6. C++ advanced (I)

Repetition: vectors, pointers and iterators, range for, keyword auto, a class for vectors, subscript-operator, move-construction, iterators

We look back...

```
#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 << " ";
}
```

We want to understand this in depth!

At least this is too pedestrian

Useful tools (1): auto (C++11)

The keyword auto:

The type of a variable is inferred from the initializer.

Examples

```
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 << " ";
  }
}
```

Useful tools (2): range `for` (C++11)

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

range-declaration: named variable of element type specified via the sequence in range-expression

range-expression: Expression that represents a sequence of elements via iterator pair `begin()`, `end()` or in the form of an intializer list.

Examples

```
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;
```


That is indeed 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 << " ";
}
```


For our detailed understanding

We build a vector class with the same capabilities ourselves!

On the way we learn about

- RAII (Resource Acquisition is Initialization) and move construction
- Index operators and other utilities
- Templates
- Exception Handling
- Functors and lambda expressions

A class for vectors

```
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
}
```

Element access

```
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;
}
```


What's the problem here?

```
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;
}
```

Now it is correct, but cumbersome.

More elegant this way:

```
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;
}
```


Syntactic sugar.

Getters and setters are poor. We want an index operator.

Overloading! So?

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

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

Nein!

Reference types!

```
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;
}
```


So far so good.

```
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;
}
```

Number copies

How often is `v` being copied?

```
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); // allocation of elems[16]
    v = v + 1;   // copy when assigned!
    return 0;    // deconstruction of v
}
```

`v` is copied twice

Move construction and move assignment

```
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;
}
```


Explanation

When the source object of an assignment will not continue existing after an assignment the compiler can use the move assignment instead of the assignment operator.[3] A potentially expensive copy operations is avoided this way.

Number of copies in the previous example goes down to 1.

[3] Analogously so for the copy-constructor and the move constructor

Range `for`

We wanted this:

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

In order to support this, an iterator must be provided via `begin` and `end` .

Iterator for the vector

```
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 for the vector

```
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;
}
```


Intermediate result

```
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;
}
```


Useful tools (3): `using` (C++11)

`using` replaces in C++11 the old `typedef`.

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

Beispiel

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

7. Sorting I

Simple Sorting

7.1 Simple Sorting

Selection Sort, Insertion Sort, Bubblesort [Ottman/Widmayer, Kap. 2.1, Cormen et al, Kap. 2.1, 2.2, Exercise 2.2-2, Problem 2-2

Problem

Input: An array $A = (A[1], ..., A[n])$ with length n.

Output: a permutation A' of A, that is sorted: $A'[i] \leq A'[j]$ for all $1 \leq i \leq j \leq n$.

Algorithm: IsSorted(A)

Input : Array $A = (A[1], ..., A[n])$ with length n.
Output : Boolean decision "sorted" or "not sorted"

```
for i ← 1 to n − 1 do
    if A[i] > A[i + 1] then
        return "not sorted";
return "sorted";
```

Observation

IsSorted(A):“not sorted”, if $A[i] > A[i+1]$ for an i.

$\Rightarrow$ idea:

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


Give it a try

5 ⟷	6	2	8	4	1	$(j = 1)$
5	6 ⟷	2	8	4	1	$(j = 2)$
5	2	6 ⟷	8	4	1	$(j = 3)$
5	2	6	8 ⟷	4	1	$(j = 4)$
5	2	6	4	8 ⟷	1	$(j = 5)$
5	2	6	4	1	8	

- Not sorted! ☹.
- But the greatest element moves to the right
  $\Rightarrow$ new idea! ☺

Try it out

5	6	2	8	4	1	$(j = 1, i = 1)$
5	6	2	8	4	1	$(j = 2)$
5	2	6	8	4	1	$(j = 3)$
5	2	6	8	4	1	$(j = 4)$
5	2	6	4	8	1	$(j = 5)$
5	2	6	4	1	8	$(j = 1, i = 2)$
2	5	6	4	1	8	$(j = 2)$
2	5	6	4	1	8	$(j = 3)$
2	5	4	6	1	8	$(j = 4)$
2	5	4	1	6	8	$(j = 1, i = 3)$
2	5	4	1	6	8	$(j = 2)$
2	4	5	1	6	8	$(j = 3)$
2	4	1	5	6	8	$(j = 1, i = 4)$
2	4	1	5	6	8	$(j = 2)$
2	1	4	5	6	8	$(i = 1, j = 5)$
1	2	4	5	6	8	

- Apply the procedure iteratively.
- For $A[1, \dots, n]$,
  then $A[1, \dots, n-1]$,
  then $A[1, \dots, n-2]$,
  etc.

Algorithm: Bubblesort

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

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

Analysis

Number key comparisons $\sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2} = \Theta(n^2)$.

Number swaps in the worst case: $\Theta(n^2)$

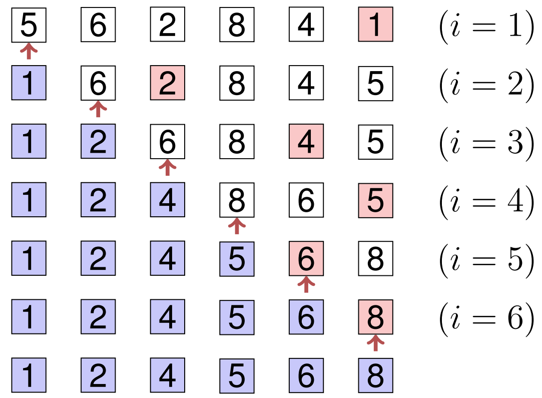
? What is the worst case?

! If A is sorted in decreasing order.

? Algorithm can be adapted such that it terminates when the array is sorted. Key comparisons and swaps of the modified algorithm in the best case?

! Key comparisons = $n-1$. Swaps = 0.

Selection Sort



- Iterative procedure as for Bubblesort.
- Selection of the smallest (or largest) element by immediate search.

Algorithm: Selection Sort

Input : Array $A = (A[1], \dots, A[n])$, $n \geq 0$.
Output : Sorted 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])
```

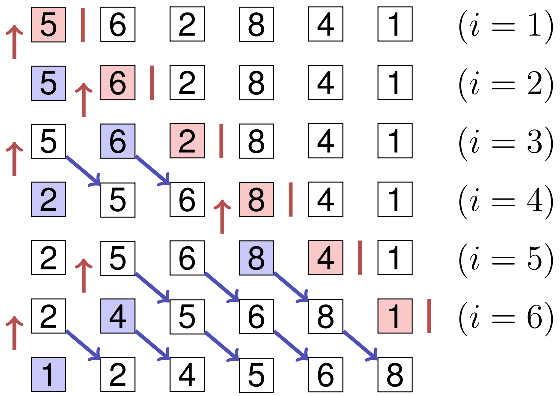

Analysis

Number comparisons in worst case: $\Theta(n^2)$.

Number swaps in the worst case: $n-1 = \Theta(n)$

Best case number comparisons: $\Theta(n^2)$.

Insertion Sort



- Iterative procedure: $i = 1...n$
- Determine insertion position für element i.
- Insert element i array block movement potentially required

Insertion Sort

? What is the disadvantage of this algorithm compared to sorting by selection?

! Many element movements in the worst case.

? What is the advantage of this algorithm compared to selection sort?

! The search domain (insertion interval) is already sorted. Consequently: binary search possible.

Algorithm: Insertion Sort

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

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


Analysis

Number comparisons in the worst case:
$\sum_{k=1}^{n-1} a \cdot \log k = a \log((n-1)!) \in \mathcal{O}(n \log n)$.

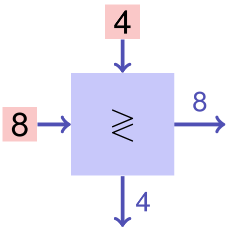
Number comparisons in the best case $\Theta(n \log n)$.[4]

Number comparisons in the worst case $\sum_{k=2}^{n} (k-1) \in \Theta(n^2)$

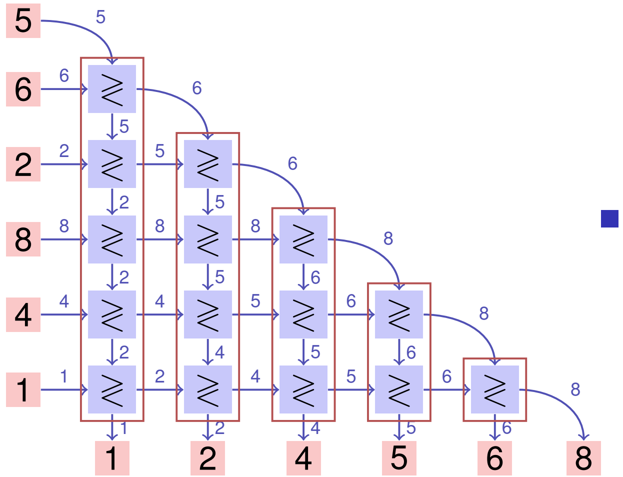
[4]With slight modification of the function BinarySearch fot eh minimum / maximum: $\Theta(n)$

Different point of view

Sortierknoten:

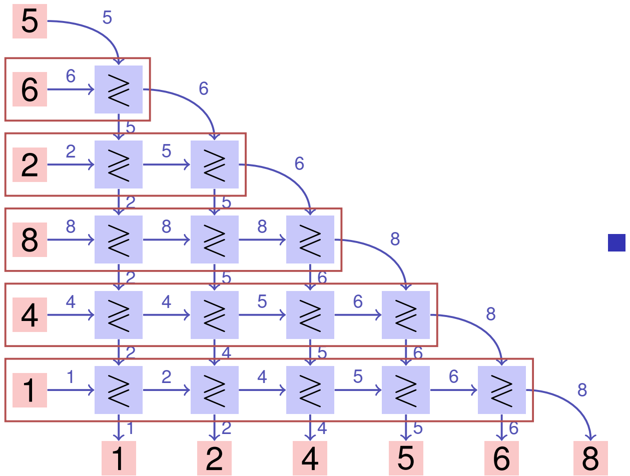


Different point of view



- Like selection sort
  [und like Bubblesort]

Different point of view



- Like insertion sort

Conclusion

In a certain sense, Selection Sort, Bubble Sort and Insertion Sort provide the same kind of sort strategy. Will be made more precise. [5]

[5] In the part about parallel sorting networks. For the sequential code of course the observations as described above still hold.

Shellsort

Insertion sort on subsequences of the form $(A_{k\cdot i})$ $(i \in \mathbb{N})$ with decreasing distances k. Last considered distance must be $k = 1$.

Good sequences: for example sequences with distances $k \in \{2^i 3^j | 0 \leq i, j\}$.

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, $k = 1$