Lambdas

Lambdas: (mehr als) Funktionen

- Einfachste Form: wie eine Funktion ohne Namen, die am Ort der Nutzung definiert werden kann
- Beispiel: Nimmt x und y als Argumente und gibt x+y zurück
  - Python:

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

  - C++ (mit Angabe der Datentypen):

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

  - Aufbau
    - `[]`: jetzt kommt ein Lambda
    - `(argumente)`: Welche Argumente nimmt das Lambda (wie bei funktionen)
    - `-> Rückgabetyp` (optional): welchen Datentyp hat das Ergebnis
  - 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; // Aufruf wie Funktion
```

Lambdas: Wichtigste Verwendung

Algorithmen der Standard-Bibliothek

```
// Gerade Zahlen zählen mit dem std::count_if Algorithmus
std::vector<int> v = {-2,4,17,2,8}; // Vektor v mit Ganzzahlen
int n_even_numbers = std::count_if(v.begin(), v.end(),
        [](int i) { return i % 2 ==0; }); // Lambda prüft ob i gerade ist
```

Beispiele

```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(), // für 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 zählt 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
}
```

Lambdas können ihre Umgebung einfangen (capture)

- Eine Variable aus der Umgebung als Kopie einfangen

  ```cpp
  int x = 1;
  auto f = [x]() { return x; };// x ist 1. Siehe eine Zeile höher
  ```

```
int 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

```
// Funktions-Template nimmt einen Vektor mit Elementen vom Typ T
// und zählt, wie viele kleiner als das limit (ebenfalls vom Typ T) sind
// T Kann sein: int, float, double, ..., alles was man mit < vergleichen kann
template <typename T> // Template-Parameter: Der Datentyp T
int // Rückgabe-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(), // Zähle 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
}
```

Warum immer nur int, double und std::string?

Klassen

Wie in Python, zum Anordnen von zusammenhängenden Daten und zugehörigen Methoden.

```
// reine Daten-Klasse, vorerst keine Methoden
class Student { // Deklaration Klasse namens Student
  public: // die folgenden Variablen und Funktioenn sind äffentlich nutzbar
    std::string name; // Zeichenkette namens name
    int exam_score;  // Ganzzahl namens exam_score
};

// Jetzt gibt es Student als Datentyp wie int und double
// Zwei Variablen vom Typ Student erzeugen
Student student1, student2;
student1.name = "Some Name"; // student1 ausfuellen
student1.exam_score = 42;
```

```
student2.name = "other name"; // student2 ausfuellen
student2.exam_score = 50;
```

Eigene Datentypen und Container

Container kann man auch mit eigenen Datentypen wie Student bestellen

```
class Student { // deklaration Klasse namens Student
  public: // die folgenden Variablen und Funktioenn sind oeffentlich nutzbar
    std::string name; // Zeichenkette namens name
    int exam_score;  // Ganzzahl namens exam_score
};

std::vector<Student> v;
Student s; // Variable s ist jetzt vom Typ Student
s.name = "name"; s.exam_score=32; // befüllen
v.push_back(s); // an den Vektor anhängen
```

Bessere Student-Klasse

- Wir wollen erzwingen, dass die Klasse sofort mit Daten gefüllt wird.
- So kann niemand vergessen, eines der Felder zu setzen.
- Der Konstruktor ist eine Funktion, die beim Erzeugen der Klasse aufgerufen wird.
- Der Konstruktor heißt wie die Klasse
- Der Konstruktor hat keinen Rückgabetyp
- Konstruktoren können Argumente haben wie Funktionen
- Man kann mehrere Konstruktoren mit unterschiedlichen Argument-Typen angeben
- In Python war das die `__init__()`-Funktion

```
class Student { // Deklaration Klasse namens Student
  public: // die folgenden Variablen und Funktionen sind öffentlich nutzbar
    Student(std::string s_name, int s_score) { // Konstructor
      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
};
Student s("some name", 99); // Student mit Name "some name" und 99 Punkten
```

Eigene Klassen und Algorithmen

- Container, die eigene Klassen speichern (z.B. `std::vector<Student>`), können mit den Algorithmen der Standard-Bibliothek verwendet werden
- Oft muss man sich dann aber darum kümmern, Vergleiche o.ä. zu definieren
- d.h. wonach sollen Studierende sortiert werden?
- Siehe Beispiel “Studierende nach Prüfungspunkten sortieren”