Einführung in die Programmierung

Vorlesungsteil 7
Felder und Strukturierung von Daten

PD Dr. Thomas Hinze

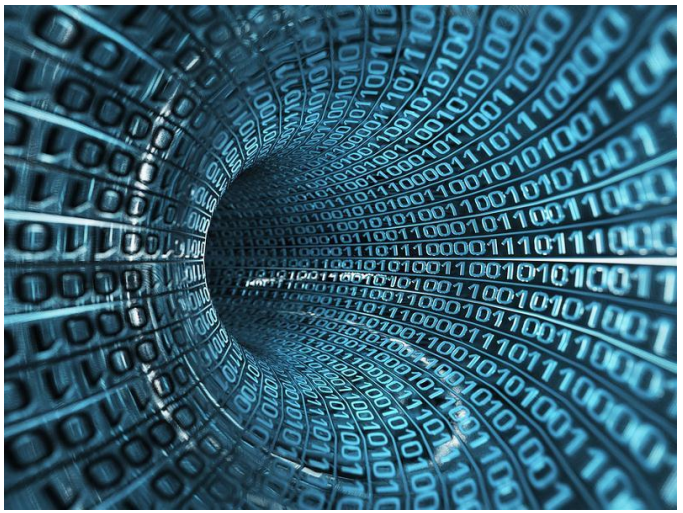
Brandenburgische Technische Universität Cottbus – Senftenberg
Institut für Informatik, Informations- und Medientechnik

Wintersemester 2015/2016

Brandenburgische
Technische Universität
Cottbus - Senftenberg

Große Datenmengen sind häufig zu verarbeiten

www.bund.de



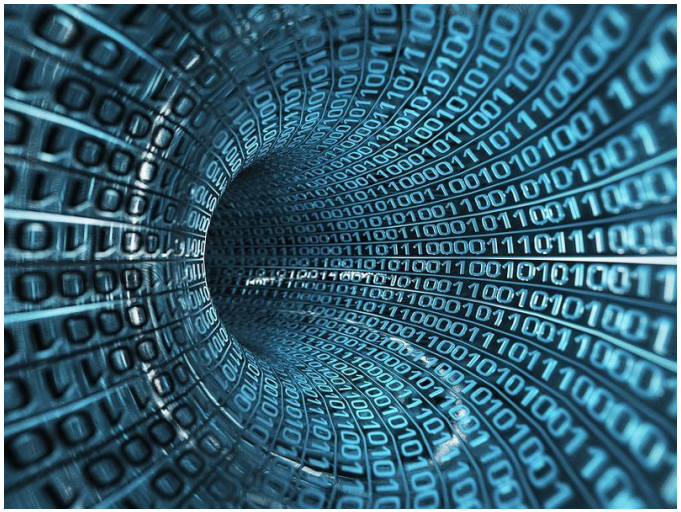
www.ingenieur.de

- Weltweites Datenvolumen verdoppelt sich etwa alle zwei Jahre*

*: Studie der UC Berkeley School of Information, 2013

Große Datenmengen sind häufig zu verarbeiten

www.bund.de



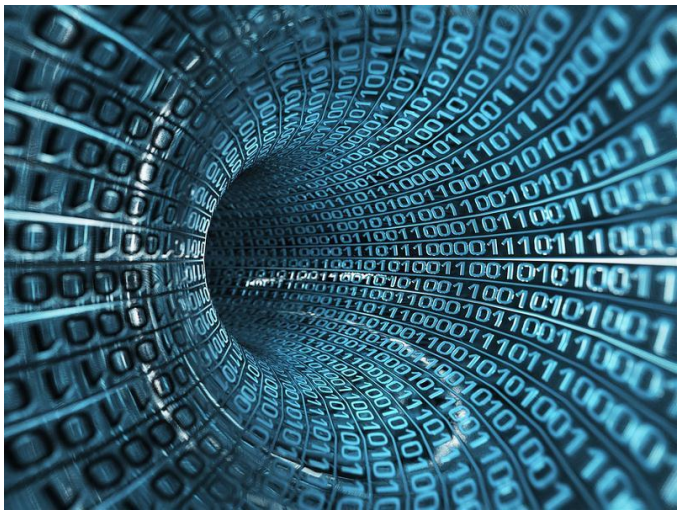
www.ingenieur.de

- Weltweites Datenvolumen verdoppelt sich etwa alle zwei Jahre*
- Weltweiter Datenbestand auf gegenwärtig etwa $5 \cdot 10^{21}$ Bytes (5 Zettabytes) geschätzt*

*: Studie der UC Berkeley School of Information, 2013

Große Datenmengen sind häufig zu verarbeiten

www.bund.de



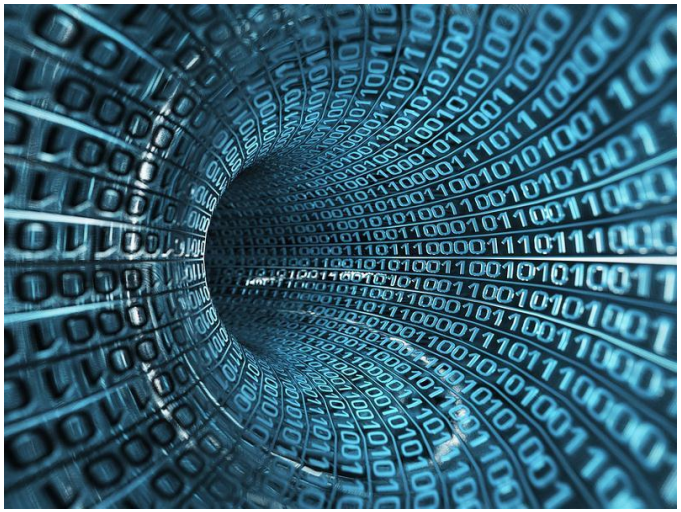
www.ingenieur.de

- Weltweites Datenvolumen verdoppelt sich etwa alle zwei Jahre*
- Weltweiter Datenbestand auf gegenwärtig etwa $5 \cdot 10^{21}$ Bytes (5 Zettabytes) geschätzt*
- Speicherkapazität des gesunden menschlichen Gehirns bei etwa 10^{15} Bytes (15 Petabytes) vermutet

* : Studie der UC Berkeley School of Information, 2013

Große Datenmengen sind häufig zu verarbeiten

www.bund.de



www.ingenieur.de

- Weltweites Datenvolumen verdoppelt sich etwa alle zwei Jahre*
- Weltweiter Datenbestand auf gegenwärtig etwa $5 \cdot 10^{21}$ Bytes (5 Zettabytes) geschätzt*
- Speicherkapazität des gesunden menschlichen Gehirns bei etwa 10^{15} Bytes (15 Petabytes) vermutet
- *Big Data* als vielversprechendes Forschungsgebiet

*: Studie der UC Berkeley School of Information, 2013

Umfangreiche Datenmengen effizient verarbeiten

Angenommen, Sie haben 12 Messwerte

```
4.31, 4.72, 4.49, 4.18, 4.07, 4.13,
4.56, 4.38, 4.71, 4.52, 4.64, 4.45
```

und wollen daraus den Durchschnitt (arithmetisches Mittel) berechnen.

Umfangreiche Datenmengen effizient verarbeiten

Angenommen, Sie haben 12 Messwerte

```
4.31, 4.72, 4.49, 4.18, 4.07, 4.13,
4.56, 4.38, 4.71, 4.52, 4.64, 4.45
```

und wollen daraus den Durchschnitt (arithmetisches Mittel) berechnen.

- Allein durch Nutzung elementarer Datentypen müssten wir dafür 12 *verschiedene Variablen* anlegen.

Umfangreiche Datenmengen effizient verarbeiten

Angenommen, Sie haben 12 Messwerte

```
4.31, 4.72, 4.49, 4.18, 4.07, 4.13,
4.56, 4.38, 4.71, 4.52, 4.64, 4.45
```

und wollen daraus den Durchschnitt (arithmetisches Mittel) berechnen.

- Allein durch Nutzung elementarer Datentypen müssten wir dafür 12 *verschiedene Variablen* anlegen.
- Die Implementierung der Formel zur Berechnung des Durchschnitts wäre *umständlich* hinzuschreiben.

Umfangreiche Datenmengen effizient verarbeiten

Angenommen, Sie haben 12 Messwerte

```
4.31, 4.72, 4.49, 4.18, 4.07, 4.13,
4.56, 4.38, 4.71, 4.52, 4.64, 4.45
```

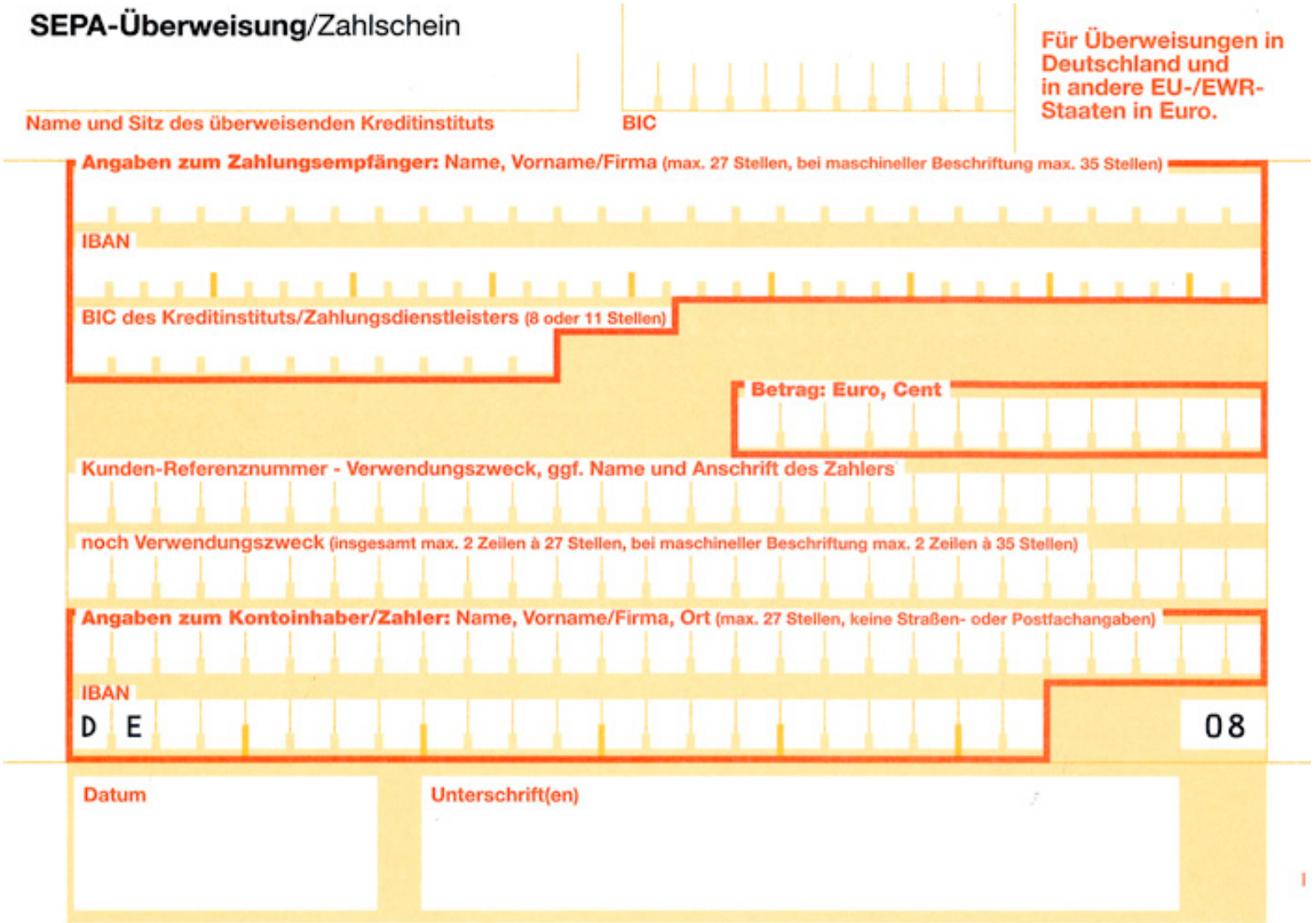
und wollen daraus den Durchschnitt (arithmetisches Mittel) berechnen.

- Allein durch Nutzung elementarer Datentypen müssten wir dafür 12 *verschiedene Variablen* anlegen.
- Die Implementierung der Formel zur Berechnung des Durchschnitts wäre *umständlich* hinzuschreiben.
- Angenommen, es gäbe hin und wieder nur 11 oder vielleicht auch einmal 13 Messwerte. Wir müssten dann jedesmal *umfangreiche Änderungen im Quelltext* vornehmen, um auf diese Situationen reagieren zu können.

Feld

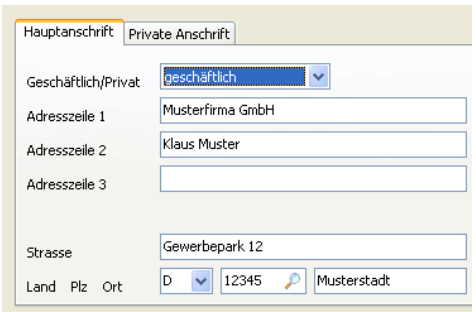
Ein *Feld* (engl. *array*) gibt uns die Möglichkeit, eine beliebig große, aber bekannte Anzahl von Datenwerten über einen einheitlichen Bezeichner zu erfassen und auf jeden einzelnen dieser Datenwerte direkt lesend oder schreibend zuzugreifen.

Ein Datensatz fasst Einzeldaten zu Einheit zusammen



Datensatz

Ein *Datensatz* (engl. *record*) gibt uns die Möglichkeit, mehrere zusammengehörende Einzeldatenwerte beliebiger Typen im Sinne einer Karteikarte oder eines Formulars zu bündeln und als gemeinsames Datenpaket zu behandeln. Dies vereinfacht programmiertechnisch die Übergabe zwischen Funktionen und erleichtert die Prüfung auf Vollständigkeit der Einzeldatenwerte.



Vorlesung Einführung in die Programmierung mit C

1. **Einführung und erste Schritte** . . . . . Installation C-Compiler, ein erstes Programm: HalloWelt, Blick in den Computer
2. **Elementare Datentypen, Variablen, Arithmetik, Typecast** . . . . . C als Taschenrechner nutzen, Tastatureingabe → Formelberechnung → Ausgabe
3. **Imperative Kontrollstrukturen** . . . . . Befehlsfolgen, Verzweigungen und Schleifen programmieren
4. **Aussagenlogik in C** . . . . . Schaltbelegungstabellen aufstellen, optimieren und implementieren
5. **Funktionen selbst programmieren** . . . . . Funktionen als wiederverwendbare Werkzeuge, Werteübernahme und -rückgabe
6. **Rekursion** . . . . . selbstaufrufende Funktionen als elegantes algorithmisches Beschreibungsmittel
7. **Felder und Strukturierung von Daten** . . . . . effizientes Handling größerer Datenmengen und Beschreibung von Datensätzen
8. **Sortieren** . . . . . klassische Sortierverfahren im Überblick, Laufzeit und Speicherplatzbedarf
9. **Zeiger, Zeichenketten und Dateiarbeit** . . . . . Texte analysieren, ver- und entschlüsseln, Dateien lesen und schreiben
10. **Dynamische Datenstruktur „Lineare Liste“** . . . . . unsere selbstprogrammierte kleine Datenbank
11. **Ausblick und weiterführende Konzepte** . . . . .

Eindimensionales Feld anlegen und initialisieren

(messwertfeld.c)

```c
#include <stdio.h>

#define N 12  //Anzahl Messwerte

int main(void)
{
  float messwert[N] = { 4.31, 4.72, 4.49, 4.18, 4.07, 4.13,
                        4.56, 4.38, 4.71, 4.52, 4.64, 4.45 };

  int i;

  for (i = 0; i < N; i++)
  {
    printf("%2d   %.2f\n", i, messwert[i]);
  }
  return 0;
}
```

```
 0   4.31
 1   4.72
 2   4.49
 3   4.18
 4   4.07
 5   4.13
 6   4.56
 7   4.38
 8   4.71
 9   4.52
10   4.64
11   4.45
```

Arithmetisches Mittel aus den Messwerten berechnen

(messwertfeld-durchschnitt.c)

```c
#include <stdio.h>

#define N 12  //Anzahl Messwerte

int main(void)
{
  float messwert[N] = { 4.31, 4.72, 4.49, 4.18, 4.07, 4.13,
                        4.56, 4.38, 4.71, 4.52, 4.64, 4.45 };

  int i;
  float summe = 0.0;

  for (i = 0; i < N; i++)
  {
    summe = summe + messwert[i];
  }
  printf("Durchschnitt der Messwerte: %f\n", summe / N);
  return 0;
}
```

```
Durchschnitt der Messwerte: 4.430000
```

Maximalen Messwert bestimmen

(messwertfeld-maximum.c)

```c
#include <stdio.h>

#define N 12  //Anzahl Messwerte

int main(void)
{
  float messwert[N] = { 4.31, 4.72, 4.49, 4.18, 4.07, 4.13,
                        4.56, 4.38, 4.71, 4.52, 4.64, 4.45 };

  int i;
  float max = messwert[0];

  for (i = 1; i < N; i++)
  {
    if (max < messwert[i])
    {
      max = messwert[i];
    }
  }
  printf("Groesster Messwert: %.2f\n", max);
  return 0;
}
```

```
Groesster Messwert: 4.72
```

Begriff Feld in der Programmierung

Ein **Feld** (engl. array) bezeichnet in der Programmierung eine *Zusammenfassung von Speicherplätzen* (Variablenwerten) *gleichen Typs*, die über einen oder mehrere *Indizes* angesprochen werden können.

Begriff Feld in der Programmierung

Ein **Feld** (engl. array) bezeichnet in der Programmierung eine *Zusammenfassung von Speicherplätzen* (Variablenwerten) *gleichen Typs*, die über einen oder mehrere *Indizes* angesprochen werden können.

- Die lückenlose Nummerierung der Feldelemente heißt *Index* (Plural: Indizes) und beginnt immer bei 0.

Begriff Feld in der Programmierung

Ein **Feld** (engl. array) bezeichnet in der Programmierung eine *Zusammenfassung von Speicherplätzen* (Variablenwerten) *gleichen Typs*, die über einen oder mehrere *Indizes* angesprochen werden können.

- Die lückenlose Nummerierung der Feldelemente heißt *Index* (Plural: Indizes) und beginnt immer bei 0.
- Der Index ist immer vom Ganzzahltyp und wird in eckige Klammern `[...]` geschrieben.

Begriff Feld in der Programmierung

Ein **Feld** (engl. array) bezeichnet in der Programmierung eine *Zusammenfassung von Speicherplätzen* (Variablenwerten) *gleichen Typs*, die über einen oder mehrere *Indizes* angesprochen werden können.

- Die lückenlose Nummerierung der Feldelemente heißt *Index* (Plural: Indizes) und beginnt immer bei 0.
- Der Index ist immer vom Ganzzahltyp und wird in eckige Klammern `[...]` geschrieben.
- Ein Feld lässt sich durch fortlaufend durchnummerierte Schubfächer veranschaulichen.

Begriff Feld in der Programmierung

Ein **Feld** (engl. array) bezeichnet in der Programmierung eine *Zusammenfassung von Speicherplätzen* (Variablenwerten) *gleichen Typs*, die über einen oder mehrere *Indizes* angesprochen werden können.

- Die lückenlose Nummerierung der Feldelemente heißt *Index* (Plural: Indizes) und beginnt immer bei 0.
- Der Index ist immer vom Ganzzahltyp und wird in eckige Klammern `[...]` geschrieben.
- Ein Feld lässt sich durch fortlaufend durchnummerierte Schubfächer veranschaulichen.
- Mathematische Vorbilder für Felder sind *Vektoren*, *endliche Zahlenfolgen* und *Matrizen*.

Begriff Feld in der Programmierung

Ein **Feld** (engl. array) bezeichnet in der Programmierung eine *Zusammenfassung von Speicherplätzen* (Variablenwerten) *gleichen Typs*, die über einen oder mehrere *Indizes* angesprochen werden können.

- Die lückenlose Nummerierung der Feldelemente heißt *Index* (Plural: Indizes) und beginnt immer bei 0.
- Der Index ist immer vom Ganzzahltyp und wird in eckige Klammern `[...]` geschrieben.
- Ein Feld lässt sich durch fortlaufend durchnummerierte Schubfächer veranschaulichen.
- Mathematische Vorbilder für Felder sind *Vektoren*, *endliche Zahlenfolgen* und *Matrizen*.
- Die Anzahl Feldelemente muss in C bereits beim Compilieren bekannt sein.

Feldelemente per Direktzugriff mit Werten belegen

(quadratzahlen.c)

```c
#include <stdio.h>

#define N 11 //Anzahl Feldelemente. Index: 0 ... N-1

int main(void)
{
  long quadratzahlen[N]; //Speicherplatz fuer Feld ausfassen
  int i;

  for (i = 0; i < N; i++)
  {
    quadratzahlen[i] = i * i;  //Feldelemente mit Werten belegen
  }
  for (i = 0; i < N; i++)
  {
    printf("%2d   %2ld\n", i, quadratzahlen[i]);
  }
  return 0;
}
```

```
 0    0
 1    1
 2    4
 3    9
 4   16
 5   25
 6   36
 7   49
 8   64
 9   81
10  100
```

Feld an eine Funktion übergeben

(quadratzahlen2.c)

```c
#include <stdio.h>

#define N 11 //Anzahl Feldelemente. Index: 0 ... N-1

void ausgabe(long feld[]) //operiert im Speicherbereich des Feldes quadratzahlen
{
  int k;

  for (k = 0; k < N; k++)
  {
    printf("%2d   %2ld\n", k, feld[k]);
  }
  return;
}


int main(void)
{
  long quadratzahlen[N]; //Speicherplatz fuer Feld ausfassen
  int i;

  for (i = 0; i < N; i++)
  {
    quadratzahlen[i] = i * i;  //Feldelemente mit Werten belegen
  }
  ausgabe(quadratzahlen); //Anfangsadresse des Feldes wird uebergeben
  return 0;
}
```

Anfangsadresse des Feldes übergeben. Feldelemente werden **nicht** kopiert.

Feld an eine Funktion übergeben

(quadratzahlen3.c)

```c
#include <stdio.h>

#define N 11 //Anzahl Feldelemente. Index: 0 ... N-1

void ausgabe(long *feld) //operiert im Speicherbereich des Feldes quadratzahlen
{
  int k;

  for (k = 0; k < N; k++)
  {
    printf("%2d   %2ld\n", k, feld[k]);
  }
  return;
}


int main(void)
{
  long quadratzahlen[N]; //Speicherplatz fuer Feld ausfassen
  int i;

  for (i = 0; i < N; i++)
  {
    quadratzahlen[i] = i * i;  //Feldelemente mit Werten belegen
  }
  ausgabe(quadratzahlen); //Anfangsadresse des Feldes wird uebergeben
  return 0;
}
```

`feld[]` als Übergabeparameter ist bedeutungsgleich mit `*feld`

Feld als zusammenhängender Speicherbereich

(quadratzahlen4.c)

```c
#include <stdio.h>

#define N 11 //Anzahl Feldelemente. Index: 0 ... N-1

void ausgabe(long *feld) //operiert im Speicherbereich des Feldes quadratzahlen
{
  int k;

  for (k = 0; k < N; k++)
  {
    printf("%2d   %2ld\n", k, *(feld+k)); // +k erhoeht Adresse um k Feldelemente
  }
  return;
}


int main(void)
{
  long quadratzahlen[N]; //Speicherplatz fuer Feld ausfassen
  int i;

  for (i = 0; i < N; i++)
  {
    quadratzahlen[i] = i * i;  //Feldelemente mit Werten belegen
  }
  ausgabe(quadratzahlen); //Anfangsadresse des Feldes wird uebergeben
  return 0;
}
```

Notation **`feld[k]`** ist bedeutungsgleich mit Adressrechnung **`*(feld+k)`**

Zweidimensionales Feld anlegen, befüllen, übergeben

(kleines1x1.c) – Kleines Einmaleins, Zeilen i: 0 . . . 10, Spalten k: 0 . . . 15

```c
#include <stdio.h>

#define ZEILEN  11
#define SPALTEN 16

void ausgabe(long zahlentabelle[ZEILEN][SPALTEN])
{
  int i, k;

  for (i = 0; i < ZEILEN; i++)
  {
    for (k = 0; k < SPALTEN; k++)
    {
      printf("%3ld  ", zahlentabelle[i][k]);
    }
    printf("\n");
  }
}

int main(void)
{
  long produkte[ZEILEN][SPALTEN];
  int i, k;

  for (i = 0; i < ZEILEN; i++)
  {
    for (k = 0; k < SPALTEN; k++)
    {
      produkte[i][k] = i * k;
    }
  }
  ausgabe(produkte);
  return 0;
}
```

```
  0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
  0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
  0    2    4    6    8   10   12   14   16   18   20   22   24   26   28   30
  0    3    6    9   12   15   18   21   24   27   30   33   36   39   42   45
  0    4    8   12   16   20   24   28   32   36   40   44   48   52   56   60
  0    5   10   15   20   25   30   35   40   45   50   55   60   65   70   75
  0    6   12   18   24   30   36   42   48   54   60   66   72   78   84   90
  0    7   14   21   28   35   42   49   56   63   70   77   84   91   98  105
  0    8   16   24   32   40   48   56   64   72   80   88   96  104  112  120
  0    9   18   27   36   45   54   63   72   81   90   99  108  117  126  135
  0   10   20   30   40   50   60   70   80   90  100  110  120  130  140  150
```

Feld entspricht Matrix oder

Tabelle aus Spalten und Zellen

Speicherplatzbedarf bestimmen mit `sizeof`

```c
#include <stdio.h>

#define ZEILEN  11
#define SPALTEN 16

int main(void)
{
  long produkte[ZEILEN][SPALTEN];
  unsigned char i, k;

  for (i = 0; i < ZEILEN; i++)
  {
    for (k = 0; k < SPALTEN; k++)
    {
      produkte[i][k] = i * k;
    }
  }
  printf("Speicherbedarf eines Wertes der Variablen i: %3u Bytes\n", sizeof(i));         // 1 Byte
  printf("Speicherbedarf eines Wertes vom Typ long:    %3d Bytes\n", sizeof(long));      // 4 Bytes
  printf("Speicherbedarf des long-Feldes produkte:     %3d Bytes\n", sizeof(produkte)); // 704 = 4 * 16 * 11 Bytes
  return 0;
}
```

```
Speicherbedarf eines Wertes der Variablen i:   1 Bytes
Speicherbedarf eines Wertes vom Typ long:      4 Bytes
Speicherbedarf des long-Feldes produkte:     704 Bytes
```

- **`sizeof`** ist ein Schlüsselwort
- dient zur Bestimmung des Speicherplatzbedarfs von Variablen oder Typen. Variablen- oder Typname in runden Klammern übergeben
- liefert im Ergebnis den Wert in *Bytes*

Eigenschaften jedes Feldes

- Alle Feldelemente besitzen den *gleichen Datentyp*.

Eigenschaften jedes Feldes

- Alle Feldelemente besitzen den *gleichen Datentyp*.
- Die *Größe* des Feldes (Anzahl Elemente und Dimensionierung) muss in C bereits *beim Compilieren feststehen*.

Eigenschaften jedes Feldes

- Alle Feldelemente besitzen den *gleichen Datentyp*.
- Die *Größe* des Feldes (Anzahl Elemente und Dimensionierung) muss in C bereits *beim Compilieren feststehen*.
- Jeder *Feldindex* beginnt bei 0.

Eigenschaften jedes Feldes

- Alle Feldelemente besitzen den *gleichen Datentyp*.
- Die *Größe* des Feldes (Anzahl Elemente und Dimensionierung) muss in C bereits *beim Compilieren feststehen*.
- Jeder *Feldindex* beginnt bei 0.
- Über den Feldindex besteht direkter *wahlfreier Zugriff* auf jedes Feldelement.

Eigenschaften jedes Feldes

- Alle Feldelemente besitzen den *gleichen Datentyp*.
- Die *Größe* des Feldes (Anzahl Elemente und Dimensionierung) muss in C bereits *beim Compilieren feststehen*.
- Jeder *Feldindex* beginnt bei 0.
- Über den Feldindex besteht direkter *wahlfreier Zugriff* auf jedes Feldelement.
- Der Feldindex ist immer von einem Ganzzahltyp und wird stets in eckige Klammern `[...]` geschrieben.

Eigenschaften jedes Feldes

- Alle Feldelemente besitzen den *gleichen Datentyp*.
- Die *Größe* des Feldes (Anzahl Elemente und Dimensionierung) muss in C bereits *beim Compilieren feststehen*.
- Jeder *Feldindex* beginnt bei 0.
- Über den Feldindex besteht direkter *wahlfreier Zugriff* auf jedes Feldelement.
- Der Feldindex ist immer von einem Ganzzahltyp und wird stets in eckige Klammern `[...]` geschrieben.
- Ein Feld darf beliebig, aber endlich viele *Dimensionen* haben (mehr als vier Dimensionen aber sehr selten).

Eigenschaften jedes Feldes

- Alle Feldelemente besitzen den *gleichen Datentyp*.
- Die *Größe* des Feldes (Anzahl Elemente und Dimensionierung) muss in C bereits *beim Compilieren feststehen*.
- Jeder *Feldindex* beginnt bei 0.
- Über den Feldindex besteht direkter *wahlfreier Zugriff* auf jedes Feldelement.
- Der Feldindex ist immer von einem Ganzzahltyp und wird stets in eckige Klammern `[...]` geschrieben.
- Ein Feld darf beliebig, aber endlich viele *Dimensionen* haben (mehr als vier Dimensionen aber sehr selten).
- Die Elemente eines Feldes sind im Speicher *unmittelbar aufeinanderfolgend* abgelegt.

Eigenschaften jedes Feldes

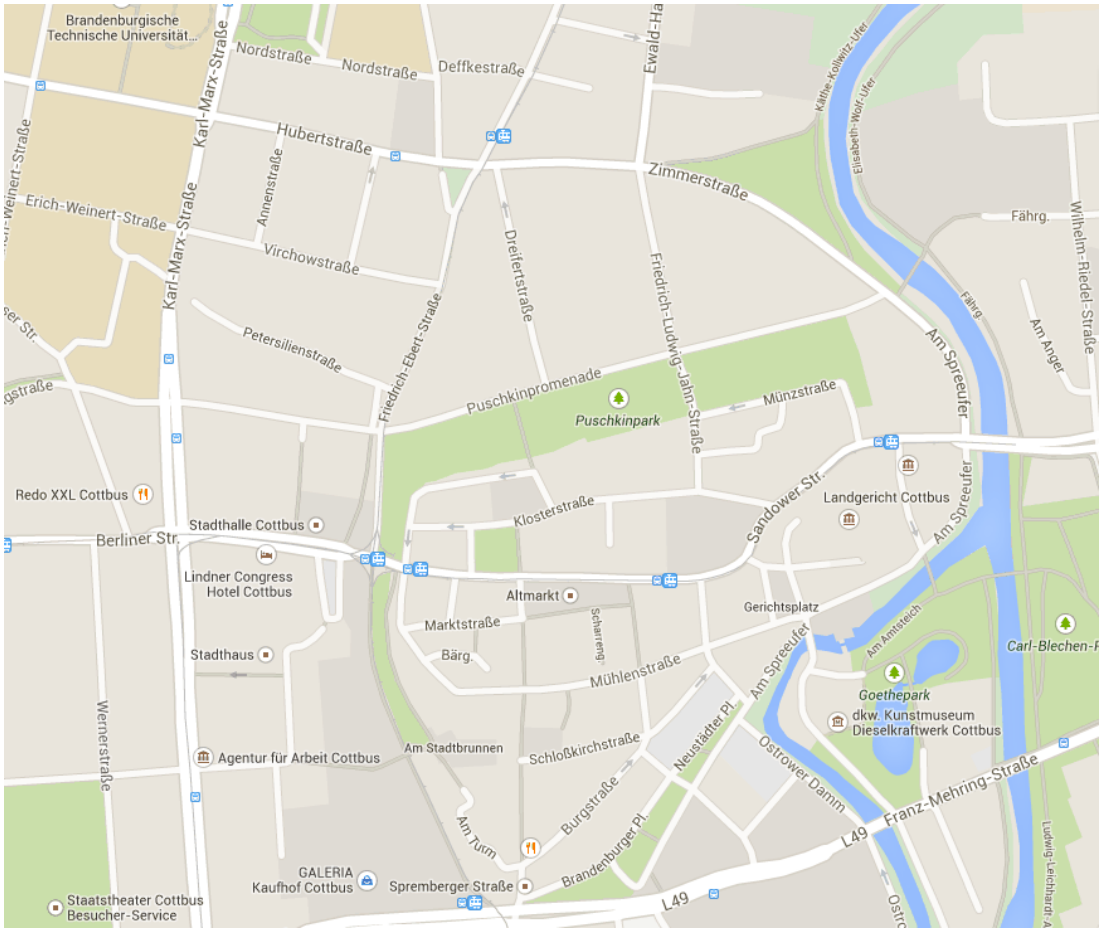
- Alle Feldelemente besitzen den *gleichen Datentyp*.
- Die *Größe* des Feldes (Anzahl Elemente und Dimensionierung) muss in C bereits *beim Compilieren feststehen*.
- Jeder *Feldindex* beginnt bei 0.
- Über den Feldindex besteht direkter *wahlfreier Zugriff* auf jedes Feldelement.
- Der Feldindex ist immer von einem Ganzzahltyp und wird stets in eckige Klammern `[...]` geschrieben.
- Ein Feld darf beliebig, aber endlich viele *Dimensionen* haben (mehr als vier Dimensionen aber sehr selten).
- Die Elemente eines Feldes sind im Speicher *unmittelbar aufeinanderfolgend* abgelegt.
- Der gewählte *Feldname* verkörpert in C die *Anfangsadresse*, ab der das Feld abgespeichert ist.

Eigenschaften jedes Feldes

- Alle Feldelemente besitzen den *gleichen Datentyp*.
- Die *Größe* des Feldes (Anzahl Elemente und Dimensionierung) muss in C bereits *beim Compilieren feststehen*.
- Jeder *Feldindex* beginnt bei 0.
- Über den Feldindex besteht direkter *wahlfreier Zugriff* auf jedes Feldelement.
- Der Feldindex ist immer von einem Ganzzahltyp und wird stets in eckige Klammern `[...]` geschrieben.
- Ein Feld darf beliebig, aber endlich viele *Dimensionen* haben (mehr als vier Dimensionen aber sehr selten).
- Die Elemente eines Feldes sind im Speicher *unmittelbar aufeinanderfolgend* abgelegt.
- Der gewählte *Feldname* verkörpert in C die *Anfangsadresse*, ab der das Feld abgespeichert ist.
- Bei *Parameterübergaben* wird ein Feld nicht elementweise kopiert, sondern seine *Anfangsadresse weitergegeben*.

Netzwerk-Strukturen im Computer erfassen

Beispiel: Stadtplan Cottbus

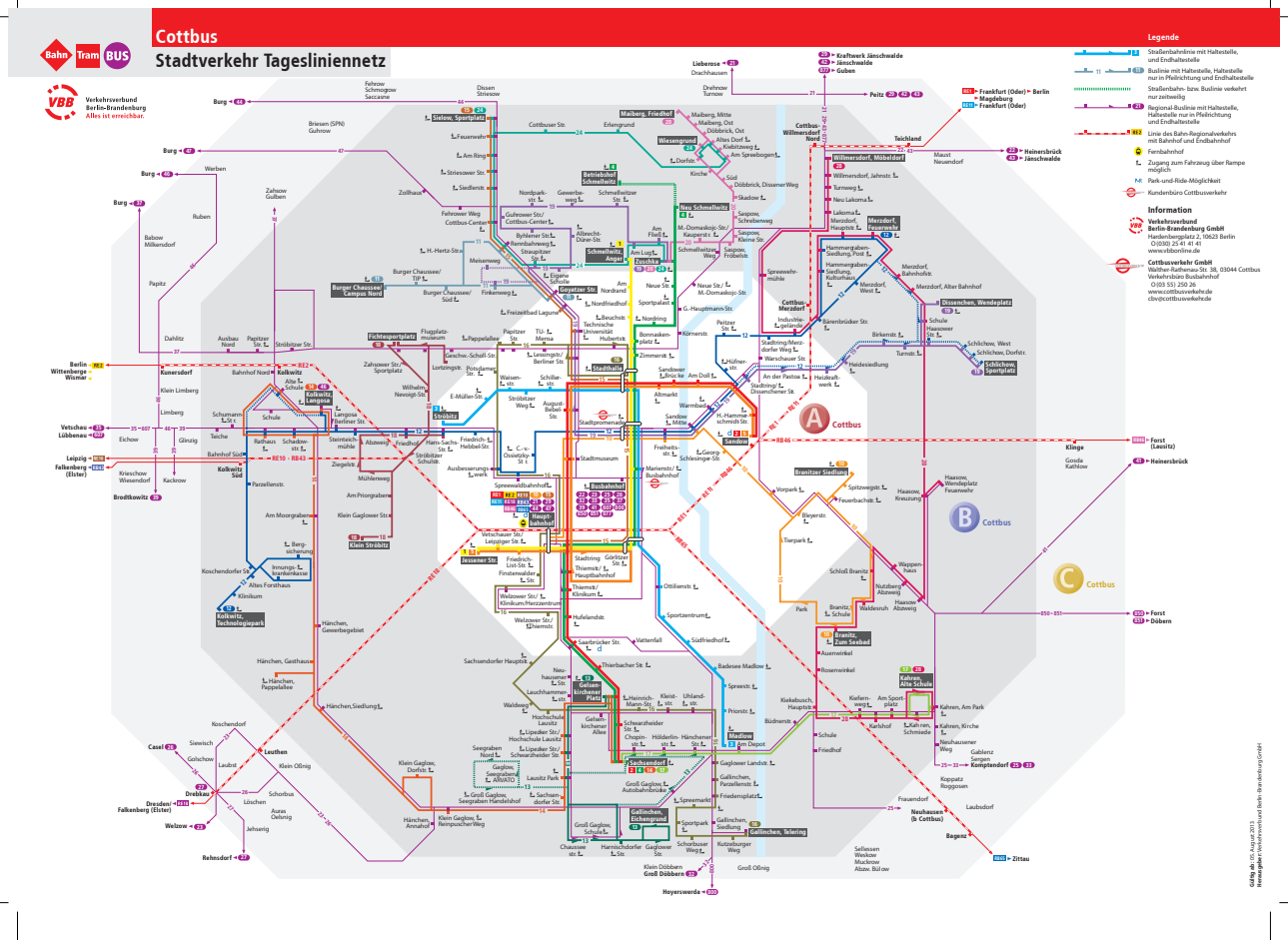


Quelle: Google Maps

$\Longrightarrow$ Kürzesten Weg von *A* nach *B* finden

Netzwerk-Strukturen im Computer erfassen

Beispiel: Liniennetzplan Tram/Bus in Cottbus



Quelle: Cottbuser Verkehrsbetriebe

$\Longrightarrow$ Schnellste Verbindung von *A* nach *B* finden

Netzwerk-Strukturen im Computer erfassen

Beispiel: Freundschaftsverbindungen bei facebook

Quelle: facebook.com

„Über maximal 6 Personen sind zwei beliebige facebook-Nutzer weltweit miteinander verbunden.“

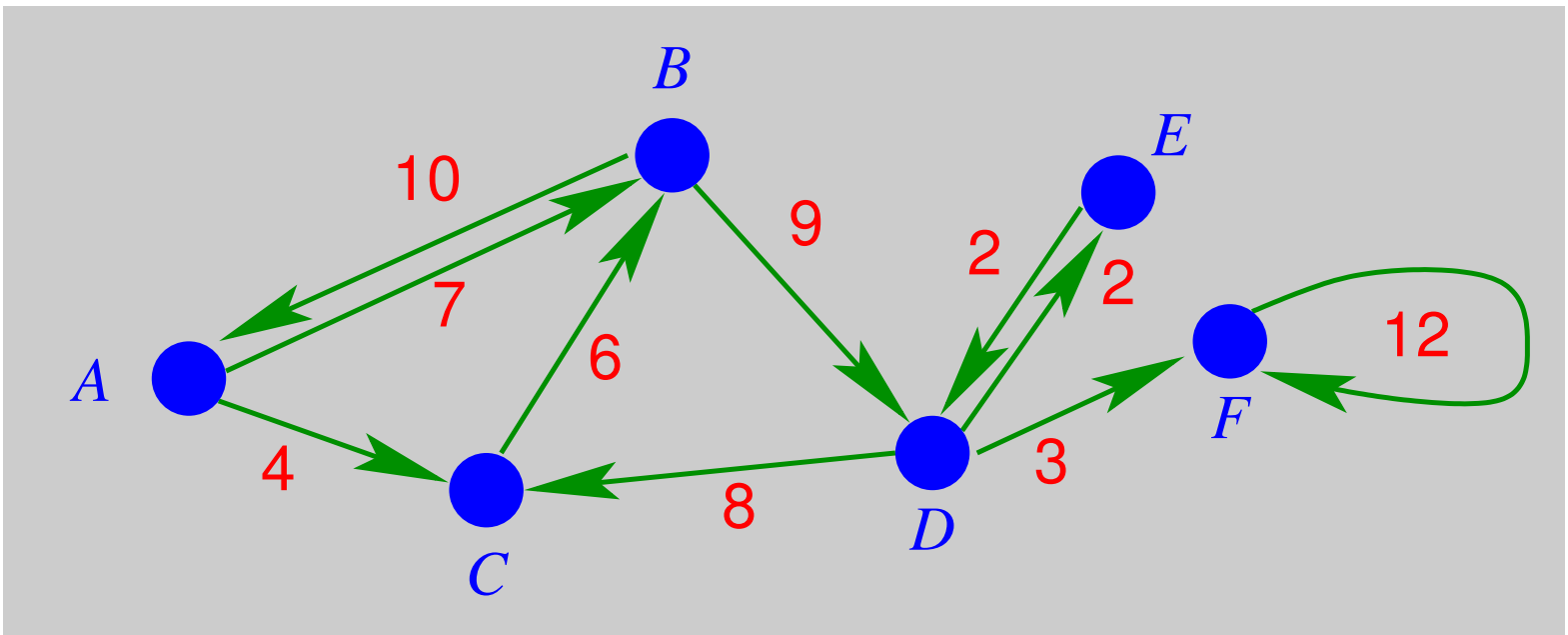
Netzwerk-Strukturen im Computer erfassen

Beispiel: Chemieanlage mit Rohrleitungsnetzwerk

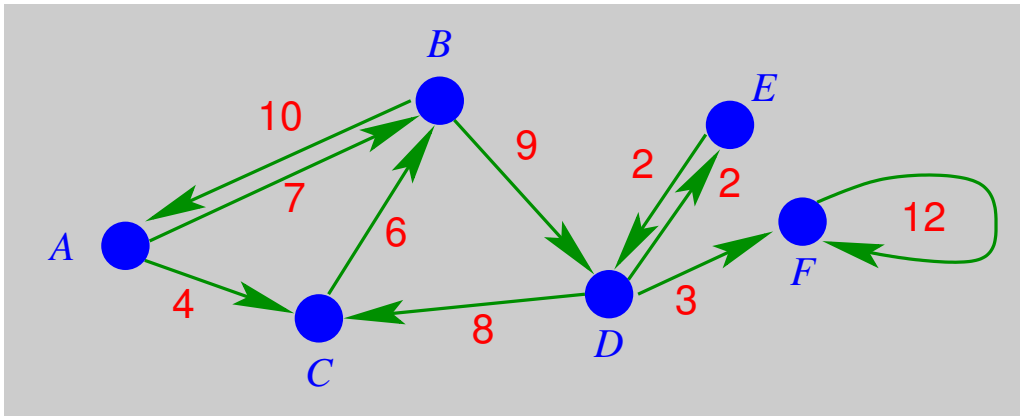
$\Longrightarrow$ Optimalen Durchfluss in der Gesamtanlage steuern, wenn eine Rohrleitung zur Reinigung abgeschaltet ist

Netzwerk als Graph abstrakt beschreiben

Ein **gerichteter Graph** ist ein Netzwerk aus *Knoten* und *Kanten*, wobei jede Kante von einem Startknoten zu einem Zielknoten führt und eine *Kantenbewertung* (z.B. Entfernung) tragen kann.



Graph durch Knoten- und Kantenmenge darstellen



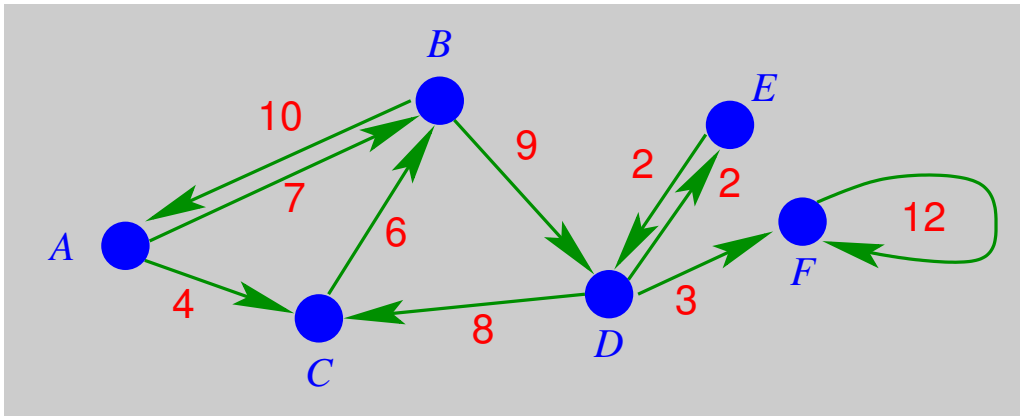
V: Knotenmenge, E: Kantenmenge

$$
\begin{aligned}
G &= (V, E) \text{ mit } E \subseteq V \times V \\
V &= \{A, B, C, D, E, F\} \\
E &= \{(A, B), (A, C), (B, A), (B, D), \\
&\quad (C, B), (D, C), (D, E), (D, F), \\
&\quad (E, D), (F, F)\}
\end{aligned}
$$

Kantenbewertungsfunktion $f : E \longrightarrow \mathbb{R}$

$$
\begin{aligned}
f(A, B) &= 7 & f(A, C) &= 4 \\
f(B, A) &= 10 & f(B, D) &= 9 \\
&\ldots & f(F, F) &= 12
\end{aligned}
$$

Graph durch Knoten- und Kantenmenge darstellen



V: Knotenmenge, E: Kantenmenge

$$
\begin{aligned}
G &= (V, E) \text{ mit } E \subseteq V \times V \\
V &= \{A, B, C, D, E, F\} \\
E &= \{(A,B),(A,C),(B,A),(B,D), \\
&\quad (C,B),(D,C),(D,E),(D,F), \\
&\quad (E,D),(F,F)\}
\end{aligned}
$$

Kantenbewertungsfunktion $f : E \longrightarrow \mathbb{R}$

$$
\begin{aligned}
f(A,B) &= 7 & f(A,C) &= 4 \\
f(B,A) &= 10 & f(B,D) &= 9 \\
&\ldots & f(F,F) &= 12
\end{aligned}
$$

Graph als Matrix
(zweidimensionales Feld)

	A	B	C	D	E	F
A	∞	7	4	∞	∞	∞
B	10	∞	∞	9	∞	∞
C	∞	6	∞	∞	∞	∞
D	∞	∞	8	∞	2	3
E	∞	∞	∞	2	∞	∞
F	∞	∞	∞	∞	∞	12

Im deutschen Flugnetz gut angebundene Flughäfen

12 Flughäfen mit jeweils mehr als einem innerdeutschen Linienziel

Entfernungstabelle innerdeutsche Direktflüge in km

	Berlin	Bremen	Dresden	Düsseldorf	FrankfurtM	Hamburg	Hannover	Köln/Bonn	Leipzig/Halle	München	Nürnberg	Stuttgart
Berlin	-			478	424			477		504	379	512
Bremen		-			320					583		479
Dresden			-	486	372	377		474		359		412
Düsseld.	478		486	-	183	339			389	487	364	
Frankf.M	424	330	372	183	-	393	262	153	293	304	187	152
Hamburg			377	339	393	-		356		612	462	534
Hannover					262		-			489		402
KölnBonn	477		474		153	356		-	380	456		
LeipzigH				389	293			380	-	360		365
München	504	583	359	487	304	612	489	456	360	-	150	191
Nürnberg	379			364	187	462				150	-	
Stuttgart	512	479	412		152	534	402		365	191		-

Entfernungstabelle innerdeutsche Direktflüge in km

	Berlin	Bremen	Dresden	Düsseldorf	FrankfurtM	Hamburg	Hannover	Köln/Bonn	Leipzig/Halle	München	Nürnberg	Stuttgart
Berlin	-			478	424			477		504	379	512
Bremen		-			320					583		479
Dresden			-	486	372	377		474		359		412
Düsseld.	478		486	-	183	339			389	487	364	
Frankf.M	424	330	372	183	-	393	262	153	293	304	187	152
Hamburg			377	339	393	-		356		612	462	534
Hannover					262		-			489		402
KölnBonn	477		474		153	356		-	380	456		
LeipzigH				389	293			380	-	360		365
München	504	583	359	487	304	612	489	456	360	-	150	191
Nürnberg	379			364	187	462				150	-	
Stuttgart	512	479	412		152	534	402		365	191		-

Von jedem Flughafen zu jedem Flughafen kürzesten Weg im Fluglinniennetz bestimmen, auch über Umstiege. Beispiel: Von Dresden nach Bremen gibt es keine Direktverbindung, aber Bremen von Dresden aus via FrankfurtM mit innerdeutschen Linienflügen auf kürzestem Weg erreichbar

Bestimmen der kürzesten Wege im Fluglinniennetz

flugverbindungen.c – Floyd-Warshall-Algorithmus

```c
#include <stdio.h>

#define N   12   //Anzahl Flughaefen
#define INF  9999  //infinity: Entfernungsmasszahl wenn keine Direktverbindung

void findeKuerzesteVerbindung(int km[N][N]); //Funktionsprototyp Floyd-Warshall-Algorithmus


int main(void)
{
  int z, s;
  char flughafen[N][15] = {"Berlin", "Bremen", "Dresden", "Duesseldorf",
                           "FrankfurtM", "Hamburg", "Hannover", "KoelnBonn",
                           "LeipzigHalle", "Muenchen", "Nuernberg", "Stuttgart"};

  int entfernung[N][N] = { {  0, INF, INF, 478, 424, INF, INF, 477, INF, 504, 379, 512}, //Berlin
                           {INF,   0, INF, INF, 330, INF, INF, INF, INF, 583, INF, 479}, //Bremen
                           {INF, INF,   0, 486, 372, 377, INF, 474, INF, 359, INF, 412}, //Dresden
                           {478, INF, 486,   0, 183, 339, INF, INF, 389, 487, 364, INF}, //Duesseldorf
                           {424, 330, 372, 183,   0, 393, 262, 153, 293, 304, 187, 152}, //FrankfurtM
                           {INF, INF, 377, 339, 393,   0, INF, 356, INF, 612, 462, 534}, //Hamburg
                           {INF, INF, INF, INF, 262, INF,   0, INF, INF, 489, INF, 402}, //Hannover
                           {477, INF, 474, INF, 153, 356, INF,   0, 380, 456, INF, INF}, //KoelnBonn
                           {INF, INF, INF, 389, 293, INF, INF, 380,   0, 360, INF, 365}, //LeipzigHalle
                           {504, 583, 359, 487, 304, 612, 489, 456, 360,   0, 150, 191}, //Muenchen
                           {379, INF, INF, 364, 187, 462, INF, INF, INF, 150,   0, INF}, //Nuernberg
                           {512, 479, 412, INF, 152, 534, 402, INF, 365, 191, INF,   0}  //Stuttgart
                         };

  findeKuerzesteVerbindung(entfernung); //Aktualisieren der Entfernungsmatrix mit den Minimalwerten
```

Initialisierung der Entfernungsmatrix und der Flughafennamensmatrix, Funktionsaufruf **`findeKuerzesteVerbindung`** zur Optimierung

Bestimmen der kürzesten Wege im Fluglinniennetz

Flughäfen `Berlin` ... `Stuttgart` per Indexwert 0 ... 11 durchnummeriert

```
void findeKuerzesteVerbindung(int km[N][N])  //Floyd-Warshall-Algorithmus
{
  int i, j, k;
  int a;

  for (k = 0; k < N; k++)
  {
    for (i = 0; i < N; i++)
    {
      for(j = 0; j < N; j++)
      {
        a = km[i][k] + km[k][j];
        if (a < km[i][j])
        {
          km[i][j] = a;
        }
      }
    }
  }
  return;
}
```

Alle Kanten $i \rightarrow j$ durchlaufen sowie alle Umwege über jeden Zwischenknoten k betrachtet. Ist die Entfernung von i nach j über k kürzer als die Direktverbindung, so wird die Matrix aktualisiert, also die eingetragene Entfernung durch die kürzere Entfernung ersetzt.

Bestimmen der kürzesten Wege im Fluglinniennetz

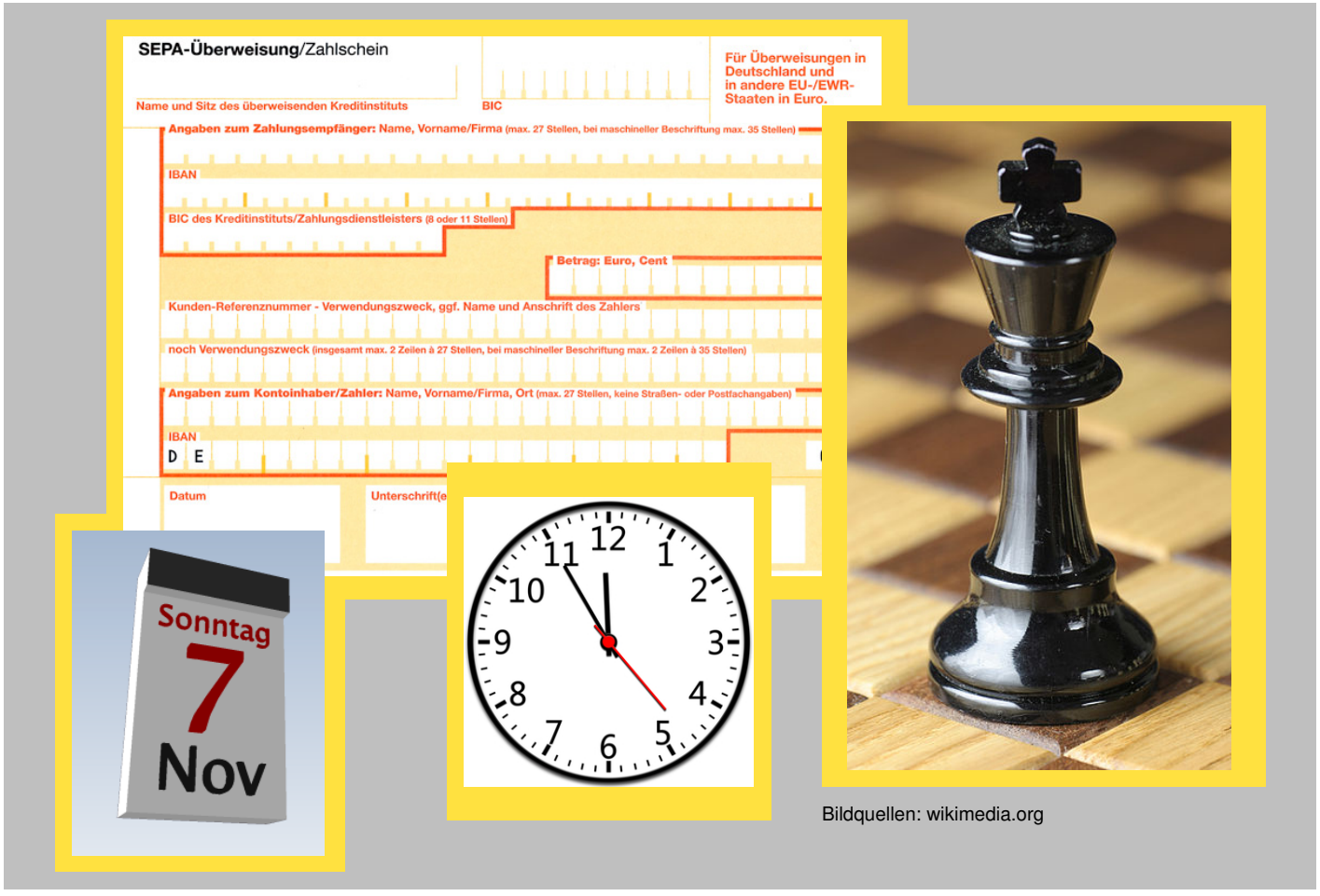
Schleifenvariablen z: Zeile, s: Spalte durchlaufen die Felder

```
  findeKuerzesteVerbindung(entfernung); //Aktualisieren der Entfernungsmatrix mit den Minimalwerten

  printf("\n             |"); //Ausgabe Tabellenueberschrift
  for (z = 0; z < N; z++)
  {
    for (s = 0; s < 5; s++)
    {
      printf("%c", flughafen[z][s]);
    }
    printf("|", flughafen[z]);
  }
  printf("\n");
  for (z = 0; z < N; z++)  //formatierte Ausgabe der optimierten Entfernungsmatrix
  {
    printf("%-13s|", flughafen[z]);
    for (s = 0; s < N; s++)
    {
      printf("%4d |", entfernung[z][s]);
    }
    printf("\n");
  }
  return 0;
}
```

	Berli	Breme	Dresd	Duess	Frank	Hambu	Hanno	Koeln	Leipz	Muenc	Nuern	Stutt
Berlin	0	754	796	478	424	817	686	477	717	504	379	512
Bremen	754	0	702	513	330	723	592	483	623	583	517	479
Dresden	796	702	0	486	372	377	634	474	665	359	509	412
Duesseldorf	478	513	486	0	183	339	445	336	389	487	364	335
FrankfurtM	424	330	372	183	0	393	262	153	293	304	187	152
Hamburg	817	723	377	339	393	0	655	356	686	612	462	534
Hannover	686	592	634	445	262	655	0	415	555	489	449	402
KoelnBonn	477	483	474	336	153	356	415	0	380	456	340	305
LeipzigHalle	717	623	665	389	293	686	555	380	0	360	480	365
Muenchen	504	583	359	487	304	612	489	456	360	0	150	191
Nuernberg	379	517	509	364	187	462	449	340	480	150	0	339
Stuttgart	512	479	412	335	152	534	402	305	365	191	339	0

Formatierte Ausgabe der optimierten Entfernungsmatrix im zweiten Teil der **main**-Funktion

Datensatz zur Bündelung zusammengehöriger Daten



Bildquellen: wikimedia.org

Ein Datensatz erfasst alle Daten zur Beschreibung eines Sachverhalts oder eines Gegenstands

Allgemeiner Aufbau eines Datensatzes

Ein **Datensatz** besteht üblicherweise aus mehreren *Komponenten*. Durch die Komponenten, die auch ihrerseits wieder aus weiteren Komponenten zusammengesetzt sein dürfen, erhält ein Datensatz seine *feste Struktur*. Diese Struktur ist der Rahmen zur Datenerfassung. Die Datenwerte aller Komponenten liefern eine vollständige Datensatzbelegung.

Beispiele für Datensatzstrukturen und ihre Komponenten

- Uhrzeit: Stunde, Minute, Sekunde
- Kalenderdatum: Tag, Monat, Jahr

Allgemeiner Aufbau eines Datensatzes

Ein **Datensatz** besteht üblicherweise aus mehreren *Komponenten*. Durch die Komponenten, die auch ihrerseits wieder aus weiteren Komponenten zusammengesetzt sein dürfen, erhält ein Datensatz seine *feste Struktur*. Diese Struktur ist der Rahmen zur Datenerfassung. Die Datenwerte aller Komponenten liefern eine vollständige Datensatzbelegung.

Beispiele für Datensatzstrukturen und ihre Komponenten

- Uhrzeit: Stunde, Minute, Sekunde
- Kalenderdatum: Tag, Monat, Jahr
- Überweisungsträger: Empfänger, Empfängerkonto IBAN, BIC, Überweisungsbetrag, Währung, Verwendungszweck, Kontoinhaber IBAN, *Kalenderdatum*, Autorisierungsvermerk

Allgemeiner Aufbau eines Datensatzes

Ein **Datensatz** besteht üblicherweise aus mehreren *Komponenten*. Durch die Komponenten, die auch ihrerseits wieder aus weiteren Komponenten zusammengesetzt sein dürfen, erhält ein Datensatz seine *feste Struktur*. Diese Struktur ist der Rahmen zur Datenerfassung. Die Datenwerte aller Komponenten liefern eine vollständige Datensatzbelegung.

Beispiele für Datensatzstrukturen und ihre Komponenten

- Uhrzeit: Stunde, Minute, Sekunde
- Kalenderdatum: Tag, Monat, Jahr
- Überweisungsträger: Empfänger, Empfängerkonto IBAN, BIC, Überweisungsbetrag, Währung, Verwendungszweck, Kontoinhaber IBAN, *Kalenderdatum*, Autorisierungsvermerk
- Schachfigur: Figurart (Bauer, . . ., Turm), Farbe (schwarz, weiß), Position auf dem Schachbrett (x, y), geschlagen (ja, nein)

Datensatzstrukturen in C anlegen

```
struct <Name>
{
   <Typ> <Komponentenname_1>; //Komponente_1
            ⋮
   <Typ> <Komponentenname_n>; //Komponente_n
};
```

- `struct <Name>` bildet einen selbstdefinierten *Datentyp*

Datensatzstrukturen in C anlegen

```
struct <Name>
{
   <Typ> <Komponentenname_1>; //Komponente_1
            ⋮
   <Typ> <Komponentenname_n>; //Komponente_n
};
```

- `struct <Name>` bildet einen selbstdefinierten *Datentyp*
- Im Quelltext deklariert man Datensatzstrukturen meist *global*, also unmittelbar hinter dem Präprozessorteil und *vor* den eigenen Funktionen.

Datensatzstrukturen in C anlegen

```
struct <Name>
{
   <Typ> <Komponentenname_1>; //Komponente_1
            ⋮
   <Typ> <Komponentenname_n>; //Komponente_n
};
```

- `struct <Name>` bildet einen selbstdefinierten *Datentyp*
- Im Quelltext deklariert man Datensatzstrukturen meist *global*, also unmittelbar hinter dem Präprozessorteil und *vor* den eigenen Funktionen.
- Jede *Komponente* innerhalb der Struktur wird beschrieben durch ihren *Typ* und einen selbstgewählten *Komponentennamen*.

Datensatzstrukturen in C anlegen

```
struct <Name>
{
   <Typ> <Komponentenname_1>; //Komponente_1
            ⋮
   <Typ> <Komponentenname_n>; //Komponente_n
};
```

- `struct <Name>` bildet einen selbstdefinierten *Datentyp*
- Im Quelltext deklariert man Datensatzstrukturen meist *global*, also unmittelbar hinter dem Präprozessorteil und *vor* den eigenen Funktionen.
- Jede *Komponente* innerhalb der Struktur wird beschrieben durch ihren *Typ* und einen selbstgewählten *Komponentennamen*.
- Eine Struktur darf ihrerseits Komponente in einer anderen Struktur sein (beliebig, aber endlich tief *schachtelbar*).

Struktur für Weckzeit anlegen, initialisieren, auslesen

weckzeit.c

```c
#include <stdio.h>

struct TUhrzeit
{
  int stunde;        //0...23
  int minute;        //0...59
  int sekunde;       //0...59
  char zeitzone[5]; //z.B. "GMT" oder "MEZ"
};                   //Semikolon am Ende nicht vergessen!


int main(void)
{
  struct TUhrzeit myalarm = {7, 30, 0, "MEZ"};

  printf("Meine Weckzeit: %2d : %2d : %2d (%s)\n",
         myalarm.stunde, myalarm.minute, myalarm.sekunde, myalarm.zeitzone);
  return 0;
}
```

```
Meine Weckzeit:  7 : 30 :  0 (MEZ)
```

- `struct TUhrzeit myalarm = {7, 30, 0, "MEZ"};` legt Variable `myalarm` vom Typ `struct TUhrzeit` an.
- Komponenten werden in der Reihenfolge, wie sie in der Struktur definiert sind, mit Werten belegt.

Struktur für Weckzeit anlegen, initialisieren, auslesen

weckzeit.c

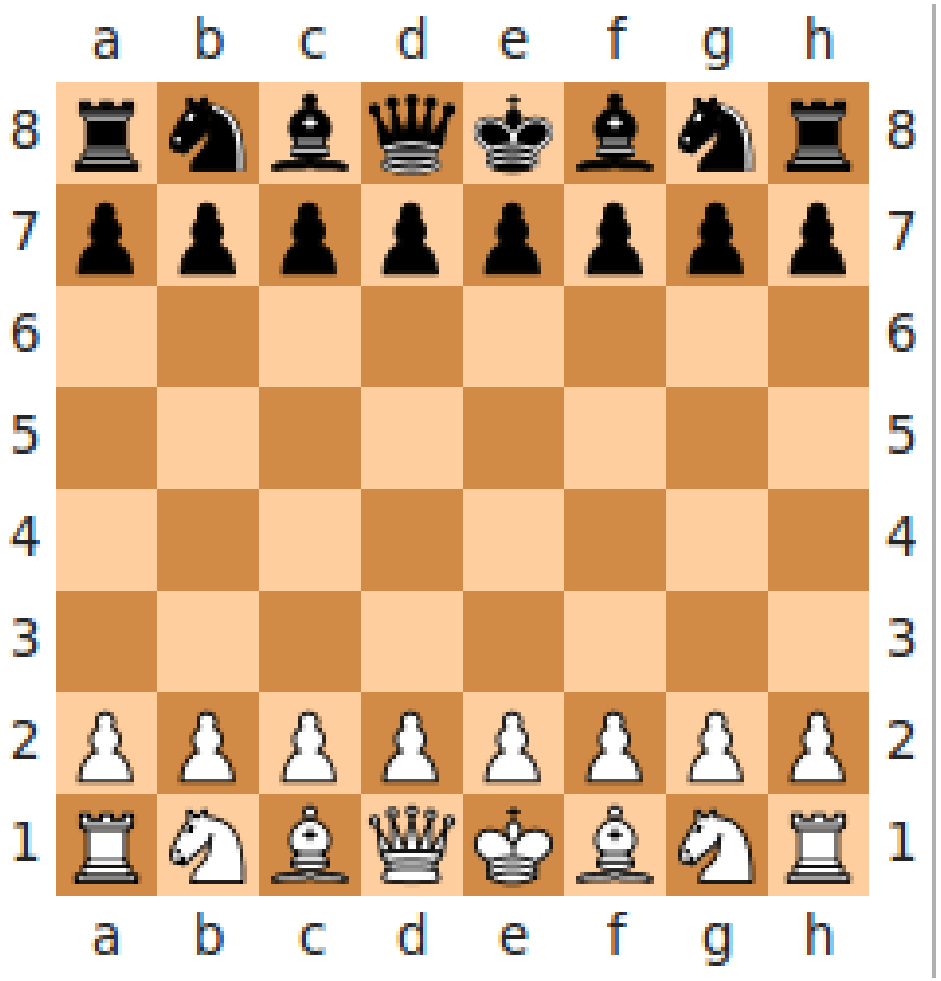
```c
#include <stdio.h>

struct TUhrzeit
{
  int stunde;        //0...23
  int minute;        //0...59
  int sekunde;       //0...59
  char zeitzone[5]; //z.B. "GMT" oder "MEZ"
};                   //Semikolon am Ende nicht vergessen!


int main(void)
{
  struct TUhrzeit myalarm = {7, 30, 0, "MEZ"};

  printf("Meine Weckzeit: %2d : %2d : %2d (%s)\n",
         myalarm.stunde, myalarm.minute, myalarm.sekunde, myalarm.zeitzone);
  return 0;
}
```

```
Meine Weckzeit:  7 : 30 :  0 (MEZ)
```

- Punktoperator `.` erlaubt lesenden wie schreibenden Zugriff auf die Komponenten.
- Beispielsweise ändert eine zusätzliche Programmzeile **`myalarm.minute = 15;`** den entsprechenden Eintrag.

Schachfiguren auf Spielfeld platzieren und bewegen

Programmierprojekt, das Felder und Strukturen kombiniert



www.wikipedia.de

Datenstruktur einer Schachfigur

```
struct TSchachfigur        //Datentyp TSchachfigur definieren
{
  char figur;              //'B': Bauer, 'D': Dame, 'K': Koenig, 'L': Laeufer, 'S': Springer, 'T': Turm
  unsigned char sw;        //0: schwarz, 1: weiss
  unsigned char posx;      //1: a, ..., 8: h
  unsigned char posy;      //1...8
  unsigned char sichtbar; //0: nein, 1: ja
};
```

Komponenten

- **`char figur`** `'B'` Bauer, `'D'` Dame, `'K'` König, `'L'` Läufer, `'S'` Springer, `'T'` Turm
- **`unsigned char sw`** `0` schwarz, `1` weiß
- **`unsigned char posx`** `1` a, `2` b, . . . `8` h
- **`unsigned char posy`** `1`, `2`, . . ., `8`
- **`unsigned char sichtbar`** `0` nein (Figur geschlagen), `1` ja

Feld von Schachfiguren

- Zum Schachspiel gehören insgesamt *32 Figuren*.
- Wir wollen alle diese Figuren jeweils als Datensatzbelegungen im Programm speichern.
- Dazu bietet es sich an, ein *Feld von Schachfiguren* anzulegen

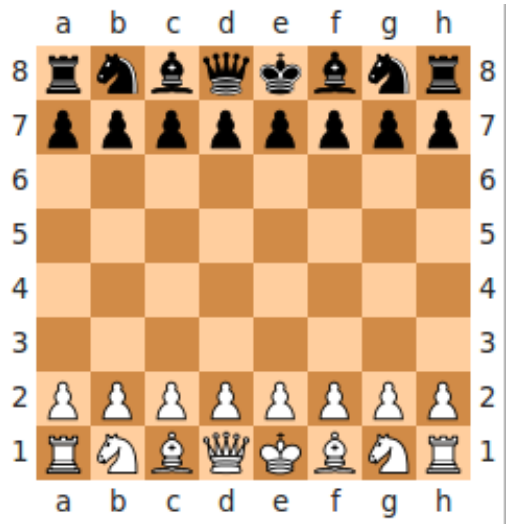
```
struct TSchachfigur figuren[32]
```

Feld von Schachfiguren

- Zum Schachspiel gehören insgesamt *32 Figuren*.
- Wir wollen alle diese Figuren jeweils als Datensatzbelegungen im Programm speichern.
- Dazu bietet es sich an, ein *Feld von Schachfiguren* anzulegen

```
struct TSchachfigur figuren[32]
```

- Das Feld beherbergt alle 32 Figuren und wird mit den Daten der *Schachgrundstellung* initialisiert.
- Dies geschieht gleich beim Anlegen des Feldes in der **`main`**-Funktion.



Feld der Schachfiguren initialisieren

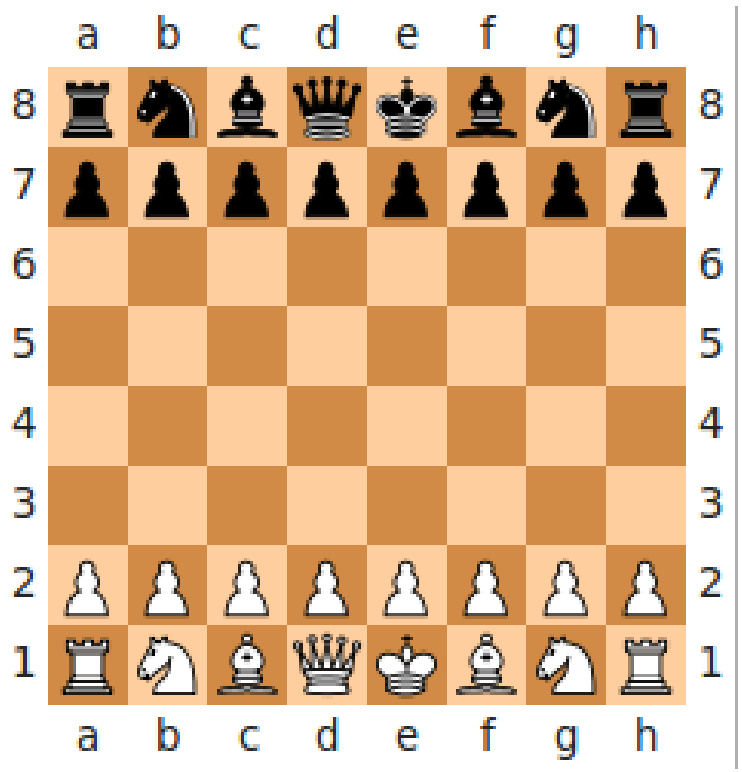
schachzuege2.c – Anfangsteil der main-Funktion

```
int main(void)
{
  struct TSchachfigur figuren[32] = { {'T', 1, 1, 1, 1}, //weisser Turm a1
                                      {'S', 1, 2, 1, 1}, //weisser Springer b1
                                      {'L', 1, 3, 1, 1}, //weisser Laeufer c1
                                      {'D', 1, 4, 1, 1}, //weisse Dame d1
                                      {'K', 1, 5, 1, 1}, //weisser Koenig e1
                                      {'L', 1, 6, 1, 1}, //weisser Laeufer f1
                                      {'S', 1, 7, 1, 1}, //weisser Springer g1
                                      {'T', 1, 8, 1, 1}, //weisser Turm h1
                                      {'B', 1, 1, 2, 1}, {'B', 1, 2, 2, 1}, //weisse Bauern a2, b2
                                      {'B', 1, 3, 2, 1}, {'B', 1, 4, 2, 1}, //weisse Bauern c2, d2
                                      {'B', 1, 5, 2, 1}, {'B', 1, 6, 2, 1}, //weisse Bauern e2, f2
                                      {'B', 1, 7, 2, 1}, {'B', 1, 8, 2, 1}, //weisse Bauern g2, h2
                                      {'T', 0, 1, 8, 1}, //schwarzer Turm a8
                                      {'S', 0, 2, 8, 1}, //schwarzer Springer b8
                                      {'L', 0, 3, 8, 1}, //schwarzer Laeufer c8
                                      {'D', 0, 4, 8, 1}, //schwarze Dame d8
                                      {'K', 0, 5, 8, 1}, //schwarzer Koenig e8
                                      {'L', 0, 6, 8, 1}, //schwarzer Laeufer f8
                                      {'S', 0, 7, 8, 1}, //schwarzer Springer g8
                                      {'T', 0, 8, 8, 1}, //schwarzer Turm h8
                                      {'B', 0, 1, 7, 1}, {'B', 0, 2, 7, 1}, //schwarze Bauern a7, b7
                                      {'B', 0, 3, 7, 1}, {'B', 0, 4, 7, 1}, //schwarze Bauern c7, d7
                                      {'B', 0, 5, 7, 1}, {'B', 0, 6, 7, 1}, //schwarze Bauern e7, f7
                                      {'B', 0, 7, 7, 1}, {'B', 0, 8, 7, 1}, //weisse Bauern g7, h7
                                    };

  zeichneSchachbrett(figuren); //Anfangsaufstellung der Figuren
```

Schachbrett zeichnen und Schachfigurenfeld auslesen

schachzuege2.c – Funktion zeichneSchachbrett

```
   a  b  c  d  e  f  g  h
  +--+--+--+--+--+--+--+--+
8 |sT|sS|sL|sD|sK|sL|sS|sT|
  +--+--+--+--+--+--+--+--+
7 |sB|sB|sB|sB|sB|sB|sB|sB|
  +--+--+--+--+--+--+--+--+
6 |  |  |  |  |  |  |  |  |
  +--+--+--+--+--+--+--+--+
5 |  |  |  |  |  |  |  |  |
  +--+--+--+--+--+--+--+--+
4 |  |  |  |  |  |  |  |  |
  +--+--+--+--+--+--+--+--+
3 |  |  |  |  |  |  |  |  |
  +--+--+--+--+--+--+--+--+
2 |wB|wB|wB|wB|wB|wB|wB|wB|
  +--+--+--+--+--+--+--+--+
1 |wT|wS|wL|wD|wK|wL|wS|wT|
  +--+--+--+--+--+--+--+--+
```



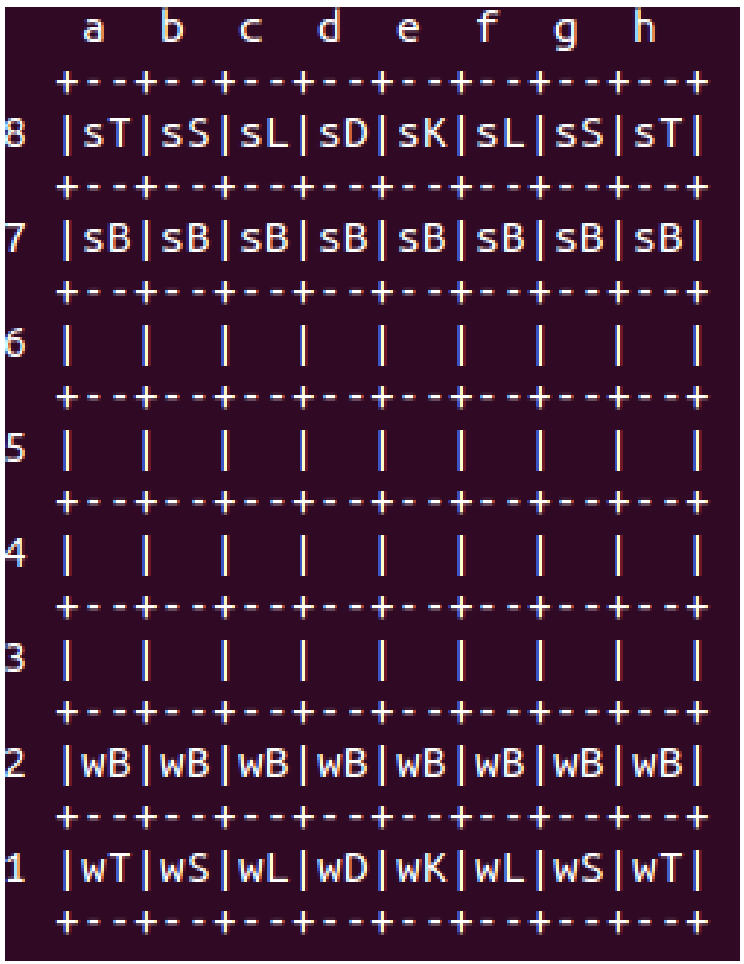
Schachbrett zeichnen und Schachfigurenfeld auslesen

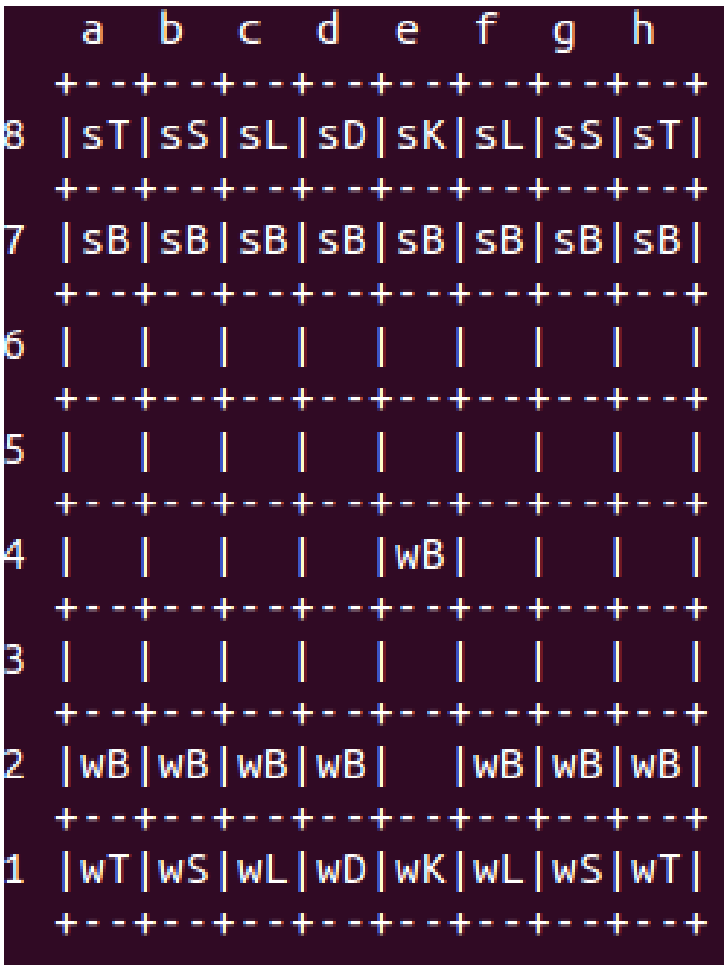
schachzuege2.c – Funktion zeichneSchachbrett

```
void zeichneSchachbrett(struct TSchachfigur objekte[])
{
  int z; //Zeile
  int s; //Spalte
  int i;
  char color;
  char stein;

  printf("\n   a  b  c  d  e  f  g  h\n  +--+--+--+--+--+--+--+--+\n");
  for(z = 8; z > 0; z--)
  {
    printf("%1d ", z);
    for(s = 1; s <= 8; s++)
    {
      color = ' ';
      stein = ' ';
      for (i = 0; i < 32; i++)  //sichtbarer Spielstein an der Position (z,s)?
      {
        if ((objekte[i].posy == z) && (objekte[i].posx == s) && (objekte[i].sichtbar))
        {
          stein = objekte[i].figur;
          if (objekte[i].sw)
          {
            color = 'w'; //weiss
          }
          else
          {
            color = 's'; //schwarz
          }
        }
      }
      printf("|%c%c", color, stein);
    }
    printf("|\n  +--+--+--+--+--+--+--+--+\n");
  }
  return;
}
```

Einen Schachzug ausführen

Weiß beginnt und zieht Bauer e2 nach e4 („Italienische Eröffnung“)





Einen Schachzug ausführen

schachzuege2.c – Funktion ziehe

```
void ziehe(struct TSchachfigur s[], unsigned char vonx, unsigned char vony, unsigned char nachx, unsigned char nachy)
{
  int i;
  char c;

  if ((vonx > 0) && (vonx < 9) && (vony > 0) && (vony < 9) && (nachx > 0) && (nachx < 9) && (nachy > 0) && (nachy < 9))
  {
    for (i = 0; i < 32; i++)  //Falls eine Figur auf Zielposition steht, dann schlagen, also unsichtbar schalten
    {
      if ((s[i].posx == nachx) && (s[i].posy == nachy) && (s[i].sichtbar))
      {
        s[i].sichtbar = 0;
      }
    }
    for (i = 0; i < 32; i++)  //Zug ausfuehren. Es erfolgt keine Ueberpruefung auf Einhaltung der Schach-Zugregeln
    {
      if ((s[i].posx == vonx) && (s[i].posy == vony) && (s[i].sichtbar))
      {
        s[i].posx = nachx;
        s[i].posy = nachy;
      }
    }
  }
  zeichneSchachbrett(s);
  printf("\n<Taste>\n\n"); scanf("%c", &c);
  return;
}
```

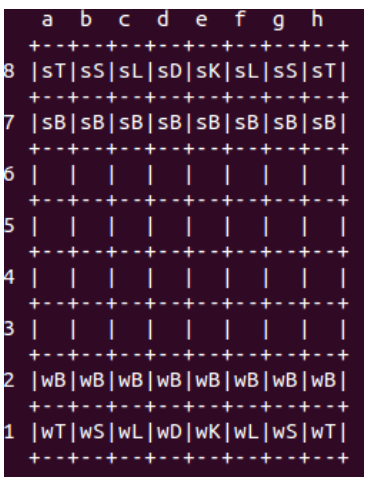
Keine Überprüfung auf Einhaltung der Zugregeln.

`ziehe(figuren, 5, 2, 5, 4);` für Zug e2 nach e4

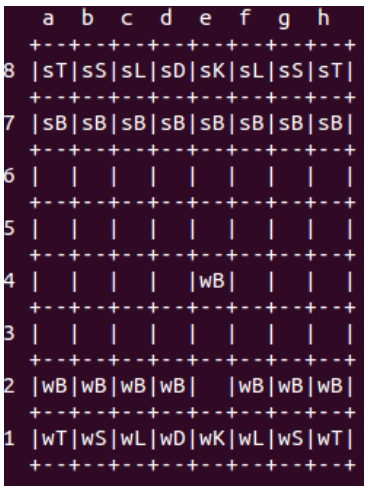
Schachspielsimulation – Italienische Eröffnung

schachzuege2.c – Endteil der main-Funktion

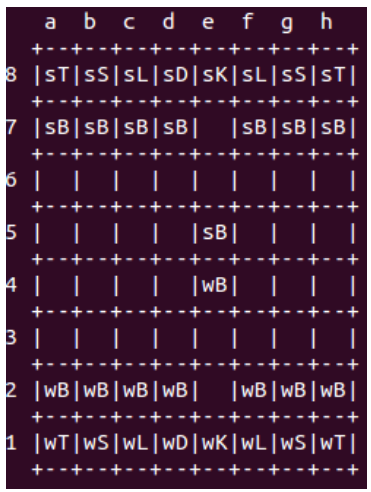
```
  zeichneSchachbrett(figuren); //Anfangsaufstellung der Figuren
  //Italienische Eroeffnung
  ziehe(figuren, 5,2, 5,4); //weiss beginnt und zieht Bauer e2 nach e4
  ziehe(figuren, 5,7, 5,5); //schwarz zieht Bauer e7 nach e5
  ziehe(figuren, 7,1, 6,3); //weiss zieht Springer g1 nach f3
  ziehe(figuren, 2,8, 3,6); //schwarz zieht Springer b8 nach c6
  ziehe(figuren, 6,1, 3,4); //weiss zieht Laeufer f1 nach c4
  ziehe(figuren, 6,8, 3,5); //schwarz zieht Laeufer f8 nach c5
  return 0;
}
```



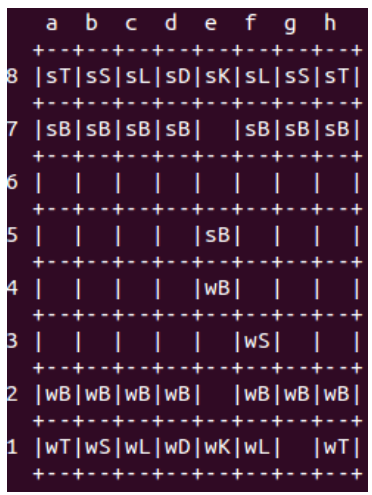
weiß zieht $e2 \rightarrow e4$



schwarz zieht $e7 \rightarrow e5$



weiß zieht $g1 \rightarrow f3$



schwarz zieht $b8 \rightarrow c6$