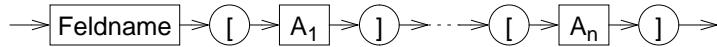


6 Zugriffe auf Feldelemente bei mehrdim. Feldern

C-12 Felder

- 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;
```

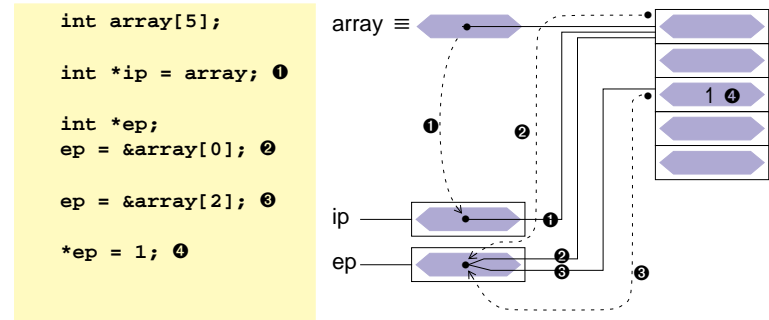
- ist äquivalent zu:

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

C-13 Zeiger und Felder

C-13 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:



7 Initialisierung eines mehrdimensionalen Feldes

C-12 Felder

- 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:

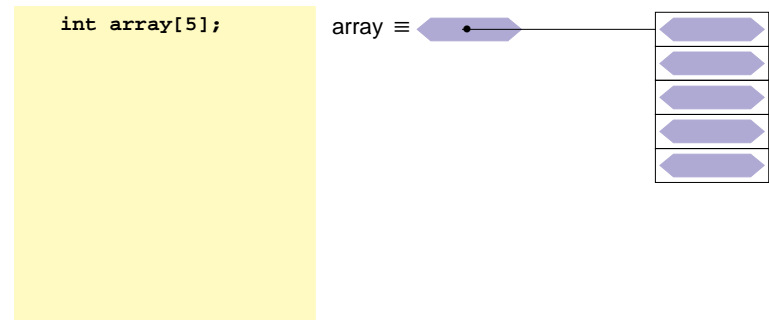
```
int feld[3][4] = {
    { 1, 3, 5, 7 }, /* feld[0][0-3] */
    { 2, 4, 6   } /* feld[1][0-2] */
};
```

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

C-13 Zeiger und Felder

C-13 Zeiger und Felder

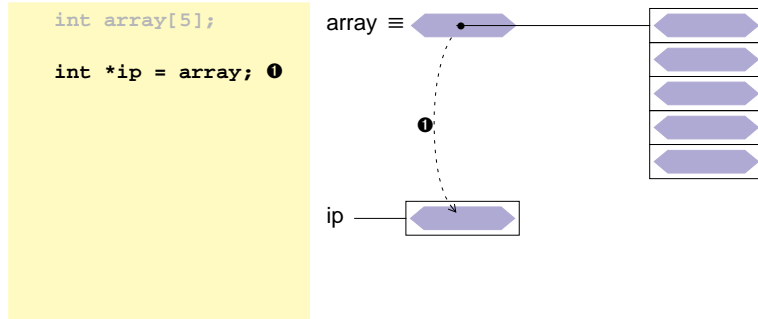
- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:



C-13 Zeiger und Felder

C-13 Zeiger und Felder

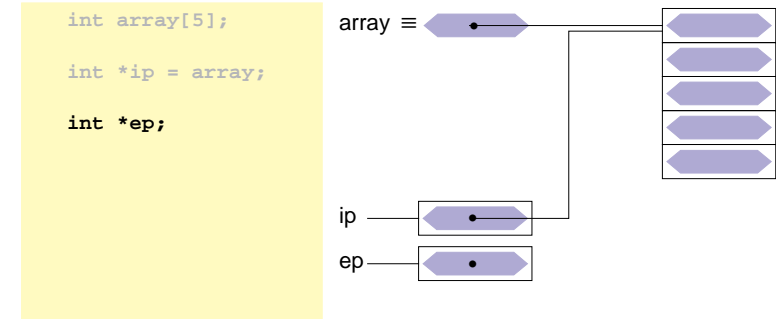
- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:



C-13 Zeiger und Felder

C-13 Zeiger und Felder

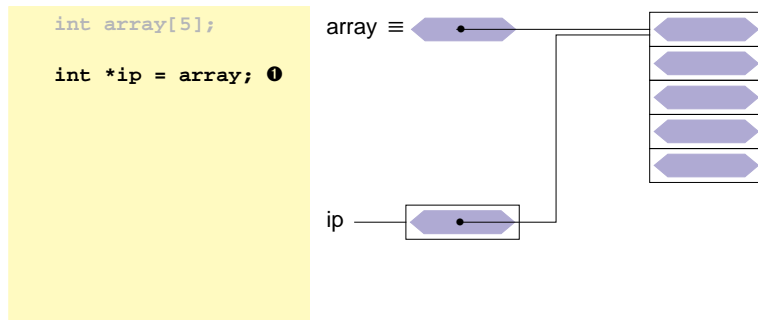
- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:



C-13 Zeiger und Felder

C-13 Zeiger und Felder

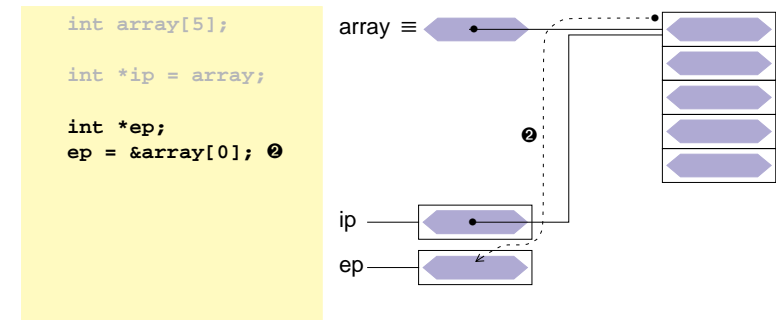
- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:



C-13 Zeiger und Felder

C-13 Zeiger und Felder

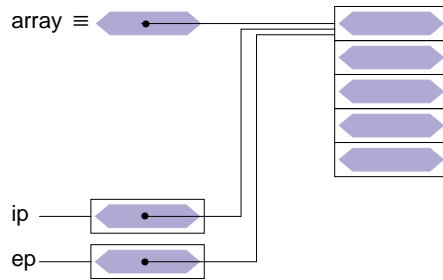
- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:



C-13 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:

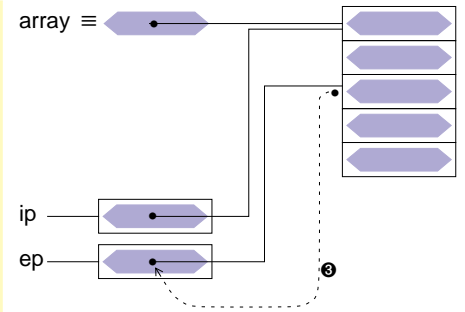
```
int array[5];  
  
int *ip = array;  
  
int *ep;  
ep = &array[0]; ②
```



C-13 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:

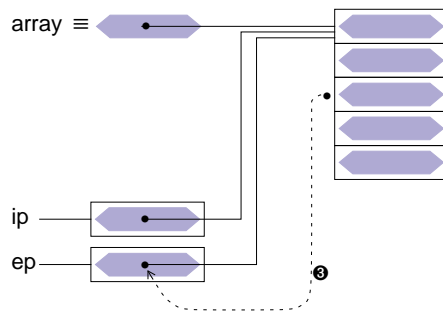
```
int array[5];  
  
int *ip = array;  
  
int *ep;  
ep = &array[2]; ③
```



C-13 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:

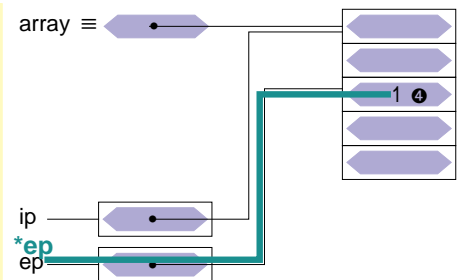
```
int array[5];  
  
int *ip = array;  
  
int *ep;  
ep = &array[0];  
  
*ep = &array[2]; ④
```



C-13 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 = 1; ④
```



- 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:

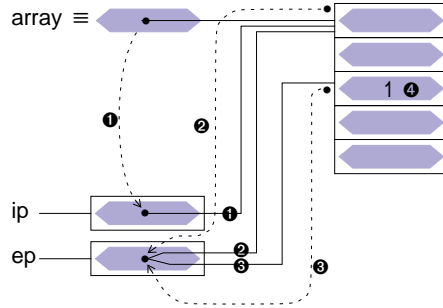
```
int array[5];

int *ip = array; ①

int *ep;
ep = &array[0]; ②

ep = &array[2]; ③

*ep = 1; ④
```



- 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;

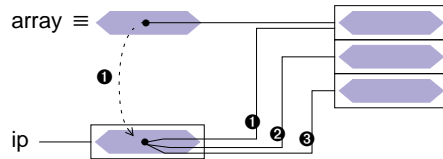
ep = &array[i]; *ep = 1;
ep = array+i; *ep = 1;
```

1 Arithmetik mit Adressen

■ ++ -Operator: Inkrement = nächstes Objekt

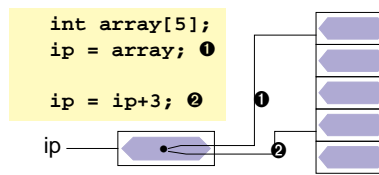
```
int array[3];
int *ip = array; ①

ip++; ②
ip++; ③
```



■ -- -Operator: Dekrement = vorheriges Objekt

■ +, -
Addition und Subtraktion von Zeigern und ganzzahligen Werten.
Dabei wird immer die Größe des Objekttyps berücksichtigt!

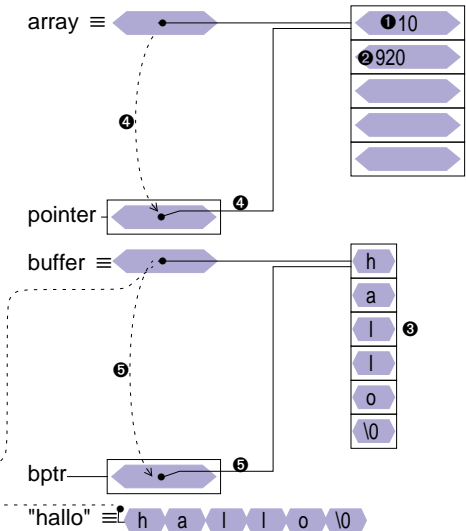


!!! **Achtung:** Assoziativität der Operatoren beachten !!

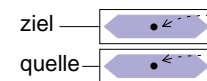
2 Zeigerarithmetik und Felder

```
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



2 Zeigerarithmetik und Felder

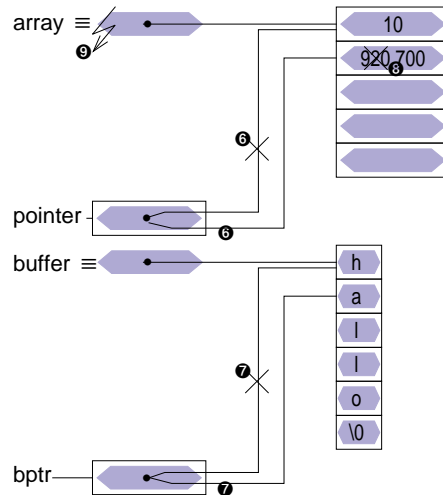
C-13 Zeiger und Felder

```
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++;
```



C-14 Eindimensionale Felder als Funktionsparameter (2)

C-14 Eindimensionale Felder als Funktionsparameter

- 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!

C-14 Eindimensionale Felder als Funktionsparameter

C-14 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

C-14 Eindimensionale Felder als Funktionsparameter (3)

C-14 Eindimensionale Felder als Funktionsparameter

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

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

C-14 Eindimensionale Felder als Funktionsparameter (4)

C-14 Eindimensionale Felder als Funktionsparameter

■ 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" */
```

Softwaresysteme I

© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2005

C-Einf.fm 2005-04-14 08.55

C.89

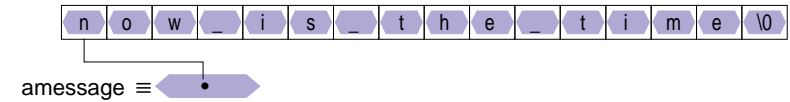
Reproduktion jeder Art oder Verwendung dieser Unterlagen, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

C-15 ... Zeiger, Felder und Zeichenketten (2)

C-15 Zeiger, Felder und Zeichenketten

- 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";
```



Sos I

Softwaresysteme I

© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2005

C-Einf.fm 2005-04-14 08.55

C.89

Reproduktion jeder Art oder Verwendung dieser Unterlagen, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

Sos I

Softwaresysteme I

© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2005

C-Einf.fm 2005-04-14 08.55

C.91

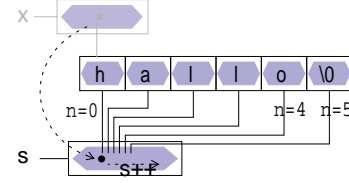
Reproduktion jeder Art oder Verwendung dieser Unterlagen, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

C-15 Zeiger, Felder und Zeichenketten

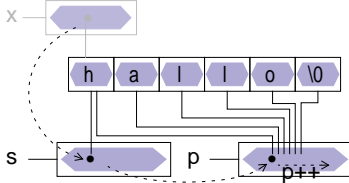
C-15 Zeiger, Felder und Zeichenketten

- 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);
}
```



Sos I

Softwaresysteme I

© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2005

C-Einf.fm 2005-04-14 08.55

C.90

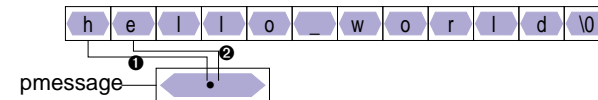
Reproduktion jeder Art oder Verwendung dieser Unterlagen, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

C-15 ... Zeiger, Felder und Zeichenketten (3)

C-15 Zeiger, Felder und Zeichenketten

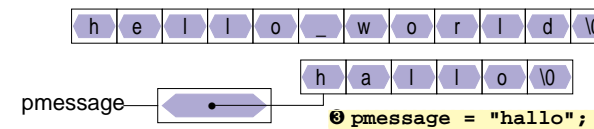
- wird eine Zeichenkette zur Initialisierung eines **char**-Zeigers verwendet, ist der Zeiger eine Variable, die mit der Anfangsadresse der Zeichenkette initialisiert wird

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



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

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



Sos I

Softwaresysteme I

© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2005

C-Einf.fm 2005-04-14 08.55

C.92

Reproduktion jeder Art oder Verwendung dieser Unterlagen, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

Sos I

Softwaresysteme I

© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2005

C-Einf.fm 2005-04-14 08.55

C.92

Reproduktion jeder Art oder Verwendung dieser Unterlagen, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

C-15 ... 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

SoSe

C-15 ... Zeiger, Felder und Zeichenketten (6)

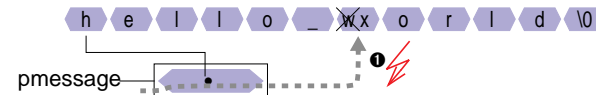
- in ANSI-C können Zeichenketten in nicht-modifizierbaren Speicherbereichen angelegt werden (je nach Compiler)

- 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!
```

SoSe

C-15 ... 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++ )  
        ;  
}
```

SoSe

C-16 Felder von Zeigern

- Auch von Zeigern können Felder gebildet werden

- Deklaration

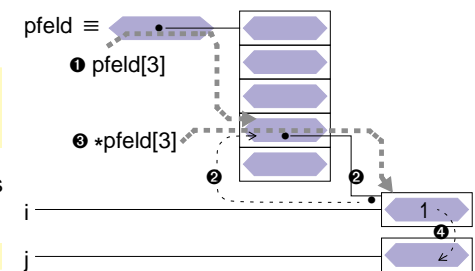
```
int *pfeld[5];  
int i = 1;  
int j;
```

- Zugriffe auf einen Zeiger des Feldes

```
pfeld[3] = &i;
```

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

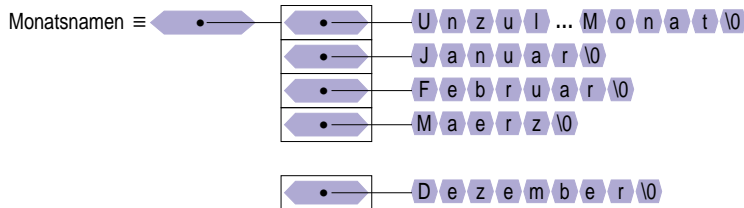
```
j = *pfeld[3];
```



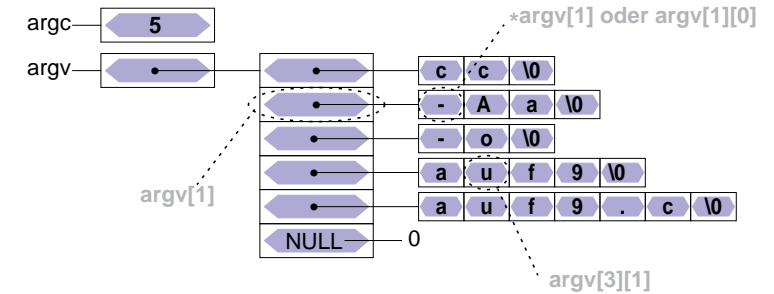
SoSe

■ 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] );
}
```



```
Kommando: cc -Aa -o auf9 auf9.c
Datei cc.c:
...
main(int argc, char *argv[]) {
...
}
```



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

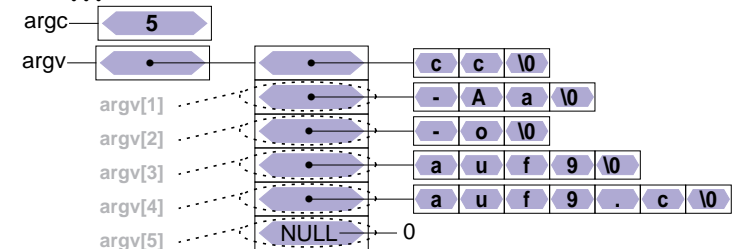
```
int main (int argc, char *argv[])
{
    ...
}
oder
int main (int argc, char **argv)
{
    ...
}
```

- 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]**)

- 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

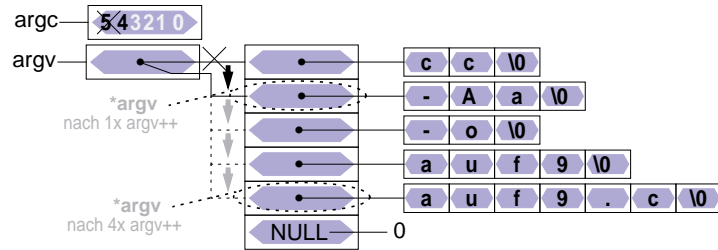


2 Zugriff Beispiel: Ausgeben aller Argumente (2)

- 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' );
    }
    ...
}
```

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



SOS I

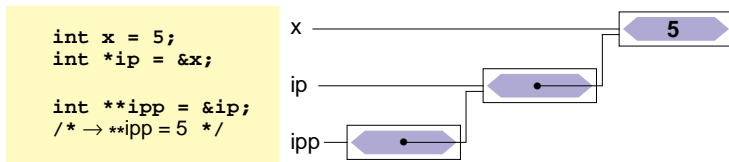
C-19 Strukturen

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

SOS I

C.18 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)

SOS I

1 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

SOS I

2 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);
}
```

4 Zeiger auf Felder von Strukturen

- Ergebnis der Addition/Subtraktion abhängig von Zeigertyp!
- Beispiel

```
struct student gruppe8[35];
struct student *gp1, *gp2;

gp1 = gruppe8; /* gp1 zeigt auf erstes Element des Arrays */
printf("Nachname des ersten Studenten: %s", gp1->nachname);

gp2 = gp1 + 1; /* gp2 zeigt auf zweite Element des Arrays */
printf("Nachname des zweiten Studenten: %s", gp2->nachname);

printf("Byte-Differenz: %d", (char*)gp2 - (char*)gp1);
```

3 Felder von Strukturen

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

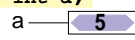
```
struct student gruppe8[35];
int i;
for (i=0; i<35; i++) {
    printf("Nachname %d. Stud.: ", i);
    scanf("%s", gruppe8[i].nachname);
    ...
    gruppe8[i].gruppe = 8;

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

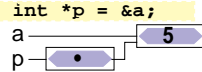
5 Zusammenfassung

- Variable

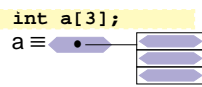

```
int a;
```


- Zeiger

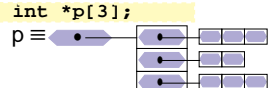

```
int *p = &a;
```


- Feld



```
int a[3];
```


- Feld von Zeigern



```
int *p[3];
```


- Struktur

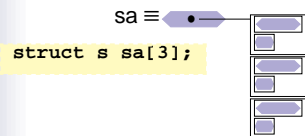

```
struct s{int a; char c;};
struct s s1 = {2, 'a'};
```


- Zeiger auf Struktur


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


- Feld von Strukturen


```
struct s sa[3];
```



■ Datentyp: Zeiger auf Funktion

◆ Variablendef.: `<Rückgabotyp> (*<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");
```

■ Jedes C-Programm erhält beim Start automatisch 3 E-/A-Kanäle:

◆ `stdin` Standardeingabe

- normalerweise mit der Tastatur verbunden
- Dateiende (EOF) wird durch Eingabe von CTRL-D am Zeilenanfang signalisiert
- bei Programmaufruf in der Shell auf Datei umlenkbar
`prog <eingabedatei`
 (bei Erreichen des Dateiendes wird EOF signalisiert)

◆ `stdout` Standardausgabe

- normalerweise mit dem Bildschirm (bzw. dem Fenster, in dem das Programm gestartet wurde) verbunden
- bei Programmaufruf in der Shell auf Datei umlenkbar
`prog >ausgabedatei`

◆ `stderr` Ausgabekanal für Fehlermeldungen

- normalerweise ebenfalls mit Bildschirm verbunden

■ E-/A-Funktionalität nicht Teil der Programmiersprache

■ Realisierung durch "normale" Funktionen

- Bestandteil der Standard-Funktionsbibliothek
- einfache Programmierschnittstelle
- effizient
- portabel
- betriebssystemnah

■ Funktionsumfang

- Öffnen/Schließen von Dateien
- Lesen/Schreiben von Zeichen, Zeilen oder beliebigen Datenblöcken
- Formatierte Ein-/Ausgabe

■ Pipes

- ◆ die Standardausgabe eines Programms kann mit der Standardeingabe eines anderen Programms verbunden werden
- Aufruf
`prog1 | prog2`

! Die Umlenkung von Standard-E/A-Kanäle ist für die aufgerufenen Programme völlig unsichtbar

■ automatische Pufferung

- ◆ Eingabe von der Tastatur wird normalerweise vom Betriebssystem zeilenweise zwischengespeichert und erst bei einem NEWLINE-Zeichen (`'\n'`) an das Programm übergeben!

2 Öffnen und Schließen von Dateien

- Neben den Standard-E/A-Kanälen kann ein Programm selbst weitere E/A-Kanäle öffnen

➤ Zugriff auf Dateien

- Öffnen eines E/A-Kanals

➤ Funktion `fopen`:

```
#include <stdio.h>
FILE *fopen(char *name, char *mode);
```

name Pfadname der zu öffnenden Datei

mode Art, wie die Datei geöffnet werden soll

- "r" zum Lesen
- "w" zum Schreiben
- "a" append: Öffnen zum Schreiben am Dateiende
- "rw" zum Lesen und Schreiben

➤ Ergebnis von `fopen`:
Zeiger auf einen Datentyp `FILE`, der einen Dateikanal beschreibt
im Fehlerfall wird ein `NULL`-Zeiger geliefert

3 Zeichenweise Lesen und Schreiben

- Lesen eines einzelnen Zeichens

◆ von der Standardeingabe

```
int getchar( )
```

◆ von einem Dateikanal

```
int getc(FILE *fp )
```

- lesen das nächste Zeichen
- geben das gelesene Zeichen als `int`-Wert zurück
- geben bei Eingabe von `CTRL-D` bzw. am Ende der Datei `EOF` als Ergebnis zurück

- Schreiben eines einzelnen Zeichens

◆ auf die Standardausgabe

```
int putchar(int c)
```

◆ auf einen Dateikanal

```
int putc(int c, FILE *fp )
```

- schreiben das im Parameter `c` übergebene Zeichen
- geben gleichzeitig das geschriebene Zeichen als Ergebnis zurück

2 Öffnen und Schließen von Dateien (2)

- Beispiel:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    FILE *eingabe;

    if (argv[1] == NULL) {
        fprintf(stderr, "keine Eingabedatei angegeben\n");
        exit(1); /* Programm abbrechen */
    }

    if ((eingabe = fopen(argv[1], "r")) == NULL) {
        /* eingabe konnte nicht geöffnet werden */
        perror(argv[1]); /* Fehlermeldung ausgeben */
        exit(1); /* Programm abbrechen */
    }

    ... /* Programm kann jetzt von eingabe lesen */
}
```

- Schließen eines E/A-Kanals

```
int fclose(FILE *fp)
```

➤ schließt E/A-Kanal `fp`

3 Zeichenweise Lesen und Schreiben (2)

- Beispiel: `copy`-Programm, Aufruf: `copy Quelldatei Zieldatei`

```
#include <stdio.h>
// Teil 1: Aufrufargumente auswerten

int main(int argc, char *argv[]) {
    FILE *quelle, *ziel;
    int c; /* gerade kopiertes Zeichen */

    if (argc < 3) { /* Fehlermeldung, Abbruch */ }

    if ((quelle = fopen(argv[1], "r")) == NULL) {
        perror(argv[1]); /* Fehlermeldung ausgeben */
        exit(EXIT_FAILURE); /* Programm abbrechen */
    }

    if ((ziel = fopen(argv[2], "w")) == NULL) {
        /* Fehlermeldung, Abbruch */
    }

    while ( ( c = getc(quelle) ) != EOF ) {
        putc(c, ziel);
    }

    fclose(quelle);
    fclose(ziel);
}
```

3 Zeilenweise Lesen und Schreiben (3)

- Lesen einer Zeile von der Standardeingabe

```
char *fgets(char *s, int n, FILE *fp)
```

- ▶ liest Zeichen von Dateikanal `fp` in das Feld `s` bis entweder `n-1` Zeichen gelesen wurden oder `'\n'` oder `EOF` gelesen wurde
- ▶ `s` wird mit `'\0'` abgeschlossen (`'\n'` wird nicht entfernt)
- ▶ gibt bei `EOF` oder Fehler `NULL` zurück, sonst `s`
- ▶ für `fp` kann `stdin` eingesetzt werden, um von der Standardeingabe zu lesen

- Schreiben einer Zeile

```
int fputs(char *s, FILE *fp)
```

- ▶ schreibt die Zeichen im Feld `s` auf Dateikanal `fp`
- ▶ für `fp` kann auch `stdout` oder `stderr` eingesetzt werden
- ▶ als Ergebnis wird die Anzahl der geschriebenen Zeichen geliefert

4 Formatierte Ausgabe (2)

- Zeichen im `format`-String können verschiedene Bedeutung haben

- ▶ normale Zeichen: werden einfach auf die Ausgabe kopiert
- ▶ Escape-Zeichen: z. B. `\n` oder `\t`, werden durch die entsprechenden Zeichen (hier Zeilenvorschub bzw. Tabulator) bei der Ausgabe ersetzt
- ▶ Format-Anweisungen: beginnen mit `%`-Zeichen und beschreiben, wie der dazugehörige Parameter in der Liste nach dem `format`-String aufbereitet werden soll

- Format-Anweisungen

- `%d, %i` `int` Parameter als Dezimalzahl ausgeben
- `%f` `float` Parameter wird als Fließkommazahl (z. B. 271.456789) ausgegeben
- `%e` `float` Parameter wird als Fließkommazahl in 10er-Potenz-Schreibweise (z. B. 2.714567e+02) ausgegeben
- `%c` `char`-Parameter wird als einzelnes Zeichen ausgegeben
- `%s` `char`-Feld wird ausgegeben, bis `'\0'` erreicht ist

4 Formatierte Ausgabe

- Bibliotheksfunktionen — Prototypen (Schnittstelle)

```
int printf(char *format, /* Parameter */ ... );
int fprintf(FILE *fp, char *format, /* Parameter */ ... );
int sprintf(char *s, char *format, /* Parameter */ ... );
int snprintf(char *s, int n, char *format, /* Parameter */ ... );
```

- Die statt ... angegebenen Parameter werden entsprechend der Angaben im `format`-String ausgegeben

- ▶ bei `printf` auf der Standardausgabe
- ▶ bei `fprintf` auf dem Dateikanal `fp` (für `fp` kann auch `stdout` oder `stderr` eingesetzt werden)
- ▶ `sprintf` schreibt die Ausgabe in das `char`-Feld `s` (achtet dabei aber nicht auf das Feldende -> Pufferüberlauf möglich!)
- ▶ `snprintf` arbeitet analog, schreibt aber maximal nur `n` Zeichen (`n` sollte natürlich nicht größer als die Feldgröße sein)

5 Formatierte Eingabe

- Bibliotheksfunktionen — Prototypen (Schnittstelle)

```
int scanf(char *format, /* Parameter */ ... );
int fscanf(FILE *fp, char *format, /* Parameter */ ... );
int sscanf(char *s, const char *format, /* Parameter */ ... );
```

- Die Funktionen lesen Zeichen von `stdin` (`scanf`), `fp` (`fscanf`) bzw. aus dem `char`-Feld `s`.

- `format` gibt an, welche Daten hiervon extrahiert und in welchen Datentyp konvertiert werden sollen

- Die folgenden Parameter sind Zeiger auf Variablen der passenden Datentypen (bzw. `char`-Felder bei Format `%s`), in die die Resultate eingetragen werden

- relativ komplexe Funktionalität, hier nur Kurzüberblick für Details siehe Manual-Seiten

5 Formatierte Eingabe (2)

- *White space* (Space, Tabulator oder Newline \n) bildet jeweils die Grenze zwischen Daten, die interpretiert werden
 - ▶ *white space* wird in beliebiger Menge einfach überlesen
 - ▶ Ausnahme: bei Format-Anweisung %c wird auch *white space* eingelesen
- Alle anderen Daten in der Eingabe müssen zum `format`-String passen oder die Interpretation der Eingabe wird abgebrochen
 - ▶ wenn im format-String normale Zeichen angegeben sind, müssen diese exakt so in der Eingabe auftauchen
 - ▶ wenn im Format-String eine Format-Anweisung (%...) angegeben ist, muß in der Eingabe etwas hierauf passendes auftauchen
 - ↳ diese Daten werden dann in den entsprechenden Typ konvertiert und über den zugehörigen Zeiger-Parameter der Variablen zugewiesen
- Die `scanf`-Funktionen liefern als Ergebnis die Zahl der erfolgreich an die Parameter zugewiesenen Werte

6 Fehlerbehandlung

- Fast jeder Systemcall/Bibliotheksaufruf kann fehlschlagen
 - ◆ Fehlerbehandlung unumgänglich!
- Vorgehensweise:
 - ◆ Rückgabewerte von Systemcalls/Bibliotheksaufrufen abfragen
 - ◆ Im Fehlerfall (meist durch Rückgabewert -1 angezeigt): Fehlercode steht in der globalen Variable `errno`
- Fehlermeldung kann mit der Funktion `perror` auf die Fehlerausgabe ausgegeben werden:

```
#include <errno.h>
void perror(const char *s);
```

5 Formatierte Eingabe (3)

| | |
|------------------------|---|
| %d | int |
| %hd | short |
| %ld | long int |
| %lld | long long int |
| | |
| %f | float |
| %lf | double |
| %Lf | long double |
| analog auch %e oder %g | |
| | |
| %c | char |
| %s | String, wird automatisch mit '\0' abgeschl. |

- nach % kann eine Zahl folgen, die die maximale Feldbreite angibt
 - %3d = 3 Ziffern lesen
 - %5c = 5 char lesen (Parameter muß dann Zeiger auf char-Feld sein)
 - ▶ %5c überträgt exakt 5 char (hängt aber kein '\0' an!)
 - ▶ %5s liest max. 5 char (bis white space) und hängt '\0' an

Beispiele:

```
int a, b, c, d, n;
char s1[20]="XXXXXX", s2[20];
n = scanf("%d %d %3d %5c %s %d",
          &a, &b, &c, s1, s2, &d);
Eingabe: 12 1234567 sowas hmmm
Ergebnis: n=5, a=12, b=12, c=345
s1="67 soX", s2="was"
```