

# 12. Wörterbücher

Wörterbuch, Selbstandornung, Implementation Wörterbuch mit Array / Liste / Skipliste. [Ottman/Widmayer, Kap. 3.3,1.7, Cormen et al, Kap. Problem 17-5]

# Wörterbuch (Dictionary)

ADT zur Verwaltung von Schlüsseln aus  $\mathcal{K}$  mit Operationen

- **insert**( $k, D$ ): Hinzufügen von  $k \in \mathcal{K}$  in Wörterbuch  $D$ . Bereits vorhanden  $\Rightarrow$  Fehlermeldung.
- **delete**( $k, D$ ): Löschen von  $k$  aus dem Wörterbuch  $D$ . Nicht vorhanden  $\Rightarrow$  Fehlermeldung.
- **search**( $k, D$ ): Liefert **true** wenn  $k \in D$ , sonst **false**.

# Idee

Implementiere Wörterbuch als sortiertes Array.

Anzahl Elementaroperationen im schlechtesten Fall

Suchen  $\mathcal{O}(\log n)$  😊

Einfügen  $\mathcal{O}(n)$  😞

Löschen  $\mathcal{O}(n)$  😞

# Andere Idee

Implementiere Wörterbuch als verkettete Liste

Anzahl Elementaroperationen im schlechtesten Fall

Suchen	$\mathcal{O}(n)$	☹️
Einfügen	$\mathcal{O}(1)$ <sup>13</sup>	😊
Löschen	$\mathcal{O}(n)$	☹️

---

<sup>13</sup>Unter der Voraussetzung, dass wir die Existenz nicht prüfen wollen.

# Selbstanordnung

Problematisch bei der Verwendung verketteter Listen: lineare Suchzeit

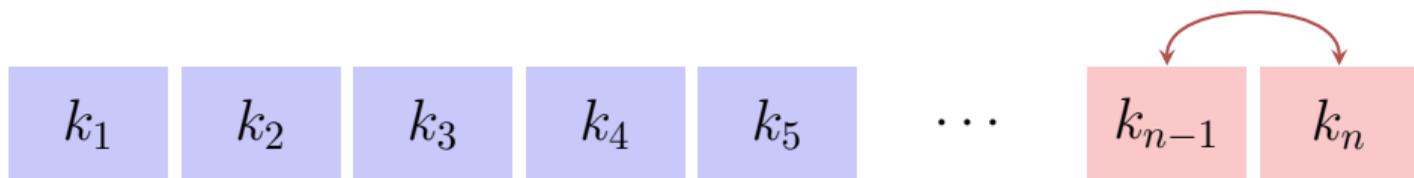
*Idee:* Versuche, die Listenelemente so anzuordnen, dass Zugriffe über die Zeit hinweg schneller möglich sind

Zum Beispiel

- Transpose: Bei jedem Zugriff auf einen Schlüssel wird dieser um eine Position nach vorne bewegt.
- Move-to-Front (MTF): Bei jedem Zugriff auf einen Schlüssel wird dieser ganz nach vorne bewegt.

# Transpose

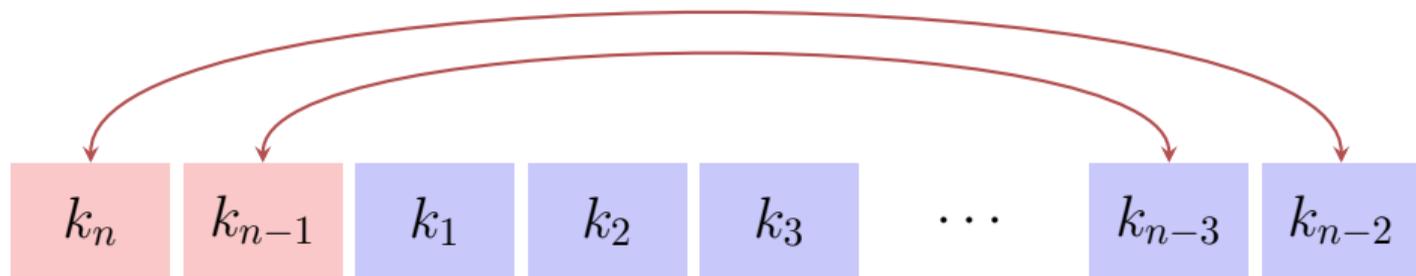
Transpose:



Worst case:  $n$  Wechselnde Zugriffe auf  $k_{n-1}$  und  $k_n$ . Laufzeit:  $\Theta(n^2)$

# Move-to-Front

Move-to-Front:



$n$  Wechselnde Zugriffe auf  $k_{n-1}$  und  $k_n$ . Laufzeit:  $\Theta(n)$

Man kann auch hier Folge mit quadratischer Laufzeit angeben, z.B. immer das letzte Element. Aber dafür ist keine offensichtliche Strategie bekannt, die viel besser sein könnte als MTF.

# Analyse

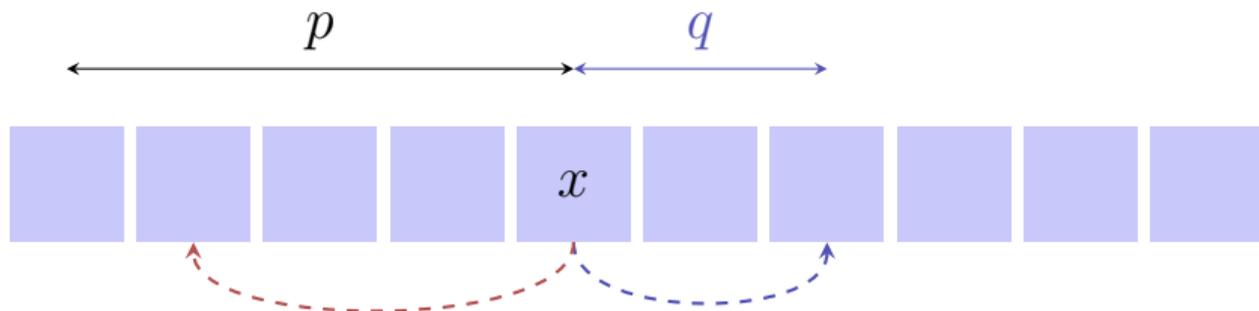
Vergleichen MTF mit dem bestmöglichen Konkurrenten (Algorithmus) A. Wie viel besser kann A sein?

Annahme: MTF und A dürfen jeweils nur das zugegriffene Element  $x$  verschieben. MTF und A starten mit derselben Liste.  $M_k$  und  $A_k$  bezeichnen die Liste nach dem  $k$ -ten Schritt.  $M_0 = A_0$ .

# Analyse

Kosten:

- Zugriff auf  $x$ : Position  $p$  von  $x$  in der Liste.
- Keine weiteren Kosten, wenn  $x$  **vor**  $p$  verschoben wird.
- Weitere Kosten  $q$  für jedes Element, das  $x$  von  $p$  aus nach **hinten** verschoben wird.



# Amortisierte Analyse

Sei eine beliebige Folge von Suchanfragen gegeben und seien  $G_k^{(M)}$  und  $G_k^{(A)}$  jeweils die Kosten im Schritt  $k$  für Move-to-Front und A. Suchen Abschätzung für  $\sum_k G_k^{(M)}$  im Vergleich zu  $\sum_k G_k^{(A)}$ .

⇒ Amortisierte Analyse mit Potentialfunktion  $\Phi$ .

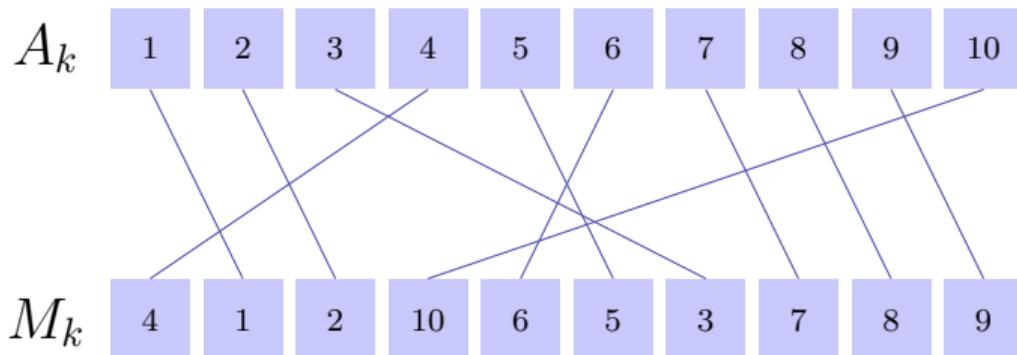
# Potentialfunktion

Potentialfunktion  $\Phi =$  Anzahl der Inversionen von  $A$  gegen MTF.

Inversion = Paar  $x, y$  so dass für die Positionen von  $x$  und  $y$

$$p^{(A)}(x) < p^{(A)}(y) \wedge p^{(M)}(x) > p^{(M)}(y) \text{ oder}$$

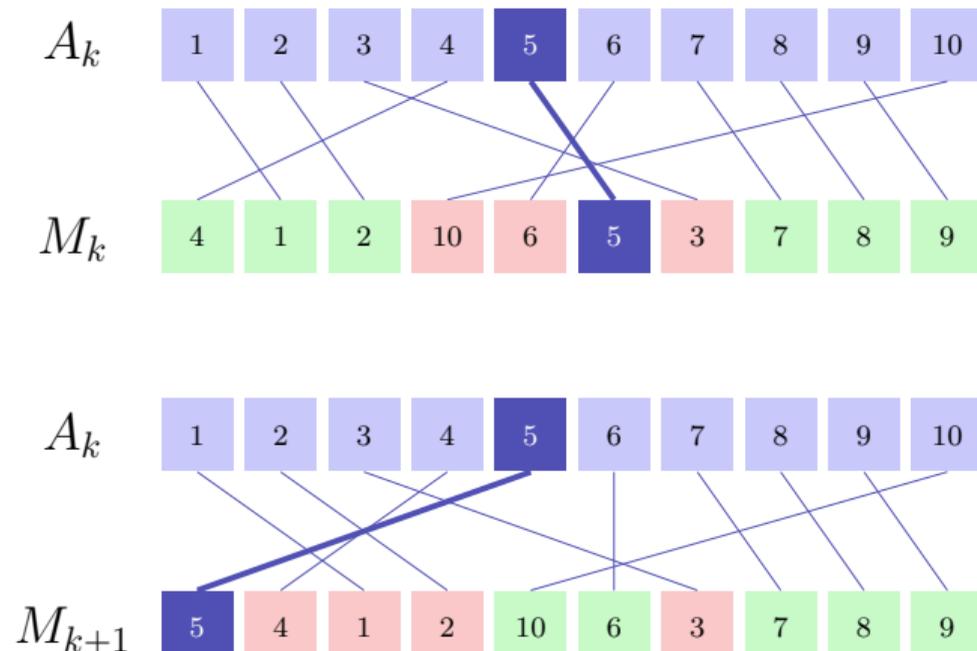
$$p^{(A)}(x) > p^{(A)}(y) \wedge p^{(M)}(x) < p^{(M)}(y)$$



#Inversionen = #Kreuzungen

# Abschätzung der Potentialfunktion: MTF

- Element  $i$  an Position  
 $p_i := p^{(M)}(i)$ .
- Zugriffskosten  $C_k^{(M)} = p_i$ .
- $x_i$ : Anzahl Elemente, die in M vor  $p_i$  und in A nach  $i$  stehen.
- MTF löst  $x_i$  Inversionen auf.
- $p_i - x_i - 1$ : Anzahl Elemente, die in M vor  $p_i$  und in A vor  $i$  stehen.
- MTF erzeugt  $p_i - 1 - x_i$  Inversionen.





# Abschätzung

$$\Phi_{k+1} - \Phi_k \leq -x_i + (p_i - 1 - x_i) + X_k^{(A)}$$

Amortisierte Kosten von MTF im  $k$ -ten Schritt:

$$\begin{aligned} a_k^{(M)} &= C_k^{(M)} + \Phi_{k+1} - \Phi_k \\ &\leq p_i - x_i + (p_i - 1 - x_i) + X_k^{(A)} \\ &= (p_i - x_i) + (p_i - x_i) - 1 + X_k^{(A)} \\ &\leq C_k^{(A)} + C_k^{(A)} - 1 + X_k^{(A)}. \end{aligned}$$

# Abschätzung

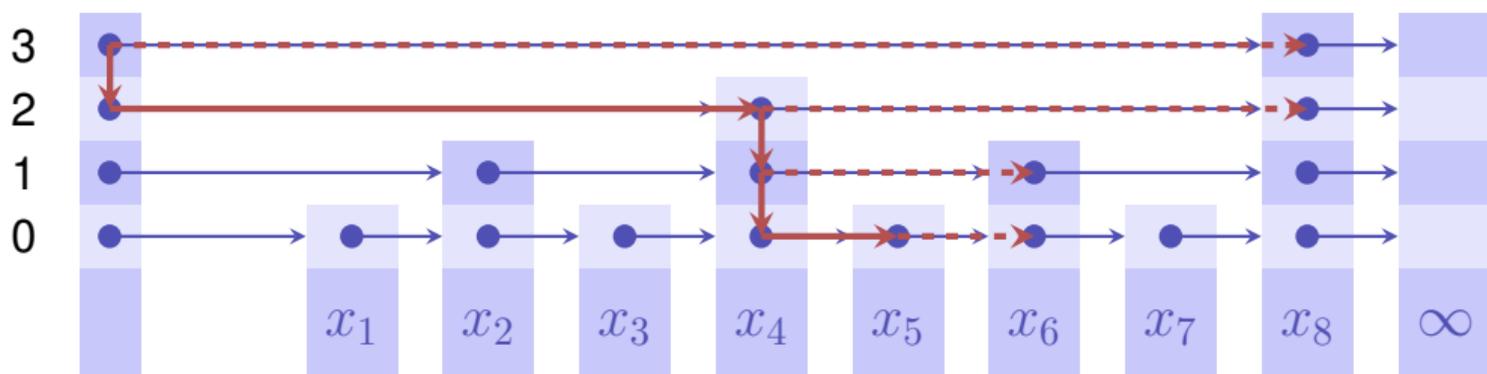
Kosten Summiert

$$\begin{aligned}\sum_k G_k^{(M)} &= \sum_k C_k^{(M)} \leq \sum_k a_k^{(M)} \leq \sum_k 2 \cdot C_k^{(A)} - 1 + X_k^{(A)} \\ &\leq \sum_k 2 \cdot C_k^{(A)} + X_k^{(A)} \leq 2 \cdot \sum_k C_k^{(A)} + X_k^{(A)} \\ &= 2 \cdot \sum_k G_k^{(A)}\end{aligned}$$

MTF führt im schlechtesten Fall höchstens doppelt so viele Operationen aus wie eine optimale Strategie.

# Cooler Idee: Skiplisten

## Perfekte Skipliste



$x_1 \leq x_2 \leq x_3 \leq \dots \leq x_9$ .

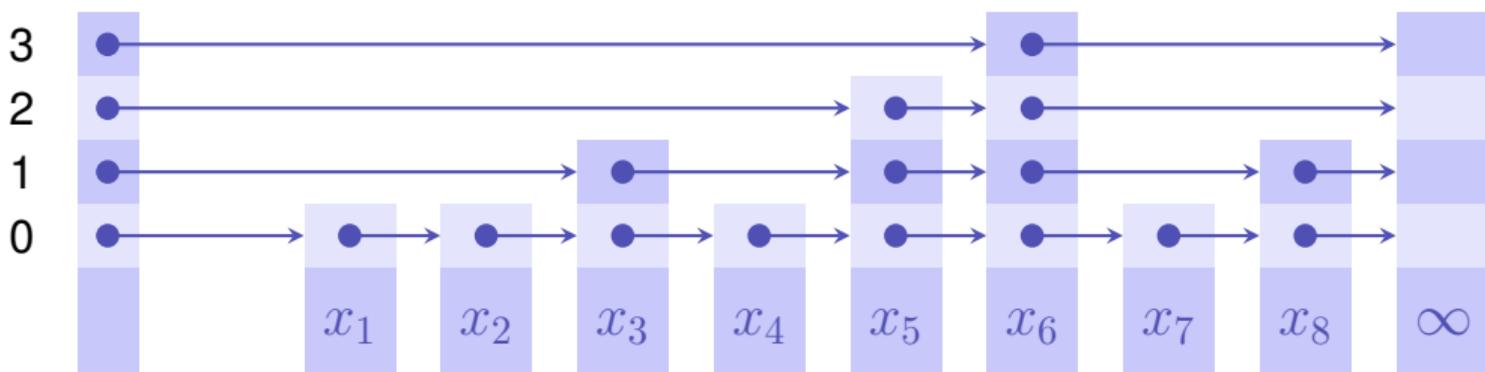
Beispiel: Suche nach einem Schlüssel  $x$  mit  $x_5 < x < x_6$ .

# Analyse Perfekte Skipliste (schlechtester Fall)

Suchen in  $\mathcal{O}(\log n)$ . Einfügen in  $\mathcal{O}(n)$ .

# Randomisierte Skipliste

Idee: Füge jeweils einen Knoten mit zufälliger Höhe  $H$  ein, wobei  $\mathbb{P}(H = i) = \frac{1}{2^{i+1}}$ .



# Analyse Randomisierte Skipliste

## Theorem

*Die Anzahl an erwarteten Elementaroperationen für Suchen, Einfügen und Löschen eines Elements in einer randomisierten Skipliste ist  $\mathcal{O}(\log n)$ .*

Der längliche Beweis, welcher im Rahmen dieser Vorlesung nicht geführt wird, betrachtet die Länge eines Weges von einem gesuchten Knoten zurück zum Startpunkt im höchsten Level.

# 13. C++ vertieft (III): Funktoren und Lambda

## **13.1 Nachträge zum vorigen C++ Kapitel**

# Nachtrag zur Move-Semantik

```
// nonsense implementation of a "vector" for demonstration purposes
class vec{
public:
    vec () {
        std::cout << "default constructor\n";}
    vec (const vec&) {
        std::cout << "copy constructor\n";}
    vec& operator = (const vec&) {
        std::cout << "copy assignment\n"; return *this;}
    ~vec() {}
};
```

# Wie viele Kopien?

```
vec operator + (const vec& a, const vec& b){  
    vec tmp = a;  
    // add b to tmp  
    return tmp;  
}  
  
int main (){  
    vec f;  
    f = f + f + f + f;  
}
```

## Ausgabe

default constructor

copy constructor

copy constructor

copy constructor

copy assignment

4 Kopien des Vektors

# Nachtrag zur Move-Semantik

```
// nonsense implementation of a "vector" for demonstration purposes
class vec{
public:
    vec () { std::cout << "default constructor\n";}
    vec (const vec&) { std::cout << "copy constructor\n";}
    vec& operator = (const vec&) {
        std::cout << "copy assignment\n"; return *this;}
    ~vec() {}
    // new: move constructor and assignment
    vec (vec&&) {
        std::cout << "move constructor\n";}
    vec& operator = (vec&&) {
        std::cout << "move assignment\n"; return *this;}
};
```

# Wie viele Kopien?

```
vec operator + (const vec& a, const vec& b){  
    vec tmp = a;  
    // add b to tmp  
    return tmp;  
}  
  
int main (){  
    vec f;  
    f = f + f + f + f;  
}
```

## Ausgabe

default constructor

copy constructor

copy constructor

copy constructor

move assignment

3 Kopien des Vektors

# Wie viele Kopien?

```
vec operator + (vec a, const vec& b){  
    // add b to a  
    return a;  
}  
  
int main (){  
    vec f;  
    f = f + f + f + f;  
}
```

## Ausgabe

default constructor  
copy constructor  
move constructor  
move constructor  
move constructor  
move assignment

1 Kopie des Vektors

**Erklärung:** Move-Semantik kommt zum Einsatz, wenn ein x-wert (expired) zugewiesen wird. R-Wert-Rückgaben von Funktionen sind x-Werte.

# Wie viele Kopien

```
void swap(vec& a, vec& b){  
    vec tmp = a;  
    a=b;  
    b=tmp;  
}
```

```
int main (){  
    vec f;  
    vec g;  
    swap(f,g);  
}
```

## Ausgabe

default constructor  
default constructor  
copy constructor  
copy assignment  
copy assignment

3 Kopien des Vektors

# X-Werte erzwingen

```
void swap(vec& a, vec& b){
    vec tmp = std::move(a);
    a=std::move(b);
    b=std::move(tmp);
}
int main (){
    vec f;
    vec g;
    swap(f,g);
}
```

## Ausgabe

default constructor  
default constructor  
move constructor  
move assignment  
move assignment

0 Kopien des Vektors

**Erklärung:** Mit `std::move` kann man einen L-Wert Ausdruck zu einem X-Wert (genauer: zu einer R-Wert Referenz) machen. Dann kommt wieder Move-Semantik zum Einsatz. <http://en.cppreference.com/w/cpp/utility/move>

## 13.2 Funktoren und Lambda-Expressions

# 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";
}
```

# 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 LargerThan{
    int value; // state
public:
    LargerThan(int x):value{x}{};

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

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,LargerThan(value)); // 9,11,16,19
```

# Funktor: Objekt mit überladenem Operator ()

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

    bool operator() (T par){
        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,LargerThan<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 << "\n"; // 83
```

# Summe aller Elemente - mit Funktor

```
template <typename T>
struct Sum{
    T & value = 0;
    Sum (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;
Sum<int> sum(s);
sum = std::for_each(a.begin(), a.end(), sum);
std::cout << s << "\n"; // 83
```

# Summe aller Elemente - mit $\Lambda$

```
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 << "\n";
```

# 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;}`

Capture      Parameter      Rück-  
gabe-  
typ      Anweisung

# 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 sein.

# Simple Lambda-Expression

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

Aufruf:

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

# Minimale Lambda-Expression

```
[] {}
```

- Rückgabetypp kann inferiert werden, wenn kein oder nur ein return:

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

- Wenn keine Parameter und keine Rückgabe, kann () weggelassen werden

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

- [...] kann nie weggelassen werden.

# Beispiele

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

Output: 20

# Beispiele

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

Output: 16

# Beispiele

```
int k = 8;  
[](int v) {v += v;} (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 Scope 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

```
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: 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;
    }
};
```

Der `this` pointer wird per default implizit kopiert

# Capture – Lambdas

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

```
mutant m;  
m.do();  
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 () () {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>