

Überblick: Teil D Betriebssystemabstraktionen

15 Nebenläufigkeit

16 Ergänzungen zur Einführung in C

17 Betriebssysteme

18 Dateisysteme

19 Programme und Prozesse

20 Speicherorganisation

21 Nebenläufige Prozesse



Zeiger, Felder und Zeichenketten

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

- 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";
```



amessage $\equiv$

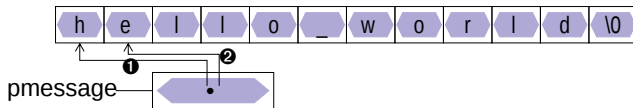
- es wird ein Speicherbereich für 16 Bytes reserviert und die Zeichen werden in diesen Speicherbereich hineinkopiert
- amessage ist ein konstanter Zeiger auf den Anfang des Speicherbereichs und kann nicht verändert werden
- der Inhalt des Speicherbereichs kann aber modifiziert werden
`amessage[0] = 'h';`



... Zeiger, Felder und Zeichenketten (2)

- 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*/
```

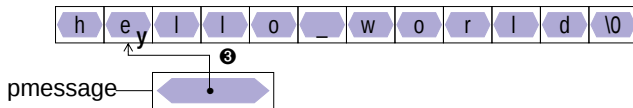
- es wird ein Speicherbereich für einen Zeiger reserviert (z. B. 4 Byte) und der Compiler legt die Zeichenkette `hello world` an irgendeiner Adresse im Speicher des Programms ab
- `pmessage` ist ein variabler Zeiger, der mit dieser Adresse initialisiert wird, aber jederzeit verändert werden darf
`pmessage++;`



... 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 = '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

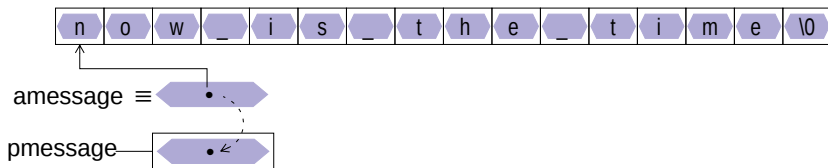


... Zeiger, Felder und Zeichenketten (4)

- 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



■ 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++ )
        ;
}
```



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

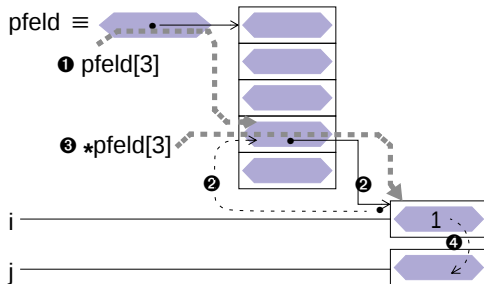
- Zugriffe auf das Objekt, auf das ein Zeiger des Feldes verweist

```
j = *pfeld[3]; ④
```

①

③

④

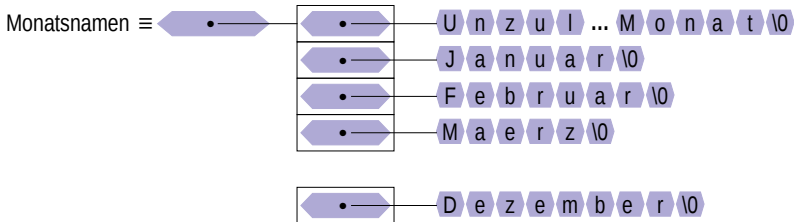


Felder von Zeigern (2)

- Beispiel: Definition und Initialisierung eines Zeigerfeldes:

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

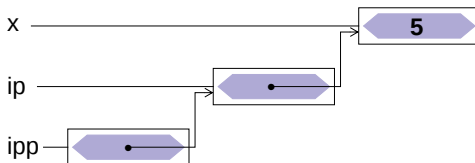
    return ( (n<0 || n>12) ?
             Monatsnamen[0] : Monatsnamen[n] );
}
```



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)



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)  
{  
    ...  
}
```

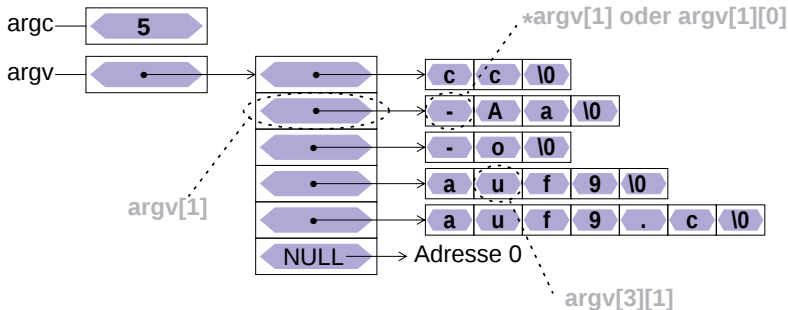
- der Parameter **argc** enthält die Anzahl der Argumente, mit denen das Programm aufgerufen wurde
- der Parameter **argv** ist ein Feld von Zeiger auf die einzelnen Argumente (Zeichenketten)
- der Kommandoname wird als erstes Argument übergeben (**argv[0]**)



Datenaufbau

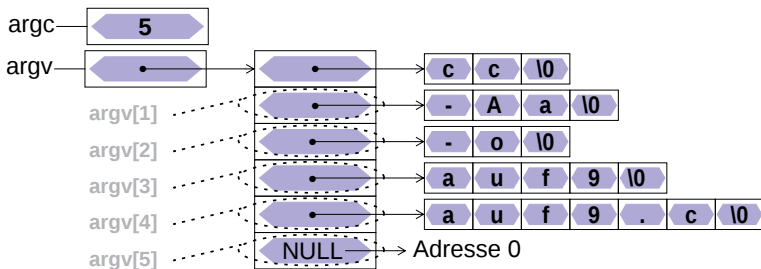
Kommando: `cc -Aa -o auf9 auf9.c`

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



- 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 ", argv[i]);  
    }  
    printf("\n");  
    ...  
}
```



Verbund-Datentypen / Strukturen (structs)

- Zusammenfassen mehrerer Daten zu einer Einheit (ein zusammenfassender Datentyp)
- Strukturdeklaration

```
struct person {  
    char name[20];  
    int  alter;  
};
```

- Definition einer Variablen vom Typ der Struktur

```
struct person p1;
```

- Zugriff auf ein Element der Struktur

```
p1.alter = 20;
```



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
 - eine Struktur kann die Adresse einer weiteren Struktur desselben Typs enthalten



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 gut lesbar!

- Syntaktische Verschönerung

➡ `->-Operator`

`pstud->alter = 21;`



Rekursive Strukturen

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

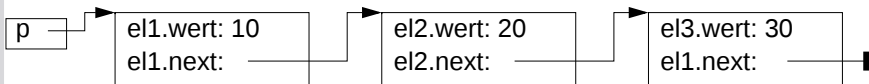
- ➡ Programmieren rekursiver Datenstrukturen
z. B. verkettete Listen



Verkettete Listen

- Mehrere Strukturen desselben Typs werden über Zeiger miteinander verkettet

```
struct liste { int wert; struct liste *next };  
struct liste el1, el2, el3;  
struct liste *p = &el1;  
el1.wert = 10; el2.wert = 20; el3.wert = 30;  
el1.next = &el2; el2.next = &el3; el3.next = NULL;
```



- Laufen über eine verkettete Liste

```
int summe = 0;  
while (p != NULL) {  
    summe += p-> wert;  
    p = p->next;  
}
```

