Inhalt

12 Strukturierte Datentypen - Strukturen

Eigenschaften von Feldern
- Ein Feld fasst Objekte zusammen, die für das Programm eine Einheit bilden.
- Die Komponenten eines Feldes können sequentiell verarbeitet werden.
- **Alle Komponenten eines Feldes besitzen stets den gleichen Typ.**

Insbesondere ist der letzte Punkt eine Einschränkung, unterschiedliche Typen sind oft wünschenswert. Deshalb verwendet man **Strukturen** als weiteren Datentyp.

12.1 Definition und Deklaration von Strukturtypen und Strukturen

Struktur $=_{df}$ **Tupel über eine Menge von Objekte (verschiedenen Typs)**

Strukturen sind strukturierte Datentypen, die für das Programm eine Einheit bilden und dabei Daten unterschiedlichen Typs zusammenfassen.

Analog den Variablen haben Strukturen einen **Namen** und einen speziell zu deklarierenden **Typ**, aber **keinen Wert**. Die Objekte, die durch eine Struktur zusammengefasst werden, heißen **Strukturelemente**.

Deklaration von *Strukturtypen*
Syntax: **struct** *Strukturtypname* **{** *Deklaration* **. . . } ;**

```
struct kartes { float x , y ; } ;
struct polar { float rho , phi ; } ;
```

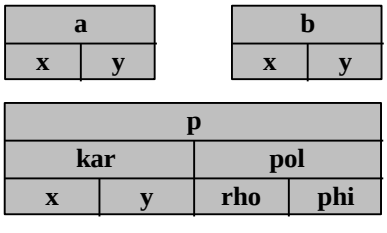
Strukturtypname:
- *C-Name*

Deklaration:
- Festlegung der Anzahl, der Namen und der Typen der *Strukturelemente*.
- Die *Strukturelemente* können selbst einen *strukturierten Typ* besitzen.

```
struct punkt
{ struct kartes kar ; struct polar pol ; } ;
```

Deklaration von *Strukturen*
Syntax: **struct** *Strukturtypname Strukturname* **;**

```
struct kartes a , b ;
```




```
struct punkt p ;
```

Strukturname:
- *C-Name*
- Mehrere Objekte **unterschiedlichen Typs** werden unter dem *Strukturnamen* zusammengefasst.
- Der *Strukturname* ist der **symbolische Bezeichner** für die Feldanfangsadresse im Speicher.

Zusammenfassung beider Deklarationen
Syntax: **struct** *Strukturtypname* **{** *Deklaration* **...** **}** *Strukturname* **;**

```
struct kartes { float x , y ; } a , b ;
struct punkt
{ struct kartes kar ; struct polar pol ; } p ;
```

Zugriff auf Strukturelemente durch den Punkt-Operator
Syntax: *Strukturname* **.** *Strukturelement*

a . x	**a . y**	**b . x**	**b . y**
p . kar . x	**p . kar . y**	**p . pol . rho**	**p . pol . phi**

Operationen
Operationen mit **Strukturelementen** sind analog den *einfachen Variablen* erlaubt.

=> **a . x *= 2 ;**

Operationen mit **Strukturen** sind *nur für Zuweisungen* möglich:
Zuweisungsoperatoren wirken elementweise, die Gleichheit des Strukturtyps vorausgesetzt.

=> **a = b ;** ist äquivalent mit **a . x = b . x ;** und **a . y = b . y ;**

Explizite Anfangswertzuweisung
Syntax: **struct** *Strukturtypname Strukturname* **=** *Wertemenge* **;**
struct *Strukturtypname* **{** *Deklaration* **...** **}** *Strukturname* **=** *Wertemenge* **;**

Wertemenge:
- Geordnete Menge von *C-Konstanten* entsprechenden *Typs*.

Beispiel einer zusammengefassten Deklaration mit Initialisierung

```
struct termin
{
  int tag; char monat[ 6]; int jahr;
} datum = { 15, "Dez", 1994};
```

datum.tag => 15, **datum.monat** => Dez, **datum.jahr** => 1994

12.2 Strukturen als Funktionsparameter und Funktionswert

Das folgende Programm rechnet kartesische Koordinaten in Polarkoordinaten um. An diesem Beispiel soll der Umgang von Strukturen auch im Zusammenhang mit Funktionen gezeigt werden. (Beachte: π ist nicht Bestandteil des ansi-Standard, π ≈ 3.14159265359)

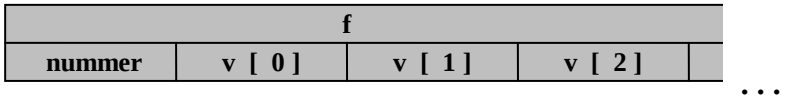
poltokar.c

```
/*
 * Polarkoordinaten  ->  kartesische Koordinaten
 */

# include <stdio.h>
# include <math.h>                          /* sin cos */
# define M_PI 3.141593

                    /* Deklaration eines Strukturtyps */
struct kartes{ float x, y;};
                    /* Deklaration eines Strukturtyps */
struct polar{ float rho, phi;};
        /* Funktion mit einer Struktur als Parameter */
             /* und einer Struktur als Funktionswert */
struct kartes kartesisch( struct polar);

int main()
{
  struct polar a;    /* Deklaration einer Struktur a */
  struct kartes b;   /* Deklaration einer Struktur b */

  printf( "Polarkoordinaten (Abstand/Winkel): ");
                     /* Zugriff auf Strukturelemente */
  scanf( "%f%f", &a.rho, &a.phi);
    /* Funktionsaufruf mit Struktur als Argument und */
                                /* als Funktionswert */
  b = kartesisch( a);
                     /* Zugriff auf Strukturelemente */
  printf( "\nKartesischen Koordinaten : (%f,%f).\n",
            b.x, b.y);

  return 0;
}


/*
 * Koordinatenumrechnung
 */
struct kartes kartesisch( struct polar p)
{
  struct kartes k;
  p.phi *= M_PI / 180;   /* Winkelmass --> Bogenmass */
  k.x = p.rho * cos( p.phi);           /* Umrechnung */
  k.y = p.rho * sin( p.phi);
  return k;
}
```

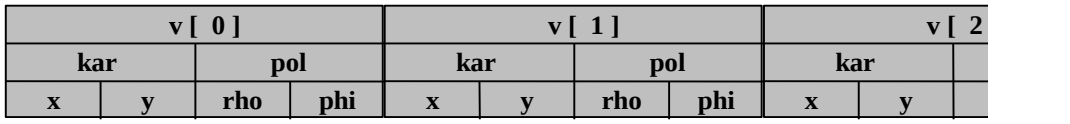
12.3 Felder und Strukturen

Im Zusammenhang mit Feldern kann man vereinbaren:

Struktur mit Feldern: **struct feld { int nummer , v [10] } f ;**



Feld von Strukturen: **struct punkt v [10] ;**



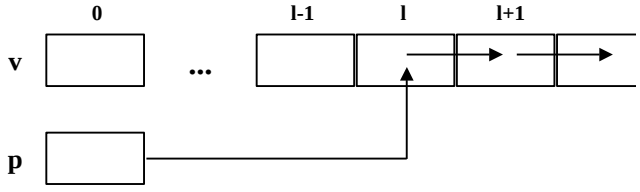
Im folgenden sollen Punktmengen in Polarkoordinatendarstellung nach ihren kartesischen Koordinaten (x primär) geordnet werden. Das Sortieren erfolgt beim Einlesen in ein Feld von Strukturen als Komponenten.

sort.c

```
/*
 * Sortieren von Punkten mit Feldern von Strukturen
 */

# include <stdio.h>
# include <math.h>
# define MAX 10
# define M_PI 3.141593

struct kartes{ float x, y;};
struct polar{ float rho, phi;};
struct punkt{ struct kartes kar; struct polar pol;};

/*
 * Koordinatenumrechnung
 */
struct kartes kartesisch( struct polar);

/*
 * Sortierfunktion, Zeiger auf Struktur als Parameter
*/
void sortier_ein( struct punkt *, struct punkt, int);

int main()
{
  struct punkt v[ MAX], p;  /* v Feld von Strukturen */
  int gelesen = 0, i;

  printf( "Polarkoordinaten (Abstand/Winkel):\n");
  printf( "Schliessen Sie die Eingabe mit ^d ab!\n");
  printf( "\n");
```

```
/* Einlesen der Elemente der geschachtelten Struktur */
  while( gelesen < MAX &&
     ( scanf( "%f%f", &p.pol.rho, &p.pol.phi) != EOF ))
  {
    p.kar = kartesisch( p.pol);
    sortier_ein( v, p, gelesen++);
  }

                      /* Ausgabe der sortierten Folge */
  printf( "sortiert (polar/kartesisch):\n");
  for( i = 0; i < gelesen; i++)
  {
    printf( "(%f,%f) ", v[ i].pol.rho, v[ i].pol.phi);
    printf( "(%f,%f)\n", v[ i].kar.x, v[ i].kar.y);
  }

  return 0;
}


struct kartes kartesisch( struct polar p)
{ ...
}


void sortier_ein
( struct punkt *v, struct punkt p, int l)
{
  while( l > 0 && ( v[ l-1].kar.x > p.kar.x ||
                  ( v[ l-1].kar.x == p.kar.x &&
                    v[ l-1].kar.y > p.kar.y)))
  {
    v[ l] = v [ l-1];       /* 16 Bytes umgespeichert */
    l--;
  }

  v[ l] = p;

  return;
}
```

In der Funktion **sortier_ein** ist **l** der Index der freien Struktur, in welche der neue Punkt eingefügt wird. Alle darauffolgenden Strukturen wurden vorher um einen Index nach hinten kopiert.



12.4 Zeiger und Strukturen

12.4.1 Zeiger auf Strukturen

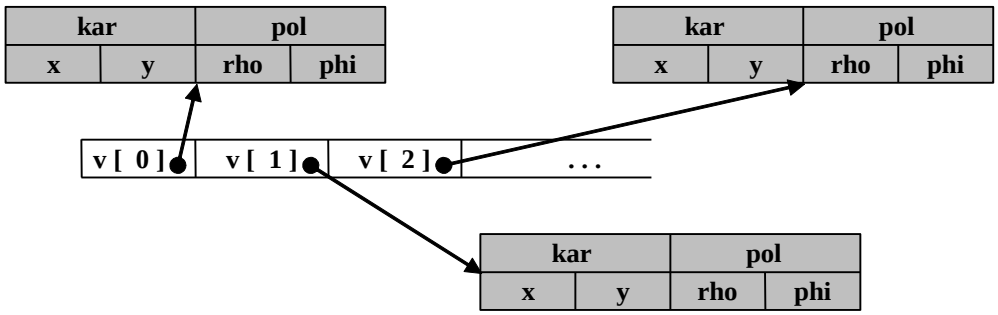
```
struct punkt * z , p;
```

Referenzierung eines Zeigers auf eine Struktur: **z = &p ;**
Dereferenzierung eines Zeigers auf eine Struktur: **(* z) . pol . rho = 45 ;**
Dereferenzierung mittels **Pfeil-Operator:** **z -> pol . rho = 45 ;**

Feld von Zeigern auf Strukturen

Im Beispiel der Punktfelder (letztes Kapitel) belegte jede Komponente des Feldes mindestens 16 Bytes (float, mindestens 4 Bytes) => **Aufwendiges Umspeichern**

Günstiger wäre ein Feld von Zeigern auf Strukturen, dann ließe sich das Problem durch Umspeichern von Adressen lösen. Jeder Zeiger verweist auf einen der eingelesenen Punkte. Beim Sortieren müssen jetzt nur noch die Zeiger des Feldes umgespeichert werden.



Nachteilig ist immer noch, dass man vor der Dateneingabe angeben muss, um wie viel Strukturen es sich handelt. Besser wäre es, wenn eine Struktur selbst auf ihre nächste verweisen könnte.

12.4.2 Struktur mit Zeigern

Rekursive Funktionen (Kapitel Rekursionen) sind erlaubt. Es muss jedoch immer ein Abbruchkriterium geben.

Rekursive Strukturen würden unendlich große Datenmengen erzeugen. Ein Abbruchkriterium lässt sich hier nicht definieren.

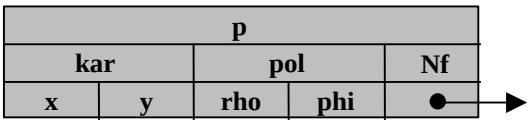
Rekursive Strukturtypdeklarationen sind deshalb grundsätzlich verboten!

```
struct rekursiv { struct rekursiv r ; } ;          /* nicht erlaubt */
```

Partielle Deklarationen von Strukturtypen *mit Zeigern* auf Strukturen sind erlaubt. Abbruch durch NULL - Zeiger möglich!

Struktur mit Zeigern:

```
struct  punkt
{  struct  kates  kar ;
   struct  polar  pol ;
   struct  punkt  * nf ;        /* Zeiger auf den Nachfolger */
} p;
```



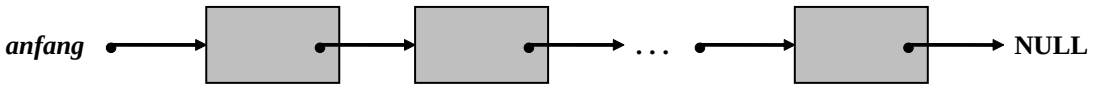
Partielle Deklarationen

Der Name des Strukturtyps wird innerhalb ihrer Deklaration verwendet, obwohl diese noch nicht vollständig abgeschlossen ist.

Einfach verkettete lineare Listen

Strukturtypen mit Zeigen ermöglichen das Hintereinanderverketten von Strukturen. Solch eine Folge von Strukturen nennt man **einfach verkettete lineare Liste**.

Man benötigt einen Zeiger auf den Listenanfang, d.h. auf das erste Element. Jedes Element verweist auf seinen Nachfolger. Das letzte Element, welches keinen Nachfolger besitzt, wird durch den NULL - Zeiger gekennzeichnet.



Kreuzverweis auf Strukturen

```
struct  s1 {  struct  s2  * z ; } ;
struct  s2 {  struct  s1  * z ; } ;
```