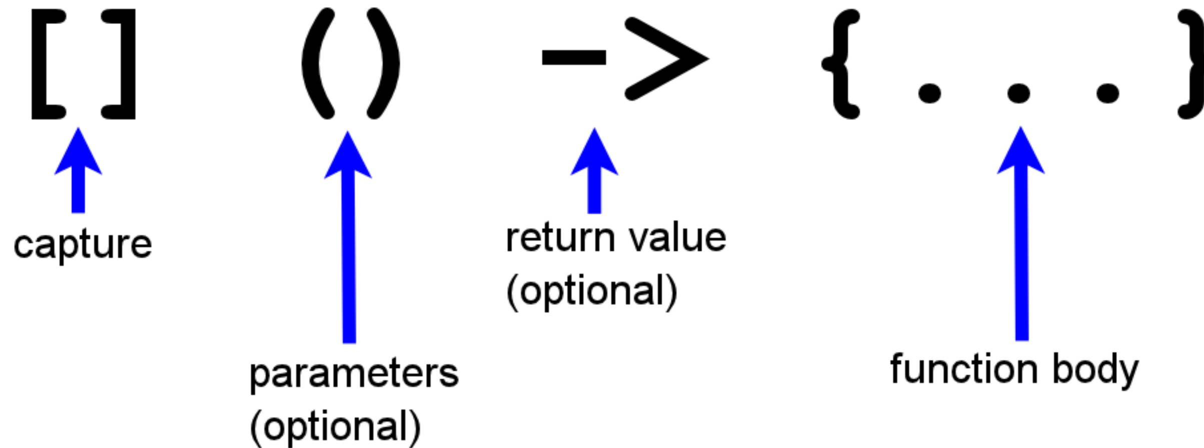
Lambda-Funktionen

- Lambda-Funktionen
  - sind Funktionen ohne Namen.
  - definieren ihre Funktionalität an Ort und Stelle.
  - können wie Daten kopiert werden.
  - können ihren Aufrufkontext speichern.
- Lambda-Funktionen sollen
  - kurz und knackig sein.
  - selbsterklärend sein.

Lambda-Funktionen: Syntax



- **[]:** Bindung der verwendeten Variablen per Copy oder Referenz möglich
- **():** Bei Parametern notwendig
- **->:** Bei komplexeren Lambda-Funktionen notwendig
- **{}:** Funktionskörper , per Default `const`

  `[]() mutable -> {...}` besitz einen nicht-konstanten Funktionskörper

Lambda-Funktionen: Closure

Lambda-Funktionen können ihren Aufrufkontext binden.

➡ Closure

Bindung	Beschreibung
`[]`	Kein Bezug
`[a]`	`a` per Kopie
`[&a]`	`a` per Referenz
`[=]`	Alle verwendeten Variablen per Kopie
`[&]`	Alle verwendeten Variablen per Referenz
`[=, &a]`	Standardmäßig per Kopie; `a` per Referenz
`[&, a]`	Standardmäßig per Referenz; a per Kopie
`[this]`	Daten und Mitglieder der umgebenden Klasse per Kopie
`[l = std::move(lock)]`	Verschiebt `lock` (C++14)

Lambda-Funktionen: First-Class-Function

Lambda-Funktionen können sowohl in Variablen gespeichert, als auch als Argument oder Rückgabewert einer Funktion verwendet werden.

➡ First-Class-Function

```
auto addTwoNumber= [](int a, int b){return a + b;};

std::thread th([]{std::cout <<  "Hello from thread" << std::endl;});

std::function<int(int, int)> makeAdd(){
  return [](int a, int b){return a + b;};
}

std::function<int(int,int)> myAdd= makeAdd();
myAdd(2000, 11);   // 2011
```

Lambda-Funktionen: Implementierung

Eine Lambda-Funktion entspricht einem Funktionsobjekt, das an Ort und Stelle implizit definiert und instanziiert wird.

```
auto add= [](int a, int b){
  return a+b;
}
```

```
class TmpAdd{
public:
  int operator()(int a, int b) const{
    return a + b;
}
TmpAdd tmpAdd;
```

```
add(2000, 11); // 2011
```

```
tmpAdd(2000, 11); // 2011
```

Generische Lambda-Funktionen: C++14

In C++14 können Lambda-Funktionen den Typ ihrer Argumente automatisch bestimmen.

```
auto add11 =[](int i, int i2){return i + i2;};
auto add14 = [](auto i, auto i2){return i + i2;};

std::vector<int> myVec{1, 2, 3, 4, 5};
auto res11 = std::accumulate(myVec.begin(), myVec.end(), 0, add11);
auto res14 = std::accumulate(myVec.begin(), myVec.end(), 0, add14);
  // res11 == res14 == 15;

std::vector<std::string> myVecStr{"Hello"s, " World"s};
auto st = std::accumulate(myVecStr.begin(), myVecStr.end(), ""s, add14);
std::cout << st << std::endl;   // Hello World
```

Lambda-Funktionen

- Beispiele:
  - `lambdaFunction.cpp`
  - `lambdaFunctionClosure.cpp`
  - `lambdaFunctionCapture.cpp`
  - `lambdaFunctionThis.cpp`
  - `lambdaFunctionGeneric.cpp`
- Aufgaben:
  - Das Programm `lambdaFunctionCapture.cpp` besitzt undefiniertes Verhalten. Korrigieren Sie das Programm.
    - Lösung: `lambdaFunctionCapture.cpp`
  - Die Regeln rund um Lambda-Funktionen werden schnell anspruchsvoll. Überzeugen Sie sich.
- Weitere Informationen:
  - [Lambda-Funktionen]()

Vereinheitlichte Initialisierung mit { }

- **Grundregel**: Eine { }-Initialisierung ist in allen Initialisierungen anwendbar.

- Zwei Formen:
  - Direkte Initialisierung

    ```
    string str{"my String"};
    ```

  - Kopierinitialisierung

    ```
    string str ={"my String"};
    ```

Vereinheitlichte Initialisierung mit { }

Die Initialisierung mit { } erlaubt keine Verengung (narrowing).

➡ Heimlicher Verlust der Datengenauigkeit.

```
int i1(3.14);                   // OK
int i2{3.14};                   // ERROR
int i3 = {3.14};                // ERROR

char c1(999);                   // OK
char c2{999};                   // ERROR

char c3{8};                     // OK
```

Initialisiererlisten für Konstruktoren

- Initialisiererlisten für Konstruktoren erlauben das einfache und direkte Initialisieren der Container der STL.

- Initialisiererlisten: `std::initializer_list<type>`
  - leichtgewichtiger Proxy um eine Array von Objekten
  - besitzen die drei Methoden `begin`, `end` und `size`
  - benötigen die Header-Datei `<initializer_list>`

- Beispiele:

```
std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9};
std::map<std::string, int> myMap{{"C++98", 1998}, {"C++11", 2011}};
for (int i: {1, 2, 3, 4, 5}) std::cout << i << " ";  // 1 2 3 4 5
```

Vereinheitlichte Initialisierung mit { }

Initialisiererlisten für Konstruktoren ermöglichen neue Anwendungsfälle.

- Container der STL

```
std::vector<int> intVec{1, 2, 3, 4, 5};
```

- Konstante Heap-Array

```
const float* p = new const float[2]{1.2, 2.1};
```

- Konstantes C-Array als Attribut einer Klasse

```
struct MyArray{
  MyArray(): data{1, 2, 3, 4, 5}{}
  const int data[5];
};
```

- Default-Initialisierung eines beliebigen Objekts

```
std::string s{};
```

- Initialisieren eines beliebigen Objekts

```
MyClass class{2011, 3.14};
```

Sequenzkonstruktor

- Ein Sequenzkonstruktor ist ein Konstruktor, der eine Initialisiererliste annimmt.
- Ein Sequenzkonstruktor
  - besitzt eine höhere Präzedenz als ein klassischer Konstruktor.
  - ist für die Container der STL definiert.
  - lässt sich für eigene Datentypen definieren.

- Sequenzkonstruktor für den `std::vector`

```
template<typename T>
class vector{
  public:
    vector(std::initializer_list<T> inList){ . . .
```

Funktionen können auch Initialisiererlisten annehmen.

Vereinheitlichte Initialisierung

- Beispiele:
  - `uniformInitialization.cpp`
  - `initializerList.cpp`

- Aufgaben:
  - Initialisieren Sie die verschiedenen Container `std::array`, `std::vector`, `std::set` und `std::unordered_set` mit der `{-10, 5, 1, 4, 5}` -Initialisiererliste.
    - Lösung: `initializerList.cpp`

- Weitere Informationen:
  - [std::initializer_list]()

Konstruktoren: Delegation

- Ein Konstruktor kann einen anderen Konstruktor der gleichen Klasse aufrufen.
- Dieser Konstruktor muss in der Klassen-Initialisiererliste spezifiziert werden.

```
struct Account{
  Account(): Account(0.0){}
  Account (double b): balance(b){}
};
```

- Regeln:
  - Sobald der erste Konstruktor fertig ist, ist das Objekt konstruiert.
  - Konstruktoren dürfen nicht direkt oder indirekt rekursiv aufgerufen werden.

  ➡ undefined behaviour

- Idee:
  - Gemeinsame Initialisierungsaufgaben können in einem Konstruktor implementiert und von allen anderen Konstruktoren verwendet werden.

Konstruktor: Delegation

- Beispiele:
  - `constructorDelegation.cpp`

- Aufgaben:
  - Schreiben Sie eine Klasse, in der sich Konstruktoren rekursive aufrufen. Was passiert?
    - Lösung: `constructorDelegationRecursive.cpp`

Konstruktoren: Vererbung

- Durch die `using`-Deklaration erbt eine Klasse alle Konstruktoren ihrer direkten Basisklasse.
- Der Default-Konstruktor, der Copy- und Move-Konstruktor wird nicht vererbt.

```
class Account{
public:
  Account(double amount){};
};
class BankAccount: public Account{
public:
  using Account::Account;
};

BankAccount bankAccount(100.0);
```

Konstruktoren: Vererbung

Die abgeleitete Klasse erbt alle Charakteristiken des Konstruktors:

- `public`, `protected` und `private` Zugriffsbeschränkungen
- `explicit` und `constexpr` Deklarationen

Default-Argumente für Parameter eines Basisklassenkonstruktors werden nicht vererbt.

➡ Die abgeleitete Klasse erhält einen zusätzlichen Konstruktor, der einen Parameter für das Default-Argument enthält.

Konstruktoren mit denselben Parametern wie die abgeleitete Klasse, werden nicht vererbt.

Das Vererben von Konstruktoren birgt die Gefahr, dass ein Attribut in der erbenden Klasse nicht initialisiert wird.

Konstruktor: Vererbung

- Beispiele:
  - `constructorInheriting.cpp`
- Aufgaben:
  - Leiten Sie `public`, `protected` und `private` von einer Klasse ab und verwenden Sie das Vererben von Konstruktoren. Die geerbte Konstruktoren der Basisklasse behalten ihre Sichtbarkeit. Die abgeleitete Klasse schränkt ihr Sichtbarkeit ein. Was bedeutet das für die Sichtbarkeit der geerbten Konstruktoren?
    - Lösung: `constructorInheriting.cpp`