Eidgenössische Technische Hochschule Zürich

Datenstrukturen und Algorithmen

Summer Semester 2026

Lecture Notes

Contents

Datenstrukturen und Algorithmen

Fynn Krebser–fkrebser@student.ethz.ch

Sommersemester 2026

Lec 1

1 Introduction

Ziel des Kurses ist es über Effizienz von Algorithmen zu sprechen. Weiter wird ein vertiefter Einblick in C++ sowie ein wenig über paralleles Programmieren gegeben.

Dazu sprechen wir zunächst über Komplexität und danach über verschiedene Standardalgorithmen. Danach werden wir uns mit Graphen beschäftigen. Wenn wir über Algorithmen sprechen müssen wir auch über Datenstrukturen sprechen, da Algorithmen auf Datenstrukturen arbeiten. Ausserdem werden wir über generische und funktionale Programmierung sprechen.

Wir beginnen mit der Komplexität. Dazu eine Wiederholung des Algorithmus

Definition 1.1:

Ein Algorithmus ist eine wohldefinierte Berechnungsvorschrift welche aus INPUT einen OUTPUT berechnet.

Wir schauen uns das Problem der Sortierung an, da diese recht einfach ist. Der Input ist eine Folge von n Zahlen und der Output ist eine Permutation $(a'_1, \dots, a'_n)$ der Inputfolge, so dass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$ gilt.

Schwierig an diesem Problem ist, wenn wir sehr viele Zahlen haben, also n sehr gross ist. Auch spielt die ursprüngliche Reihenfolge der Zahlen eine Rolle, z.B. wenn die Zahlen bereits sortiert sind, dann ist es einfacher als wenn sie in umgekehrter Reihenfolge vorliegen. Hierbei kommt es auf den Algorithmus an, was einfach und was schwierig ist.

Jedes Beispiel eines Inputs nennen wir eine INSTANZ des Problems. Es gibt unendlich viele Instanzen zum Sortierproblem, da es unendlich viele Zahlen gibt.

Grundsätzlich gibt es gute und schlechte Instanzen für einen Algorithmus. Daher bewerten wir Algorithmen meistens im Worst-Case. Teilweise kann man aber auch den Durchschnitt oder den Best-Case betrachten.

Example 1.2:

Ein Beispielalgorithmus ist wie folgt:

- Vergleiche a_0 mit jedem weiteren Element a_i für $i = 1, \dots, n-1$. Wenn $a_0 > a_i$ dann vertausche a_0 und a_i.
- Vergleiche a_1 mit jedem weiteren Element a_i für $i = 2, \dots, n-1$. Wenn $a_1 > a_i$ dann vertausche a_1 und a_i.
- ...

In einer Instanz mit 4 Zahlen würde der Algrithmus 6 Schritte benötigen. Im Fall von $n = 6$ Zahlen wären es 15 Schritte.

Der Code für diesen Algorithmus könnte wie folgt aussehen:

```
1 void sort(std::vector<int>& a){
2   int n = a.size();
3   for (int i = 0; i<n-1; ++i){
4     for (int j = i+1; j<n; ++j){
5       if (a[j] < a[i]){
6         std::swap(a[i],a[j])
7 } } } }
```

Wir möchten nun wissen, wie oft welche Zeile ausgeführt wird. Zeile 2 wird genau einmal ausgeführt, Zeile 3 wird abhängig von n ausgeführt, sagen wir $f_1(n) = n - 1$ mal ab. Für die Zeile 4 ist nun für jedes mal wenn Zeile 3 ausgeführt wird, Zeile 4 abhängig von i und n eine gewisse Anzahl von Malen ausgeführt

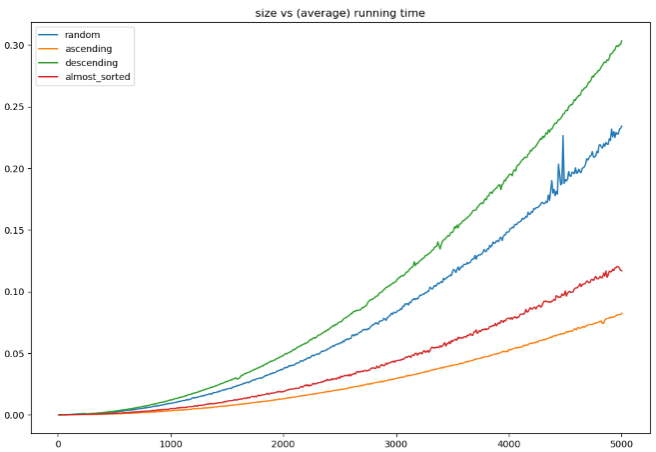
$$f_2(n, i) = \sum_{j=i+1}^{n-1} 1 = n - i - 1.$$

Die 5. Zeile wird dann

$$f_3(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n - i - 1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}.$$

Für sehr grosse n ist $f_3(n)$ ungefähr $\frac{n^2}{2}$, da die $-n$ im Zähler nicht mehr so relevant ist.

Der Swap ist nun interessant. Es könnte sein, dass die Zahlen bereits sortiert sind, dann wird Zeile 5 nie ausgeführt. Es könnte aber auch sein, dass die Zahlen in umgekehrter Reihenfolge vorliegen, dann wird Zeile 5 jedes Mal ausgeführt. Also haben wir im besten Fall $f_4(n) = 0$ und im schlechtesten Fall $f_4(n) = \frac{n(n-1)}{2}$.



Grundsätzlich werden Datenstrukturen verwendet um Daten zu organisieren, so dass diese effizient verarbeitet werden können.

2 Effizienz

Ziel ist es das Laufzeitverhalten von Algorithmen Maschinenunabhängig zu beschreiben. Weiter wollen wir die Effizienz von Algorithmen vergleichen abhängig von ihrer Eingabegrösse. Dazu benötigen wir ein Modell. Dazu abstrahieren wir die Technologie. Z.B. sprechen wir nicht mehr über ein Programm sondern über einen Algorithmus. Weiter werden wir eher in Pseudocode sprechen als in einer konkreten Programmiersprache.

Dazu gibt es zwei häufig verwendete Modelle:

Definition 2.1: Random Access Machine (RAM Modell)

Instruktionen werden der Reihe nach auf einem Prozessorkern ausgeführt.

Das Speichermodell wird durch konstante Zugriffszeit auf alle Adressen charakterisiert.

Elementare Operationen wie Addition, Multiplikation, Vergleich, etc. werden in konstanter Zeit ausgeführt.

Letzten Endes haben wir Fundamentaltypen wie grössenbeschränkte Ints oder Floats.

Definition 2.2: Pointer Machine Modell

Objekte beschränkter Grösse können dynamisch erzeugt werden in konstanter Zeit 1.

Auf Attribute der Objekte kann in konstanter Zeit 1 zugegriffen werden.

Die genaue Laufzeit eines Algorithmus hängt von der Implementierung, der Hardware, der Eingabe, etc. ab. Daher wollen wir die Laufzeit von Algorithmen asymptotisch beschreiben. Dabei ignorieren wir konstante Faktoren. Also ist lineares Wachstum mit Steigung 1 genauso gut wie lineares Wachstum mit Steigung 1000, da beide asymptotisch linear wachsen.

Wir schreiben $\Theta(n^2)$ und meinen dass der Algroithmus sich für grosse n wie n^2 verhält. Das bedeutet: Verdoppelt sich die Problemgrösse, so vervierfacht sich die Laufzeit.

Definition 2.3: Asymptotisch obere Schranke

Gegeben sei $g : \mathbb{N} \to \mathbb{R}$. Dann ist

$$\mathcal{O}(g) = \{f : \mathbb{N} \to \mathbb{R} \mid \exists c > 0, n_0 \in \mathbb{N} \\ : 0 \leq f(n) \leq c \cdot g(n) \forall n \geq n_0\}.$$

Graphisch ist dies in Abbildung 1 dargestellt. Alle Funktionen f welche für grosse n unterhalb von $c \cdot g(n)$ liegen, gehören zu $\mathcal{O}(g)$. Die obige Definition kann man umdrehen für eine asymptotisch untere Schranke $\Omega(g)$.

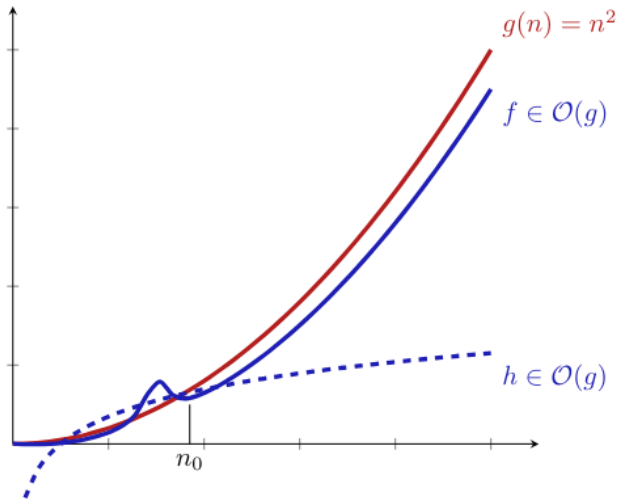


Figure 1: Graphische Darstellung von $\mathcal{O}(g)$

Definition 2.4: Asymptotisch untere Schranke

Gegeben sei $g : \mathbb{N} \to \mathbb{R}$. Dann ist

$$\Omega(g) = \{f : \mathbb{N} \to \mathbb{R} \mid \exists c > 0, n_0 \in \mathbb{N} : 0 \leq c \cdot g(n) \leq f(n) \forall n \geq n_0\}.$$

Weiter gibt es dann die asymptotisch scharfe Schranke $\Theta(g)$.

$$\Theta(g) = \mathcal{O}(g) \cap \Omega(g).$$

Man kann $f = \mathcal{O}(g)$ schreiben, jedoch sollte dieser Ausdruck nicht als Gleichung verstanden werden, sondern als Zugehörigkeit von f zu der Menge $\mathcal{O}(g)$. Daher wird vorzugsweise $f \in \mathcal{O}(g)$ geschrieben.

Lec 2

Definition 2.5: Komplexität

Die Komplexität sind die minimalen Kosten über alle Algorithmen welche das Problem lösen.

3 Algorithmen: Beispiele

3.1 Ägyptische Multiplikation

Eine Beispiel eines Algorithmus ist die Ägyptische Multiplikation. Sie funktioniert wie folgt:

- Gegeben seien zwei Zahlen a und b. Wir wollen $a \cdot b$ berechnen.
- Solange $a \geq 1$ gilt, wiederhole die folgenden Schritte:
  - Wenn a ungerade ist, addiere b zum Ergebnis hinzu.
  - Halbiere a (abrunden) und verdopple b.

Example 3.1:

Berechnen wir $11 \cdot 26$. Dann folgt

a	b	Ergebnis
11	26	0
5	52	26
2	104	78
1	208	78
0	416	286

Das Ergebnis ist 286.

Dieser Algorithmus ist effizient da der Computer sehr schnell mit 2 multiplizieren und zu dividieren ist. Wir wollen nun zeigen, dass die Ägyptische Multiplikation korrekt ist.

Proof. Wir schreiben zunächst unser Problem etwas mathematischer. Im allgemeinen ist

$$a \cdot b = \begin{cases} \frac{a}{2} \cdot 2b & \text{wenn } a \text{ gerade ist} \\ \frac{a-1}{2} \cdot 2b + b & \text{wenn } a \text{ ungerade ist} \\ b & \text{wenn } a = 1 \end{cases}.$$

Als Rekursionsgleichung wäre dies

$$f(a,b) = \begin{cases} f(\frac{a}{2}, 2b) & \text{wenn } a \text{ gerade ist} \\ f(\frac{a-1}{2}, 2b) + b & \text{wenn } a \text{ ungerade ist} \\ b & \text{wenn } a = 1 \end{cases}.$$

Wir wollen nun zeigen, dass $f(a,b) = a \cdot b$ für alle $a, b \in \mathbb{N}$ gilt. Wir verwenden vollständige Induktion über a. $H(1)$: $f(1,b) = b = 1 \cdot b$. $H(a)$: Wir nehmen an, dass $f(a',b) = a' \cdot b$ für alle $a' < a$ gilt. $H(a+1)$: Es gibt zwei Fälle:

- $a+1$ ist gerade: Dann ist $f(a+1,b) = f(\frac{a+1}{2}, 2b)$. Da $\frac{a+1}{2} < a+1$ gilt, können wir die Induktionsannahme anwenden und erhalten $f(\frac{a+1}{2}, 2b) = \frac{a+1}{2} \cdot 2b = (a+1) \cdot b$.
- $a+1$ ist ungerade: Dann ist $f(a+1,b) = f(\frac{a}{2}, 2b) + b$. Da $\frac{a}{2} < a+1$ gilt, können wir die Induktionsannahme anwenden und erhalten $f(\frac{a}{2}, 2b) = \frac{a}{2} \cdot 2b = a \cdot b$. Also ist $f(a+1,b) = a \cdot b + b = (a+1) \cdot b$.

□

Proof. Alternativ könne wir mit invarianten Argumentieren. Dazu wandeln wir unser Programm in ein iteratives um. Wir haben eine Variable res welche der dritten Zeile aus Beispiel 3.1 entspricht. Dann ist die Invariante $res + a \cdot b = a_0 \cdot b_0$, wobei a_0 und b_0 die ursprünglichen Werte von a und b sind.

Am Ende des Algorithmus ist $a = 0$, also $res = a_0 \cdot b_0$. Am Anfang des Algorithmus ist $res = 0$, also $res + a \cdot b = a_0 \cdot b_0$. □

3.2 Schnelle Multiplikation von Zahlen

In der Primarschule lernen wir die Multiplikation von Zahlen mit der sogenannten *long multiplication*. Diese sieht wie folgt aus:

62	37
434	
1860	
	2294

Im allgemeinen braucht dieser Algorithmus n^2 einzahlige Multiplikationen.

Wir bemerken dass wir schreiben können:

$$\begin{aligned} ab \cdot cd &= (10a + b)(10c + d) \\ &= 100ac + 10ac + 10bd + bd + 10(a - b)(d - c). \end{aligned}$$

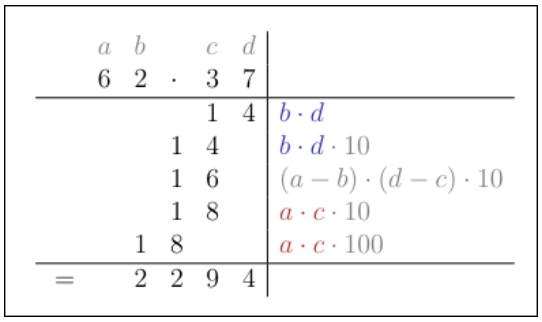


Figure 2: Karatsuba Multiplikation

Wenn wir nun grössere Zahlen haben, so können wir diese in Blöcke von $n/2$ Ziffern aufteilen. Dann können wir die Multiplikation mit nur 3 Multiplikationen von Zahlen der Länge $n/2$ durchführen, anstatt 4 Multiplikationen zu benötigen.

$$6237 \cdot 5898 = \underbrace{62}_{\text{a}}\underbrace{37}_{\text{b}} \cdot \underbrace{58}_{\text{c}}\underbrace{98}_{\text{d}}.$$

Folglich benötigen wir für eine vierstellige Multiplikation nur drei zweistellige Multiplikationen also 9 einstellige Multiplikationen, anstatt 16 einstellige Multiplikationen zu benötigen.

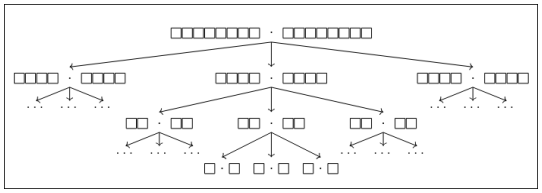


Figure 3: Karatsuba for 8 digit numbers

Nehmen wir nun an dass wir zwei n stellige Zahlen haben mit $n = 2^k$. Dann können wir schreiben

$$\begin{aligned} &(10^{n/2}a + b)(10^{n/2}c + d) \\ &= 10^n ac + 10^{n/2}ac + 10^{n/2}bd + bd + 10^{n/2}(a - b)(d - c). \end{aligned}$$

Dieser Algorithmus ist der Algorithmus von Karatsuba und Ofman.

Die Grundidee hier war Divide and Conquer

Theorem 3.2: Divide and Conquer

Zerlege das Problem in kleinere Teilprobleme, löse diese Teilprobleme und kombiniere die Lösungen der Teilprobleme zu einer Lösung des ursprünglichen Problems.

Wie viele einstellige Multiplikationen benötigen wir nun für diesen Algorithmus? Die Anzahl einstelliger Multiplikationen ist

$$M(2^k) = \begin{cases} 1 & \text{wenn } k = 0 \\ 3M(2^{k-1}) & \text{wenn } k > 0 \end{cases}.$$

Eine Möglichkeit diese Rekursionsgleichung zu lösen ist Teleskopieren. Dabei ersetzen wir $M(2^{k-1})$ durch $3M(2^{k-2})$ und so weiter, bis wir $M(1)$ erreichen. In diesem Fall ist

$$M(2^k) = \underbrace{3 \cdot 3 \cdot \cdots \cdot 3}_{k \text{ mal}} M(1) = 3^k.$$

Eine andere Möglichkeit wäre ein Beweis mit vollständiger Induktion über k zu führen.

Folglich ist $M(n) = M(2^{\log_2 n}) = n^{\log_2 3}$. Da $\log_2 3 < 1.6$ ist, ist dies deutlich besser als die n^2 einstellige Multiplikationen welche die long multiplication benötigt.

Dies ist nicht die schnellste bekannte Methode zur Multiplikation von Zahlen, es gibt sogar Algorithmen welche in $\mathcal{O}(n \log n)$ Zeit multiplizieren können. Dies geht aber über den Inhalt dieses Kurses hinaus.

3.3 Maximum Subarray Problem

Betrachten wir das folgende Problem:

Definition 3.3: Maximum Subarray Problem

Gegeben sei eine Folge von n Zahlen $a_1, \ldots, a_n \in \mathbb{R}$. Finde die zusammenhängende Teilfolge mit der grössten Summe.

Ein erster naiver Algorithmus wäre, alle möglichen zusammenhängenden Teilfolgen zu betrachten und die Summe zu berechnen. Es gibt n Möglichkeiten für den Anfang der Teilfolge und n Möglichkeiten für das Ende der Teilfolge, also gibt es n^2 mögliche Teilfolgen. Das Berechnen der Summe einer Teilfolge dauert im schlimmsten Fall n Schritte, also ist die Laufzeit dieses Algorithmus $\Theta(n^3)$.

Dieser Algorithmus kann verbessert werden durch Präfixsummen. Wenn wir wissen, was die Summe der ersten i Elemente ist, dann können wir die Summe einer Teilfolge von i bis j berechnen, indem wir die Summe der ersten j Elemente minus die Summe der ersten $i - 1$ Elemente nehmen.

$$S_{i,j} = \sum_{k=i}^{j} a_k = \underbrace{\sum_{k=1}^{j} a_k}_{S_j} - \underbrace{\sum_{k=1}^{i-1} a_k}_{S_{i-1}} = S_j - S_{i-1}.$$

Wir berechnen also zuerst die Präfixsummen S_i für $i = 1, \ldots, n$ in $\mathcal{O}(n)$ Zeit. Danach können wir die Summe jeder Teilfolge in konstanter Zeit berechnen, da wir nur zwei Präfixsummen subtrahieren müssen. Da es n^2 mögliche Teilfolgen gibt, ist die Laufzeit dieses Algorithmus $\Theta(n^2)$.

Wir können nun divide and conquer verwenden um den Algorithmus weiter zu verbessern. Wir teilen die Folge in zwei Hälften auf und berechnen das Maximum Subarray in der linken Hälfte, das Maximum Subarray in der

rechten Hälfte und das Maximum Subarray welches die Mitte überschreitet. Das Maximum Subarray, welches die Mitte überschreitet, kann in linearer Zeit berechnet werden, indem wir von der Mitte aus nach links und nach rechts gehen und das Maximum Links und Rechts der Suffix und Präfixsumme betrachten. Die Laufzeit dieses Algorithmus ist dann $\Theta(n \log n)$, da wir die Folge in jeder Rekursion halbieren und die Berechnung des Maximum Subarray, welches die Mitte überschreitet, linear ist.

$$T(n) = \begin{cases} d & \text{wenn } n = 1 \\ 2T(n/2) + c \cdot n & \text{wenn } n > 1 \end{cases}.$$

Die Beobachtung um den Algorithmus zu verbessern ist, dass wenn wir wissen wo der Endpunkt ist, dann können wir den Startpunkt in linearer Zeit berechnen. Daher können wir einfach von rechts nach links gehen und die Summe berechnen, bis wir die maximale Summe erreichen. Dazu setzen wir

$$R_i := \max_{1 \le r \le i+1} S_{r,i} = \max_{1 \le r \le i+1} \sum_{k=r}^{i} a_k.$$

Angenommen, wir haben R_{i-1} berechnet, dann können wir R_i berechnen durch

$$R_i = \max\{R_{i-1} + a_i, a_i\}.$$

4 Effizientes Speichermanagement

Lec 3

Eine Grosse Stärke von C++ ist die Möglichkeit, Variablen mit Werten zu darzustellen. Zum Beispiel kann mit `std::vector<int> w=v` ein ganzer Vektor kopiert werden.

Dies bedeutet auch, dass standardmässig alle Variablen BY VALUE an Funktionen übergeben werden. Somit verändert sich ein Vektor nicht, wenn er in einer Funktion verwendet wird. Andererseits, muss der Speicher kopiert werden, was Zeit kostet.

Es können Referenztypen verwendet werden, um Argumente als Alias zu übergeben, so dass kein Speicher kopiert werden muss. Nachteil davon ist, dass so keine Werte übergeben werden können, welche keine Adresse im Speicher haben, da es auch keinen Sinn macht z.B. `swap(3,a+a)` aufzurufen.

Mögliche Nachteile einer pass by reference sind, dass wir keine R-Werte übergeben können und dass wir nicht mehr sicher sein können, dass die übergebenen Werte nicht verändert werden, da sie als Alias übergeben werden.

Eine Möglichkeit ist eine Const Referenz zu verwenden. Dies löst beide unsere Probleme.

4.1 Wertekategorien

Audrücke in C++ haben einen Typ, repräsentieren einen Wert und repräsentieren möglicherweise ein Objekt mit Adresse.

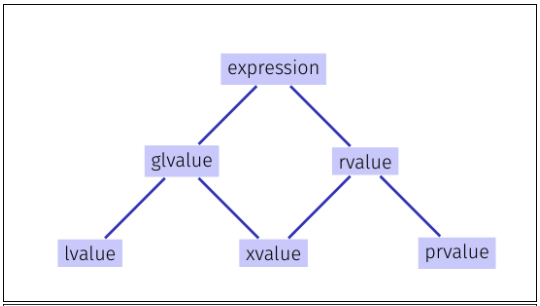
Wenn wir einen Zuweisung wie L=R ausführen möchten, müssen die Typen übereinstimmen aber auch die Werte Kategorien von L und R müssen stimmen.

In Informatik haben wir Werte welche Links von einer Zuweisung stehen L-Werte genannt und Werte welche Rechts von einer Zuweisung stehen R-Werte genannt.

Somit kann ein L-Wert verwendet werden, wo ein R-Wert erwartet wird, aber nicht umgekehrt.

Heutzutage hat sich C++ weiterentwickelt und es gibt weitere Wertekategorien.

z.B. die generalized l value, die pure r value und die expired value.



Eigenschaft		Hat Identität	Keine Identität
		glvalue	
Kann verschoben werden	rvalue	xvalue	prvalue
Kann nicht verschoben werden		lvalue	

Figure 4: Die Wahrheit

Wenn wir v=w schreiben können, dann muss v weiter leben und somit kann v nicht verschoben werden. Wenn

wir z.B. output(abs(v)) schreiben, dann könnte abs(v) verschoben werden, da es nicht weiter lebt. Eine prvalue sind Literale. Sie haben keinen Namen und können nicht verschoben werden, da sie keinen Speicher haben.

Table 1: Aufschlüsselung der Wertekategorien

Kartegorie	Beispiel
L-Wert	Variablen, Zeiger
xvalue	Resultat eines Funktionaufrufs
prvalue	Literale, Resultat von Operatoren

Definition 4.1: R-Wert-Referenz

Eine R-Wert-Referenz ist eine Referenz auf einen R-Wert. Sie wird mit `&&` deklariert.

Wenn das Quellobjekt einer Zuweisung direkt nach der Zuweisung nicht weiter existiert, dann kann der Compiler den Move-Zuweisungsoperator anstelle des Zuweisungsoperators einsetzen. Damit wird eine potentiell teure Kopie vermieden.

Dabei wird der Move-Konstruktor oder der Move-Zuweisungsoperator aufgerufen, wenn das Quellobjekt ein R-Wert ist. Wenn das Quellobjekt ein L-Wert ist, wird der Kopierkonstruktor oder der Kopierzuweisungsoperator aufgerufen.

Definiert werden die Move-Konstruktor und der Move-Zuweisungsoperator wie folgt:

```
class MyClass {
public:
    // Move-Konstruktor
    MyClass(MyClass&& other) {
        // Ressourcen von 'other' übernehmen
    }
    MyClass& operator=(MyClass&& other) {
        if (this != &other) {
            // Ressourcen von 'other' übernehmen
        }
        return *this;
    }
};
```

Ein Beispiel dafür wäre der Code

```
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;
}
```

Hierbei werden ganze 4 Kopien erstellt. Wenn wir den Move operator hinzufügen, werden nur noch 3 Kopien erstellt. Hierbei könnte man den Compiler unterstützen und den Vektor a by value übergeben.

```
Vec operator + (Vec a, const Vec& b) {
    // Add b to a
    return a;
}
```

Nun wird nur noch eine Kopie erstellt, bei der ersten Addition.

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

5 C++ Advanced: Templates

Betrachte die folgende Funktion:

```
double sum(const std::vector<int>& v){
  double result = 0;
  for (auto x: v)
    result += x;
  return result;
}
```

Wir fragen uns nun, wie wir diese Funktion generischer machen können.

- Anderer Rückgabetyp
- Anderer Datentyp im Vektor
- Anderer Startwert
- Operation Anpassbar
- Datenstruktur anpassbar
- Eine Selektion der Daten

Für den letzten Punkt können wir ein Iterator-Paar übergeben.

Als laufendes Beispiel verwenden wir

```
class Pair{
  int left; int right;
public:
    Pair(int l, int r): left(l), right(r) {}

    int min() return left < right ? left : right;
};
```

Wir können nun vor der Definition der Klasse eine Template-Deklaration hinzufügen, um die Klasse generisch zu machen.

```
template<typename T>
class Pair{
    T left; T right;
public:
    Pair(T l, T r): left(l), right(r) {}
};
```

Dies nennt man Parametrischer Polymorphismus. Wenn die Klasse zu Ende ist hört dieses Typename automatisch auf. Es kann gelesen werden wie: Für alle Typen T.

Der Compiler generiert dann für jede Instanziierung der Klasse eine neue Klasse. Zum Beispiel wird die Klasse `Pair<int>` generiert, wenn wir `Pair<int> p(1,2)` schreiben.

Aktuell haben wir noch das Problem, dass wir nicht ein Pair von Pairs erstellen können. Der Compiler überprüft nur so wenig wie Nötig was potentiell zu Problemen führen kann.

Für Funktionen können wir ebenfalls Templates verwenden. Zum Beispiel könnte die Funktion

```
template<typename T>
T min(T l, T r){
    return l < r ? l : r;
}
```

Hierbei kann man auch die Funktion min<double> aufrufen, um die Funktion für double aufzurufen.

Bei unserer Implementation müssen nun aber beide Typen identisch sein. Hierfür können wir die Funktion wie folgt anpassen:

```
template<typename T1, typename T2>
auto min(T1 l, T2 r){
    return l < r ? l : r;
}
```

Diese Technik können wir auf unserem Pair Beispiel verwenden, um die Funktion min zu implementieren.

```
template<typename T>
class Pair{
    T left; T right;
public:
    Pair(T l, T r): left(l), right(r) {}
    auto min() return min(left, right);
};
```

Wenn der Typ bei der Instanzierung nicht inferiert werden kann, muss er explizit angegeben werden.

Index