

# Grundlagen der Informatik

## Vorlesungsskript

Prof. Dr. T. Gervens, Prof. Dr.-Ing. B. Lang, Prof. Dr.-Ing. C. Westerkamp

<b>10 DYNAMISCHE SPEICHERPLATZBEREITSTELLUNG .....</b>	<b>2</b>
10.1 DYNAMISCHES ANFORDERN VON SPEICHERPLATZ .....	2
10.2 BEISPIELE: EIN- UND ZWEIDIMENSIONALE, DYNAMISCHE FELDER.....	3
10.3 VERÄNDERUNG DER GRÖÖE EINES BEREITGESTELLTEN SPEICHERBEREICHS .....	6
10.4 FREIGABE EINES BEREITGESTELLTE SPEICHERBEREICHS.....	7
<b>11 STRUKTUREN.....</b>	<b>8</b>
11.1 EINFACHE STRUKTUREN.....	8
11.2 GESCHACHELTE STRUKTUREN .....	10
11.3 OPERATIONEN MIT STRUKTUREN .....	12
11.4 ZEIGER AUF STRUKTUREN.....	13
11.5 UNION-DATENSTRUKTUR.....	15
11.6 BITFELDER.....	17
<b>12 AUFZÄHLUNGSTYPEN .....</b>	<b>19</b>
<b>13 TYPDEFINITIONEN.....</b>	<b>20</b>
<b>14 DIREKTE SPRUNGBEFEHLE.....</b>	<b>21</b>

## 10 Dynamische Speicherplatzbereitstellung

### 10.1 Dynamisches Anfordern von Speicherplatz

Statische Felder wie

```
double x[1000];
```

besitzen den Nachteil, dass zur Kompilierzeit die Größe des Feldes festliegen muss. Reicht der die Größe des Feldes für eine Applikation nicht aus, muss die Größe erhöht und das Programm neu kompiliert werden.

Man kann die Anpassung an veränderte Dimensionen durch Verwendung von Präprozessordefinitionen zentral vornehmen:

```
#define DIMFIELD 1000
...
double x[FIELD];
```

Dann kann man überall, wo die Feldgröße benötigt wird, den symbolischen Wert **DIMFIELD** eintragen. Dennoch muss bei einer Änderung von **DIMFIELD** das Programm erneut kompiliert werden.

Wählt man, um spätere Änderungen zu vermeiden, von Anfang an sehr große Felder in einem Programm, führt dies zur Verschwendung von Speicherplatz und trotzdem ist man nie sicher, ob die verwendeten Größen für jedes Feld immer ausreichen.

Abhilfe aus dieser Situation bietet das dynamische Einrichten von Speicherplatz. Es erfolgt zur Programmlaufzeit in zwei Schritten:

1. Vom Betriebssystem wird für das Feld ein zusammenhängender Speicherbereich bestimmter Größe angefordert.
2. Es muss dafür gesorgt werden, dass der eingerichtete Speicherplatz als Feld des gegebenen Typs angesprochen werden kann.

Der erste Schritt erfolgt in C mit der Bibliotheksfunktion **malloc**. Das Interface dieser Funktion ist in „*malloc.h*“ deklariert:

```
void *malloc (unsigned int length);
```

Über die Funktion **malloc** stellt das Betriebssystem Speicherplatz der Größe **length** Bytes zur Verfügung. Die Funktion **malloc** gibt einen unspezifischen **void**-Zeiger auf den Anfang dieses freien Speicherbereichs der Größe **length** Bytes zurück, der nun vom Programm verwendet werden darf.

Die Größe **length** muss aus der gewünschten Feldgröße und der Größe eines einzelnen Feldelements ab. Sie wird im Programm wie folgt ermittelt:

```
length = <Feldgröße> * sizeof(<Typ>)
```

Kann das Betriebssystem den gewünschten Speicherbereich nicht zur Verfügung stellen, dann liefert **malloc** einen **NULL**-Zeiger, also den Wert 0, zurück.

Um den in Schritt 2 bereitgestellten Speicherplatz als Feld eines bestimmten Typs ansprechen zu können, weist man die Anfangsadresse des Bereichs nach einer Typumwandlung mit Hilfe des *cast*-Operators (`<Typ>`) einer Zeigervariablen vom Typ „`<Typ>*`“ (Zeiger auf „`<Typ>`“) zu.

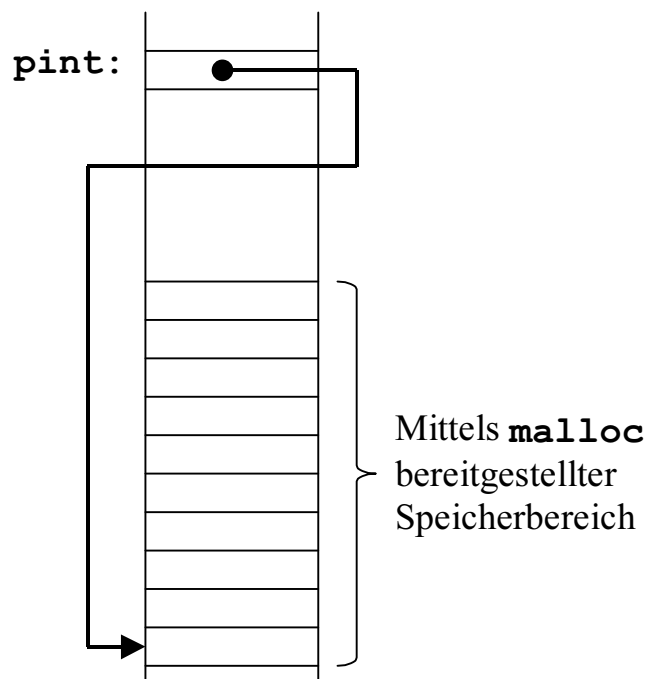
## 10.2 Beispiele: Ein- und zweidimensionale, dynamische Felder

Beispiel: Dynamisches Anfordern eines Speicherbereichs für ein eindimensionales Feld mit  $n$  Komponenten vom Typ `int`:

```
#include <malloc.h>

int main() {
    int n;
    int *pint;
    ...
    n=1000;
    pint = (int *)malloc(n*sizeof(int));
    ...
}
```

Wird von der `malloc`-Funktion für `pint` ein gültiger Zeigerwert (kein `NULL`-Zeiger) zurückgeliefert, so liegt folgende Situation vor:



Auf die Komponenten des dynamisch angelegten Feldes kann man genauso zugreifen, wie man es bei statischen Feldern gewohnt ist:

Inhalt der 1-ten Komponente: `*pint` oder `pint[0]`

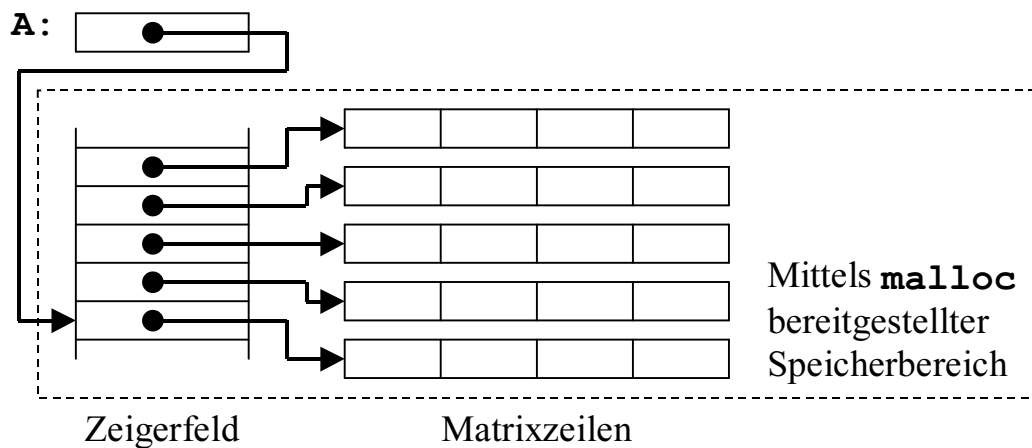
Inhalt der 2-ten Komponente: `*(pint+1)` oder `pint[1]`

usw.

Beispiel: Dynamisches Anfordern eines Speicherbereichs für ein **zweidimensionales** Feld mit  $n$  Zeilen und  $m$  Spalten vom Typ **double**:

Die dynamische Bereitstellung zweidimensionaler Felder ist komplizierter als der eindimensionale Fall, da  $n$  Zeilen der Größe  $m$  ( $m$  Spalten) dynamisch angelegt werden müssen.

Ein zweidimensionales Feld mit dynamischer Speicherbereitstellung kann mit folgender Struktur aufgebaut werden:



Die zweidimensionale Matrix wird durch einen Zeiger dargestellt, der zunächst auf ein Feld von Zeigern verweist. Jeder Zeiger dieses Zeigerfelds zeigt dann auf eine Matrixzeile. Somit kann über das Zeigerfeld auf die Matrixzeilen zugegriffen werden.

Die Zeigervariable **A** muss im Programm als ein „Zeiger auf einen Zeiger“ vereinbart werden. Dies wird in C durch zwei Sterne **\*\*** gekennzeichnet. Sowohl das Zeigerfeld als auch die Matrixzeilen werden dynamisch mit **malloc** angefordert:

```
int i;
double **A;
...
A=(double **)malloc(n*sizeof(double*));
for(i=0; i<n; i++){
    A[i]=(double *) malloc(m*sizeof(double));
}
```

Da die Speicherbereitstellung durch das Betriebssystem vergleichsweise viel Zeit erfordert, erzielt man eine schnellere Laufzeit, wenn man den Speicherplatz für das Zeigerfeld und die Matrixzeilen in einem Block anfordert:

```
A = (double **) malloc (n*sizeof(double*)
                        + n*m*sizeof(double));
A[0]=(double *) (A+n);
for(i=1; i<n; i++) {
    A[i]=A[i-1]+m;
}
```

Auf den Wert der Matrixelemente kann wie im statischen Fall über einen zweidimensionalen Feldzugriff **A[zeile][spalte]** oder (bevorzugt) **\*(\*(A+zeile)+spalte)** zugegriffen werden.

Die Zugriffe auf die einzelnen Datenelemente der dynamischen Feldstruktur sind in nachfolgender Tabelle erläutert:

<b>A</b>	Zeigervariable, die auf den Anfang des Zeigerfeldes zeigt.
<b>A+i</b>	Zeiger auf den Zeiger i im Zeigerfeld.
<b>*(A+i)</b> oder <b>A[i]</b>	Zeiger auf den Beginn der Matrixzeile i.
<b>*(A+i)+j</b>	Zeiger auf Element j der Matrixzeile i.
<b>*(*(A+i)+j)</b> oder <b>A[i][j]</b>	Wert von Element j der Matrixzeile i

Beispiel: Programm zum Einrichten eines dynamischen, zweidimensionalen Feldes, dessen Zeilen- und Spaltenanzahl mit **scanf** eingelesen werden.

```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
int main() {
    int ze, sp, n, m; /* n Zeilen, m Spalten */
    int **A;
    scanf("%d %d", &n,&m);
    A = (int **)malloc(n*sizeof(int *));
    if (A==NULL) {
        return 1; /* Fehlerabbruch */
    }
    for(ze=0; ze<n; ++ze) {
        *(A+ze)=(int*)malloc(m*sizeof(int));
        if(*(A+ze)==NULL) {
            return 2;
        }
    }
}
```

```

    /* Belegen der Komponenten */
    for(ze=0; ze<n; ze++) {
        for(sp=0; sp<m; sp++) {
            *(*(A+ze)+sp)=ze*sp;
        }
    }
    /* Ausdruck der Matrix */
    for(ze=0; ze<n; ze++) {
        for(sp=0; sp<m; sp++) {
            printf("A[%d][%d]=%d", ze, sp, *(*(A+ze)+sp));
        }
        printf("\n");
    }
    getchar();
    return 0;
}

```

### 10.3 Veränderung der Größe eines bereitgestellten Speicherbereichs

Mit Hilfe der Bibliotheksfunktion **realloc** lässt sich die Größe eines mit **malloc** bereitgestellten Speicherbereichs nachträglich verändern.

```
void *realloc(void *ptr, unsigned int length);
```

Die Funktion **realloc** benötigt zwei Argumente, den Zeiger **ptr** auf einen durch **malloc** bereitgestellten Speicherbereich und die neue Größe **length** des Bereichs in Bytes.

Beispiel:

```

int n;
double *a;
n=50;
a=(double *) malloc(n*sizeof(double));
n=70;
a=(double *) realloc(a, n*sizeof(double));
n=20;
a=(double *) realloc(a, n*sizeof(double));

```

Die Funktion **realloc** sorgt dafür, dass der Inhalt des bisherigen Speicherbereichs erhalten bleibt. Wenn **realloc** direkt hinter dem bisherigen Speicherbereich noch freien Speicherplatz findet, verwendet es diesen und der Zeiger auf den Speicherbereich bleibt erhalten. Ist hinter dem bisherigen Speicherbereich nicht mehr genügend Speicher vorhanden, reserviert **realloc** einen völlig neuen Speicherbereich der neuen Größe und kopiert den Inhalt des bisherigen Bereichs an den Anfang des neuen Bereichs. In diesem Fall ändert sich der Zeiger auf den Speicherbereich und damit auch die Adressen aller Datenobjekte in diesem Bereich.

#### **10.4 Freigabe eines bereitgestellte Speicherbereichs**

Wird ein zuvor mit **malloc** dynamisch angeforderter Speicherbereich nicht mehr benötigt, kann dieser mit Hilfe der Funktion **free** wieder an das Betriebssystem zurückgegeben werden, d.h. frei gegeben werden:

```
void free(void *z);
```

Das Argument von **free** muss ein zuvor von **malloc** oder **realloc** gelieferter Adresswert sein.

# 11 Strukturen

## 11.1 Einfache Strukturen

Mit Hilfe von Strukturen können in C Datenobjekte unterschiedlichen Typs zu einer logischen Einheit (Verbund, Record) zusammengefasst werden. Die Datenobjekte bilden dann die Komponenten der Struktur.

Beispiel: Personaldaten

Mit der nachfolgenden Definition werden zwei Strukturvariable mit Namen **person1** und **person2** angelegt, welche mehrere Datenobjekte zur Beschreibung einer Person enthalten:

```
struct {
    char          name[30] ;
    char          vorname[20] ;
    short        alter;
    char          abteilung[50] ;
    unsigned int  eintrittsjahr;
    long int      telefon;
} person1, person2;
```

**Allgemeine Syntax:**

Werden wenige Variablen einer Struktur benötigt, fasst man die Beschreibung der Struktur und die Definition der zugehörigen Variablen in einer Vereinbarung zusammen:

```
struct {
    <Variablendeklaration1>;
    ...
    <VariablendeklarationN>;
} <Strukturvariable1>, ..., <StrukturvariableM>;
```

Werden mehrere Datensätze derselben Struktur benötigt, so deklariert man zunächst die Struktur als einen neuen Datentyp mit der Vergabe eines Strukturtypnamens:

```
struct <Strukturtypname> {
    <Variablendeklaration1>;
    ...
    <VariablendeklarationN>;
};
```

Anschließend definiert man die gewünschten Strukturvariablen unter Verwendung des neuen Datentyps **struct** <Strukturtypname>:

```
struct <Strukturtypname> <Strukturvariable>;
```

Eine Struktur fasst somit mehrere Datenobjekte in einer gemeinsamen Datenstruktur zusammen. Diese Datenobjekte werden als **Komponenten** der Struktur bezeichnet.

Beispiel:

```
struct angestellter{
    char        name[30];
    char        vorname[20];
    short       alter;
    char        abteilung[50];
    unsigned int eintrittsjahr;
    long int    telefon;
};
```

```
struct angestellter person1, person2;
```

Die Strukturvariablen **person1** und **person2** besitzen dann die in **struct angestellter{...}** vereinbarte Struktur.

### Initialisierung von Strukturen

Die Initialisierung von Strukturen erfolgt ähnlich wie bei Feldern mit einer nachgestellten Initialisierungsliste:

```
struct {
    char        name[30];
    char        vorname[20];
    short       alter;
    char        abteilung[50];
    unsigned int eintrittsjahr;
    long int    telefon;
} person1, person2 = { "Mueller", "Heinz", 24,
                      "Entwicklung", 2001, 123456 };
```

### Zugriff auf die Komponenten von Strukturen

Auf die Komponenten der Struktur kann über den Punkt-Operator, der den Namen der Strukturvariablen mit dem Namen der Komponentenvariablen verbindet, zugegriffen werden:

*<Strukturvariable>.<Komponentenvariable>*

Dieser Ausdruck kann wie eine normale Variable verwendet werden.

Beispiele:

```
strcpy(person1.name, "Meinert");
person1.telefon=654321;
printf("Eintrittsjahr von %s ein: ", person1.name);
scanf("%d", &person1.eintrittsjahr);
printf("%s in Abteilung %s ist seit %d im
      Unternehmen.\n",
      person1.abteilung,
      person1.name,
      person1.eintrittsjahr);
}
```

### **Anordnung von Strukturen im Speicher**

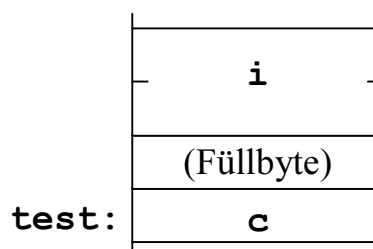
Die Komponenten einer Strukturvariablen werden unter Beachtung eines prozessorabhängigen Alignments vom Compiler zusammenhängend im Speicher abgelegt.

Unter Alignment versteht man, dass Datenobjekte abhängig vom Typ nur an bestimmten Adressvielfachen beginnen dürfen. Für *integer*-Datentypen besteht beispielweise meist ein Alignment der Prozessorwortbreite. Bei einer Wortbreite von 16 Bit müssen dann *integer*-Variable immer auf geraden Byte-Adressen beginnen. Folgt in einer Struktur z.B. auf eine *char*-Komponente eine *integer*-Komponente, dann werden hinter der *char*-Komponenten soviele Füllbytes eingefügt, bis das Alignment für die *integer*-Komponente gegeben ist.

Beispiel:

Anordnung einer Struktur **test** im Speicher bei 16-Bit Alignment.

```
struct {
    char c;
    int i;
} test;
```



Für die Anfangsadresse einer Struktur gilt meist ebenfalls ein Alignment, welches der Dokumentation des Compilers und des Prozessors zu entnehmen ist.

### **11.2 Geschachtelte Strukturen**

Strukturen können als Komponenten jeden Datentypen, insbesondere auch andere Strukturen enthalten. Dabei können Strukturen beliebig tief geschachtelt werden.

Beispiel:

```
struct betrieb {
    char name[30];
    char ort[30];
};
struct produkt {
    char name[30];
    unsigned int preis;
    struct betrieb B;
};
```

Beispiel:

```
struct punkt{
    double x;
    double y;
};
struct rechteck{
    struct punkt links_oben;
    struct punkt rechts_unten;
};
struct kreis {
    double radius;
    struct punkt mittelpunkt;
};
```

Die Initialisierung geschachtelter Strukturen erfolgt mit geschachtelten Initialisierungslisten.

Beispiel (mit oben vereinbartem Strukturtyp **struct kreis**):

```
struct kreis K1={4.0, {1.0,1.0} };
```

Der Zugriff auf die Komponenten geschachtelter Strukturen erfolgt durch mehrfache Anwendung des Punkt-Operators. Dabei müssen die Komponentennamen innerhalb einer Teil-Struktur eindeutig sein.

Beispiel (mit oben definierter Strukturvariabler **K1** vom Typ **struct kreis**):

Auf die Komponenten von **K1** greift man wie folgt zu:

```
K1.radius=8.0;
K1.mittelpunkt.x=10.0;
K1.mittelpunkt.y=10.0;
```

### 11.3 Operationen mit Strukturen

Mit Strukturvariablen sind 4 Operationen erlaubt:

1. Zuweisung
2. Selektion einer Komponente mit dem Punkt-Operator
3. Ermittlung der Größe mit dem **sizeof**-Operator
4. Ermittlung der Adresse mit dem **&**-Operator

Weiterhin können sie als Parameter und als Ergebniswert einer Funktion auftreten. Weitere Operationen wie z.B. der Vergleich zweier Strukturen sind nicht erlaubt.

Beispiel: Rechnen mit komplexen Zahlen

Eine Funktion für die komplexe Multiplikation **mul** kann wie folgt realisiert werden:

```
struct complex {
    double re, im;
};
struct complex mul (struct complex zahl1,
                   struct complex zahl2) {
    struct complex hilf;
    hilf.re =    zahl1.re*zahl2.re
               - zahl1.im*zahl2.im;
    hilf.im =    zahl1.re*zahl2.im
               + zahl1.im*zahl2.re;
    return hilf;
}
```

Analog lassen sich Funktionen für Addition **add**, Division **div** und Subtraktion **sub** schreiben.

Die komplexe Rechnung  $a=x*y-z/y$  lässt sich dann wie folgt ausführen:

```
complex a, x, y, z;
a=sub(mul(x, y), div(z, y))
```

#### Felder von Strukturen

Mehrere Strukturen gleichen Typs können zu Feldern zusammengefasst werden. Jede Komponente des Feldes ist dann eine Struktur.

Beispiel:

Nachfolgende Definition legt Speicherplatz für 100 Strukturen des Typs **struct complex** an:

```
struct complex vector[100];
```

Ein Zugriff auf die 54. Feldkomponente erfolgt dann beispielsweise mit:

```
vector[53].re = 0.7;
vector[53].im = 0.2;
```

## 11.4 Zeiger auf Strukturen

Der Inhalt einer Strukturvariablen vom Typ **X** wird, wie bei allen anderen Objekte, die Speicherplatz belegen, unter einer zugeordneten Adresse im Speicher abgelegt. Diese Adresse kann einer Zeigervariablen vom Typ 'Zeiger auf Struktur vom Typ **X**' zugewiesen werden.

Beispiel:

```
struct stud {
    char name[30];
    char vorname[20];
    short semester;
};
struct stud Person1, *pPerson;
pPerson=&Person1;
```

Um über den Zeiger auf die Komponenten zuzugreifen, kann der Inhaltsoperator verwendet werden:

```
printf("%s studiert im %d Semester.",
      (*Person).name, (*pPerson).semester);
```

### Der -> Operator

Da die Klammerschreibweise umständlich ist, gibt es in C den -> Operator als Abkürzung. Beispielsweise kann statt:

```
(*pPerson).name
```

alternativ auch

```
pPerson->name
```

geschrieben werden. Allgemein kann also mit:

```
<Strukturzeiger>-><Komponentenname>
```

über einen Zeiger auf die Komponente einer Struktur zugegriffen werden.

### Einbinden von Komponenten des eigenen Typs

Strukturen dürfen einen Zeiger auf den eigenen Strukturtyp enthalten, nicht aber eine Komponente vom eigenen Typ.

Beispiel:

```
struct richtig {
    struct richtig *pA; /* erlaubt */
};
struct fehlerhaft {
    struct fehlerhaft A; /* nicht erlaubt !!! */
};
```

Somit ist die Verwendung des eigenen Strukturtyps zur Vereinbarung eines Zeigers schon zu einem Zeitpunkt möglich, an dem die Deklaration des Typs selber noch nicht abgeschlossen ist.

## Gegenseitiger Verweis zweier Strukturen aufeinander

Benötigt man zwei Strukturtypen, die gegenseitig über Zeiger aufeinander verweisen, so muss man einen Strukturtyp schon bekannt machen, bevor er deklariert ist. Dies erfolgt mit einer **unvollständigen Typdeklaration**.

### Beispiel:

Es sollen die Datenstrukturen flip und flop vereinbart werden, die gegenseitig aufeinander verweisen:

```
struct flip; /* Unvollständige Typdeklaration */
struct flop { /* Deklaration von struct flop */
    struct flip *f;
};
struct flip { /* Deklaration von struct flip */
    struct flop *f;
};
```

Im selben Gültigkeitsbereich, in dem die unvollständige Typdeklaration erfolgt, muss auch die vollständige Deklaration durchgeführt werden.

### Beispiel: Vollständiges Programm

```
#include <stdio.h>
#include <stdlib.h>

int main () {
    int i ;
    struct person {
        char name [20];
        char vorname [20];
    };
    struct student {
        struct person name;
        float zensur;
    };
    struct student stud , *stud_p;
    struct student gruppe_WS02[40];

    strcpy (stud.name.name, "Meiser");
    strcpy (stud.name.vorname, "Hans");
    stud.zensur = 1.3f;
```

```

stud_p = (struct student *)
          malloc(sizeof(struct student));
if (stud_p == NULL) {
    puts ("Fehler beim Allokieren");
    return 1;
}
strcpy ((*stud_p).name.name, "Muster");
strcpy (stud_p->name.vorname, "Wilhelm");
stud_p->zensur = 2.0f;

gruppe_WS02[0] = stud;
gruppe_WS02[1] = *stud_p;
for (i = 0 ; i < 2 ; i++) {
    printf ("Zensur fuer %s %s: %3.1f\n",
           gruppe_WS02[i].name.vorname,
           gruppe_WS02[i].name.name,
           gruppe_WS02[i].zensur);
}
free (stud_p);
return 0;
}

```

### 11.5 union-Datenstruktur

Ergänzend zur **struct**-Datenstruktur können in C über eine **union**-Datenstruktur eigene zusammengesetzte Typen vereinbart werden. Die Form entspricht der **struct**-Vereinbarung, es wird jedoch das Schlüsselwort **union** verwendet. Im Unterschied zum **struct** werden bei einer **union**-Vereinbarung alle spezifizierten Elemente im Speicher auf den gleichen Speicherplatz gelegt. Sie stellen somit eine Alternative dar. Bei Verwendung einer **union**-Datenstruktur sollte somit immer nur eines der Elemente angesprochen werden.

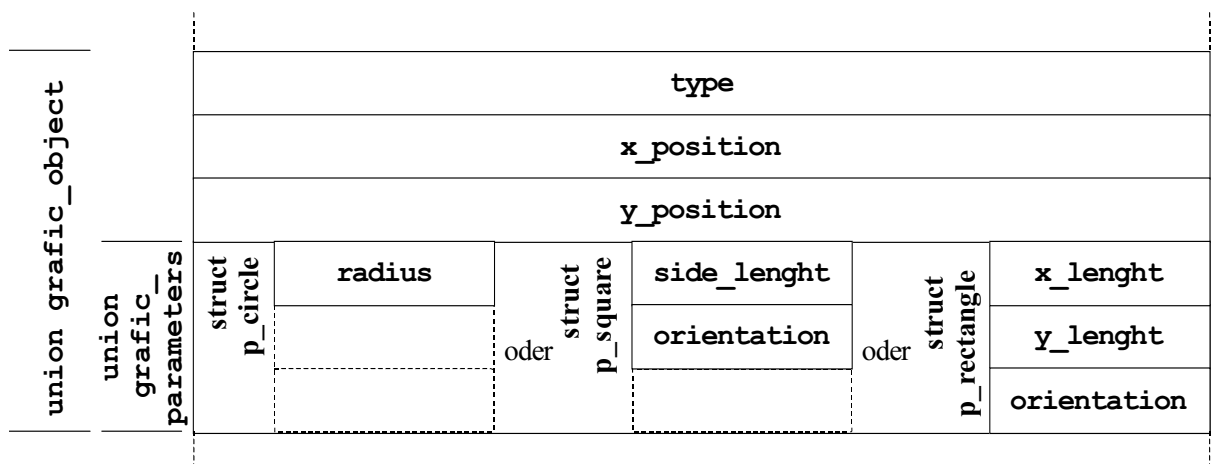
Beispiel: Vereinbarung einer Datenstruktur für Grafikobjekte

```

struct grafic_object {
    int type; /* 0-Point, 1-Circle,
              2-Square, 3-Rectangle */
    int x_position;
    int y_position;
    union grafic_parameters {
        struct p_circle {
            int radius;
        } circle;
        struct p_square {
            int side_length;
            int orientation;
        } square;
        struct p_rectangle {
            int x_length;
            int y_length;
            int orientation;
        } rectangle;
    } param;
};

```

Anordnung im Speicher:



Zugriff:

```

struct grafic_object g1;
g1.type = 2; /* square */
g1.x_position=10; g1.y_position = 20;
g1.param.square.side_length = 100;
g1.param.square.orientation = 45;

```

Achtung: Der Zugriff auf andere Elemente der **union**-Struktur wird nicht überwacht. Folgende Zugriffe sind möglich, aber vermutlich nicht gewünscht:

```
struct grafic_object g2;
g2.type = 1; /* circle */
g2.x_position=15; g1.y_position = 29;
g2.param.rectangle.x_length = 100; /* !! */
g2.param.rectangle.y_length = 15; /* !! */
g2.param.rectangle.orientation = 180; /* !! */
```

Damit wird im Beispiel der Radius indirekt auf 100 gesetzt.

Die Überwachung der Konsistenz der **union**-Datenstruktur obliegt alleine dem Programmierer. Von Seiten C wird keine Konsistenzüberprüfung vorgenommen.

## 11.6 Bitfelder

Eine besondere Komponente der **struct**- und **union**-Datenstrukturen ist das Bitfeld. Es erlaubt den Zugriff auf einzelne Bits eines Speicherwortes.

Als Bitfeld-Typ ist üblicherweise nur der **int**-Datentyp als **signed** oder **unsigned** erlaubt, manche Compiler erlauben auch die Verwendung vom Typ **char**.

Bitfelder sind zur Unterstützung von Hardware gedacht, wo in Peripheriebausteinen einzelne Bitgruppen gelesen und geschrieben werden müssen.

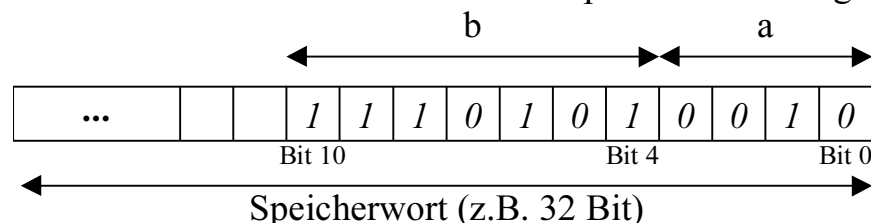
Beispiel: Nachfolgender Code zeigt die Verwendung von Bitfeldern:

```
struct Bitfeld_Beispiel {
    unsigned int a:4;
    signed int b:7;
};

struct Bitfeld_Beispiel bf;

bf.a=2; /* 0010 */
bf.b=-11; /* 1110101 */
```

Im Speicher werden  $4+7=11$  Bits in einem Speicherwort belegt:



Die Komponente a wird als positive Zahl mit 4 Bit, die Komponente b als ganze 7-Bit Zahl in 2er-Komplementdarstellung verwendet.

Werden Lücken zwischen zwei Bitfeldern benötigt, können Komponenten ohne Namen eingefügt werden. Muss ein Bitfeld am Wortanfang stehen, wird vorher ein Bitfeld der Länge 0 eingefügt.

Beispiel: Lücken zwischen Bitfeldern, Wort-Alignment.

```

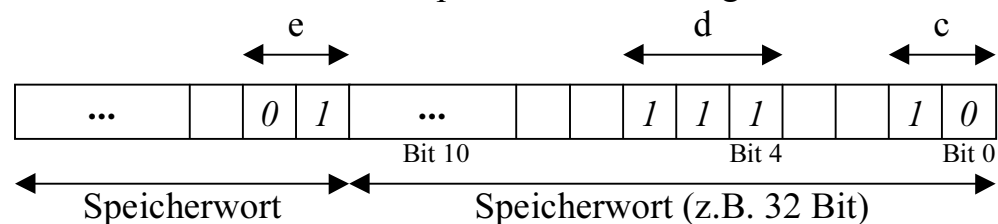
struct Bitfeld_Beispiel {
    unsigned int c:2;
    unsigned int :2;
    signed int d:3;
    unsigned int :0;
    unsigned int e:2;
};

struct Bitfeld_Beispiel bf;

bf.c = 2; /* 10 */
bf.d = -1; /* 111 */
bf.e = 1; /* 01 */

```

Im Speicher werden Bits in zwei Speicherworten belegt:



Es ist zu beachten, dass die Implementierung von Bitfelder sehr stark von dem jeweils verwendeten Rechner abhängen. Daher sollte man beim Erstellen portabler Programme, die auf unterschiedlichen Rechnern und Prozessoren ablaufen sollen, auf die Verwendung von Bitfeldern verzichten. Die Implementierung von Bitoperationen kann dann alternativ mit den bitweisen Logikoperatoren (&, |, ~, ^) erfolgen.

## 12 Aufzählungstypen

Mit dem **enum**-Schlüsselwort (engl.: enumeration) lassen sich Aufzählungstypen vereinbaren:

```
enum Boolean {FALSE, TRUE};
```

Mit dem Beispiel wird ein neuer Datentyp **enum Boolean** vereinbart, dem die Aufzählungskonstanten **FALSE** und **TRUE** zugeordnet werden.

Die Aufzählungskonstanten haben einen ganzzahligen Wert vom Typ **int**. Werden den Aufzählungskonstanten keine Werte zugeordnet, erhalten sie mit 0 beginnend aufeinanderfolgende Integerwerte. In obigem Beispiel ist somit der Wert von **FALSE** 0 und von **TRUE** 1.

Den Aufzählungskonstanten können auch explizit Werte zugeordnet werden:

```
enum Zahlen = {EINS=1, ZWEI, VIER=4, DREI=3};
```

Werden Konstanten nicht explizit belegt (im Beispiel die Konstante **ZWEI**), erhalten sie den um 1 erhöhten Wert der vorhergehenden Konstante zugeordnet.

Im selben Gültigkeitsbereich darf eine Aufzählungskonstante nur in einer **enum**-Typvereinbarung vorkommen. Auch darf der Name nicht mit einem Variablennamen im gleichen Bereich übereinstimmen.

Die Vereinbarung einer **enum**-Variablen erfolgt durch Angabe des **enum**-Typs gefolgt von dem Variablennamen:

```
enum Boolean Schalter;  
enum Zahlen Test;
```

Das Ansprechen von **enum**-Variablen erfolgt wie mit integer-Variablen. Man kann einer **enum**-Variablen auch integer-Werte zuweisen. Eine Überprüfung auf den spezifizierten Wertebereich erfolgt nicht.

```
Schalter = TRUE;  
Test = VIER;  
Schalter = 27; /* Erlaubt aber unschoen */  
Test = 11; /* Erlaubt aber unschoen */
```

## 13 Typdefinitionen

In C können mit Hilfe der **typedef**-Anweisung neue Typnamen definiert werden. Dabei werden jedoch keine neuen Variablentypen geschaffen, sondern vorhandene Typen mit einem neuen Namen versehen.

Solch einen neuen Namen bezeichnet man auch als Alias-Namen (alias: eng.: Deckname).

**typedef** verhält sich ähnlich wie **#define**, nur dass der Textersatz vom Compiler übernommen wird. Dabei überprüft der Compiler auch sofort, ob der verwendete Typname korrekt angewendet wird.

Durch den Ersatz einer komplizierten Typvereinbarung in C durch einen neuen Namen lassen sich komplizierte Datentypen in C übersichtlicher darstellen.

Beispiel:

```
typedef int TYP_INT;  
TYP_INT i, j, k, l;
```

Der Typname **TYP\_INT** kann wie der Typ **int** genutzt werden.

Beispiele:

```
typedef int *PINT; /* PINT ist Synonym für int* */  
PINT k;           /* k ist Zeiger auf int. */  
  
typedef struct {  
    char ort[40];  
    int plz;  
    char *land;  
    long int anzahl;  
} LAND_STRUKTUR;  
LAND_STRUKTUR Niedersachsen, Nordrheinwestfalen;  
  
typedef int *(*FP)(char *,char *);  
FP f; /* f ist Zeiger auf eine Funktion mit  
      Rückgabewert Zeiger auf int und  
      2 Parametern vom Typ char* */  
FP (*g)[10]; /* Zeiger auf ein Feld mit  
             10 Komponenten vom Typ FP */  
/* Gleichartige Definition ohne typedef */  
typedef int *(*(*h)[10])(char *,char *);
```

Verabredungsgemäß werden Typnamen **GROSS** geschrieben. Dies ist jedoch keine Vorgabe des Compilers sondern nur eine Vereinbarung. Diese Vereinbarung bietet den Vorteil, dass man sofort selbstdefinierte Typen in einem Programmcode erkennt.

## 14 Direkte Sprungbefehle

Im Kontrast zum strukturierten Ansatz von C steht die Sprunganweisung **goto**. Sie erlaubt beliebige Sprünge innerhalb einer Funktion. Ein Sprung aus einer Funktion in eine andere Funktion ist mit der **goto**-Anweisung nicht möglich.

Die Sprunganweisung benötigt eine Marke (Label) als Sprungziel. Eine Marke kann vor jede ausführbare Anweisung gesetzt werden. Sie wird durch einen Namen gefolgt von einem Doppelpunkt deklariert.

Beispiel: Überspringen einer Anweisung

```
#include <stdio.h>
int main() {
    goto MacheNichts;      /* Sprung zur Marke*/
    printf("Hello World\n");
    MacheNichts: return 0; /* Marke vereinbaren */
}
```

Bei obigem Code wird die **printf**-Anweisung nicht ausgeführt.

Da die **goto**-Anweisung dem Gedanken der strukturierten Programmierung widerspricht, sollte sie bei der Implementierung von Algorithmen keine Verwendung finden. Hilfreich ist diese Anweisung jedoch, wo die algorithmische Struktur aufgrund von Fehlern verlassen werden muss und dies mit **break** und **continue** nur umständlich möglich ist. In diesem Fall kann direkt von der fehlerhaften Stelle des Algorithmus in eine Fehlerbehandlung gesprungen werden.

Beispiel: Fehlerbehandlung ohne und mit **goto**-Anweisungen

Nachfolgendes Code-Beispiel zeigt eine Überprüfung der Fehler im Algorithmus und eine konzentrierte Fehlerbehandlung am Ende der Funktion. Man erkennt, dass durch die Fehlerbehandlung die Schachtelungstiefe im Algorithmus mit jeder Fehlerüberprüfung zunimmt.

Man könnte auch die Fehlerbearbeitung verteilt im Algorithmus durchführen und die Funktion im Fehlerfall im Inneren des Algorithmus mit **return** verlassen.

Beide Lösungen sind nicht sonderlich zufriedenstellend.

Das zweite Codebeispiel zeigt die Verwendung der **goto**-Anweisung im Fehlerfall. Sobald ein Fehler entdeckt wird, wird die Ursache des Fehlers in einer Fehlervariablen gespeichert und ein völlig separater Fehlerbehandlungsteil der Funktion mittels **goto Fehler** angesprungen. Damit ist eine saubere Trennung zwischen dem zu realisierenden Algorithmus und der Fehlerbehandlung möglich.

## Codebeispiel 1: Fehlerbehandlung innerhalb des Algorithmus

```
#include <stdio.h>
#define ZUWENIG_ARGUMENTE      1
#define KANN_IN_NICHT_OEFFNEN  2
#define KANN_OUT_NICHT_OEFFNEN 3

int main(int argc, char*argv[]) {
    FILE *in, *out;
    int error;
    if (argc<3) {
        error = ZUWENIG_ARGUMENTE;
    } else {
        in=fopen(argv[1], "rb");
        if (NULL==in) {
            error = KANN_IN_NICHT_OEFFNEN;
        } else {
            out=fopen(argv[2], "wb");
            if (NULL==in) {
                error = KANN_OUT_NICHT_OEFFNEN;
            } else {
                ... /* Programm weiter ausführen,
                    aber jede Fehlerbehandlung
                    erhöht die Schachtelung */
            } /* of 3rd else */
        } /* of 2nd else */
    } /* of 1st else */
    if (error) {
        /* Fehlerbehandlung */
        switch(error) {
            case ZUWENIG_ARGUMENTE:
                ...
                break;
            case KANN_IN_NICHT_OEFFNEN:
                ...
                break;
            case KANN_OUT_NICHT_OEFFNEN:
                ...
                break;
        }
        return 3;
    } else { return 0; }
}
```

## Codebeispiel 2: Fehlerbehandlung mit goto

```
#include <stdio.h>
#define ZUWENIG_ARGUMENTE      1
#define KANN_IN_NICHT_OEFFNEN  2
#define KANN_OUT_NICHT_OEFFNEN 3

int main(int argc, char*argv[]) {
    FILE *in, *out;
    int error;
    if (argc<3) {
        error = ZUWENIG_ARGUMENTE;
        goto Fehler;
    }
    in=fopen(argv[1], "rb");
    if (NULL==in) {
        error = KANN_IN_NICHT_OEFFNEN;
        goto Fehler;
    }
    out=fopen(argv[2], "wb");
    if (NULL==in) {
        error = KANN_OUT_NICHT_OEFFNEN;
        goto Fehler;
    }
    /* ... Programm weiter ausführen,
       Fehlerbehandlung erhöht
       nicht mehr die Schachtelung */
    return 0;

/* ----- Fehlerbehandlung ----- */
Fehler:
    switch(error) {
        case ZUWENIG_ARGUMENTE:
            /*...*/
            break;
        case KANN_IN_NICHT_OEFFNEN:
            /*...*/
            break;
        case KANN_OUT_NICHT_OEFFNEN:
            /*...*/
            break;
    }
    return 3;
}
```