ModProg 15-16, Vorl. 10

Richard Grzibovski

Jan. 6, 2016

Übersicht

1. Strukturen in C
2. Einige Anwendungen für Strukturen
3. Zwei Methoden zur Polynomauswertung
4. Vewendung eigener Typbezeicher: `typedef`
5. Unionen in C
6. Aufzählungen mit `enum`

Das Ableiten eigener Datentypen

Bisher betrachteten wir

- Elementare Datentypen: `int`, `float`, `char`, ...
- Entsprechende Felder, Zeiger: `int a[10]`, `char *s`, ...

Programmiersprache C bietet auch die möglichkeit, **eigene Datentypen zu gestalten**, d.h. *bereits bekannte Typen zu neuen zusammenfassen*. Dadurch kann man

- Datenobjekte an die Problemstellung anpassen,
- eigene Datentypen für neue Einsatzzwecke wieder verwenden.

Mit den reservierten Wörten `struct` und `union` fasst man die bekannte Typen zusammen. Um neue Datentypen zu definieren, verwendet man `typedef`.

Strukturen

Struktur

Eine Struktur ist ein Verbund, bestehend aus einer oder mehrerer Variablen, die von unterschiedlichem Typ sein können.

Die Deklaration einer Struktur wird mit dem reservierten Wort `struct` eingeleitet. Sie hat allgemein die folgende Form:

```
struct <StruktBez> {
    Datentyp1 Bezeichner1;
    ...
    DatentypN BezeichnerN;
} <Var1, ... , VarN>;
```

- `Bezeichner1`, ..., `BezeichnerN` heißen *Strukturkomponenten* oder *Strukturelemente* (engl. members).
- Der *Strukturbezeichner* (Etikett) `StruktBez` sowie die Liste `Var1`, ..., `VarN` von Variablen sind optional.
- Wird kein Strukturbezeichner angegeben, so muss die Liste der entsprechenden Variablen folgen.

Strukturendeklaration: Beispiel

Zur Identifizierung von Studierenden wäre beispielsweise folgende Datenstruktur denkbar:

```
struct student {
        char Name[50];
        char Vornamen[100];
        unsigned int MatrNr;
};
```

Hierdurch wird eine Struktur namens `struct student` deklariert, die die eingangs erwähnten Komponenten besitzt. Variablen vom Typ `struct student` wurden in dieser Deklaration keine angegeben. Dies kann man z.B. im Hauptprogramm durch die Deklaration

```
struct student stud1, stud2;
```

nachholen.

Strukturendeklaration

Ein weiteres Beispiel: Um einen Punkt in der Ebene mit kartesischen Koordinaten als Struktur zu implementieren, dient

```
struct punkt2d {
      double x;
      double y;
};
```

Später deklarieren wir die Variablen:

```
struct punkt2d pnkt1, pnkt2;
```

Die Komponenten der Struktur `struct punkt2d` sind `x`- und `y`-Koordinate des Punktes. Natürlich wäre dies auch mit einem `double`-Feld der Länge 2 zu realisieren. Die Struktur ist allerdings "intuitiver".

Bemerkung

Man beachte, dass man in solchen Deklarationen das reservierte Wort `struct` nicht weglassen darf. Der Strukturbezeichner ist nämlich kein Typbezeichner, sondern lediglich eine Abkürzung für den Block, der die Komponenten deklariert!

Initialisierung und Zuweisungen

Strukturen lassen sich wie Felder initialisieren, z.B.

```
struct punkt2d p1 = {1.0, -1.0};
```

oder

```
struct student s1 = {"Limo", "Hansi", 2900612};
```

Davon werden wir kaum Gebrauch machen, sondern folgende Methode verwenden:

Auswahloperator

Der Auswahloperator `.` dient dazu, auf die Komponenten einer Struktur(variablen) zuzugreifen. Ein solcher Zugriff hat die Form

`Strukturvariablenname.Komponentenname`

Der Zugriff ist sowohl lesend und auch schreibend.

Statt der obigen Initialisierung mit geschweiften Klammern kann man auch die Zuweisungen

```
1    p1.x = 1.0;
2    p1.y = -1.0;
```

verwenden.

Zuweisungen und Vergleich

Um den Inhalt der Strukturvariablen `p1` in `p2` zu kopieren, genügt eine einfache Zuweisungsoperation

```
struct punkt2d p2, p1={-1.0, 1.0};
p2 = p1;
```

Hierdurch werden automatisch alle Komponenten von `p2` mit den entsprechenden Werten belegt, was ein Vorteil gegenüber der Implementierung über Felder ist. Es ist auch klar, dass diese Zuweisungseigenschaft bei der Struktur `struct student` besonders angenehm ist, denn es werden uns alle nötigen `strcpy()`-Operationen abgenommen.

Es gilt:

- Den Inhalt einer Struktur kann man mit dem Zuweisungsoperator `=` als Ganzes einer anderen Struktur übergeben.
- Der Test auf Gleichheit mittels `==` hingegen ist nicht erlaubt: Hierzu muss eine entsprechende Funktion geschrieben werden, die den Vergleich (komponentenweise) vornimmt.

Strukturen und Funktionen

Strukturen dürfen sowohl als *Argumente* als auch als *Rückgabewerte* von Funktionen auftreten.

Beispiel:

```
1  #include<math.h>
2  #include<stdio.h>
3  struct punkt2d {
4      double x,y;
5  };
6  double abstand2d(struct punkt2d p1, struct punkt2d p2) {
7      return sqrt((p1.x - p2.x)*(p1.x - p2.x)
8                  + (p1.y - p2.y)*(p1.y - p2.y));
9  }
10 int main() {
11     struct punkt2d pa={0,0}, pb={-3,8};
12     fprintf(stdout,"der Abstand pa<-->pb ist %lf\n",
13                         abstand2d(pa,pb));
14     return 0;
15 }
```

```c
#include<stdio.h>
#include<string.h>
struct student {
    char *name, *vname;
    unsigned int matr;
};
struct student neuer_stud(
  const char* name,const char* vname, unsigned int nr) {
    struct student st;
    st.name  = malloc((strlen( name)+1)*sizeof(char));
    st.vname = malloc((strlen(vname)+1)*sizeof(char));
    assert(st.name!=NULL); assert(st.vname!=NULL);
    strcpy(st.name , name);
    strcpy(st.vname,vname);
    st.matr = nr;
    return st;
}
int main() {
    struct student stud;
    stud=neuer_stud("Musterman","Max Jens",1234);
    return 0;
}
```

Schachtelung von Strukturen

Strukturen dürfen als Komponenten von Strukturen auftreten, aber:

> Eine Struktur darf sich **nicht** selbst als Komponente besitzen.

```
1  struct laufrad {
2      char marke[100];
3      float durchmesser, alter;
4      int anz_speichen;
5  };
6  struct fahrrad {
7      char marke[100], farbe[50];
8      struct laufrad radv, radh;
9  };
10 ...
11 struct fahrrad fr;
12 ...
13 if(fr.radv.anz_speichen==fr.radh.anz_speichen) ...
```

Felder, Zeiger und Call by Reference

Die Deklaration statischer Felder von Strukturen erfolgt wie immer.
Z.B.

```
struct punkt2d pfeld[16];
```

Zugriff ist durch

```
pfeld[10].x;
```

Zeiger auf Strukturen werden wie üblich deklariert und initialisiert:

```
struct punkt2d p, *pzgr;
pzgr = &p;
```

Ein dynamisches Feld erzeugt man durch

```
pzgr = malloc(N*sizeof(struct punkt2d));
```

Dereferenzierung und der Auswahloperator ->

Dereferenzierung mit dem Inhaltsoperator *:

```
struct punkt2d p, *pz;
pz=&p;
 ...
printf("x-pos: %le\n", (*pz).x );
```

Ist äquivalent zur Dereferenzierung mit dem Auswahloperator ->:

```
printf("x-pos: %le\n", pz->x );
```

Ist `structzgr` ein Zeiger auf eine Struktur, so wird auf den Inhalt der Komponente `komp` durch
`structzgr->komp`
zugegriffen (lesend und schreibend).
Dereferenzierung des Zeigers und Komponentenauswahl werden also in einem Schritt erledigt.

Zeiger auf Strukturen als Funktionsargumente

Struktur $\sim$ "große Variable".
Wenn wir diese als Funktionsargumente übergeben, wird der Struktureninhalt kopiert.
Nachteile:

- Kopierungsoperation kostet Zeit und Speicherplatz.
- Eventuelle Änderungen von Daten verschwinden nach dem Verlass der Funktion.

Es ist günstiger, Funktionen so zu implementieren, dass sie die Zeiger auf Strukturen als Funktionsargumente annimmt. Somit:

- nur die Sktrukturadresse wird kopiert;
- man bekommt den Möglichkeit die Struktureninhalt zu ändern.
  Diese Möglichkeit kann entfernt werden durch den Einsatz des Modifizierers `const`.

Zeiger auf Strukturen als Funktionsargumente

```
1  struct student {
2          char Name[50], Vornamen[100];
3          unsigned int MatrNr;
4  };
5  void student_drucken(FILE* fl, struct student st) {
6      fprintf(fl,"Name    = %s\n",st.Name);
7      fprintf(fl,"Vorname= %s\n",st.Vornamen);
8      fprintf(fl,"MatrNr = %i\n",st.MatrNr);
9  }
```

Eine bessere Alternative wäre:

```
1  void student_drucken(FILE* fl,
2                       const struct student *st) {
3      fprintf(fl,"Name    = %s\n",st->Name);
4      fprintf(fl,"Vorname= %s\n",st->Vornamen);
5      fprintf(fl,"MatrNr = %i\n",st->MatrNr);
6  }
```

Einige Anwendungen für Strukturen

1. Die adressierte 1D-Implementierung von Matrizen
2. Polynome und komplexe Zahlen
3. Messung von Zeitspannen
4. Verkettete Listen

Die adressierte 1D-Implementierung von Matrizen

```
1  struct matrix {
2      int zl, sp;
3      double **eintrag, *werte;
4  };
5  void neu_mat(struct matrix *A,const int m,const int n){
6      int i;
7      A->zl = m;
8      A->sp = n;
9      A->werte = malloc(n*m*sizeof(double));
10     assert(A->werte!=NULL);
11     A->eintrag = malloc(m*sizeof(double*));
12     assert(A->eintrag!=NULL);
13     for(i=0;i<m;i++) A->eintrag[i] = A->werte + i*n;
14 }
15 void freigabe_mat(struct matrix *A) {
16     free(A->eintrag);
17     free(A->werte);
18     A->sp = A->zl = 0;
19 }
```

Die adressierte 1D-Implementierung von Matrizen

```
1  struct matrix {
2      int zl, sp;
3      double **eintrag, *werte;
4  };
5  ... ... ...
6  void zeige_mat(const struct matrix *A) {
7      int i, j;
8      for(i=0;i<A->zl;i++) {
9          for(j=0;j<A->sp;j++)
10             printf("%e ", A->eintrag[i][j]);
11         printf("\n");
12     }
13 }
```

Vorteile:

- die Struktur beinhaltet *alle* nötigen Informationen über die Matrix,
- kürzere Parameterlisten für Funktionen.

Polynome

Um ein Polynom vom Grad n mit reellen Koeffizienten

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a0 = \sum_{k=0}^{n} a_k x^k, \; a_n \neq 0,$$

zu implementieren, bietet sich eine dynamisch angelegte Struktur der Form

```
1  struct polynom {
2      unsigned grad;
3      double *koeff;
4  };
```

an. Um das Polynom $2.2x^2 + 3x - 1$ darzustellen, schreibt man

```
1  struct polynom p;
2  p.grad=2;
3  p.koeff=malloc((p.grad+1)*sizeof(double));
4  p.koeff[0]= -1.0;
5  p.koeff[1]=  3.0;
6  p.koeff[2]=  2.2;
```

Zwei Methoden zur Polynomauswertung

Um ein Polynom vom Grad $n \in \mathbb{N}$

$$P(x) = \sum_{k=0}^{n} a_k x^k,$$

an einer Stelle $x_0 \in \mathbb{R}$ auszuwerten, kann man zunächst folgende "naive" Methode verwenden:

1. Setze $p_0 := a_0$.
2. Für $k = 1, 2, \ldots, n$:
   - setze Hilfsgröße $M := a_k$,
   - Für $m = 1, \ldots, k$ berechne $M := M \cdot x_0$.
   - $p_0 := p_0 + M$.
3. Liefere p_0 zurück.

Zwei Methoden zur Polynomauswertung

In jedem der n Schritte werden jeweils eine Addition und k Multiplikationen durchgeführt. Da wir den Zeitaufwand für die Zuweisungen gegenüber den arithmetischen Operationen vernachlüssigen künnen, ist die Gesamtanzahl der Operationen

$$Op(n) = \sum_{k=1}^{n}(k+1) = n + \sum_{k=1}^{n} k = n + \frac{1}{2}n(n+1) = \frac{1}{2}n^2 + \frac{3}{2}n.$$

Man erkennt sofort, wo die Schwäche dieser Methode liegt: Für jeden Summanden im Polynom wird x^k komplett neu berechnet!

Zwei Methoden zur Polynomauswertung

Eine wesentlich effzientere Vorgehensweise erhält man, indem man das Polynom in Form ineinander geschachtelter Linearfaktoren schreibt:

$$P(x) = (\ldots(((a_n x + a_{n-1})\,x + a_{n-2})\,x + a_{n-3})\ldots)\,x + a_0$$

Man erhält so die folgende Methode, die auch Horner-Schema genannt wird:

1. Setze Hilfsgröße $p_0 := a_n$.
2. Für $k = n-1, \ldots, 0$ berechne

$$p_0 := p_0 \cdot x + a_k.$$

3. Liefere p_0 zurück.

Diese Vorgehensweise ist nicht nur kürzer in der Formulierung: In jedem der n Schritte werden lediglich eine Addition und nur eine Multiplikation vorgenommen, so dass für diese Methode $Op(n) = 2n$ folgt.

Komplexe Zahlen

Zur Implementierung von komplexen Zahlen $z \in \mathbb{C}$, d.h.

$$z = a + \mathrm{i}b,\ \ a, b \in \mathbb{R},\ \ \mathrm{i}^2 = -1,$$

mit Realteil a und Imaginärteil b, ist die folgende Struktur nahe liegend, wenn man die arithmetischen Operationen mit doppelter Gleitpunktgenauigkeit durchführen lassen will:

```
1  struct komplex {
2      double re, im;
3  };
```

Somit, ist die Multiplizierungsoperation

```
1  struct komplex kompl_mult(struct komplex a,
2                            struct komplex b) {
3      sruct komplex c;
4      c.re = a.re*b.re - a.im*b.im;
5      c.im = a.re*b.im + a.im*b.re;
6      return c;
7  }
```

Messung von Zeitspannen

In der Headerdatei <sys/time.h> findet sich die folgende Struktur:

```
1  struct timeval {
2    time_t      tv_sec;  /*      seconds */
3    suseconds_t tv_usec; /* microseconds */
4  };
```

Dabei sind `time_t` und `suseconds_t` ganzzahlige Datentypen, die für die Zeitmessung vorgesehen sind.
Mit Hilfe der folgenden Funktion kann man die Hardwareuhr des Computers mikrosekundengenau auslesen:

```
int gettimeofday(struct timeval *tv, struct timezone *tz);
```

- Die ausgelesene Zeit wird in der Zeitstrukturvariablen abgelegt, auf die `tv` zeigt.
- Die Struktur `struct timezone` ist für uns nicht weiter relevant: Bei Aufruf der Funktion werden wir hierfür als Argument `NULL` ubergeben.
- Die Funktion liefert 0 zurück, wenn keine Fehler aufgetreten sind und -1 sonst.

Messung von Zeitspannen

Die Funktion kann u.a. dazu verwendet werden, die Zeitspanne zu messen, die die Ausführung eines bestimmten Programmteils in Anspruch nimmt:

```
1   #include<sys/time.h>
2   ...
3   struct timeval start, stop;
4   float dauer;
5   ....
6   gettimeofday(&start,NULL);
7   ....
8   gettimeofday(&stop,NULL);
9   dauer = stop.tv_sec-start.tv_sec
10        + 1.0e-6f*(stop.tv_usec-start.tv_usec);
```

Verkettete Listen

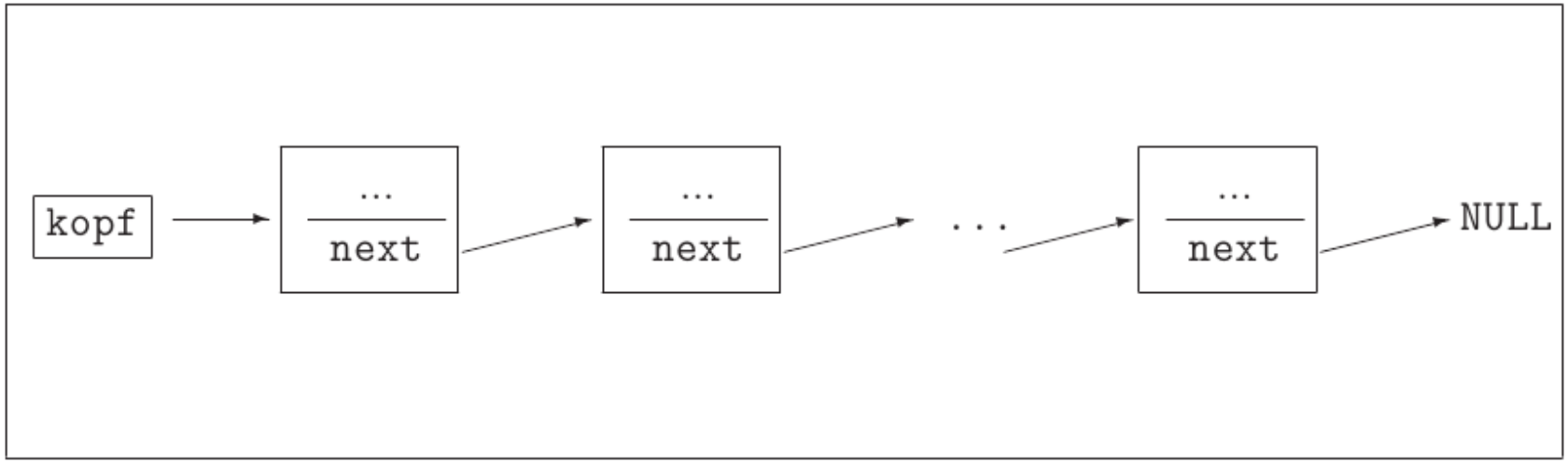
Strukturen dürfen sich zwar nicht selbst als Komponente besitzen, aber

> ein Zeiger auf die Struktur ist als Komponente erlaubt.

Eine Anwendung ist die *(einfach) verkettete Liste*, die zum Einsatz kommt, wenn man von einer Feldindizierung nicht profitiert bzw. diese sogar hinderlich ist.
Für ein Listenelement wird allgemein eine Struktur der folgenden Gestalt verwendet

```
1  struct listelem {
2      ....
3      struct listelem *next;
4  };
```

Verkettete Listen



Allgemein sind folgende Aspekte charakteristisch für die Listenimplementierung:

- Der Zeiger `next` zeigt auf das jeweils nächste Listenelement.
- Ausgehend von einem so genannten Listenkopf vom Typ `struct listelem *`, der auf den Listenanfang (das erste Element also) zeigt, werden die Listeneinträge uber ihren Zeiger auf das nächste Element miteinander verkettet.
- Das Listenende (d.h. das letzte Element) ist dadurch gekennzeichnet, dass `next` den Wert `NULL` besitzt.
- Eine leere Liste liegt vor, wenn `kopf` den Wert `NULL` hat.

Verkettete Listen

```
1  struct zielort {
2      char name[50]
3      struct punkt2d koord;
4      struct zielort *next;
5  };
6  ...
7    struct zielort *kopf, *ort;
8    int i=0;
9  ...
10   for(ort=kopf;ort!=NULL;ort=ort->next) {
11       i++;
12       printf("%i -- name= %s\n",i, ort->name);
13   }
14 ...
```

Listen vs Felder

Felder	Listen
bieten einfachen Zugriff zum Elementen durch die Indexierung; ligen als ein Block in Speicher;	bieten einfache Entfernungen/Einfügungen von Elementen;

Vewendung eigener Typbezeicher: `typedef`

> Mit dem reservierten Wort `typedef` können neue Typbezeichner für elementare und abgeleitete Datentypen eingeführt werden.

Eine solche Typbenennung hat einen ähnlichen Aufbau wie die Deklaration von Variablen:

```
typedef Typname NeuerTypname <, *NeuerZeigerTypname>;
```

Optional kann man mit einer `typedef`-Anweisung auch gleichzeitig einen neuen Typnamen für Zeiger auf den betreffenden Datentypen einführen. Faustregel:

> In einer `typedef`-Vereinbarung steht der neue Typbezeichner an der Stelle, an der bei einer Variablendeklaration der Variablenname steht, kurz:
>
> `Typdeklaration` = `typedef` + `Variablendeklaration`.

Vewendung eigener Typbezeicher: typedef

Für das eingangs erwähnte Beispiel punkt2d bedeutet dies, dass der neue Typbezeichner Punkt2d durch

```
typedef struct punkt2d Punkt2d;
```

eingeführt wird. Die Variablendeklaration wird hierdurch stark verkürzt:

```
Punkt2d p1, p2;
```

Wir betrachten einige weitere Beispiele:
String als Datentyp "Zeiger auf char" und Strzgr als "Zeiger auf String":

```
typedef char * String, *Strzgr;
```

Danach können Stringvariablen bzw. -felder einfach mit

```
String str1, str2;
Strzgr strfeld1, strfeld2;
```

dekariert werden. Diese sind also nach der typedef-Vereinbarung synonym zu

```
char *str1, *str2;
char **strfeld1, **strfeld2;
```

Vewendung eigener Typbezeicher: typedef

Matrix1DA als Datentypbezeichner für eine Matrix, die mit der adressierten 1D-Implementierung realisiert wird:

```
typedef struct matrix Matrix1DA;
```

Man kann die Strukturdeklaration auch mit der Typbenennung verbinden, wobei das Strukturetikett auch weggelassen werden kann, wenn es nicht benötigt wird:

```
1  typedef struct {
2    int zl, sp;
3    double **eintrag, *werte;
4  } Matrix1DA;
5
6  void neu_mat(Matrix1DA *A, const int m, const int n);
7  ...
8  Matrix1DA mat;
9  ...
```

Beispiel: verkettete Listen

```
1  typedef struct zielort {
2      char name[50]
3      struct punkt2d koord;
4      struct zielort *next;
5  } Zielort, *ZielZgr;
6
7  ZielZgr NeuerOrt(char* name, double x, double y) {
8     ZielZgr neu;
9     neu = malloc(sizeof(Zielort));
10    assert(neu!=NULL);
11    neu->koord.x = x; neu->koord.y = y;
12    strncpy(neu->name,name,50);
13    return neu;
14 }
15
16 ZielZgr OrtVoranst(ZielZgr kopf, ZielZgr ort) {
17    assert(ort!=NULL);
18    ort->next = kopf;
19    return ort;
20 }
```

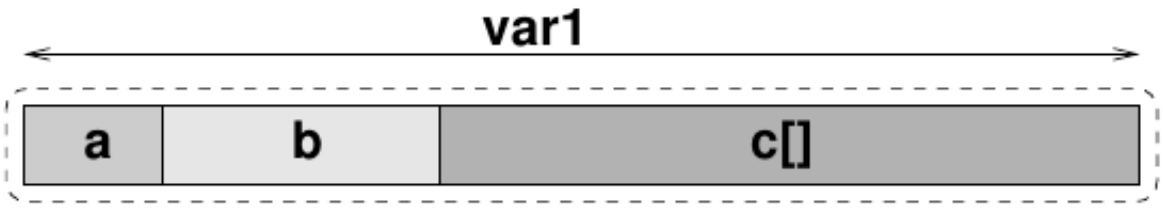
Unionen

Während Strukturen (meist) mehrere Komponenten (ggf.) verschiedener Typen zur *gleichen Zeit besitzen*, ist die bei Unionen zu *verschiedenen Zeiten* der Fall, d.h. es ist jeweils **nur eine Komponente gültig** und der Wert einer Union ist durch ihre aktuelle Komponente bestimmt. Die anderen Komponenten werden von der aktuellen uberlagert. Eine Union ist deshalb so groß, dass sie auf jeden Fall ihre größte Komponente speichern kann.
Die Deklaration erfolgt analog zu Strukturen:

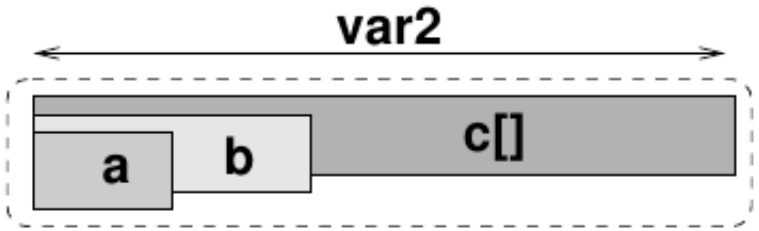
```
union <Unionbezeichner> {
      Deklaration Komponente 1
      ...
      Deklaration Komponente N
} <Liste von Variablen>;
```

Unionen

```
struct St {
    int a ;
    long int b ;
    char c[12] ;
} var1 ;
```




```
union Un {
    int a ;
    long int b ;
    char c[20] ;
} var2 ;
```



Die anderen Komponenten werden von der aktuellen uberlagert. Eine Union ist deshalb so groß, dass sie auf jeden Fall ihre größte Komponente speichern kann.

- Vereinbarung und Komponentenzugriff analog zu `struct`
- alle Komponenten erhalten dieselbe Anfangsadresse
- Komponenten nur alternativ nutzbar

Unionen: Beispiel

```
1  typedef struct {
2        char author[100], titel[100];
3        int isbn[15];
4  } Buch;
5
6  typedef struct {
7        char name[100];
8        int nr, jahr;
9  } Journal;
10
11 typedef union {
12       Buch buch;
13       Journal journal;
14 } BibEinheit;
15
16 typedef struct {
17       int id, typ;
18       BibEinheit einheit;
19 } BibArtikel;
```

Aufzählungen mit enum

In C existiert das reservierte Wort enum, um Aufzählungen zu vereinbaren:

```
enum <tname> { Name1<=Wert1>, ... ,NameN<=WertN> } <vname>;
```

Die einzelnen Namen in einer solchen Aufzählung heißen Aufzählungskonstanten, denen ganzzahlige Werte zugeordnet werden können. Wir betrachten z.B.

```
enum {maennlich=1, weiblich=2};
```

Hierdurch wird erreicht, dass maennlich dem ganzzahligen Wert 1 und weiblich dem ganzzahligen Wert 2 entspricht.
Die Zuweisungen können auch entfallen, dann entspricht der ersten Aufzählungkonstante der Wert 0, der zweiten der Wert 1 usw.

```
enum {maennlich, weiblich};
```

Jetzt steht maennlich für den Wert 0 und weiblich für 1.

Aufzählungen mit enum

```c
#include <stdio.h>

enum Wochentag {Montag, Dienstag, Mittwoch,
                Donnestag, Freitag, Samstag, Sonntag};
int main() {
    enum Wochentag tag;
    int j = 0;
    printf("Geben Sie ein Tag ein (0 bis 6)\n");
    scanf("%d",&j);
    tag = j;
    if(tag == Sonntag || tag == Samstag)
        printf("Wochenende ist da!\n");
    else if(tag == Mittwoch)
        printf("ModProg Vorlesung um 16:15!\n");
    else
        printf("Der Tag hat kein Reiz.\n");
    return 0;
}
```

Verwendung von union oft innerhalb eines struct zusammen mit einem "Merker", der die momentan benutzte union-Alternative angibt:

```
1     struct st {
2       enum { Variante1, Variante2 } nutzung ;
3       union {
4             Komponente1;
5             Komponente2;
6       } u;
7     };
```

struct, enum und union

```
 1  typedef union {
 2        Buch buch;
 3        Journal journal;
 4  } BibEinheit;
 5  typedef struct {
 6        int id;
 7        enum {JOURNAL, BUCH} typ;
 8        BibEinheit einheit;
 9  } BibArtikel;
10  BibArtikel art[50];
11  ...
12  art[11].id=5;
13  art[11].typ = BUCH;
14  strcpy(  art[11].einheit.buch.author, "Adams, Douglas");
15  ...
```

struct, enum und union

```
1  struct JaegerUndSammler {
2      enum { JAEGER, SAMMLER } typ;
3      char name[20];
4      union {
5          struct Jg {
6              unsigned speer_nr;
7              double   reichweite;
8              float    geschwindigkeit;
9          } jaeger ;
10         struct Sm {
11             float    maxLast;
12             unsigned erfahrung;
13         } sammler;
14     } daten;
15 } mitarbeiter[200];
16 mitarbeiter[5].typ = SAMMLER;
17 strcpy( mitarbeiter[ 5 ].name, "Karl" );
18 mitarbeiter[5].daten.sammler.maxLast = 50.3;
19 mitarbeiter[5].daten.sammler.erfahrung = 23;
```