Strukturen

- **Strukturtyp**: - selbst definierter Datentyp
  - zusammengesetzt aus Komponenten *verschiedener Typen*
- Variable von Typ Struktur (Verbund/Record) kann Datensatz speichern

Personalnr.	Nachname	Vorname	Straße	Hausnr.	PLZ	Wohnort	Gehalt
`int`	`char[20]`	`char[20]`	`char[20]`	`int`	`int`	`char[20]`	`float`

- *Komponenten* haben einen eigenen Namen und Typ (statt Index)

Deklaration eines Strukturtyps

- **`struct`** *name* `{`
  komponententyp$_1$ *komponentenname*$_1$`;`
  komponententyp$_2$ *komponentenname*$_2$`;`
  `...`
  komponententyp$_n$ *komponentenname*$_n$`;`
  `};`

- **`struct`** ist Schlüsselwort ➔ **Datentyp**: **`struct`** *name*
- *name* ist Bezeichner (sog. Etikett)
- Anzahl der Komponenten bei Deklaration festgelegt
- Semikolon nach } (kein Anweisungsblock!)

Strukturvariablen

- **Definition**: `struct` *name variablenname*;
  ➔ zusammengesetzte Variable vom Typ **`struct`** *name*
- besteht aus mehreren Komponentenvariablen
  ➔ bei Stukturtypdeklaration vereinbart
- gleichzeitige Vereinbarung von Strukturtyp und -variablen möglich:

```
struct point {
float x;
float y;
} punkt1;

struct point punkt2, punkt3;
```

Beispiel

```
struct adresse {
    char strasse[20];
    int hausnummer;
    int plz;
    char wohnort[20];
};
struct einwohner {
    char name[20];
    char vorname[20];
    struct adresse anschrift;
};
struct einwohner meyer, mueller;
struct einwohner wohngebiet[100]; // Array aus 100 Einwohnern
```

Zugriff auf Komponentenvariablen

1. **Punktoperator**: *Strukturvariable.Komponentenvariable*

• Beispiele:
```
punkt1.x
meyer.name
meyer.anschrift.plz
```

• lesender und schreibender Zugriff:
```
meyer.adresse.plz = 14482;
strcpy(meyer.anschrift.wohnort, "Potsdam");
printf("%5d %s\n",
meyer.anschrift.plz,
meyer.anschrift.wohnort);
```

Zugriff auf Komponentenvariablen

2. **Pfeiloperator**: *Pointer_auf_Strukturvariable->Komponentenvariable*
   - Beispiel: `(&punkt1)->x`
     `(&meyer)->anschrift.plz`
   - Pfeil: Minuszeichen und Grösserzeichen
   - lesender und schreibender Zugriff

Initialisierungslisten

- nur direkt bei der Definition der Strukturvariablen
- wie bei Arrays (mit Ausdrücken passenden Typs)
- Beispiel:

```
struct einwohner meyer = {
    "Meyer",
    "Peter",
    {"Weberplatz",
     98,
     14482,
     "Potsdam"
    }
};
```

Strukturen als Parameter und Rückgabewerte von Funktionen

- Voraussetzung: Deklaration des Strukturtyps *außerhalb* und vor den Funktionen
- Übergabe wie einfache Datentypen:
  ➔ komplett als zusammengesetzte Variable
- Alternative: Übergabe eines Pointers auf Strukturtyp
  (call-by-value beachten!)

Ausblick: Dynamische Speicherverwaltung

- Dynamische Variablen
  - kein Name, keine explizite Vereinbarung
  - bei Bedarf zur Laufzeit durch Bibliotheksfunktionen angelegt
  - Gültigkeit bis Speicherfreigabe, unabhängig von Programmstruktur
- Bibliotheksfunktionen zur dynamischen Speicherverwaltung
  - in `<stdlib.h>`
  - zum Anfordern von Speicher: liefern Pointer zurück, z.B.:
    `void * malloc (size_t` *size*`)`
  - zum Freigeben von Speiocher: werden mit Pointer aufgerufen:
    `void free (void *` *pointer*`)`

Komplexere Typen - Zusammenfassung

Beispiel	`alpha` **ist ...**
`int * alpha`	Pointer auf `int`
`float alpha[10]`	Array von 10 `float`-Komponenten
`char * alpha[20]`	Array von 20 Pointern auf `char` $\rightsquigarrow$ Array von 20 Strings
`char (* alpha) [10]`	Pointer auf ein Array von 10 `char`-Komponenten
`struct point alpha`	zusammengesetzte Variable vom Strukturtyp `point` (`point` muss zuvor definiert sein)

Vereinbarung eigener Typnamen

- Vereinbarung eines Aliasnamens für bereits deklarierte Datentypen
- `typedef` *Datentyp Aliasname*;
- `typedef int integer;`
- Anwendung:
  - Vereinfachung von Typnamen
    - `typedef struct einwohner EINWOHNER;`
      - ➔ `EINWOHNER meyer;`
    - `typedef unsigned long long ULL`
  - Vorbereitung Portierung maschinenabhängiger Datentypen
    - `typedef int INT;  typedef short INT;`

Prozesse

- **Definition eines Prozesses**
  Ein Prozess ist ein sich in der Ausführung befindliches Programm mit seiner Ausführungsumgebung (Environment).

- **Die Prozessverwaltung**
  - Da Unix ein echtes *Multitaskingsystem* ist, werden mehrer Programme (Prozesse, Tasks) gleichzeitig ausgeführt.
  - Wobei *Gleichzeitig* nicht ganz stimmt. Ein Prozessor (CPU) kann immer nur mit einem Prozess arbeiten.
  - Soll wirklich gleichzeitig das mehrere Programme parallel ablaufen, werden entweder mehrere Rechner oder mehrere Prozessoren benötigt.
  - Jeder Prozess hat einen extra virtuellen Adressraum. Damit kann keiner der Prozesse den anderen Stören oder Beeinflussen.
  - Zugegriffen wird auf die einzelnen Prozesse nacheinander von der CPU.
  - Wie lange und in welcher Reihenfolge, entscheidet der Prozess-Scheduler.
  - Unix ist ein *preemptives* Multitaskingsystem. Dies bedeutet das Unix selbst entscheidet wie lange ein Prozess die CPU benutzen darf und wann der nächste Prozess an der Reihe ist.
  - Will man in den Prozess-Scheduler eingreifen, kann man dies als `root` mit dem Kommando `nice` machen.

Prozesse

- **Die Prozessverwaltung**
  - Welche Prozesse im Augenblick laufen, kann man mit dem Kommando `ps` in Erfahrung bringen.

  Beispiel:

  ```
  ps -x
  ```

  Nun wird ein Liste der momentan laufenden Prozesse ausgegeben.

Prozesse

- **Der Systemaufruf `fork`** - Kreieren eines Prozesses
  Zwei Möglichkeiten der Prozesserzeugung
  - Austausch eines Prozesses durch einen anderen in der gleichen Arbeitsumgebung
  - Erzeugung eines neuen Prozesses durch Funktionsaufruf `fork()`

```
#include <sys/types>
#include <unistd.h>
 pid_t fork(void);
```

`pid_t` ist in Primitiver Datentyp, der für Prozess-ID oder auch Gruppen-Prozess-ID steht.

Prozesse

- **Der Systemaufruf `fork`** - Kreieren eines Prozesses

Beispiel für `fork`

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(int argc, char **argv)
{ pid_t kind1;
  while(1)
  { switch(kind1=fork())
    { case -1 : exit(0); break;
      case  0 : system("konsole -caption 'kind' -vt_sz 20x10"); break;
      default :
        printf("Elternprozess: Warte auf das Ende vom Kindprozess!\n");
         printf("Um den Prozess %s zu killen gib bitte ein : ",*argv);
         printf("kill -9 %d (%d=PID)\n",kind1,kind1);exit(1);
    }
  }
  return 0;
}
```