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

Generische Programmierung: Funktionen höherer Ordnung

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";
}
```

Frage: wann funktioniert ein Aufruf der **`filter`**-Templatefunktion?

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";
}
```

Frage: wann funktioniert ein Aufruf der **`filter`**-Templatefunktion?

Antwort: wenn das erste Argument einen Iterator anbietet und das zweite auf Elemente des Iterators angewendet werden kann und dessen Rückgabewert zu bool konvertiertbar 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
```

Kontext und Überblick

- Letzte C++-Vorlesung: mache Code parametrisch in den Datentypen, auf denen er operiert
  - **`Pair<T>`** für Elementypen **`T`**
  - **`print<C>`** für alle iterierbaren Container **`C`**
- Jetzt: mache Code parametrisch auf einem (Teil) des Algorithmus
  - **`filter(container, predicate)`**
  - **`apply(signal, transformation/filter)`**
- Wir lernen etwas über
  - Funktionen höherer Ordnung: Funktionen mit Funktionen als Argumente
  - Funktoren: Objekte mit überladenem Funktionsoperator **`()`**.
  - Lambda-Ausdrücke: anonyme Funktoren (Syntaktischer Zucker)
  - Closures: Lambdas, die ihre Umgebung speichern

Funktoren: Motivierung

```
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";
}
```

- Anforderung an **f**: muss aufgerufen werden können (...)
- **f** ist eine Art von Funktion ⇒ **filter** ist eine Funktion, die eine Funktion entgegennimmet
- Eine Funktion, die eine Funktion entgegennimmt (oder zurückgibt) heisst **Funktion höherer Ordnung**
- Funktionen höherer Ordnung sind parametrisierbar in ihrer Funktionalität (oder generieren Funktionen)

Was wenn ...

der Filter flexibler sein soll:

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

template <typename T>
bool largerThan(T x, T y){
    return x > y;
}

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

Was wenn …

der Filter flexibler sein soll:

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

template <typename T>
bool largerThan(T x, T y){
    return x > y;
}

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

(Nein, das gibt es so nicht)

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
```

Beobachtungen

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

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

Dem Prädikat muss ein **Name** gegeben werden

- Oft unnötig, viele werden nur einmal benutzt
- Gute Namen nicht immer möglich
- Code an anderer Stelle als seine Anwendung

Overhead: Prädikat mit Zustand als Funktor

- Umständlich, eigentlich ja nur `par > value`

Dasselbe mit Lambda-Expression

Anonyme Funktionen mit **Lambda-Ausdrücken**:

```
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;} );
```

Das ist bloss **syntaktischer Zucker**, aus dem der Compiler einen passenden Funktor generiert

Einschub: Sortieren mit eigenem Komparator

- **`std::sort`** ist generisch
  - im Typ des Iterators
  - in den Werten, über die iteriert wird
  - im verwendeten Vergleichsoperator

```
std::sort (v.begin(), v.end(), std::less());
```

- Der Komparator gibt **`true`** zurück, wenn die verglichenen Elemente in der gewünschten Reihenfolge stehen.

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);}
);
```

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 $v =$

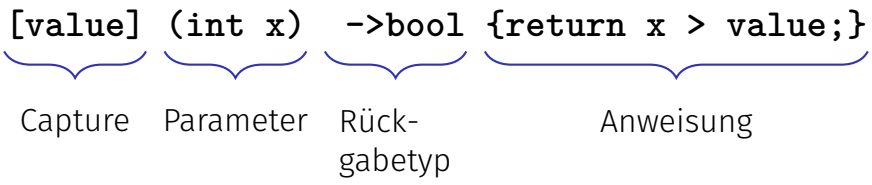
Sortieren, mal anders

```
// 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



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";}
```

Simple Lambda-Expression

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

Aufruf:

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

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:[20]

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

- Keine Parameter und kein expliziter Rückgabetyp ⇒ () kann weggelassen werden

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

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

[20] Seit C++14 auch mehrere returns, sofern derselbe Rückgabetyp deduziert wird

Beispiele

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

Output:

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:

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:

Beispiele

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

Output: 8

Oft genutzt: `std::foreach`

- Übliche Aufgabe: Iteriere über einen Container und führe etwas für jedes Element aus

```
for (auto& name: names) {
  std::cout << name << ' ';
}
```

- Dieses Muster ist typischerweise implementiert als Funktion höherer Ordnung: **`foreach`**

```
for_each(names, [](auto name) {std::cout << name << " ";});
```

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 - klassisch

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

- Aufgabe: erhöhe **sum** für jeden Aufruf in **for_each**
- Problem: **for_each** benötigt Zugriff auf den Kontext (hier **sum**)

Anderes Beispiel **filter**: speichere Elemente eines Vektors in einen anderen Vektor abhängig von einer Bedingung.

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 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
```

- Ok: löst das Problem, operiert aber nicht auf der **sum**-Variablen

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
```

Geht natürlich sehr ähnlich auch mit Zeigern

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;
```

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

```
std::vector<int> v = {1,2,4,8,16};

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

std::cout << "sum=" << sum << " elements=" << elements << std::endl;
```

Ausgabe:

Capture – Lambdas

```
std::vector<int> v = {1,2,4,8,16};

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

std::cout << "sum=" << sum << " elements=" << elements << std::endl;
```

Ausgabe: **sum=31 elements=5**

Capture – Lambdas

```
template <typename T>
void sequence(vector<int> & v, T done){
  int i=0;
  while (!done()) v.push_back(i++);
}

vector<int> s;
sequence(s, [&] {return s.size() >= 5;} )
```

jetzt v =

Capture – Lambdas

```
template <typename T>
void sequence(vector<int> & v, T done){
  int i=0;
  while (!done()) v.push_back(i++);
}

vector<int> s;
sequence(s, [&] {return s.size() >= 5;} )
```

jetzt v = 0 1 2 3 4

Capture – Lambdas

```
template <typename T>
void sequence(vector<int> & v, T done){
  int i=0;
  while (!done()) v.push_back(i++);
}

vector<int> s;
sequence(s, [&] {return s.size() >= 5;} )
```

jetzt v = 0 1 2 3 4

Die capture liste bezieht sich auf den Kontext der Lambda Expression

Capture – Lambdas

Wann wird der Wert gelesen?

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

Ausgabe:

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.

Capture – Lambdas

(Warum) funktioniert das?

```
class Limited{
  int limit = 10;
 public:
  // count entries smaller than limit
  int count(const std::vector<int>& a){
    int c = 0;
    std::for_each(a.begin(), a.end(),
      [=,&c] (int x) {if (x < limit) c++;}
    );
    return c;
  }
};
```

Capture – Lambdas

(Warum) funktioniert das?

```
class Limited{
  int limit = 10;
 public:
  // count entries smaller than limit
  int count(const std::vector<int>& a){
    int c = 0;
    std::for_each(a.begin(), a.end(),
      [=,&c] (int x) {if (x < limit) c++;}
    );
    return c;
  }
};
```

Der **this** pointer wird per default implizit kopiert

Capture – Lambdas

```
struct mutant{
  int i = 0;
  void action(){ [=] {i=42;}();}
};

mutant m;
m.action();
std::cout << m.i;
```

Ausgabe:

Capture – Lambdas

```
struct mutant{
  int i = 0;
  void action(){ [=] {i=42;}();}
};

mutant m;
m.action();
std::cout << m.i;
```

Ausgabe: 42

Der `this` **pointer** wird per default implizit kopiert

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];
  };
}
```

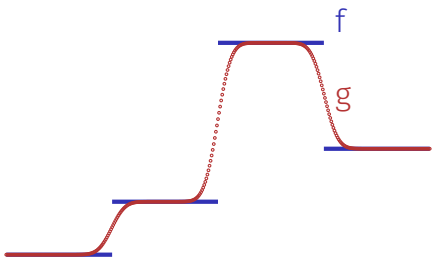
Beispiel

```
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

- Funktionen höherer Ordnung sind in ihrer Funktionalität parametrisch: flexibler $\rightarrow$ breiter anwendbar $\rightarrow$ mehr Wiederverwendung von Code
- Die Möglichkeit, Funktionen weiterzugeben, bedeutet, dass man ganze Berechnungen weitergeben kann.
- Lambda-Ausdrücke erleichtern die Verwendung von "einmaligen" Funktionen, die oft in Kombination mit Funktionen höherer Ordnung verwendet werden.
- Die Rückgabe von Lambdas ermöglicht die Implementierung von Funktionsgeneratoren, die ganze Familien von Funktionen erzeugen können.
- In C++ werden Lambda-Ausdrücke in Funktionsobjekte (Funktoren) übersetzt.
- Funktionen höherer Ordnung und Lambdas sind ein wichtiger und heute gängiger Baustein des funktionalen Programmierparadigmas.