

3. C++ vertieft (III): Funktoren und Lambda

Was lernen wir heute?

- Funktoren: Objekte mit überladenem Funktionsoperator ().
- Closures
- Lambda-Ausdrücke: Syntaktischer Zucker
- Captures

Funktoren: Motivierung

Ein simpler Ausgabefilter

```
template <typename T, typename Function>
void filter(const T& collection, Function f){
    for (const auto& x: collection)
        if (f(x)) std::cout << x << " ";
    std::cout << "\n";
}
```

filter funktioniert wenn das erste Argument einen Iterator anbietet und das zweite auf Elemente des Iterators angewendet werden kann und das Resultat zu bool konvertierbar ist.

Funktoren: Motivierung

```
template <typename T, typename Function>
void filter(const T& collection, Function f);

template <typename T>
bool even(T x){
    return x % 2 == 0;
}

std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
filter(a,even<int>); // output: 2,4,6,16
```

Funktor: Objekt mit überladenem Operator ()

```
class GreaterThan{  
    int value; // state  
public:  
    GreaterThan(int x):value{x}{}  
  
    bool operator() (int par) const {  
        return par > value;  
    }  
};
```

Ein Funktor ist ein aufrufbares Objekt. Kann verstanden werden als Funktion mit Zustand.

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};  
int value=8;  
filter(a,GreaterThan(value)); // 9,11,16,19
```

Funktor: Objekt mit überladenem Operator ()

```
template <typename T>
class GreaterThan{
    T value;
public:
    GreaterThan(T x):value{x}{}

    bool operator() (T par) const{
        return par > value;
    }
};
```

(geht natürlich auch mit
Template)

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int value=8;
filter(a,GreaterThan<int>(value)); // 9,11,16,19
```

Dasselbe mit Lambda-Expression

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};  
int value=8;  
  
filter(a, [value](int x) {return x > value;});
```

Summe aller Elemente - klassisch

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};  
int sum = 0;  
for (auto x: a)  
    sum += x;  
std::cout << sum << std::endl; // 83
```

Summe aller Elemente - mit Funktor

```
template <typename T>
struct Sum{
    T value = 0;

    void operator() (T par){ value += par; }
};

std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
Sum<int> sum;
// for_each copies sum: we need to copy the result back
sum = std::for_each(a.begin(), a.end(), sum);
std::cout << sum.value << std::endl; // 83
```

Summe aller Elemente - mit Referenzen

```
template <typename T>
struct SumR{
    T& value;
    SumR (T& v):value{v} {}

    void operator() (T par){ value += par; }
};

std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int s=0;
SumR<int> sum{s};
// cannot (and do not need to) assign to sum here
std::for_each(a.begin(), a.end(), sum);
std::cout << s << std::endl; // 83
```

Summe aller Elemente - mit Λ

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
```

```
int s=0;
```

```
std::for_each(a.begin(), a.end(), [&s] (int x) {s += x;} );
```

```
std::cout << s << std::endl;
```

Sortieren, mal anders

```
// pre: i >= 0
// post: returns sum of digits of i
int q(int i){
    int res =0;
    for(;i>0;i/=10)
        res += i % 10;
    return res;
}
```

```
std::vector<int> v {10,12,9,7,28,22,14};
std::sort (v.begin(), v.end(),
    [] (int i, int j) { return q(i) < q(j);}
);
```

Jetzt $v = 10, 12, 22, 14, 7, 9, 28$ (sortiert nach Quersumme)

Lambda-Expressions im Detail

`[value] (int x) ->bool {return x > value;}`

The diagram illustrates the components of the lambda expression `[value] (int x) ->bool {return x > value;}`. Blue curly braces are placed under each part of the expression to group them with their respective labels below:

- `[value]` is labeled "Capture".
- `(int x)` is labeled "Parameter".
- `->bool` is labeled "Rückgabetyp" (Return type).
- `{return x > value;}` is labeled "Anweisung" (Statement).

Closure

```
[value] (int x) ->bool {return x > value;}
```

- Lambda-Expressions evaluieren zu einem temporären Objekt – einer closure
- Die closure erhält den Ausführungskontext der Funktion, also die captured Objekte.
- Lambda-Expressions können als Funktoren implementiert werden.

Simple Lambda-Expression

```
[] ()->void {std::cout << "Hello World";}
```

Aufruf:

```
[] ()->void {std::cout << "Hello World";}();
```

Zuweisung:

```
auto f = [] ()->void {std::cout << "Hello World";};
```

Minimale Lambda-Expression

`[] {}`

- Rückgabetyp kann inferiert werden, wenn kein oder nur ein `return`:²

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

- Keine Parameter und kein expliziter Rückgabetyp $\Rightarrow$ `()` kann weggelassen werden

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

- `[...]` kann nie weggelassen werden.

²Seit C++14 auch mehrere `returns`, sofern derselbe Rückgabetyp deduziert wird

Beispiele

```
[](int x, int y) {std::cout << x * y;} (4,5);
```

Output: 20

Beispiele

```
int k = 8;  
auto f = [](int& v) {v += v;};  
f(k);  
std::cout << k;
```

Output: 16

Beispiele

```
int k = 8;  
auto f = [](int v) {v += v;};  
f(k);  
std::cout << k;
```

Output: 8

Capture – Lambdas

Für Lambda-Expressions bestimmt die capture-Liste über den zugreifbaren Teil des Kontextes

Syntax:

- `[x]`: Zugriff auf kopierten Wert von x (nur lesend)
- `&x`: Zugriff zur Referenz von x
- `&x,y`: Zugriff zur Referenz von x und zum kopierten Wert von y
- `&`: Default-Referenz-Zugriff auf alle Objekte im Kontext der Lambda-Expression
- `=`: Default-Werte-Zugriff auf alle Objekte im Kontext der Lambda-Expression

Capture – Lambdas

```
int elements=0;
int sum=0;
std::for_each(v.begin(), v.end(),
    [&] (int k) {sum += k; elements++;} // capture all by reference
)
```

Capture – Lambdas

Wann wird der Wert gelesen?

```
int v = 42;  
auto func = [=] {std::cout << v << "\n"};  
v = 7;  
func();
```

Ausgabe: 42

Werte werden bei der Definition der (temporären) Lambda-Expression zugewiesen.

Lambda Ausdrücke sind Funktoren

```
[x, &y] () {y = x;}
```

kann implementiert werden als

```
unnamed {x,y};
```

mit

```
class unnamed {  
    int x; int& y;  
    unnamed (int x_, int& y_) : x (x_), y (y_) {}  
    void operator () () {y = x;}  
};
```

Lambda Ausdrücke sind Funktoren

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

kann implementiert werden als

```
unnamed {x,y};
```

mit

```
class unnamed {  
    int x; int y;  
    unnamed (int x_, int y_) : x (x_), y (y_) {}  
    int operator () () const {return x + y;}  
};
```

Polymorphic Function Wrapper `std::function`

```
#include <functional>

int k= 8;
std::function<int(int)> f;
f = [k](int i){ return i+k; };
std::cout << f(8); // 16
```

Kann verwendet werden, um Lambda-Expressions zu speichern.
Andere Beispiele

```
std::function<int(int,int)>; std::function<void(double)> ...
```

<http://en.cppreference.com/w/cpp/utility/functional/function>

Beispiel

```
template <typename T>
auto toFunction(std::vector<T> v){
    return [v] (T x) -> double {
        int index = (int)(x+0.5);
        if (index < 0) index = 0;
        if (index >= v.size()) index = v.size()-1;
        return v[index];
    };
}
```

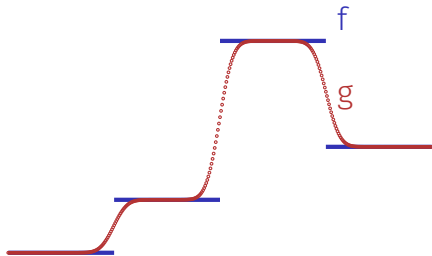
Beispiel

```
auto Gaussian(double mu, double sigma){  
    return [mu,sigma](double x) {  
        const double a = ( x - mu ) / sigma;  
        return std::exp( -0.5 * a * a );  
    };  
}
```

```
template <typename F, typename Kernel>  
auto smooth(F f, Kernel kernel){  
    return [kernel,f] (auto x) {  
        // compute convolution ...  
        // and return result  
    };  
}
```

Beispiel

```
std::vector<double> v {1,2,5,3};  
auto f = toFunction(v);  
auto k = Gaussian(0,0.1);  
auto g = smooth(f,k);
```



Zusammenfassung

- Funktoren erlauben die funktionale Programmierung mit C++. Lambdas sind syntaktischer Zucker, der das deutlich vereinfacht
- Mit Funktoren/Lambdas sind klassische Muster aus der funktionalen Programmierung (z.B. map / filter / reduce) auch in C++ möglich.
- In Kombination mit Templates und Typinferenz (**auto**) können sehr mächtige Funktionen in Variablen gespeichert werden, Funktionen können sogar Funktionen zurückgeben (sog. Funktionen höherer Ordnung).

Übung heute

- Wenden Sie Lambda-Ausdrücke in einem Map/Filter Algorithmus auf einen Vektor an.
- Implementieren Sie eine Funktion (ein Lambda), welche die Ableitung einer gegebenen Funktion approximiert.