Kommentierte Code-Beispiele

Primzahl-Finder mit Funktionen und dem `std::vector` Container

(primes_without_test.cpp)

```
#include <iostream>  // fuer Ausgabe
#include <vector> // fuer std::vector
// Funktion
bool // Ergebnistyp
is_prime( // Funktionsname
          int candidate // Argument mit Name candidate vom Typ int
        ) {
    bool is_prime = true; // Variable namens is_prime vom Typ bool
                          // mit Startwert true
    // Schleife ueber test (Typ int, Startwert 2),
    // solange wie test < candidate, test jeweils um eins erhoehen
    for (int test=2; test<candidate; test++) {
        if (candidate % test == 0) { // candidate durch test teilbar? (Rest 0)
            return false; // Funktion mit false als Antwort verlassen
        }
    }
    return true; // wenn wir noch da sind, true zurueckgeben
}

// Funktion, Rueckgabe-Typ ist std::vector<int> (Vektor von Ganzzahlen)
std::vector<int> get_primes_below(int limit) { // Argument limit, Typ int
    std::vector<int> result; // Leeren Vektror anlegen, der ints speichert
    for (int candidate = 2; candidate<limit; candidate++) { // For-Schleife
        if (is_prime(candidate)) { // Funktion is_prime mit candidate aufrufen
            // Wenn die Funktion true gesagt hat
            result.push_back(candidate); // candidate an Vektor result anhaengen
        }
    }
    return result; // den Vektor result zurueckgeben
}

int main() {
while (true) { // endlos-Schleife
    int limit; // Variable limit vom Typ int deklarieren
    std::cout << "Limit (0 to stop): "; // Ausgabe
    std::cin >> limit; // Eingabe >> Variable limit
    if (limit == 0) {
        break; // abbrechen
    }

    // Die Funktion get_primes_below() mit limit aufrufen,
    // Ergebnis in primes speichern
    // Datentyp entspricht den Rueckgabe-Typ der get_primes_below()-Funktion
    auto primes = get_primes_below(limit);
    for (auto p : primes) { // fuer ale elemente p aus dem Vektor primes
      std::cout <<p << " "; // Inhalt von p und ein Leerzeichen ausgeben
    }
    std::cout << std::endl; // neue Zeile
}
}
```

Strings in einen Vektor einlesen und sortieren

(sort_strings.py/cpp)

Python

```python
def read_strings():
    v = []  # leere Liste
    while True:  # Endlos-Schleife
        s = input("Wort, Ende mit quit: ")
        if s == "quit":
            break  # abbrechen
        v.append(s)  # Eingabe an Liste anhaengen
    return v


v = read_strings()  # Ergebnis der Funktion read_strings landet in v
v = sorted(v)  # sortieren
print("Sorted:")
for s in v:  # fuer alle Strings s aus der Liste v
    print(s)
```

C++

```cpp
#include <algorithm> // Für std::sort
#include <iostream>  // für std::cin, std::cout, std::endl
#include <string>    // für Zeichenketten std::string
#include <vector>    // für std::vector

// Gibt eine vectore der eingegebenen Zeichenketten zurück
std::vector<std::string> read_strings() {
  std::cout << "Wörter eingeben. Ende mit quit" << std::endl;
  std::vector<std::string> v; // Leerer vector, der std::strings speichert
  while (true) {              // Endlos-Schleife
    std::string input;        // Ziechenkette fuer die Eingabe
    std::cin >> input;        // Zeichenkette einlesen
    if (input == "quit") {
      break;
    };                  // abbrechen bei leerer Zeichenkette
    v.push_back(input); // und an den vector anhängen
  }
  return v; // den Vektor zurückgeben
}

int main() {
  auto v = read_strings(); // Ergbenis der Funktion read_strings() landetin v

  std::cout << "Sortiert:" << std::endl;
  std::sort(v.begin(), v.end()); // Sortiert l
  for (auto s : v) {             // Schleife über alle Strings s im Vector v
    std::cout << s << std::endl; // Ausgeben
  }
}
```

Zahlen in einem Vektor quadrieren mit std::transform

(square_elements.cpp)

```cpp
#include <algorithm>
#include <iostream>
#include <vector>
// Funktion, die die Elemente des Containers anzeigt
// Rückgabetyp ist void (nichts)
// Ein Argument vom Typ const std::vector<double>&
// cosnt: wir werden den Inhalt nicht ändern.
// &: nicht kopieren, sondern einen Verweis nutzen
// Man nennt die Kombination const-Referenz
void print_container(const std::vector<double>& container) {
  for (const auto &v : container) { // fuer alle Elemente v aus container
    std::cout << v << " "; // Ausgeben
  }
  std::cout << std::endl; // neue Zeile
}

// Rueckgabetyp: double. Argument-Typ: const double&
// Wieder const-Referenz. Nicht aenderbar und ein Verweis statt einer Kopie
double square(const double &v) { return v * v; }; // Gibt v*v zurueck

int main() {
  std::vector<double> v = {1, 2.5, -3, 4, 5}; // Vektor mit den gegebenen Werten
  print_container(v); // Ruft print_container auf

  // Transformation. Eingabe: von v.begin() bis (nicht einschliesslich) v.end()
  // Start der Ausgabe: v.begin(), also ziel=quelle -> ueberschrieben der Werte
  std::transform(v.begin(), v.end(), // Eingabe
                 v.begin(), // start der Ausgabe
                 square); // Unary Operation (Funktion mit einem Argument
  print_container(v); // ruft print_container mit v auf.
}
```

Algorithmen mit Lambda-Ausdrücken

Demonstiert std::for_each, std::count_if und std::accumulate (algo_demo_lambda.cpp)

```
#include <algorithm> // fuer std::for_each, std::count_if
#include <iostream> // fuer Ein- und Ausgabe
#include <numeric> // fuer std::accumulate
#include <vector> // fuer std::vector
// Funktion zeigt alle Elemente aus einem Container an
// Die Wahl des Containers ist beliebig, er muss nur begin() un end() liefern
// und std::cout muss die Elemente des Containers verstehen
// (double, ind, std::string, etc.)
template <typename T> void print_container(T &container) {
  std::for_each(container.begin(), container.end(), // fuer alle Elemente
                [](auto v) // Lambda-Kopf mit einem Argument v
                  { std::cout << v << " "; }); // v ausgeben
  std::cout << std::endl; // neue Zeile
}

int main() {
  std::vector<int> v = {1, 3, 2, 5, 4}; // Vektor von ein paar Ganzzahlen
  print_container(v); // Aufruf von print_container().
                      // T ist der Datentyp von v, std::vector<double>
  std::cout << "Gerade Werte in v: "
            << std::count_if(v.begin(), v.end(), // Elemente von v zählen
                             [](int i) // Lambda-Kopf, ein Argument vom Typ int
                             { return i % 2 == 0; }) // i durch zwei teilbar?
            << std::endl; // neue Zeile
  std::cout << "Summe von v: "
            << std::accumulate( // accumulate zaehlt zusammen
                v.begin(), v.end(), 0) // Startwert ist 0
            << std::endl; // neue Zeile
  // Man kann bei accumulate statt plus eine andere Operation reinstecken
  std::cout << "Produkt der Werte in v: " // z.B. multiplikation
            << std::accumulate(v.begin(), v.end(), 1, // startwert ist 1
                   [](auto x, auto y) // lambda-Kopf, zwei Argumente, x und y
                   { return x * y; }) // x*y zurueckgeben
            << std::endl; // neue Zeile
}
```

Count_if mit Lambda und eingefangener Variable

Zählt Werte kleiner als ein gegebenes Limit (count_values_smaller_than.cpp)

```cpp
#include <algorithm>
#include <iostream>
#include <vector>
// Funktions-Template nimmt einen Vektor mit Elementen vom Typ T
// und zaehlt, wie viele kleiner als das limit (ebenfalls typ T) sind
// T Kann sein: int, float, double, ..., alles was man mit < vergleichen kann
template <typename T> // Template-Parameter: Der Datentyp T
int // Rueckgabe-Typ (ganzzahl)
values_less_than(std::vector<T> v, // 1. Argument: Vektor mit Element-Typ T
                 T &limit) { // 2. Argument. Ein Wert vom Typ T
  return std::count_if(v.begin(), v.end(), // Zaehle im Vektor v
               [limit](auto x) // Lambda-Kopf. Variable limit einfangen,
                               // ein Argument, x. Datentyp wird abgeleitet
               { return x < limit; }); // sagt true wenn x < limit
}

int main() {
  std::vector<double> v = {-1, 0, 0.1, 1, -2.5, 2}; // Vektor mit ein paar Zahlen
  for (double l : {0, 1}) { // Fuer die Fliesskommazahlen 0 und 1
    std::cout << "Values in v less than " << l << ": " // Ausgabe
    << values_less_than(v, l) // Ruft values_less_than() mit v und l auf
              << std::endl; // neue Zeile
  }
}
```

Studierende nach Punkten in der Prüfung sortieren

```cpp
#include <iostream>
#include <vector>
#include <algorithm> // fuer std::sort

class Student { // deklaration Klasse namens Student
  public: // die folgenden Variablen und Funktioenn sind oeffentlich nutzbar
    Student(std::string s_name, int s_score) { // Constructor
      name = s_name; // setzt die Variablen der Klasse
      exam_score = s_score;
    };
    std::string name; // Zeichenkette namens name
    int exam_score;  // Ganzzahl namens exam_score
};

bool compare_scores(Student s1, Student s2) {
  return s1.exam_score < s2.exam_score; // Hat s1 weniger Punkte als s2?
}
int main() {
  std::vector<Student> students; // Vektor mit Studierenden
  // Ein par Studierende hinzufuegen
  students.push_back(Student("name name", 90));
  students.push_back(Student("other name", 80));
  students.push_back(Student("third name", 85));

  // Sortieren mit eigneem vergleichskriterium
  std::sort(students.begin(), students.end(),
      compare_scores); // Die Funktion compare_scores ist das Kriterium
  // Ausgabe
  for (auto s: students) { // fuer alle Studierenden s aus students
    std::cout << s.name <<" "<< s.exam_score<<std::endl;
  }
}
```