

F Zeiger, Felder und Strukturen in C

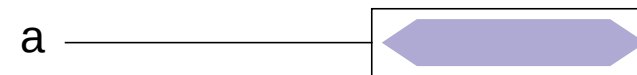
F.1 Zeiger(-Variablen)

1 Einordnung

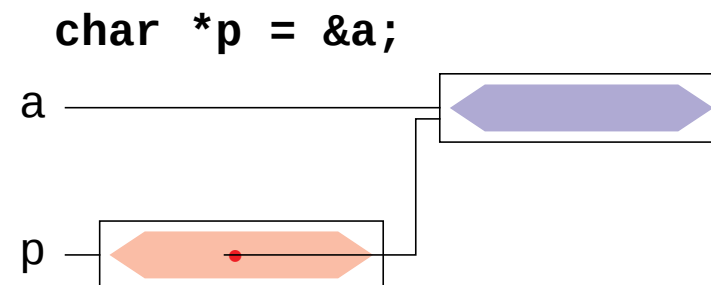
- **Konstante:**
Bezeichnung für einen Wert

'a' ≡ 0110 0001

- **Variable:**
Bezeichnung eines Datenobjekts



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



2 Überblick

- Eine Zeigervariable (***pointer***) enthält als Wert einen Verweis auf den Inhalt einer anderen Variablen
 - ↳ *der Zeiger verweist auf die Variable*
- Über diese Adresse kann man **indirekt** auf die andere 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

3 Definition von Zeigervariablen

■ Syntax:

```
Typ *Name ;
```

4 Beispiele

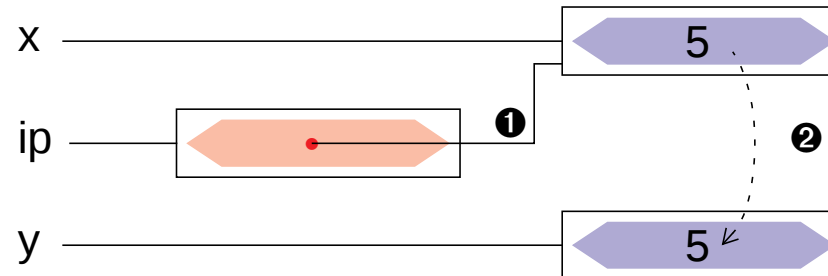
```
int x = 5;
```

```
int *ip;
```

```
int y;
```

```
ip = &x; ❶
```

```
y = *ip; ❷
```



5 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

6 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*

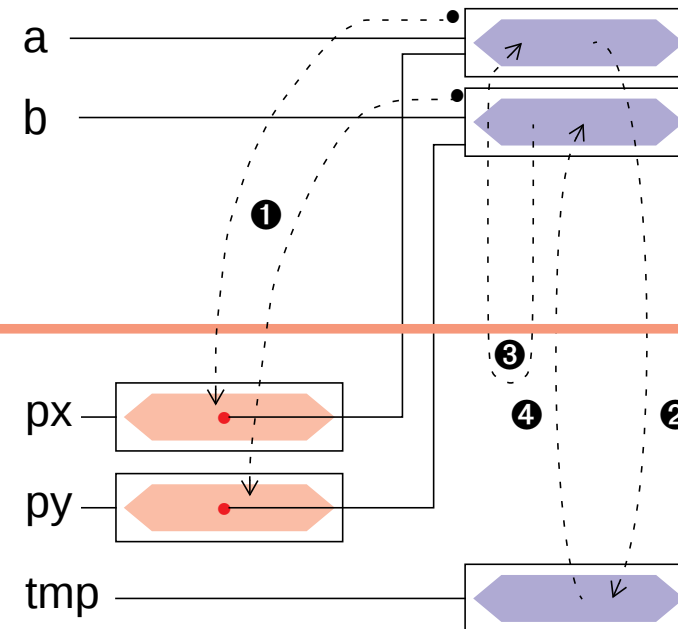
6 ... Zeiger als Funktionsargumente (2)

■ Beispiel:

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

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

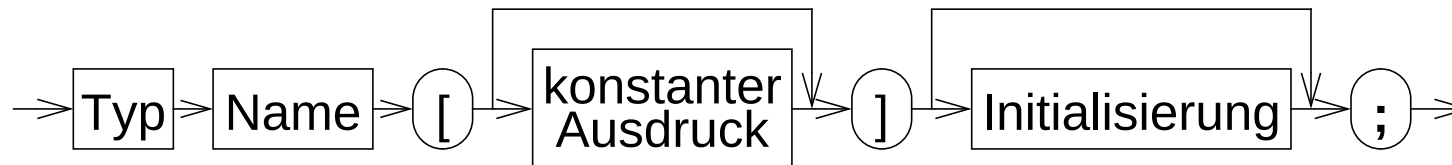
    tmp = *px; ❷
    *px = *py; ❸
    *py = tmp; ❹
}
```



F.2 Felder

1 Eindimensionale Felder

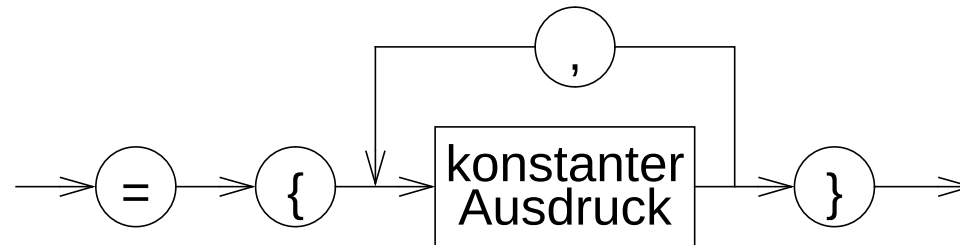
- eine Reihe von Daten desselben Typs kann zu einem **Feld** zusammengefasst werden
- bei der Definition wird die Anzahl der Feldelemente angegeben, die Anzahl ist konstant!
- der Zugriff auf die Elemente erfolgt durch **Indizierung**, beginnend bei Null
- Definition eines Feldes



- Beispiele:

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

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

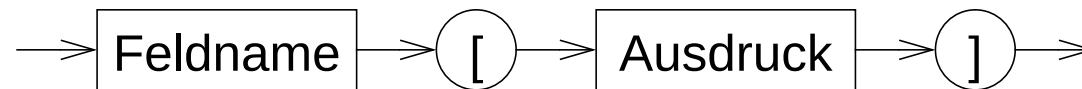
2 ... Initialisierung eines Feldes (2)

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

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

3 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'

```

■ Beispiel Vektoraddition:

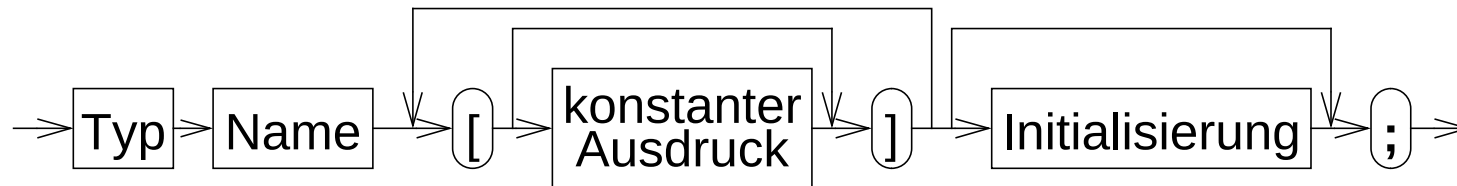
```

float v1[4], v2[4], sum[4];
int i;
...
for ( i=0; i < 4; i++ )
    sum[i] = v1[i] + v2[i];
for ( i=0; i < 4; i++ )
    printf("sum[%d] = %f\n", i, sum[i]);

```

4 Mehrdimensionale Felder

- neben eindimensionalen Felder kann man auch mehrdimensionale Felder vereinbaren
 - ◆ ihre Bedeutung in C ist gering
- Definition eines mehrdimensionalen Feldes

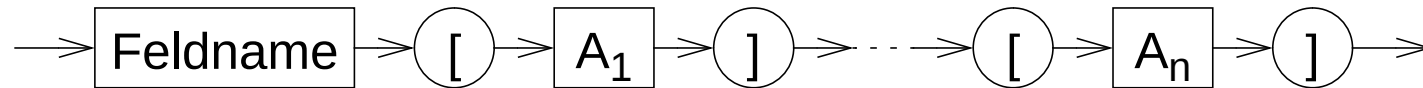


- Beispiel:

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

5 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:

feld[2][3] = 10;

6 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 nicht initialisiert!

F.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:

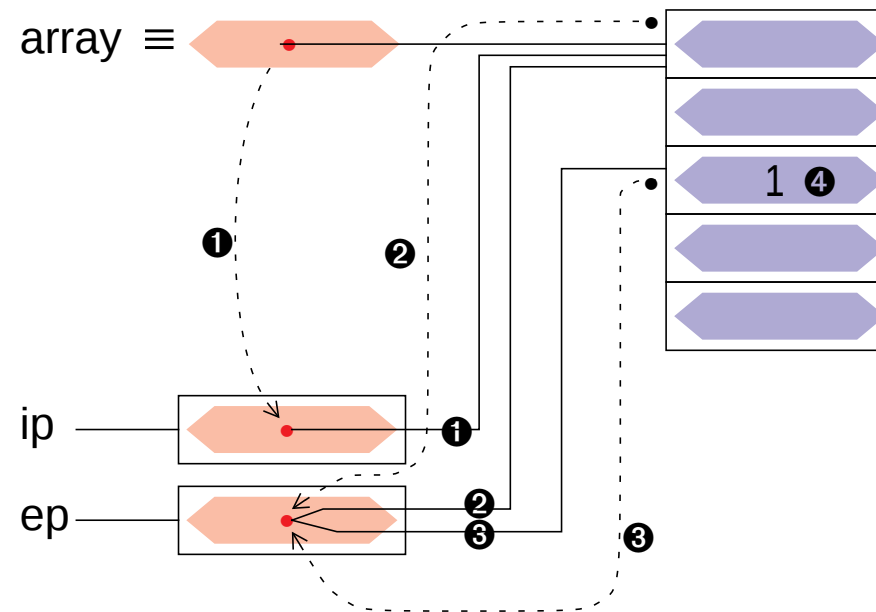
```
int array[5];

int *ip = array; ❶

int *ep;
ep = &array[0]; ❷

ep = &array[2]; ❸

*ep = 1; ❹
```



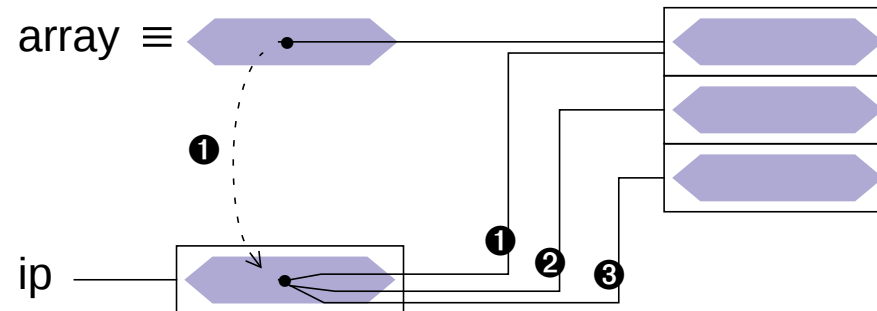
1 Arithmetik mit Adressen

- ++ -Operator: Inkrement = nächstes Objekt

```
int array[3];
int *ip = array; ❶
```

```
ip++; ❷
```

```
ip++; ❸
```



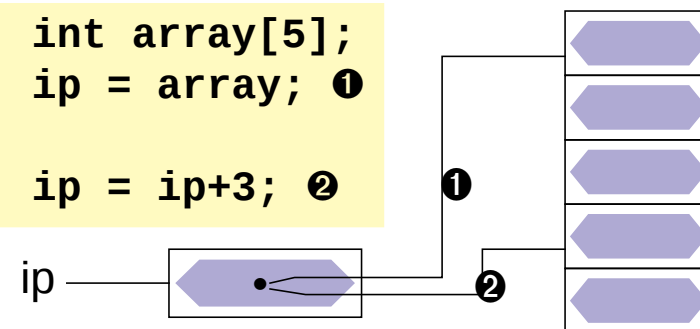
- -- -Operator: Dekrement = vorheriges Objekt

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

Dabei wird immer die Größe des
Objektyps berücksichtigt!

```
int array[5];
ip = array; ❶
```

```
ip = ip+3; ❷
```



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

2 Vorrangregeln bei Operatoren

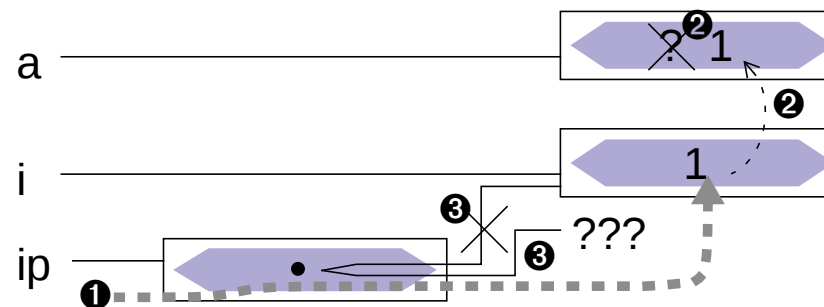
Operatorklasse	Operatoren	Assoziativität
primär	() Funktionsaufruf []	von links nach rechts
unär	! ~ ++ -- + - * &	von rechts nach links
multiplikativ	* / %	von links nach rechts
...		

3 Beispiele

```
int a, i, *ip;
i = 1;
ip = &i;
```

```
a = *ip++;
```

(1) $a = \underbrace{*ip}_{\text{②}} \underbrace{++}_{\text{①}};$
 (2) $a = \underbrace{*}_{\text{③}} \underbrace{ip}_{\text{②}} \underbrace{++}_{\text{③}};$

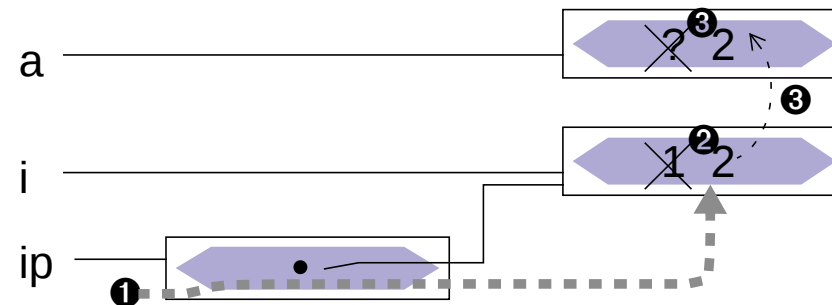


3 Beispiele (2)

```
int a, i, *ip;
i = 1;
ip = &i;
```

```
a = ++*ip;
```

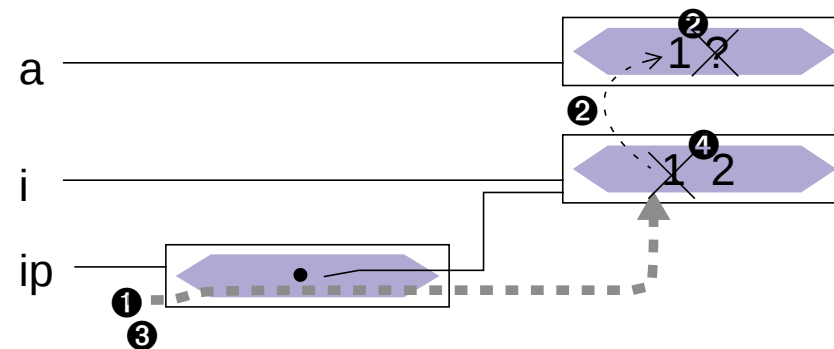
↳ (1) `a = ++*ip;`
 ↓
 ① *ip
 ② ++ → *ip
 ↓
 (2) `a = ++*ip;`
 ↓
 ③



```
int a, i, *ip;
i = 1;
ip = &i;
```

```
a = (*ip)++;
```

↳ (1) `a = (*ip)++;`
 ↑
 ②
 ↓
 (2) `a = (*ip)++;`
 ↓
 ③
 ↓
 ④ ++ → *ip



4 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;

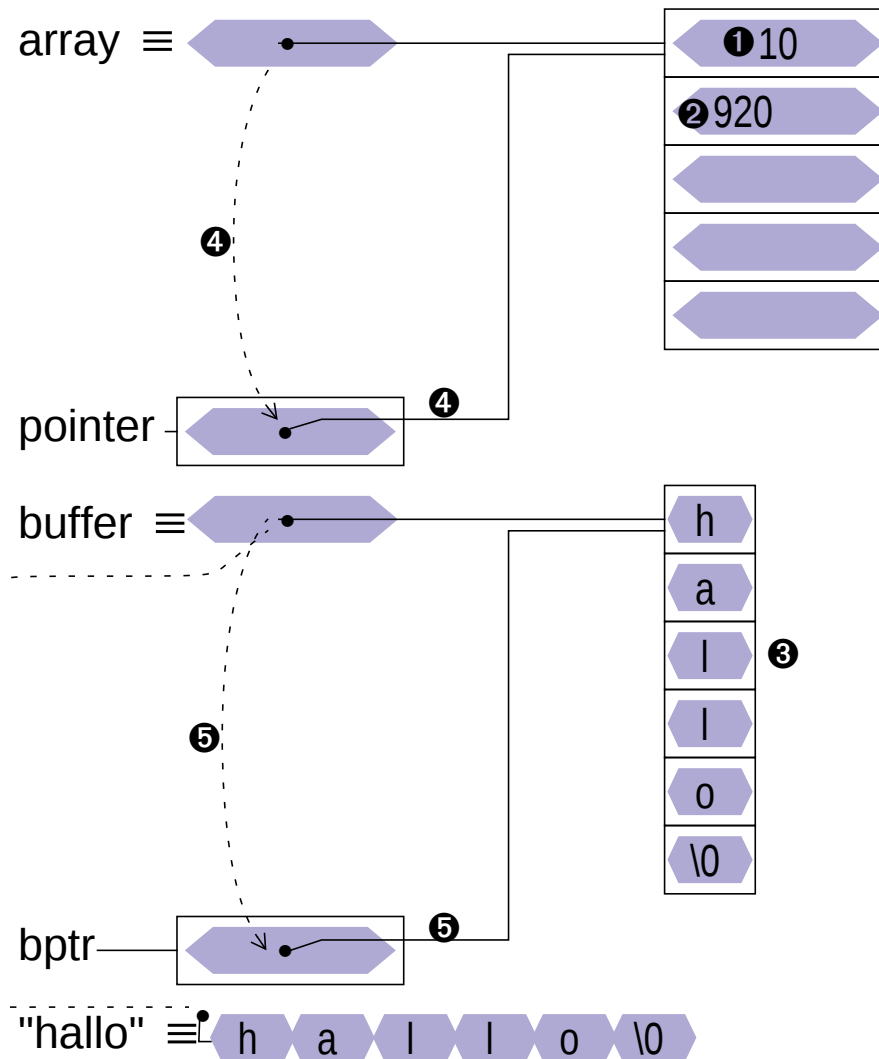
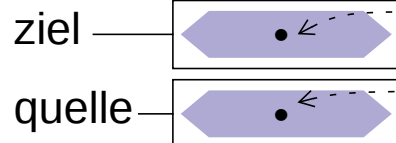
ep = &array[i]; *ep = 1;
ep = array+i; *ep = 1;
```

4 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



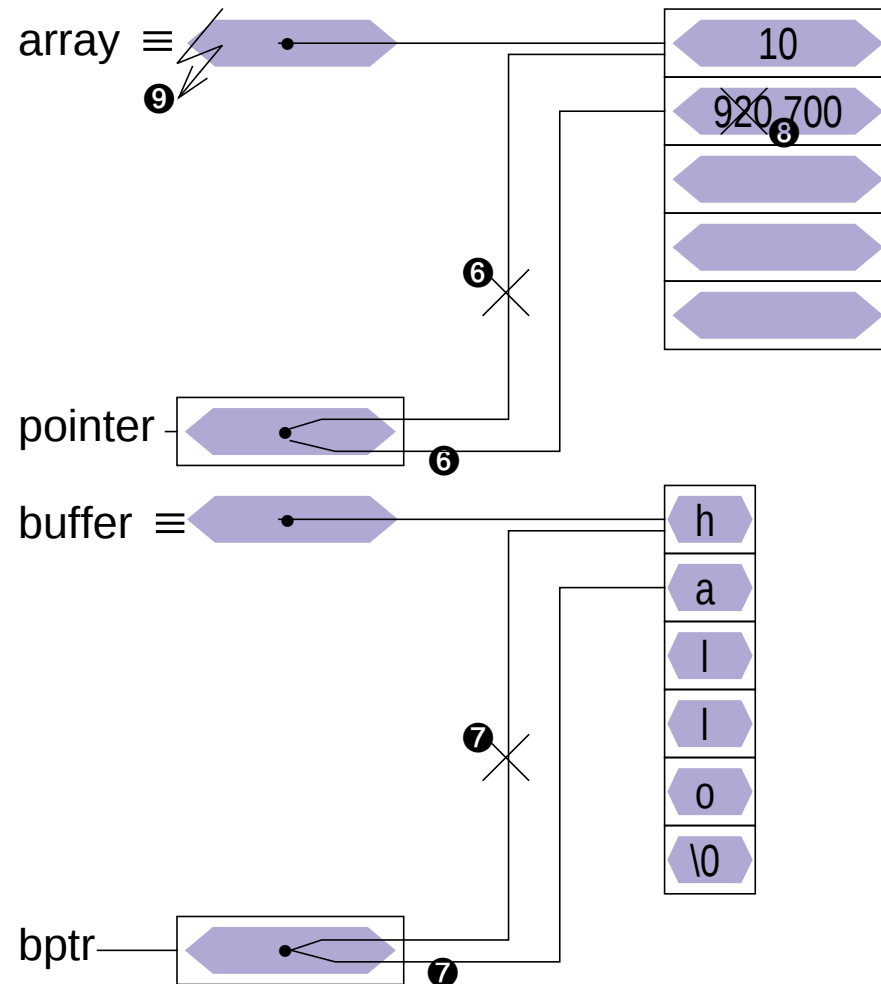
4 Zeigerarithmetik und Felder

```
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++; ⚡
```



5 Vergleichsoperatoren und Adressen

- Neben den arithmetischen Operatoren lassen sich auch die Vergleichsoperatoren auf Zeiger (allgemein: Adressen) anwenden:

<	kleiner
<=	kleiner gleich
>	größer
>=	größer gleich
==	gleich
!=	ungleich

F.4 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

F.4 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!!!

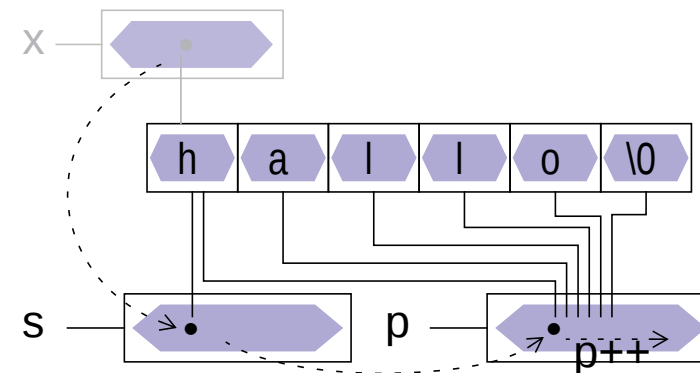
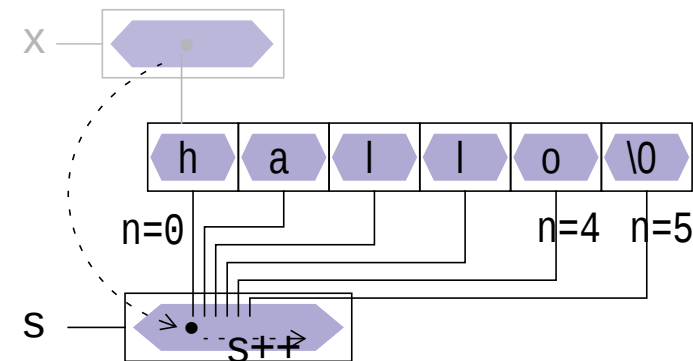

```
int f[] = {1, 2, 3}; /* initialisiertes Feld mit 3 Elementen */
int f1[];         /* ohne Initialisierung und Dimension nicht erlaubt! */
int *p;             /* Zeiger auf einen int */
```

F.5 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(char *s)
{
    int n;
    for (n=0; *s != '\0'; s++)
        n++;
    return(n);
}
```

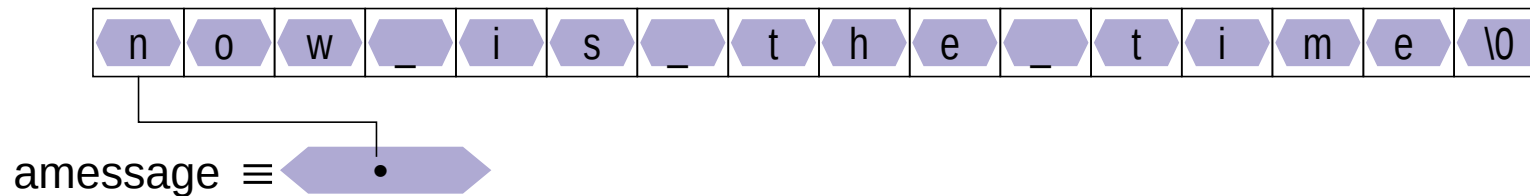
```
/* 2. Version */
int strlen(char *s)
{
    char *p = s;
    while (*p != '\0')
        p++;
    return(p-s);
}
```



F.5 ... 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";
```

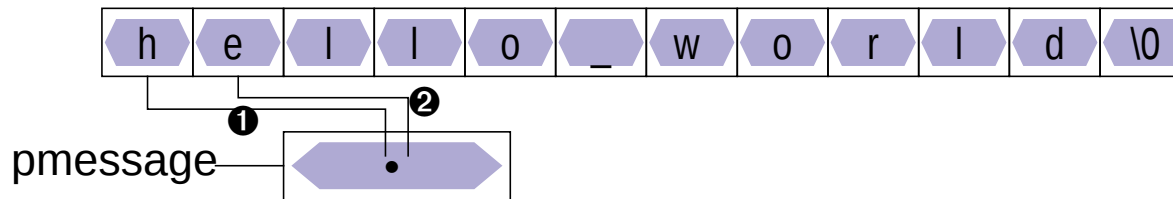


- es wird ein Speicherbereich für 16 Bytes reserviert und die Zeichen werden in diesen Speicherbereich hineinkopiert
- amessage ist ein konstanter Zeiger auf den Anfang des Speicherbereichs und kann nicht verändert werden
- der Inhalt des Speicherbereichs kann aber modifiziert werden
amessage[0] = 'h';

F.5 ... 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";
```



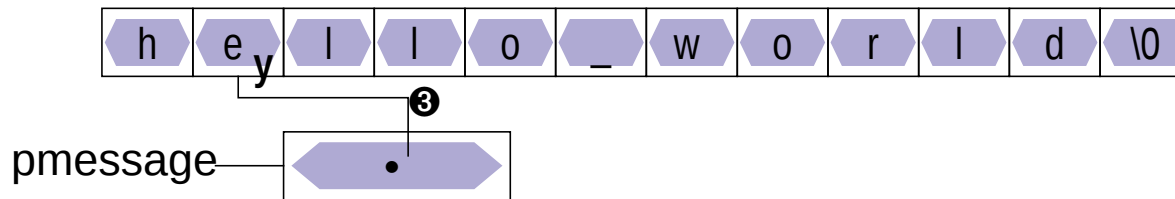
```
pmessage++; ②  
printf("%s", pmessage); /* gibt "ello world" aus */
```

- es wird ein Speicherbereich für einen Zeiger reserviert (z. B. 4 Byte) und der Compiler legt die Zeichenkette hello world an irgendeiner Adresse im Speicher des Programms ab
- pmessage ist ein variabler Zeiger, der mit dieser Adresse initialisiert wird, aber jederzeit verändert werden darf
pmessage++;

F.5 ... Zeiger, Felder und Zeichenketten (4)

- 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 = 'y'; ③ ⚡
```

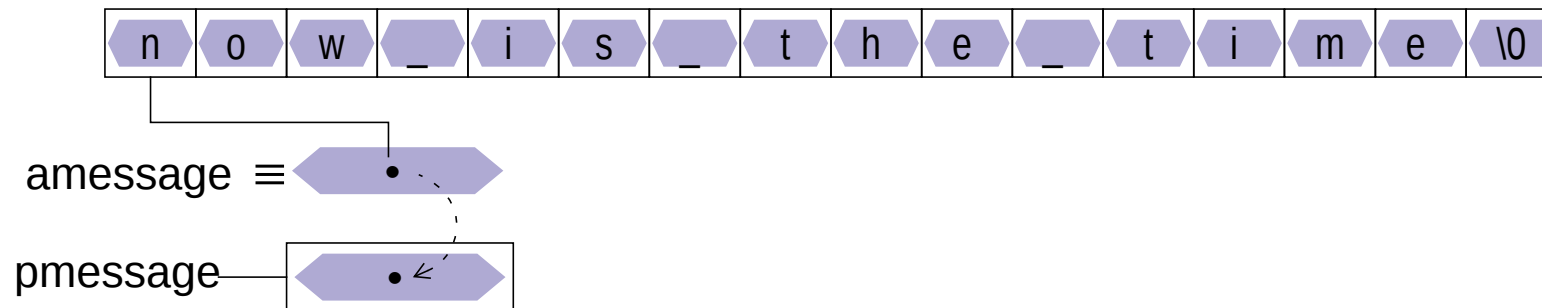
- der Speicherbereich von hello world darf aber nicht verändert werden
 - manche Compiler legen solche Zeichenketten in schreibgeschütztem Speicher an
 - ➔ Speicherschutzverletzung beim Zugriff
 - sonst funktioniert der Zugriff obwohl er nicht erlaubt ist
 - ➔ Programm funktioniert nur in manchen Umgebungen

F.5 ... Zeiger, Felder und Zeichenketten (5)

- 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

F.5 ... Zeiger, Felder und Zeichenketten (6)

■ Zeichenketten kopieren

```
/* 1. Version */
void strcpy(char s[], t[])
{
    int i=0;
    while ( (s[i] = t[i]) != '\0' )
        i++;
}

/* 2. Version */
void strcpy(char *s, *t)
{
    while ( (*s = *t) != '\0' )
        s++, t++;
}

/* 3. Version */
void strcpy(char *s, *t)
{
    while ( *s++ = *t++ )
        ;
}
```

F.6 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

F.6 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 der include-Datei `stdlib.h` definiert
#include <stdlib.h>

F.7 Explizite Typumwandlung — Cast-Operator

- C enthält Regeln für eine automatische Konvertierung unterschiedlicher Typen in einem Ausdruck (vgl. Abschnitt D.5.10)

Beispiel:

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

$d = (i * f);$

Diagramm: Ein Kreis umschließt den Ausdruck `i * f`. Ein Pfeil zeigt von `i` nach oben rechts zu `→float`. Ein Pfeil zeigt von `f` nach unten rechts zu `→double`.

- 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 *)

F.8 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;      /*  $\Rightarrow$  b = 2 oder b = 4 */  
b = sizeof(double) /*  $\Rightarrow$  b = 8 */
```

F.9 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

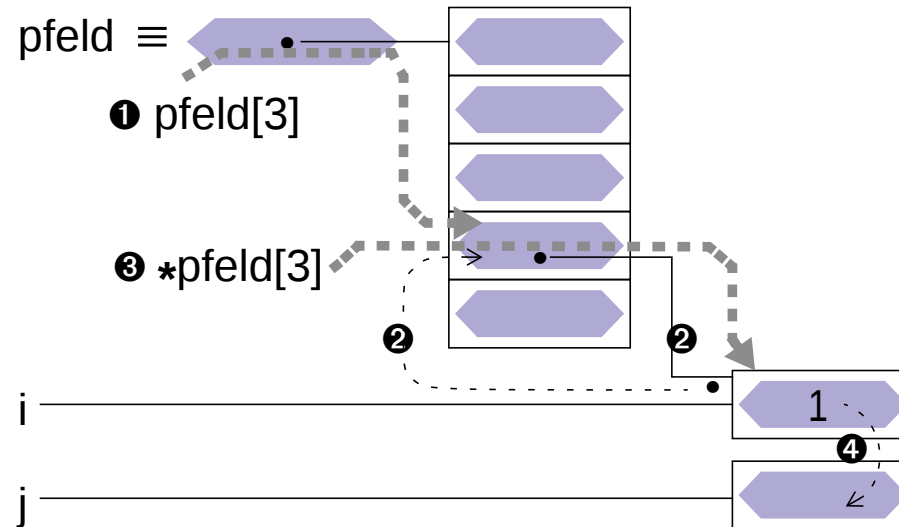
```
pfeld[3] = &i; ②
```

①

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

```
j = *pfeld[3]; ④
```

①
③
④

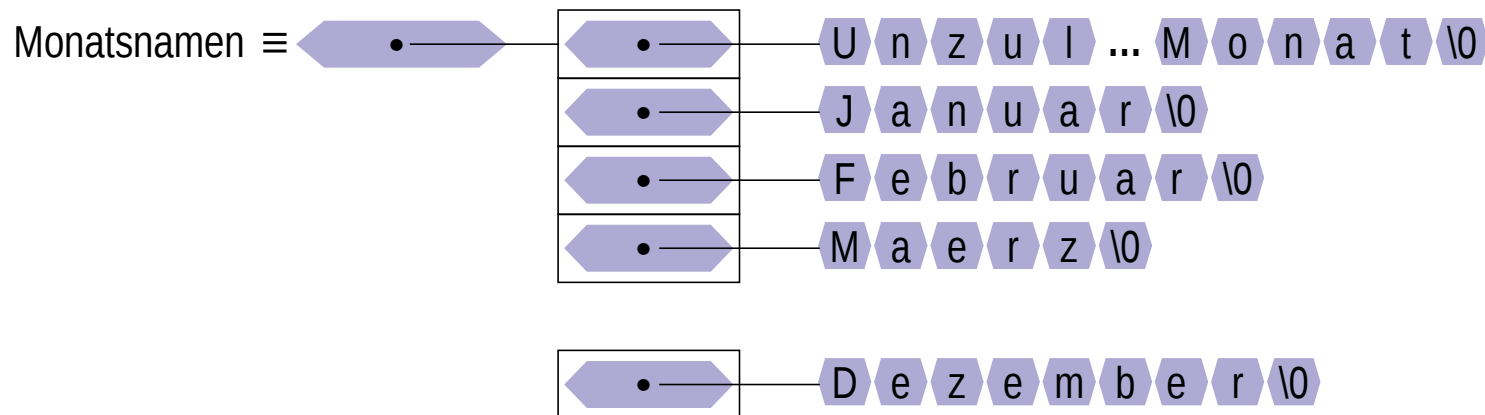


F.9 Felder von Zeigern (2)

- 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] );
}
```



F.10 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

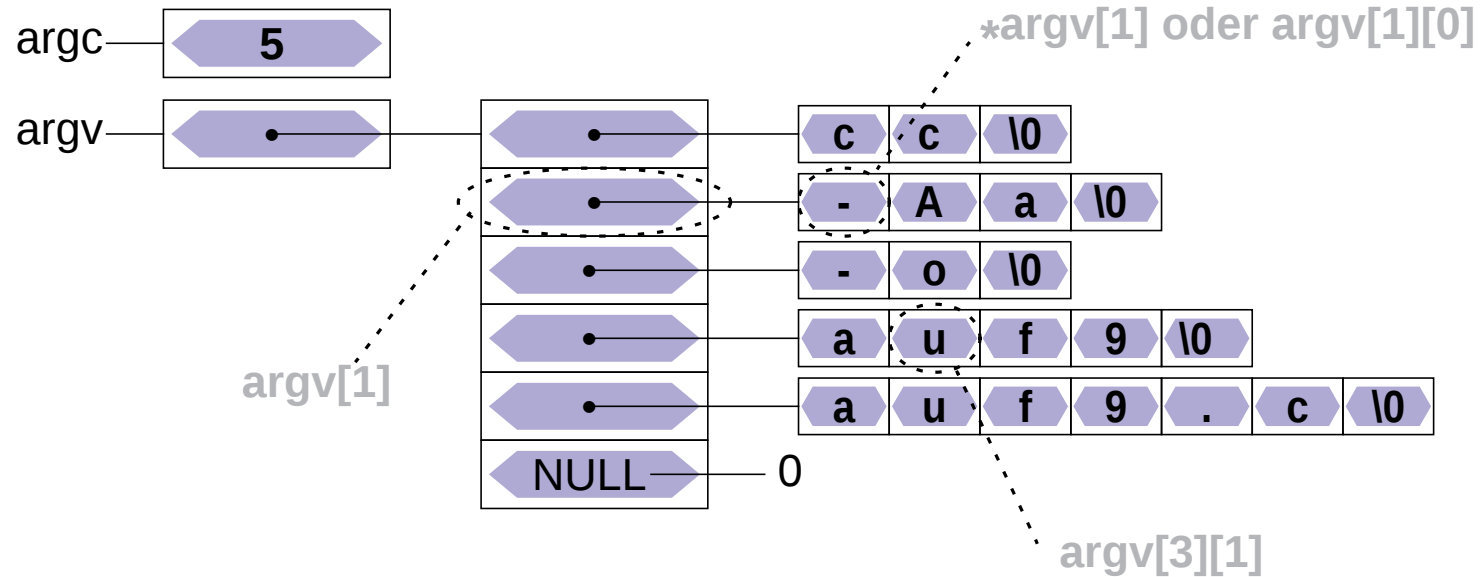
```
int  
main (int argc, char **argv)  
{  
    ...  
}
```

- 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]**)

1 Datenaufbau

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

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



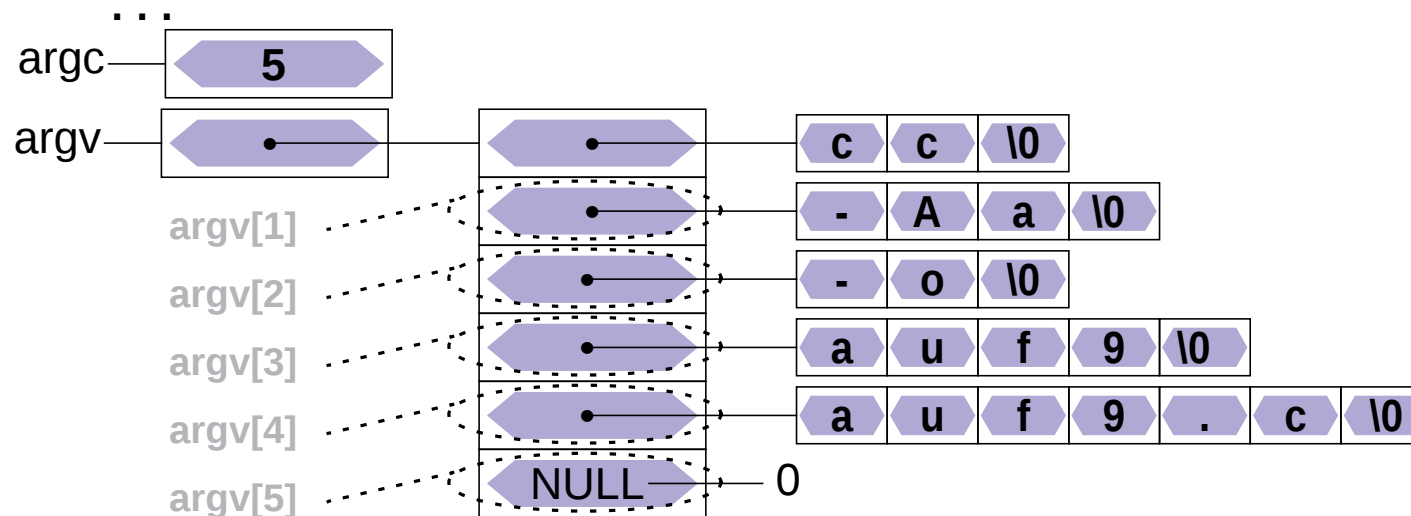
2 Zugriff

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



2 Zugriff

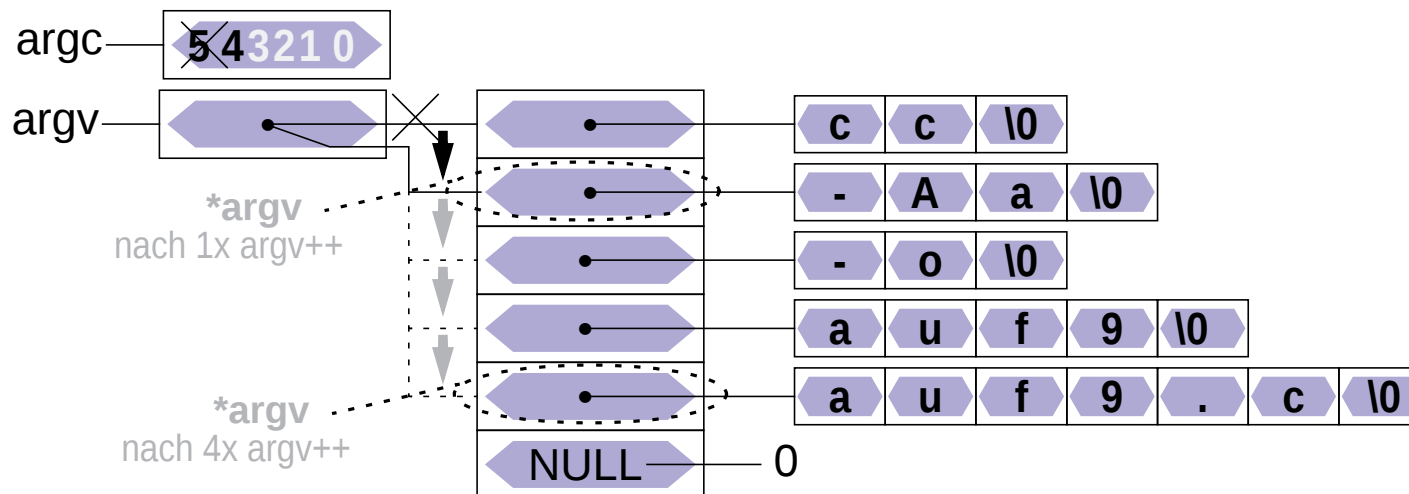
Beispiel: Ausgeben aller Argumente (2)

- 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

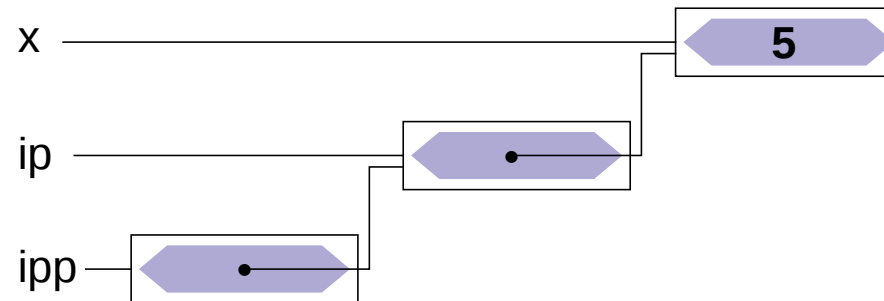
2. Version



F.11 Zeiger auf Zeiger

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

```
int x = 5;  
int *ip = &x;  
  
int **ipp = &ip;  
/* → **ipp = 5 */
```



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

F.12 Zeiger auf Funktionen

■ Datentyp: Zeiger auf Funktion

◆ Variablendef.: *<Rückgabetyt> (*<Variablenname>)(<Parameter>);*

```
int (*fptr)(int, char*);
```

```
int test1(int a, char *s) { printf("1: %d %s\n", a, s); }
```

```
int test2(int a, char *s) { printf("2: %d %s\n", a, s); }
```

```
fptr = test1;
```

```
fptr(42, "hallo");
```

```
fptr = test2;
```

```
fptr(42, "hallo");
```

F.13Strukturen

1 Motivation

- Felder fassen Daten eines einheitlichen Typs zusammen
 - ungeeignet für gemeinsame Handhabung von Daten unterschiedlichen Typs
- Beispiel: Daten eines Studenten

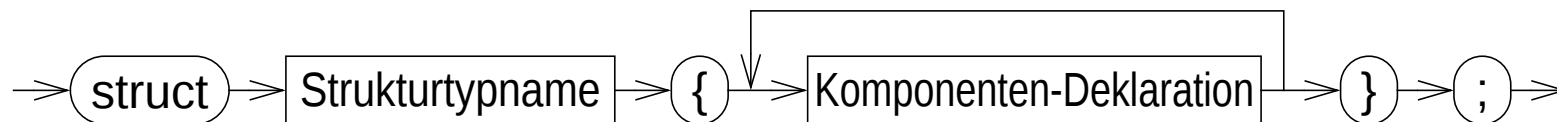
– Nachname	<code>char nachname[25];</code>
– Vorname	<code>char vorname[25];</code>
– Geburtsdatum	<code>char gebdatum[11];</code>
– Matrikelnummer	<code>int matrnr;</code>
– Übungsgruppennummer	<code>short gruppe;</code>
– Schein bestanden	<code>char best;</code>
- Möglichkeiten der Repräsentation in einem Programm
 - ◆ einzelne Variablen ➔ sehr umständlich, keine "Abstraktion"
 - ◆ Datenstrukturen

2 Deklaration eines Strukturtyps

- Durch eine Strukturtyp-Deklaration wird dem Compiler der Aufbau einer Datenstruktur unter einem Namen bekanntgemacht

➔ deklariert einen neuen Datentyp (wie **int** oder **float**)

- Syntax



- **Strukturtypname**

- ◆ beliebiger Bezeichner, kein Schlüsselwort
- ◆ kann in nachfolgenden Struktur-Definitionen verwendet werden

- **Komponenten-Deklaration**

- ◆ entspricht normaler Variablen-Definition, aber keine Initialisierung!
- ◆ in verschiedenen Strukturen dürfen die gleichen Komponentennamen verwendet werden (eigener Namensraum pro Strukturtyp)

2 Deklaration eines Strukturtyps (2)

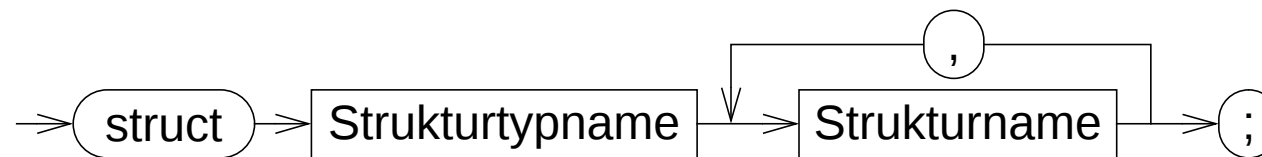
■ Beispiele

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

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

3 Definition einer Struktur

- Die Definition einer Struktur entspricht einer Variablen-Definition
 - ◆ Name der Struktur zusammen mit dem Datentyp bekanntmachen
 - ◆ Speicherplatz anlegen
- Eine Struktur ist eine Variable, die ihre Komponentenvariablen umfasst
- Syntax



- Beispiele

```
struct student stud1, stud2;  
struct komplex c1, c2, c3;
```

- Strukturdeklaration und -definition können auch in einem Schritt vorgenommen werden

4 Zugriff auf Strukturkomponenten

■ .-Operator

- **x.y** $\equiv$ Zugriff auf die Komponente **y** der Struktur **x**
- **x.y** verhält sich wie eine normale Variable vom Typ der Strukturkomponenten **y** der Struktur **x**

■ Beispiele

```
struct komplex c1, c2, c3;  
...  
c3.re = c1.re + c2.re;  
c3.im = c1.im + c2.im;  
  
struct student stud1;  
...  
if (stud1.matrnr < 1500000) {  
    stud1.best = 'y';  
}
```

5 Initialisieren von Strukturen

- Strukturen können — wie Variablen und Felder — bei der Definition initialisiert werden
- Beispiele

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

!!! Vorsicht

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

bei der Initialisierung jedoch nur durch die Position

→ potentielle Fehlerquelle bei Änderungen der Strukturtyp-Deklaration

6 Strukturen als Funktionsparameter

- Strukturen können wie normale Variablen an Funktionen übergeben werden
 - ◆ Übergabesemantik: **call by value**
 - Funktion erhält eine Kopie der Struktur
 - auch wenn die Struktur ein Feld enthält, wird dieses komplett kopiert!
 - !!! Unterschied zur direkten Übergabe eines Feldes
- Strukturen können auch Ergebnis einer Funktion sein
 - Möglichkeit mehrere Werte im Rückgabeparameter zu transportieren
- Beispiel

```
struct komplex komp_add(struct komplex x, struct komplex y) {  
    struct komplex ergebnis;  
    ergebnis.re = x.re + y.re;  
    ergebnis.im = x.im + y.im;  
    return(ergebnis);  
}
```

7 Zeiger auf Strukturen

■ Konzept analog zu "Zeiger auf Variablen"

- Adresse einer Struktur mit &-Operator zu bestimmen
- Name eines Feldes von Strukturen = Zeiger auf erste Struktur im Feld
- Zeigerarithmetik berücksichtigt Strukturgröße

■ Beispiele

```
struct student stud1;  
struct student gruppe8[35];  
struct student *pstud;  
pstud = &stud1;           /* ⇒ pstud → stud1 */  
pstud = gruppe8;          /* ⇒ pstud → gruppe8[0] */  
pstud++;                  /* ⇒ pstud → gruppe8[1] */  
pstud += 12;              /* ⇒ pstud → gruppe8[13] */
```

■ Besondere Bedeutung zum Aufbau

↳ rekursiver Strukturen

7 Zeiger auf Strukturen (2)

- Zugriff auf Strukturkomponenten über einen Zeiger

- Bekannte Vorgehensweise

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



```
(*pstud).best = 'n';
```

unleserlich!

- Syntaktische Verschönerung



->-Operator

```
pstud->best = 'n';
```

8 Felder von Strukturen

- Von Strukturen können — wie von normale Datentypen — Felder gebildet werden
- Beispiel

```
struct student gruppe8[35];
int i;
for (i=0; i<35; i++) {
    printf("Nachname %d. Stud.: ", i);
    scanf("%s", gruppe8[i].nachname);
    ...
    gruppe8[i].gruppe = 8;

    if (gruppe8[i].matrnr < 1500000) {
        gruppe8[i].best = 'y';
    } else {
        gruppe8[i].best = 'n';
    }
}
```

9 Strukturen in Strukturen

- Die Komponenten einer Struktur können wieder Strukturen sein
- Beispiel

```
struct name {
    char nachname[25];
    char vorname[25];
};

struct student {
    struct name name;
    char gebdatum[11];
    int matrnr;
    short gruppe;
    char best;
}

struct student stud1;
strcpy(stud1.name.nachname, "Meier");
if (stud1.name.nachname[0] == 'M') {
    ...
}
```

```
struct prof {
    struct name pname;
    char gebdatum[11];
    int lehrstuhlnr;
}
```

10 Rekursive Strukturen

- Strukturen in Strukturen sind erlaubt — aber
 - ◆ die Größe einer Struktur muss vom Compiler ausgerechnet werden können
 - Problem: eine Struktur enthält sich selbst

```
struct liste {  
    struct student stud;  
    struct liste rest;  
};
```

falsch!

- ◆ die Größe eines Zeigers ist bekannt (meist 4 Byte)
 - eine Struktur kann einen Zeiger auf eine gleichartige Struktur enthalten

```
struct liste {  
    struct student stud;  
    struct liste *rest;  
};
```

➔ Programmieren rekursiver Datenstrukturen

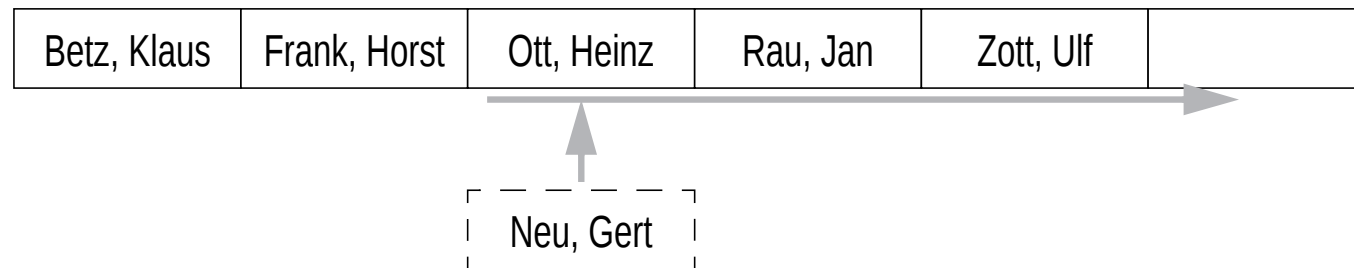
10 Rekursive Strukturen (2)

■ Problem:

- ◆ es sollen beliebig viele Studentendaten eingelesen werden und in sortierter Form im Programm verwaltet werden

Lösung 1: Feld

- wie groß machen? — und was, wenn es nicht reicht?
- Einsortieren =
richtige Position suchen + Rest nach oben verschieben + eintragen



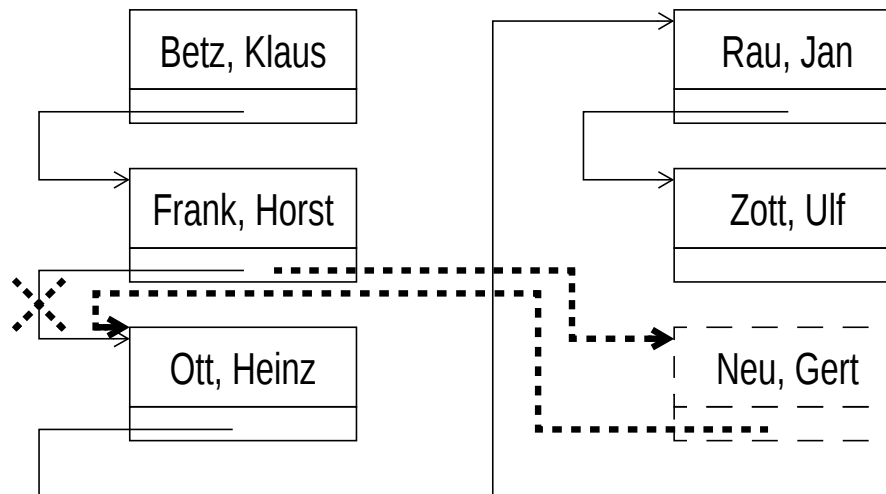
10 Rekursive Strukturen (3)

■ Problem:

- ◆ es sollen beliebig viele Studentendaten eingelesen werden und in sortierter Form im Programm verwaltet werden

Lösung 2: verkettete Liste von dynamisch angeforderten Strukturen

- Speicher für jeden Eintrag mit malloc() anfordern
- Einsortieren =
richtige Position suchen + zwei Zeiger setzen



10 Rekursive Strukturen (4)

■ Realisierung von Lösung 2 (Skizze):

```

struct eintrag {
    struct student stud;
    struct eintrag *naechster;
};
struct eintrag leer = { {"", ""}, NULL};          /* Leeres Listenelement */
struct eintrag *stud_liste;                       /* Zeiger auf Listen-Anfang */
struct eintrag *akt_eintrag;                      /* aktuell bearbeiteter Eintrag */
struct eintrag *einfuege_pos;                    /* Einfuegeposition */

int student_lesen(struct student *);
struct eintrag *suche_pos(struct eintrag *liste, struct eintrag *element);
/* erstes Listen-Element anfordern */
akt_eintrag = (struct eintrag *)malloc(sizeof (struct eintrag));
stud_liste = &leer;                             /* Listenanfang auf leeres Element setzen (vermeidet später
                                                    /* eine Sonderbehandlung für Listenanfang) */

while (student_lesen(&akt_eintrag->stud) != EOF ) {
    /* Eintrag, hinter dem einzufügen ist suchen */
    einfuege_pos = suche_pos(stud_liste, akt_eintrag);
    /* akt_eintrag einfügen */
    akt_eintrag->naechster = einfuege_pos->naechster;
    einfuege_pos->naechster = akt_eintrag;
    /* nächstes Listen-Element anfordern */
    akt_eintrag = (struct eintrag *)malloc(sizeof (struct eintrag));
}

```

F.14 Verbundstrukturen — Unions

- In einer Struktur liegen die einzelnen Komponenten hintereinander, in einem Verbund liegen sie übereinander
 - die gleichen Speicherzellen können unterschiedlich angesprochen werden
 - Beispiel: ein int-Wert (4 Byte) und die einzelnen Bytes des int-Werts

```
union intbytes {  
    int ival;   
    char bvalue[4];  
} u1;  
...  
u1.ival = 259000;  
printf("Wert=%d, Byte0=%d, Byte1=%d, Byte2=%d, Byte3=%d\n",  
    u1.ival, u1.bvalue[0], u1.bvalue[1], u1.bvalue[2], u1.bvalue[3]);
```

- Einsatz nur in sehr speziellen Fällen sinnvoll
konkretes Wissen über die Speicherorganisation unbedingt erforderlich!

F.15 Bitfelder

- Bitfelder sind Strukturkomponenten bei denen die Zahl der für die Speicherung verwendeten Bits festgelegt werden kann.
- Anwendung z. B. um auf einzelnen Bits eines Registers zuzugreifen

```
struct cregister {  
    unsigned int protection : 1;  
    unsigned int interrupt_mask : 3;  
    unsigned int enable_read : 1;  
    unsigned int enable_write : 1;  
    unsigned int pedding : 2;  
    unsigned int address : 8;  
};
```

F.15 Bitfelder (2)

- Struktur mit Bitfeld kann ihrerseits Teil einer Union sein
 - Zugriff auf Register als Ganzes und auf die einzelnen Bits

```
union cregister {  
    unsigned short all;  
    struct bits {  
        unsigned int protection : 1;  
        unsigned int interrupt_mask : 3;  
        unsigned int enable_read : 1;  
        unsigned int enable_write : 1;  
        unsigned int pedding : 2;  
        unsigned int address : 8;  
    };  
};
```

- Adresse und Aufbau eines Registers steht üblicherweise in der Hardwarebeschreibung.

F.15 Bitfelder (3)

■ Beispiel:

- Adresse auf Register anlegen,
Registerinhalt sichern
Bits verändern
Registerinhalt wieder herstellen

```
union cregister *creg;  
unsigned short oldvale;  
creg = (union cregister *)0x2400; /* Addr. aus Manual */  
oldvalue = creg->all;           /* Wert sichern */  
creg->bits.protection = 0;  
creg->bits.enable_read = 1;  
creg->bits.address = 0x40;  
...  
creg->all = oldvalue;           /* Wert restaurieren */
```

F.16 Typedef

- Typedef erlaubt die Definition neuer Typen
 - neuer Typ kann danach wie die Standardtypen (int, char, ...) genutzt werden

- Beispiele:

```
typedef int Laenge;  
Laenge l = 5;  
Laenge *p1; Laenge f1[20];
```

```
typedef struct student Student;  
Student s; Student *ps1;  
s.matrnr = 1234567; ps1 = &s; ps1->best = 'n';
```

```
typedef struct student *Studptr;  
Studptr ps2;  
ps2 = &s; ps2->best = 'n';
```

F.17 Ergänzungen zum Präprozessor

1 Parametrisierte Makros

- Präprozessor-Makros können mit Parametern versehen werden, die in den Ersatztext eingebaut werden

- entspricht "optisch" einem Funktionsaufruf
- Unterschied: Makro wird zur Übersetzungszeit expandiert, Funktionsaufruf erfolgt zur Laufzeit
- parametrisierte Makros sind damit ähnlich zu inline-Funktionen (wie z. B. in C++)
- ein Makro wird durch die **#define**-Anweisung definiert

- Syntax:

```
#define Makroname(Par1, Par2, ...) Ersatztext
```

- Vorkommen von *Par1*, *Par2*, etc. im Ersatztext werden entsprechend eingesetzt

F.17 Ergänzungen zum Präprozessor (2)

■ Beispiele:

```
#define max(a, b) ((a) < (b) ? (b) : (a))
...
x = max (n+m, y+z);
```

```
#define numeric(c) ((c) >= '0' && (c) <= '9')
...
c = getchar();
if numeric(c) { ...
```

- im Ersatztext Parameter unbedingt in () klammern
(sonst evtl. Probleme mit Operator-Vorrang im expandierten Text!)
- einige Sonderregeln im Umgang mit Zeichenketten
(Kernighan und Ritchie schreiben dazu:
*The details ... are described more precisely in the ANSI standard ...
Some of the new rules ... are bizarre.)*

F.17 Ergänzungen zum Präprozessor (3)

2 Deaktivieren von Makros

- Makros wirken ab der Zeile in der sie definiert wurden bis zum Ende der Datei - können aber auch wieder deaktiviert werden
- Syntax:

```
#undef Makroname
```

- ab dieser Zeile unterbleibt Expansion des Makros