

Programmstruktur & Module

7.1 Softwaredesign

- Grundsätzliche Überlegungen über die Struktur eines Programms **vor** Beginn der Programmierung
- Verschiedene Design-Methoden
 - ◆ Top-down Entwurf / Prozedurale Programmierung
 - traditionelle Methode
 - bis Mitte der 80er Jahre fast ausschließlich verwendet
 - an Programmiersprachen wie Fortran, Cobol, Pascal oder C orientiert
 - ◆ Objekt-orientierter Entwurf
 - moderne, sehr aktuelle Methode
 - Ziel: Bewältigung sehr komplexer Probleme
 - auf Programmiersprachen wie C++, Smalltalk oder Java ausgerichtet

7.2 Top-down Entwurf

■ Zentrale Fragestellung

◆ was ist zu tun?

◆ in welche Teilaufgaben lässt sich die Aufgabe untergliedern?

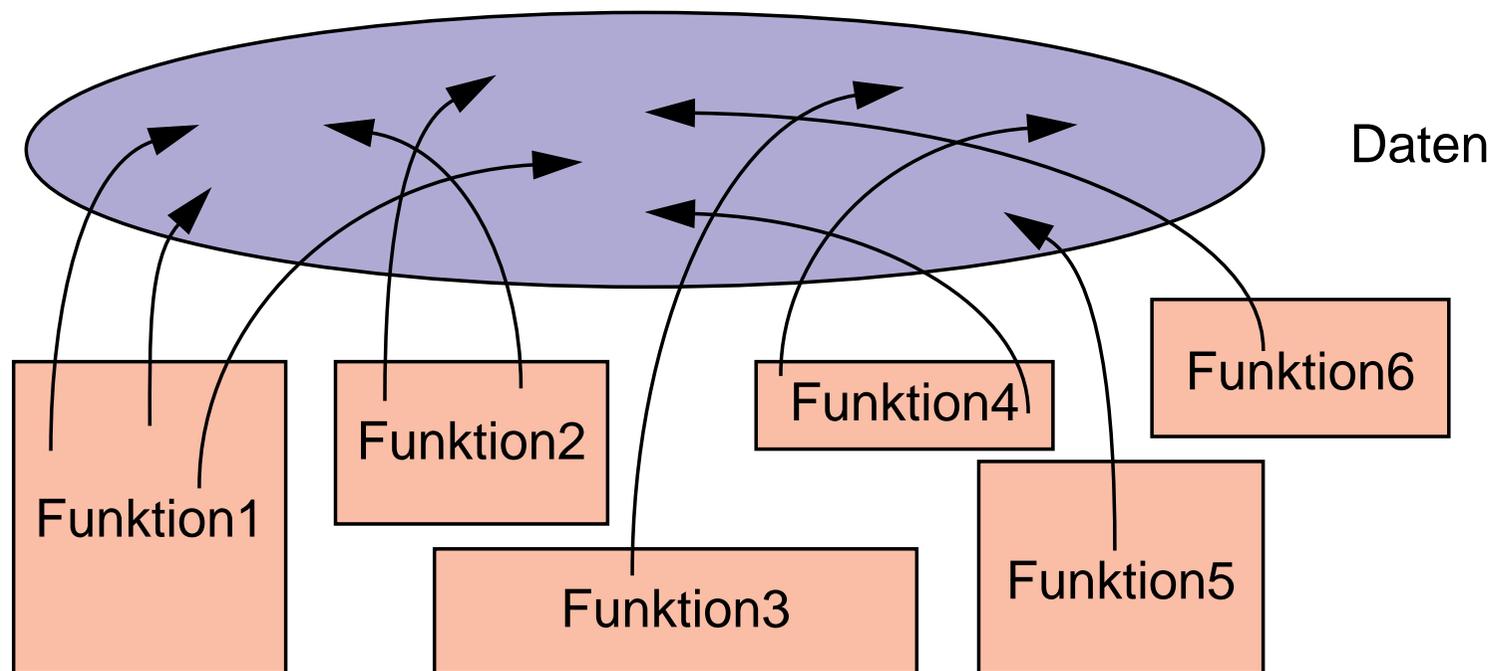
➤ Beispiel:

Rechnung für Kunden ausgeben

- Rechnungspositionen zusammenstellen
 - Lieferungspositionen einlesen
 - Preis für Produkt ermitteln
 - Mehrwertsteuer ermitteln
- Rechnungspositionen addieren
- Positionen formatiert ausdrucken

7.2 Top-down Entwurf (2)

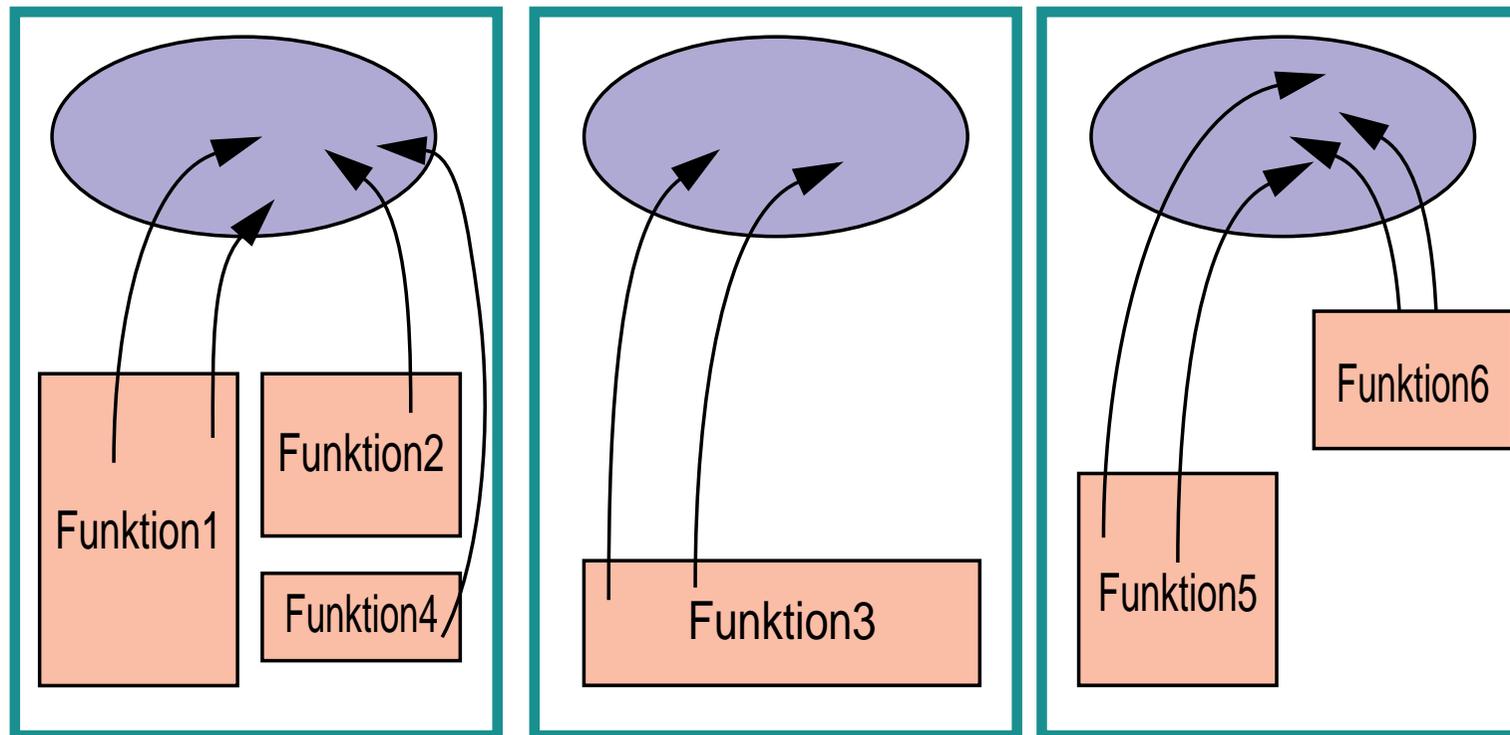
- **Problem:**
Gliederung betrifft nur die Aktivitäten, nicht die Struktur der Daten
- **Gefahr:**
Sehr viele Funktionen arbeiten "wild" auf einer Unmenge schlecht strukturierter Daten



7.2 Top-down Entwurf (3) — Modul-Bildung

- Lösung:
Gliederung von Datenbeständen zusammen mit Funktionen, die darauf operieren

➔ **Modul**



7.3 Module in C

- Teile eines C-Programms können auf mehrere `.c`-Dateien (C-Quelldateien) verteilt werden
- Logisch zusammengehörende Daten und die darauf operierenden Funktionen sollten jeweils zusammengefasst werden

➔ **Modul**

- Jede C-Quelldatei kann separat übersetzt werden (Option `-c`)
 - Zwischenergebnis der Übersetzung wird in einer `.o`-Datei abgelegt

```
% cc -c prog.c           (erzeugt Datei prog.o )
% cc -c f1.c             (erzeugt Datei f1.o )
% cc -c f2.c f3.c        (erzeugt f2.o und f3.o )
```

- Das Kommando `cc` kann mehrere `.c`-Dateien übersetzen und das Ergebnis — zusammen mit `.o`-Dateien — binden:

```
% cc -o prog prog.o f1.o f2.o f3.o f4.c f5.c
```

7.3 Module in C (2)

- !!! **.c-Quelldateien auf keinen Fall mit Hilfe der `#include`-Anweisung in andere Quelldateien einkopieren**
- Bevor eine Funktion aus einem anderen Modul aufgerufen werden kann, muss sie **deklariert** werden
 - Parameter und Rückgabewerte müssen bekannt gemacht werden
- Makrodefinitionen und Deklarationen, die in mehreren Quelldateien eines Programms benötigt werden, werden zu **Header-Dateien** zusammengefasst
 - ◆ *Header-Dateien* werden mit der `#include`-Anweisung des Präprozessors in C-Quelldateien einkopiert
 - ◆ der Name einer *Header-Datei* endet immer auf **.h**

7.4 Gültigkeit von Namen

- Gültigkeitsregeln legen fest, welche Namen (Variablen und Funktionen) wo im Programm bekannt sind

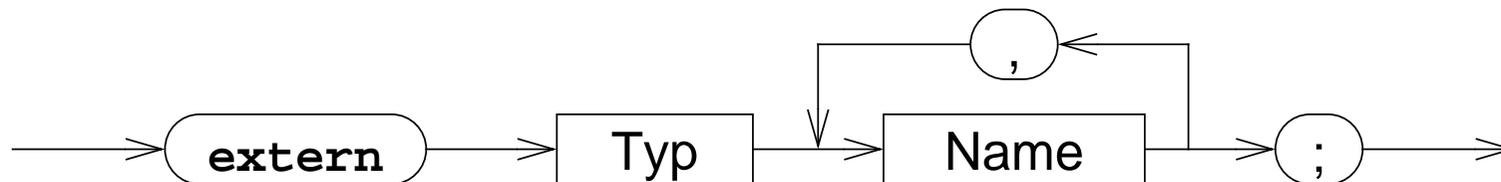
- Mehrere Stufen
 1. Global im gesamten Programm
(über Modul- und Funktionsgrenzen hinweg)
 2. Global in einem Modul
(auch über Funktionsgrenzen hinweg)
 3. Lokal innerhalb einer Funktion
 4. Lokal innerhalb eines Blocks

- Überdeckung bei Namensgleichheit
 - eine lokale Variable innerhalb einer Funktion überdeckt gleichnamige globale Variablen
 - eine lokale Variable innerhalb eines Blocks überdeckt gleichnamige globale Variablen und gleichnamige lokale Variablen in umgebenden Blöcken

7.5 Globale Variablen

Gültig im gesamten Programm

- Globale Variablen werden außerhalb von Funktionen definiert
- Globale Variablen sind ab der Definition in der gesamten Datei zugreifbar
- Globale Variablen, die in anderen Modulen **definiert** wurden, müssen vor dem ersten Zugriff bekanntgemacht werden
(**extern-Deklaration** = Typ und Name bekanntmachen)



- Beispiele:

```
extern int a, b;  
extern char c;
```

7.5 Globale Variablen (2)

■ Probleme mit globalen Variablen

- ◆ Zusammenhang zwischen Daten und darauf operierendem Programmcode geht verloren
- ◆ Funktionen können Variablen ändern, ohne dass der Aufrufer dies erwartet (Seiteneffekte)
- ◆ Programme sind schwer zu pflegen, weil bei Änderungen der Variablen erst alle Programmteile, die sie nutzen gesucht werden müssen

➔ **globale Variablen möglichst vermeiden**

7.5 Globale Funktionen

- Funktionen sind generell global
(es sei denn, die Erreichbarkeit wird explizit auf das Modul begrenzt)
- Funktionen aus anderen Modulen müssen ebenfalls vor dem ersten Aufruf **deklariert** werden
(= Typ, Name und Parametertypen bekanntmachen)
- Das Schlüsselwort **extern** ist bei einer Funktionsdeklaration nicht notwendig
- Beispiele:

```
double sinus(double);  
float power(float, int);
```
- Globale Funktionen (und soweit vorhanden die globalen Daten) bilden die äußere Schnittstelle eines Moduls
 - "vertragliche" Zusicherung an den Benutzer des Moduls

7.6 Einschränkung der Gültigkeit auf ein Modul

- Zugriff auf eine globale Variable oder Funktion kann auf das Modul (= die Datei) beschränkt werden, in der sie definiert wurde
 - Schlüsselwort `static` vor die Definition setzen

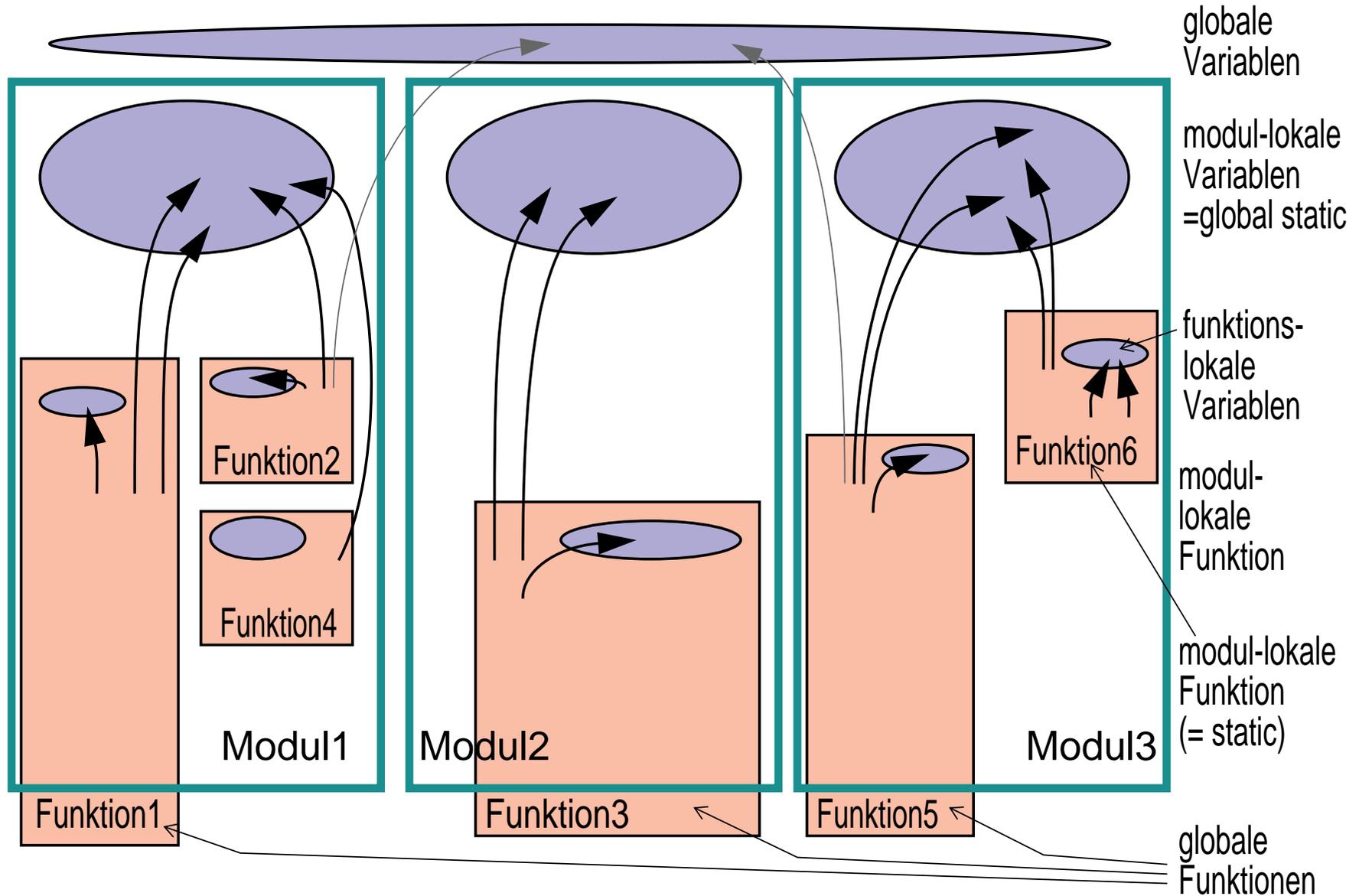


- `extern`-Deklarationen in anderen Modulen sind nicht möglich
- Die `static`-Variablen bilden zusammen den Zustand eines Moduls, die Funktionen des Moduls operieren auf diesem Zustand
- Hilfsfunktionen innerhalb eines Moduls, die nur von den Modulfunktionen benötigt werden, sollten immer `static` definiert werden
 - sie werden dadurch nicht Bestandteil der Modulschnittstelle (= des "Vertrags" mit den Modulbenutzern)
- !!! das Schlüsselwort `static` gibt es auch bei lokalen Variablen (mit anderer Bedeutung! - dort jeweils *kursiv* geschrieben)

7.7 Lokale Variablen

- Variablen, die innerhalb einer Funktion oder eines Blocks definiert werden, sind lokale Variablen
- bei Namensgleichheit zu globalen Variablen oder lokalen Variablen eines umgebenden Blocks gilt die jeweils letzte Definition
- lokale Variablen sind außerhalb des Blocks, in dem sie definiert wurden, nicht zugreifbar und haben dort keinen Einfluss auf die Zugreifbarkeit von Variablen

7.8 Gültigkeitsbereiche — Übersicht



7.9 Lebensdauer von Variablen

- Die Lebensdauer einer Variablen bestimmt, wie lange der Speicherplatz für die Variable aufgehoben wird

- Zwei Arten
 - ◆ Speicherplatz bleibt für die gesamte Programmausführungszeit reserviert
 - statische (*static*) Variablen

 - ◆ Speicherplatz wird bei Betreten eines Blocks reserviert und danach wieder freigegeben
 - dynamische (*automatic*) Variablen

7.9 Lebensdauer von Variablen (2)

auto-Variablen

- Alle lokalen Variablen sind automatic-Variablen
 - der Speicher wird bei Betreten des Blocks / der Funktion reserviert und bei Verlassen wieder freigegeben
 - ➔ der Wert einer lokalen Variablen ist beim nächsten Betreten des Blocks nicht mehr sicher verfügbar!

- Lokale auto-Variablen können durch beliebige Ausdrücke initialisiert werden
 - die Initialisierung wird bei jedem Eintritt in den Block wiederholt
 - !!! wird eine auto-Variable nicht initialisiert, ist ihr Wert vor der ersten Zuweisung undefiniert (= irgendwas)**

7.9 Lebensdauer von Variablen (3)

static-Variablen

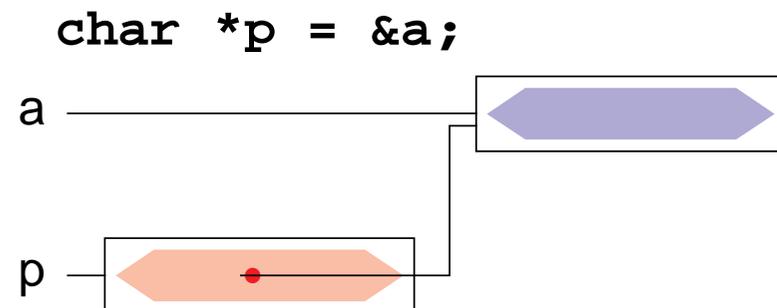
- Der Speicher für alle globalen Variablen ist generell von Programmstart bis Programmende reserviert
- Lokale Variablen erhalten bei Definition mit dem Schlüsselwort *static* eine **Lebensdauer über die gesamte Programmausführung** hinweg
 - ➔ der Inhalt bleibt bei Verlassen des Blocks erhalten und ist bei einem erneuten Eintreten in den Block noch verfügbar
- !!! Das Schlüsselwort *static* hat bei globalen Variablen eine völlig andere Bedeutung (Einschränkung des Zugriffs auf das Modul)
- *Static*-Variablen können durch beliebige konstante Ausdrücke initialisiert werden
 - die Initialisierung wird nur einmal beim Programmstart vorgenommen (auch bei lokalen Variablen!)
 - erfolgt keine explizite Initialisierung, wird automatisch mit 0 vorbelegt

Zeiger(-Variablen)

8.1 Einordnung

- **Konstante:**
Bezeichnung für einen Wert
- **Variable:**
Bezeichnung eines Datenobjekts
- **Zeiger-Variable (Pointer):**
Bezeichnung einer Referenz auf ein Datenobjekt

'a' ≡  0110 0001



8.2 Überblick

- Eine Zeigervariable (***pointer***) enthält als Wert die Adresse einer anderen Variablen
 - ↳ *der Zeiger verweist auf die Variable*
- Über diese Adresse kann man **indirekt** auf die Variable zugreifen
- Daraus resultiert die große Bedeutung von Zeigern in C
 - ↳ Funktionen können ihre Argumente verändern (***call-by-reference***)
 - ↳ dynamische Speicherverwaltung
 - ↳ effizientere Programme
- Aber auch Nachteile!
 - ↳ Programmstruktur wird unübersichtlicher (welche Funktion kann auf welche Variable zugreifen?)
 - ↳ häufigste Fehlerquelle bei C-Programmen

8.3 Definition von Zeigervariablen

■ Syntax:

```
Typ *Name ;
```

▲ Beispiele

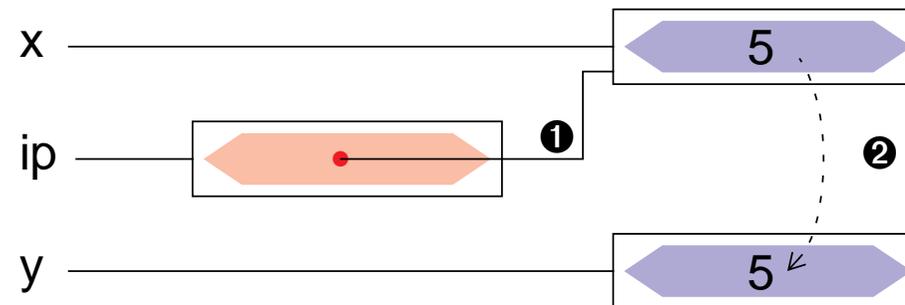
```
int x = 5;
```

```
int *ip;
```

```
int y;
```

```
ip = &x; ①
```

```
y = *ip; ②
```



8.4 Adressoperatoren

▲ Adressoperator `&`

`&x` der unäre Adress-Operator liefert eine Referenz auf den Inhalt der Variablen (des Objekts) `x`

▲ Verweisoperator `*`

`*x` der unäre Verweisoperator `*` ermöglicht den Zugriff auf den Inhalt der Variablen (des Objekts), auf die der Zeiger `x` verweist

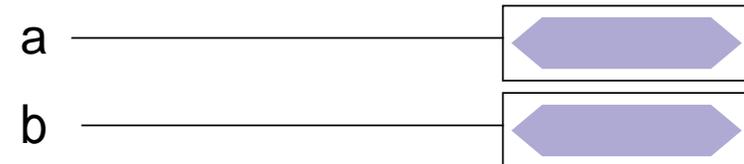
8.5 Zeiger als Funktionsargumente

- Parameter werden in C *by-value* übergeben
- die aufgerufene Funktion kann den aktuellen Parameter beim Aufrufer nicht verändern
- auch Zeiger werden *by-value* übergeben, d. h. die Funktion erhält lediglich eine Kopie des Adressverweises
- über diesen Verweis kann die Funktion jedoch mit Hilfe des *-Operators auf die zugehörige Variable zugreifen und sie verändern
 - ↳ *call-by-reference*

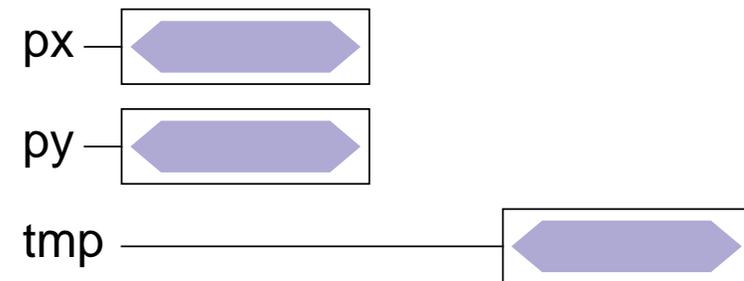
8.5 Zeiger als Funktionsargumente (2)

■ Beispiel:

```
void swap (int *, int *);  
int main() {  
    int a, b;  
    ...  
    swap(&a, &b);  
    ...  
}
```



```
void swap (int *px, int *py)  
{  
    int tmp;  
  
    tmp = *px;  
    *px = *py;  
    *py = tmp;  
}
```

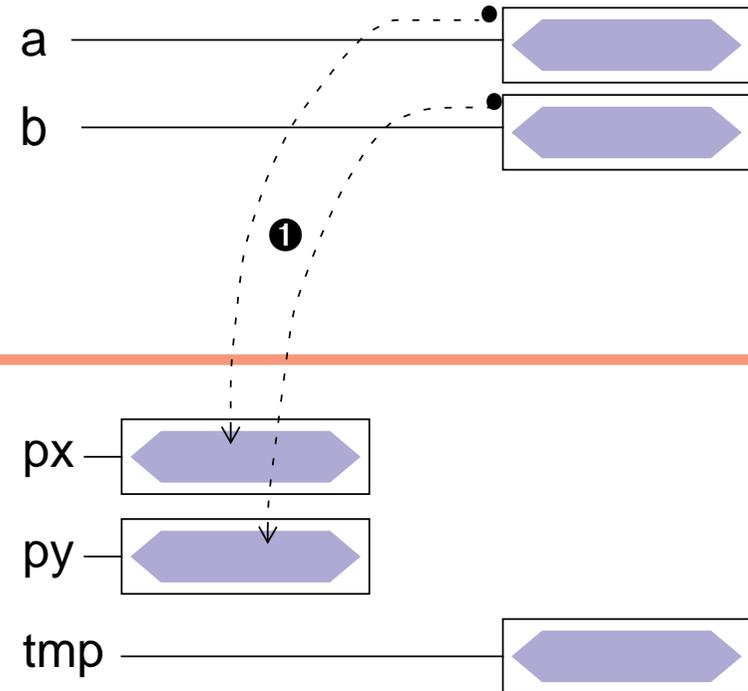


8.5 Zeiger als Funktionsargumente (2)

■ Beispiel:

```
void swap (int *, int *);  
int main() {  
    int a, b;  
    ...  
    swap(&a, &b); ❶  
    ...  
}
```

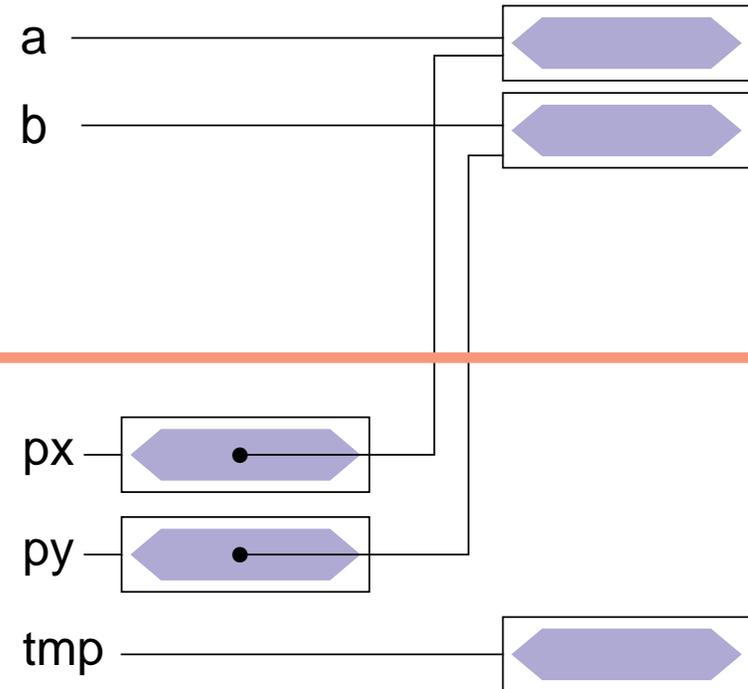
```
void swap (int *px, int *py)  
{  
    int tmp;  
  
    tmp = *px;  
    *px = *py;  
    *py = tmp;  
}
```



8.5 Zeiger als Funktionsargumente (2)

■ Beispiel:

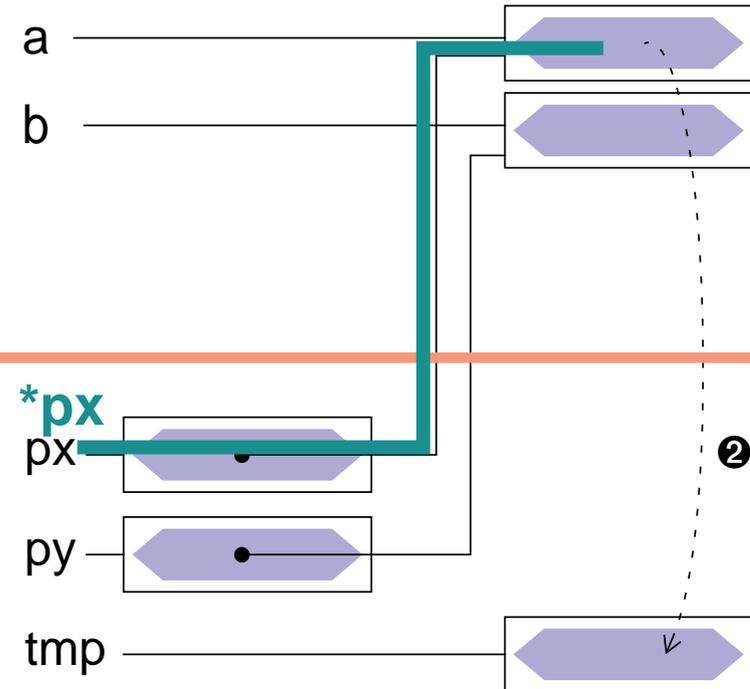
```
void swap (int *, int *);  
int main() {  
    int a, b;  
    ...  
    swap(&a, &b);  
    ...  
}  
  
void swap (int *px, int *py)  
{  
    int tmp;  
  
    tmp = *px;  
    *px = *py;  
    *py = tmp;  
}
```



8.5 Zeiger als Funktionsargumente (2)

■ Beispiel:

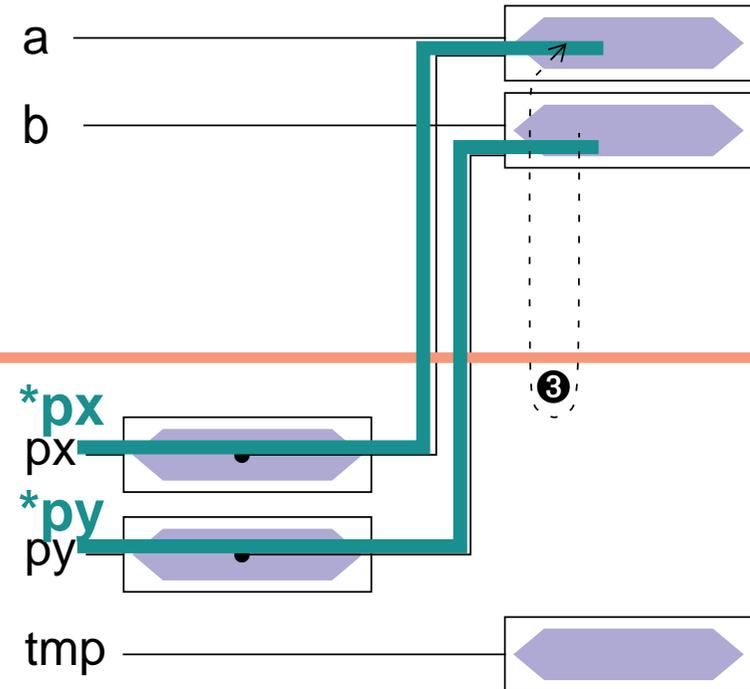
```
void swap (int *, int *);  
int main() {  
    int a, b;  
    ...  
    swap(&a, &b);  
    ...  
}  
  
void swap (int *px, int *py)  
{  
    int tmp;  
  
    tmp = *px; ②  
    *px = *py;  
    *py = tmp;  
}
```



8.5 Zeiger als Funktionsargumente (2)

■ Beispiel:

```
void swap (int *, int *);  
int main() {  
    int a, b;  
    ...  
    swap(&a, &b);  
    ...  
}  
  
void swap (int *px, int *py)  
{  
    int tmp;  
  
    tmp = *px;  
    *px = *py; ③  
    *py = tmp;  
}
```



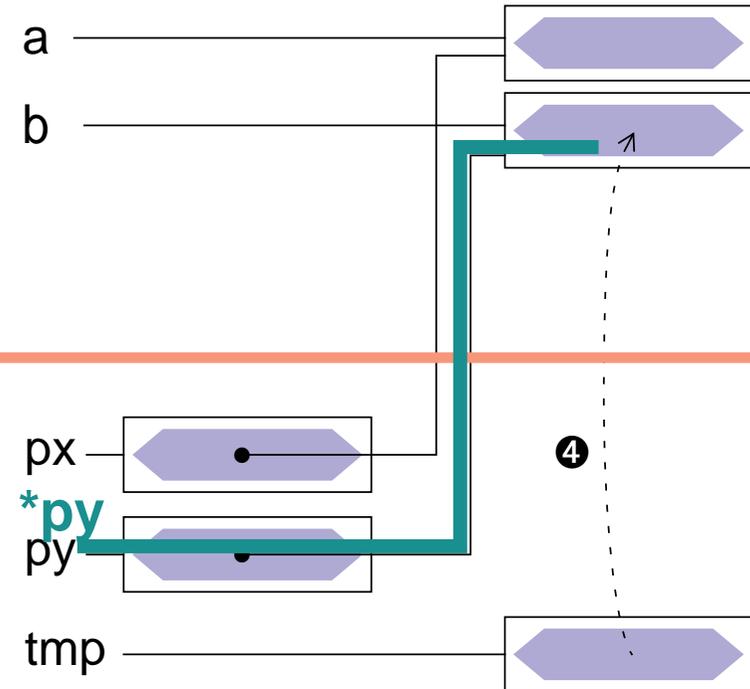
8.5 Zeiger als Funktionsargumente (2)

■ Beispiel:

```
void swap (int *, int *);
int main() {
    int a, b;
    ...
    swap(&a, &b);
    ...
}

void swap (int *px, int *py)
{
    int tmp;

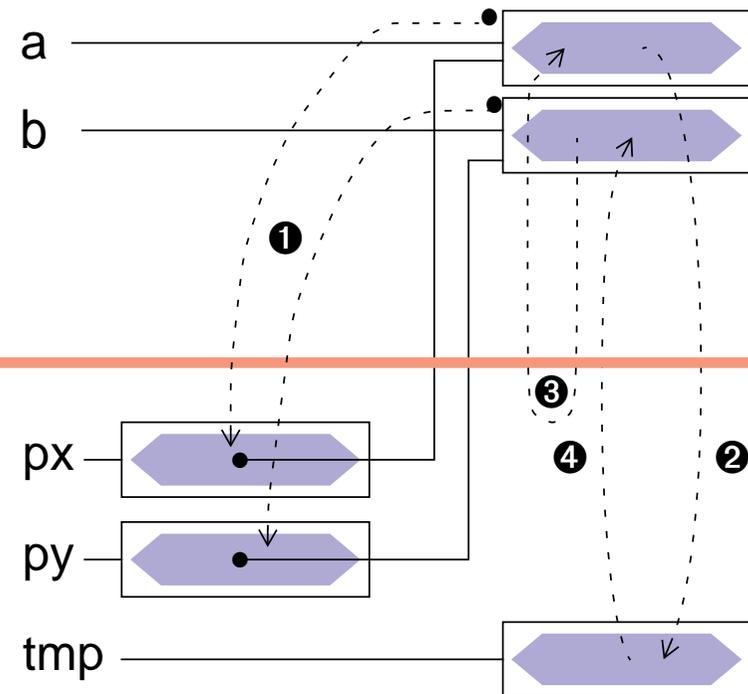
    tmp = *px;
    *px = *py;
    *py = tmp; ④
}
```



8.5 Zeiger als Funktionsargumente (2)

■ Beispiel:

```
void swap (int *, int *);  
int main() {  
    int a, b;  
    ...  
    swap(&a, &b); ①  
    ...  
}  
  
void swap (int *px, int *py)  
{  
    int tmp;  
  
    tmp = *px; ②  
    *px = *py; ③  
    *py = tmp; ④  
}
```



8.6 Zeiger auf Strukturen

- Konzept analog zu "Zeiger auf Variablen"
 - Adresse einer Struktur mit &-Operator zu bestimmen
 - Zeigerarithmetik berücksichtigt Strukturgröße

- Beispiele

```
struct person stud1;  
struct person *pstud;  
pstud = &stud1;           /* ⇒ pstud → stud1 */
```

- Besondere Bedeutung zum Aufbau verketteter Strukturen

8.6 Zeiger auf Strukturen (2)

■ Zugriff auf Strukturkomponenten über einen Zeiger

■ Bekannte Vorgehensweise

- *-Operator liefert die Struktur
- .-Operator zum Zugriff auf Komponente
- Operatorenvorrang beachten



```
(*pstud).alter = 21;
```

unleserlich!

■ Syntaktische Verschönerung



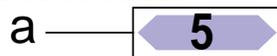
->-Operator

```
pstud->alter = 21;
```

8.7 Zusammenfassung

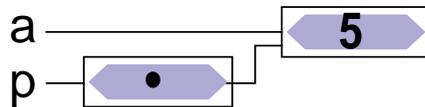
■ Variable

```
int a;
```



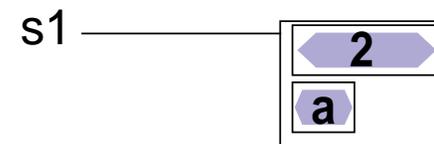
■ Zeiger

```
int *p = &a;
```



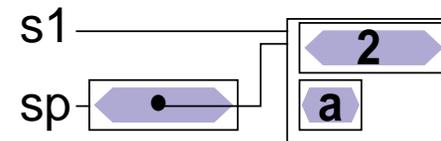
■ Struktur

```
struct s{int a; char c;};  
struct s s1 = {2, 'a'};
```



■ Zeiger auf Struktur

```
struct s *sp = &s1;
```



Felder

9.1 Eindimensionale Felder

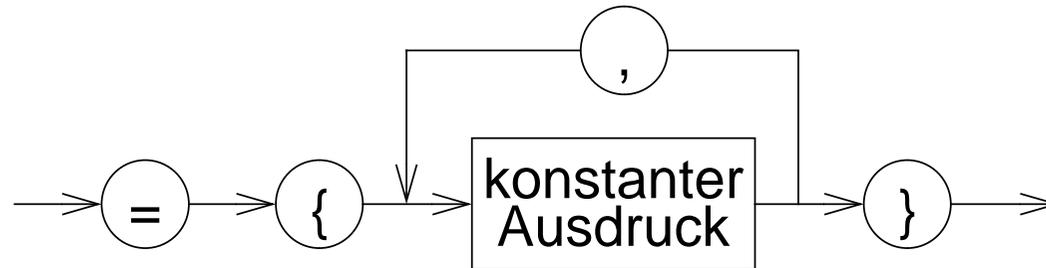
- eine Reihe von Daten desselben Typs kann zu einem **Feld** zusammengefasst werden
- bei der Definition wird die Größe des Felds angegeben
 - Größe muss eine Konstante sein
 - ab C99 bei lokalen Feldern auch zur Laufzeit berechnete Werte zulässig
- der Zugriff auf die Elemente erfolgt durch **Indizierung**, beginnend bei Null
- Definition eines Feldes



- Beispiele:

```
int x[5];  
double f[20];
```

9.2 Initialisierung eines Feldes



- Ein Feld kann durch eine Liste von konstanten Ausdrücken, die durch Komma getrennt sind, initialisiert werden

```
int prim[4] = {2, 3, 5, 7};  
char name[5] = {'0', 't', 't', 'o', '\0'};
```

- wird die explizite Felddimensionierung weggelassen, so bestimmt die Zahl der Initialisierungskonstanten die Feldgröße

```
int prim[] = {2, 3, 5, 7};  
char name[] = {'0', 't', 't', 'o', '\0'};
```

- werden zu wenig Initialisierungskonstanten angegeben, so werden die restlichen Elemente mit 0 initialisiert

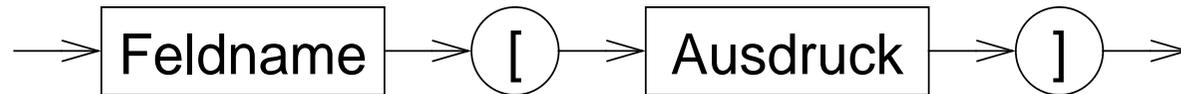
9.3 Initialisierung eines Feldes (2)

- Felder des Typs ***char*** können auch durch String-Konstanten initialisiert werden

```
char name1[5] = "Otto";  
char name2[] = "Otto";
```

9.4 Zugriffe auf Feldelemente

■ Indizierung:



wobei: $0 \leq \text{Wert}(\text{Ausdruck}) < \text{Feldgröße}$

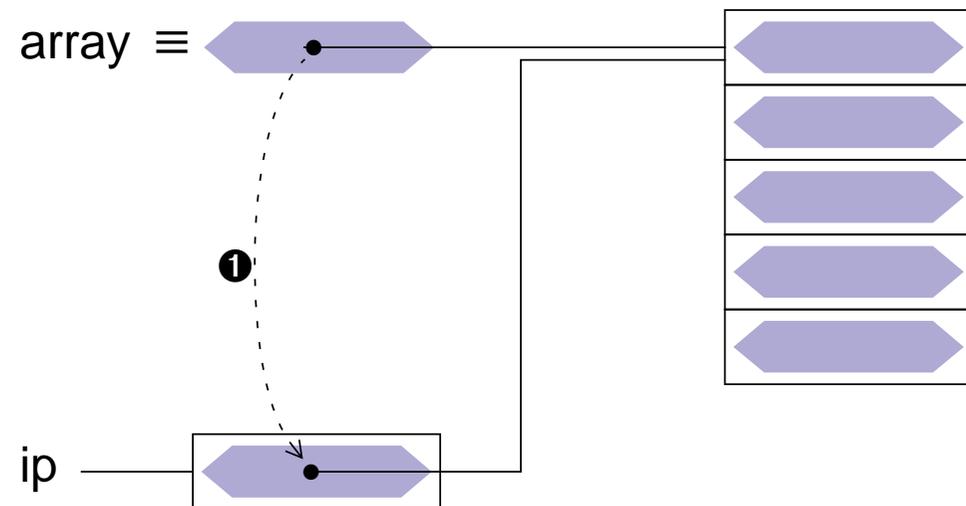
■ Beispiele:

```
prim[0] == 2  
prim[1] == 3  
name[1] == 't'  
name[4] == '\0'
```

Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

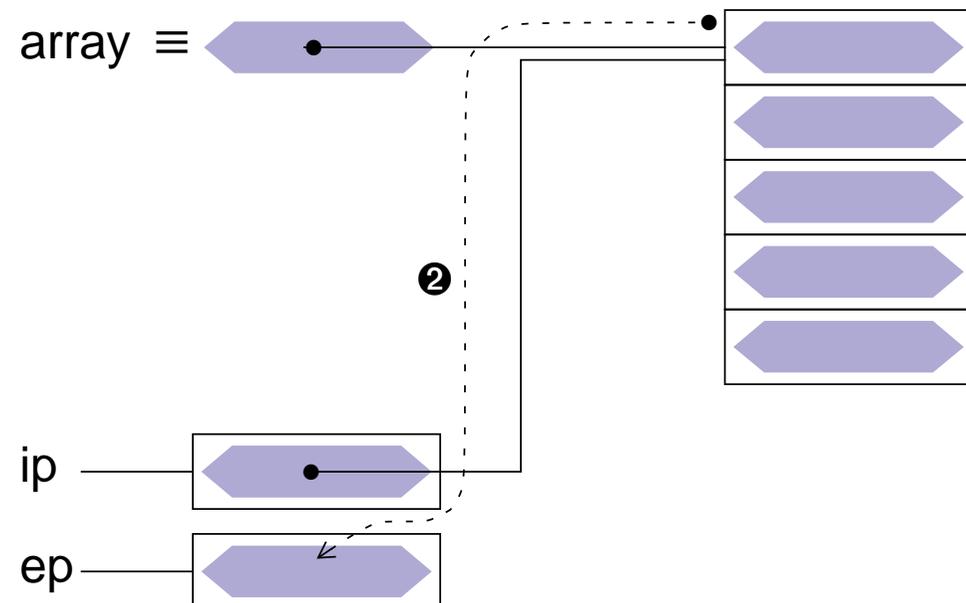
```
int array[5];  
  
int *ip = array; ❶
```



Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

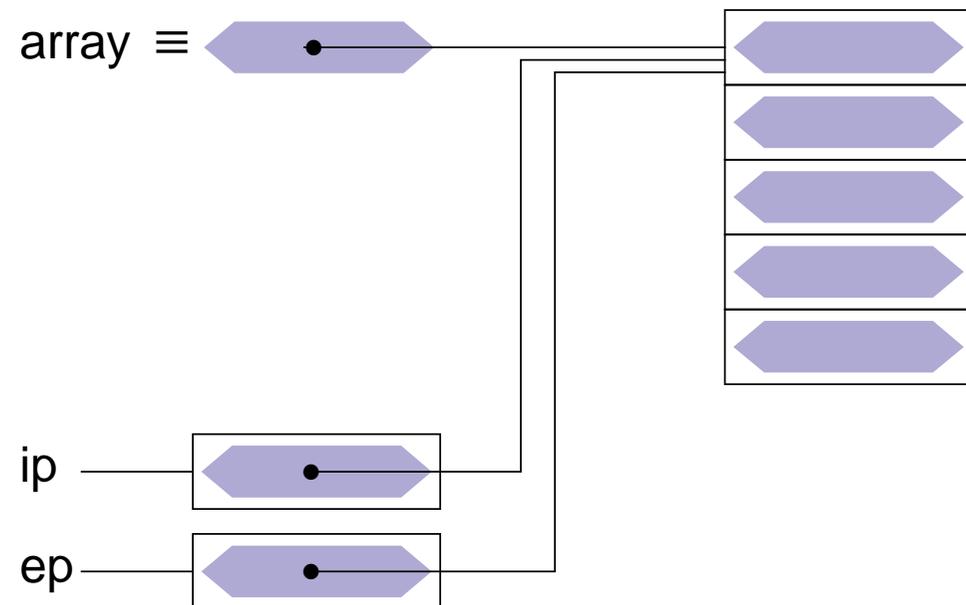
```
int array[5];  
  
int *ip = array;  
  
int *ep;  
ep = &array[0]; ②
```



Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

```
int array[5];  
  
int *ip = array;  
  
int *ep;  
ep = &array[0]; ②
```



Zeiger und Felder

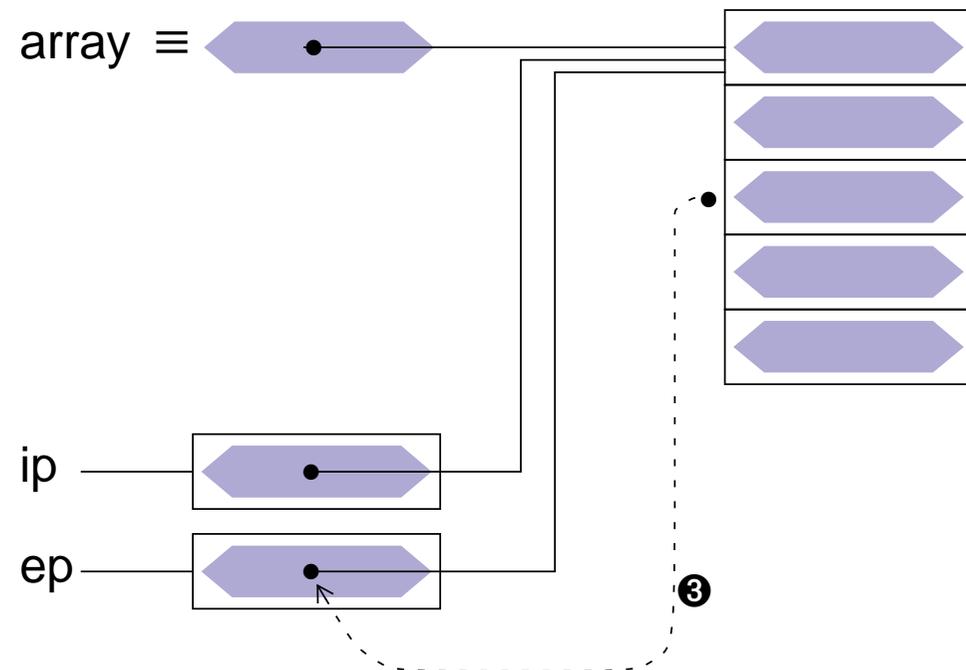
- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

```
int array[5];

int *ip = array;

int *ep;
ep = &array[0];

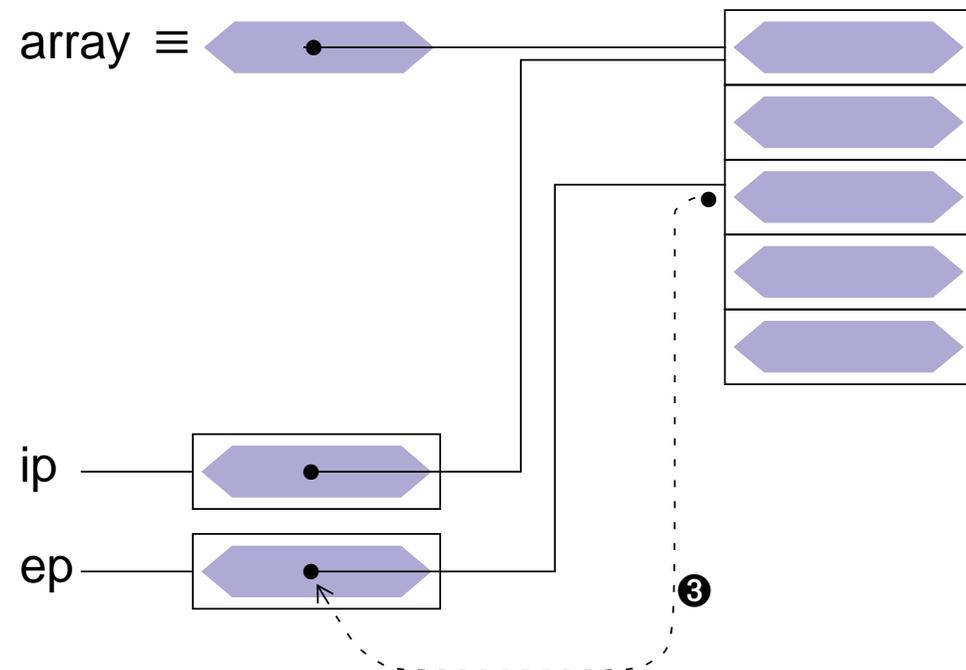
ep = &array[2]; ③
```



Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

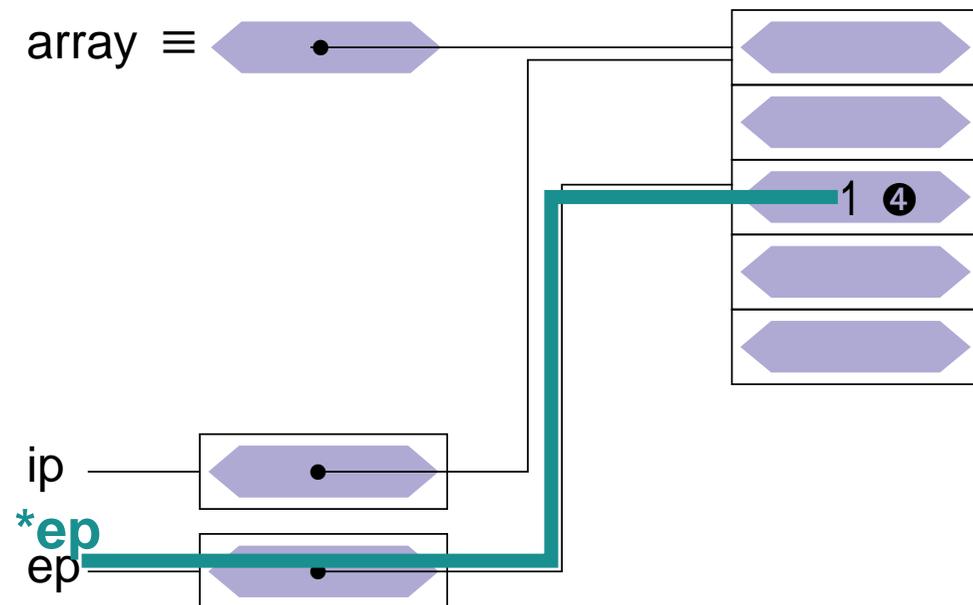
```
int array[5];  
  
int *ip = array;  
  
int *ep;  
ep = &array[0];  
  
ep = &array[2]; ③
```



Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

```
int array[5];  
  
int *ip = array;  
  
int *ep;  
ep = &array[0];  
  
ep = &array[2];  
  
*ep = 1; ④
```



Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

```

int array[5];

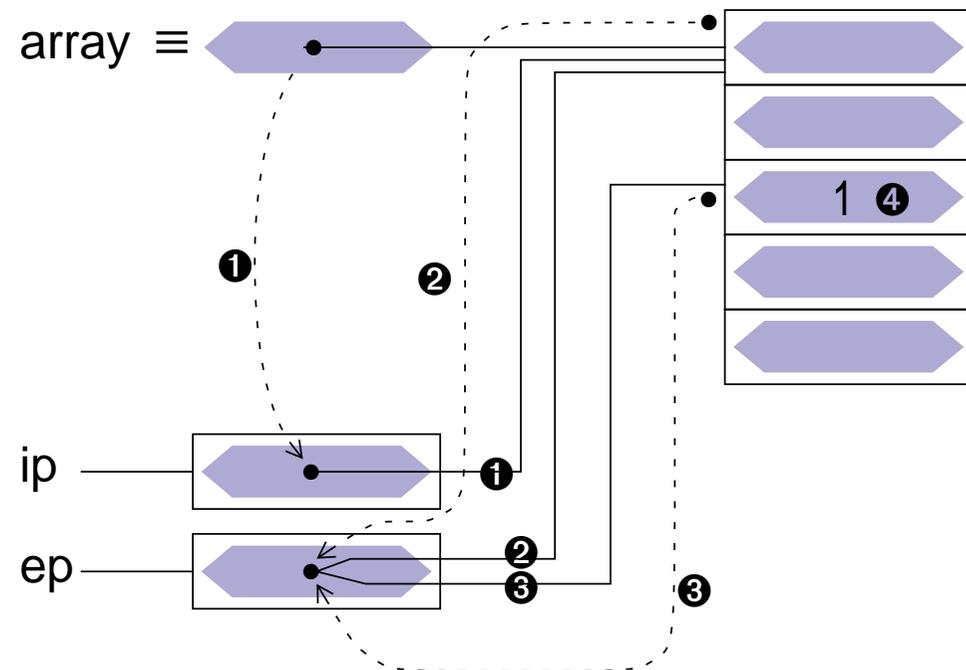
int *ip = array; ❶

int *ep;
ep = &array[0]; ❷

ep = &array[2]; ❸

*ep = 1; ❹

```

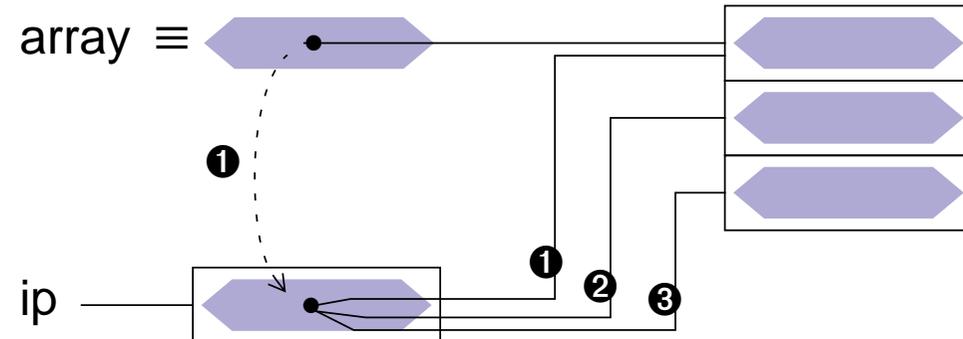


10.1 Arithmetik mit Adressen

- ++ -Operator: Inkrement = nächstes Objekt

```
int array[3];
int *ip = array; ❶

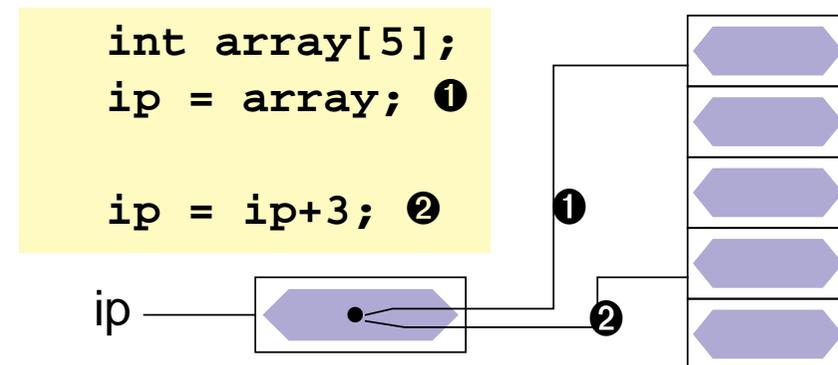
ip++; ❷
ip++; ❸
```



- -- -Operator: Dekrement = vorheriges Objekt

- +, -
Addition und Subtraktion von
Zeigern und ganzzahligen Werten.

Dabei wird immer die Größe des
Objektyps berücksichtigt!



!!! Achtung: Assoziativität der Operatoren beachten

10.2 Zeigerarithmetik und Felder

- Ein Feldname ist eine Konstante für die Adresse des Feldanfangs
 - ↳ Feldname ist ein ganz normaler Zeiger
 - Operatoren für Zeiger anwendbar (*, [])
 - ↳ aber keine Variable → keine Modifikationen erlaubt
 - keine Zuweisung, kein ++, --, +=, ...

- es gilt:

```
int array[5]; /* → array ist Konstante für den Wert &array[0] */
int *ip = array; /* ≡ int *ip = &array[0] */
int *ep;

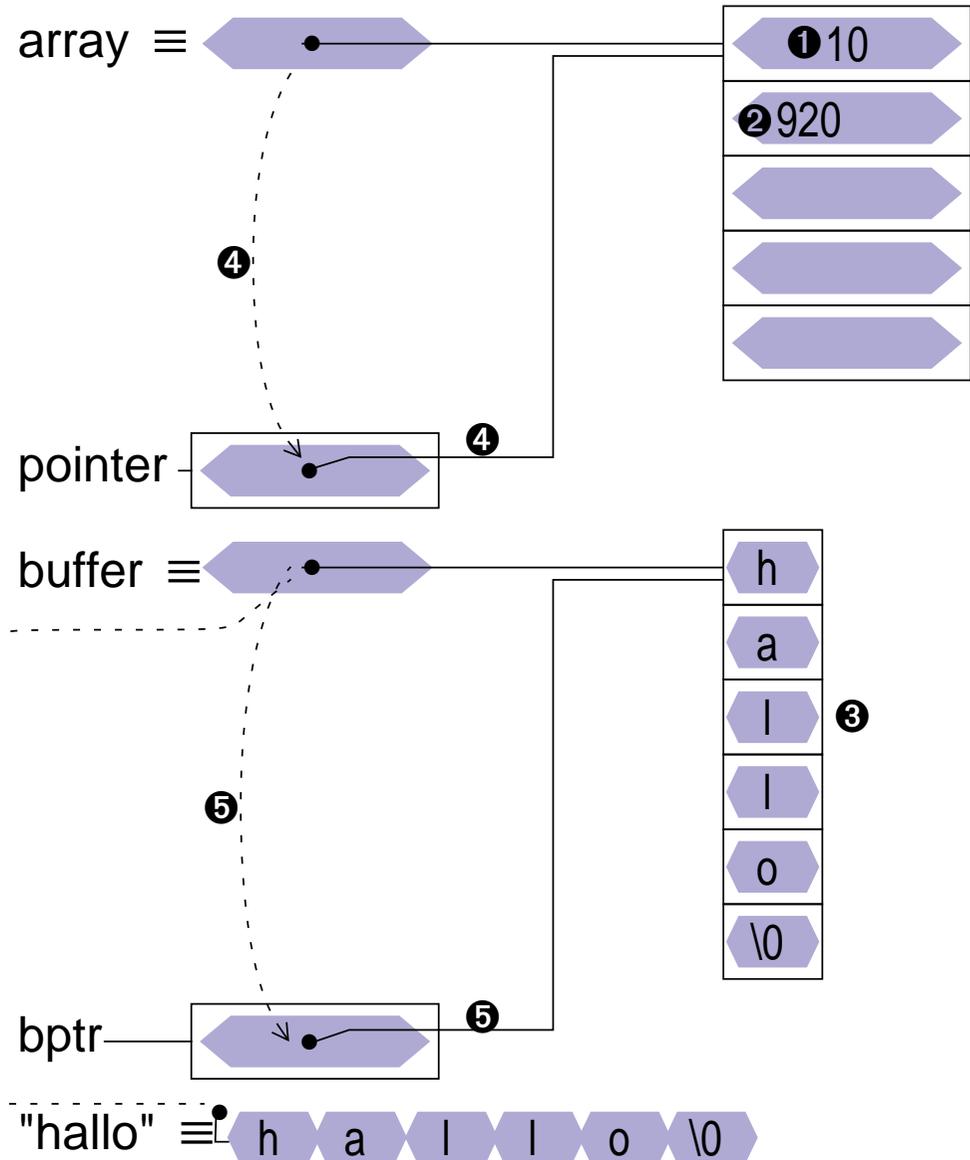
/* Folgende Zuweisungen sind äquivalent */
array[i] = 1;
ip[i] = 1;
*(ip+i) = 1;      /* Vorrang ! */
*(array+i) = 1;

ep = &array[i]; *ep = 1;
ep = array+i; *ep = 1;
```

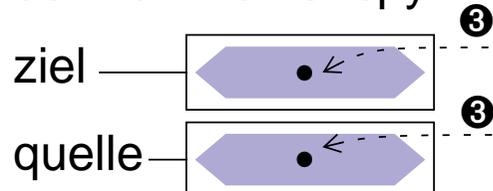
10.2 Zeigerarithmetik und Felder

```
int array[5];
int *pointer;
char buffer[6];
char *bptr;
```

- ① array[0] = 10;
- ② array[1] = 920;
- ③ strcpy(buffer, "hallo");
- ④ pointer = array;
- ⑤ bptr = buffer;



Formale Parameter
der Funktion strcpy



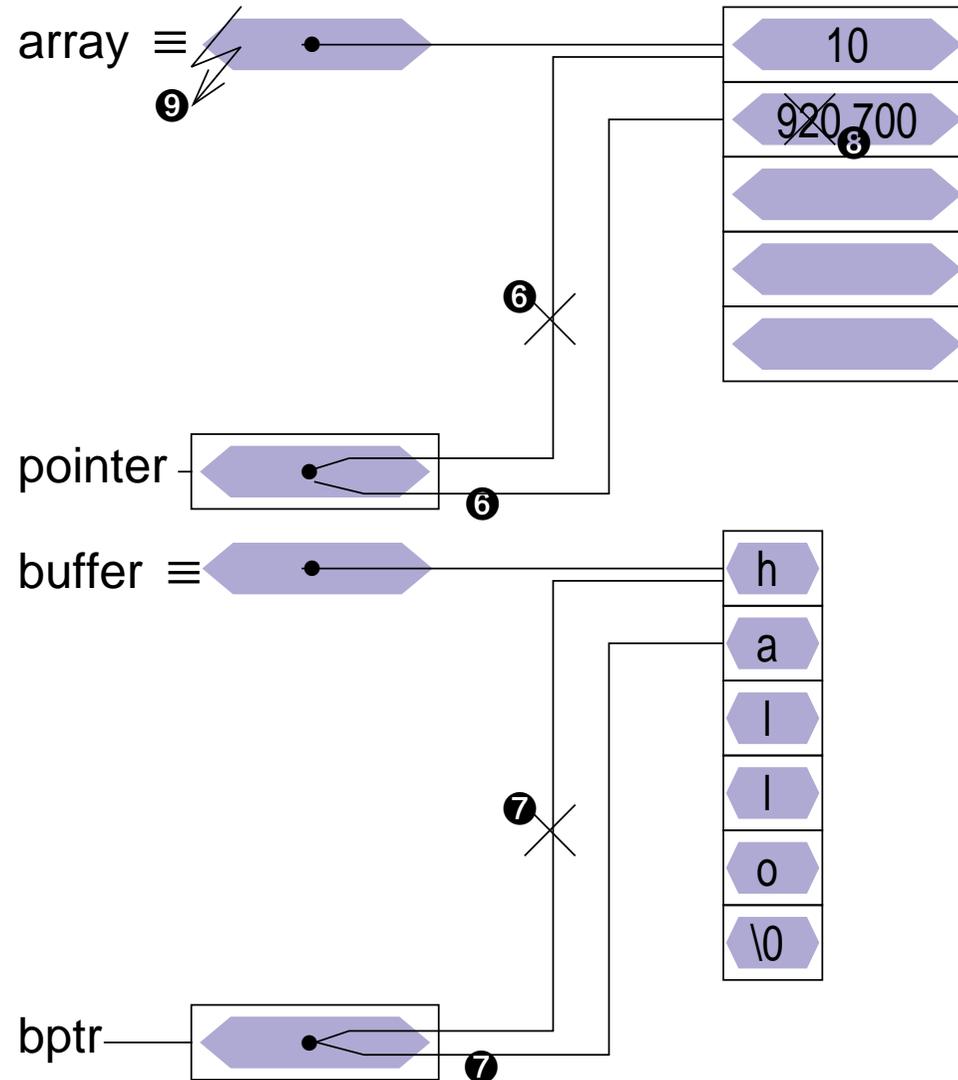
10.2 Zeigerarithmetik und Felder

```
int array[5];
int *pointer;
char buffer[6];
char *bptr;
```

```
① array[0] = 10;
② array[1] = 920;
③ strcpy(buffer, "hallo");
④ pointer = array;
⑤ bptr = buffer;
```

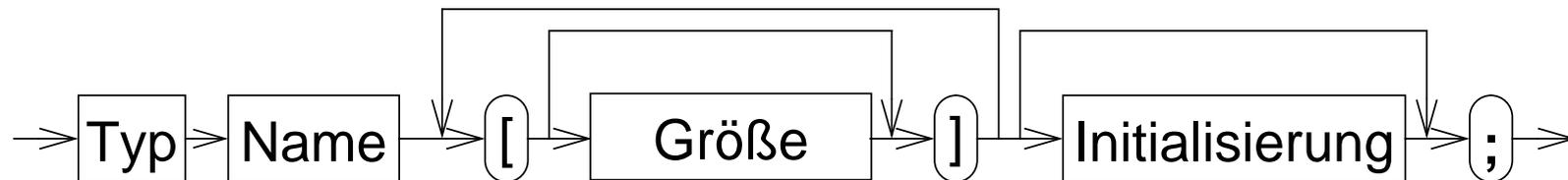
```
⑥ pointer++;
⑦ bptr++;
⑧ *pointer = 700;
```

```
⑨ array++;
```



10.3 Mehrdimensionale Felder

- neben eindimensionalen Felder kann man auch mehrdimensionale Felder vereinbaren
- Definition eines mehrdimensionalen Feldes



- Beispiel:

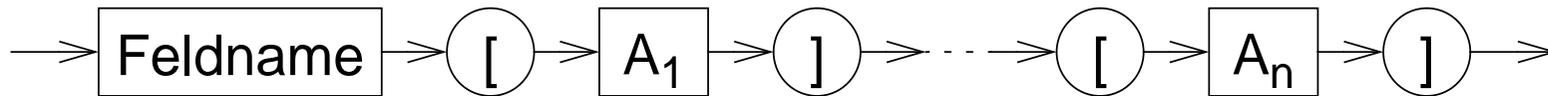
```
int matrix[4][4];
```

- Realisierung:

- in der internen Speicherung werden die Feldelemente zeilenweise hintereinander im Speicher abgelegt
- Felddefinition: `int f[2][2];`
Ablage der Elemente: `f[0][0]`, `f[0][1]`, `f[1][0]`, `f[1][1]`
`f` ist ein Zeiger auf `f[0][0]`

10.4 Zugriffe auf Feldelemente bei mehrdim. Feldern

■ Indizierung:



wobei: $0 \leq A_i < \text{Größe der Dimension } i \text{ des Feldes}$
 $n = \text{Anzahl der Dimensionen des Feldes}$

■ Beispiel:

```
int feld[5][8];
feld[2][3] = 10;
```

◆ ist äquivalent zu:

```
int feld[5][8];
int *f1;
f1 = (int*)feld;
f1[2*8 + 3] = 10;
oder
*(f1 + (2*8 + 3)) = 10;
```

10.5 Initialisierung eines mehrdimensionalen Feldes

- ein mehrdimensionales Feld kann - wie ein eindimensionales Feld - durch eine Liste von konstanten Werten, die durch Komma getrennt sind, initialisiert werden
- wird die explizite Felddimensionierung weggelassen, so bestimmt die Zahl der Initialisierungskonstanten die Größe des Feldes
- Beispiel:

```
int feld[3][4] = {  
    { 1, 3, 5, 7}, /* feld[0][0-3] */  
    { 2, 4, 6     } /* feld[1][0-2] */  
};
```

`feld[1][3]` und `feld[2][0-3]` werden in dem Beispiel mit 0 initialisiert!

Dynamische Speicherverwaltung

- Felder können (mit einer Ausnahme im C99-Standard) nur mit statischer Größe definiert werden
- Wird die Größe eines Feldes erst zur Laufzeit des Programm bekannt, kann der benötigte Speicherbereich dynamisch vom Betriebssystem angefordert werden: Funktion `malloc`
 - Ergebnis: Zeiger auf den Anfang des Speicherbereichs
 - Zeiger kann danach wie ein Feld verwendet werden (`[]`-Operator)
- `void *malloc(size_t size)`

```
int *feld;
int groesse;
...
feld = (int *) malloc(groesse * sizeof(int));
if (feld == NULL) {
    perror("malloc feld");
    exit(1);
}
for (i=0; i<groesse; i++) { feld[i] = 8; }
...
```

cast-Operator (points to `(int *)`)

sizeof-Operator (points to `sizeof(int)`)

Dynamische Speicherverwaltung (2)

- Dynamisch angeforderte Speicherbereiche können mit der **free**-Funktion wieder freigegeben werden

- **void free(void *ptr)**

```
double *dfeld;  
int groesse;  
...  
dfeld = (double *) malloc(groesse * sizeof(double));  
...  
free(dfeld);
```

- die Schnittstellen der Funktionen sind in in der include-Datei `stdlib.h` definiert

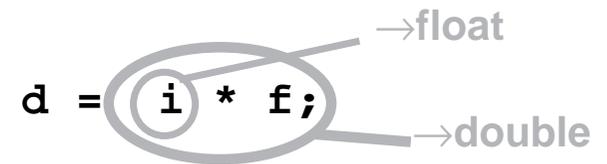
```
#include <stdlib.h>
```

Explizite Typumwandlung — Cast-Operator

- C enthält Regeln für eine automatische Konvertierung unterschiedlicher Typen in einem Ausdruck

Beispiel:

```
int i = 5;
float f = 0.2;
double d;
```



- In manchen Fällen wird eine explizite Typumwandlung benötigt (vor allem zur Umwandlung von Zeigern)

◆ Syntax:

(Typ) Variable

Beispiele:

```
(int) a
(float) b
```

```
(int *) a
(char *) a
```

◆ Beispiel:

```
feld = (int *) malloc(groesse * sizeof(int));
```

malloc liefert Ergebnis vom Typ (void *)

cast-Operator macht daraus den Typ (int *)

sizeof-Operator

- In manchen Fällen ist es notwendig, die Größe (in Byte) einer Variablen oder Struktur zu ermitteln
 - z. B. zum Anfordern von Speicher für ein Feld (→ malloc)

- Syntax:

`sizeof x` liefert die Größe des Objekts `x` in Bytes

`sizeof (Typ)` liefert die Größe eines Objekts vom Typ `Typ` in Bytes

- Das Ergebnis ist vom Typ `size_t` (\equiv `int`)
(`#include <stddef.h>!`)

- Beispiel:

```
int a; size_t b;  
b = sizeof a;           /* ⇒ b = 2 oder b = 4 */  
b = sizeof(double)     /* ⇒ b = 8 */
```

Eindimensionale Felder als Funktionsparameter

- ganze Felder können in C **nicht *by-value*** übergeben werden
- wird einer Funktion ein Feldname als Parameter übergeben, wird damit der Zeiger auf das erste Element "by value" übergeben
 - ↳ die Funktion kann über den formalen Parameter (=Kopie des Zeigers) in gleicher Weise wie der Aufrufer auf die Feldelemente zugreifen (und diese verändern!)
- bei der Deklaration des formalen Parameters wird die Feldgröße weggelassen
 - die Feldgröße ist automatisch durch den aktuellen Parameter gegeben
 - die Funktion kennt die Feldgröße damit nicht
 - ggf. ist die Feldgröße über einen weiteren `int`-Parameter der Funktion explizit mitzuteilen
 - die Länge von Zeichenketten in `char`-Feldern kann normalerweise durch Suche nach dem `\0`-Zeichen bestimmt werden

Eindimensionale Felder als Funktionsparameter (2)

- wird ein Feldparameter als `const` deklariert, können die Feldelemente innerhalb der Funktion nicht verändert werden
- Funktionsaufruf und Deklaration der formalen Parameter am Beispiel eines `int`-Feldes:

```
int a, b;
int feld[20];
func(a, feld, b);
...
int func(int p1, int p2[], int p3);
oder:
int func(int p1, int *p2, int p3);
```

- die Parameter-Deklarationen `int p2[]` und `int *p2` sind vollkommen äquivalent!
 - im Unterschied zu einer Variablendefinition


```
int f[] = {1, 2, 3}; // initialisiertes Feld mit 3 Elementen
int f1[];      // ohne Initialisierung oder Dimension nicht erlaubt!
int *p;          // Zeiger auf einen int
```

Eindimensionale Felder als Funktionsparameter (3)

■ **Beispiel 1:** Bestimmung der Länge einer Zeichenkette (*String*)

```
int strlen(const char string[])
{
    int i=0;
    while (string[i] != '\0') ++i;
    return(i);
}
```

Eindimensionale Felder als Funktionsparameter (4)

■ Beispiel 2: Konkateniere Strings

```
void strcat(char to[], const char from[])
{
    int i=0, j=0;
    while (to[i] != '\0') i++;
    while ( (to[i++] = from[j++]) != '\0' )
        ;
}
```

◆ Funktionsaufruf mit Feld-Parametern

- als aktueller Parameter beim Funktionsaufruf wird einfach der Feldname angegeben

```
char s1[50] = "text1";
char s2[] = "text2";
strcat(s1, s2); /* → s1= "text1text2" */
strcat(s1, "text3"); /* → s1= "text1text2text3" */
```

Zeiger, Felder und Zeichenketten

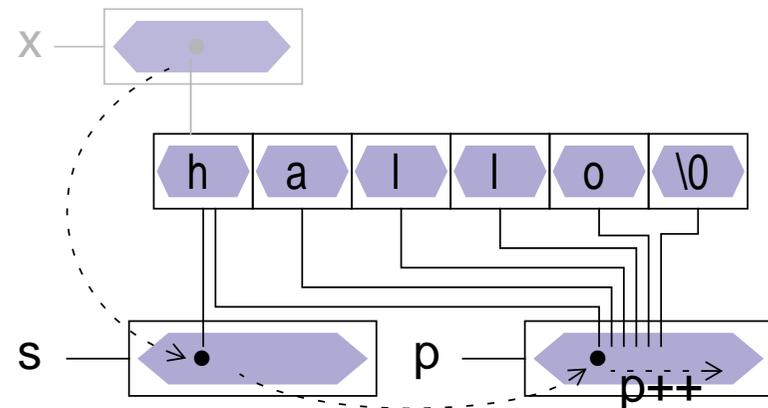
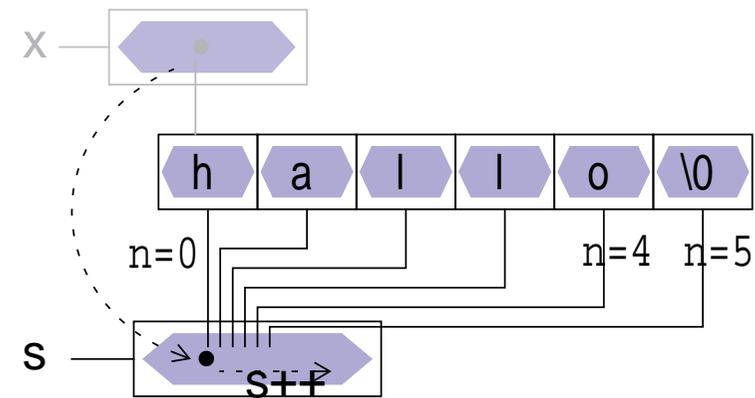
- Zeichenketten sind Felder von Einzelzeichen (`char`), die in der internen Darstellung durch ein `'\0'`-Zeichen abgeschlossen sind
- Beispiel: Länge eines Strings ermitteln — Aufruf `strlen(x)`;

```

/* 1. Version */
int strlen(const char *s)
{
    int n;
    for (n=0; *s != '\0'; s++)
        n++;
    return(n);
}
    
```

```

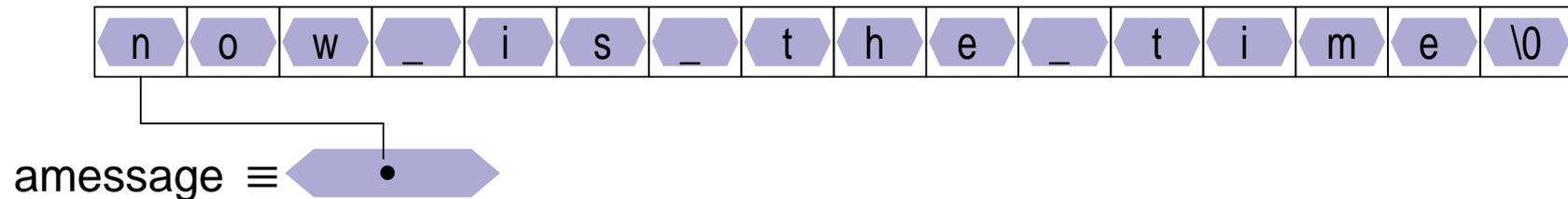
/* 2. Version */
int strlen(const char *s)
{
    char *p = s;
    while (*p != '\0')
        p++;
    return(p-s);
}
    
```



Zeiger, Felder und Zeichenketten (2)

- wird eine Zeichenkette zur Initialisierung eines char-Feldes verwendet, ist der Feldname ein konstanter Zeiger auf den Anfang der Zeichenkette

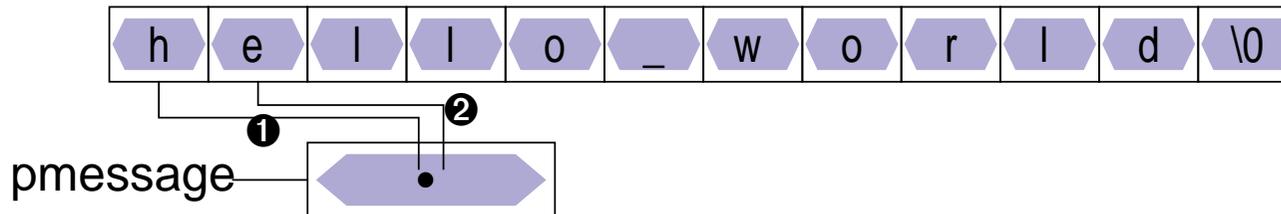
```
char amessage[] = "now is the time";
```



Zeiger, Felder und Zeichenketten (3)

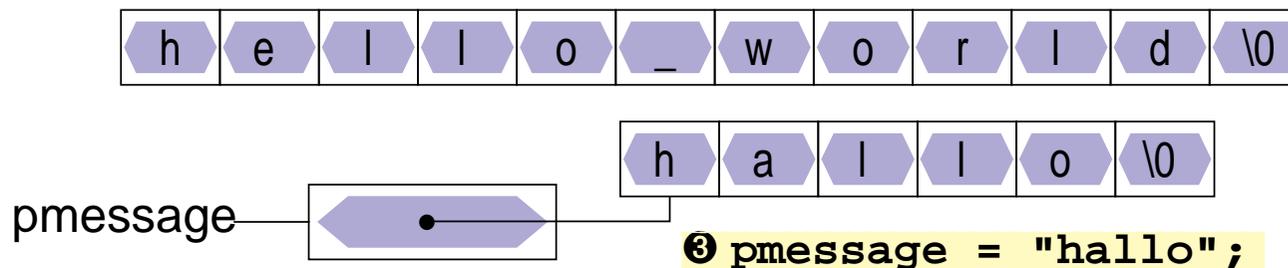
- wird eine Zeichenkette zur Initialisierung eines `char`-Zeigers verwendet, ist der Zeiger eine Variable, die mit der Anfangsadresse der Zeichenkette initialisiert wird

```
char *pmessage = "hello world";
```



```
pmessage++; ②  
printf("%s", pmessage); /* gibt "ello world" aus */
```

➔ wird dieser Zeiger überschrieben, ist die Zeichenkette nicht mehr adressierbar!

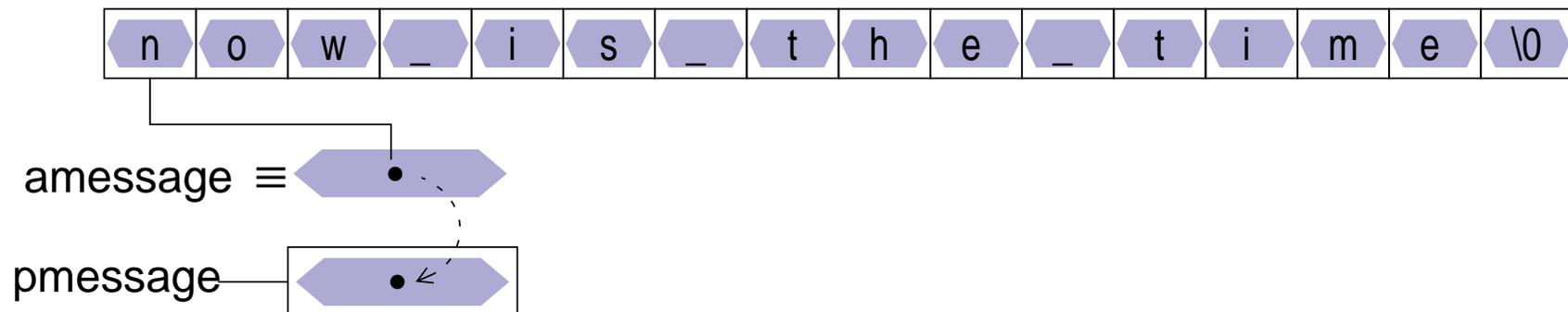


Zeiger, Felder und Zeichenketten (4)

- die Zuweisung eines `char`-Zeigers oder einer Zeichenkette an einen `char`-Zeiger bewirkt kein Kopieren von Zeichenketten!

```
pmessage = amessage;
```

weist dem Zeiger `pmessage` lediglich die Adresse der Zeichenkette `"now is the time"` zu



- wird eine Zeichenkette als aktueller Parameter an eine Funktion übergeben, erhält diese eine Kopie des Zeigers

Zeiger, Felder und Zeichenketten (5)

■ Zeichenketten kopieren

```
/* 1. Version */
void strcpy(char to[], const char from[])
{
    int i=0;
    while ( (to[i] = from[i]) != '\0' )
        i++;
}

/* 2. Version */
void strcpy(char *to, const char *from)
{
    while ( (*to = *from) != '\0' )
        to++, from++;
}

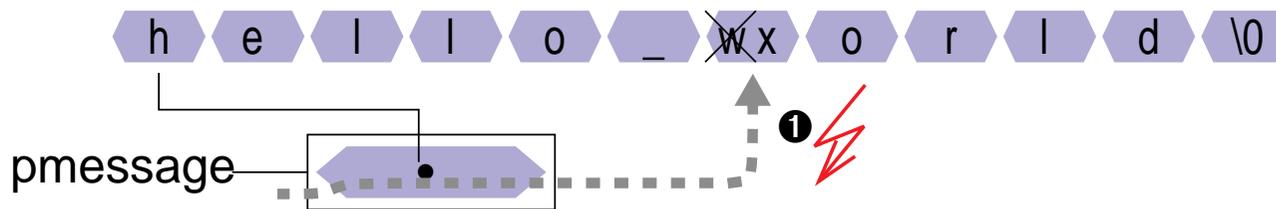
/* 3. Version */
void strcpy(char *to, const char *from)
{
    while ( *to++ = *from++ )
        ;
}
```

Zeiger, Felder und Zeichenketten (6)

- in ANSI-C können Zeichenketten in nicht-modifizierbaren Speicherbereichen angelegt werden (je nach Compiler)
 - ↳ Schreiben in Zeichenketten (Zuweisungen über dereferenzierte Zeiger) kann zu Programmabstürzen führen!
 - Beispiel:

```
strcpy("zu ueberschreiben", "reinschreiben");
```

```
char *pmessage = "hello world";
```



```
pmessage[6] = 'x';
```

aber!

```
char amessage[] = "hello world";
amessage[6] = 'x';
```

ok!

Felder von Zeigern

- Auch von Zeigern können Felder gebildet werden

- Deklaration

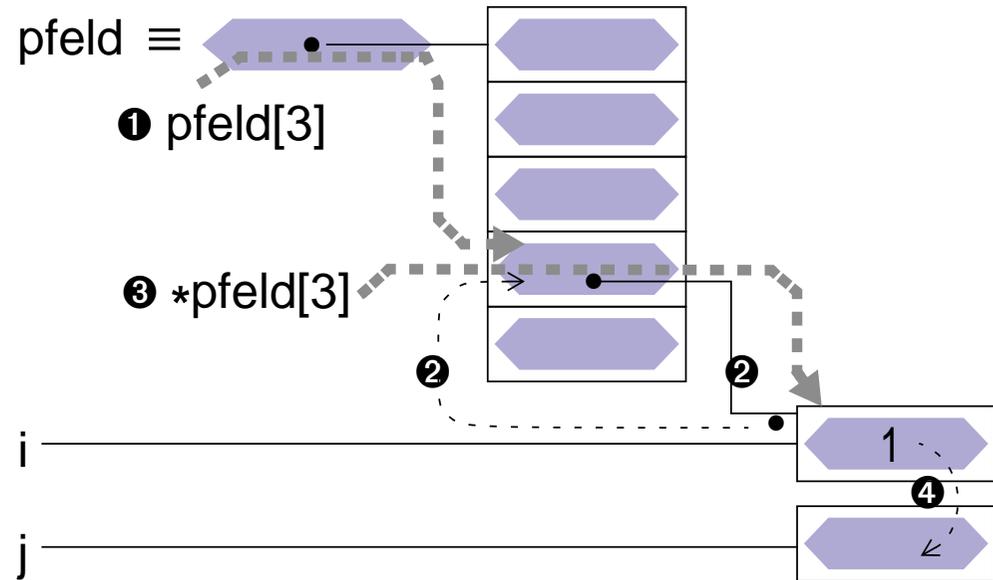
```
int *pfeld[5];
int i = 1;
int j;
```

- Zugriffe auf einen Zeiger des Feldes

```
pfeld[3] = &i; ②
```

- Zugriffe auf das Objekt, auf das ein Zeiger des Feldes verweist

```
j = *pfeld[3]; ④
```

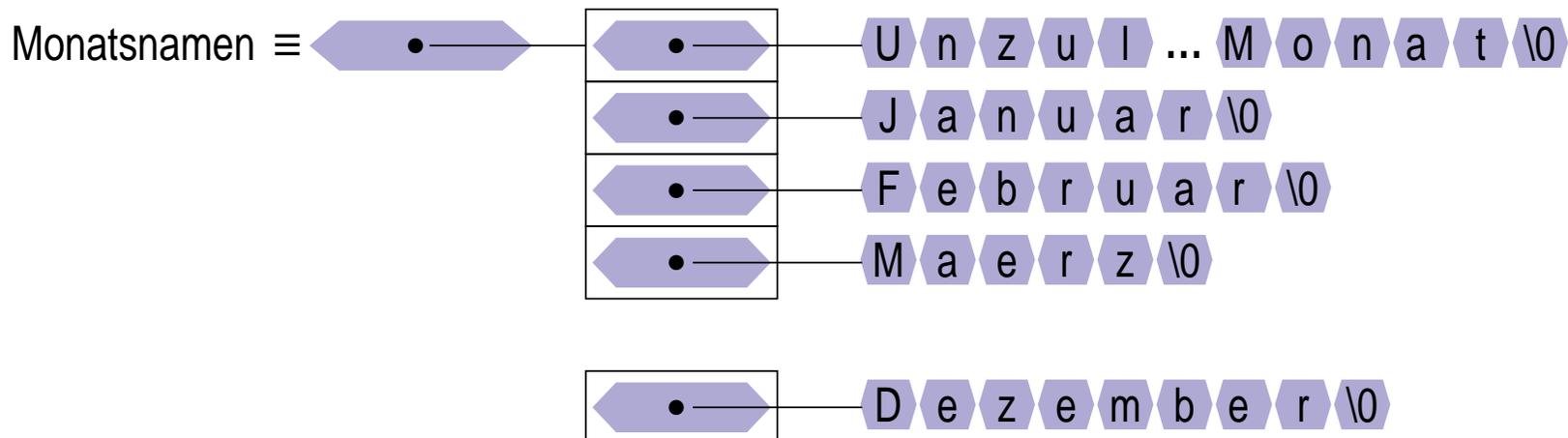


Felder von Zeigern (2)

- Beispiel: Definition und Initialisierung eines Zeigerfeldes:

```
char *month_name(int n)
{
    static char *Monatsnamen[] = {
        "Unzulaessiger Monat",
        "Januar",
        ...
        "Dezember"
    };

    return ( (n<0 || n>12) ?
             Monatsnamen[0] : Monatsnamen[n] );
}
```



Argumente aus der Kommandozeile

- beim Aufruf eines Kommandos können normalerweise Argumente übergeben werden
- der Zugriff auf diese Argumente wird der Funktion ***main()*** durch zwei Aufrufparameter ermöglicht:

```
int
main (int argc, char *argv[])
{
    ...
}
```

oder

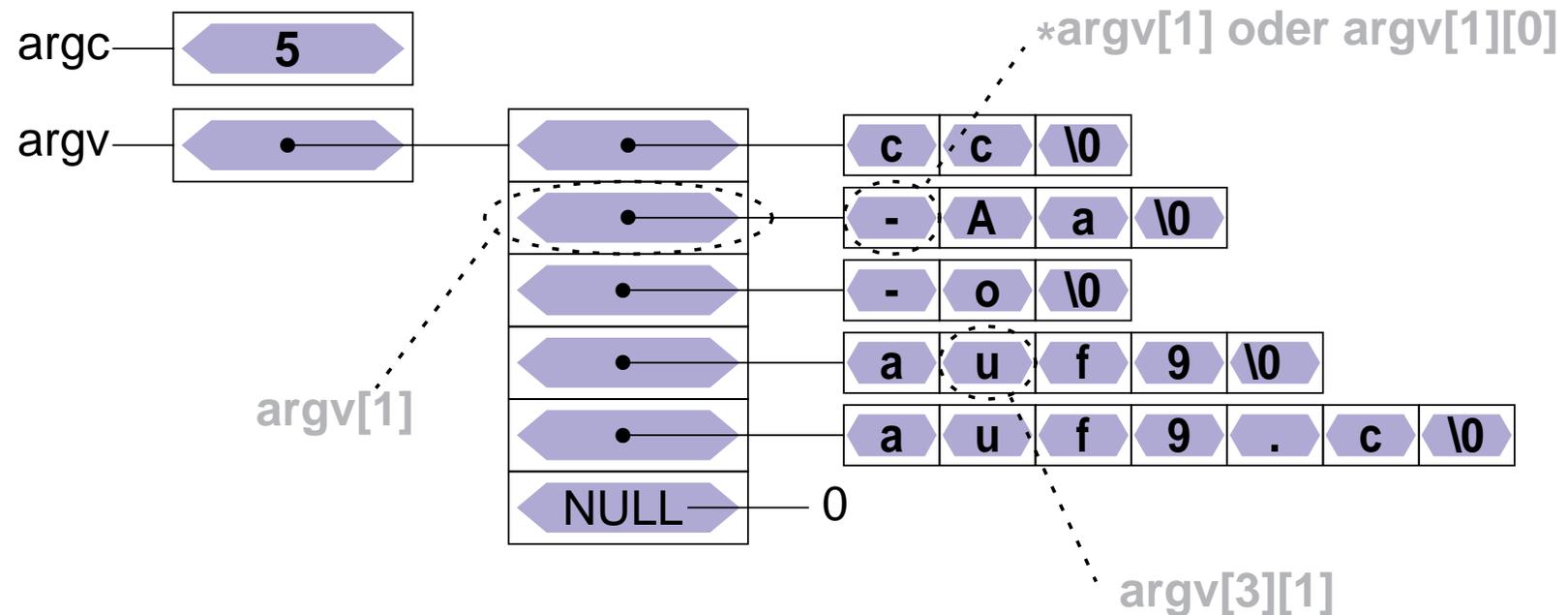
```
int
main (int argc, char **argv)
{
    ...
}
```

- der Parameter **`argc`** enthält die Anzahl der Argumente, mit denen das Programm aufgerufen wurde
- der Parameter **`argv`** ist ein Feld von Zeiger auf die einzelnen Argumente (Zeichenketten)
- der Kommandoname wird als erstes Argument übergeben (**`argv[0]`**)

17.1 Datenaufbau

Kommando: `cc -Aa -o auf9 auf9.c`

Datei cc.c:
`...`
`main(int argc, char *argv[]) {`
`...`



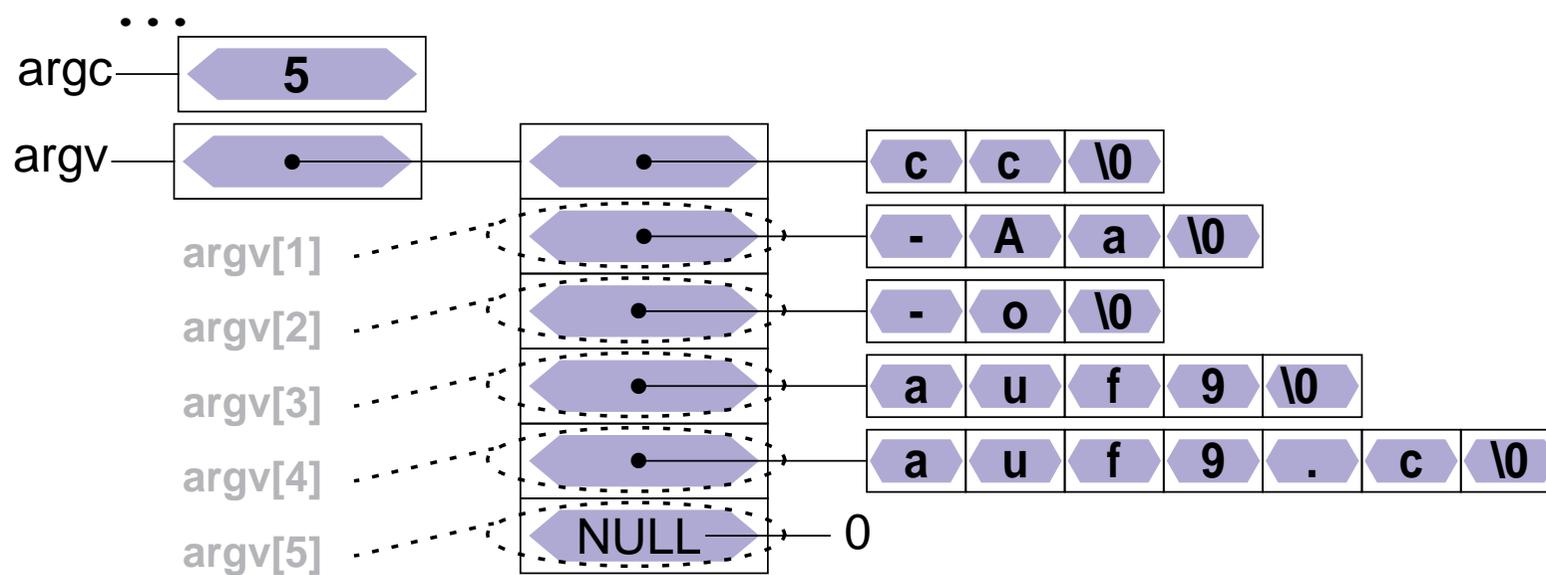
17.2 Zugriff — Beispiel: Ausgeben aller Argumente (1)

- das folgende Programmstück gibt alle Argumente der Kommandozeile aus (außer dem Kommandonamen)

```

int
main (int argc, char *argv[])
{
    int i;
    for ( i=1; i<argc; i++) {
        printf("%s%c", argv[i],
            (i < argc-1) ? ' ':'\n' );
    }
}
    
```

1. Version



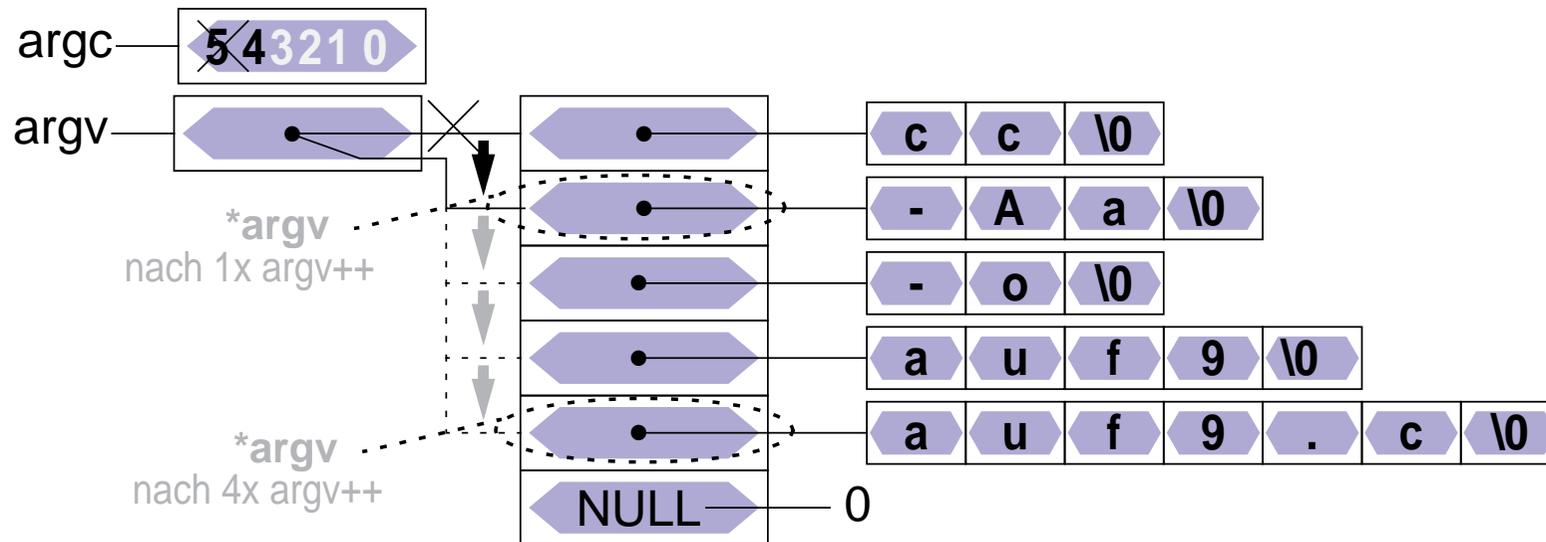
17.2 Zugriff — Beispiel: Ausgeben aller Argumente (2)

- das folgende Programmstück gibt alle Argumente der Kommandozeile aus (außer dem Kommandonamen)

```
int
main (int argc, char **argv)
{
    while (--argc > 0) {
        argv++;
        printf("%s%c", *argv, (argc>1) ? ' ' : '\n' );
    }
    ...
}
```

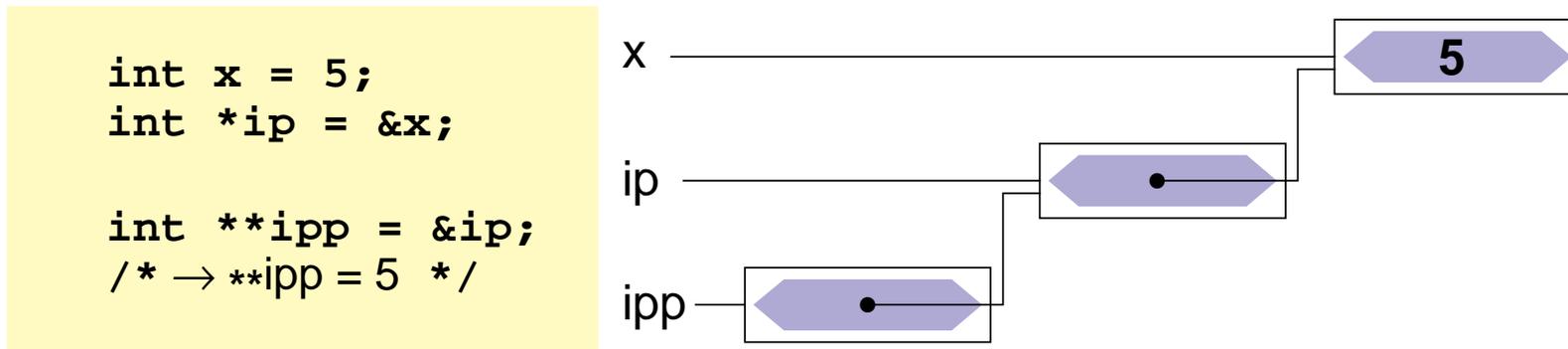
linksseitiger Operator:
erst dekrementieren,
dann while-Bedingung prüfen
→ Schleife läuft für argc=4,3,2,1

2. Version



Zeiger auf Zeiger

- ein Zeiger kann auf eine Variable verweisen, die ihrerseits ein Zeiger ist



- wird vor allem bei der Parameterübergabe an Funktionen benötigt, wenn ein Zeiger "call bei reference" übergeben werden muss (z. B. swap-Funktion für Zeiger)

Strukturen

■ Beispiele

```
struct student {  
    char nachname[25];  
    char vorname[25];  
    char gebdatum[11];  
    int matrnr;  
    short gruppe;  
    char best;  
};
```

```
struct komplex {  
    double re;  
    double im;  
};
```

- Initialisierung
- Strukturen als Funktionsparameter
- Felder von Strukturen
- Zeiger auf Strukturen

19.1 Initialisieren von Strukturen

- Strukturen können — wie Variablen und Felder — bei der Definition initialisiert werden
 - die Zuordnung zu den Komponenten erfolgt entweder aufgrund der Reihenfolge oder aufgrund des angegebenen Namens (in C++ nur aufgrund der Reihenfolge möglich!)
- Beispiele

```
struct student stud1 = {  
    "Meier", "Hans", "24.01.1970", 1533180, 5, 'n'  
};  
  
struct komplex c1 = {1.2, 0.8}, c2 = {.re=0.5, .im=0.33};
```

!!! **Vorsicht**

bei Zugriffen auf eine Struktur werden die Komponenten immer durch die Komponentennamen identifiziert,

bei der Initialisierung nach Reihenfolge aber nur durch die Position

➔ potentielle Fehlerquelle bei Änderungen der Strukturtyp-Deklaration