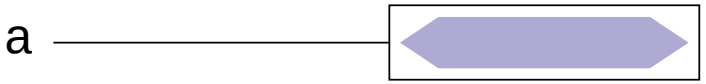
# Zeiger(-Variablen)

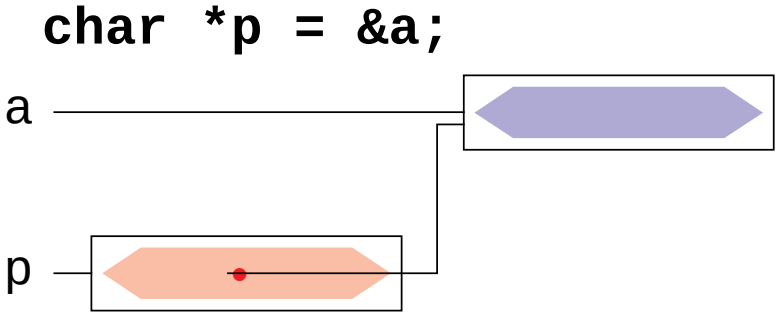
## 8.1 Einordnung

- **Konstante:**
  Bezeichnung für einen Wert


- **Variable:**
  Bezeichnung für ein Datenobjekt




- **Zeiger-Variable (Pointer):**
  Bezeichnung einer Referenz auf ein Datenobjekt



# 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 (indirekt) ihre Aufrufparameter 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

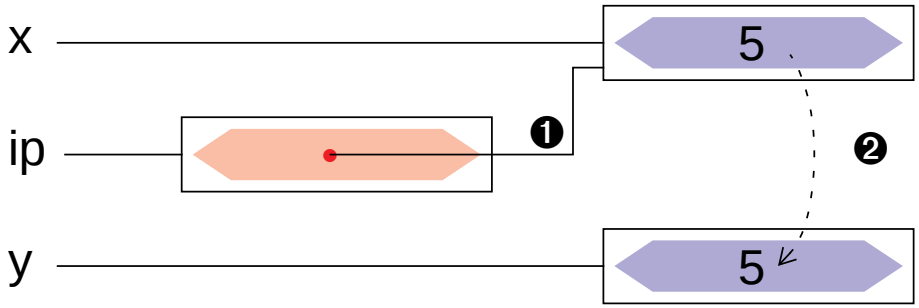
# 8.3 Definition von Zeigervariablen

- Syntax:

```
Typ *Name ;
```

- **Beispiele**

```
int x = 5;

int *ip;

int y;

ip = &x; ❶

y = *ip; ❷
```



# 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

- ★ Unterschied des Symbols *****
  in einer Variablendefinition und in einem Ausdruck

  - ➤ **`int *ip;`** ***** in einer Variablendefinition:
    **ip** ist eine Variable vom Typ **(int *)**,
    eine Variable die auf ein Objekt vom Typ **(int)** verweist

  - ➤ **`y = *ip;`** ***** als Operator in einem Ausdruck:
    **ip** ist eine Variable, die auf ein Objekt vom Typ **(int)** verweist,
    der Ausdruck ***ip** ermittelt den Inhalt dieses Objekts, also den int-Wert
    ➥ das Ergebnis des Ausdrucks ***ip** ist ein Wert vom Typ (int)

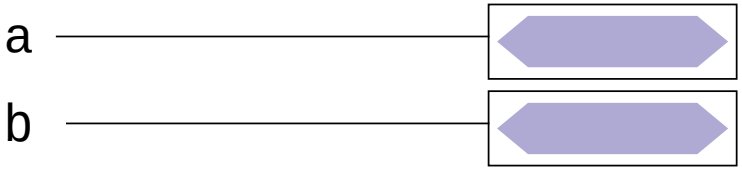
# 8.5 Zeiger als Funktionsargumente

- 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*

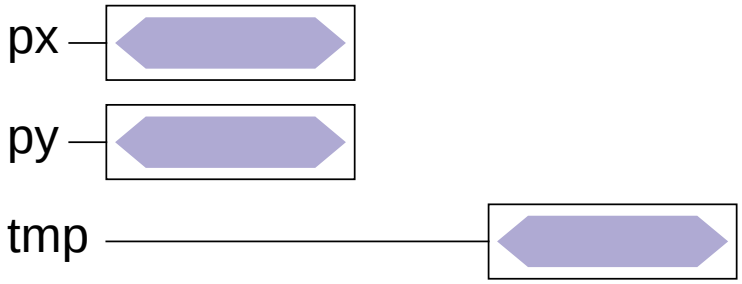
# 8.5 Zeiger als Funktionsargumente (2)

- Beispiel:

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



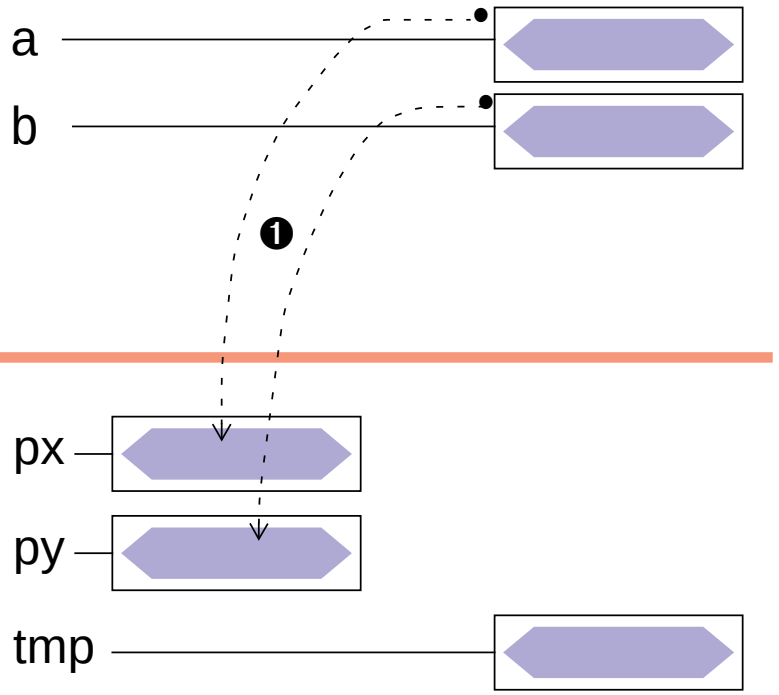

```
void swap (int *px, int *py)
{
    int tmp;

    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;

}
```

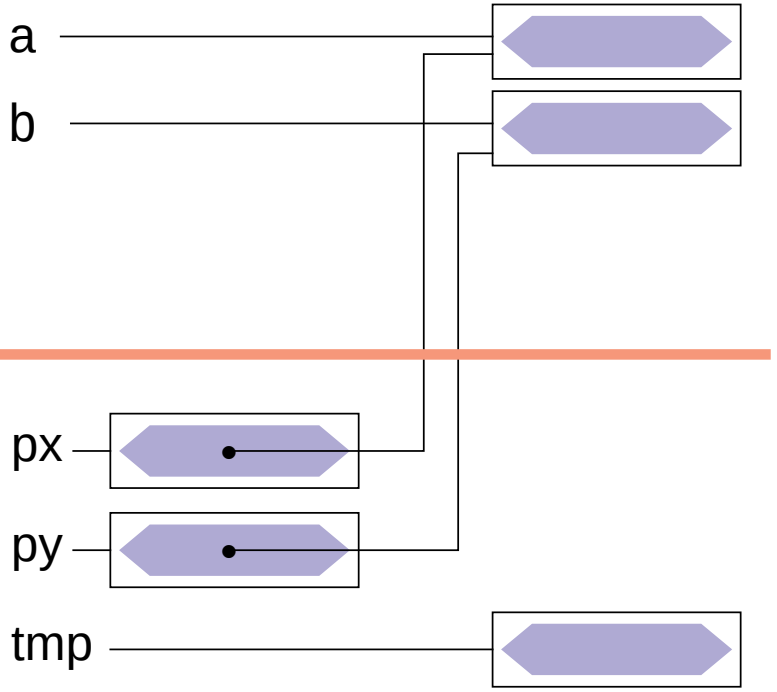


# 8.5 Zeiger als Funktionsargumente (2)

- Beispiel:

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

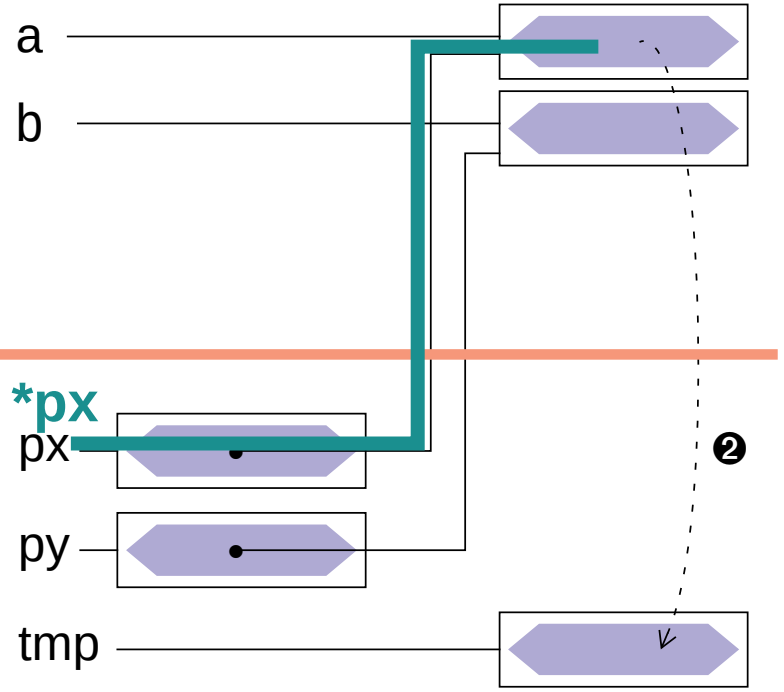
```
void swap (int *px, int *py)
{
    int tmp;

    tmp = *px;
    *px = *py;
    *py = tmp;

}
```



# 8.5 Zeiger als Funktionsargumente (2)

- Beispiel:

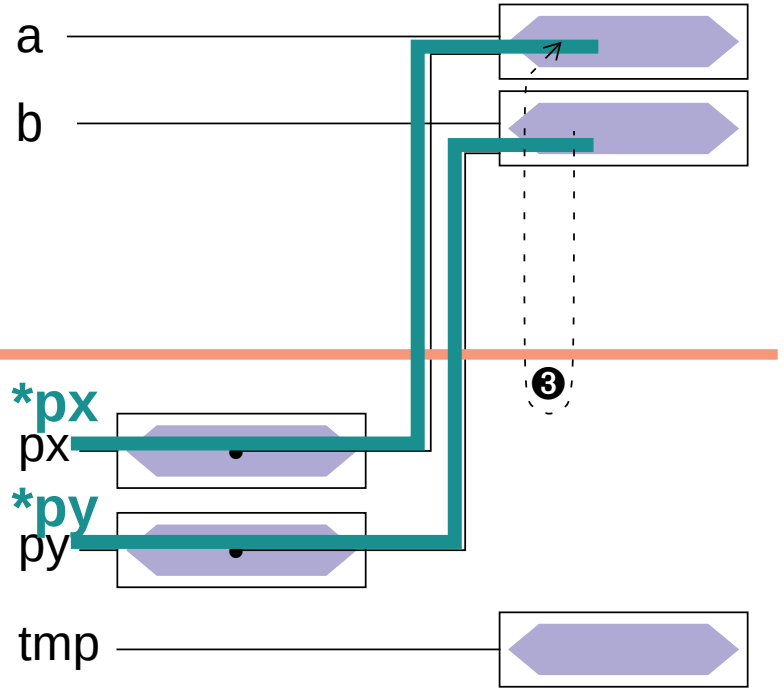
```
void swap (int *, int *);
int main() {
    int a, b;
    ...
    swap(&a, &b);
    ...
}

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

    tmp = *px; ❷
    *px = *py;
    *py = tmp;

}
```



# 8.5 Zeiger als Funktionsargumente (2)

- Beispiel:

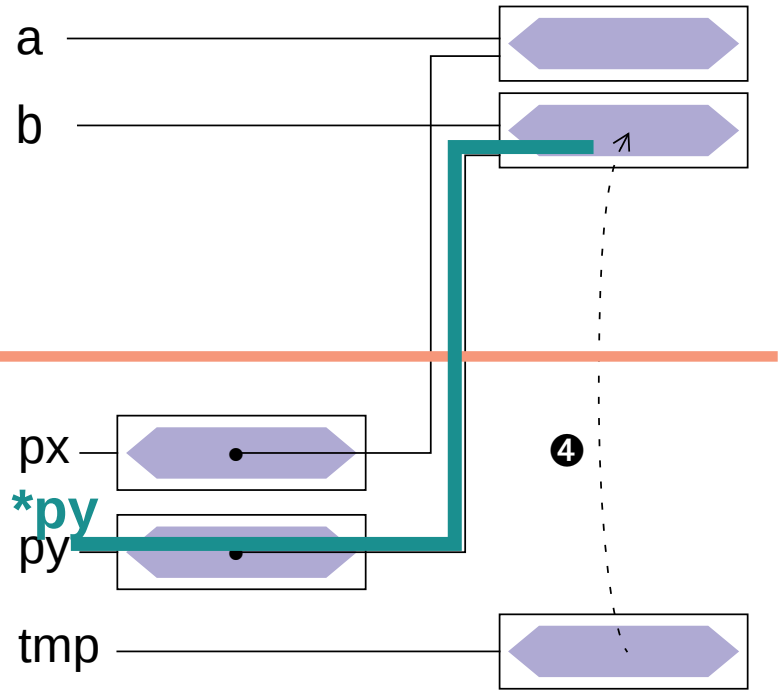
```
void swap (int *, int *);
int main() {
    int a, b;
    ...
    swap(&a, &b);
    ...
}

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

    tmp = *px;
    *px = *py; ❸
    *py = tmp;

}
```



# 8.5 Zeiger als Funktionsargumente (2)

- Beispiel:

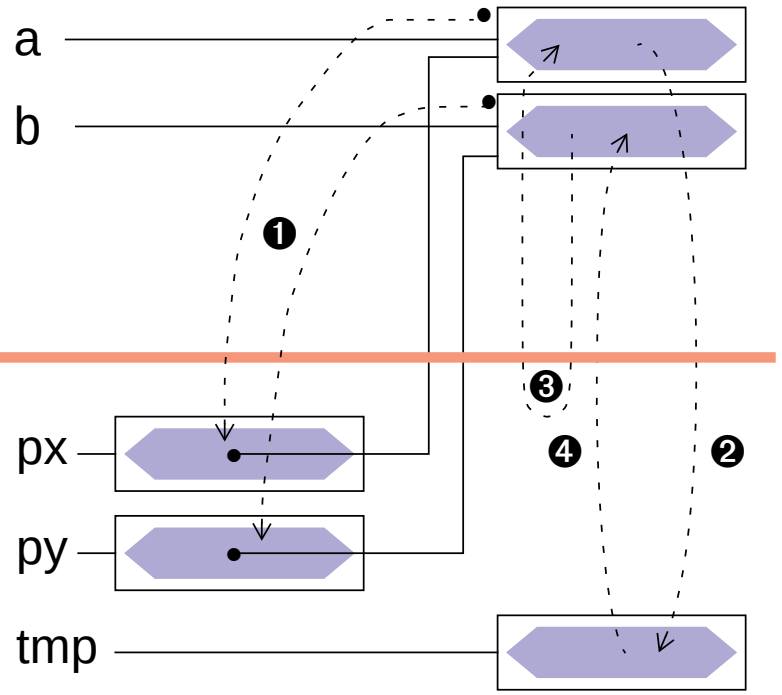
```
void swap (int *, int *);
int main() {
    int a, b;
    ...
    swap(&a, &b);
    ...
}

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

    tmp = *px;
    *px = *py;
    *py = tmp; ❹

}
```



# 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; ❷
    *px = *py; ❸
    *py = tmp; ❹

}
```



# 8.6 Zeiger auf Strukturen

- Konzept analog zu "Zeiger auf Variablen"
  - ➤ Adresse einer Struktur mit &-Operator zu bestimmen
- 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;
  ```
  nicht so get leserlich!

- Syntaktische Verschönerung

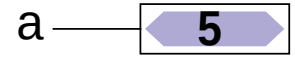
  ➥ `->`-Operator

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

# 8.7 Zusammenfassung

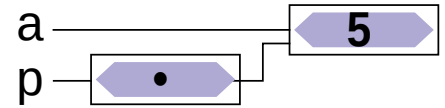
- Variable

```
int a;
```



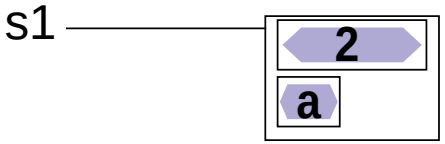

- Zeiger

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



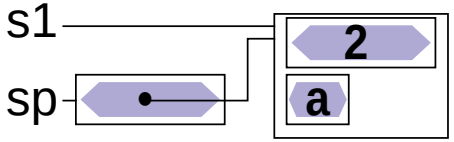

- Struktur

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




- Zeiger auf Struktur

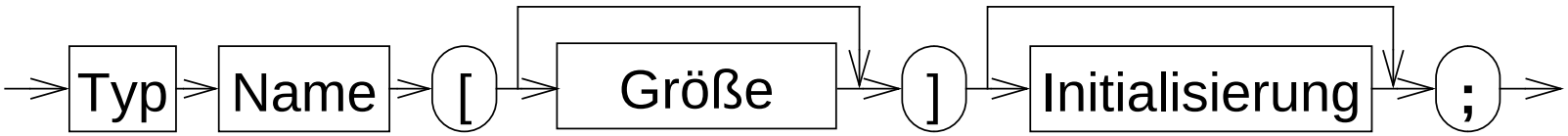
```
struct s *sp = &s1;
```



# Felder

## 9.1 Eindimensionale Felder

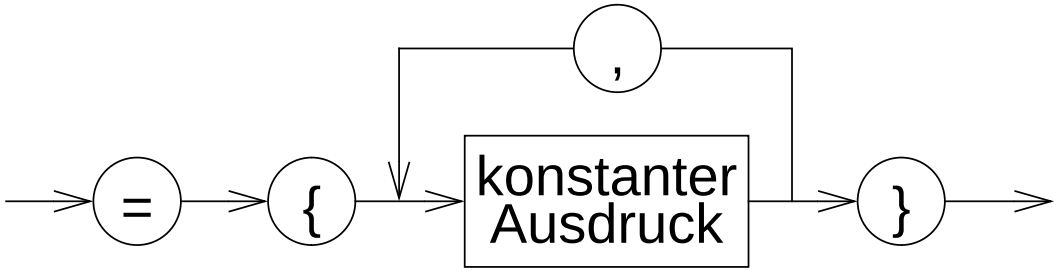
- eine Reihe von Daten desselben Typs kann zu einem **Feld** zusammengefasst werden
- bei der Definition wird die Größe des Felds angegeben
  - ➤ Größe muss eine Konstante sein
  - ➤ ab C99 bei lokalen Feldern auch zur Laufzeit berechnete Werte zulässig
- der Zugriff auf die Elemente erfolgt durch **Indizierung**, 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] = {'O', '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[] = {'O', 't', 't', 'o', '\0'};
```

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

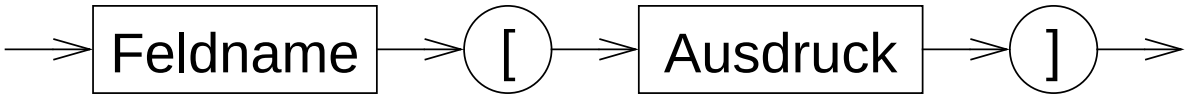
# 9.3 Initialisierung eines Feldes (2)

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

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

# 9.4 Zugriffe auf Feldelemente

- Indizierung:




wobei: 0 <= Wert(Ausdruck) < Feldgröße

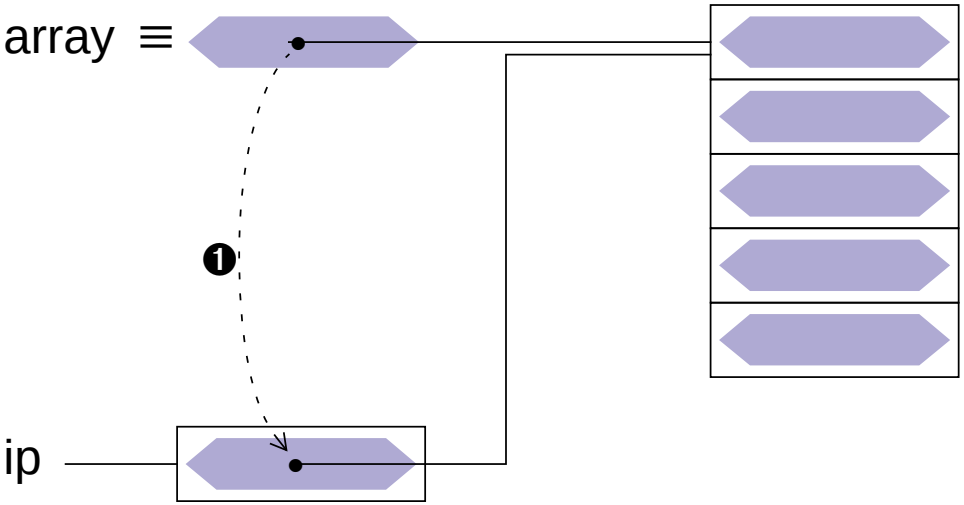
- Beispiele:

```
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:

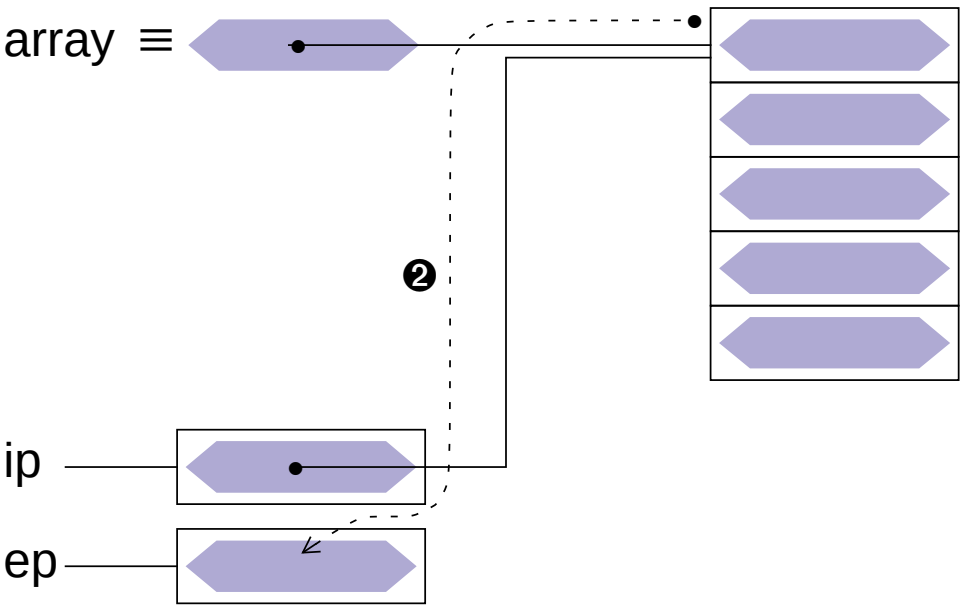
```
int array[5];

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:

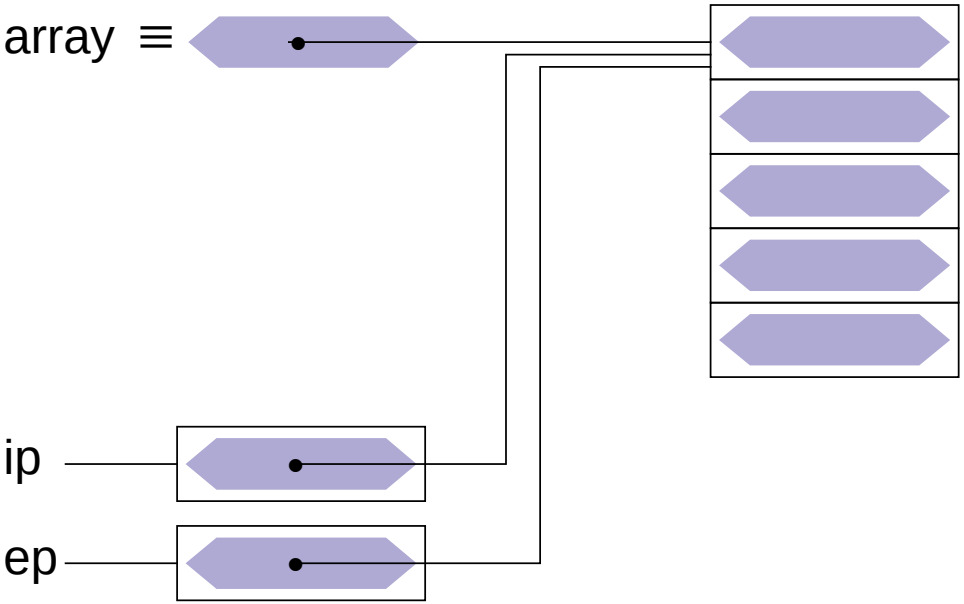
```
int array[5];

int *ip = array;

int *ep;
ep = &array[0]; ❷
```



# 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:

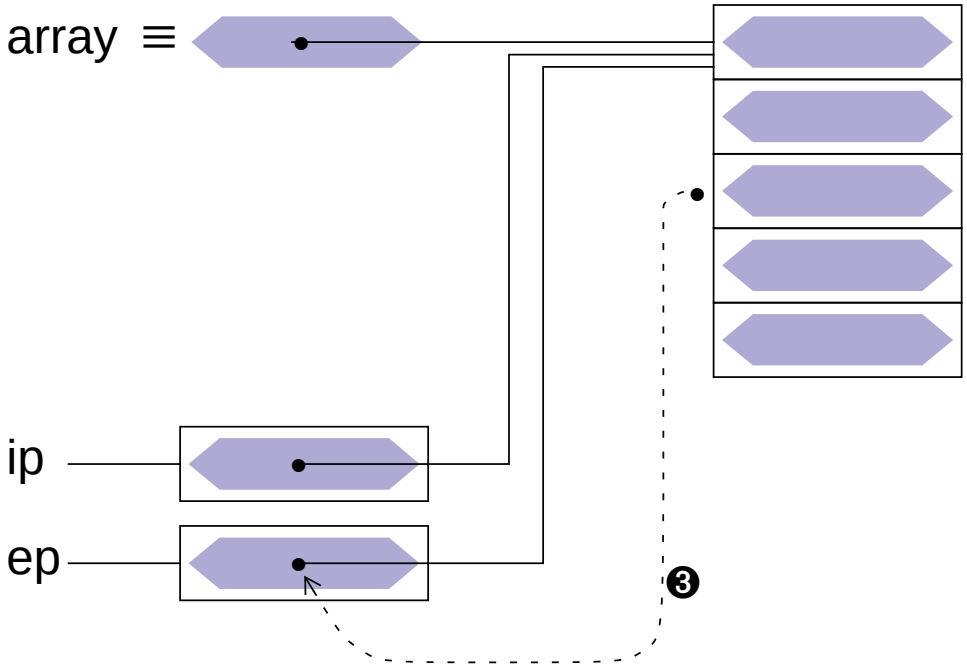
```
int array[5];

int *ip = array;

int *ep;
ep = &array[0]; ❷
```



# 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:

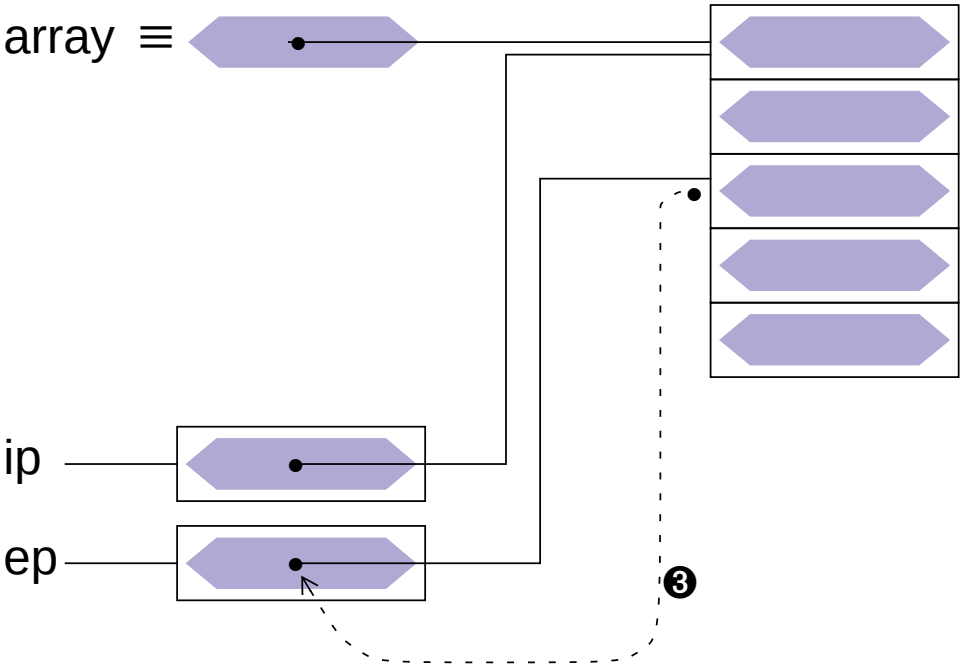
```
int array[5];

int *ip = array;

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

ep = &array[2]; ❸
```



# 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:

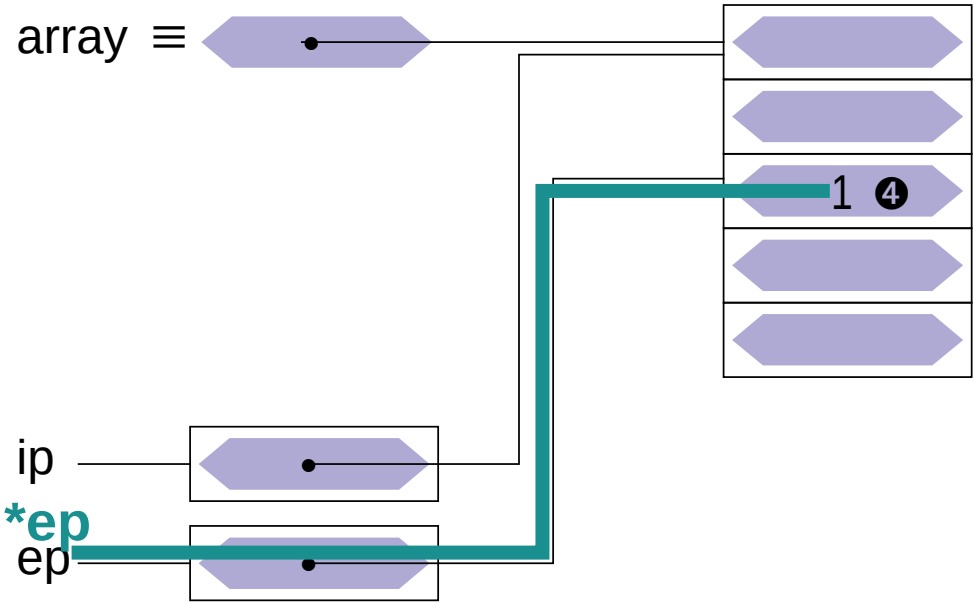
```
int array[5];

int *ip = array;

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

ep = &array[2]; ❸
```



# 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:

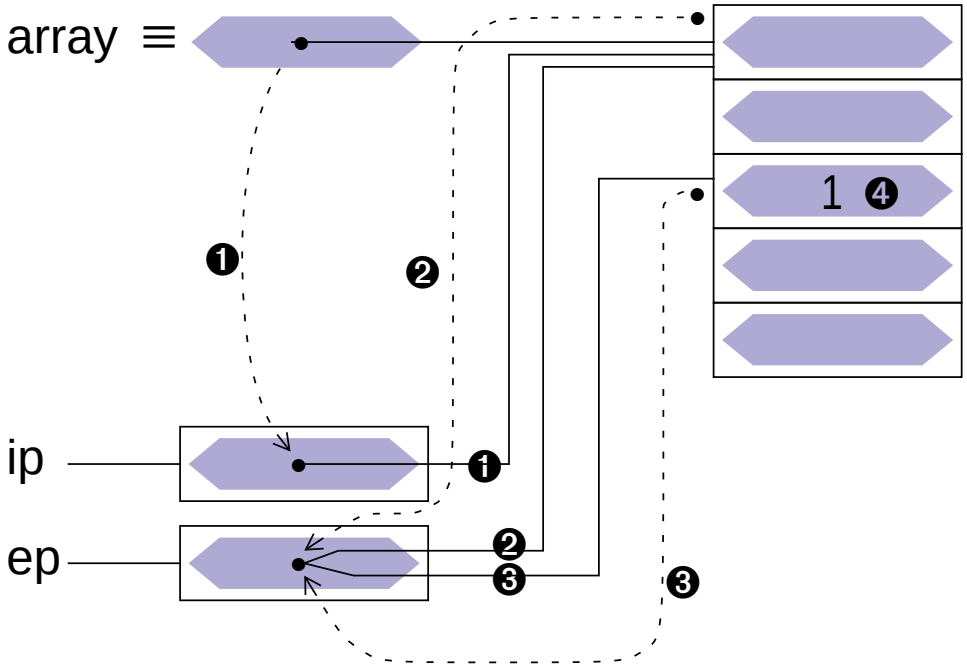
```
int array[5];

int *ip = array;

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

ep = &array[2];

*ep = 1; ❹
```



# 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:

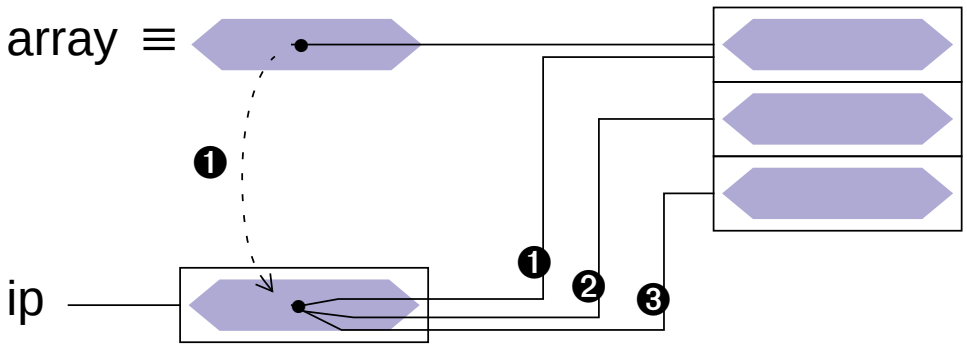
```
int array[5];

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
```

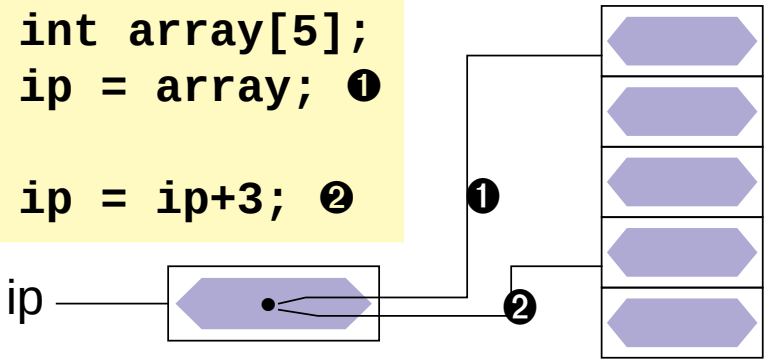



- **--** -Operator: Dekrement = vorheriges Objekt

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

  Dabei wird immer die Größe des Objekttyps berücksichtigt!

```
int array[5];
ip = array; ❶

ip = ip+3; ❷
```




!!! **Achtung:** Assoziativität der Operatoren beachten

# 10.2 Zeigerarithmetik und Felder

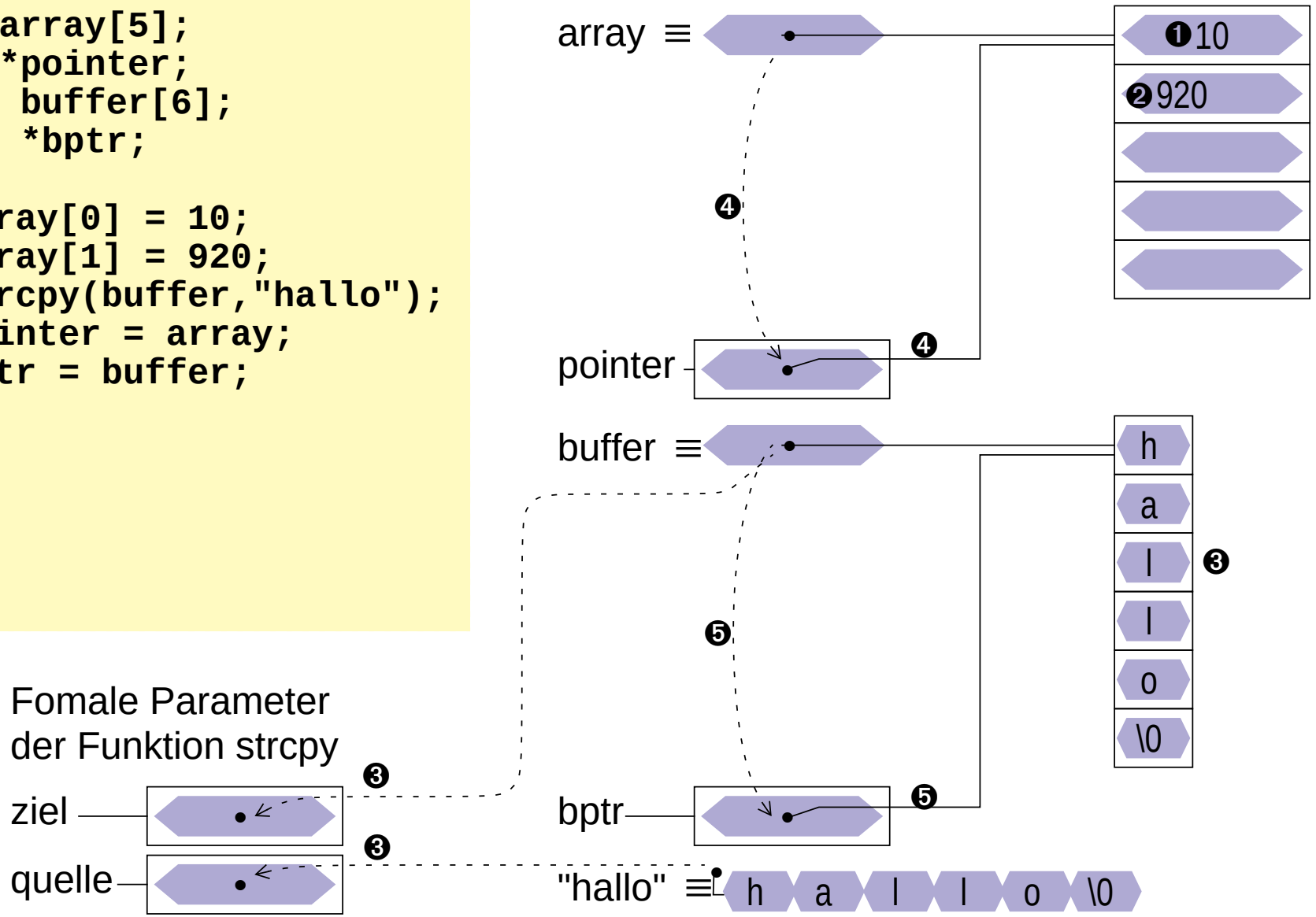
- Ein Feldname ist eine Konstante für die Adresse des Feldanfangs
  - ➥ Feldname ist ein ganz normaler Zeiger
    - ➤ Operatoren für Zeiger anwendbar ( *****, **[]** )
  - ➥ aber keine Variable ➔ keine Modifikationen erlaubt
    - ➤ keine Zuweisung, kein **++**, **--**, **+=**, …

- 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;
```

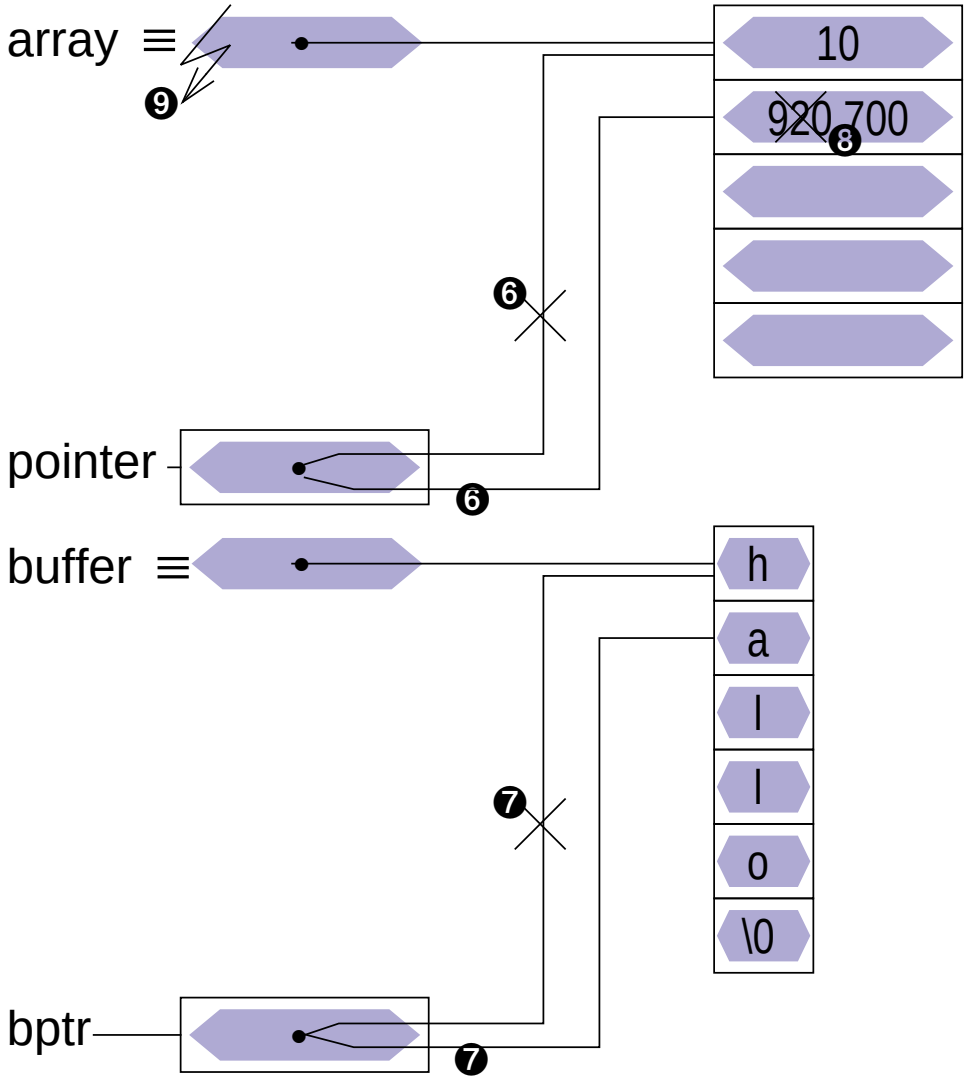
# 10.2 Zeigerarithmetik und Felder

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

❶ array[0] = 10;
❷ array[1] = 920;
❸ strcpy(buffer,"hallo");
❹ pointer = array;
❺ bptr = buffer;
```



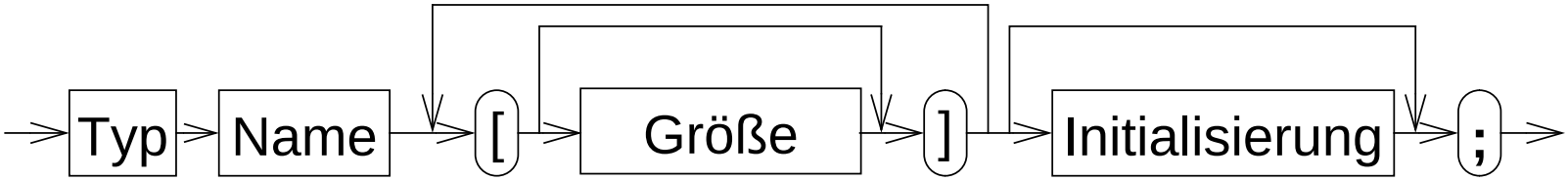
# 10.2 Zeigerarithmetik und Felder

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

❶ array[0] = 10;
❷ array[1] = 920;
❸ strcpy(buffer,"hallo");
❹ pointer = array;
❺ bptr = buffer;

❻ pointer++;
❼ bptr++;
❽ *pointer = 700;

❾ array++;
```



# 10.3 Mehrdimensionale Felder

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



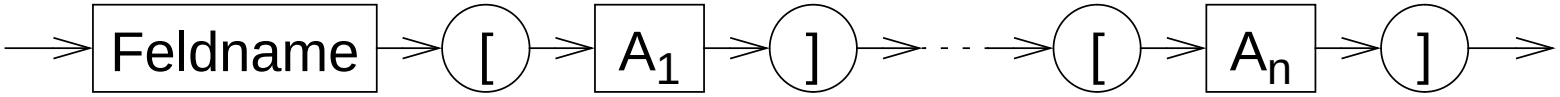

- Beispiel:

```
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: $0 <= A_i <$ Größe der Dimension i des Feldes
$n =$ 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

sizeof-Operator

# 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

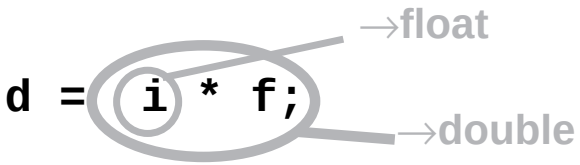
**`#include <stdlib.h>`**

# 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`** (≡ **`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

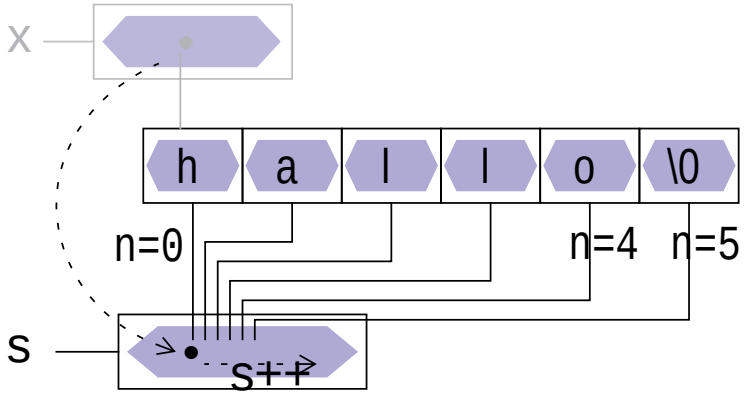
```
char s1[50] = "text1";
char s2[] = "text2";
strcat(s1, s2);/* → s1= "text1text2" */
strcat(s1, "text3");/* → s1= "text1text2text3" */
```

# Zeiger, Felder und Zeichenketten

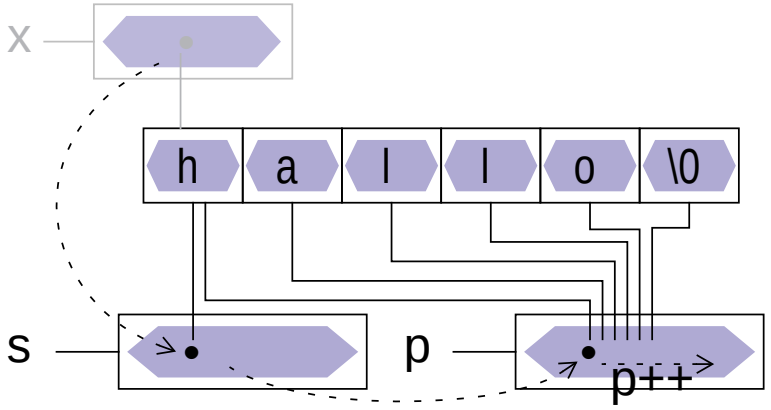
- Zeichenketten sind Felder von Einzelzeichen (**`char`**), die in der internen Darstellung durch ein **`'\0'`**–Zeichen abgeschlossen sind

- 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);
}
```



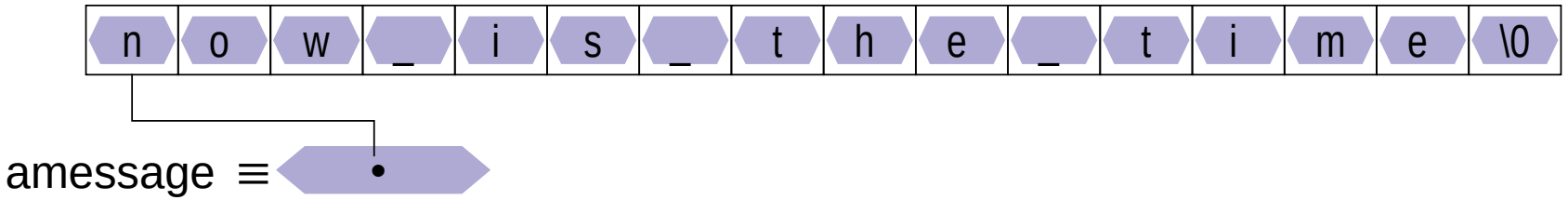

```
/* 2. Version */
int strlen(const char *s)
{
    char *p = s;
    while (*p != '\0')
        p++;
    return(p-s);
}
```



# 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

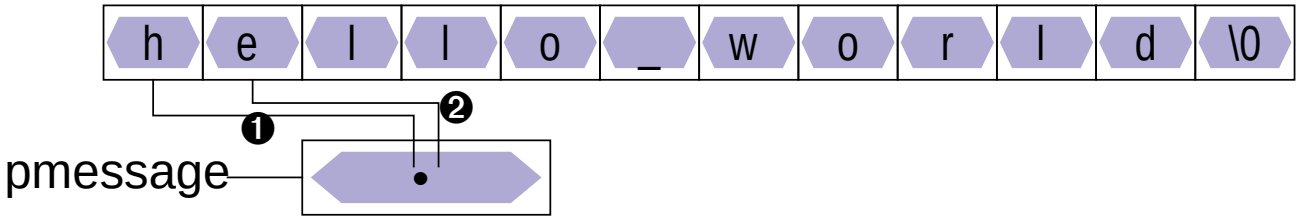
```
char amessage[] = "now is the time";
```



# Zeiger, Felder und Zeichenketten (3)

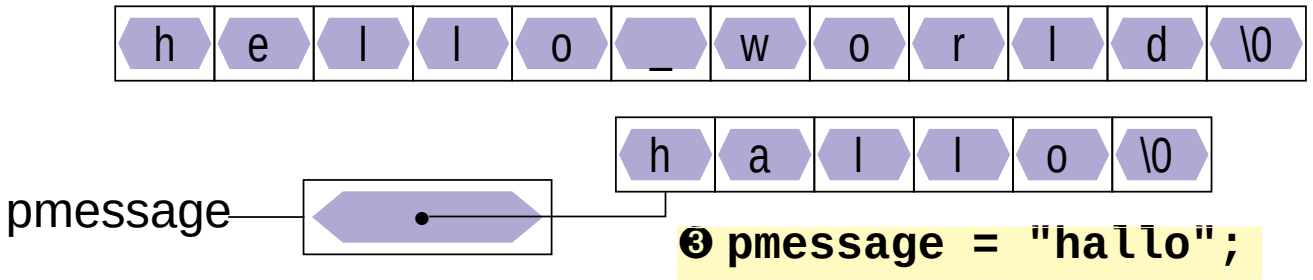
- wird eine Zeichenkette zur Initialisierung eines **`char`**–Zeigers verwendet, ist der Zeiger eine Variable, die mit der Anfangsadresse der Zeichenkette initialisiert wird

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




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

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

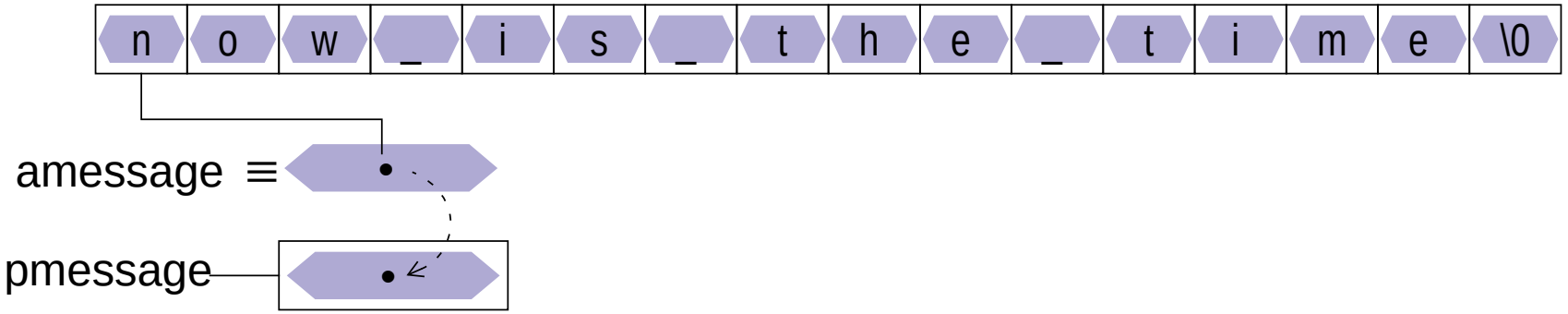


# Zeiger, Felder und Zeichenketten (4)

- 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

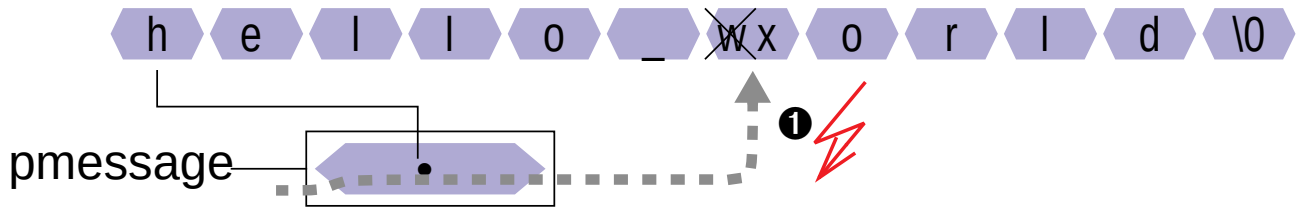
```
/* 1. Version */
void strcpy(char to[], const char from[])
{
    int i=0;
    while ( (to[i] = from[i]) != '\0' )
        i++;
}

/* 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'; ❶
```

**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; ❷
❶
```

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

```
j = *pfeld[3]; ❹
    ❶
  ❸
❹
```