Stoff der C++-Klausur

1) `for`-Schleifen mit Index:
   Python `range(-3, 71, 5):` **`for (int i{-3}; i < 71; i += 5)`** …

2) Range-`for` Schleifen
   Python `for i in X:` **`for (auto i : X)`**…; **`for (auto& i : X)`**…

3) `if-else :`

```
if (i % 2 == 0)
     …
else
     …
```

4) einfache Funktionen schreiben

```
int f(int a, int b)
{    return 2*a - b; }
```

5) modifizierende Funktionen schreiben

```
void begrenze(int& a)
{    a = max(a, 5);      }     // maximaler Wert soll 5 sein
```

6) einfache (und generische) Lambdas

```
auto lambda         = [](int  x){ return x*x; };
auto generic_lambda = [](auto x){ return x*x; };
```

7) modifizierende Lambdas

```
auto modifying_lambda = [](int&  x){ x = 17; };
```

8) Lambdas mit Capture:

```
x = 7;
auto kleiner_als_x = [x](auto i){ return i < x; };
kleiner_als_x(7); // ergibt false
```

9) modifizierende Lambdas mit Capture:

```
int x{7}, i{3};

add_x = [x](auto& i){ i += x; };
add_x(i); // i hat danach den Wert 10
```

10) Standard-Eingabe-Schleifen von `cin:` *blabla* ist irgendein Typ

```
vector<blabla> v;
for (blabla x; cin >> x) // solange etwas gelesen werden konnte
     v.push_back(x);
```

11) Einfaches `struct` mit Stream-Eingabe, Stream-Ausgabe, z.B.

```
struct Name_mit_Wert {
     string name;
     int wert;

     friend ostream& operator<<(ostream& os,
           const Name_mit_Wert& x)
     {   os << "Name: " << x.name << ", Wert = "
               << x.wert;
          return os;
     }

     friend istream& operator<<(istream& is,
           Name_mit_Wert& x)
     {   is >> x.name >> x.wert;
          return is;
     }
     …
```

12) `vector<…>` und `string` als Container verwenden
Methoden `push_back()`, `operator[]` (Index-Zugriff)
Funktionen `size()`, `begin()`, `end()`

13) Algorithmen: Verwendung folgender Algorithmen - mir ist egal, ob im Namespace `std::` (klassische Algorithmen) oder Namespace `std::ranges` (seit C++20): Letztere können auch ohne begin/end aufgerufen werden und wirken auf den ganzen Container!

Folgende Algorithmen können bei der Klausur auftauchen:

```
for_each(), count(), cout_if(),
min_element(), max_element(), sort()
accumulate() (aus <numeric>)
```

```
vector<Name_mit_wert> V{….};
string name{"Reinhard Stix"};
auto print = [](auto x){ cout << x << '\n'; };
```

for_each: Eine Funktion/Lambda auf jedes Element anwenden

```
std::for_each(begin(v), end(v), print);
std::ranges::for_each(begin(v), end(v), print);
std::ranges::for_each(v, print);
std::for_each(rbegin(v), rend(v), print);    // umgekehrt
```

count: Zählen wie oft ein Wert drin vorkommt

```
std::count(begin(s), end(s), 'a');      // ergibt 1
std::ranges::count(begin(s), end(s), 'a');        // ergibt 1
std::ranges::count(s, 'a');        // ergibt 1
```

count_if: Zählen wie oft eine Bedingung wahr ist: wie oft ist der Name Stix

```
std::count_if(begin(V), end(V),
     [](const auto& x){ return x.name == "Stix";});
std::ranges::count_if(begin(V), end(V),
     [](const auto& x){ return x.name == "Stix";});
std::ranges::count_if(V,
     [](const auto& x){ return x.name == "Stix";});
```

mit Ranges geht hier auch `count`, wenn man eine Projektion einsetzt:

```
std::ranges::count(s, "Stix", &Wert_mit_Name::name);
```

min_element, max_element: Iterator auf Extrema zurückgeben
man kann die Ordnungsrelation hier optional angeben (wenn nicht, wird < genommen)!
Die Differenz vom Anfang ergibt den Index (Position)

"kleinsten" Buchstaben im String:

```
auto it{std::min_element(begin(s), end(s))};
*it              // der "minimale" Wert
it - begin(s)    // Position des Minimums

auto it{std::ranges::min_element(s)};
```

kleinsten Wert in V ermitteln:

```
it = std::min_element(begin(V), end(V),  [](const auto& x,
     const auto& y){ return x.wert < y.wert; });
```

```
it = std::ranges::min_element(V, {}, &Name_mit_Wert::wert);
                                     ↑              ↑
                                Default Vergleich <   Projektion
```

sort: Sortieren eines Bereichs. Die Default-Reihenfolge ist <

```
std::sort(begin(s), end(s));  // wachsend nach Ascii-Werte sortieren
std::ranges::sort(begin(s), end(s));
std::ranges::sort(s);
```

Will man anders sortieren, muss man die Sortierreihenfolge angeben. Dazu übergibt man ein Prädikat (Funktion/Lambda mit Ergebnis `bool`, das angibt, ob 2 Elemente `x, y` richtig angeordnet sind.

V aufsteigend nach Werten sortieren:

```
std::sort(begin(V), end(V), [](const auto& x, const auto& y)
     { return x.wert < y.wert; }
std::ranges::sort(V, {}, &Name_mit_Wert::wert);
                         ↑              ↑
                  Default Vergleich <   Projektion
```

absteigende Sortierung erreicht man entweder über ein selbstgeschriebenes Lambda oder über vordefinierte Vergleichsoperationen aus `<functional>`:

```
<:    std::less<>{}      std::ranges::less{}
>:    std::greater<>{}   std::ranges::greater{}
```

accumulate: Akkumulieren eines Bereichs. Die Default-Aktion ist + (Addition aller Werte), eine andere Methode kann als 4. Argument übergeben werden. Dieser Algorithmus kommt erst 2023 in einer Ranges-Version:

```
std::accumulate(begin(s), end(s), 0);   Ascii-Werte aufsummieren
std::accumulate(begin(V), end(V), 0,    // alle Werte addieren
     [](auto sum, const auto& x){ return sum + x.wert;});
```

```
std::accumulate(begin(V), end(V), 0,    // alle geraden Werte  addieren
     [](auto sum, const auto& x){
          if (x.wert % 2 == 0)          // diesen Wert addieren
               return sum + x.wert;
          else                          // diesen Wert ignorieren
               return sum;});
```