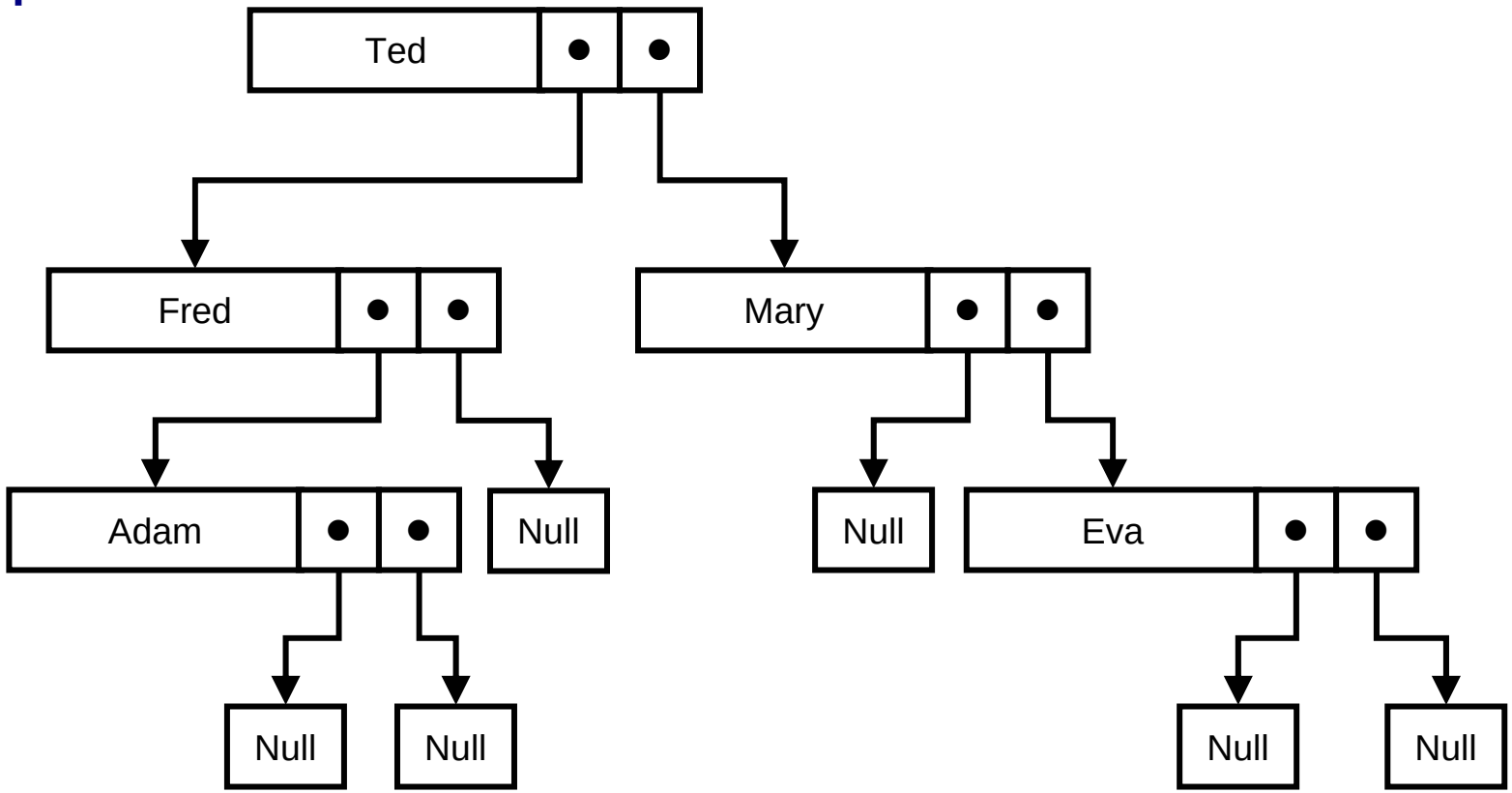
5. Zeiger und komplexe Datenstrukturen

5.1 Zeiger (pointer)

Ein Zeiger (pointer) ist ein Verweis auf ein anderes Datenobjekt.

Beispiel:

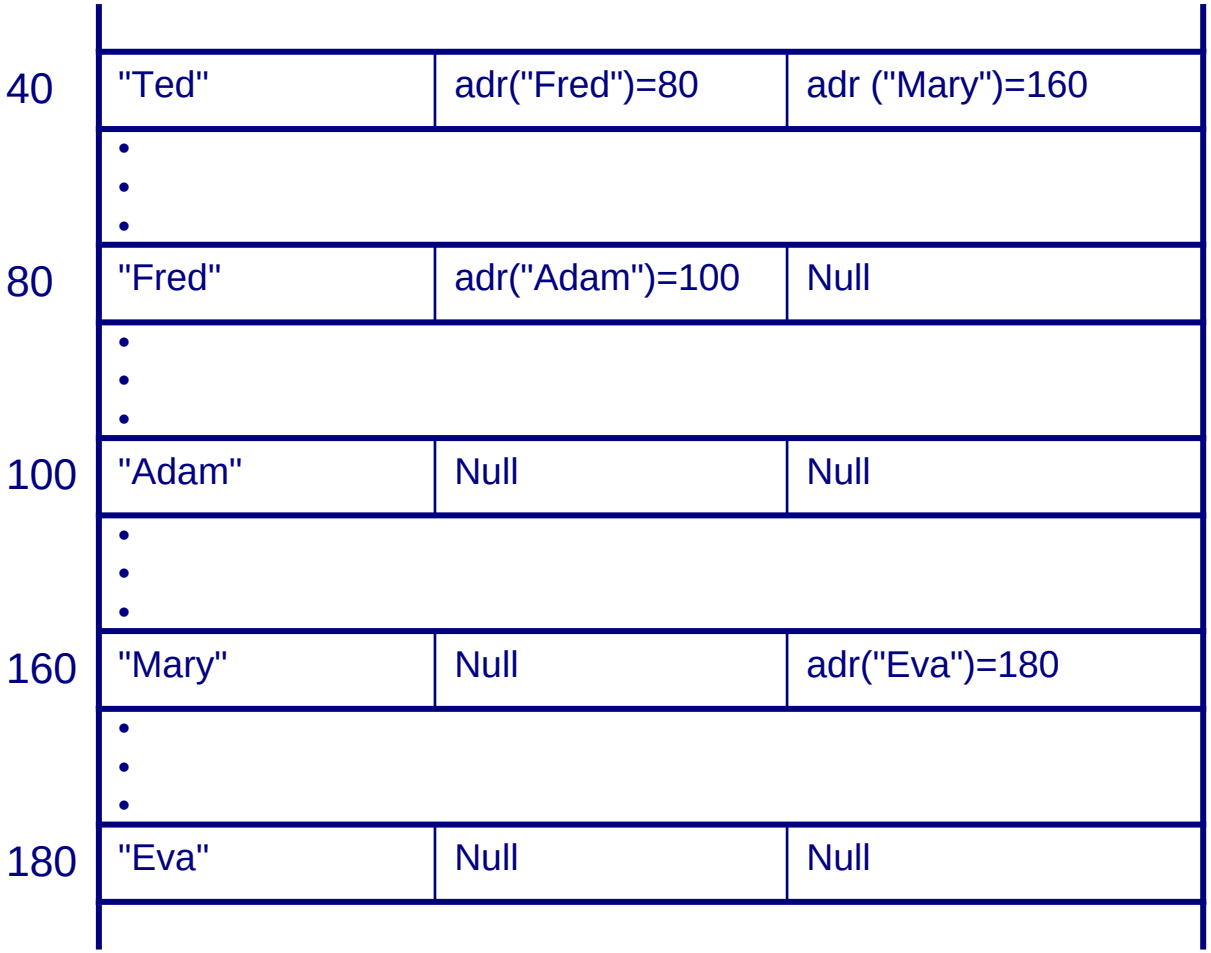


Implementierung von Zeigern (1)

Der Speicher des Computers ist in fortlaufend nummerierte Speicherzellen (Bytes) aufgeteilt, wobei die Nummer die Adresse der Speicherzelle darstellt.

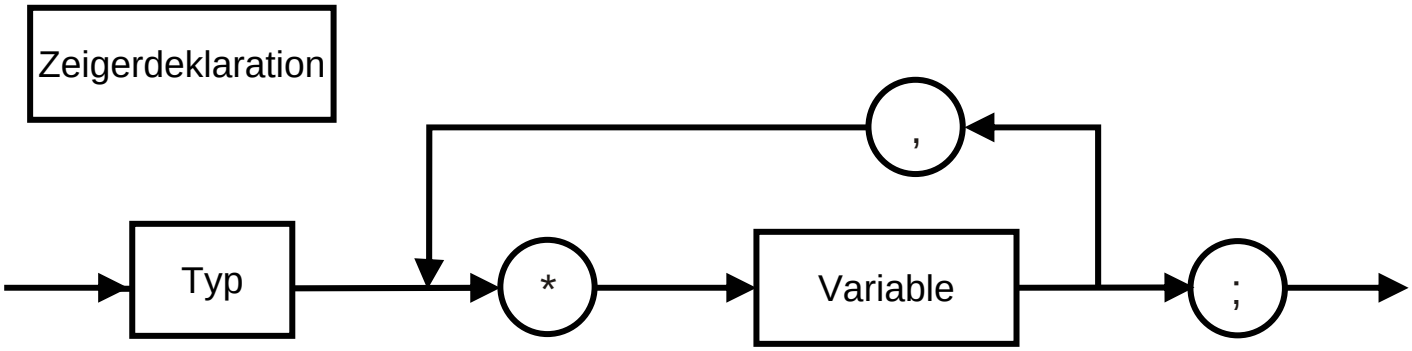
Ein Zeiger wird in C repräsentiert durch die Speicheradresse des Objekts, auf das verwiesen werden soll.

Implementierung von Zeigern (2)



Zeiger-Syntax in ANSI C (1)

Deklaration:



Beispiele:

```
int *a;

struct person {
   char name[20];
   struct person *vater, *mutter;
}
```

Zeiger-Syntax in ANSI C (2)

Werte, die ein Zeiger in C annehmen kann:

- positive Ganzzahl (Adresse im Speicher des Rechners)
- spezielle Adresse NULL (Zeiger, der nirgendwo hinzeigt). Ist in <stdio.h> definiert.

Merke:

Der Typ des Datenelements, auf das gezeigt werden kann (Bezugstyp), ist aus der Deklaration des Zeigertyps ersichtlich!

Mit der Deklaration "void *" kann ein unspezifischer (generischer) Zeiger (ohne Typ) deklariert werden. Solche Zeiger dürfen jedoch nicht selbst zum Zugriff verwendet werden, sondern können nur als Platzhalter für Argumente in Funktionen dienen.

Operationen auf Zeigern (1)

a) Wertzuweisung

Einem Zeiger kann der Wert eines anderen Zeigers zugewiesen werden.

Beispiel:

```
typedef int ganze_zahl;
typedef (ganze_zahl *) zeiger_auf_ganze_zahl;
ganze_zahl a;
zeiger_auf_ganze_zahl p1, p2;

... {

...

p2 = p1;
/* p2 zeigt nun auf dasselbe Objekt wie p1  */

...

}
```

Operationen auf Zeigern (2)

Es ist in C auch möglich, Zeigern absolute Speicheradressen zuzuweisen.

Beispiel:

```
p1 = (zeiger_auf_ganze_zahl)1501;
/* Typecast vermeidet Compiler-Warnung */
```

Merke:

Es ist gefährlich, absolute Speicheradressen zu verwenden, da die Einteilung des Speichers in der Regel nicht bekannt ist (Compiler-abhängig, Hardware-abhängig). Ein Programm ist dann in der Regel nicht mehr portabel. Absolute Speicheradressen werden in der Regel nur bei der Programmierung von Treibern oder innerhalb des Betriebssystems verwendet.

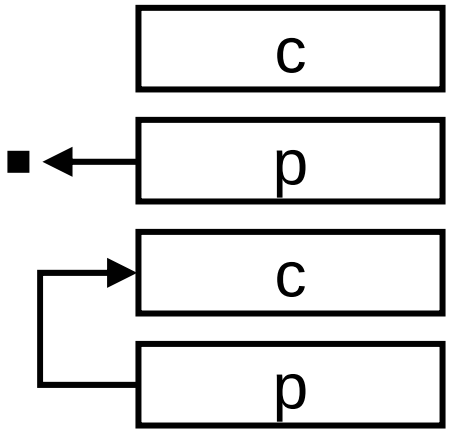
Operationen auf Zeigern (3)

b) Unärer Adressoperator &

Einem Zeiger kann die Adresse eines Objekts zugewiesen werden.

Beispiel:

```
int c;
int *p;
... {
   ...
   p = &c;
   /* p enthält nun die Adresse von c */
   ...
}
```



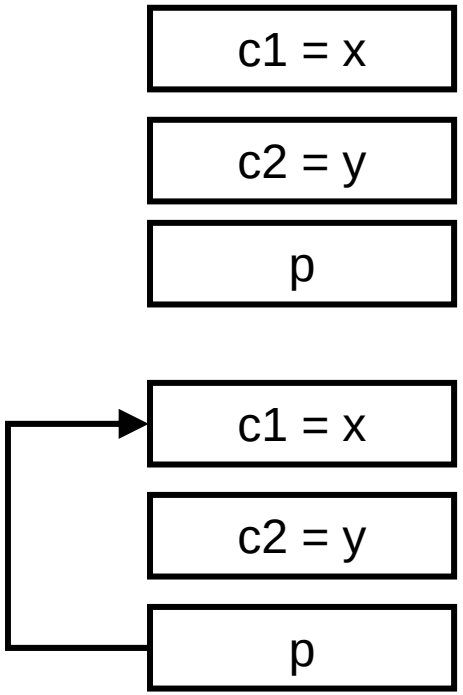
Operationen auf Zeigern (4)

c) Unärer Inhaltsoperator *

Einem Objekt kann der Inhalt eines anderen Objektes zugewiesen werden, auf das ein Zeiger zeigt (de-referencing).

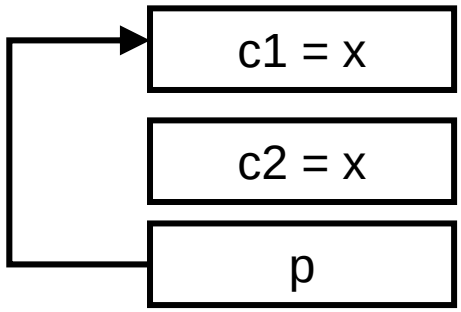
Beispiel:

```
int c1, c2;
int *p;
... {

  ...
  p = &c1;
```

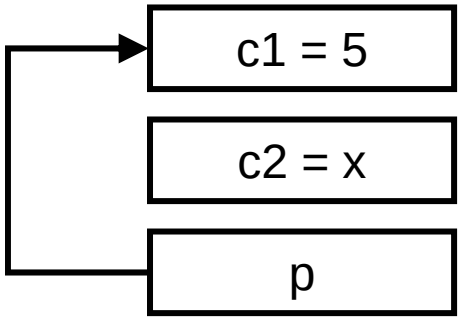


Operationen auf Zeigern (5)

```
    ...
    c2 = *p;
```




```
    ...
    *p = 5;
```




```
  ...
}
```

Operationen auf Zeigern (6)

Merke:
Die unären Adressoperatoren * und & haben höheren Vorrang als die dyadischen arithmetrischen Operatoren.

Beispiele:

`y = *p + 1`	liest den Wert, auf den p zeigt, aus, erhöht ihn um 1 und weist das Ergebnis y zu
`*p += 1`	inkrementiert den Inhalt der Speicherstelle, auf die p zeigt, um 1
`++*p`	ebenso (≡ `++(*p)`)
`(*p)++`	ebenso
`*p++`	inkrementiert p (die Adresse) (es wird von rechts nach links zusammen gefasst) (≡ `*(p++)`)

Operationen auf Zeigern (7)

d) Einrichten eines neuen Datenobjekts, auf das der Zeiger zeigt

Die Funktion `malloc` dient zum Allokieren (Reservieren) von Speicherplatz der angegebenen Größe. Sie liefert die Anfangsadresse des allokierten Speicherbereichs als generischen Pointer zurück.

```
type *p;
...
p = (type *) malloc (sizeof(type));
```

`malloc` wird dazu verwendet, Speicherplatz für ein Datenobjekt des Bezugstyps von p (Bezugsvariable) einzurichten. Die Funktion lässt dann den Zeiger p auf diesen Speicherplatz zeigen.

Operationen auf Zeigern (8)

Beispiel:

```
int  *p,
p = NULL;
p = (int *) malloc(sizeof(int));
```

Datenobjekt vom Typ int

Operationen auf Zeigern (9)

Der unäre Operator `sizeof` wird dabei verwendet, um die Größe des zu allokierenden Speicherplatzes anzugeben.

`sizeof` Objekt	liefert die Größe des angegebenen Speicherobjekts (z. B. Variable) in Bytes.
`sizeof` (Typname)	liefert die Größe des angegebenen Typs in Bytes.

Beispiel:

```
sizeof(char)  /*  liefert den Wert 1  */
```

Operationen auf Zeigern (10)

Zu Allokierung mehrerer gleichartiger Objekte am Stück gibt es eine spezielle Funktion:

```
void * calloc(size_t nitems, size_t size)
```

`calloc` reserviert für "nitems" Objekte der Größe "size" Speicherplatz. Diese Funktion ist insbesondere für das dynamische Erzeugen eines Vektors (arrays) interessant.

Beispiel:

```
p = calloc(7,sizeof(int));
```

Operationen auf Zeigern (11)

e) Löschen eines Datenobjekts, auf das ein Zeiger zeigt

```
free(p);
```

Der Speicherplatz für das Datenobjekt, auf das p zeigt, wird freigegeben. Der Wert von p ist anschließend undefiniert.

Merke:

Die Lebensdauer von Bezugsvariablen wird vom Programmierer explizit durch `malloc` und `free` bestimmt. Sie ist nicht an die Blockstruktur des Programms gebunden. Die Lebensdauer der Zeigervariablen selbst folgt dagegen der dynamischen Blockstruktur, wie die aller anderen Variablen auch.

Operationen auf Zeigern (12)

f) Vergleich von Zeigern

Alle Vergleichsoperatoren sind auch auf Zeigern möglich, aber nicht immer ist das Ergebnis sinnvoll.

Wenn die Zeiger p1 und p2 auf Elemente im gleichen linearen Adressraum zeigen, dann sind Vergleiche wie

==, !=, <, >, <=, >=

sinnvoll. Insbesondere ist dies bei einem Vektor von Elementen der Fall.

Beispiel:
Seien p1 und p2 Zeiger auf Elemente eines Vektors (array oder mit `calloc` bereit gestellt).

p1 < p2 gilt, wenn p1 auf ein früheres Element im Vektor zeigt als p2.

Sind p1 und p2 jedoch nicht Elemente des gleichen Vektors, ist das Ergebnis nicht definiert/nicht vorhersagbar.

Operationen auf Zeigern (13)

g) Arithmetik mit Zeigern

Addition und Subtraktion ganzzahliger Werte sind auf Zeigern erlaubt.

Beispiel:

`p += n` setzt p auf die Adresse des n-ten Objektes nach dem Objekt, auf das p momentan zeigt, (nicht n-tes Byte!)

`p++` setzt p auf die Adresse des nächsten Objektes

`p--` setzt p auf die Adresse des vorherigen Objektes

Merke:
Alle anderen Operationen mit Zeigern sind verboten, wie Addition, Multiplikation, Division oder Subtraktion zweier Zeiger, Bitoperationen oder Addition von Gleitpunktwerten auf Zeiger. Leider lassen manche Compiler solche Operationen trotzdem zu. Sie sind aber nicht standardkonform und sollten deshalb auch nicht verwendet werden.

Operationen auf Zeigern (14)

Merke:
Abgesehen von "void *" sollte ohne explizite Umwandlungsoperatoren kein Zeiger auf einen Datentyp an einen Zeiger auf einen anderen Datentyp zugewiesen werden!

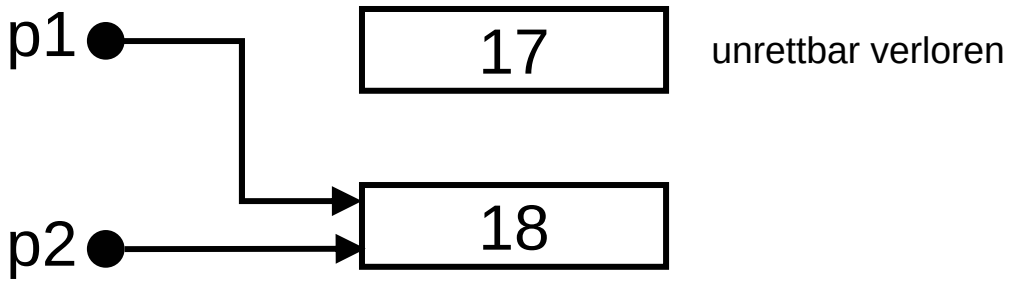
"Verlieren" von Datenobjekten im Speicher (1)

Das Umsetzen eines Zeigers ohne "free" führt dazu, dass die Bezugsvariable zwar noch existiert, aber nicht mehr adressiert werden kann. Der nicht freigegebene Speicherplatz bleibt blockiert.

Beispiel 1:

```
int *p1, *p2;
        .
        .
        .
p1  = (int *) malloc(sizeof(int));
*p1 =  17;
p2  = (int *) malloc(sizeof(int));
*p2 = 18;
p1  = p2;
```

"Verlieren" von Datenobjekten im Speicher (2)



"Verlieren" von Datenobjekten im Speicher (3)

Beispiel 2:

```
int *p1;
        .
        .
        .
p1  = (int *) malloc(sizeof(int));
*p1 = 17;
p1  = (int *) malloc(sizeof(int));
```

Auch hier ist das Datenelement, das den Wert "17" enthält, nicht mehr auffindbar.

Zeiger und Funktionen (1)

Durch die Übergabe von Zeigern als Parameter an Funktionen können Variablen der aufrufenden Umgebung geändert werden (**call by reference**), ohne dass diese als globale Variablen deklariert werden müssen.

Werden Variablen nicht über Zeiger an die Funktion übergeben, so kann diese ihren Inhalt nicht ändern, da sie nur auf den Wert der Variablen, nicht aber auf die Variable selbst zugreifen kann (**call by value**).

Zeiger und Funktionen (2)

Beispiel:

```
void swap (int x, int y)  /*  falsch  */ {
   int temp;
   temp = x;
   x = y;
   y = temp;          /*  Aufruf swap(a,b) vertauscht
                          nur die Kopien von a und b! */
}
void swap (int *px, int *py)  /* richtig */ {
   int temp;
   temp = *px;
   *px = *py;
   *py = temp;        /* Aufruf swap(*a,*b) vertauscht
                         die Inhalte von a und b */
}
```

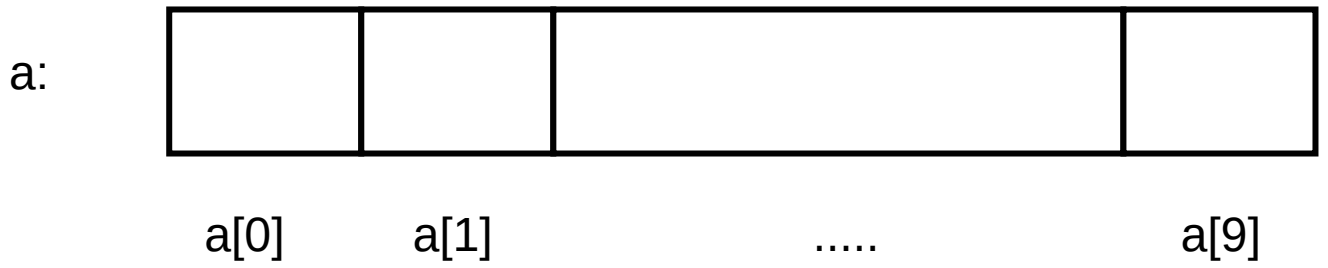
5.2 Vektoren (arrays)

Ein Vektor (array) ist ein Datenobjekt, das aus einer bestimmten Anzahl gleicher Teilobjekte (Elemente) besteht, die über einen Index angesprochen werden.

Beispiel:

```
int a[10];
```

definiert einen Vektor mit 10 Elementen mit den Namen a[0], ... a[9].



Vektoren: Programmierhinweise

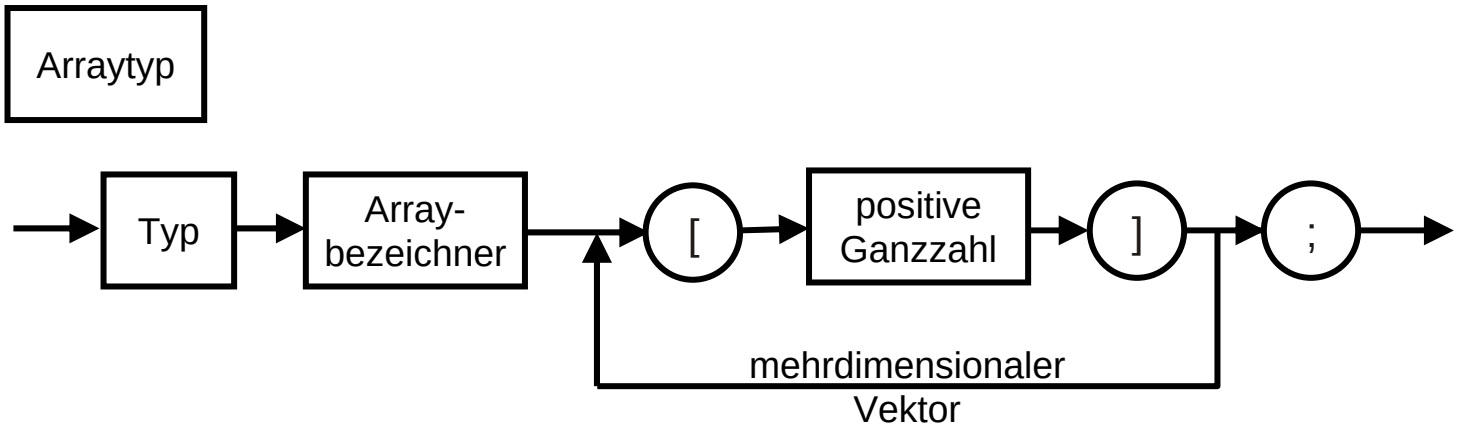
Hinweis 1:
Es ist gute Programmierpraxis, die Größe eines Arrays als Präprozessor-Konstante (Compiler-Konstante) zu definieren. Dann kann man sie später leichter ändern.

```
#define ASIZE 10;

int a[ASIZE];
```

Hinweis 2:
Ein Array-Bezeichner ist eine Konstante, nämlich die Anfangsadresse des Arrays (konstanter Zeiger). Auf ihm sind Zeigeroperationen nicht definiert. Jedoch kann ein Zeiger auf ein Element des Arrays gesetzt werden.

Array-Syntax in ANSI C

Deklaration:



Beispiel:

```
char name[20];
```

Prinzipiell kann der positive Integerwert, der die Vektorgröße der Dimension angibt, bei der ersten Dimension auch fehlen. Eine solche Deklaration ist jedoch nur in wenigen Situationen sinnvoll.

Array-Beispiel

Beispiel:

Falls ein Array bei der Deklarationen mit einer Serie von Werten initialisiert wird, muss die Größe nicht deklariert werden, sondern besteht automatisch aus der Anzahl der Werte.

```
char name[ ] = {'E','f','f','e','l','s','b','e','r','g'};
```

implizite Größe: 10 Elemente.

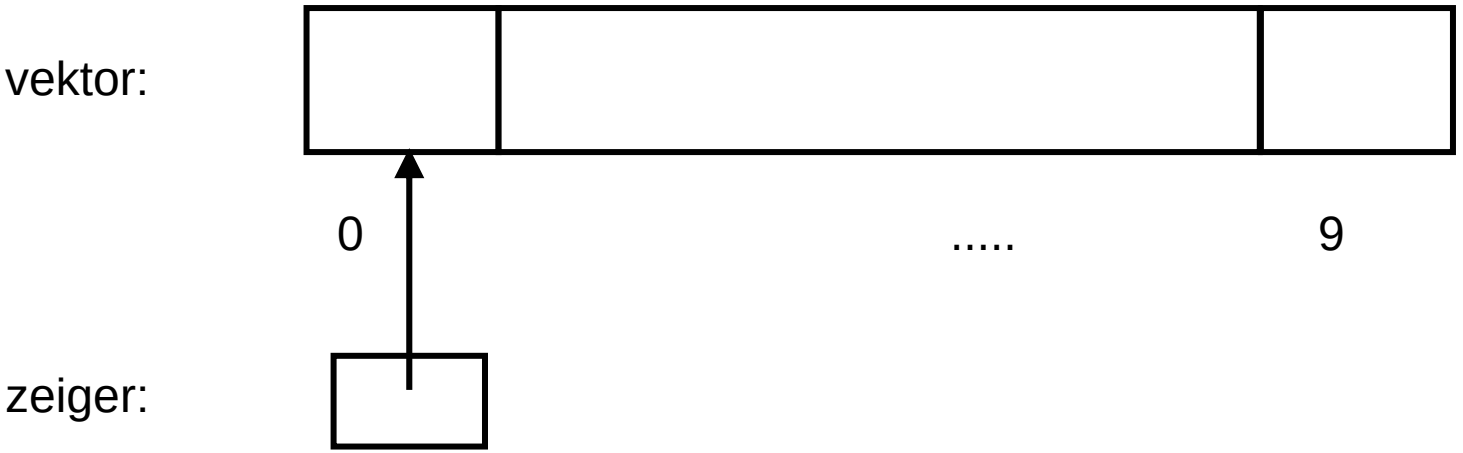
Weiteres Beispiel:

Als formaler Parameter im Routinenkopf

```
gib_zurueck (int vector[], int anz_elemente);
```

Unterschied zwischen Zeiger und Vektor (1)

```
int vektor[10];
int *zeiger = &vektor[0];
```



Unterschied zwischen Zeiger und Vektor (2)

Der Arrayname ist, wie schon erwähnt, ein konstanter Zeiger auf die Adresse des ersten Elementes. Damit sind folgende Zuweisungen erlaubt:

```
zeiger = vektor;      /* setzt zeiger auf vektor[0]  */
zeiger = vektor + i;  /* setzt zeiger auf vektor[i]  */
```

Änderungen von `vektor` (**konstanter** Zeiger) sind jedoch verboten:

```
vektor++
vektor +=2
vektor = zeiger
```

o. ä. SIND VERBOTEN!

Mehrdimensionaler Vektor

Beispiele:

```
int a[100];             /* 1-dimensionaler Vektor */
int b[3][5];            /* 2-dimensionaler Vektor */
int c[7][9][2];         /* 3-dimensionaler Vektor */
```

Elemente werden folgendermaßen angesprochen:

```
b[2][2]         /* Spalte 3, Zeile 3 der Matrix b  */
```

(Erinnerung: das erste Element eines Arrays hat den Index 0, nicht 1)

Ein mehrdimensionaler Vektor benötigt mindestens so viel zusammen hängenden Speicherplatz, wie sich aus dem Produkt der einzelnen Dimensionen errechnet.

Array-Rechnungen

In C gilt: Arrays werden **zeilenweise** abgelegt (letzter Index läuft zuerst).

Ausdrücke, die äquivalent zu b[i][j] sind:

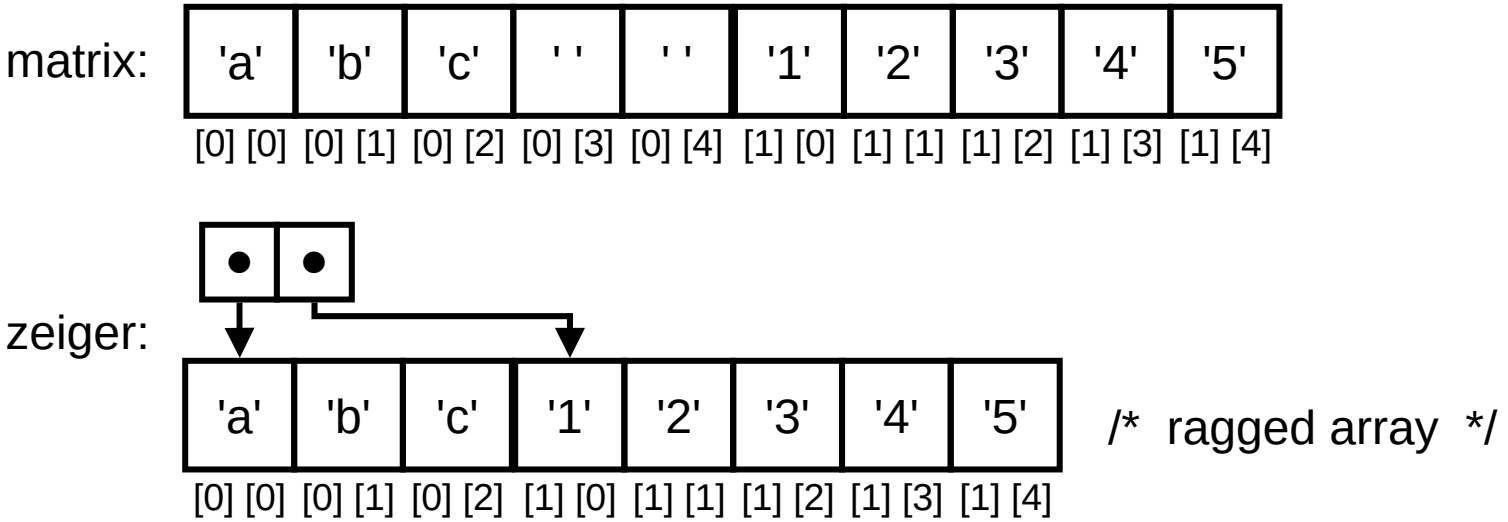
```
*(b[i] + j)
(*(b+i))[j]
*((*(b+i)) + j)
*(&b[0][0] + 5*i + j)
```

"Ausgefranste Arrays" (ragged arrays) (1)

Beispiel:

```
char matrix [2][5] = {{'a','b','c',' ',' '}, \
  {'1','2','3','4','5'}};

char *zeiger[2] = {{'a','b','c'}, {'1','2','3','4','5'}};
                /* '[ ]' bindet stärker als '*'   */
```

Speicherdarstellung:



"Ausgefranste Arrays" (ragged arrays) (2)

Vorteile der Zeigerverwendung in diesem Fall:

- geringerer Speicherplatzbedarf:

  `zeiger` belegt 2 Speicherzellen für Zeiger und keinen Speicherplatz für die leeren Stellen

  `matrix` belegt immer 2*5 Speicherzellen für die Zeichen

- schnellere Zugriffszeit:

  Über den Verweis geht der Zugriff schneller, als wenn die Position im Array berechnet werden muss:

  ```
  matrix[1][3]: matrix[0][0] + 5 * 1 + 3
  ```

  ```
  zeiger[1][3]: zeiger[1] + 3
  ```

5.3 Zeichenketten (strings)

Zeichenketten sind in C eindimensionale Vektoren vom Typ `char`, die mit einem **speziellen Endezeichen** (dem **Null**-Zeichen) versehen sind.

```
char name[ ] = {'E','f','f','e','l','s','b','e','r','g','\0'};
```

(**beachte:** die implizite Größe des Vektors `name` ist 11!)

Abkürzungsschreibweise:

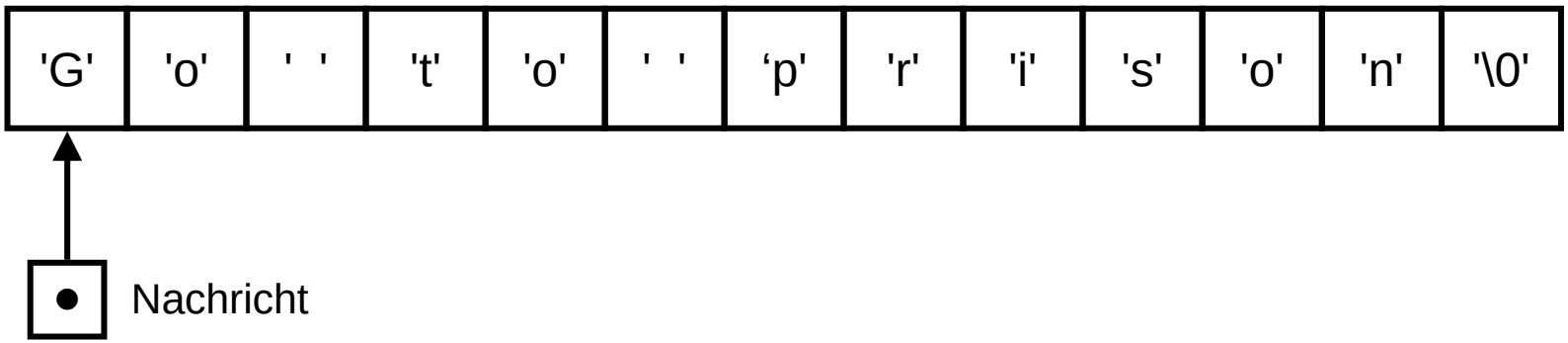
```
char name[ ] = "Effelsberg";
```

Merke:

- "a" mit Doppel-Anführungszeichen ist ein String, der mit dem Null-Zeichen beendet wird (Länge: 2).
- 'a' mit einfachen Anführungszeichen ist eine character-Konstante (Länge:1).
- '\0' besteht zwar aus zwei Zeichen, belegt aber nur einen Speicherplatz vom Typ `char` (Sonderzeichen, ESC-Zeichen) (Länge: 1).

Zeichenketten vs. Zeiger (1)

```
char *nachricht = "Go to prison";
        /* Zeiger auf eine Zeichenkette */
```

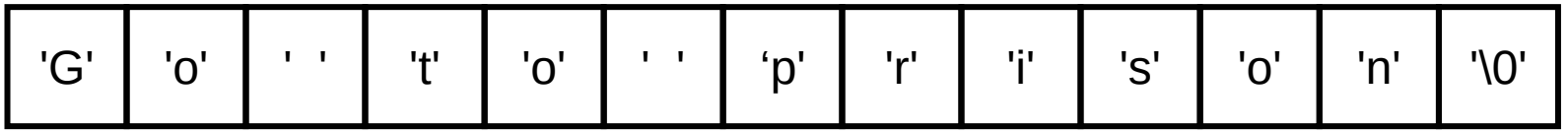


- Einzelne Zeichen in der Regel unveränderbar
- Zeiger veränderbar

Zeichenketten vs. Zeiger (2)

```
char zeichenkette[] = "Go to prison";
          /* Vektor mit Zeichen */
```

Zeichenkette:



- Inhalt veränderbar, auch die einzelnen Zeichen durch Direktzugriff
- Der Vektorbezeichner ist ein konstanter Zeiger auf die Anfangsadresse des Vektors.

Standard-String-Funktionen (1)

Beispiele: aus <string.h>

```
int strcmp(char *s1, char *s2)
```

lexikographischer Vergleich von s1 und s2 (gibt 0 zurück, wenn s1 gleich s2)

```
int strlen(char *s)
```

gibt Länge der Zeichenkette s ohne '\0' zurück

```
char *strcat(char *s1, char *s1)
```

gibt s1 um s2 verlängert zurück (Konkatenation). s1 muss genügend Platz für die Aufnahme von s2 bereit stellen.

```
char *strcpy(char *s1, char *s2)
```

kopiert s2 nach s1. s1 muss genügend Platz für die Aufnahme von s2 bereit stellen.

Standard String-Funktionen (2)

```
char *strchr(char *s1, char c)
```

gibt Zeiger auf das erste Vorkommen von c in s1 zurück

```
char *strstr( char *s1, char *s2)
```

gibt Zeiger auf erste Kopie von s2 in s1 zurück

5.4 Komplexe Datenstrukturen (structs)

Eine **Struktur** (struct) ist ein zusammengesetzter Datentyp. Er dient zur Konstruktion von Typen mit **inhomogenen** Komponenten (im Gegensatz zum Array).

Beispiel für eine komplexe Datenstruktur

```
struct    quadrat {
          int zahl;
          char buchstabe;
          } q;
```

"quadrat" ist das “Etikett“ für die Struktur („structure tag“).

Die Variablendefinition kann entweder direkt bei der Deklaration der Struktur stehen (wie oben mit q) oder extra.

Mit obiger Deklaration kann eine Variable wie folgt definiert werden:

```
struct quadrat q1;
```

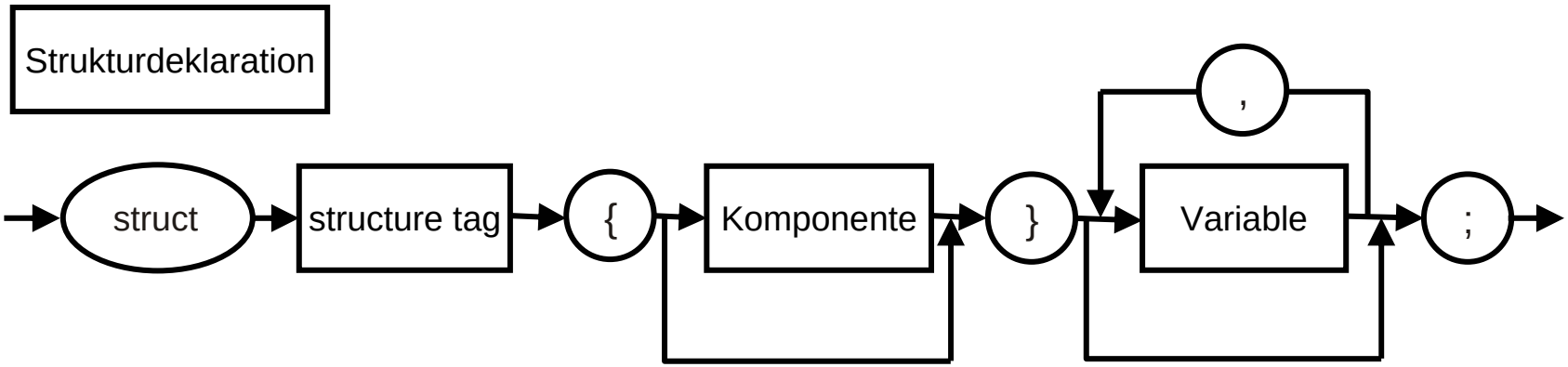
Wertzuweisung mit der Punkt-Notation:

```
q.zahl = 5;
q.buchstabe = 'A';
```

Merke:
Erst bei der Variablendefinition wird Speicherplatz belegt, nicht schon bei der Deklaration der Struktur.

Struct-Syntax in ANSI-C (1)

Syntax der Deklaration:



Das **Etikett** (structure tag) wird in Verbindung mit dem Schlüsselwort "struct" als Abkürzung für den Teil der Vereinbarung in geschweiften Klammern verwendet und definiert so einen neuen Typ.

Die einzelnen Felder, die in einer Struktur angegeben werden, heißen **Komponenten** (member fields) und können selbst wieder strukturierte Datentypen sein.

Struct-Syntax in ANSI-C (2)

Ein Zugriff auf eine Komponente erfolgt mit der folgenden Syntax (Punkt-Notation):

`strukturvariablenname.komponentenname,`

wobei die Komponentennamen eindeutig sein müssen.

Merke:
Komponenten verschiedener Strukturen können denselben Namen haben. Dies ist aber in der Regel nur sinnvoll, wenn die beiden Komponenten für verwandte Objekte stehen.

Möglichkeiten, mit Strukturen Variable zu definieren

1.)
```
struct quadrat  {
        :   }
```
=> deklariert eine Struktur namens quadrat.

Variablendefinition: `struct quadrat q1;`

2.)
```
typedef struct quadrat  {
            :
            } QUADRAT
```
=> deklariert einen neuen Typ namens QUADRAT.

Variablendefinition: QUADRAT q2;

3.)
```
struct quadrat  {
        :
        } q3;
```
=> deklariert eine Struktur namens quadrat.

=> definiert eine Variable namens q3, die diese Struktur hat.

Benutzerdefinierte Datenstrukturen (1)

Zeiger werden häufig für die Definition eigener komplexer Datenstrukturen eingesetzt.

Beispiel: lineare Listen (linked lists)

Alle Komponenten haben denselben Typ. Die Anordnung der Komponenten ist linear (nicht hierarchisch oder netzförmig). Für den Zeiger auf die erste Komponente wird häufig ein **Anker** (Listenkopf) eingerichtet.

Benutzerdefinierte Datenstrukturen (2)

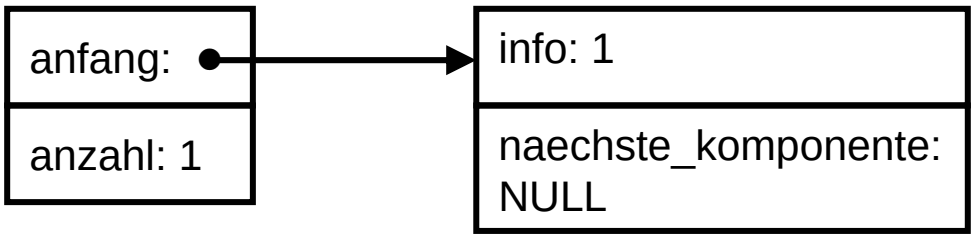
```
/* Typdefinition */
struct element {
       int info;
       struct element      *naechste_komponente;
       };

typedef struct element  KOMPONENTE;
typedef       KOMPONENTE * ZEIGER;
typedef       struct anker {
       ZEIGER anfang;
       int anzahl;
       } ANKER;
/* Variablendefinition */
ANKER  listenkopf;
ZEIGER p;
```

Die lineare Liste in C (1)

a) Erstes Element initialisieren

```
p = (ZEIGER) malloc(sizeof(KOMPONENTE));
(*p).info = 1;   /* Das info-Feld wird durchgezaehlt */
(*p).naechste_komponente = NULL;
listenkopf.anfang = p;
listenkopf.anzahl = 1;
```

listenkopf



Die lineare Liste in C (2)

Merke:

Der Operator "." hat eine stärkere Bindung (höhere Präzedenz) als " * ", so dass beim Ansprechen von Komponenten einer Struktur, auf die verwiesen wird, Klammern nötig sind. Diese Konstruktion kommt jedoch sehr oft vor, weshalb es eine Abkürzung dafür gibt:

`(*p).info` `p -> info`

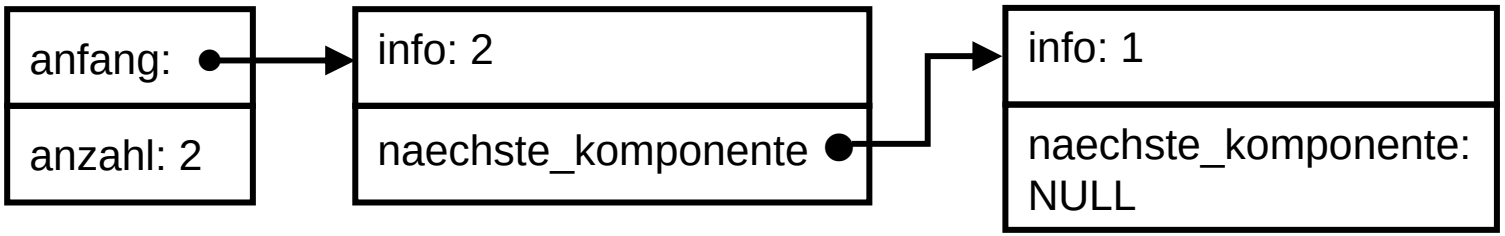
Bedeutung: zeiger_auf_Struktur -> komponente_der_struktur

Die lineare Liste in C (3)

b) Zweites Element mit Daten füllen

```
p = (ZEIGER) malloc(sizeof(KOMPONENTE));
p -> naechste_komponente = listenkopf.anfang;
p -> info = 2;
listenkopf.anfang = p;
listenkopf.anzahl++;
```

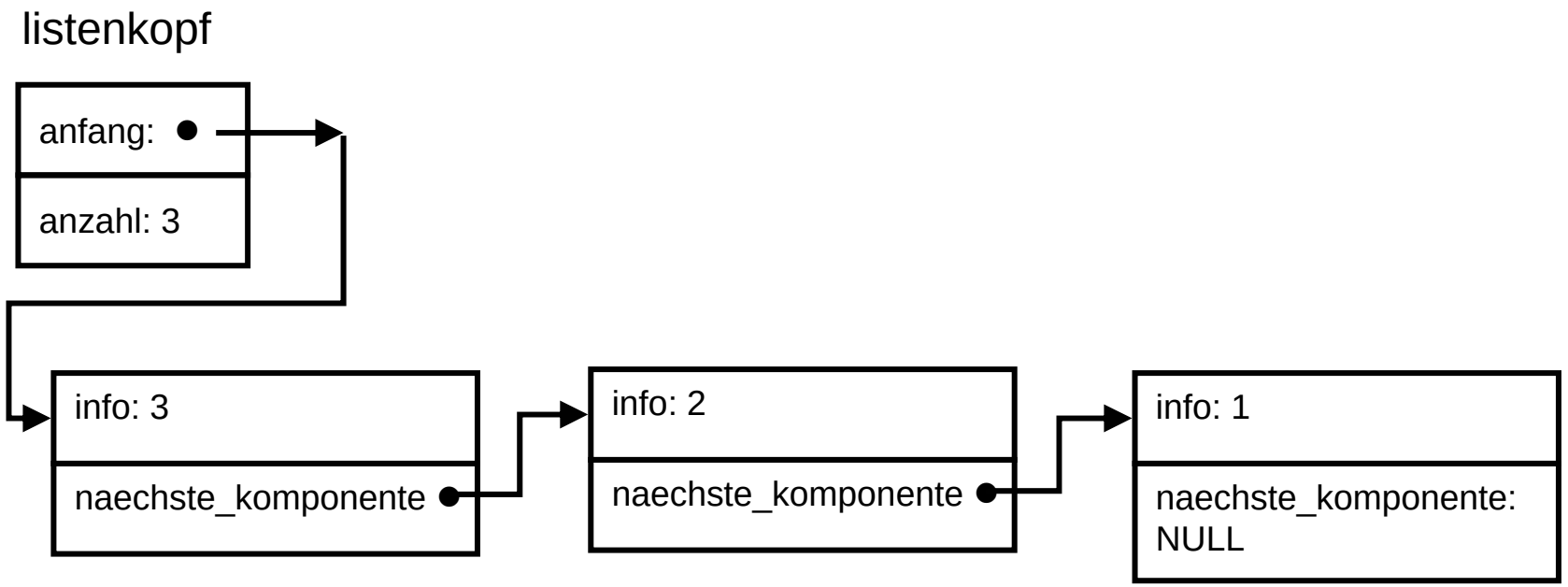
listenkopf



Die lineare Liste in C (4)

c) Drittes Element mit Daten füllen

```
p = (ZEIGER) malloc(sizeof(KOMPONENTEN);
p -> naechste_komponente = listenkopf.anfang;
p -> info = 3;
listenkopf.anfang = p;
listenkopf.anzahl++;
```



Die lineare Liste in C (5)

d) Ausgabe der Listenelemente

```
  .
  .
  .
p = listenkopf.anfang;
while (p != NULL)
{
   printf ("%d\n", p -> info);
   p = p -> naechste_komponente;
}
  .
  .
  .
```

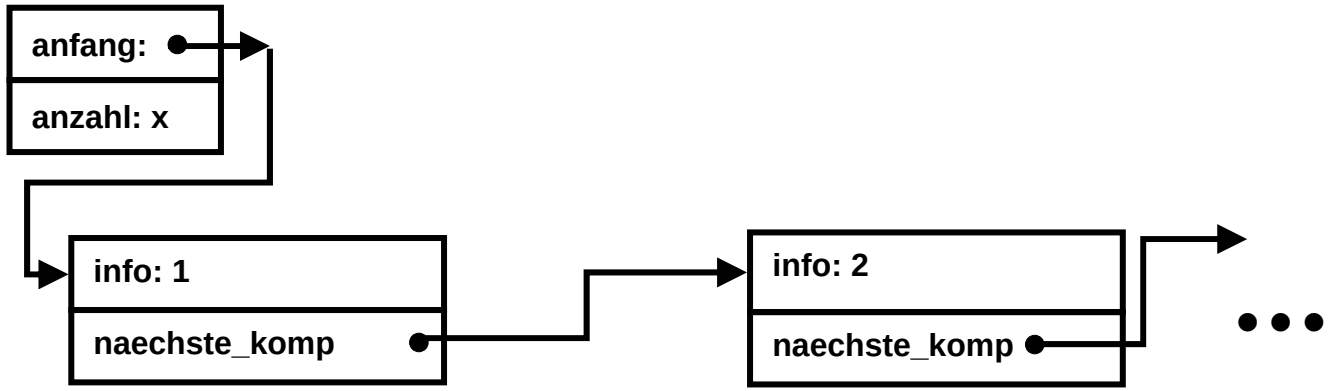
Die lineare Liste in C (6)

e) Freigabe der Listenelemente

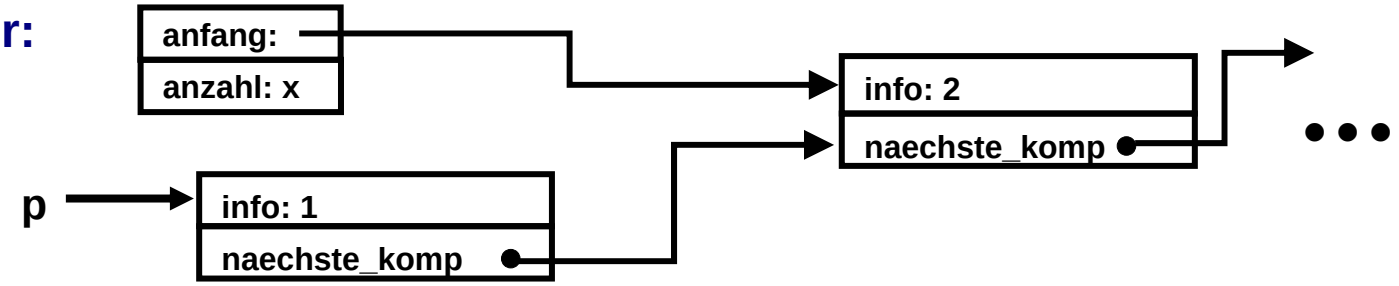
```
...
for  (p = listenkopf.anfang; p != NULL;
    \ p = listenkopf.anfang)
{
  listenkopf.anfang = p->naechste_komponente;
  free(p);
}
    ...
```

Die lineare Liste in C (7)

vorher:

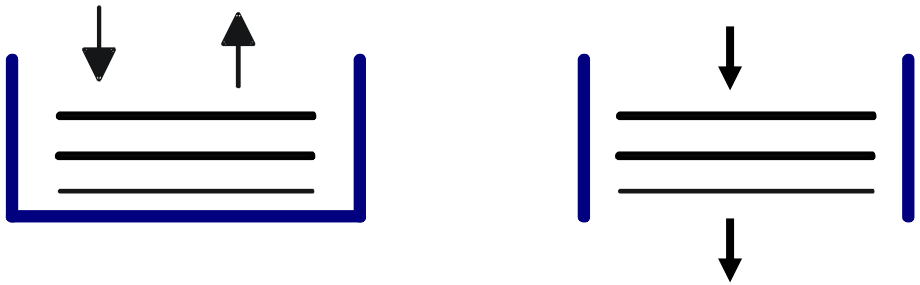


nachher:



Vorteile der Listenstruktur

- variable Länge; belegt wird nur der wirklich benötigte Speicherplatz (anders als beim Vektor)
- sehr flexibel: Einfügen und Löschen an beliebiger Stelle möglich
- durch Restriktionen bzgl. der Einfüge- und Löschposition entsteht ein Kellerspeicher (stack) oder eine Warteschlange (queue)



- durch Zeiger sind Verknüpfungen von Variablen verschiedener Typen möglich (anders als beim Vektor)