2.7 Strukturierte Datentypen

Motivation:

Strukturierte Datentypen entstehen durch eine Zusammenfassung von Grunddatentypen und bereits definierten strukturierten Datentypen. In C besitzen Felder und Strukturen große Bedeutung.

2.7.1 Strukturen (struct)

- Eine Struktur ist eine Zusammenfassung von Objekten unterschiedlichen Typs
- eingeleitet wird eine Strukturvereinbarung durch das Schlüsselwort `struct`
- als Komponenten einer Struktur können beliebige Datentypen vereinbart werden, also auch Strukturen und Felder
- Eine Struktur kann sich nicht selbst als Komponente enthalten

```
struct datum /*Deklaration des Typs */
{
   short jahr;
   char monat[4];
   short tag;
};
```

- Die Deklaration reserviert keinen Speicherplatz, sondern legt lediglich die Typen der zu diesem Strukturtyp gehörenden Komponenten fest. Die Länge der Struktur ergibt sich nicht aus der Summe der Elementlängen

2.7.1 Strukturen deklarieren

```
struct datum heute, hochzeits_tag; /* Definition */
```

- Dadurch erfolgt die Definition der Variablenbezeichner `heute` und `hochzeits_tag` als Strukturinstanzen des Typs `struct datum` und damit auch die Speicherplatzvergabe.
- Deklaration und Definition in einer Anweisung:

```
struct datum {                          /* Deklaration des Typs */
     short jahr;
     char monat[4];
     short tag;
};
```

2.7.1 Strukturen deklarieren

- Wenn Sie den Typnamen dieser Struktur nicht benötigen, kann sie auch ohne deklariert werden:

```
struct {
   int seite;
   char titel[30];
} lib;
```

- Strukturen können natürlich ebenso wie normale Datentypen direkt bei der Deklaration mit Werten initialisiert werden:

```
struct index {
   int seite;
   char titel[30];
} lib = { 308, "Strukturen" };


struct index lib = { 55, "Einführung in C" };
```

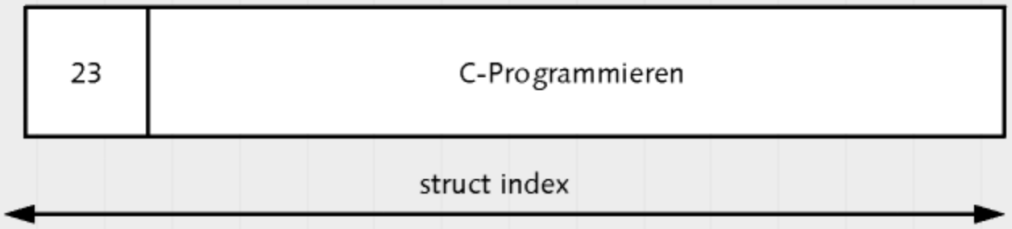
2.7.1 Zugriff auf Strukturen

- Auf die einzelnen Variablen einer Struktur greifen Sie mithilfe des Punktoperators (.) zu. Ansonsten erfolgen die Initialisierung und der Zugriff wie bei normalen Variablen.

```c
struct index {
   int seite;
   char titel[30];
};

int main(void) {
   struct index lib;

   lib.seite = 23;
   strcpy(lib.titel, "C-Programmieren");
   printf("%d, %s\n",lib.seite, lib.titel);
   return EXIT_SUCCESS;
}
```



2.7.1 Strukturen deklarieren (C99)

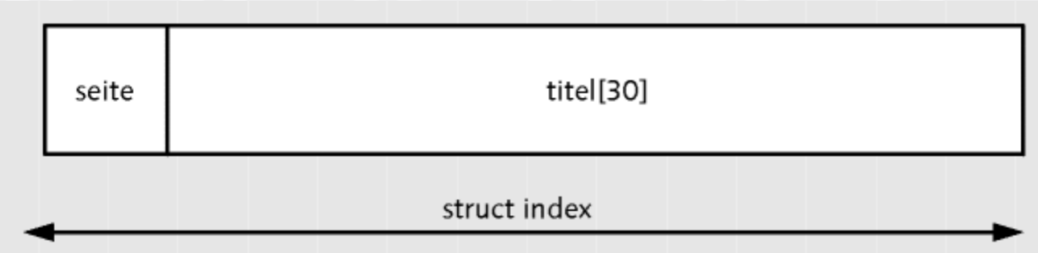
- Ab dem C99-Standard ist es auch möglich, nur bestimmte Elemente einer Struktur zu initialisieren. Als Initialisierer wird hierbei ein sogenannter Elementbezeichner verwendet.

```
stuct adres {
    char vname[20];
    char nname[20];
    long PLZ;
    char ort[20];
    int geburtsjahr;
};
```

```
struct adres adressen = {
    .nname = "Wolf",
    .ort   = "Mering"
};
struct adres adressen = {
    "Jürgen",          // geht automatisch an vname
    "Wolf",            // geht automatisch an nname
    .ort = "Mering"    // nötig, weil das 3. Element PLZ ist
};
struct adres adressen = {
    .PLZ    =  1234,
    "Mering",            // geht autmatisch an ort
    1974                 // geht automatisch an geburtsjahr
};
```

2.7.2 Speicher von Strukturen

```
struct index
{
    int seite;
    char titel[30];
} meinIndex;
```



- In diesem Beispiel wurde eine Struktur vom Typ index deklariert. Diese Struktur kann einen int-Wert und einen String von 30 Zeichen Länge aufnehmen. Folglich wäre die Gesamtgröße der Struktur 34 Bytes (auf 16-Bit-Systemen entsprechend 32 Bytes).

2.7.3 Arrays von Strukturen

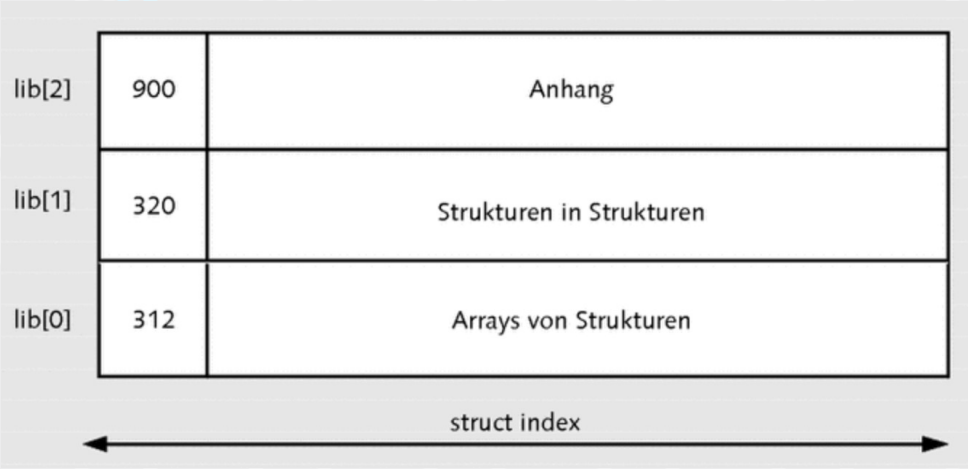
Bei Arrays von Strukturen gilt dasselbe Prinzip wie im Abschnitt zuvor dargestellt. Die Wertzuweisung funktioniert ebenfalls wie bei den normalen Arrays, nämlich mithilfe des Indizierungsoperators ([]).

```
struct index {
   int seite;
   char titel[30];
};

int main(void) {
   int i;
   struct index lib[3];

   lib[0].seite=312;
   strcpy(lib[0].titel, "Arrays von Strukturen");
   lib[1].seite=320;
   strcpy(lib[1].titel, "Strukturen in Strukturen");
   lib[2].seite=900;
   strcpy(lib[2].titel, "Anhang");

   for(i=0; i<3; i++)
      printf("Seite %3d\t %-30s\n", lib[i].seite, lib[i].titel);
   return EXIT_SUCCESS;
}
```



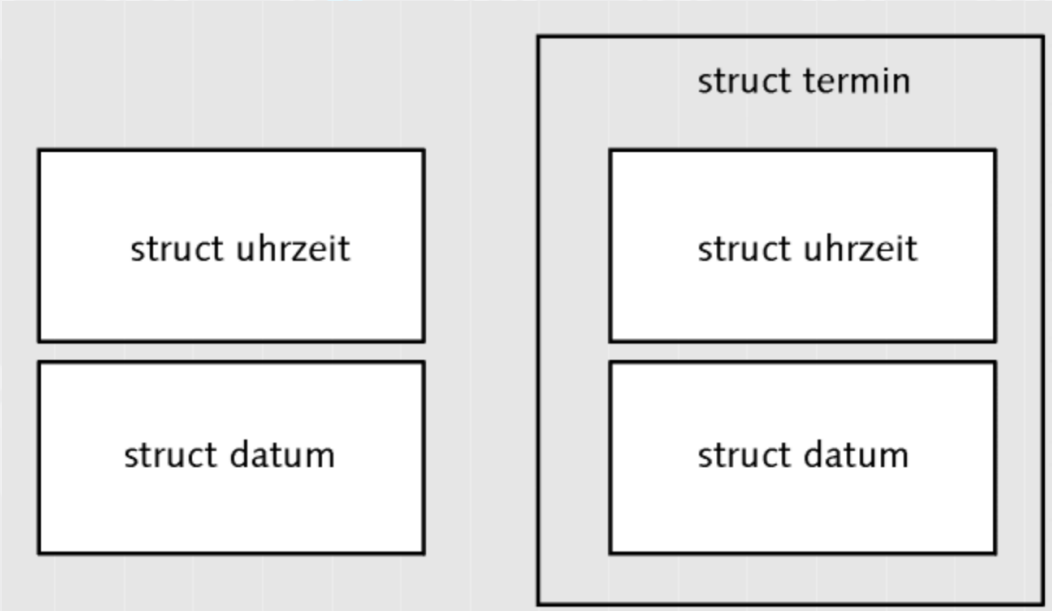
2.7.4 Strukturen in Strukturen

Neben der Kombination von Arrays und Strukturen können auch Strukturen innerhalb von Strukturen verwendet werden.

```
struct uhrzeit {
   unsigned int stunde;
   unsigned int minute;
   unsigned int sekunde;
};

struct datum {
   unsigned int tag;
   unsigned int monat;
   int jahr;
};

struct termin {
   struct datum d;
   struct uhrzeit z;
} t;
```



2.7.5 Zeiger auf Strukturen

Definition eines Zeigers auf Struktur datum:

```
struct datum *pd;
```

Der Operator -> dient der Bezugnahme auf Komponenten einer Struktur, die über einen Zeiger adressiert wird.

```
pd->jahr = 1989;
```

ist die vereinfachte Schreibweise für

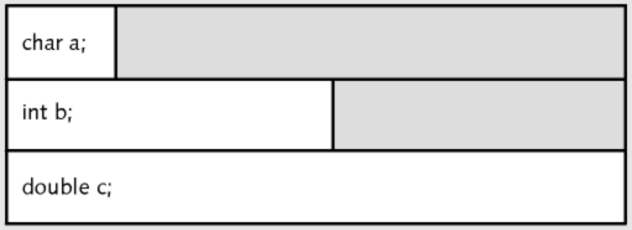
```
(*pd).jahr = 1989;
```

Der Operator . hat höheren Vorrang als *

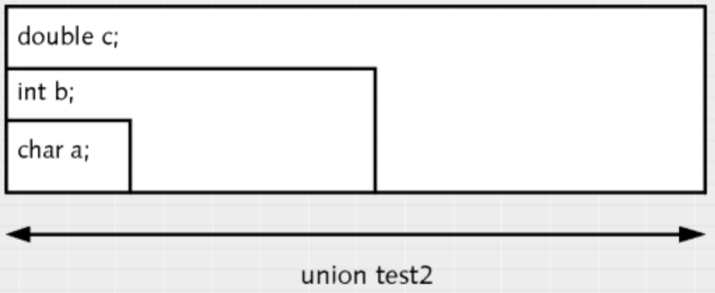
2.8.1 Union

Eine weitere Möglichkeit, Daten zu strukturieren, sind Unions (auch Varianten genannt). Abgesehen von einem anderen Schlüsselwort, bestehen zwischen Unions und Strukturen keine syntaktischen Unterschiede. Der Unterschied liegt in der Art und Weise, wie mit dem Speicherplatz der Daten umgegangen wird.

```
struct test1 {
   char a;
   int b;
   double c;
};
```




```
union test2 {
   char a;
   int b;
   double c;
};
```



Eine Union ist eine spezielle Form des Typs `struct`. Jede Komponente einer Union-Instanz wird auf den gleichen Speicherbereich abgebildet, d.h. zu einer Zeit ist immer nur ein Wert in einer Union gespeichert.

2.8.2 Der Aufzählungstyp »enum«

Mit dem Aufzählungstyp wird ein Wertebereich explizit durch eine Bezeichnerliste festgelegt. Die Bezeichner sind einfach eine symbolische Konstantenrepräsentation.

```
enum zahl { NU_LL, EINS, ZWEI, DREI, VIER};

int main(void) {
   enum zahl x;
   x=NU_LL;
   printf("%d\n",x);

   x=EINS;
   printf("%d\n",x);

   x=ZWEI;
   printf("%d\n",x);

   x=DREI;
   printf("%d\n",x);

   x=VIER;
   printf("%d\n",x);
   return EXIT_SUCCESS;
}
```

Bei Ausführung des Programms werden die Zahlen von null bis vier auf dem Bildschirm ausgegeben.

2.8.2 Der Aufzählungstyp »enum«

In der Regel beginnt der Aufzählungstyp, sofern nicht anders angegeben, mit 0; also NU_LL=0. Das nächste Feld hat, wenn nicht anders angegeben, den Wert 1.

```
enum zahl { NU_LL=0, EINS=1 ,ZWEI=2 ,DREI=3 ,VIER=4 };
```

Wird `enum` hingegen so benutzt:

```
enum farben { rot, gelb=6, blau, gruen };
```

würden folgende Konstanten definiert werden:

```
enum farben { 0, 6, 7, 8 };
```

2.8.3 Typendefinition mit »typedef«

Eine Typdefinition ist nur eine Neubenennung, kein neuer Typ, d.h. `Bezeichner` wird als Synonym für typ benutzt.

```
typedef Typendefinition Bezeichner;
```

```
struct adres {
   char vname[MAX];
   char nname[MAX];
   long PLZ;
   char ort[MAX];
   int geburtsjahr;
} adressen[100];

typedef struct adres ADRESSE;

ADRESSE neueadressen[100];
```

2.8.3 Typendefinition mit »typedef«

Die Typdefinition kann ebenso auf andere Variablen angewendet werden. Recht häufig sind folgende Definitionen zu sehen:

```
typedef unsigned char BYTE;     // 1 Byte = 8 BIT
typedef unsigned int WORD;      // 1 WORD = 16 BIT
typedef unsigned long DWORD;    // 1 DOUBLE WORD = 32 BIT
typedef unsigned double QWORD; // 1 QUAD WORD = 64 BIT
typedef unsigned int uint;
typedef unsigned char uchar;
```