Container und Algorithmen

Die Algorithmen der Standardbibliothek

- Erledigen häufige Aufgaben, oft auf Containern
- Statt einem genauen Rezept schreibt man das gewünschte Ergebnis auf
- (vgl. List Comprehension in Python)

Beispiele

- Zähle Elemente, die durch 2 teilbar sind: *count_if*
- Hänge die Elemente aus einem Container an einen anderen an: *copy*
- Sortieren (mit wählbarem Kriterium): *sort*
- Finde den Kandidaten/die Kandidatin mit der höchsten Punktzahl in der Klausur: *max_element*
- Enthalten zwei Container die selben Elemente, aber in anderer Reihenfolge? *is_permutation*
- u.v.m.

Wichtige Zutaten: Iteratoren

- Eingabe-Iterator (engl: input iterator)
  - liefert Werte (etwa aus einem Container)
  - weis, welcher Wert gerade dran ist (etwa: Position im Container)
  - weiß, wie man zum nächsten Wert weiterschaltet
  - Retro-Vergleich: Lesen und Vorspulen beim Tonband
- Ausgabe-Iterator (engl: output iterator)
  - schreibt Werte (etwa in einen Container)
  - weiß, wo gerade hingeschrieben wird (Position im Container)
  - weiß, wie man zur nächsten Schreibposition weiterschaltet
- Kann es Iteratoren geben, die nichts mit einem Container zu tun haben?

Iteratoren und Container

- Container stellen i.d.R. passende Iteratoren bereit:
  - `begin()`: Zeigt auf den Anfang und kann von dort vorwärts gehen
  - `end()`: Zeigt *hinter* das letzte Element. Dort kann man keinen Wert mehr lesen
- Iteratoren von Hand weiterschalten

```
std::vector<double> v = {1,2,3,4,5};
auto it = v.begin();
```

```
v++; // eins weiter
v += 2; // zwei Schritte weiter
```

- Iteratoren nach dem aktuellen Wert fragen
  - Mit dem *-Operator: de-referenzieren

    ```
    std::vector<double> v = {1,2,3,4,5};
    auto it = v.begin();
    std::cout << *it<<std::endl;
    ```

  - Was wird ausgegeben?
- Iteratoren vergleichen

  ```
  std::vector<double> v = {1,2,3,4,5};
  auto it = v.begin();
  while (it != v.end()) { // so lange man nicht hinter dem letzten Element ist
      // ... Etwas machen
      it++; // Weiterschalten
  }
  ```

Iteratoren und Container: Beispiel

Jedes zweite Element aus einem Container anzeigen

```
#include <iostream>
#include <vector>

int main() {
  std::vector<double> v = {1, 2, 3, 4, 5, 6};
  auto it = v.begin();
  while (it < v.end()) {
    auto value = *it;                 // Wert auslesen
    std::cout << value << std::endl; // Ausgabe
    it += 2;                          // Weiterschalten
  }
}
```

Wichtige Zutaten: Prädikate und Operatoren

- Filterkriterien (Prädikat, Engl: predicate)
  - Algorithmen wie *count_if* benötigen ein "Filterkriterium".
  - Funktion, die einen Wert nimmt und `true` oder `false` sagt

  ```
  bool is_even(int v) { return v % 2 == 0; };
  ```
- Operatoren
  - Der *transform*-Algorithmus wandelt Werte um

  - z.B. ersetze `wert` durch `wert * wert`
  - Die Transformation wird von einem Operator durchgeführt.
    `double square(double x) { return x*x; };`
  - *Unary* Operator: Ein Argument (wie im Beispiel)
  - *Binary* Operator: zwei Argumente (z.B. Vergleichsregel beim Sortieren)

Algorithmen in Aktion

```
#include <algorithm>
#include <iostream>
#include <vector>

template <typename T>
void print_container(const T & container) {
  for (const auto &v : container) {
    std::cout << v << " ";
  }
  std::cout << std::endl;
}

bool is_even(int x) {
  return x % 2 == 0;
}

int main() {
  std::vector<int> v = {1, 2, 3, 4, 5};
  std::vector<std::string> l = {"hallo", "welt", "hello", "world"};
  print_container(v);
  std::cout << "Gerade Werte in v: "
            << std::count_if(v.begin(), v.end(), is_even) << std::endl;

  std::cout << "Original: ";
  print_container(l);
  std::sort(l.begin(), l.end());
  std::cout << "Sorted: ";
  print_container(l);
}
```

Lesen in der C++-Referenz

- Einge gute C++-Referenz findet sich unter http://cppreference.com
- ist meistens der erste Treffer bei Google für `std::sort`, etc.

- Die deutsche Version ist maschinell übersetzt. Eher nicht benutzen.
- Lernen, die Referenz zu nutzen, ohne dass man alles versteht, was dort steht

Lese-Hinweise

- *C++20* ist noch nicht fertig. Sachen, die erst ab C++20 gelten, also vorerst ignorieren
- *Execution policy* erstmal ignorieren (es geht um gleichzeitige Ausführung)
- Die Varianten mit weniger Argumenten zuerst beachten
- Die Anforderungen sind oft sehr formell geschrieben. Nicht verrückt machen lassen.
- Lest das Code-Beispiel

Aufgabe:

Alle Elemente des Vektors `std::vector<double> v` sollen quadriert werden. Beantwortet mit Hilfe der Dokumentation von `std::transform`:

- Welcher Header muss eingebunden werden? `#include ...`
- Welche Variante ist die passende?
- Kann das Ergebnis an Ort und Stelle gespeichert werden, oder brauchen wir einen neuen Vektor?
- Wie sieht die Signatur der Quadrier-Funktion aus? (Signatur: Rückgabe-Typ und Typ und Anzahl der Argumente der Funktion)
- Schreibt den Code für die Quadrierfunktion und den Aufruf von `std::transform` hin

Lösung: Elemente quadrieren

```
#include <algorithm>
#include <iostream>
#include <vector>

template <typename T> void print_container(const T & container) {
  std::for_each(container.begin(), container.end(),
    [](auto v) { std::cout << v << " "; });
  std::cout << std::endl;
}

double square(const double & v) {
  return v * v;
};

int main() {
```

```
  std::vector<double> v = {1, 2.5, -3, 4, 5};
  print_container(v);
  std::transform(v.begin(), v.end(), v.begin(), square);
  print_container(v);
}
```

Lambdas

Lambdas: (mehr als) Funktionen

- Einfachste Form: wie eine Funktion ohne Namen, die am Ort der Nutzung definiert werden kann
  - Python

  ```
  lambda x,y : x + y
  ```

  - C++ (mit Angabe der Datentypen):

  ```
  [](double x, double y) -> double { return x + y; }
  ```

  - Der Rückgabewert kann fast immer ausgelassen werden (er ergibt sich aus dem Typ des Ausdrucks hinter `return`)
  - Oft können auch die Datentypen der Argumente automatisch bestimmt werden (generic lambda)

    ```
    [](auto x, auto y) { return x + y; }
    ```

- Lambdas in einer Variable speichern und später aufrufen

  ```
  auto addieren = [](auto x, y) { return x + y; };
  std::cout << addieren(2,3) << std::endl;
  ```

- Wichtigste Verwendung: Algorithmen der Standard-Bibliothek

  ```
  // Gerade Zahlen zählen
  std::count_if(v.begin(), v.end(),
      [](int i) { return i % 2 ==0; });
  ```

Beispiele

```
#include <algorithm> // für count_if
#include <numeric> // fuer accumulate
#include <iostream> // für cout, endl
#include <vector> // für vector
```

```cpp
template <typename T> void print_container(const T & container) {
  std::for_each(container.begin(), container.end(),
                [](auto v) { std::cout << v << " "; });
  std::cout << std::endl;
}

int main() {
  std::vector<int> v = {1, 3, 2, 5, 4};
  std::vector<std::string> l = {"hallo", "welt", "hello", "world"};
  print_container(v);
  std::cout << "Gerade Werte in v: "
            << std::count_if(v.begin(), v.end(),
                             [](int i) { return i % 2 == 0; })
            << std::endl;
  std::cout << "Summe von v: " << std::accumulate(v.begin(), v.end(), 0)
            << std::endl;
  std::cout << "Produkt der Werte in v: "
            << std::accumulate(v.begin(), v.end(), 1,
                               [](auto x, auto y) { return x * y; })
            << std::endl;
}
```

Lambdas können ihre Umgebung einfangen (capture)

- Eine Variable aus der Umgebung als Kopie einfangen

  ```cpp
  cint x = 1;
  auto f = [x]() { return x; };
  y = f(); // y ist dann 1
  ```

- Variable schreibbar (als Referenz) einfangen mit `[variable&]`
- Man kann mehrere Variablen einfangen (mit `,` getrennt)
- Alle variablen einfangen
  - mit `[=]` als Kopie
  - mit `[&]` als Referenz, schreibbar
- Beispiel: Funktion, die Elemente in einem Vektor zählt, die kleiner als eine gegeben Zahl sind

  ```cpp
  template <typename T>
  int values_less_than(const std::vector<T> & v, T limit) {
    return std::count_if(v.begin(), v.end(),
                         [limit](auto x) { return x < limit; });
  }
  ```

Vollständiger Code

```
#include <algorithm>
#include <iostream>
#include <vector>

template <typename T>
int values_less_than(const std::vector<T> & v, T &limit) {
  return std::count_if(v.begin(), v.end(),
                       [limit](auto x) { return x < limit; });
}

int main() {
  std::vector<double> v = {-1, 0, 0.1, 1, -2.5, 2};
  for (double l : {0, 1}) {
    std::cout << "Values in v less than " << l << ": " << values_less_than(v, l)
              << std::endl;
  }
}
```

Integrierte Entwicklungsumgebungen (Integrated Development Environments)

Integrated Development Environments (IDEs)

- vereinen alle für die Softwareentwicklung nötigen/nützlichen Werkzeuge in einer Oberfläche:
  - Texteditor
  - Dateimanager
  - Compiler, inkl. Übersicht über die ausgegebenen Warnungen/Fehlermeldungen
  - Querverweise zwischen Code in unterschiedlichen Dateien und zwischen Code und Dokumentation
  - Versionskontrolle (alte Versionen meines Codes speichern und gemeinsam mit anderen Personen am selben Code arbeiten)
  - Debugger (laufendes Programm untersuchen: was steht in den Variablen, welche Funktion hat diese Funktion aufgerufen, . . .)
- Beispiele
  - KDevelop (für Linux)
  - Microsoft Visual Studio (für Windows)
  - Apple Xcode (für macOS)
  - Jetbrains CLion (für alle drei Systeme)