

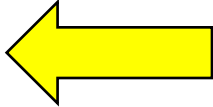
**Folien zur Vorlesung
Grundlagen systemnahes Programmieren
Sommersemester 2016
(Teil 4)**

Prof. Dr. Franz Korf

Franz.Korf@haw-hamburg.de

Kapitel 4: Programmiersprache C - Fortgeschrittene Themen

Gliederung

- Adressen und Zeiger 
- Felder
- Strings
- Strukturen
- Dynamische Speicherverwaltung
- Zeigerarithmetik
- Selbstdefinierte Datentypen
- Zusammenfassung

Adressen und Zeiger

Zeiger / Pointer

- eine Variable, die eine **Adresse** enthält
- ist Typ-gebunden

Deklaration:

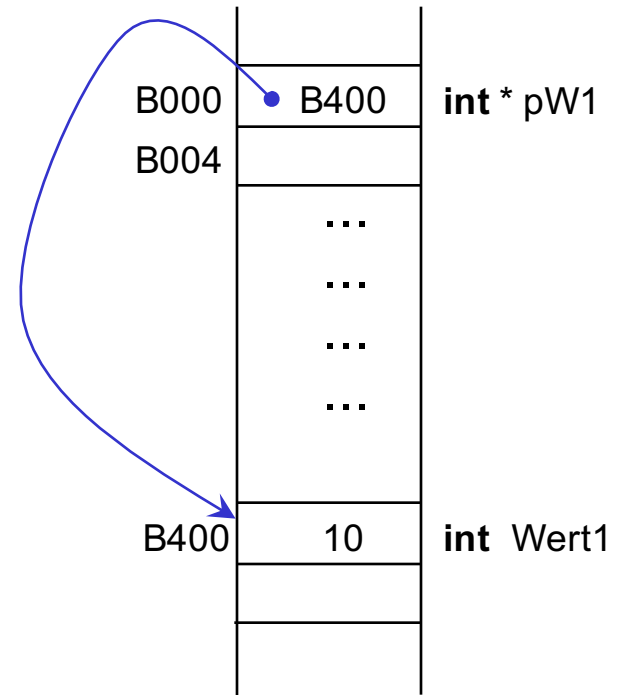
- durch * vor dem Variablenname

Initialisierung:

- Ein Zeiger muss mit einer gültigen Adresse belegt werden.
- Die Adresse einer Variablen erhält man durch den Adressoperator &.

Beispiel:

```
int Wert1 = 10;  
  
int *pW1; /* Zeiger auf int */  
  
/* Initialisierung */  
pW1 = &Wert1;
```



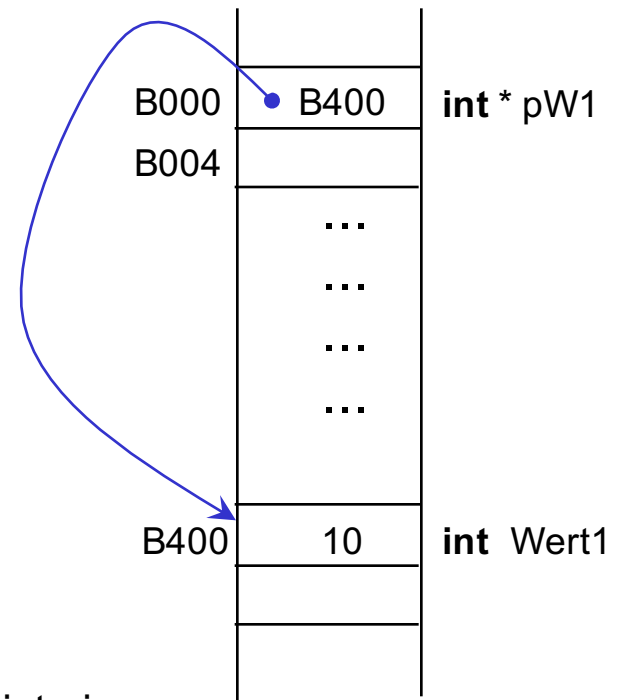
Zugriff auf Daten über Zeiger

Wendet man den unären Operator `*` auf einen Zeiger an, so erhält man den Wert des Objektes, auf den der Zeiger gerade zeigt.

```
int  Wert1 = 10,  *pW1;  
  
/* Initialisierung */  
pW1 = &Wert1;  
  
/* Zugriff */  
printf("Wert %d, Adresse %X", *pW1, pW1);
```

WICHTIG:

- In der **Deklaration** bedeutet `*pW1`: "pWert1 ist ein Zeiger auf ein Objekt von Typ"
- In **Ausdrücken** bedeutet `*pW1`: „Inhalt der Speicherzelle, auf den der Zeiger pW1 zeigt“.

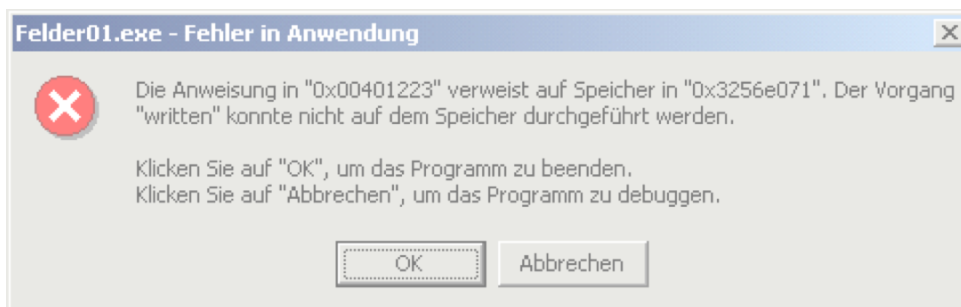


Zugriff auf Daten über Zeiger (Fortsetzung)

Fehlergefahr "dangling pointer" -- Pointer sind nicht initialisiert.

```
int *pW1;  
/* Zugriff */  
*pW1=10;
```

Bedeutung: Schreibe eine 10 in die Speicherstelle, auf die pW1 zeigt, aber pW1 „zeigt in die Wüste“.



Fehlermeldung QNX:
Segmentation Fault in
Debugger

Man erhält nicht immer
diese Fehlermeldung,
manchmal findet der
fehlerhafte Zugriff statt.

Zeiger stets initialisieren (ggf. mit NULL)

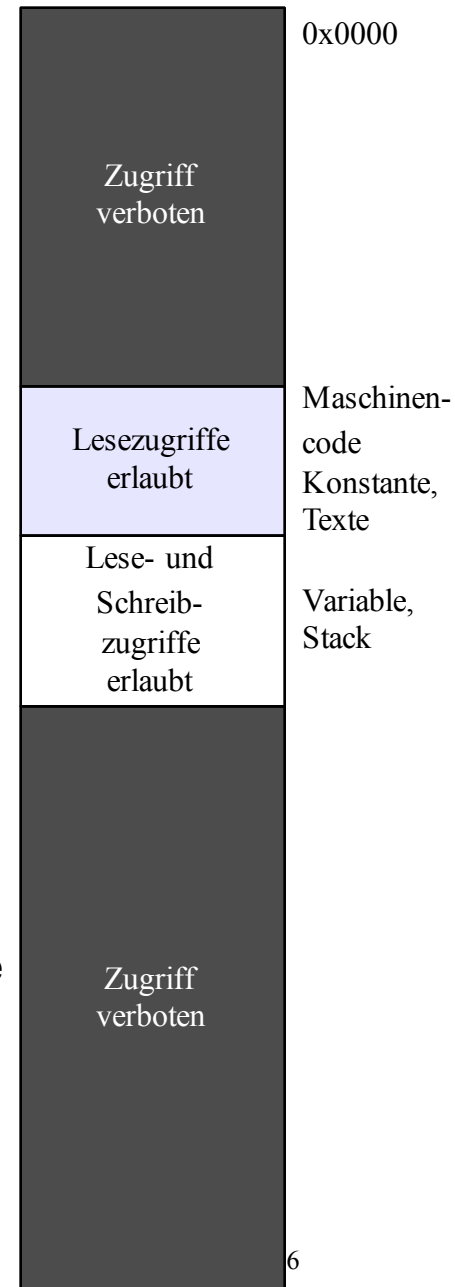
Zugriff auf Daten über Zeiger (Fortsetzung)

Häufige Fehlerursachen von "dangling pointer" Fehlern:

- nicht initialisierte Zeiger (Pointer)
- Verweis auf dynamisch erzeugte Elemente, die in der Zwischenzeit wieder gelöscht worden sind.

Wichtig

- Zugriffe via Zeiger überwacht das Programm nicht!
 - Einziger Schutz durch das Betriebssystem und die MMU (Memory-Management Unit).
 - Betriebssystem löst ein Signal aus und stoppt damit in der Regel das Programm: Speicherzugriffsfehler, Segmentation fault
 - Globaler nicht initialisierter Pointer enthält die Adresse 0 → Fehlerhafter Zugriff wird erkannt
 - Lokaler nicht initialisierter Pointer enthält beliebige Adresse
 - Fehlerhafter Zugriff wird nicht immer erkannt.
 - Führt zu sehr schwer auffindbaren Fehlern
- Alle Zeiger mit NULL initialisieren



ÜBUNG: Einfache Zeigeroperationen

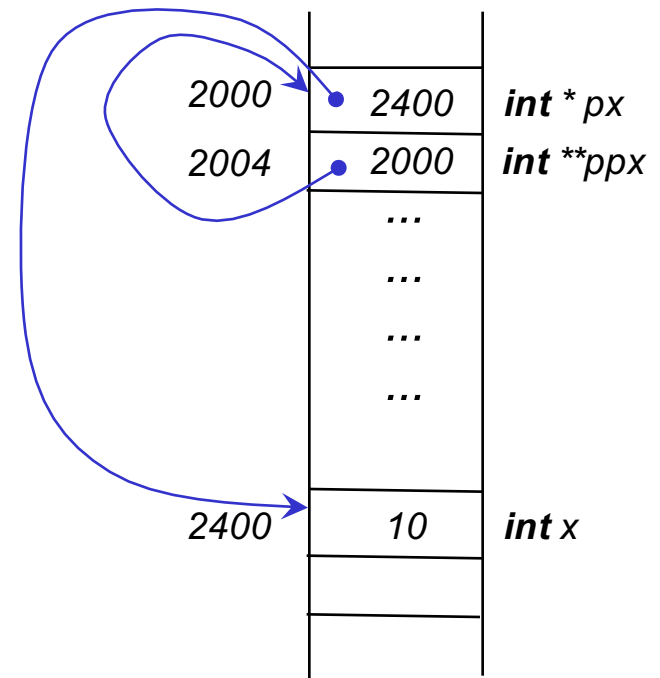
Gegeben sind das Programm

```
int x = 10;
int *px;
int **ppx;
```

```
px = &x;
ppx = &px;
```

```
printf("%d %d %d", x, &x, px);
printf("%d %d %d", &px, *px, *ppx);
printf("%d %d %d", &ppx, **ppx, ppx);
```

und das Speicherbild (Memory Map)



Was wird ausgedruckt?

```
10 2400 2400
2000 10 2400
2004 10 2000
```

Wiederholung: Funktionsparameter: Call-by-Value vs. Call-by-Reference

Die Funktionsparameter in C sind **Call-by-Value** Parameter, d.h. es werden die Werte übergeben.

Beispiel: Die Funktion

```
void swap(int x, int y)    /* FALSCH */
{
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

macht nichts sinnvolles, da sie nur mit den Werten und nicht auf den Speicherplätzen arbeitet.

In C wird **Call-by-Reference** über Zeiger **nachgebildet**, die Call-by-Value Parameter sind.

Wiederholung: Funktionsparameter: Call-by-Value vs. Call-by-Reference

Zeiger als Funktionsparameter modellieren Call-by-Reference in C.

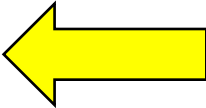
Beispiel: Die Funktion

```
void swap(int *x, int *y)
{
    int tmp;
    tmp = *x;
    *x = *y;
    *y = tmp;
}
```

implementiert die Vertauschung.

Kapitel 4: Programmiersprache C - Fortgeschrittene Themen

Gliederung

- Adressen und Zeiger
- Felder 
- Strings
- Strukturen
- Dynamische Speicherverwaltung
- Zeigerarithmetik
- Selbstdefinierte Datentypen
- Zusammenfassung

Ein- und mehrdimensionale Felder

Ein **Feld** ist eine Menge von Variablen des gleichen Typs, die im Speicher hintereinander abgelegt sind.

Die Elemente eines Feldes werden über den Feldnamen und ihre Position - Index im Feld referenziert.

Es werden keine Zusatzinformationen wie z.B. die Länge gespeichert.

Definition: Typ Variablenname [Anzahl der Elemente]

Zugriff: Index des ersten Feldes ist 0 !

```
/* Definition */  
int  x[2];    // Feld aus 2 int Werten  
char txt[2]; // Feld aus 2 char Werten
```

```
/* Initialisierung */  
x[0]=12;      x[1]=25;  
txt[0]='A';   txt[1]='b';
```

```
printf("x[0] ist %d \n", x[0]);  
printf("txt[0] ist %c \n", txt[0]);
```

Definition und Initialisierung von Feldern

```
/* mit Angabe der Feldgröße */  
int  x[10]  = {3, 6, 9, 12, 15, 18, 21, 24, 27, 30};  
char txt[5] = {'a', 'b', 'c', 'd', 'e' };
```

```
/* ohne Angabe der Feldgröße */  
int  x[]    = {3, 6, 9};  
char txt[]  = {'a', 'b', 'c', 'd', 'e' };
```

```
/* Teilinitialisierung */  
int  x[10]  = {3, 6};  
char txt[5] = {'a', 'b'};
```

Vermeidung von Fehlern beim Zugriff auf Feldelemente

```
/* mit Angabe der Feldgröße */  
int x[10] = {3, 6, 9, 12, 15, 18, 21, 24, 27, 30};  
int sum;  
  
sum=0;  
for( i=0; i<10; i++){  
    sum += x[i];  
}
```

**Fehlerträchtig:
Angabe der Feldgröße ist
mehrfach im Programm
enthalten!**

Besser

```
#define ANZAHL 10  
  
/* mit Angabe der Feldgröße */  
int x[ANZAHL] = {3, 6, 9, 12, 15, 18, 21, 24, 27, 30};  
int sum;  
  
sum=0;  
for( i=0; i<ANZAHL; i++){  
    sum += x[i];  
}
```

Bestimmung der Anzahl der Feldelemente

```
/* ohne Angabe der Feldgröße */  
int  x[]    = {3, 6, 9};  
char txt[] = {'a', 'b', 'c', 'd', 'e' };
```

- C speichert keine Zusatzinformationen zum Feld ab!
- Es gibt weder eine Funktion noch ein Attribut zur Ermittlung der Feldgröße!!
 - allerdings kann der Speicherbedarf eines Feldes ermittelt werden:
 - `sizeof(x)` liefert die benötigte Gesamtzahl der Bytes des Feldes x
 - `sizeof(x[0])` liefert die benötigte Anzahl Bytes eines Elementes von x
 - `n = sizeof(x)/sizeof(x[0])` liefert die Anzahl Elemente von x

Mehrdimensionale Felder

```
/* mit Angabe der Feldgröße */
```

```
int x2[2][3];
```

```
char c2[3][2] = { {'a', 'b'}, /* Feld von 3 Elementen */  
                  {'c', 'd'}, /* mit jew. 2 Elementen */  
                  {'e', 'f'} };
```

```
char c2[][2] = { {'t', 'f'}, /* Felddimension 1 */  
                 {'f', 'f'}, /* kann entfallen. */  
                 {'f', 'x'} };
```

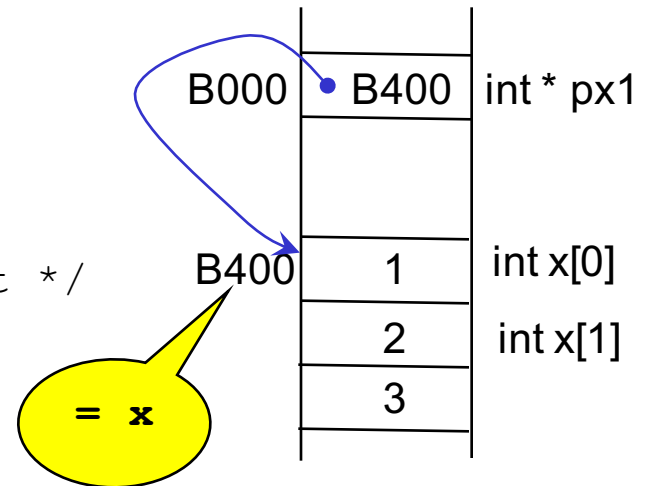
```
int x3[4][3][2] = { { {0,1},{1,0},{1,1} },  
                     { {1,1},{0,0},{1,0} },  
                     { {0,0},{1,1},{1,1} },  
                     { {0,1},{0,1},{0,1} } };
```

Felder und Zeiger

Zeiger und Felder sind eng miteinander verbunden – jede Operation mit Feldern kann auch mit Zeigern formuliert werden.

```
int x[] = {1, 2, 3, 4, 5, 6, 7};
int *px1 = x;
```

```
/* Folgende Schreibweisen sind äquivalent */
printf("Adr. des 1. Zeichens=%X", &x[0]);
printf("Adr. des 1. Zeichens=%X", x);
printf("Adr. des 1. Zeichens=%X", px1);
```



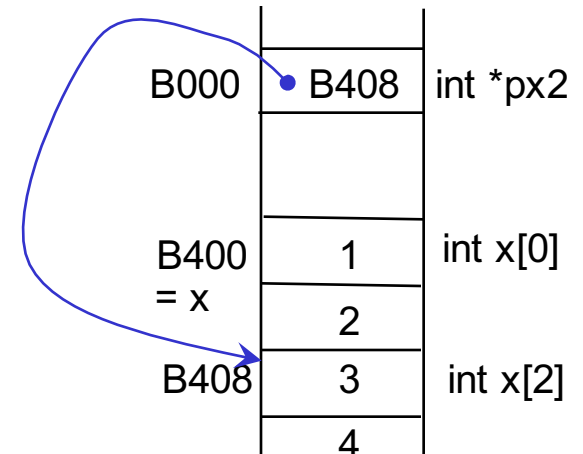
Der Feldname ohne Index liefert die Adresse des 1. Feldelements.

```
int x[] = {1, 2, 3, 4, 5, 6, 7};
int *px2 = &x[2];
```

```
printf("%d %d %d", *px2, px2[0], px2[3]);
```

Ein indizierter Zeiger liefert den indizierten Wert.

Ausgabe: 3 3 6



Übungsaufgabe

```
int x[] = {1, 2, 3, 4, 5, 6, 7};  
int *px = x;  
  
printf("%d %d %d", *px, px[0], px[3]);
```

Lösung: 1 1 4

Hinweise:

- Ein indizierter Zeiger liefert den indizierten Wert.
- Zeiger sind sehr fehleranfällig

Schreiben Sie den Code so einfach wie möglich und so trickreich wie nötig.

Felder und Zeiger (Fortsetzung)

```

/* Definition und Initialisierung */
char x[4][3][2] =
    { { { 0, 1}, { 2, 3}, { 4, 5} },
      { { 6, 7}, { 8, 9}, {10,11} },
      { {12,13}, {14,15}, {16,17} },
      { {18,19}, {20,21}, {22,23} } };
/* Zugriff auf Feldelemente */
y = x[3][2][1]; /* 23 */

```

$x = x[0] = x[0][0] = 1000$

1002

1004

$x[1] = x[1][0] = 1006$

1008

$x[1][2] = 100A$

00	int x[0][2][1]
01	
02	
03	
04	
05	int x[1][2][0]
06	
07	
08	
09	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	
20	
21	
22	
23	

Unvollständige Feldnamen sind Adressen und können wie Zeiger verwendet werden.

Allerdings sind die Typen unterschiedlich!

Beispiele:

- x $= x[0]$ $= x[0][0] = \$1000$
- $x[1]$ $= x[1][0] = \$1006$
- $x[1][2]$ $= \$100A$
- $x[1][2][0]$ $= 10$ (**der Wert!!**)

Felder und Zeiger (Fortsetzung)

```
/* Definition und Initialisierung */
char x[4][3][2] = { { { 0, 1},{ 2, 3},{ 4, 5} },
                    { { 6, 7},{ 8, 9},{10,11} },
                    { {12,13},{14,15},{16,17} },
                    { {18,19},{20,21},{22,23} } };

/* Zugriff auf Feldelemente */
y = x[3][2][1]; /* 23 */
```

Zeiger auf Teile des Feldes:

```
char* px;
char (*pxa)[2]; /* nicht *pxa[2]! */
char (*pxb)[3][2];
```

- Klammern (*pxa) notwendig, da [] eine höhere Präzedenz hat als *
- char (*pxa)[2] deklariert einen Zeiger auf ein Feld mit 2 char Werten.
- char *pxa[2] deklariert ein Feld mit zwei Zeigern, die auf char zeigen.

Felder und Zeiger (Fortsetzung)

```
/* Definition und Initialisierung */
char x[4][3][2] = { { { 0, 1},{ 2, 3},{ 4, 5} },
                    { { 6, 7},{ 8, 9},{10,11} },
                    { {12,13},{14,15},{16,17} },
                    { {18,19},{20,21},{22,23} } };

/* Zugriff auf Feldelemente */
y = x[3][2][1]; /* 23 */
```

Verwendung der Zeiger:

```
px  = x[1][1];      /* Zeiger auf {8,9} */
a   = *px;
a   = px[1]

pxa = x[1];
                        /* Zeiger auf {{6,7},{8,9}..} */
a   = pxa[2][0];
b   = (*pxa)[1];
pxb = x;              /* Zeiger auf {{{0,1},{2,3}..} */
a   = pxb[2][1][0];
b   = (*pxb)[1][0];
```

Felder und Zeiger (Fortsetzung)

Es darf maximal die erste Dimension eines mehrdimensionalen Feldes „offen“ bleiben.

```
/* Definition und Initialisierung */
char x[][3][2] =
    { { { 0, 1},{ 2, 3},{ 4, 5} },
      { { 6, 7},{ 8, 9},{10,11} },
      { {12,13},{14,15},{16,17} },
      { {18,19},{20,21},{22,23} } };
```

Dies gilt auch für formale Funktionsparameter.

```
void f(char x[][3][2]) { ... }
```

Alternative, wenn Dimensionen erst zur Aufrufzeit bekannt sind:

```
void f(int dim_1, int dim_2, int dim_3, char *x) { ... }
```

ÜBUNG: Umgang mit Zeigern

Zeichnen Sie die Memory Map. Was wird ausgegeben?

```
int a[]={1,2,3,4,5,6,7};

int main() {
    int *p1;
    int **p2;

    p1=&a[2];
    p2=&p1;

    printf("%d\n",a[3]);
    printf("%d\n",*p1);
    printf("%d\n",p1);
    printf("%d\n",&p1);
    printf("%d\n",p2);
    printf("%d\n",&p2);
    printf("%d\n",p2[0][2]);

    return 0;
}
```

Annahmen:

- Das Feld a beginne bei Adresse 0x1000.
- Der Zeiger p1 stehe bei Adresse 0x2000 gefolgt vom Zeiger p2.

Das Beispiel hat didaktischen Wert, zeugt aber nicht gerade von gutem Stil.

Felder und Funktionsparameter

- Felder dürfen als Funktionsparameter verwendet werden.
- Parameter ist der Zeiger auf das erste Element des Feldes.
- Die Größe des Feldes wird nicht kontrolliert.
- Die Größe des Feldes ist in der Funktion nicht bekannt.

```
void fkt(int a[9]){  
    a[3] = 5;  
    printf( "sizeof(a) ist %ld.\n", sizeof(a) );  
}
```

gibt die Größe des
Zeigers aus!

```
int main(void){  
    int a[7];  
    fkt( a );  
    printf( "a[3] ist %d.\n", a[3] );  
    printf( "sizeof(a) ist %ld.\n", sizeof(a) );  
}
```

gibt die Größe des
Feldes aus!

Felder und Funktionsparameter

- Größe des Feldes in einem separaten Parameter übergeben

```
void fkt(int a[], int n) {  
    int i;  
    for( i=0; i<n; i++){  
        a[i] = 5;  
    }  
}
```

```
#define SIZEA 7  
#define SIZEB 3
```

```
int main(void) {  
    int a[SIZEA];  
    int b[SIZEB];  
    fkt( a, SIZEA );  
    fkt( b, SIZEB );  
}
```


ÜBUNG: Call-by-reference

Schreiben Sie ein Unterprogramm, das ein Feld von Integerwerten der Größe nach sortiert.

Ansatz: Durchlaufe das Feld immer wieder und vertausche jedes Mal, wenn nötig, benachbarte Elemente.

Der Vorgang kann abgebrochen werden, wenn bei einem Durchlauf kein Vertauschen mehr vorkommt (--> Bubblesort).

Weiter ist das Hauptprogramm zu schreiben, welches das Unterprogramm aufruft.

ÜBUNG: Call-by-reference (Fortsetzung)

```
#include <stdio.h>

void Sort(int *arr, int arr_size) {
    int did_swap = 1;
    int i, tmp;
    while (did_swap) {
        i = 0;
        did_swap = 0;
        while (i < arr_size - 1) {
            if (arr[i] > arr[i+1]) {
                tmp = arr[i];
                arr[i] = arr[i+1];
                arr[i+1] = tmp;
                did_swap = 1;
            }
            i++;
        }
    }
}
```

ÜBUNG: Call-by-reference (Fortsetzung)

```
void print_feld(int *arr, int arr_size)
{
    int i;

    printf(">>");
    for (i = 0; i < arr_size; i++) {
        printf(" %d", arr[i]);
    }
    printf(" <<\n");
}
```

ÜBUNG: Call-by-reference (Fortsetzung)

```
int main(void)
{
    int f1[] = {5, 4, 3, 2, 1, 6};
    int f2[] = {1, 2, 3, 4, 5, 6};
    int f3[] = {4, 3, 6, 2, 4, 5};

    print_feld(f1, 6); Sort(f1, 6);
    print_feld(f1, 6); printf("\n");


    print_feld(f2, 6); Sort(f2, 6);
    print_feld(f2, 6); printf("\n");

    print_feld(f3, 6); Sort(f3, 6);
    print_feld(f3, 6); printf("\n");

    return 0;
}
```

Kapitel 4: Programmiersprache C - Fortgeschrittene Themen

Gliederung

- Adressen und Zeiger
- Felder
- Strings 
- Strukturen
- Dynamische Speicherverwaltung
- Zeigerarithmetik
- Selbstdefinierte Datentypen
- Zusammenfassung

Zeichenketten (Strings)

Definition: Ein **string** ist

- Eine Sequenz von Zeichen.
- Eingeschlossen in Anführungszeichen.
- Mit '\0' abgeschlossen (Null-Zeichen).
- Gespeichert in einem Feld vom Typ `char[]`.
- Länge des Feldes ist um 1 größer als Länge des strings (Platz für abschließende '\0').

```
char txt1[] = "AB1";  
char txt2[4] = "AB1"; /* Platz für Nullzeichen */  
char txt3[] = {'A', 'B', '1', '\0'};
```

```
/* String ausgeben mit %s und Stringname */  
printf("%s %s %s", txt1, txt2, txt3);
```

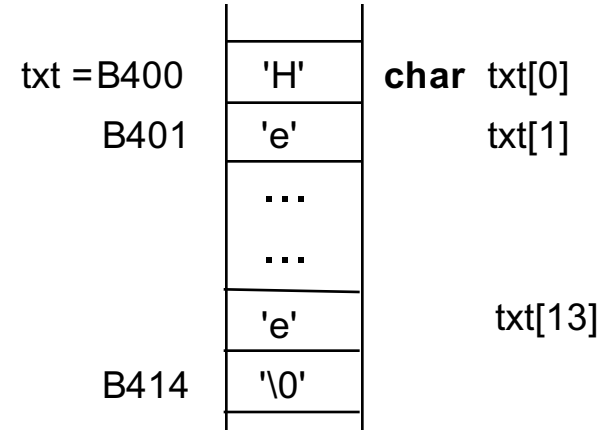
'A'	'B'	'1'	0
-----	-----	-----	---

= 0x41 0x42 0x31 0x0

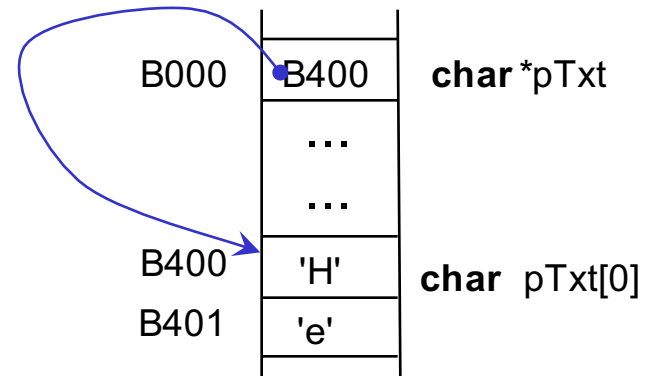
Zeichenketten und Zeiger

String Namen können wie Zeiger verwendet werden, (sind nur Adressen) !

```
char txt[] = "Herr der Ringe";  
printf("Adr. des 1. Zeichens=%X", txt);
```



```
char *pTxt = "Herr der Ringe";  
printf("Adr. des 1. Zeichens=%X", pTxt);
```



Mehrdimensionale String-Felder

Zweidimensionales **Feld von char**:

```
char names[5][10];
```

- Zugriff auf ein Zeichen: z.B. names[3][4].
- names ist ein zweidimensionales Feld mit 50 Elementen vom Typ char.

Eindimensionales **Feld mit Zeiger auf Strings**:

```
char* pnames[10];
```

- Zugriff auf ein Zeichen: z.B. pnames[3][4].
- Speicherreservierung nur für Zeiger!
- Keine Speicherreservierung für die Strings!
- Zugriff ist erst erlaubt, wenn dem Feldelement ein gültiger Zeiger zugewiesen wurde:
z.B. pnames[3] = names[3];

Mehrdimensionale String-Felder (Fortsetzung)

Beispiel 1:

Zweidimensionales Feld von char:

```
char names[][5] = {  
    "ARD",  
    "Z",  
    "RTL"  
};
```

Vorteil: Strings können verändert werden.

names[0]	A'
	R'
	D'
	\0'
	\0'
names[1]	'Z'
	\0'
	\0'
	\0'
	\0'
names[2]	'R'
	'T'
	'L'
	\0'
	\0'

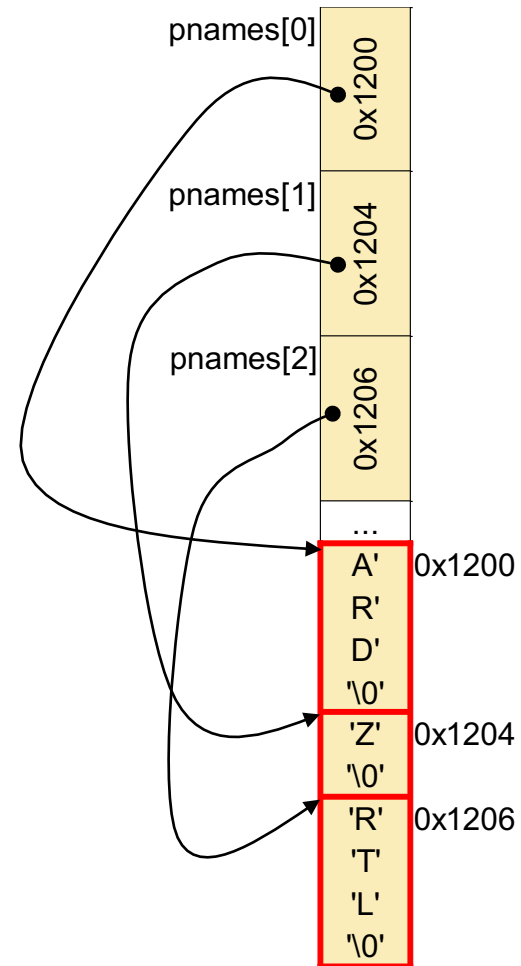
Mehrdimensionale String-Felder (Fortsetzung)

Beispiel 2:

Eindimensionales Feld von Zeigern,
die auf Strings zeigen:

```
char* pnames[] = {  
    "ARD",  
    "Z",  
    "RTL"  
};
```

Vorteil: Felder für Strings können
unterschiedlich lang sein.



ÜBUNG: Beispiele zu Feldern und Zeichenketten

Schreiben Sie eine C-Funktion, welche die Grossbuchstaben eines Strings zählt und ausgibt.

ÜBUNG: Beispiele zu Feldern und Zeichenketten (Fortsetzung)

```
int str_count(char s[])
/* str_count zaehlt die Grossbuchstaben im String s */
{
    int i = 0, count = 0;

    while (s[i] != '\0') {
        if (( 'A' <= s[i] ) && ( s[i] <= 'Z' )) {
            // Grossbuchstabe liegt vor
            count ++;
        }
        i++;
    }
    return count;
}
```

ÜBUNG: Umgang mit Strings

Gegeben sei ein String:

```
char Quellstring[]="Der Mol fühlt sich wohl am Pol."
```

Es soll ein Programm geschrieben werden, welches jedes Vorkommen der Buchstabenkombination "ol" durch "ool" ersetzt:

Stringbearbeitung: Funktionen in der C-Bibliothek (eine Auswahl)

Copying:

memcpy	Copy block of memory (function)
memmove	Move block of memory (function) (overlapping memory o.k.)
strcpy	Copy string (function)
strncpy	Copy characters from string (function)

Concatenation:

strcat	Concatenate strings (function)
strncat	Append characters from string (function)

Comparison:

memcmp	Compare two blocks of memory (function)
strcmp	Compare two strings (function)
strcoll	Compare two strings using locale (function)
strncmp	Compare characters of two strings (function)
strxfrm	Transform string using locale (function)

Vermeidung von Overflows:
Nur Funktionen nutzen, die
Größenparameter verwenden.

Fortsetzung

Searching:

memchr	Locate character in block of memory (function)
strchr	Locate first occurrence of character in string (function)
strcspn	Get span until character in string (function)
strpbrk	Locate character of char set in string (function)
strrchr	Locate last occurrence of character in string (function)
strspn	Get span of character set in string (function)
strstr	Locate substring (function)
strtok	Split string into tokens (function)

Other:

memset	Fill block of memory (function)
strerror	Get pointer to error message string (errno) (function)
strlen	Get string length (function)

Länge eines Strings: `strlen()` oder `strnlen`

Länge des Strings festgelegt durch abschließende Terminierung mit `\0`-Zeichen

- Nicht verwechseln mit `sizeof()`
- `strnlen` besser, da Grenzen überprüft werden

Beispiel:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define BUF_SIZE 255
char buf[BUF_SIZE];
...
printf ("Gebe Text ein: ");
if (NULL == fgets(buf, BUF_SIZE, stdin)) {
    exit(-1); // Error Handling required
}
printf ("Laenge der Eingabe: %zu Zeichen (incl. nl).\n",
        strnlen(buf, BUF_SIZE));
printf ("Es stehen maximal %lu Zeichen fuer den Text zur Verfuegung.\n",
        sizeof(buf)-1);
```

**Due to bounds checking
fgets and strnlen are
secure**

Vergleich zweier Strings: strcmp strcmp()

strcmp() und strcmp() vergleichen zwei Strings lexikografisch.

```
int strcmp(const char *s1, const char *s2, size_t n)
```

Rückgabewert < 0 wenn s1 ist kleiner als s2

Rückgabewert = 0 wenn s1 ist gleich s2

Rückgabewert > 0 wenn s1 ist größer als s2

Beispiel:

```
/* Passwort testen */
#define BUF_SIZE 50
char buf[BUF_SIZE];

printf("Enter password:");
scanf("%49s", buf);

if (strcmp(buf, "halligalli", BUF_SIZE)) {
printf("Wrong Password!");
}
```

Wie beseitigt man
Magic Number 49?

Schutz vor
overflow

Prüfe ob ein String Teil eines anderen ist: **strstr()** oder **strnstr()**

strstr() gibt einen Zeiger auf das erste Vorkommen eines Strings **str2** in einem anderen String **str1** zurück. Kommt **str2** nicht in **str1** vor, wird ein NULL-Pointer zurückgegeben.

strnstr() nicht in allen Unix Variante vorhanden.

```
char *strstr(char *str1, char *str2)
```

Beispiel:

```
/* Es wird ausgegeben: "toi sagt toitoitoi" */  
char  pStr;  
pStr = strstr("Tolstoi sagt toitoitoi", "toi")  
if( pStr )  
    printf("%s", pStr);  
....
```

Kopieren eines Strings: strncpy()

strncpy() kopiert die ersten num Zeichen des Strings src in den String dest. Ist src kürzer als dest wird der Rest von dest mit Nullen aufgefüllt. Ist src gleich lang oder länger als dest, wird **keine terminierende Null** angehängt! Der Zeiger auf dest wird zurückgegeben.

Es gibt auch eine Funktion strcpy (ohne Overflow schützt). Diese nach Möglichkeit nicht verwenden.

```
char * strncpy ( char * dest, char * src, int num );
```

Beispiel:

```
#define DESTLEN 6

strncpy( dest, src, DESTLEN );
if( dest[DESTLEN-1] != '\0' ){
    printf( "String ist zu lang!\n" );
} else {
    printf( "String ist: \"%s\", Laenge: %d\n", dest, strlen( dest ) );
}
```

Kopieren eines Strings `strncat()` und `strcat()`

`strncat()` hängt den Inhalt des Strings `src` und eine terminierende Null - jedoch maximal `n` Zeichen - an den String `dest` an. Das terminierende Nullzeichen von `dest` wird überschrieben. Der Zielstring muss groß genug sein – `strncat` legt keinen Speicher an. Der Zeiger auf `dest` wird zurückgegeben.

```
char *strncat(char *dest, const char *src, size_t n);
```

Beispiel:

```
#define BUF_SIZE 50
char s1[BUF_SIZE] = "Das ist das Haus ";
char s2[] = "vom Nikolaus!";
strncat(s1,s2, BUF_SIZE- strlen(s1)-1); /* Ziel ? Quelle */
printf("%s",s1);
```

Formatiertes Schreiben in einen String: `snprintf()` (und `sprintf`)

Analog zur formatierten Terminalausgabe mit `printf()` kann mit `snprintf()` formatiert in Strings geschrieben werden.

```
int snprintf(char *str, size_t size, const char Format, ....)
```

Formatbeschreiber analog zu `printf`:

<code>%d</code>	für Integer-Typen (Dezimalschreibweise)
<code>%x</code>	für Integer-Typen (Hexadezimalschreibweise)
<code>%c</code>	für char
<code>%s</code>	für Strings
<code>%f</code>	für float
<code>%lf</code>	für double

Beispiel:

```
/* Schreiben in einen String */
#define MAX_STR_SIZE
char String[MAX_STR_SIZE];

....

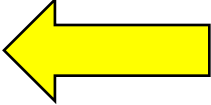
sprintf(String, MAX_STR_SIZE, "i=%2d quad=%5d \n", 6, 36);
....
```

Ausblick: Bounds-checking interfaces

- Standardisiert in C11 Annex K: Optional!
- Alternative Bibliotheksfunktionen:
 - Sicherer, kein Pufferüberlauf.
 - Bieten damit Schutz vor Angriffen.
- Erweiterung:
 - Prüfung, ob Ausgangspuffer groß genug ist für das Ergebnis.
 - Rückgabe eines Fehlerkodes, falls nicht erfolgreich.
 - String wird immer mit '\0' abgeschlossen.
 - API hat auf eine einfache Portierung geachtet.

Kapitel 4: Programmiersprache C - Fortgeschrittene Themen

Gliederung

- Adressen und Zeiger
- Felder
- Strings
- Strukturen 
- Dynamische Speicherverwaltung
- Zeigerarithmetik
- Selbstdefinierte Datentypen
- Zusammenfassung

Strukturen

Definition:

Eine **Struktur** ist eine Menge von verschiedenartigen Variablen, die einen inhaltlich zusammengehörigen Datensatz beschreiben.

Beispiel:

```
/* Deklaration */  
struct Student {  
    char    Name[25];  
    long int MatrikelNr;  
    char    Studiengang[3];  
};  
/* Definition */  
struct Student Stud1;  
  
/* Wertzuweisung */  
Stud1.MatrikelNr = 125716;
```

Die Struktur-Deklaration ist lediglich eine Beschreibung !
Dabei wird kein Speicherplatz reserviert.

Erst durch die Definition wird für die Struktur Speicherplatz
angelegt.

ÜBUNG: Struktur deklarieren, definieren und initialisieren

Deklarieren Sie eine Struktur „datum“. Diese soll 3 Integerwerte für den Tag, den Monat und das Jahr enthalten sowie ein Textfeld von 50 Zeichen für Stichworte (Memo).

Definieren Sie 2 Termine vom Typ datum und initialisieren Sie diese wie folgt:

termin1 : 31.12.2016, "Party ab 20:00."

termin2 : 10.04.2016, "Klausurtraining GS"

Lösung:

```
struct datum {
    int tag, monat;
    int jahr;
    char memo[50];
};

struct datum termin1, termin2;

termin1.tag = 31;
termin1.monat = 12;
termin1.jahr = 2016;
strncpy(termin1.memo, sizeof(memo), "Party ab 20:00.");

termin2.tag = 10;
termin2.monat = 4;
termin2.jahr = 2016;
strncpy(termin2.memo, sizeof(memo), "Klausurtraining GS");
```

Strukturen: Deklaration, Definition, Initialisierung

➤ Deklaration und Definition in einem Schritt

```
struct Student {  
    char    Name[25];  
    long int MatrikelNr;  
    char    Studiengang[3];  
} Std1, Std2, Std3;
```

➤ Definition und Initialisierung in einem Schritt

```
struct Student Stud1 = {"Meier", 125716, "TI"};  
  
struct Student Stud2 = {"Gates", 125609, "TI"};
```

Felder von Strukturen

➤ Definition von 60 Datenstrukturen vom Typ Student

```
struct Student Std[60];
```

➤ Initialisierung einer Datenstruktur des Felds

```
strncpy( Std[7].Name , sizeof(Std[7].Name)-1, "Meier");  
Std[7].MatrikelNr  = 125716;  
strncpy(Std[7].Studiengang, sizeof(Std[7].Studiengang),  
        "TI");
```

➤ Definition und Initialisierung von 3 Datenstrukturen vom Typ Student

```
struct Student Std[3] = {  
    {"Meier",  125716, "TI"},  
    {"Meiser", 125367, "TI"},  
    {"Gabski", 123362, "TI"}};
```

Achtung: Dass der Speicherplatz für den zugewiesen String groß genug ist, muss der Programmierer sicher stellen.

Übung: Felder von Strukturen definieren und initialisieren

Definieren Sie ein Feld von 20 Strukturen (Typ datum, s.o.) und initialisieren Sie die ersten Strukturen mit

- a) Definition und Initialisierung in einem Schritt.
- b) Definition und Initialisierung in getrennten Schritten.

Übung: Felder von Strukturen definieren und initialisieren (Fortsetzung)

Lösung: Definition und Initialisierung in einem Schritt

```
struct datum {  
    int tag;  
    int monat;  
    int jahr;  
    char memo[50];  
};  
  
struct datum f[20] =  
    { { 31, 12, 2016, "Party ab 20:00." } ,  
      { 10,  4, 2016, "Klausurtraining GS" }  
    };
```

Übung: Felder von Strukturen definieren und initialisieren (Fortsetzung)

Lösung: Definition und Initialisierung in getrennten Schritten

```
struct datum {
    int tag, monat;
    int jahr;
    char memo[50];
} f[20];

f[0].tag = 31;
f[0].monat = 12;
f[0].jahr = 2016;
strncpy(f[0].memo, sizeof(f[0].memo)-1, "Party ab 20:00.");

f[1].tag = 10;
f[1].monat = 4;
f[1].jahr = 2005;
strcpy(f[1].memo, "Klausurtraining GS");
```

Zeiger und Strukturen

➤ Deklaration der Struktur Student

```
struct Student {  
    char    Name[25];  
    long int MatrikelNr;  
    char    Studiengang[3];  
} ;
```

➤ Definition der Variablen Std_1

```
struct Student Std_1;
```

➤ Definition eines Zeigers auf eine Variable von Typ struct Student und Initialisierung

```
struct Student *pStd = &Std_1;
```

➤ Zugriff auf Strukturelemente über den Zeiger

```
strcpy( (*pStd).Name, "Smith");  
(*pStd).MatrikelNr = 125616;
```

... oder kürzer (und üblich)...

```
strcpy( pStd->Name, "Smith");  
pStd->MatrikelNr = 125616;
```

***v->b ist eine alternative
Notation für (*v) . b***

Strukturen und Wertzuweisungen

- Man kann den Wert einer Strukturvariablen einer anderen Strukturvariablen vom selben Typ zuweisen.

```
struct Student {  
    char    name[25];  
    long int matrikelNr;  
    char    studiengang[3];  
} std1, std2, *pstd;
```

```
std1 = std2;
```

```
pstd = &std1;
```

```
std2 = *pstd;
```


Strukturen und Parameterübergabe

➤ Call by value Übergabe

```
loben(struct Student s);
```

```
loben(studi)
```

- Kopie der Daten wird (auf den Stack) abgelegt
- Original kann nicht verändert werden.

➤ Call by reference Übergabe

```
loben(struct Student *s);
```

```
loben(&studi)
```

- : Referenz (Adresse) auf Daten wird übergeben.
- Original kann verändert werden.
- In der Regel schneller, aber ...

```
struct Student {  
    char name[50];  
    int matrikelnummer;  
} studi;
```

Für kleine Strukturen gut.
Absicherung der
Originaldaten.

Für zeitkritischen
Anwendungen bzw. bei
Speicherknappheit geeignet
(wenn Strukturen groß sind).

Strukturen und Rückgabewerte von Funktionen

- Funktionen können **Zeiger auf eine Struktur** oder **Kopie einer Struktur** zurückgeben.
- **Achtung: Rückgabe von Zeigern auf lokale Strukturen einer Funktion oder per Kopie übergebene Struktur (call by value) ist verboten – führt zu Fehlern !!**

Warum?

```
struct Student ausgezeichnet( void ){  
    struct Student std;  
  
    return std;  
}
```

```
struct Student* ausgezeichnet( struct Student* pstd ){  
  
    return pstd;  
}
```

```
struct Student ausgezeichnet( struct Student* pstd ){  
  
    return *pstd;  
}
```

Strukturen und Rückgabewerte von Funktionen (Fortsetzung)

- **Achtung: Rückgabe von Zeigern auf lokale Strukturen einer Funktion oder per Kopie übergebene Struktur (call by value) ist verboten – führt zu Fehlern !!**

```
struct Student* ausgezeichnet( void ){  
    struct Student std;  
  
    return &std;  
}
```

Fehler – nicht erlaubt.

```
struct Student* ausgezeichnet( struct Student std ){  
  
    return &std;  
}
```

Fehler – nicht erlaubt.

Diese Fehler erkennt der Compiler in der Regel nicht.

Ablage von Strukturen im Speicher

- Elemente werden im Speicher hintereinander abgelegt (meistens)
- Aus Performancegründen werden sie meistens an Maschinenwordgrenzen ausgerichtet (z. B. Pentium: 4 Byte Grenzen)

Beispiel:

```
/* Deklaration */
struct Student {
    char      Name[25];          /* 25 Bytes */
    long int  MatrikelNr;        /* 4 Bytes */
    char      Studiengang[3];    /* 3 Bytes */
};

printf( "Groesse der Struktur: %d\n", sizeof( struct Student ) );

printf( "Adr Name:           %p\n", (void*)& Std_1.Name);
printf( "Adr MatrikelNr:     %p\n", (void*)& Std_1.MatrikelNr);
printf( "Adr Studiengang:    %p\n", (void*)& Std_1.Studiengang );
```

Ausgabe: Groesse der Struktur: 36
 Adr Name: 2000
 Adr MatrikelNr: 201c
 Adr Studiengang: 2020

Problematisch, wenn man die Struktur nutzen möchte, um z. B. die Anordnung der Daten innerhalb einer Datei oder eines Telegramms zu beschreiben

Verwendung einer Struktur zur Definition eines Telegramm-Kopfes

```
int my_read( void *puffer, int size );

typedef struct tlx {
    char      absender[6];
    char      empfaenger[6];
    char      typ;
    short int laenge;
    int       pruefsumme;
} __attribute__((__packed__)) TelegrammKopf;

int main(void) {
    int rc;
    TelegrammKopf header;

    rc = my_read( &header, sizeof( header ) );
    return 0;
}
```

Compilerdirektive –
Anweisung an den
Compiler,
compilerspezifisch
Alternative: #pragma

Strukturgröße und Felder

```
typedef struct s1 {  
    int      info;  
    char      absender[5];  
} k1 ;
```

```
typedef struct s2 {  
    int      info;  
    char      absender[5];  
} __attribute__((__packed__)) k2;
```

Dann liefert:	<code>sizeof(struct s1)</code>	12
	<code>sizeof(struct s2)</code>	9

Grund: Feldelemente liegen lückenfrei hintereinander und der effiziente Zugriff auf Elemente soll sichergestellt werden.

Diskussion:

- char Feld
- Warum müssen beim Vergleich zweier Strukturen die Elemente paarweise verglichen werden?

Bitfelder

- Bitfelder dienen dem Zugriff auf einzelne Bits.

```
struct Name
{
    DATENTYP1    Element1: Bitzahl;
    DATENTYP2    Element2: Bitzahl;
    ....
} bitVar1, ...;
```

- Anwendung:

- Speicherung von Daten bei optimaler Ausnutzung des Speichers
- Zugriff auf einzelne Bits eines I/O-Bausteins

- Kommentare:

- Datentyp immer unsigned
- Die einzelnen Elemente kann man wie unsigned int Werte ansehen, deren Breite durch Bitzahl definiert ist.

Bitfelder: Beispiel

```
#define UCHAR unsigned char
// Peripherie-Baustein nachbilden
struct Timer{
    struct {
        UCHAR enable      : 1;
        UCHAR intEnable: 1;
        UCHAR intPeriod: 1;
        UCHAR              : 1;
        UCHAR preScale   : 4;
    } reg0;
    UCHAR reg1;
    struct {
        UCHAR pending    : 1;
        UCHAR            : 6;
        UCHAR error      : 1;
    } reg2;
};
```

Memory mapped IO

```
struct Timer* pTimer = (struct Timer*)0x0100;
// Zugriff auf Register des Bausteins
pTimer->reg0.enable = 1;
error = pTimer->reg2.error;
```

Aufbau eines fiktiven Peripherie-Bausteins

Register	Bit	Funktion
0	0	0 = Baustein gesperrt
		1 = Baustein freigegeben
	1	0 = Interrupt gesperrt
		1 = Interrupt freigegeben
	2	0 = einmaliger Interrupt
		1 = periodischer Interrupt
	3	nicht belegt
	4..7	Vorteiler
1	0..7	Zähler
2	0	0 = kein Interrupt anstehend
		1 = Interrupt anstehend
	1..6	nicht belegt
	7	Bausteinfehler

Anordnung (Reihenfolge = im Bitfeld teilweise Systemabhängig)

Besonderheiten bei der Verwendung von Bitfeldern

- Von einem Bitfeldelement kann keine Adresse gebildet werden (dies wäre die Adresse eines bestimmten Bits im Speicher!).
- Es sind keine Zeiger oder Referenzen auf einzelne Bitfeldelemente erlaubt. Zeiger auf das Bitfeld selbst sind erlaubt.
- Von Bitfeldelementen kann kein Feld definiert werden, wohl aber vom Bitfeld selbst.
- Zugriffe auf Bitfelder sind langsamer als der Zugriff auf 'normale' Variablen, da intern Schiebe- und Maskierungsoperationen durchgeführt werden müssen.
- Die Reihenfolge der einzelnen Bits in einem Bitfeld ist nicht im ANSI C festgeschrieben!
- Es ist nicht definiert dass das 1. Bit im Bitfeld auch das niederwertigste Bit im Speicher ist.

Union

Union ermöglicht den Zugriff auf den selben Speicherbereich mit unterschiedlichen Datentypen.

Beispiel:

```
union test1 {  
    char a;  
    int b;  
    double c;  
} v;
```

v speichert entweder eine Variable vom Typ char, oder eine vom Typ int, oder eine vom Typ double. Alle drei teilen sich den gleichen Speicherbereich – hier 8 Byte.

Über v.a greift man auf eine Variable vom Typ char, über v.b auf eine Variable vom Typ int und über v.c auf eine Variable vom Typ double.

Greift man zum Beispiel auf v.a zu, muss vorher eine char Wert über v.a in die Union geschrieben worden sein.

Beispiel: Union

```
struct TlxHeader{  
    ...  
    int type;  
} __attribute__((__packed__));
```

Im Telegrammkopf
wird der Typ des
Telegrammrumpfs
festgelegt.

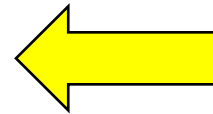
```
union TlxBody{  
    struct TlxCmd{  
        int cmd;  
        int param;  
    } __attribute__((__packed__)) TlxCmd;  
  
    struct TlxStatus{  
        int code;  
        char message[20];  
    } __attribute__((__packed__)) TlxStatus;  
  
    char data[1];  
};
```

```
struct Tlx {  
    struct TlxHeader hdr;  
    union TlxBody body;  
} __attribute__((__packed__));
```

Kapitel 4: Programmiersprache C - Fortgeschrittene Themen

Gliederung

- Adressen und Zeiger
- Felder
- Strings
- Strukturen
- Dynamische Speicherverwaltung
- Zeigerarithmetik
- Selbstdefinierte Datentypen
- Zusammenfassung



Allokierung von Speicher zur Laufzeit

Situation: Bisher wurde der benötigte Speicher immer zur Compilezeit (statisch) festgelegt, z.B.

```
int    i, j, k;
```

```
double Mat[3][4];
```

In vielen Aufgabenstellungen ist aber die Größe des benötigten Speichers zur Compilezeit gar nicht bekannt.

Beispiele:

- Unterprogramme zur Matrixberechnung
- Texteditor

Vorgehensweise: In diesen Fällen wird zur Laufzeit Speicherplatz vom Betriebssystem angefordert (allokiert).

Allokierung von Speicher zur Laufzeit (Fortsetzung)

Für die Allokierung des Speichers wird die C Bibliotheksfunktion

```
void * malloc(int n);
```

verwendet. `malloc()` allokiert `n` Bytes und gibt einen Zeiger auf den allokierten Speicherbereich zurück. Ist die Allokierung nicht möglich, wird `NULL` zurückgegeben.

Über `sizeof` wird aus dem Typ der Variablen, zu dem der Speicher erzeugt werden soll, die Anzahl der Bytes berechnet.

Über die cast Operation wird der Typ des Zeigers, den `malloc` liefert, auf den Zieltyp abgebildet.

Allokierung von Speicher zur Laufzeit (Fortsetzung)

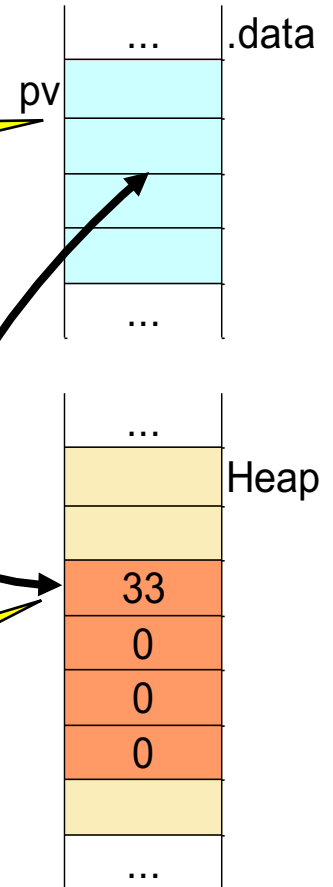
Beispiele: Über malloc wird der Speicher für eine int Variable zur Laufzeit angefordert.

```
#include <malloc.h>
int *pv = NULL;
...
pv = (int *) malloc (sizeof(int));
*pv = 33;
```

Wiederholung: Initialisieren Sie Zeiger stets mit NULL: Mehr Laufzeitfehler werden erkannt.

Zeiger pv: wurde zur Compilezeit angelegt

Speicherplatz für eine int Variable: wurde zur Laufzeit mit malloc angelegt



Anmerkung: Das Beispiel enthält keine Fehlerbehandlung für den Fall, dass malloc keinen Speicher liefern kann.

Allokierung von Speicher zur Laufzeit (Fortsetzung)

Beispiele: Über malloc wird der Speicher für ein Feld von 100 double Variablen zur Laufzeit angefordert.

```
#include <malloc.h>
double *pDFeld =NULL;
...
pDFeld = (double *) malloc (100 * sizeof(double));
pDFeld[0] = 33;
pDFeld[99] = 33334;
```


Allokierung von Speicher zur Laufzeit (Fortsetzung)

Beispiele: Über malloc wird der Speicher für eine Variable vom Typ struct Student zur Laufzeit angefordert.

```
#include <malloc.h>

struct Student {
    char        Name[25];
    long int    MatrikelNr;
    char        Studiengang[3];
} *pstd = NULL;

...

pstd = (struct Student *)
        malloc (sizeof(struct Student));
pstd->MatrikelNr = 123456;
```

Allokierung von Speicher zur Laufzeit (Fortsetzung)

Zur Freigabe von nicht mehr benötigtem Speicherplatz wird die C Bibliotheksfunktion

```
void free(void *)
```

verwendet.

Wichtig ist,

- dass es sich bei dem übergebenen Zeiger tatsächlich um einen Zeiger handelt, der auf zur Laufzeit allokierten Speicher zeigt. Ansonsten stürzt die Anwendung ab.
- Anschließend darf der Speicher nicht mehr verwendet werden.
- Für die ordnungsgemäße Speicherfreigabe hat der Programmierer zu sorgen (Ursache vieler Softwarefehler).

Allokierung von Speicher zur Laufzeit (Fortsetzung)

Beispiele für den Einsatz der Funktion free: Über malloc wird der Speicher für ein Feld von 100 int Variablen zur Laufzeit angefordert und später über free wieder frei gegeben.

```
#include <malloc.h>
int *pFeld = NULL;
...
pFeld = (int *) malloc (100 * sizeof(int));
...
free(pFeld);
pFeld = NULL;
...
```

**In einem C Programm gibt es keinen Speicherplatz mit der Adresse NULL (= 0).
In C hat die Adresse NULL die Bedeutung: Zeiger auf keinen Speicherplatz - kein sinnvoller Speicherplatz**

Guter Programmierstil: Der Anwender weist einem Zeiger die Adresse NULL zu, wenn der Zeiger auf keinen Speicherplatz zeigt.

ÜBUNG: Dynamische Datenstrukturen, einfach verkettete Liste

Auf der Basis folgender Struktur wird eine dynamische Liste von int Variablen verwaltet:

```
struct ListElem {  
    int elem;  
    struct ListElem *next;  
};
```

Folgende Funktionen werden benötigt:

```
void AddElem(int v);
```

```
void DelElem(int v);
```

```
void PrintList();
```

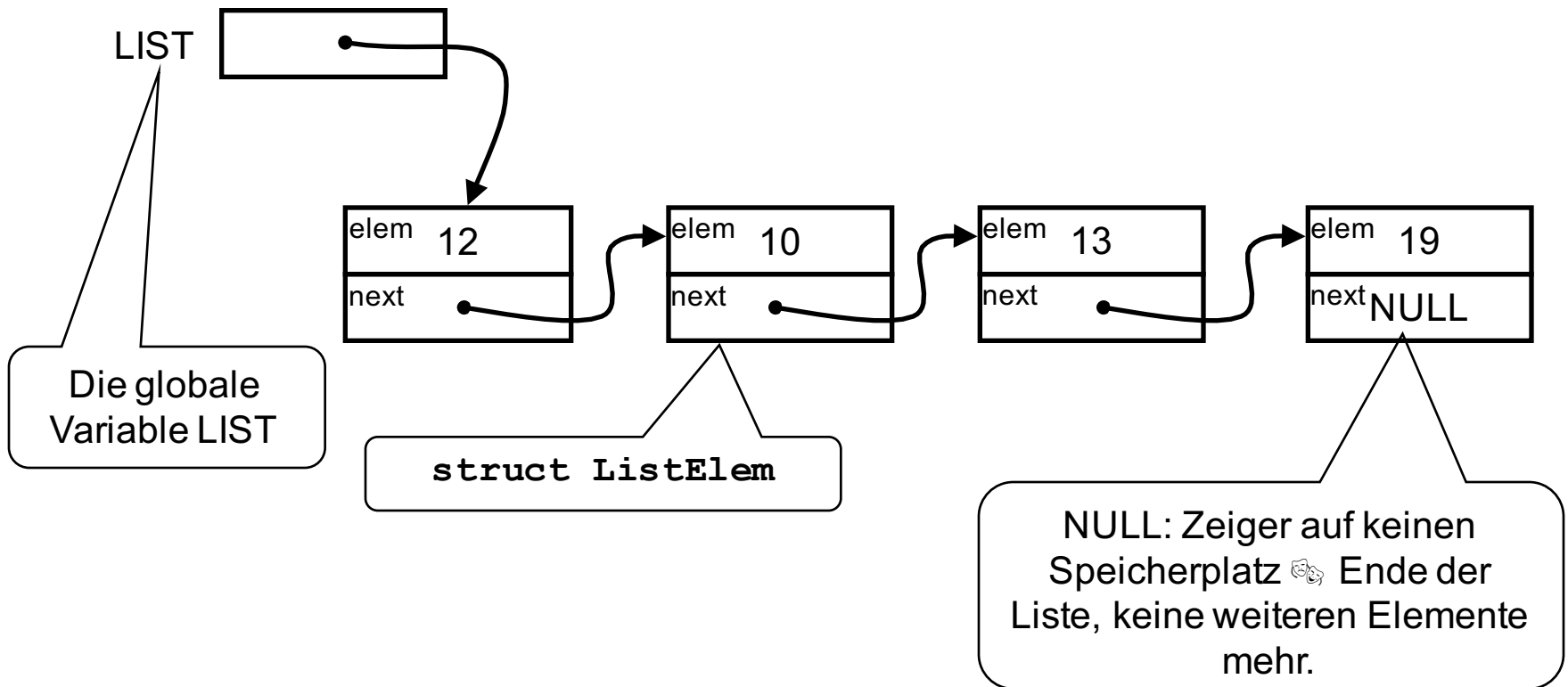
Zur Vereinfachung zeigt die globale Variable

```
struct ListElem LIST;
```

auf die Liste. Die Funktionen greifen auf LIST zu.

ÜBUNG: Dynamische Datenstrukturen, einfach verkettete Liste (Fortsetzung)

Aufbau einer einfach verketteten Liste



ÜBUNG: Dynamische Datenstrukturen, einfach verkettete Liste (Fortsetzung)

Aufbau einer einfach verketteten Liste

LIST

NULL

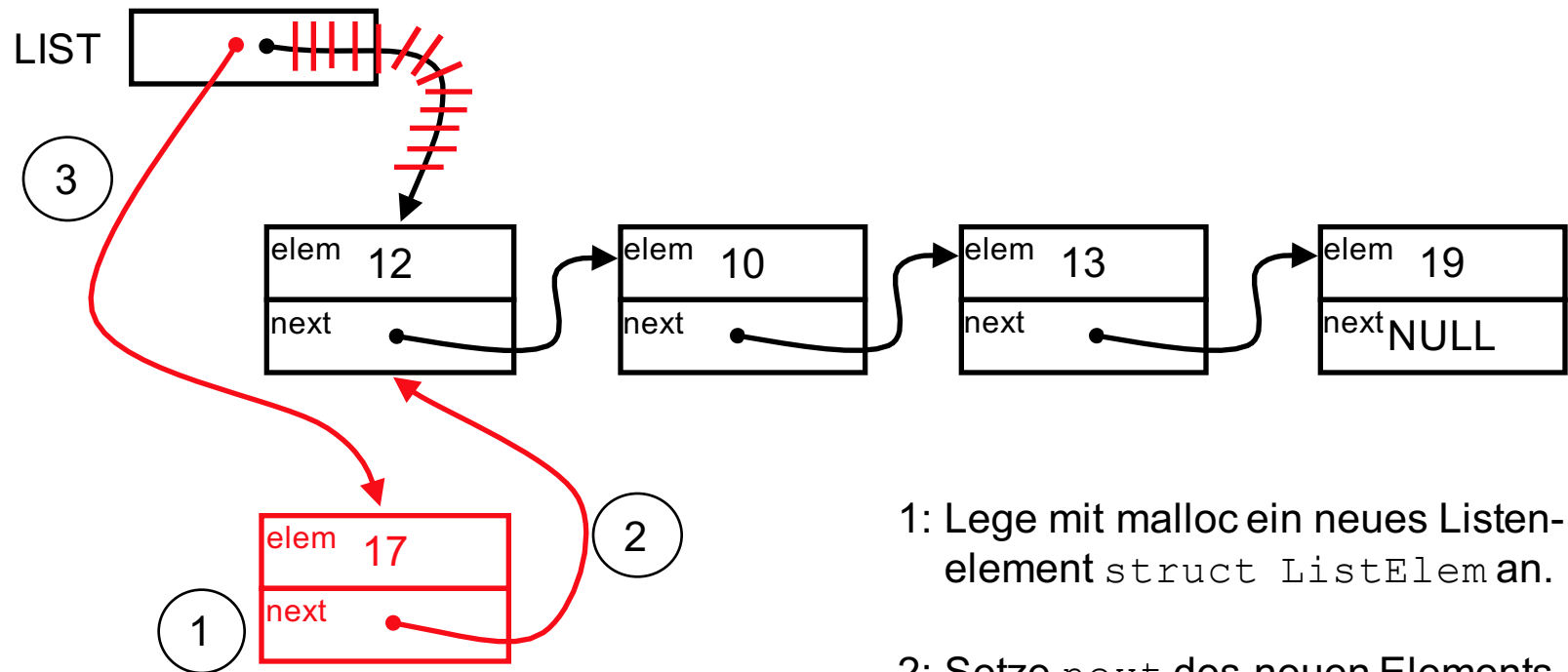
Was bedeutet diese Belegung von LIST ?

Antwort: Die Liste LIST enthält kein Element – ist leer.

Somit wird die globale Variable LIST mit NULL initialisiert.

ÜBUNG: Dynamische Datenstrukturen, einfach verkettete Liste (Fortsetzung)

Füge ein Element mit dem Wert 17 am Anfang der Liste LIST ein.



- 1: Lege mit malloc ein neues Listenelement `struct ListElem an`.
- 2: Setze `next` des neuen Elements auf den Wert von `LIST`.
- 3: Setze `LIST` auf die Adresse des neuen Elements.

ÜBUNG: Dynamische Datenstrukturen, einfach verkettete Liste (Fortsetzung)

```
#include <malloc.h>

struct ListElem {
    int elem;
    struct ListElem *next;
};

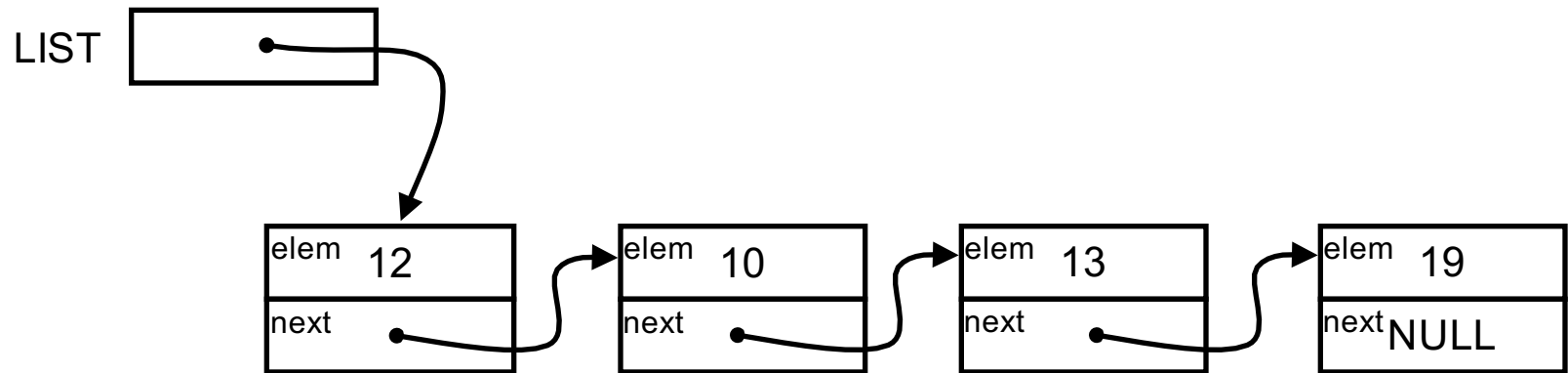
struct ListElem *LIST = NULL;

void AddElem(int v)
/* Fügt am Anfang der Liste ein Element ein */
{
    struct ListElem * NewElem;

    NewElem = (struct ListElem *) malloc(sizeof(struct ListElem)); /* 1 */
    if (NULL == NewElem) ... Error Handling
    NewElem->elem = v; /* 1 */
    NewElem->next = LIST; /* 2 */
    LIST = NewElem; /* 3 */
}
```


ÜBUNG: Dynamische Datenstrukturen, einfach verkettete Liste (Fortsetzung)

Durchlaufe die Liste und drucke die Elemente



```
struct ListElem * p = LIST;
...
while (p != NULL) {
    ... /* Aktion mit den Listenelement
*/
    p = p->next;
}
...
```

ÜBUNG: Dynamische Datenstrukturen, einfach verkettete Liste (Fortsetzung)

```
#include <malloc.h>

struct ListElem {
    int elem;
    struct ListElem *next;
};

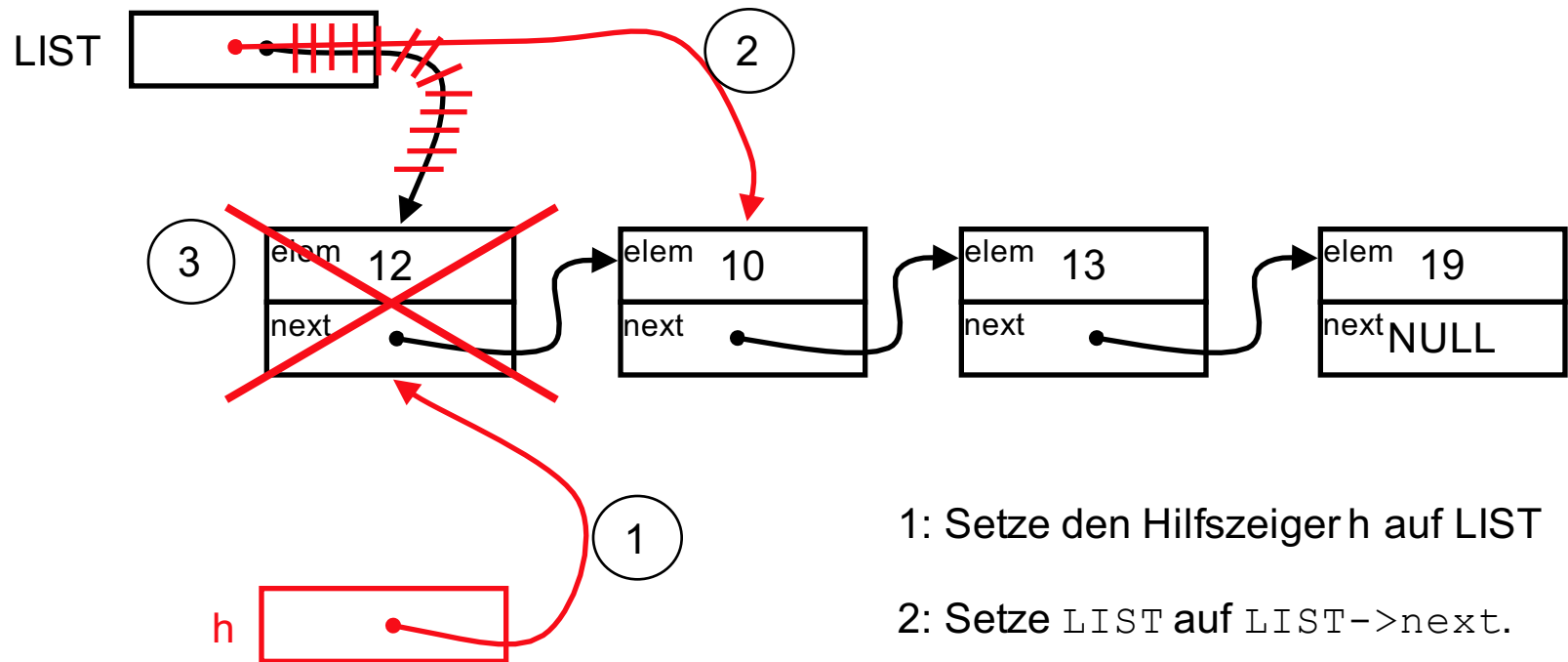
struct ListElem *LIST = NULL;

void PrintList()
/* Drucke eine Liste */
{
    struct ListElem * p = LIST;
    printf("[");

    while (p != NULL) {
        printf("%d ", p->elem);
        p = p->next;
    }
    printf("]\n");
    return;
}
```

ÜBUNG: Dynamische Datenstrukturen, einfach verkettete Liste (Fortsetzung)

Lösche das erste Element aus der Liste (wenn `LIST != NULL` ist)



1: Setze den Hilfszeiger `h` auf `LIST`

2: Setze `LIST` auf `LIST->next`.

3: `free(h)`

ÜBUNG: Dynamische Datenstrukturen, einfach verkettete Liste (Fortsetzung)

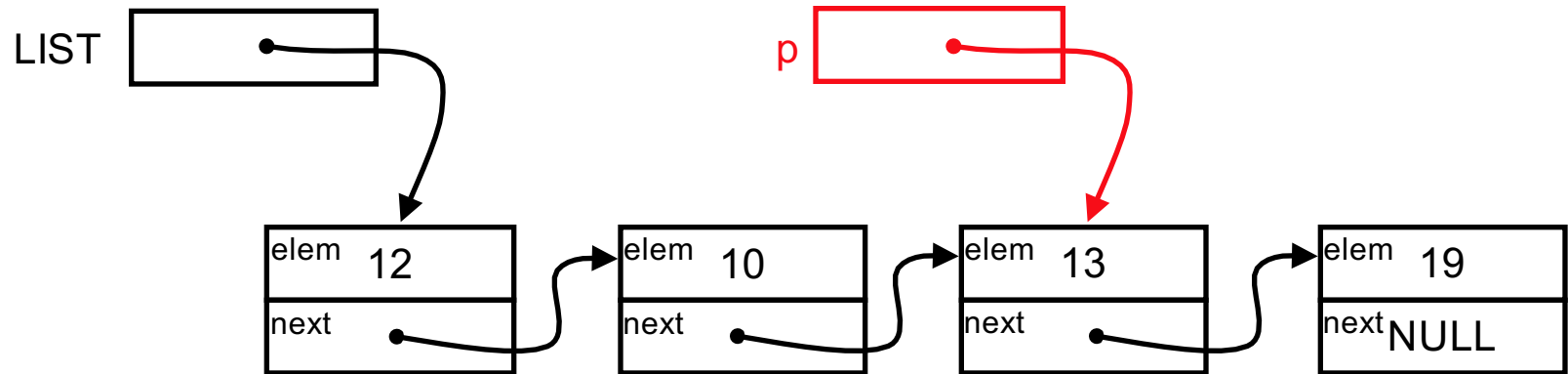
```
void DelElem(void)
/* Loescht das erste Element aus LIST. */
{
    struct ListElem * h = NULL;

    if (LIST == NULL) return;

    h = LIST;
    LIST = LIST->next;
    free(h);
    return;
}
```

ÜBUNG: Dynamische Datenstrukturen, einfach verkettete Liste (Fortsetzung)

Lösche das i-te Element aus der Liste (p zeigt auf dieses Element)

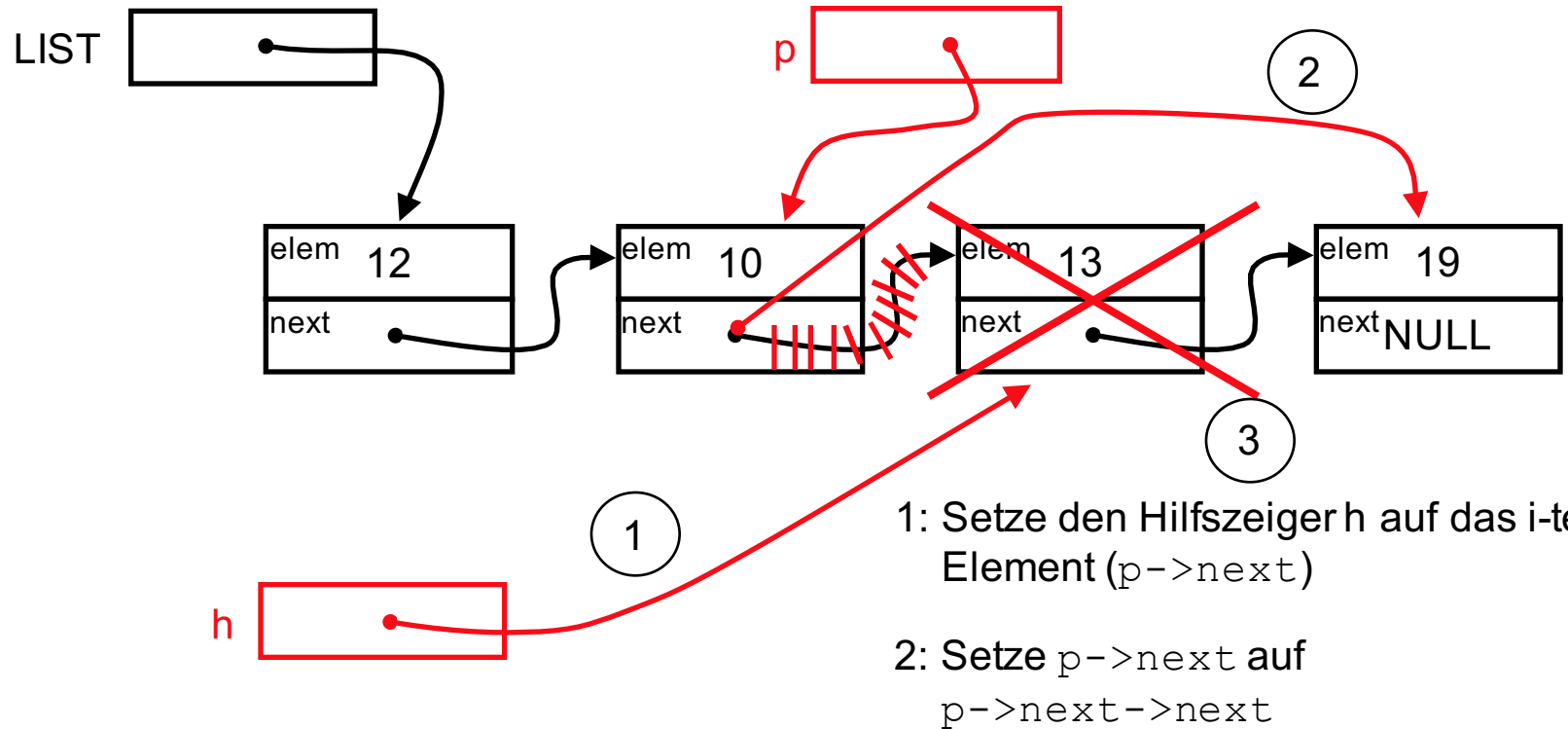


Problem: Auf den Vorgänger von p kann nicht mehr zugegriffen werden.
Somit kein dessen next Zeiger nicht modifiziert werden.

Lösung: Wenn die Liste mindestens ein Element enthält, zeigt p auf den
den Vorgänger und $p \rightarrow \text{next}$ wird gelöscht.

ÜBUNG: Dynamische Datenstrukturen, einfach verkettete Liste (Fortsetzung)

Lösche das i -te Element aus der Liste (**p zeigt auf das $(i-1)$ -te Element**)



1: Setze den Hilfszeiger **h** auf das i -te Element ($p \rightarrow next$)

2: Setze $p \rightarrow next$ auf $p \rightarrow next \rightarrow next$

3: `free(h)`

ÜBUNG: Dynamische Datenstrukturen, einfach verkettete Liste (Fortsetzung)

```
void DelElem(int v)
/*  Loescht das erste Element aus LIST, dessen Element elem der Wert v hat.
*/
{
    struct ListElem * p = NULL;
    struct ListElem * h = NULL;

    if (LIST == NULL) return;
    if (LIST->elem == v) {
        h = LIST;
        LIST = LIST->next;
        free(h);
        return;
    }

    p = LIST;
    while (p->next != NULL) {
        if (p->next->elem == v) {
            h = p->next;
            p->next = p->next->next;
            free(h);
            return;
        }
        p = p->next;
    }
    return;
}
```

LIST ist nicht leer, und
das erste Element hat
nicht den Wert v.

Es gilt: $p \rightarrow \text{elem} \neq v$

ÜBUNG: Dynamische Datenstrukturen

Auf der Basis folgender Strukturen soll eine Studentenkartei aufgebaut werden.

```
struct Datum {  
    int Tag;  
    int Monat;  
    int Jahr;  
};  
  
struct Student {  
    char Name[20];  
    int MatrNr;  
    struct Datum Einschreibung;  
    struct Student *next;  
};
```

Folgende Funktionen werden benötigt:

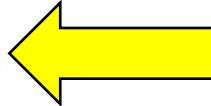
```
void AddStudent(char *Name, int MatrNr,  
                int Etag, int Emon, int Ejahr);  
  
void DelStudent(int MatrNr);  
  
void PrintStudents();
```


Dynamische Speicherverwaltung - Fazit

- Dynamische Speicherverwaltung ist komplex und kleinteilig
 - Erfordert große Sorgfalt:
 - Aller allozierter Speicher muss auch immer frei gegeben werden.
Auch im Fehlerfall.
- Gefahr der Zerstückelung. Schwer vorhersagbar.
- Verwendung nur wenn notwendig, z.B.:
 - Verwendung Strukturen mit dynamische Größe
 - Instanzen erzeugen
 - Speicherknappheit: Der gleiche Speicher muss für unterschiedliche Zwecke genutzt werden.
 -

Kapitel 4: Programmiersprache C - Fortgeschrittene Themen

Gliederung

- Adressen und Zeiger
- Felder
- Strings
- Strukturen
- Dynamische Speicherverwaltung
- Zeigerarithmetik 
- Selbstdefinierte Datentypen
- Zusammenfassung

Wiederholung: Felder und Zeiger

Zeiger und Felder sind eng miteinander verbunden – jede Operation auf Feldern kann auch mit Zeigern formuliert werden.

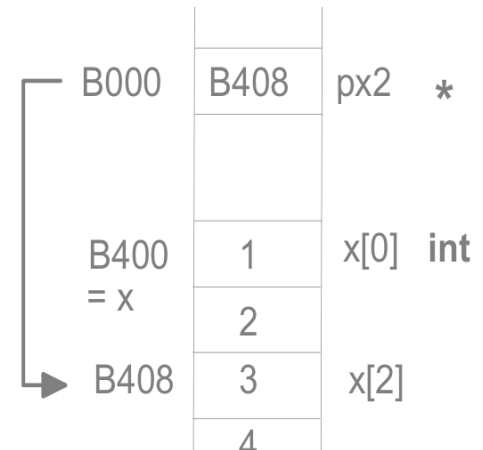
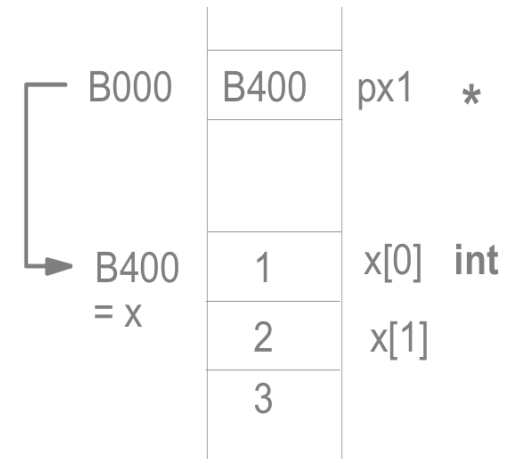
```
int x[] = {1, 2, 3, 4, 5, 6, 7};
int *px1 = NULL;
px1 = x;
/* Folgende Schreibweisen sind äquivalent */
printf("Adr. der 1.Zahl=%X", &x[0]);
printf("Adr. der 1.Zahl=%X", x);
printf("Adr. der 1.Zahl=%X", px1);
```

Der Feldname ohne Index liefert die Adresse des ersten Feldelements (Index 0).

```
int x[] = {1, 2, 3, 4, 5, 6, 7};
int *px2 = NULL;
px2 = &x[2];
printf("%d %d %d", *px2, px2[0], px2[3]);
```

Ein indizierter Zeiger liefert den indizierten Wert.

Ausgabe: 3 3 6



Zeigerarithmetik

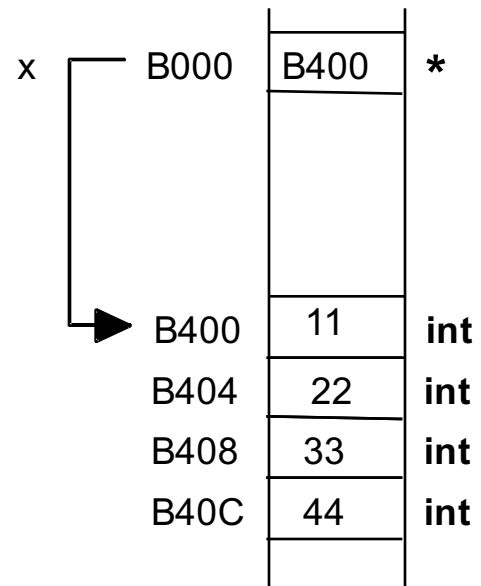
Prinzip: Zeiger auf Objekte können um die vereinbarten Typen erhöht bzw. erniedrigt werden.

```
int *x = {11, 22, 33, 44};
```

```
printf("x0=%d  x3=%d", *x, *(x+3));
```

```
/* äquivalent zu */
```

```
printf("x0=%d  x3=%d", x[0], x[3]);
```



Zeigerarithmetik (Fortsetzung)

```
struct datum {  
    int tag, monat, jahr;  
    char * memo;  
} test[12] = { { 22, 3, 2005, "GS VL" },  
               { 26, 3, 2005, "Ostereier anmalen " },  
               { 26, 3, 2005, "Ostereier suchen" },  
               { 31, 12, 2005, "Party" }  
};  
  
...  
struct datum *p;  
p = test;  
printf("test[3] = (%d,%d,%d,%s) \n",  
       (p+3)->tag,  
       test[3].monat,  
       (* (p+3)).jahr,  
       p[3].memo);
```

Ausgabe:

```
test[3] = (31,12,2005,Party)
```

Zeigerarithmetik (Fortsetzung)

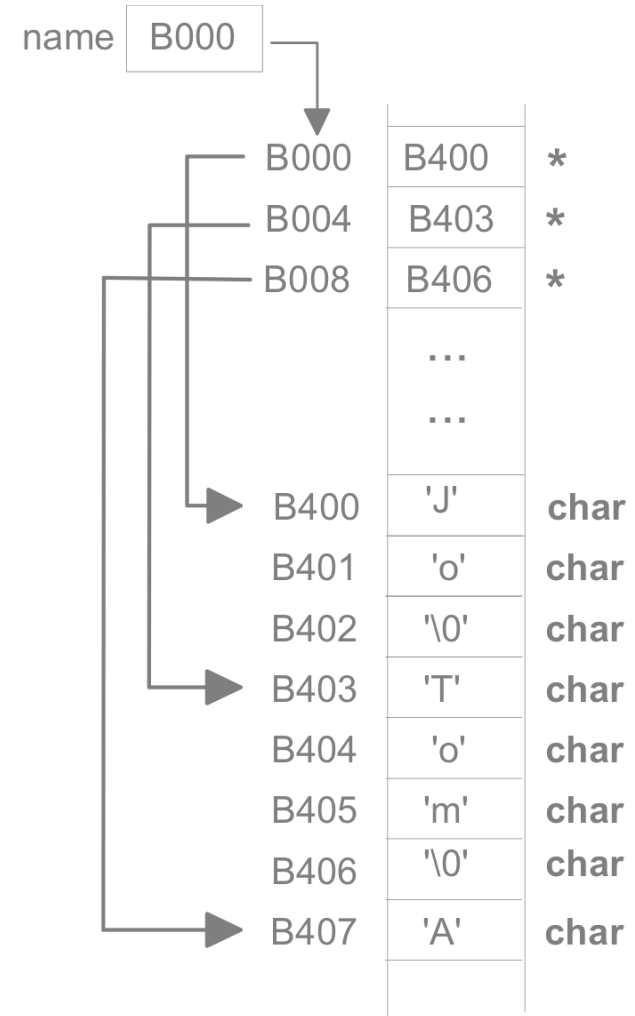
Beispiel

```
char *name[] = { "Jo"      , "Tom",
                 "Andreas", "Lou"
                 };
```

```
printf ("1.Name: %s, 3.Name: %s",
        *(name),      *(name+2) );
```

```
/* ist äquivalent zu */
```

```
printf ("1.Name: %s, 3.Name: %s",
        name[0], name[2] );
```



Vergleich: Zugriff per Zeiger ⇔ Zugriff über Index

```
struct Mitglied{  
    char name[100];  
    int beitrags;  
};
```

Zugriff per Zeiger

```
int psum( struct Mitglied *start, struct Mitglied* end ){  
    int s = 0;  
  
    while( start < end ){  
        s += (start++)->beitrags;  
    }  
    return s;  
}
```

```
struct Mitglied *mitglieder;  
  
...  
  
s = sum( mitglieder, mitglieder+cnt );
```

Performancemessung: Kein
deutlich messbarer Unterschied!
Daher: Zugriff per Index verwenden:
Besser lesbar. Einfacher auf Indexüberlauf
abzusichern.

ÜBUNG: Zeigerarithmetik (C-Puzzle, kein ernsthafter Code)

Was gibt das folgende Programm aus?

Zeichnen Sie zuvor die Memory-Map (Adressen s. Kommentar).

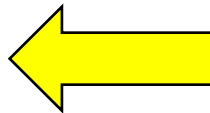
```
int a[] = {0,1,2,3,4};           /* a  beginne bei $1000 */
int *p[] = {a,a+1,a+2,a+3,a+4}; /* p  beginne bei $2000 */
int **pp = p;                    /* pp stehe hinter p    */

main() {
    printf("%d %d %d \n",          a,      a[1],      &a[2]      );
    printf("%d %d %d \n",          *p[0],    p[0],      &p[0]      );
    printf("%d %d %d \n",          *(p[1]+2),  p+3,      *p        );
    printf("%d %d %d \n",          *pp,      pp,      pp[0][4]    );
    printf("%d %d %d \n",          *( * (pp+1)+2), &pp,      **pp      );
}
```


Kapitel 4: Programmiersprache C - Fortgeschrittene Themen

Gliederung

- Adressen und Zeiger
- Felder
- Strings
- Strukturen
- Dynamische Speicherverwaltung
- Zeigerarithmetik
- Selbstdefinierte Datentypen
- Zusammenfassung



Selbstdefinierte Datentypen

In C lassen sich neue Datentypen definieren. Hierzu dient das Schlüsselwort

typedef

```
/* Definition */
```

```
typedef unsigned char  BYTE;  
typedef unsigned short WORD;  
typedef unsigned long  LONGWORD;
```

```
/* Anwendung */
```

```
BYTE b1,b2;  
WORD w[10];
```

```
/* Definition */
```

```
typedef char NAME[25];
```

```
/* Anwendung */
```

```
NAME n1;          /* äquivalent zu:  char n1[25] */
```

Selbstdefinierte Datentypen (Fortsetzung)

```
/* Deklaration eines struct Typen */
```

```
typedef struct {  
    char    Name[25];  
    long int MatrikelNr;  
    char    Studiengang[3];  
} Student, *pStudent;
```

```
/* Definition von 20 Strukturen des Typs "Student" */
```

```
Student Std[20];
```

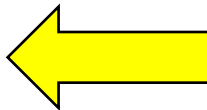
```
pStudent ZeigerAufStudentStruct = Std + 3;
```

```
strcpy( Std[0].Name, "Winter");
```

Kapitel 4: Programmiersprache C - Fortgeschrittene Themen

Gliederung

- Adressen und Zeiger
- Felder
- Strings
- Strukturen
- Dynamische Speicherverwaltung
- Zeigerarithmetik
- Selbstdefinierte Datentypen
- Zusammenfassung



Zusammenfassung

