13. Record (Strukturen) und varianter Record (Unions)

13.1. Records (Strukturen)

Während Vektoren eine Zusammenfassung von Objekten gleichen Typs sind, handelt es sich bei einer Struktur um eine Zusammenfassung von Objekten möglicherweise verschiedenen Typs zu einer Einheit. Die Verwendung von Strukturen bietet Vorteile bei der Organisation komplexer Daten. Beispiele sind Personaldaten (Name, Adresse, Gehalt, Steuerklasse, usw.) oder Studentendaten (Name, Adresse, Studienfach, Note).

Strukturen werden mit Hilfe des Schlüsselwortes `struct` vereinbart.

`struct` *`Strukturname`* `{` *`Variablen`* `}` *`Strukturvariable(n)`* `;`

Strukturname ist optional und kann nach seiner Definition für die Form (den Datentyp) dieser speziellen Struktur verwendet werden, d. h. als Abkürzung für die Angaben in den geschweiften Klammern. Variablen innerhalb der Struktur werden wie normale Variable vereinbart. Struktur- und Variablennamen innerhalb einer Struktur können mit anderen Variablennamen identisch sein ohne das Konflikte auftreten, da sie vom Compiler in einer separaten Tabelle geführt werden.

Der Aufruf der einzelnen Elemente erfolgt dann nicht über Indizes, sondern über deren Namen. Beispiel (für Definition/Deklaration):

`struct Datum`
`{  int tag;`
`  int monat;`
`  int jahr;`
`  char mon_name[4];`
`};`
	Legt nur die Form der Struktur `Datum` fest

`struct Datum {`
`  int tag;`
`  int monat;`
`  int jahr;`
`  char mon_name[4];`
`} stGeburt, stHeute;`
	Erzeugt zusätzlich die Strukturvariablen `stGeburt` und `stHeute`

`struct Point`
`{  double  spx, spy;`
`  int  farbe;`
`  char  label;`
`}  stSpot;`
	`Point` ist der Strukturname, `spx, spy`, etc. sind Elementnamen und `stSpot` ist die deklarierte Variable

`struct Point  stPunkt1, stPunkt2;`	ebenfalls so deklarierte Variablen

Durch die Angabe einer (oder mehrerer) Strukturvariablen wird diese Struktur erzeugt (d. h. Speicherplatz dafür bereitgestellt). Strukturvereinbarungen ohne Angabe einer Strukturvariablen legen nur die Form (den Prototyp) der Struktur fest. Strukturnamen sind immer mit Großbuchstaben zu beginnen, Strukturvariabeln ist immer ein st voranzustellen.

Die geschlossene Initialisierung erfolgt (analog zu den Arrays) bei der Deklaration. Zum Beispiel:

```
struct Datum stHeute = {26,9,1987,"jun"};
struct Point  stPunkt = {2.8, -33.7, 15, 'A'};
```

Für den Elementzugriff gibt es zwei eigene Operatoren. Der direkte Zugriff wird dabei mit dem Punktoperator . , der Zugriff auf einen Strukturvariable, welche ein Adresse repräsentiert erfolgt mit dem Pfeiloperator -> Operator.

```
Strukturvariable . Komponente

Zeiger auf Strukturvariable -> Elementname
```

Beispiel:

```
stPunkt1.farbe = 11;
stPunkt2.spy = spot2.spy;
stHeute.tag = 22;
stHeute.monat = 1;
stHeute.jahr = 2000;
```

Strukturvariable können an Funktionen übergeben werden, und Funktionen können Strukturen als Rückgabetyp haben. Beispiel (mit obiger Definition):

```
/* createpoint  :  bepackt Struktur 'Point' */
struct Point createpoint(double x, double y, int farbe, char
label)
  {
  struct Point stDummy;
  stDummy.spx = x;
  stDummy.spy = y;
  stDummy.farbe = farbe;   /* gleiche Bezeichnungen */
  stDummy.label = label;   /* interferieren NICHT   */
  return stDummy;
  }
```

Strukturen können als Elemente ebenfalls wieder Strukturen enthalten (allerdings nicht sich selbst) und Strukturen können zu Vektoren zusammengefasst werden:

```
struct Kunde
  {
  char name[NAMLAE];
  char adresse[ADRLAE];
  int kund nr;
  struct Datum stLiefer datum;
  struct Datum stRech datum;
  struct Datum stBez datum;
  };
struct Kunde stKunde1, stKunde2, ... ;
struct Kunde kunden[KUNANZ];
```

Strukturpointer finden so häufig Verwendung, dass es dafür eine eigene Syntax gibt; bei gegebener Adresse auf eine Strukturvariable werden deren Elemente mit '`->`' angesprochen.

Beispiel: mit der Strukturdefinition

```
struct Point
  {
  double  px, py;
  int  farbe;
  }  stPunkt, stSprueh[100];
```

sind folgende Aufrufe äquivalent:

`stPunkt.px` entspricht `(&stPunkt)->px`

`stSprueh[20].px` entspricht `(stSprueh+20)->px`

Nun wird noch ein Zeiger auf `Punkt` definiert:

```
struct Point *pstZeiger;
```

Mit der Anweisung `pstZeiger = &stPunkt;` kann man `stZeiger` auf `stPunkt` zeigen lassen. Der Zugriff auf die Komponente `px` von `Punkt` würde in reiner Pointerschreibweise lauten (erster Versuch):

```
irgendwas = *stZeiger.px;
```

Diese Notation ist aber zweideutig. Bedeutet das jetzt "*Das worauf* `stZeiger` *zeigt, nehme Komponente px*" oder "*Das worauf* `stZeiger.px` *zeigt*"? Diese Zweideutigkeit wird man los durch (zweiter Versuch):

```
irgendwas = (*stZeiger).px;
```

Nun ist es eindeutig "*Das worauf* `stZeiger` *zeigt, nehme Komponente px*". Der Operator `->` vereinfacht das nun nur noch und wir schreiben (Endfassung):

```
irgendwas = stZeiger->px;
```

Für den Zugriff auf Variablen der Struktur schreibt man normalerweise sogenannte getter - und setter – Funktionen. Dabei setzt die setter – Funktionen eine Wert und die getter - Funktion gibt einen Wert zurück.

13.2 Varianter Record (Unions)

Während eine Struktur mehrere Variablen (verschiedenen Typs) enthält, ist eine Variante eine Variable, die (aber natürlich nicht gleichzeitig) Objekte verschiedenen Typs speichern kann. Verschiedene Arten von Datenobjekten können so in einem einzigen Speicherbereich maschinenunabhängig manipuliert werden. Syntaktisch sind `union` und `struct` analog (bis auf Initialisierung). Der wesentliche Unterschied ist, dass eine Variable vom Typ `union` zu einer Zeit immer nur *eines* deren angegebener Elemente enthalten kann. Beispiel:

```
union Utype
{   int n;
    double d;  }   uIrgendwas;
```

```
uIrgendwas.n = 3;
uIrgendwas.d = 11.7;
zahl = uIrgendwas.n;     /* in diesem Fall: Fehler! */
```

Allerdings ist Buchführung notwendig, um zu wissen, welcher Datentyp in welcher `union`-Variablen zuletzt abgespeichert wurde (deshalb der obige Fehler). Beispiel:

```
if (Utype == INT)
    printf("%d\n",uval.ival);
else if (Utype == FLOAT)
    printf("%f\n",uval.fval);
else if (Utype == STRING)
    printf("%s\n",uval.pval);
else
    printf("bad type %d in utype\n",utype);
```

13.3. Typdefinitionen (typedefs)

Mit `typedef` kann man neue Datentyp**namen** definieren (Nicht aber neue Datentypen!). `typedef` ist `#define` ähnlich, aber weitergehender, da erst vom Compiler verarbeitet und nicht nur einfacher Textersatz. Die Anwendungen von `typedef` liegen darin, ein Programm gegen Portabilitätsprobleme abzuschirmen und für eine bessere interne Dokumentation zu sorgen. Die Syntax lautet "`typedef` <bekannter Typ> <neuer Typ>", z.B.:

```
typedef int t_count;
typedef unsigned long int bigint;
```

Auch wenn der neue Typ `t_count` dem `int` entspricht, kann er Programme "fehlerbewusster" machen. Definiert man eine Funktion mit einem Parameter von Typ `t_count` wird - bei entsprechend strikter Compilereinstellung bereits bei der Übersetzung ein Fehler gemeldet, wenn ein `int`-Parameter übergeben wird.

Durch `typedef` wird aber auch der Aufbau von Structures verschleiert, so dass viele Programmierer keinen Gebrauch davon machen. Zur Syntax: Der neue Typname steht an genau der Stelle, an der *ohne* `typedef` der Strukturvariablenname stünde.

```
typedef struct Verbund
  {
  int i;
  float f;
  double df;
  } Collect;
```

`collect` ist hier also *keine* Strukturvariable, sondern der so neu definierte Typname!