

[]{}

Lambdas: A tale about brackets

Lambdas seit C++11

- Ermöglichen die Definition von inline Funktionalität.
- Können als Parameter oder als lokale Objekte genutzt werden.

```
std::vector<int> vector = {1, 5, 6, 7, 10, 4};
```

```
std::transform(vector.begin(),  
               vector.end(),  
               vector.begin(),  
               [](int value) {  
                   return value * value;  
               });
```

```
std::vector<int> vector = {1, 5, 6, 7, 10, 4};
```

```
std::transform(vector.begin(),  
               vector.end(),  
               vector.begin(),  
               [](int value) {  
                   return value * value;  
               });
```

```
[capture](params) mutable throwSpec -> ret {statements;}
```

```
[capture](params) mutable throwSpec -> ret {statements;}
```

```
[capture](params) mutable throwSpec -> ret {statements;}
```

```
[] {}
```

```
[capture](params) mutable throwSpec -> ret {statements;}
```

```
[] {  
    std::cout << "Hello!";  
}
```



```
[capture](params) mutable throwSpec -> ret {statements;}
```

```
[] {  
    std::cout << "Hello!";  
}();
```

```
[capture](params) mutable throwSpec -> ret {statements;}
```

```
auto l = [] {  
    std::cout << "Hello!";  
};
```

```
l();
```

```
[capture](params) mutable throwSpec -> ret {statements;}
```

```
std::function<void()> l = [] {  
    std::cout << "Hello!";  
};
```

```
l();
```

```
[capture](params) mutable throwSpec -> ret {statements;}
```

```
auto l = [](const std::string& s) {  
    std::cout << s;  
};
```

```
l("Hello!");
```

```
[capture](params) mutable throwSpec -> ret {statements;}
```

```
[]() {  
    return 4711;  
}
```

```
[capture](params) mutable throwSpec -> ret {statements;}
```

```
[]() -> double {  
    return 4711;  
}
```

```
[capture](params) mutable throwSpec -> ret {statements;}
```

```
[]() {  
    return fA() + fB();  
}
```

decltype: extrahieren eines Typen aus einem Ausdruck

```
int x = 3;
```

```
decltype(x) y = x;
```

```
decltype(x + y) z = x + y;
```



```
[capture](params) mutable throwSpec -> ret {statements;}
```

```
[]() -> decltype(fA() + fB()) {  
    return fA() + fB();  
}
```

```
[capture](params) mutable throwSpec -> ret {statements;}
```

```
int a = 0;  
[=]() {  
    ++a;  
}
```

```
[capture](params) mutable throwSpec -> ret {statements;}
```

```
int a = 0;  
[&]() {  
    ++a;  
}
```

```
[capture](params) mutable throwSpec -> ret {statements;}
```

```
int a = 0;  
[a]() {  
    ++a;  
}
```

```
[capture](params) mutable throwSpec -> ret {statements;}
```

```
int a = 0;  
[&a]() {  
    ++a;  
}
```

`[capture](params) mutable throwSpec -> ret {statements;}`

- `[]` Erfasse keine Variablen aus dem Erstellungskontext.
- `[&]` Erfasse alle Variablen aus dem Erstellungskontext per Referenz.
- `[=]` Erfasse alle Variablen aus dem Erstellungskontext als Kopie.
- `[=, &foo]` Erfasse alle Variablen, außer foo, aus dem Erstellungskontext als Kopie. Erfasse foo als Referenz.
- `[&, foo]` Erfasse alle Variablen, außer foo, aus dem Erstellungskontext als Referenz. Erfasse foo als Kopie.

- `[bar]` Erfasse bar als Kopie. Keine andere Variable aus dem Erstellungskontext wird erfasst.
- `[this]` Erfasse den this Pointer der aktuellen Klasseninstanz.

```
[capture](params) mutable throwSpec -> ret {statements;}
```

```
int id = 0;
```

```
auto l = [id]() mutable {
```

```
    std::cout << "In lambda " << id << std::endl;
```

```
    ++id;
```

```
};
```

```
l(); l(); l();
```

```
std::cout << "Result " << id << std::endl;
```

```
Object capturedObject(1);  
[&](const Object& other) {  
    return capturedObject.getId() == other.getId();  
}
```

```
class Functor {  
public:  
    Functor(const Object& obj) : _obj(obj) {}  
  
    auto operator()(const Object& other) -> bool {  
        return _obj.getId() == other.getId();  
    }  
    ...  
};
```



```
Object capturedObject(1);  
[&](const Object& other) {  
    return capturedObject.getId() == other.getId();  
}
```

```
class Functor {  
public:  
    Functor(const Object& obj) : _obj(obj) {}  
  
    auto operator()(const Object& other) -> bool {  
        return _obj.getId() == other.getId();  
    }  
    ...  
};
```

```
std::vector<Object> objects = {...}  
Object capturedObject(1);
```

```
class Functor {  
public:  
    Functor(const Object& obj) : _obj(obj) {}  
  
    auto operator()(const Object& other) -> bool {  
        return _obj.getId() == other.getId(); }  
    ...  
};
```

```
std::find_if(objects.begin(), objects.end(), Functor(capturedObject));
```

```
std::vector<Object> objects = {...}  
Object capturedObject(1);
```

```
std::find_if(objects.begin(), objects.end(), [&](const Object& other) {  
    return capturedObject.getId() == other.getId();  
});
```

Lambdas sind Functors

~~Lambdas sind Functors~~

Lambdas sind “Syntactic Sugar”

~~Lambdas sind Functors~~

~~Lambdas sind “Syntactic Sugar”~~

Lambdas sind nur “Syntactic Sugar”

Lambdas sind auch “Syntactic Sugar”

Lambdas sind auch “Syntactic Sugar”

- Auch virtuelle Methoden sind “Syntactic Sugar”
- Lambdas erlauben ein neues Level an Abstraktion

STL und Lambdas

Effective STL – Item 43

Prefer algorithm calls to hand-written loops.

STL und Lambdas - Fehlervermeidung!

- Selbstgemacht

```
for( auto i = collection.begin(); i != collection.end(); ++i) {  
    // Do stuff  
    ...  
}
```

- STL ohne Lambdas

```
for_each(collection.begin(), collection.end(), MyFunctor());
```


- STL mit Lambdas

```
for_each(collection.begin(), collection.end(), [](const Entry& e) {  
    // Do stuff  
    ...  
});
```

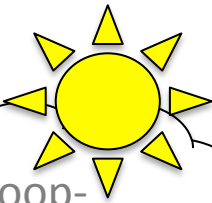
STL und Lambdas - Fehlervermeidung!

- Selbstgemacht

```
for( auto i = collection.begin(); i != collection.end(); ++i) {  
    // Do stuff  
    ...  
}
```



Bei komplexen Loop-Bodies
unklar ob Iteratoren
verändert werden

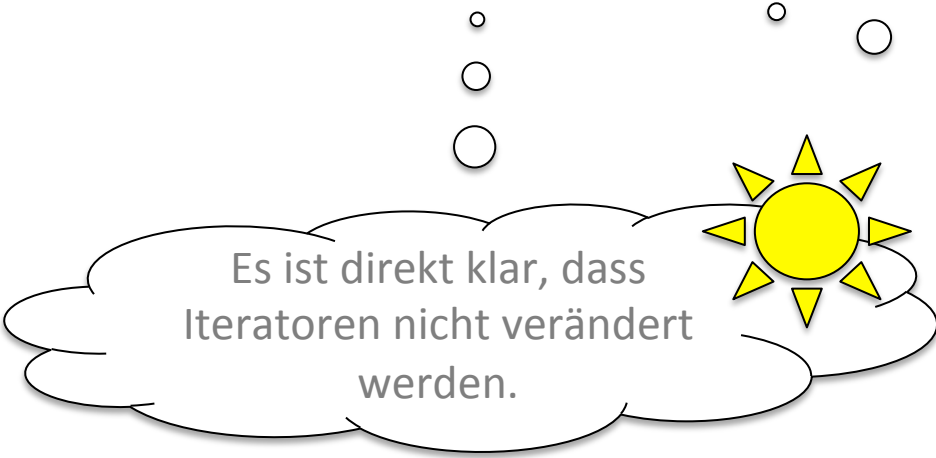


Logik des Loop-
Bodies direkt
einsehbar

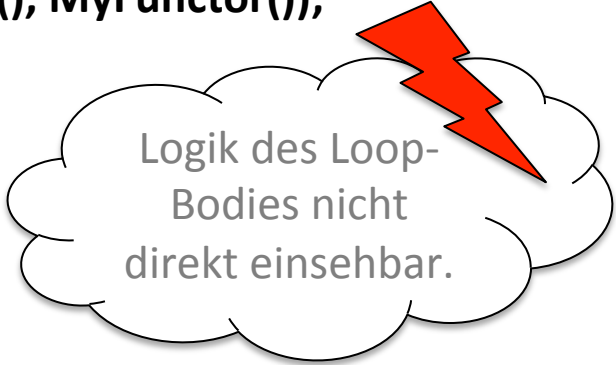
STL und Lambdas - Fehlervermeidung!

- STL ohne Lambdas

`for_each(collection.begin(), collection.end(), MyFunctor());`



Es ist direkt klar, dass
Iteratoren nicht verändert
werden.

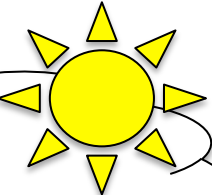


Logik des Loop-
Bodies nicht
direkt einsehbar.

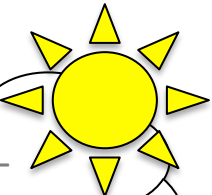
STL und Lambdas - Fehlervermeidung!

- STL mit Lambdas

```
for_each(collection.begin(), collection.end(), [](const Entry& e) {  
    // Do stuff  
    ...  
});
```



Es ist direkt klar, dass
Iteratoren nicht verändert
werden.



Logik des Loop-
Bodies direkt
einsehbar.

STL und Lambdas – Performance!

- Selbstgemacht

```
int sum = 0; long long product = 1;
for(auto i = values.begin(); i != values.end(); ++i) {
    sum += *i; product *= *i;
}
```

- STL ohne Lambdas

```
int sum = std::accumulate(values.begin(), values.end(), 0);
long long product =
    std::accumulate(values.begin(), values.end(), 1, multiplies<int>());
```

- STL mit Lambdas

```
int sum = 0; long long product = 1;
for_each(values.begin(), values.end(), [&](int value) {
    sum += value; product *= value;
});
```

STL und Lambdas - Klarheit!

- Selbstgemacht

```
auto i = v.begin();  
for(; i != v.end(); ++i) {  
    if(*i < 12 && *i > 3) break;  
}
```

- STL ohne Lambdas

```
auto position = std::find_if(v.begin(), v.end(),  
                             bind(logical_and<bool>(),  
                                   bind(less<int>(), _1, 12),  
                                   bind(greater<int>(), _1, 3)));
```

- STL mit Lambdas

```
auto position = std::find_if(v.begin(), v.end(), [](int value) {  
    return 12 > value && 3 < value;  
}));
```

Lambdas haben auch Limitierungen

STL und Lambdas - Limits!

- STL mit Lambdas

```
auto comparator = [](const Object& o1, const Object& o2) {  
    return o1.getId() < o2.getId();  
};
```

```
std::set<Object, decltype(comparator)> theSet(comparator);
```

Lambdas helfen

Lambdas helfen

- bei der Vermeidung handgeschriebener Loops

Lambdas helfen

- bei der Vermeidung handgeschriebener Loops.
- bei einer Verbesserung der Performance.

Lambdas helfen

- bei der Vermeidung handgeschriebener Loops.
- bei einer Verbesserung der Performance.
- bei klarer Code-Strukturierung.

Lambdas helfen

- bei der Vermeidung handgeschriebener Loops.
- bei einer Verbesserung der Performance.
- bei klarer Code-Strukturierung.
- bei der Vermeidung von Bugs.

Lambdas helfen

- bei der Vermeidung handgeschriebener Loops.
- bei einer Verbesserung der Performance.
- bei klarer Code-Strukturierung.
- bei der Vermeidung von Bugs.
- beim Umgang mit STL Funktionen.

But wait, there's more!

Write your own algorithms

```
synchronized(mutex, [&] {  
    // Do synchronized stuff  
    ...  
});
```

Use lambdas asynchronously

```
auto f = std::async([&] {  
    // Do async stuff  
    ...  
});  
  
f.get();
```

?