Karlsruher Institut für Technologie

Institut für Technik der Informationsverarbeitung

Institutsleitung
Prof. Dr.-Ing. K. D. Müller-Glaser
Prof. Dr.-Ing. J. Becker
Prof. Dr. rer. nat. W. Stork

Übung06: Informationstechnik (IT)

Tobias Schwalb & Michael Tansella

Institut für Technik der Informationsverarbeitung (ITIV)

Teil2: Dateiverarbeitung und Sortieralgorithmen

Inhalt: Übung06

Karlsruher Institut für Technologie

1. Besprechung Übungsaufgaben 5.01 – 5.03
2. Dateiverarbeitung
3. Sortieralgorithmen

Streams und Dateiverarbeitung

- Problemstellung:
  - Wie kann ich große Dateimengen in mein Programm einlesen?
  - Wie kann ich Ergebnisse meiner Berechnungen dauerhaft abspeichern, ohne dieser per Hand abzuschreiben?
- Lösung: Lesen und Speichern aus bzw. in Dateien
  - Verwendung von Streams zur Dateiverarbeitung
- Vorteile
  - Keine umständliche / fehlerhafte Eingabe über die Tastatur
  - Einlesen / Abspeichern großer Datenmengen möglich
- Nachteile
  - Langsamer als Datenverarbeitung aus dem Speicher
  - Etwas komplexer in der Programmierung

Streams

- Streams werden auch bei der Ausgabe auf die Konsole bzw. dem Einlesen von der Konsole verwendet
  - Beispiel:

```
cout << "Zahlx: " << endl;
int zahlx;
cin >> zahlx;
```

- Entwickler kann direkt in Streams schreiben / lesen, wenn diese eingerichtet sind
  - Muss sich weder darum kümmern, wie die Daten letztendlich dort ankommen noch wo genau sie hinsollen
  - Man unterscheidet zwischen Input- und Output-Streams

Dateiaufbau

- Eine Datei in C++ ist wie ein langes Byte-Array
  - Strukturierung ist die Aufgabe des Programmierers
  - Bsp.: Trennung einzelner Daten durch Komma in einer Zeile

- Prinzip: Sequentieller Dateizugriff
  - Daten werden nacheinander gelesen - von oben nach unten
  - Sprünge innerhalb einer Datei sind möglich, aber komplex

- Zugriff auf Dateien über File-Stream Bibliothek **`<fstream>`**
  - Klassen vorhanden → komfortable Handhabung, standardisiert
  - Es stehen nahezu die gleichen Methoden, Operatoren und Manipulatoren wie bei **`cout`** und **`cin`** zur Verfügung

Datei öffnen

- Vor dem Schreiben / Lesen muss die Datei geöffnet werden

- Es müssen Dateiname (+ evtl. Pfad) und Modus angeben werden
  - Ohne Pfad muss die Datei im aktuellen Verzeichnis liegen

- Modus beim Öffnen mit Flags angeben
  - Lesen: **`fstream myFile( "test.dat", ios::in );`**
  - Schreiben: **`fstream yourFile( "neu.dat", ios::out );`**
  - Weitere (auch gemischte) Modi möglich – siehe nächste Folie

- Beim Öffnen einer Datei zum Schreiben wird eine neue Datei angelegt, wenn diese nicht vorhanden ist. Bei einer vorhandenen Datei wird diese überschrieben - Ohne Nachfrage!

Öffnungsmodi

Flag	Wirkung
`ios::in`	Eine bereits existierende Datei wird zum Lesen geöffnet
`ios::out`	Eine Datei zum schreiben öffnen. Dieses Flag impliziert `ios::trunc`, falls es nicht mit einem der Flags `ios::in` oder `ios:app` oder `ios::ate` kombiniert wird
`ios::app`	Vor jeder Schreiboperation wird auf das Dateiende positioniert
`ios::trunc`	Eine bereits bestehende Datei wird beim öffnen der Datei auf die Länge 0 gekürzt
`ios::ate`	Schreib-/Leseposition beim Öffnen der Datei auf das Dateiende setzen. Ohne dieses Flag ist die anfängliche Dateiposition stets der Dateianfang
`ios::binary`	Schreib- / Leseoperationen im Binärmodus durchführen

Es können auch mehrere Flags kombiniert werden durch binäres ODER: |

Methoden / Fehlerbehandlung (1)

- Bevor Daten gelesen / geschrieben werden, muss überprüft werden, ob das Öffnen der Datei erfolgreich war

- Fehlerbehandlung beim Öffnen (lesen / schreiben)
  - Datei existiert nicht / keine Zugriffsrechte / …
  - Beispiel:

```
if( !myFile ) {
  cout << "Datei konnte nicht geoeffnet werden";
  exit( 1 );
}
```

oder: `if( myFile.fail() )`

Programm sofort beenden

- Abfrage, ob eine Datei geöffnet ist mit Methode `is_open()`
  - Beispiel:

```
if( myFile.is_open() ) { //... }
```

Methoden / Fehlerbehandlung (2)

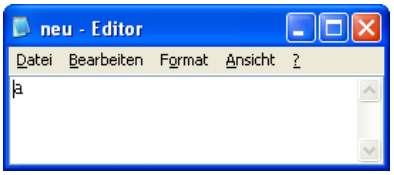
- Eine Datei kann nicht hinter dem Dateiende gelesen werden
- Abfrage für Dateiende mit **`eof()`**
  - Beispiel: `if( myFile.eof() ) { //... }`

- Zurück an den Dateianfang gehen während die Datei geöffnet ist
  - Beispiel:
```
myFile.seekg( 0, ios_base::beg );
myFile.clear();
```

- Nach der Bearbeitung sollte die Datei geschlossen werden
  - Keine Daten gehen verloren
  - Anzahl geöffneter Dateien begrenzt
  - Dateien werden am Programmende automatisch geschlossen
  - Beispiel: `myFile.close();`

Schreiben in Dateien

- Schreiben eines einzelnen Zeichens in eine Datei
  - Beispiel:
```
char c = 'a';
yourFile.put(c);
```



- Schreiben direkt von einer Variable
  - Auch formatierte Ausgabe möglich – Methoden / Operationen wie bei der Ausgabe auf dem Bildschirm
  - Beispiel:
```
double preis = 11.125;
yourFile << preis;
```

- Schreiben von komplexeren Objekten
  - Jedes Attribut einzeln schreiben
  - Schreiben der Objekte im Binär-Format bei bekannter Speichergröße

Lesen aus Dateien

- Lesen eines einzelnen Zeichens aus einer Datei
  - Beispiel:
```
char c;
myFile.get(c);
```
- Lesen direkt in eine Variable
  - Achtung: Format muss stimmen
  - Beispiel:
```
double preis;
myFile >> preis;
```
- Lesen einer Zeile als ASCII-String aus der Datei
  - Oft am Besten geeignet
  - Beispiel:
```
string zeile = "";
getline( myFile, zeile );
```
    Rückgabewert von **`getline()`** gibt an, ob das Lesen erfolgreich war

Lesen und Schreiben im Binär-Format

```
fstream& Konto::kontWrite( fstream& os ) {
  os << name << '\0'; // String schreiben
  os.write( (char*) &nr, sizeof( nr ) );
  os.write( (char*) &stand, sizeof( stand ) );
  return os;
}

fstream& Konto::kontRead( fstream& is ) {
  getline( is, name, '\0' ); //String einlesen
  is.read( (char*) &nr, sizeof( nr ) );
  is.read( (char*) &stand, sizeof( stand ) );
  return is;
}
```

```
class Konto {
  private:
    string name;
    unsigned long nr;
    double stand;

  public:
    Konto( string, unsigned long, double );
    fstream& kontRead( fstream& is );
    fstream& kontWrite( fstream& os );
    void setName( string _name );
    void setNr( unsigned long _nr );
    void setStand( double _stand );
};
```

schreibt die Daten des Kontos in den Stream **`os`**, Return-Wert: **`true`**, falls erfolgreich, sonst **`false`**

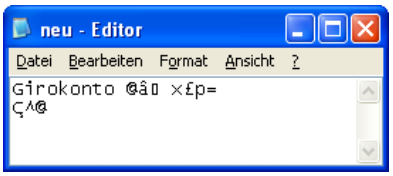
liest die Daten eines Kontos aus dem Stream **`is`** und überschreibt damit die Daten im aktuellen Objekt

- Aufpassen bei der Verwendung von **`write()`** und **`read()`** in Bezug auf Zeiger → keine Adressen Speichern!
  - Adressen und Speicherorte verändern sich

Binär-Format Beispiel

```
Konto::Konto( string _name,
              unsigned long _nr,
              double _stand ) {
  name = _name;
  nr = _nr;
  stand = _stand;
}

int main() {
  Konto kt( "Girokonto", 123456, 123.11 );

  fstream yourFile( "neu.dat", ios::out );
  kt.kontwrite( yourFile );
  yourFile.close();

  kt.setName( "" );
  kt.setNr( 0 );
  kt.setStand( 0 );

  fstream myFile( "neu.dat", ios::in );
  kt.kontRead( myFile );
  myfile.close();

  return 0;
}
```

```
class Konto {
  private:
    string name;
    unsigned long nr;
    double stand;

  public:
    Konto( string, unsigned long, double );
    fstream& kontRead( fstream& is );
    fstream& kontWrite( fstream& os );
    void setName( string _name );
    void setNr( unsigned long _nr );
    void setStand( double _stand );
};
```



Zwischenübung16: Dateiverarbeitung

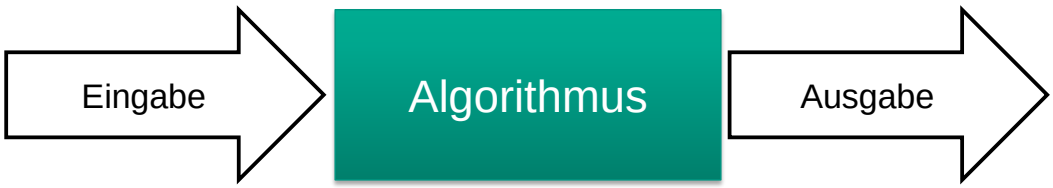
Wenn man eine Datei zum Schreiben öffnet, wird eine vorhandene Datei ohne Nachfrage überschrieben und damit gelöscht. Wie kann man dies verhindern bzw. den User fragen (wenn die Datei schon vorhanden ist), ob er diese Datei überschreiben möchte?

Zeichnen Sie hierzu ein Nassi-Schneiderman-Diagramm – kein C++ Code

Algorithmen

- Genau definierte Handlungsvorschrift zur Lösung eines Problems oder einer bestimmten Art von Problemen in endlich vielen Schritten

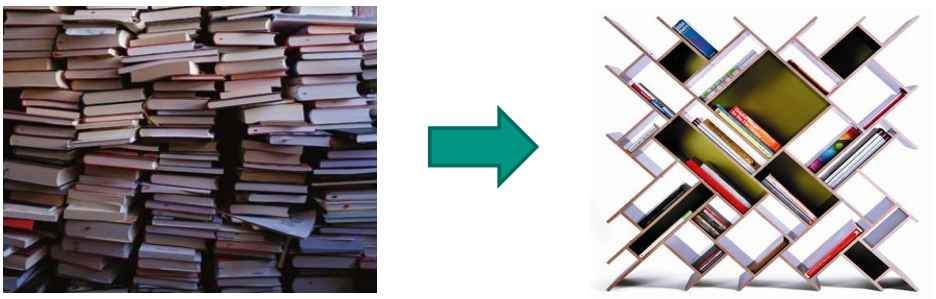
- Wichtige Problemstellungen
  - Sortieren
  - Suchen
  - Optimieren



Sortieren

- Problemstellung
  - Wie kann ich Datensätze sortieren / sortiert ausgeben?
  - Wie kann ich Personen nach Einkommen sortieren?
  - Wie kann ich Daten sortieren, um diese schneller zu verarbeiten?

- Lösung: Sortieralgorithmen
  - Sortieren eines Arrays oder einer Liste
  - Verschiedene Algorithmen
    - Unterschiedliche Geschwindigkeit, Rechen- und Speicherbedarf



BubbleSort

- Einfacher Algorithmus – langsam, aber kaum zusätzlicher Speicherplatz

- Von Pseudocode zu C++ Quellcode

```
0  A = [5, 3, 1, 4, 2]
1  for i = 1 to länge[A] – 1
2  do for j = 1 to länge[A] - i
3     do if A[j] > A[j+1]
4        then vertausche A[j] mit A[j+1]
```

```
0  int A[] = {5, 3, 1, 4, 2};
1  for( int i = 0; i < 4; i++ ) {
2    for( int j = 0; j < 4 - i; j++ ) {
3      if( A[j] > A[j+1] ) {
4        int temp = A[j];
5        A[j] = A[j+1];
6        A[j+1] = temp;
7      }
8    }
9  }
```

```
C:\WINDOWS\system32\cmd.exe
Array vorher:      5   3   1   4   2
Array nachher:     1   2   3   4   5
Drücken Sie eine beliebige Taste . . .
```


InsertionSort

- Sortieren durch Einfügen

- Relativ einfacher Algorithmus

- Vorteile
  - Effizient bei kleinen & bei schon vorsortierten Eingabemengen
  - Einfache Implementierung
  - Stabil
  - Minimal im Speicherverbrauch

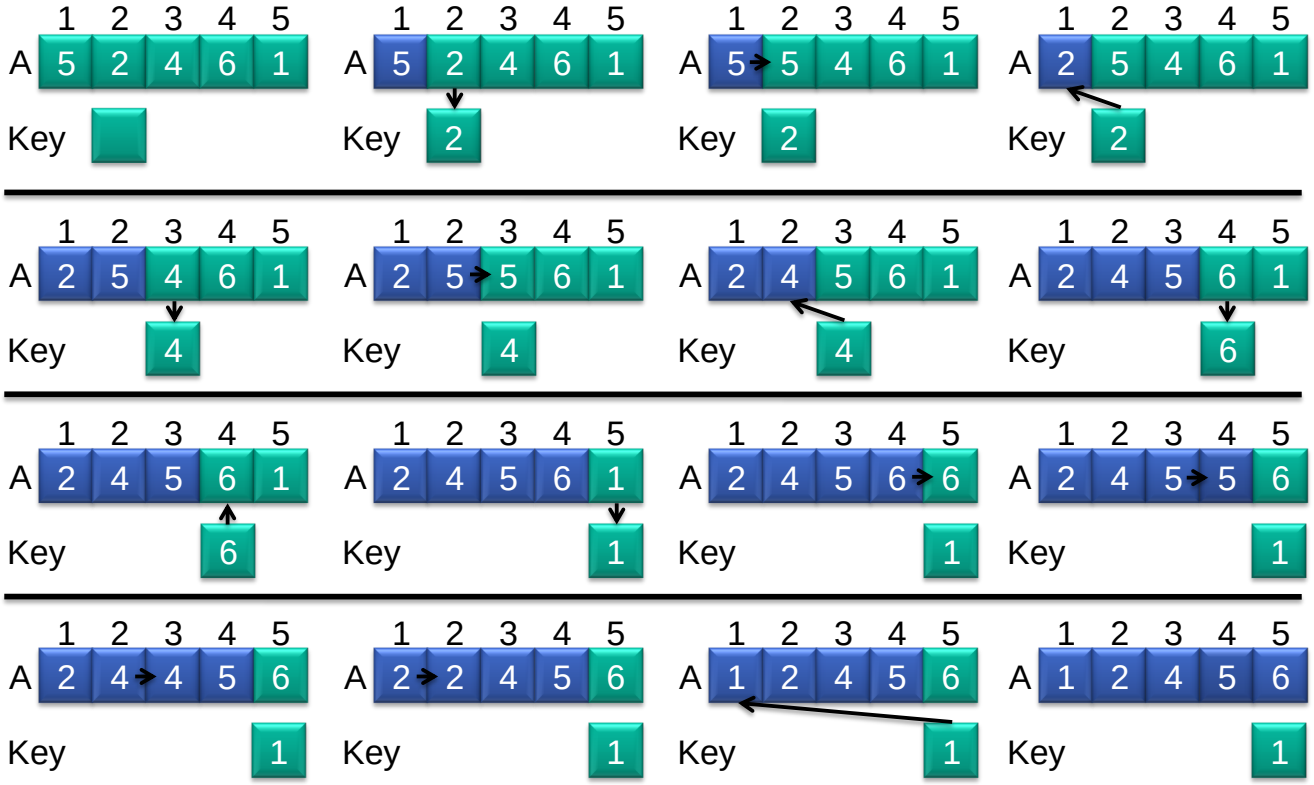
- Nachteile
  - Weniger effizient wie komplizierte Sortierverfahren

Link zu Demonstration:
http://einstein.informatik.uni-oldenburg.de/20907.html

InsertionSort

```
for ( j = 2 to length(A) ) do
    key = A[j]
    i = j - 1
    while ( i > 0 and A[i] > key ) do
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = key
```

InsertionSort Ablauf



QuickSort

- Schneller, rekursiver, nicht-stabiler Sortieralgorithmus nach dem Prinzip „Teile und Herrsche“
- Ca. 1960 von C. Antony R. Hoare in seiner Grundform entwickelt und seitdem von vielen Forschern verbessert
- Verfügt über eine sehr kurze innere Schleife (was die Ausführungsgeschwindigkeit stark erhöht)
- Kommt ohne zusätzlichen Speicherplatz aus (abgesehen von dem Platz auf dem Aufruf-Stack für die Rekursion)
- Im Mittel schnellster Sortieralgorithmus $O(n \log_2 n)$ im schlechtesten Fall $O(n^2)$

QuickSort Prinzip

- Drei Schritte für das Sortieren eines Feldes A[p ... r]:

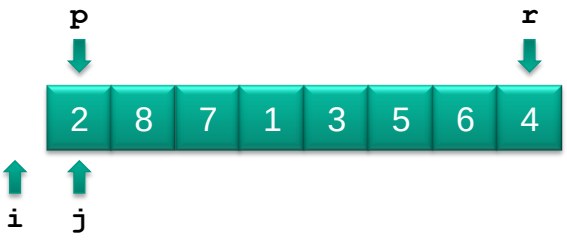
a) Teile: Zerlege das Feld `A[p..r]` so in zwei (möglicherweise leere) Teilfelder `A[p..q-1]` und `A[q+1..r]`. Dabei soll jedes Element von `A[p..q-1]` kleiner oder gleich `A[q]` und jedes Element von `A[q+1..r]` größer als `A[q]` sein. `A[q]` ist dabei ein beliebiges Element des Feldes. (Somit sind alle Werte von `A[p..q-1]` auch kleiner wie die Werte in `A[q+1..r]`.)

b) Beherrsche: Sortiere die beiden Teilfelder `A[p..q-1]` und `A[q+1..r]` wieder durch rekursiven Aufruf von QuickSort (Beginn wieder beim Teilen), wenn die Teilfelder mehr als ein Element haben.

c) Verbinde: Da die Teilfelder in-place sortiert werden (in Feld selbst) ist keine Arbeit erforderlich, um sie zu verbinden. Das gesamte Feld `A[p..r]` ist nun sortiert.

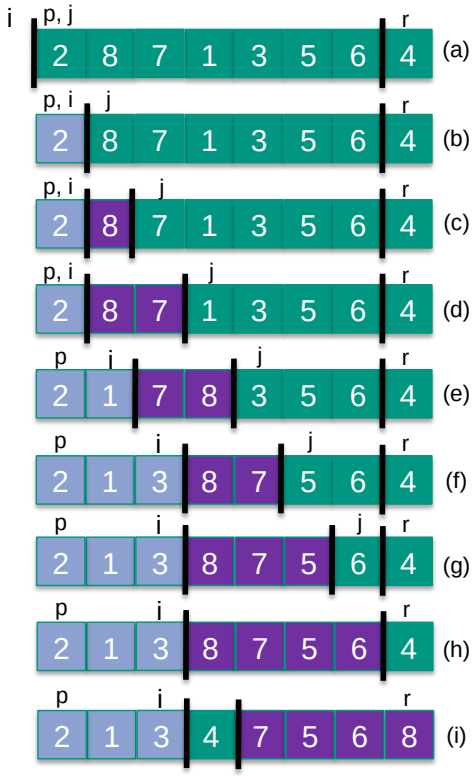
QuickSort Animation (siehe E-Studium)

```
Quicksort( A, 1, länge[A] )

Quicksort( A, p, r )
1     if p < r
2     then q = Partition( A, p, r )
3          Quicksort( A, p, q – 1 )
4          Quicksort( A, q + 1, r )

Partition( A, p, r )
1     x = A[r]
2     i = p - 1
3     for j = p to r – 1
4     do if A[j] <= x
5        then i = i + 1
6             vertausche A[i] mit A[j]
7     vertausche A[i+1] mit A[r]
8     return i+1
```



QuickSort Pseudocode

```
Quicksort( A, 1, länge[A] )

Quicksort( A, p, r )
1  if p < r
2  then q = Partition( A, p, r )
3       Quicksort( A, p, q - 1 )
4       Quicksort( A, q + 1, r )

Partition( A, p, r )
1  x = A[r]
2  i = p - 1
3  for j = p to r - 1
4  do if A[j] <= x
5     then i = i + 1
6          vertausche A[i] mit A[j]
7  vertausche A[i+1] mit A[r]
8  return i+1
```



QuickSort Partition Erklärung

- Die Arbeitsweise von Partition angewendet auf ein Beispielfeld. Hellblau schattierte Feldelemente befinden sich alle in der ersten Partition, in der die Werte nicht größer als x sind. Lila schattierte Elemente befinden sich in der zweiten Partition mit Werten, die größer als sind als x. Die grün schattierten Elemente sind noch nicht in eine der beiden ersten Partitionen eingefügt worden. Das letzte Element ist das Pivoelement.

(a) Das Anfangsfeld und die gesetzten Parameter. Keines der Elemente ist in eine der beiden ersten Partitionen eingefügt worden.

(b) Der Wert 2 wird „mit sich selbst vertauscht“ und in die Partition mit den kleineren Werten eingefügt.

(c-d) Die Werte 8 und 7 werden der Partition mit den größeren Werten hinzugefügt.

(e) Die Werte 1 und 8 werden vertauscht und die kleinere Partition wächst.

(f) Die Werte 3 und 7 werden vertauscht und die kleinere Partition wächst.

(g-h) Die größere Partition wächst, um die Elemente 5 und 6 aufzunehmen. Anschließend terminiert die Schleife.

(i) In den Zeilen 7-8 wird das Pivoelement so eingefügt, dass es zwischen den beiden Partitionen liegt.

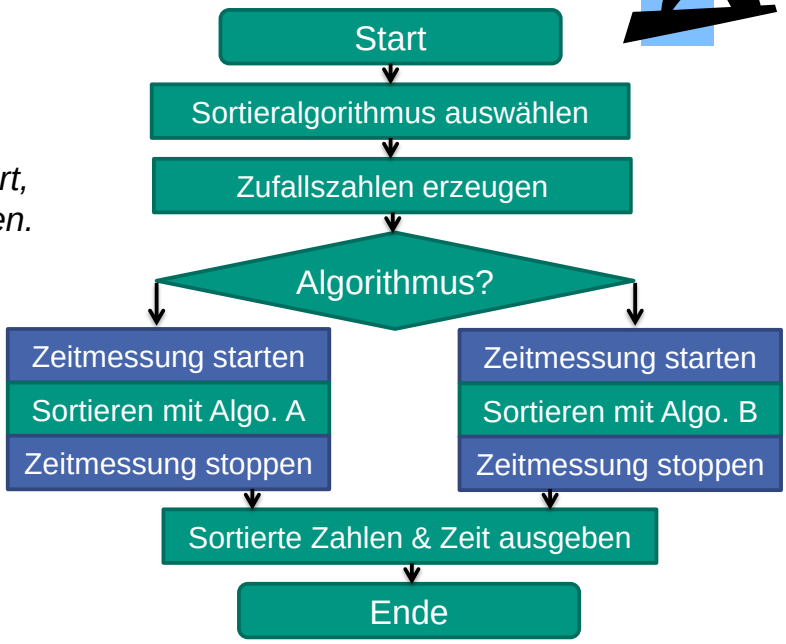
Link zu Demonstration: http://www.hermann-gruber.com/lehre/sorting/Quick/Quick.html

Zwischenübung17: Bewertung Alg.

Situationsbeschreibung:
Zwei Forscher treffen sich und möchten sich über die Geschwindigkeit von verschiedenen Sortieralgorithmen austauschen.

Forscher A hat hierzu ein Testprogramm geschrieben, welches er Forscher B als Ablaufdiagramm vorstellt und anschließend zweimal ausführt, um seine Ergebnisse zu zeigen.

Daraufhin Forscher B zu Forscher A …

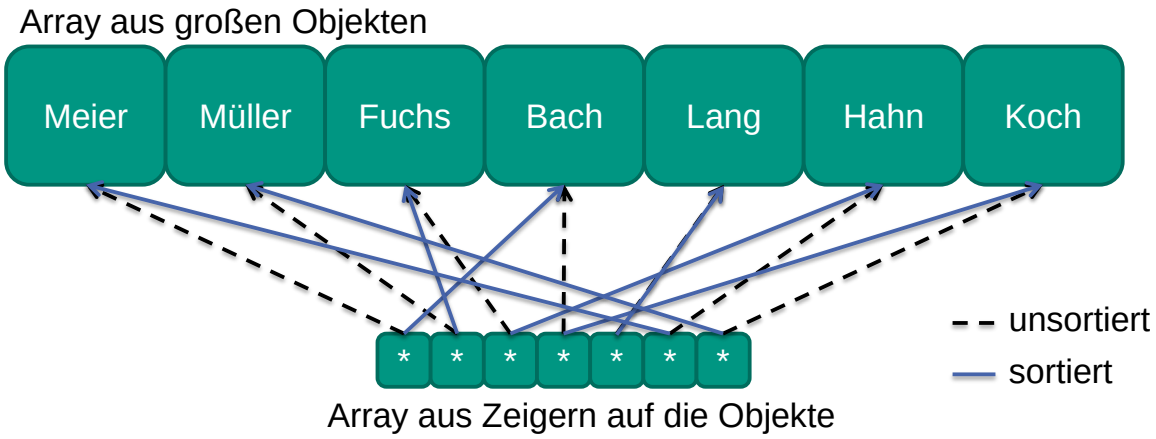


Link zu Demonstration: http://www.gf-webdesign.de/sortieralgorithmen/

Sortieren und Zeiger

- Problemstellung
  - Wie kann ich große Objekte effizient und schnell sortieren?
    Große Objekte = viele Attribute, wobei nach einem bestimmten Attribut aufsteigend oder absteigend sortiert werden soll
- Lösung: Sortieren von Zeigern auf Objekte
  - Man erstellt ein Array aus Zeigern, welche auf alle Objekte zeigen und sortiert nur diese Zeiger
- Vorteile
  - Wesentlich schnelleres sortieren bei großen Objekten, da nur Zeiger verschoben werden
  - Mit konstanten Zeiger können die Objekte vor einer ungewollten Veränderung geschützt werden (`const int* arr`)
- Nachteile
  - Es wird zusätzlicher Speicherplatz für die Zeiger benötigt
  - Komplexer in der Programmierung

Prinzip: Sortieren von Zeigern

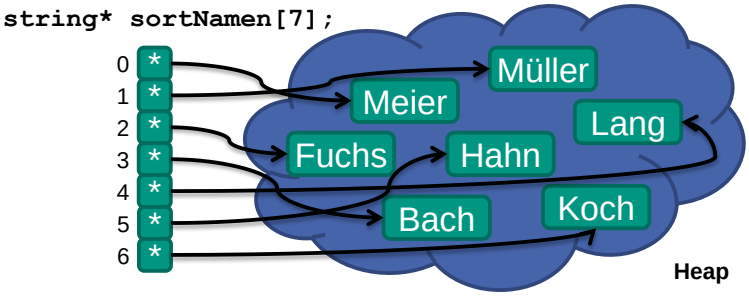


- Nur Veränderung der Zeiger, anstatt die großen Objekte im Array zu verschieben
- Für einzelne dynamisch erzeugte Objekte auf dem Heap ist kein anderes Sortierverfahren anwendbar

Beispiel: Sortieren dynamischer Objekte (1)

```
#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

class NamenSortieren {
private:
  string* sortNamen[7];
  int laenge;

public:
  void sortieren();
  void erzeugen();
  void ausgeben();
};
```

```
void NamenSortieren::erzeugen() {
  sortNamen[0] = new string( "Meier" );
  sortNamen[1] = new string( "Mueller" );
  sortNamen[2] = new string( "Fuchs" );
  sortNamen[3] = new string( "Bach" );
  sortNamen[4] = new string( "Lang" );
  sortNamen[5] = new string( "Hahn" );
  sortNamen[6] = new string( "Koch" );
  laenge = 7;
}
```



Beispiel: Sortieren dynamischer Objekte (2)

```
void NamenSortieren::sortieren() {
  string* temp = NULL;

  for( int i = 0; i < laenge - 1; i++ ) {
    for( int j = laenge - 1; j >= i + 1; j-- ) {
      if( *sortNamen[j] < *sortNamen[j-1] ) {
        temp = sortNamen[j];
        sortNamen[j] = sortNamen[j-1];
        sortNamen[j-1] = temp;
      }
    }
  }
}

void NamenSortieren::ausgeben() {
  cout << left;
  for( int i = 0; i < laenge; i++ )
    cout << setw( 10 ) << *sortNamen[i];

  cout << endl;
}
```

BubbleSort-Algorithmus zum Sortieren

Vergleich des Inhalts, worauf das Element im Zeigerarray zeigt

nur verändern der Zeiger im Zeigerarray (Adressen werden vertauscht)

Linksbündig ausgeben

Ausgabe der Namen über Dereferenzierung

Ausblick

- Wie kann ich einen bestimmten Knoten in einem Graphen finden?
- Wie kann ich die Laufzeit eines Algorithmus abschätzen?
- Wie kann man die Komplexität bzw. das Wachstumsgesetz eines Algorithmus bestimmen?
- Wie berechne ich die Laufzeit eines Algorithmus für worst- und best-case Szenarien?
- Wie kann man auch komplexe Probleme der Ordnung n! lösen?
- Wie kann man einen optimalen Weg finden, um n Städte in einer Rundreise nacheinander zu besuchen?
- …