Programmiermethoden in der Mathematik WS 02/03
Abgeleitete Datentypen – Felder und Strukturen

- **Felder (Arrays)** bestehen aus mehreren Komponenten *desselben Datentyps*. Sie können eindimensional (mathematisch wie ein Vektor) oder mehrdimensional (wie eine Matrix oder ein höherdimensionales Objekt) sein. Die Bezeichnung der Komponenten erfolgt durch *Indizes*, die *immer bei 0 beginnen*.

  – Deklaration: Die Länge eines *statischen* Feldes muss beim Compilieren bekannt sein, also

  ```
  double x[10];
  ```

  oder

  ```
  #define n 10
  ...
  double x[n];
  ```

  Der `g++` erlaubt auch

  ```
  int n=10;
  double x[n];
  ```

  Alle drei Varianten erzeugen ein eindimensionales Feld mit $n = 10$ `double`-Werten. Analog ergibt

  ```
  double x[n][m];
  ```

  ein zweidimensionales Feld mit $n \cdot m$ `double`-Werten. Analog für höherdimensionale Felder.

  – Zugriff (auf die i-te Komponente für $i = 0, \ldots, n-1$):

  ```
  y=x[i];
  ```

  Ein häufiger Fehler ist es, im obigen Beispiel auf die Komponente `x[n]` zuzugreifen. Der Compiler liefert keinen Fehler, auch das Programm bricht nicht ab, der Wert `x[n]` ist aber i.A. nicht sinnvoll.

  – Array-Zuweisungen können beim `g++` als Ganzes erfolgen:

  ```
  double x[n],y[n];
  x=y;
  ```

  Bei anderen Compilern muss man komponentenweise arbeiten:

  ```
  for (i=0;i<n;i++)  x[i]=y[i];
  ```

  Mehrdimensionale Arrays werden im Speicher allerdings auch hintereinander abgelegt, und zwar zeilenweise bzw. so, dass sich der hinterste Index am schnellsten ändert. Es ist daher am effektivsten, eine geschachtelte *for*-Schleife z.B. für ein 2-D Array in der folgenden Form zu schreiben:

  ```
  for (i=0;i<n;i++)
       for (j=0;j<m;j++)
            x[i][j]=...;
  ```

  und nicht umgekehrt.

  – Initialisierung ist auch möglich in der Form

  ```
  double x[4] = {0.0, 1.0, 2.0, 3.0};   (liefert x = (0.0, 1.0, 2.0, 3.0))
  double x[4] = {0.0, 1.0};             (liefert x = (0.0, 1.0, 0.0, 0.0)).
  ```

  Es ist auch die Variante

  ```
  double x[] = {0.0, 1.0};              (liefert x = (0.0, 1.0))
  ```

  möglich, da mit der Initialisierung die Länge festgelegt wird.

  – Ein Feld kann *nicht Rückgabewert einer Funktion* sein.

- **Zeichenketten** kann man als Felder von Zeichen (`char`) definieren. Am Ende der Kette wird automatisch das Zeichen `'\0'` gesetzt, das bei der Längenangabe berücksichtigt werden muss.

  – Deklaration (Beispiele):

  ```
  char txt1[20];
  char txt2[]="Text";
  ```

  – Operationen und Zuweisungen können elementweise erfolgen. Mit

  `txt1[5] ='\0';` oder äquivalent `txt1[5]=0;`

zeigt man z.B. das Ende der Kette nach dem fünften Zeichen an, der Rest wird bei einer Ausgabe

```
cout << txt1 << endl;
```

dann ignoriert.

Eine komfortablere Variante, um Felder und Zeichenketten zu behandeln, werden wir noch kennenlernen.

- **Strukturen** (Records) können aus mehreren Komponenten *verschiedener Datentypen* bestehen. Die Bezeichnung der Komponenten erfolgt durch *frei wählbare Namen*.
  - Die Deklaration beginnt mit dem Schlüsselwort `struct`, z.B. deklariert

    ```
    struct
    {
       int n;
       double x;
    } s1;
    ```

    eine Variable `s1`, die eine Struktur bestehend aus einer `int`- und einer `double`-Komponente ist. Alternativ kann zuerst separat mit

    ```
    struct T
    {
       int n;
       double x;
    };
    ```

    ein Typ `T` definiert und anschließend Variablen dieses Typs deklariert werden:

    ```
    T s1,s2;
    ```

  - Zugriff auf die Elemente erfolgt durch einen Punkt zwischen dem Namen der Strukturvariable und der Komponente, im obigen Beispiel etwa:

    ```
    s1.n=1;
    s1.x=0.0;
    ```

  - Strukturen können als Ganzes kopiert werden, *wenn sie vom selben Typ sind*, also im obigen Beispiel

    ```
    s1=s2;
    ```

  - Eine Struktur kann Rückgabewert einer Funktion sein.
  - Felder von Strukturen sind möglich:

    ```
    T s[n];
    ```