6. C++ vertieft (I)

Kurzwiederholung: Vektoren, Zeiger und Iteratoren
Bereichsbasiertes for, Schlüsselwort auto, eine Klasse für Vektoren, Subskript-Operator, Move-Konstruktion, Iterator.

Wir erinnern uns...

```
#include <iostream>
#include <vector>

int main(){
  // Vector of length 10
  std::vector<int> v(10,0);
  // Input
  for (int i = 0; i < v.length(); ++i)
    std::cin >> v[i];
  // Output
  for (std::vector::iterator it = v.begin(); it != v.end(); ++it)
    std::cout << *it << " ";
}
```

Das wollen wir doch genau verstehen!

Und zumindest das scheint uns zu umständlich!

Nützliche Tools (1): `auto` (C++11)

Das Schlüsselwort `auto`:

Der Typ einer Variablen wird inferiert vom Initialisierer.

Beispiele

```
int x = 10;
auto y = x; // int
auto z = 3; // int
std::vector<double> v(5);
auto i = v[3]; // double
```


Etwas besser...

```
#include <iostream>
#include <vector>

int main(){
  std::vector<int> v(10,0); // Vector of length 10

  for (int i = 0; i < v.length(); ++i)
    std::cin >> v[i];

  for (auto it = x.begin(); it != x.end(); ++it){
    std::cout << *it << " ";
  }
}
```

Nützliche Tools (2): Bereichsbasiertes `for` (C++11)

```
for (range-declaration : range-expression)
    statement;
```

range-declaration: benannte Variable vom Elementtyp der durch range-expression spezifizierten Folge.
range-expression: Ausdruck, der eine Folge von Elementen repräsentiert via Iterator-Paar `begin()`, `end()` oder in Form einer Initialisierungsliste.

Beispiele

```
std::vector<double> v(5);
for (double x: v) std::cout << x; // 00000
for (int x: {1,2,5}) std::cout << x; // 125
for (double& x: v) x=5;
```


Ok, das ist cool!

```
#include <iostream>
#include <vector>

int main(){
  std::vector<int> v(10,0); // Vector of length 10

  for (auto& x: v)
    std::cin >> x;

  for (const auto i: x)
    std::cout << i << " ";
}
```


Für unser genaues Verständis

Wir bauen selbst eine Vektorklasse, die so etwas kann!

Auf dem Weg lernen wir etwas über

- RAII (Resource Acquisition is Initialization) und Move-Konstruktion
- Index-Operatoren und andere Nützlichkeiten
- Templates
- Exception Handling
- Funktoren und Lambda-Ausdrücke

Eine Klasse für Vektoren

```
class vector{
  int size;
  double* elem;
public:
    // constructors
    vector(): size{0}, elem{nullptr} {};

    vector(int s):size{s}, elem{new double[s]} {}
    // destructor
    ~vector(){
        delete[] elem;
    }
    // something is missing here
}
```

Elementzugriffe

```
class vector{
   ...
   // getter. pre: 0 <= i < size;
   double get(int i) const{
       return elem[i];
   }
   // setter. pre: 0 <= i < size;
   void set(int i, double d){  // setter
       elem[i] = d;
   }
   // length property
   int length() const {
       return size;
   }
}
```

```
class vector{
public:
    vector();
    vector(int s);
    ~vector();
    double get(int i) const;
    void set(int i, double d);
    int length() const;
}
```


Was läuft schief?

```
int main(){
  vector v(32);
  for (int i = 0; i<v.length(); ++i)
    v.set(i,i);
  vector w = v;
  for (int i = 0; i<w.length(); ++i)
    w.set(i,i*i);
  return 0;
}
```

```
class vector{
public:
  vector();
  vector(int s);
  ~vector();
  double get(int i);
  void set(int i, double d);
  int length() const;
}
```

```
*** Error in `vector1': double free or corruption
(!prev): 0x0000000000d23c20 ***
======= Backtrace: =========
/lib/x86_64-linux-gnu/libc.so.6(+0x777e5)[0x7fe5a5ac97e5]
```


Rule of Three!

```
class vector{
...
  public:
  // Copy constructor
  vector(const vector &v):
    size{v.size}, elem{new double[v.size]} {
    std::copy(v.elem, v.elem+v.size, elem);
  }
}
```

```
class vector{
public:
  vector();
  vector(int s);
  ~vector();
  vector(const vector &v);
  double get(int i);
  void set(int i, double d);
  int length() const;
}
```


Rule of Three!

```
class vector{
...
  // Assignment operator
  vector& operator=(const vector&v){
    if (v.elem == elem) return *this;
    if (elem != nullptr) delete[] elem;
    size = v.size;
    elem = new double[size];
    std::copy(v.elem, v.elem+v.size, elem);
    return *this;
  }
}
```

```
class vector{
public:
  vector();
  vector(int s);
  ~vector();
  vector(const vector &v);
  vector& operator=(const vector&v);
  double get(int i);
  void set(int i, double d);
  int length() const;
}
```

Jetzt ist es zumindest korrekt. Aber umständlich.

Eleganter geht so:

```
class vector{
...
  // Assignment operator
  vector& operator= (const vector&v){
    vector cpy(v);
    swap(cpy);
    return *this;
  }
private:
  // helper function
  void swap(vector& v){
    std::swap(size, v.size);
    std::swap(elem, v.elem);
  }
}
```

```
class vector{
public:
  vector();
  vector(int s);
  ~vector();
  vector(const vector &v);
  vector& operator=(const vector&v);
  double get(int i);
  void set(int i, double d);
  int length() const;
}
```


Arbeit an der Fassade.

Getter und Setter unschön. Wir wollen einen Indexoperator.

Überladen! So?

```
class vector{
...
  double operator[] (int pos) const{
    return elem[pos];
  }

  void operator[] (int pos, double value){
    elem[pos] = double;
  }
}
```

Nein!

Referenztypen!

```
class vector{
...
  // for const objects
  double operator[] (int pos) const{
    return elem[pos];
  }
  // for non-const objects
  double& operator[] (int pos){
    return elem[pos]; // return by reference!
  }
}
```

```
class vector{
public:
  vector();
  vector(int s);
  ~vector();
  vector(const vector &v);
  vector& operator=(const vector&v);
  double operator[] (int pos) const;
  double& operator[] (int pos);
  int length() const;
}
```


Soweit, so gut.

```
int main(){
  vector v(32); // Constructor
  for (int i = 0; i<v.length(); ++i)
    v[i] = i; // Index-Operator (Referenz!)

  vector w = v; // Copy Constructor
  for (int i = 0; i<w.length(); ++i)
    w[i] = i*i;

  const auto u = w;
  for (int i = 0; i<u.length(); ++i)
    std::cout << v[i] << ":" << u[i] << " "; // 0:0 1:1 2:4 ...
  return 0;
}
```

```
class vector{
public:
  vector();
  vector(int s);
  ~vector();
  vector(const vector &v);
  vector& operator=(const vector&v);
  double operator[] (int pos) const;
  double& operator[] (int pos);
  int length() const;
}
```

Anzahl Kopien

Wie oft wird `v` kopiert?

```
vector operator+ (const vector& l, double r){
    vector result (l);  // Kopie von l nach result
    for (int i = 0; i < l.length(); ++i) result[i] = l[i] + r;
    return result; // Dekonstruktion von result nach Zuweisung
}

int main(){
    vector v(16); // Allokation von elems[16]
    v = v + 1;   // Kopie bei Zuweisung!
    return 0;    // Dekonstruktion von v
}
```

`v` wird zwei Mal kopiert.

Move-Konstruktor und Move-Zuweisung

```
class vector{
...
    // move constructor
    vector (vector&& v){
        swap(v);
    };
    // move assignment
    vector& operator=(vector&& v){
        swap(v);
        return *this;
    };
}
```

```
class vector{
public:
  vector();
  vector(int s);
  ~vector();
  vector(const vector &v);
  vector& operator=(const vector&v);
  vector (vector&& v);
  vector& operator=(vector&& v);
  double operator[] (int pos) const;
  double& operator[] (int pos);
  int length() const;
}
```


Erklärung

Wenn das Quellobjekt einer Zuweisung direkt nach der Zuweisung nicht weiter existiert, dann kann der Compiler den Move-Zuweisungsoperator anstelle des Zuweisungsoperators einsetzen.[3] Damit wird eine potentiell teure Kopie vermieden.

Anzahl der Kopien im vorigen Beispiel reduziert sich zu 1.

[3] Analoges gilt für den Kopier-Konstruktor und den Move-Konstruktor.

Bereichsbasiertes `for`

Wir wollten doch das:

```
vector v = ...;
for (auto x: v)
  std::cout << x << " ";
```

Dafür müssen wir einen Iterator über `begin` und `end` bereitstellen.

Iterator für den Vektor

```
class vector{
...
    // Iterator
    double* begin(){
        return elem;
    }
    double* end(){
        return elem+size;
    }
}
```

```
class vector{
public:
  vector();
  vector(int s);
  ~vector();
  vector(const vector &v);
  vector& operator=(const vector&v);
  vector (vector&& v);
  vector& operator=(vector&& v);
  double operator[] (int pos) const;
  double& operator[] (int pos);
  int length() const;
  double* begin();
  double* end();
}
```


Const Iterator für den Vektor

```
class vector{
...
        // Const-Iterator
    const double* begin() const{
        return elem;
    }
    const double* end() const{
        return elem+size;
    }

}
```

```
class vector{
public:
  vector();
  vector(int s);
  ~vector();
  vector(const vector &v);
  vector& operator=(const vector&v);
  vector (vector&& v);
  vector& operator=(vector&& v);
  double operator[] (int pos) const;
  double& operator[] (int pos);
  int length() const;
  double* begin();
  double* end();
  const double* begin() const;
  const double* end() const;
}
```


Zwischenstand

```
vector Natural(int from, int to){
  vector v(to-from+1);
  for (auto& x: v) x = from++;
  return v;
}

int main(){
  vector v = Natural(5,12);
  for (auto x: v)
    std::cout << x << " "; // 5 6 7 8 9 10 11 12
  std::cout << "\n";
  std::cout << "sum="
            << std::accumulate(v.begin(), v.end(),0); // sum = 68
  return 0;
}
```


Nützliche Tools (3): `using` (C++11)

`using` ersetzt in C++11 das alte `typedef`.

```
using identifier = type-id;
```

Beispiel

```
using element_t = double;
class vector{
        std::size_t size;
        element_t* elem;
...
}
```

7. Sortieren I

Einfache Sortierverfahren

7.1 Einfaches Sortieren

Sortieren durch Auswahl, Sortieren durch Einfügen, Bubblesort
[Ottman/Widmayer, Kap. 2.1, Cormen et al, Kap. 2.1, 2.2, Exercise 2.2-2, Problem 2-2

Problemstellung

Eingabe: Ein Array $A = (A[1], ..., A[n])$ der Länge n.

Ausgabe: Eine Permutation A' von A, die sortiert ist: $A'[i] \leq A'[j]$ für alle $1 \leq i \leq j \leq n$.

Algorithmus: IsSorted(A)

Input : Array $A = (A[1], ..., A[n])$ der Länge n.
Output : Boolesche Entscheidung "sortiert" oder "nicht sortiert"

```
for i ← 1 to n − 1 do
    if A[i] > A[i + 1] then
        return "nicht sortiert";
return "sortiert";
```

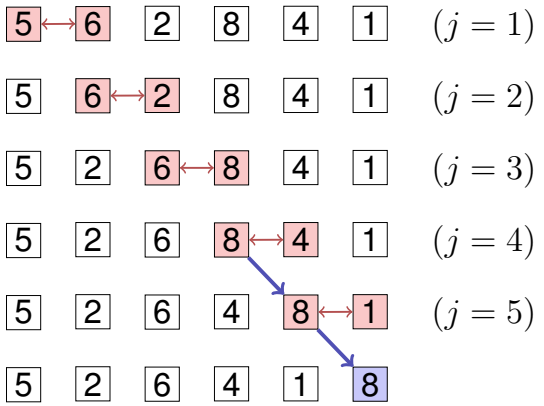
Beobachtung

IsSorted(A):“nicht sortiert”, wenn $A[i] > A[i+1]$ für ein i.

$\Rightarrow$ Idee:

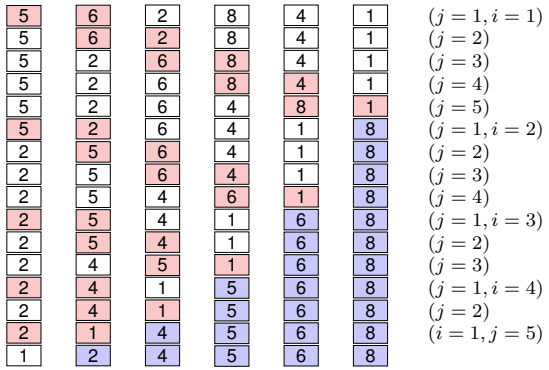
for $j \leftarrow 1$ **to** $n-1$ **do**
  if $A[j] > A[j+1]$ **then**
    swap$(A[j], A[j+1])$;

Ausprobieren



- Nicht sortiert! ☹.
- Aber das grösste Element wandert ganz nach rechts.
  $\Rightarrow$ Neue Idee! ☺

Ausprobieren



- Wende das Verfahren iterativ an.
- Für $A[1,\ldots,n]$, dann $A[1,\ldots,n-1]$, dann $A[1,\ldots,n-2]$, etc.

Algorithmus: Bubblesort

Input : Array $A = (A[1],\ldots,A[n])$, $n \geq 0$.
Output : Sortiertes Array A
for $i \leftarrow 1$ **to** $n-1$ **do**
  for $j \leftarrow 1$ **to** $n-i$ **do**
    if $A[j] > A[j+1]$ **then**
      swap$(A[j], A[j+1])$;

Analyse

Anzahl Schlüsselvergleiche $\sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2} = \Theta(n^2)$.

Anzahl Vertauschungen im schlechtesten Fall: $\Theta(n^2)$

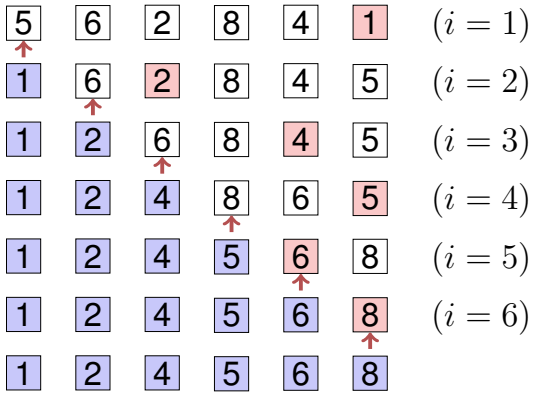
? Was ist der schlechteste Fall?

! Wenn A absteigend sortiert ist.

? Algorithmus kann so angepasst werden, dass er dann abbricht, wenn das Array sortiert ist. Schlüsselvergleiche und Vertauschungen des modifizierten Algorithmus im besten Fall?

! Schlüsselvergleiche = $n-1$. Vertauschungen = 0.

Sortieren durch Auswahl



- Iteratives Vorgehen wie bei Bubblesort.
- Auswahl des kleinsten (oder grössten) Elementes durch direkte Suche.

Algorithmus: Sortieren durch Auswahl

Input : Array $A = (A[1], \ldots, A[n])$, $n \geq 0$.
Output : Sortiertes Array A

```
for i ← 1 to n − 1 do
    p ← i
    for j ← i + 1 to n do
        if A[j] < A[p] then
            p ← j;
    swap(A[i], A[p])
```

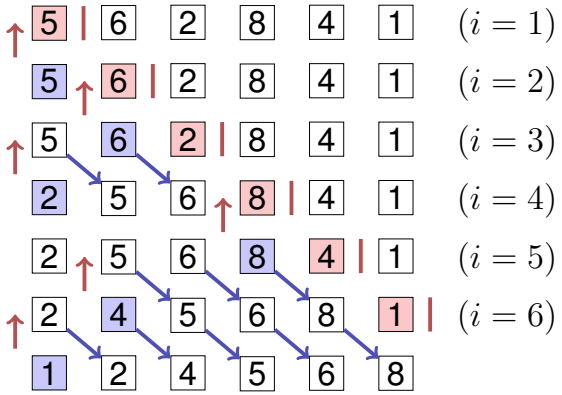

Analyse

Anzahl Vergleiche im schlechtesten Fall: $\Theta(n^2)$.

Anzahl Vertauschungen im schlechtesten Fall: $n-1 = \Theta(n)$

Anzahl Vergleiche im besten Fall: $\Theta(n^2)$.

Sortieren durch Einfügen



- Iteratives Vorgehen: $i = 1...n$
- Einfügeposition für Element i bestimmen.
- Element i einfügen, ggfs. Verschiebung nötig.

Sortieren durch Einfügen

? Welchen Nachteil hat der Algorithmus im Vergleich zum Sortieren durch Auswahl?

! Im schlechtesten Fall viele Elementverschiebungen.

? Welchen Vorteil hat der Algorithmus im Vergleich zum Sortieren durch Auswahl?

! Der Suchbereich (Einfügebereich) ist bereits sortiert. Konsequenz: binäre Suche möglich.

Algorithmus: Sortieren durch Einfügen

Input : Array $A = (A[1], \ldots, A[n])$, $n \geq 0$.
Output : Sortiertes Array A

```
for i ← 2 to n do
    x ← A[i]
    p ← BinarySearch(A[1...i − 1], x); // Kleinstes p ∈ [1, i] mit A[p] ≥ x
    for j ← i − 1 downto p do
        A[j + 1] ← A[j]
    A[p] ← x
```


Analyse

Anzahl Vergleiche im schlechtesten Fall:
$\sum_{k=1}^{n-1} a \cdot \log k = a \log((n-1)!) \in \mathcal{O}(n \log n)$.

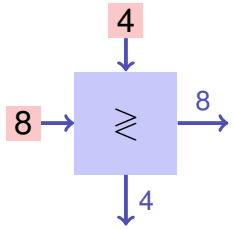
Anzahl Vergleiche im besten Fall: $\Theta(n \log n)$.[4]

Anzahl Vertauschungen im schlechtesten Fall: $\sum_{k=2}^{n}(k-1) \in \Theta(n^2)$

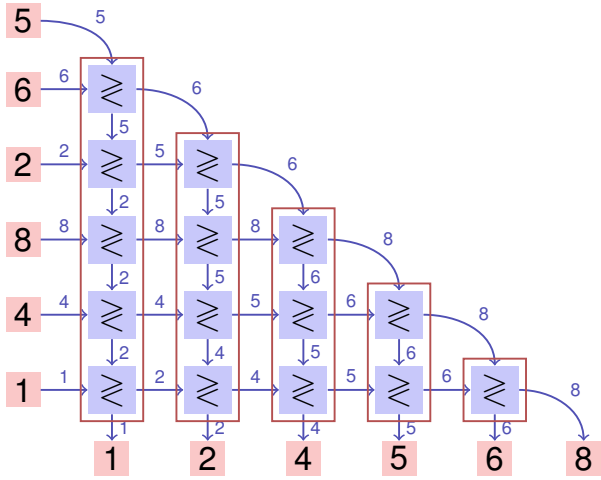
[4] Mit leichter Anpassung der Funktion BinarySearch für das Minimum / Maximum: $\Theta(n)$

Anderer Blickwinkel

Sortierknoten:

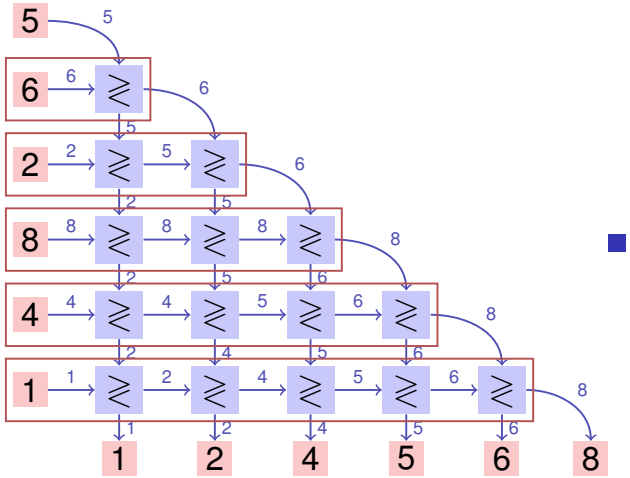


Anderer Blickwinkel



- Wie Selection Sort [und wie Bubble Sort]

Anderer Blickwinkel



- Wie Insertion Sort

Schlussfolgerung

Selection Sort, Bubble Sort und Insertion Sort sind in gewissem Sinne dieselben Sortieralgorithmen. Wird später präzisiert. [5]

[5] Im Teil über parallele Sortiernetzwerke. Für sequentiellen Code gelten natürlich weiterhin die zuvor gemachten Feststellungen.

Shellsort

Insertion Sort auf Teilfolgen der Form $(A_{k \cdot i})$ $(i \in \mathbb{N})$ mit absteigenden Abständen k. Letzte Länge ist zwingend $k = 1$.

Gute Folgen: z.B. Folgen mit Abständen $k \in \{2^i 3^j | 0 \leq i, j\}$.

Shellsort

9	8	7	6	5	4	3	2	1	0	
1	8	7	6	5	4	3	2	9	0	insertion sort, $k = 4$
1	0	7	6	5	4	3	2	9	8	
1	0	3	6	5	4	7	2	9	8	
1	0	3	2	5	4	7	6	9	8	
1	0	3	2	5	4	7	6	9	8	insertion sort, $k = 2$
1	0	3	2	5	4	7	6	9	8	
0	1	2	3	4	5	6	7	8	9	insertion sort, $k = 1$