42};  int i2{7};
    cout << (i1 == i2) << endl;  // 0
    cout << (i1 != i2) << endl;  // 1
    cout << (i1 < i2) << endl;  // 0
    cout << (i1 <= i2) << endl;  // 0
    cout << (i1 > i2) << endl;  // 1
    cout << (i1 >= i2) << endl;  // 1
    // order does not matter anymore:
    cout << (7 < i2) << endl;  // 0
}
```

auto *außerhalb* von Lambda

```cpp
#include <iostream>  // auto_wo_lambda.cpp
#include <vector>
using namespace std;

// like a function template for arbitrary type
void print_coll(const auto& coll) {
    for (const auto& elem : coll) {
        std::cout << elem << '\n';
    }
}

int main() {
    vector<int> v{1,2,3};
    print_coll(v);
    int a[]{1,2,3};
    print_coll(a);
    string s{”123”};
    print_coll(s);
}
```

lambda mit Template-Parameter

```cpp
#include <iostream>  // lambda_template_parameter.cpp
using namespace std;
int main() {
    // arbitrary types but must be identical
    auto f = []<typename T>(T x, T y){
        return x + y;
    };
    cout << f(1, 2) << ' ';
    cout << f(3.5, 2.5) << ' ';
    // cout << f(3.5, 2) << ' '; //no match for call...
    cout << f(string{"a"}, string{"bc"}) << endl;
}
```

Nontype Template Parameter

```cpp
#include <iostream>  // nontype_template_parameter.cpp
using namespace std;

// has to be a 'structural type'
struct X {
  X()=default;
  constexpr X(int i) : i{i} {}
  int i{};  // no private, no mutable
};

template <X x>
auto get_X() {
    return x;
}

int main() {
    X x;
    cout << get_X<X{123}>().i << endl;  // -> 123
    cout << get_X<1>().i << endl; //implicit conversion
}
```

constinit und consteval

```cpp
#include <iostream>  // constinit_consteval.cpp
using namespace std;

// will be evaluated at compile time!
consteval int calc_area(double a) { return a * a; }

// will be initialized at compile time!
//   *must* have static storage duration
//          *or* thread storage duration
constinit double area{calc_area(3)};

int main() {
    cout << calc_area(10) << endl;
    cout << area << endl;
    // thread local vars have thread storage duration
    constinit thread_local double area2{calc_area(3)};
    area = 42; // may be altered... if not desired then
    cout << area << endl; // add 'const' to definition
}
```

span

- nicht besitzende Sicht auf eine Sequenz von Ojekten
- änderbar (mutable)!
- static extent vs dynamic extent
  - static: Anzahl der Elemente bekannt
  - dynamic: Anzahl der Elemente eben nicht bekannt
- d.h. besteht intern aus:
  - Pointer
  - Länge
    - wenn static extent, dann nicht notwendig, da die Länge in Typ kodiert werden kann

span - 2

```cpp
#include <iostream>  // span.cpp
#include <vector>
#include <array>
#include <span>

void print_content(std::span<int> container) {
    for(const auto &e : container) {
        std::cout << e << ' ';  }
    std::cout << '\n';
}
int main() {
    int arr[]{1, 2, 3, 4, 5};
    print_content(arr);  // 1 2 3 4 5
    std::vector v{1, 2, 3, 4, 5};
    print_content(v); //1 2 3 4 5
    std::array arr2{1, 2, 3, 4, 5};
    print_content({begin(arr2) + 1, end(arr2) - 2});
      // 2 3
}
```

starts_with und ends_with

```cpp
#include <iostream>  // string.cpp

using namespace std;
int main() {
    string s{"https://www.htlwrn.ac.at"};
    cout << s.starts_with("https") << endl;  // 1
    string s2{"https"};
    cout << s.starts_with(s2) << endl;  // 1
    string_view sv{".at"};
    cout << s.ends_with(sv) << endl;  // 1
    cout << sv.ends_with('t') << endl;  // 1
}
```

using enum in switch

```cpp
#include <iostream>  // enum_namespace.cpp
using namespace std;
enum class Permission {
    read, write, execute
};

int main() {
    Permission perm{Permission::write};
    switch (perm) {
        // sadly, currently not with g++!!
        using enum Permission;
        case read:
            cout << "read" << endl;  break;
        case write:
            cout << "write" << endl; break;
        case execute:
            cout << "execute" << endl;  break;
    }
}
```

Formatierte Ein- und Ausgabe

```cpp
#include <iostream>  // format.cpp
#include <chrono>
#include <vector>
using namespace std;
using namespace std::literals;
#define FMT_HEADER_ONLY // use lib 'fmt' in header-only
#include <fmt/format.h>// later: <format>
#include <fmt/chrono.h>// formatting chrono...
#include <fmt/ranges.h>// formatting vector and the like

int main() {  // later: namespace std!
    cout << fmt::format("Hello {}!", "World") << endl;
    cout << fmt::format("{1} than {0}", "two", "one")
      << endl;  // one than two
    fmt::print("chrono literals: {} {}\n", 42s, 100ms);
      // chrono literals: 42s 100ms
    fmt::print("strftime-format: {:%H:%M:%S}\n",
      3h+15min+30s); // strftime-format: 03:15:30
    fmt::print("{}\n", vector<int>{1,2,3}); // {1, 2, 3}
}
```

Konzepte

```cpp
#include <iostream>  // concepts.cpp
#include <vector>
using namespace std;
template <typename T>
concept IsContainer = requires(const T& t) {
    { t.begin() }; // better: use free function begin()
    { t.end() };  };
// like a function template for arbitrary type
void print_coll(const IsContainer auto& coll) {
    for (const auto& elem : coll) {
        std::cout << elem << '\n';
    } }
int main() {
    vector<int> v{1,2,3};  print_coll(v);
//    int a[]{1,2,3};  print_coll(a);
//      neither begin() nor end() as member!
//      will work if you're using free functions
    string s{”123”};  print_coll(s);
}
```

Ranges

```cpp
#include <iostream>  // ranges.cpp
#include <vector>
#include <ranges>
using namespace std;

auto square = [](int val) { return val * val; };
auto is_over2 = [](int val) { return val > 2; };

void print_over2(ranges::range auto r) {
    for (int i : r | ranges::views::transform(square)
                   | ranges::views::filter(is_over2)) {
        cout << "square over 2: " << i << endl;
    }
}

int main() {
    vector<int> v{1,2,3};
    print_over2(v);
}
```

date-Erweiterungen

```cpp
#include <chrono>  // date.cpp
#include <iostream>

using namespace std;
using namespace std::chrono;
using namespace std::literals;

// later on, not needed anymore:
#include "date.h"
using namespace date;

int main() {
    auto today = floor<days>(system_clock::now());
    cout << today << '\n';
    constexpr auto date = 2016_y/May/29;
    //later on: constexpr auto date = 2016y/May/29;
    cout << date << endl;
}
```

Source location

```cpp
#include <iostream>
#include <experimental/source_location>

using namespace std;
using namespace std::experimental;
using src_loc = source_location;

void log(string message,
  const src_loc& loc=src_loc::current()) {
    cout << "info:" << loc.file_name() << ':'
         << loc.line() << ' ' << message
         << " ... in " << loc.function_name() << "\n";
}

int main() {
   cout << src_loc::current().line() << '\n';  // -> 16
   log("Hello world!");
   // -> info:src_loc.cpp:17 Hello world! ... in main
}
```

Endianess

```cpp
#include <iostream>
#include <string_view>
#include <bit>

using namespace std;

int main() {
    // if constexpr ... evaluate to compile time
    if constexpr (endian::native == endian::big) {
        cout << "big-endian" << '\n';
    } else { // otherwise endian::little
        cout << "little-endian"  << '\n';
        // -> little-endian
    }
}
```