

F Zeiger, Felder und Strukturen in C

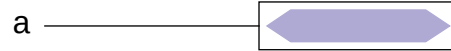
F.1 Zeiger(-Variablen)

1 Einordnung

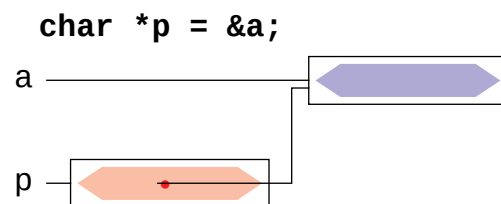
- **Konstante:**
Bezeichnung für einen Wert

'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

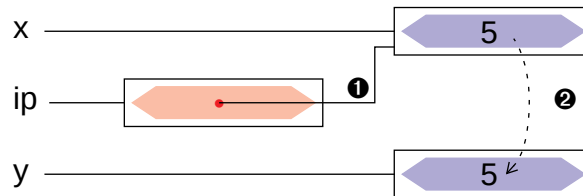
```
int x = 5;

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*

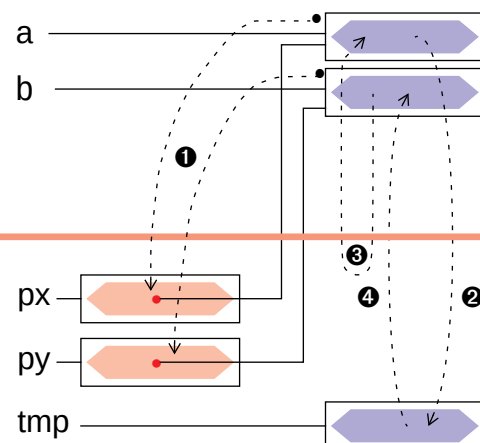
6 ... Zeiger als Funktionsargumente (2)

- Beispiel:

```
main(void) {
    int a, b;
    void swap (int *, int *);
    ...
    swap(&a, &b); ①
}
```

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

    tmp = *px; ②
    *px = *py; ③
    *py = tmp; ④
}
```



F.2 Felder

1 Eindimensionale Felder

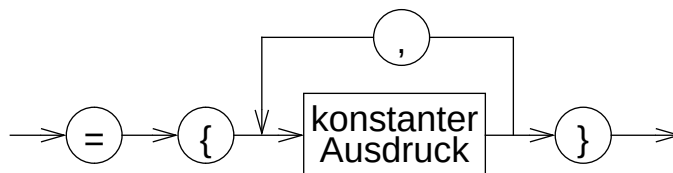
- eine Reihe von Daten desselben Typs kann zu einem **Feld** zusammengefasst werden
- bei der Definition wird die Anzahl der Feldelemente angegeben, die Anzahl ist konstant!
- der Zugriff auf die Elemente erfolgt durch **Indizierung**, beginnend bei Null
- Definition eines Feldes



- 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

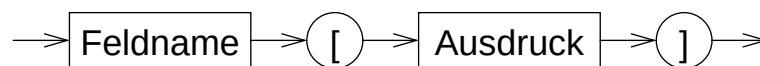
2 ... Initialisierung eines Feldes (2)

- 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'
```

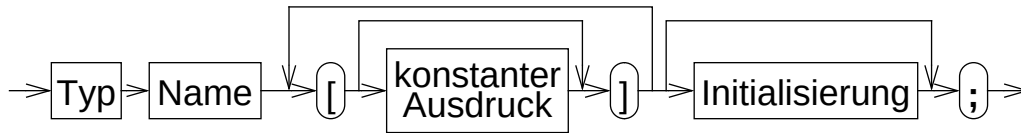
- Beispiel Vektoraddition:

```
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

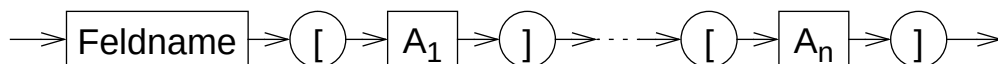


- Beispiel:

```
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:

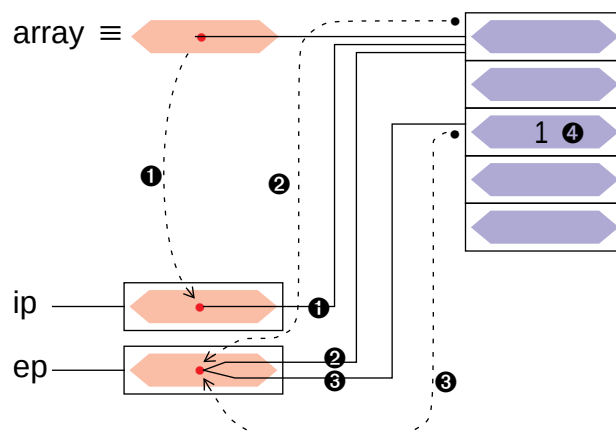
```
int array[5];

int *ip = array; ❶

int *ep;
ep = &array[0]; ❷

ep = &array[2]; ❸

*ep = 1; ❹
```

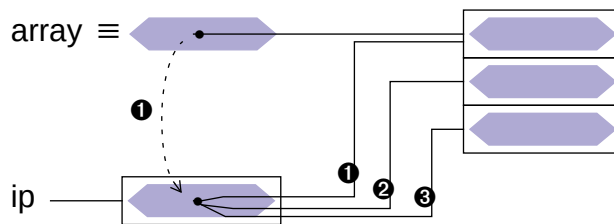


1 Arithmetik mit Adressen

- ++ -Operator: Inkrement = nächstes Objekt

```
int array[3];
int *ip = array; ❶

ip++; ❷
ip++; ❸
```



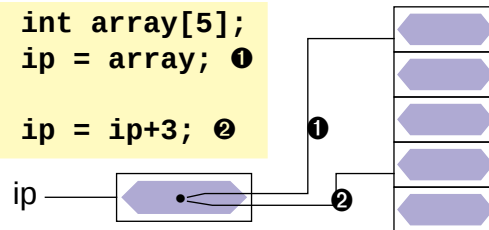
- -- -Operator: Dekrement = vorheriges Objekt

- +, -
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 !!

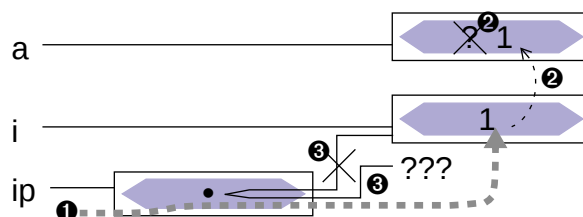
2 Vorrangregeln bei Operatoren

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;
```

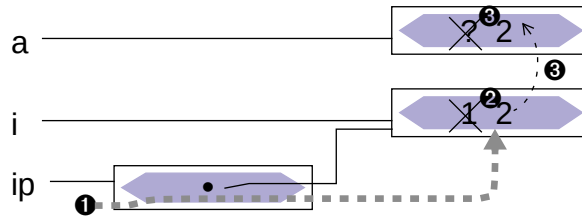
```
a = *ip++;
  ↳ (1) a = *ip++;
    ↳ (2) a = *ip++;
      ↳ (3) a = *ip++;
```



3 Beispiele (2)

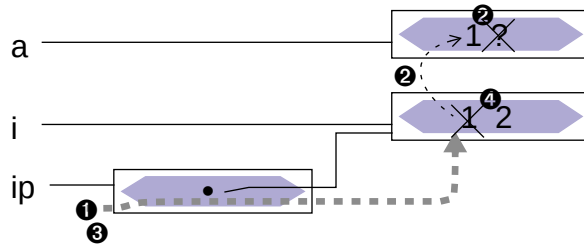
```
int a, i, *ip;
i = 1;
ip = &i;

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

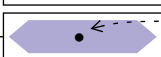
ep = &array[i]; *ep = 1;
ep = array+i; *ep = 1;
```

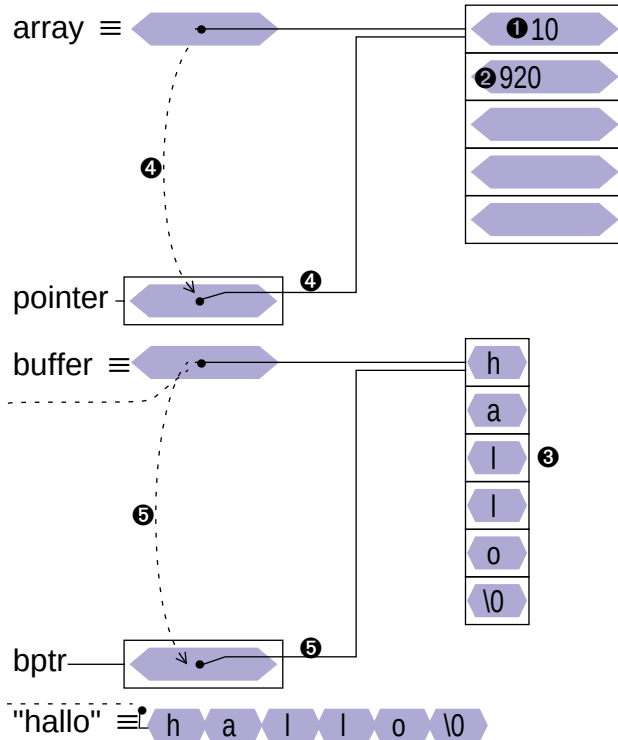
4 Zeigerarithmetik und Felder

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

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

Formale Parameter
der Funktion strcpy

ziel — 
 quelle — 



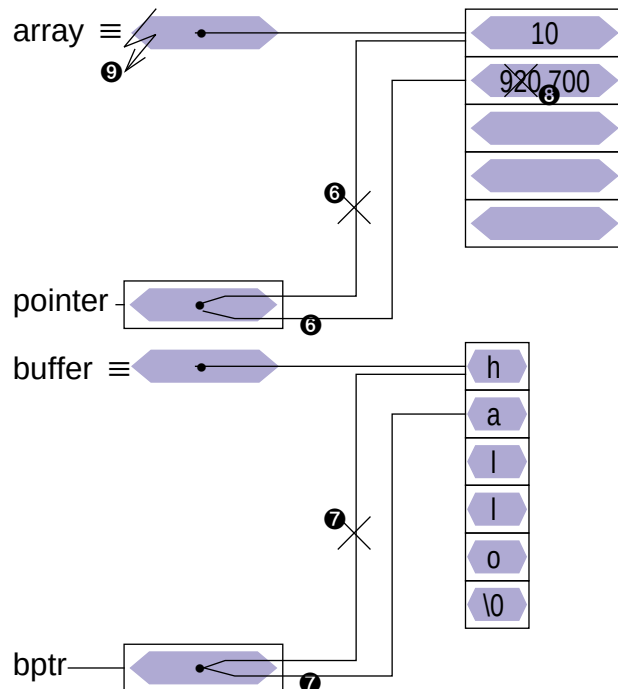
4 Zeigerarithmetik und Felder

```
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++;
```



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

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!

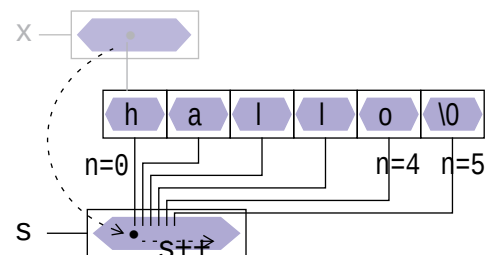
- im Unterschied zu einer Variablendefinition!!!


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

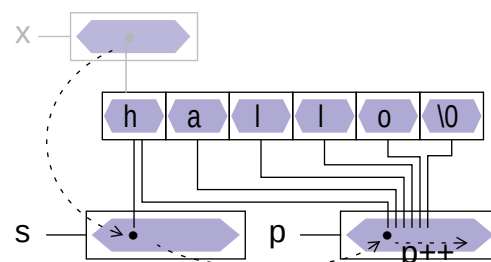
F.5 Zeiger, Felder und Zeichenketten

- 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);
}
```



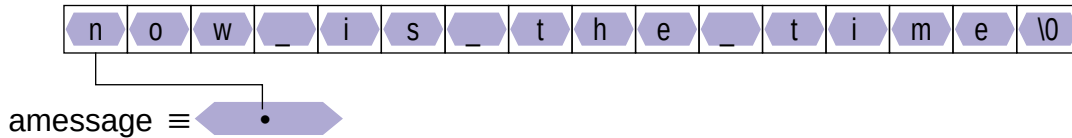
```
/* 2. Version */
int strlen(char *s)
{
    char *p = s;
    while (*p != '\0')
        p++;
    return(p-s);
}
```



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";
```

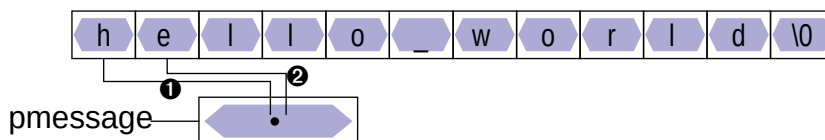


- es wird ein Speicherbereich für 16 Bytes reserviert und die Zeichen werden in diesen Speicherbereich hineinkopiert
- amessage ist ein konstanter Zeiger auf den Anfang des Speicherbereichs und kann nicht verändert werden
- der Inhalt des Speicherbereichs kann aber modifiziert werden
amessage[0] = 'h';

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";
```



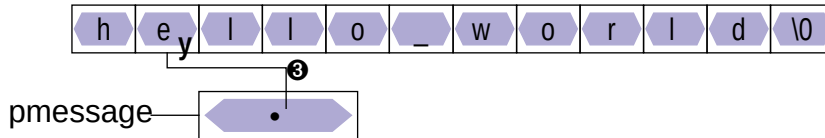
```
pmessage++; ②  
printf("%s", pmessage); /* gibt "ello world" aus */
```

- 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 ⚡
```

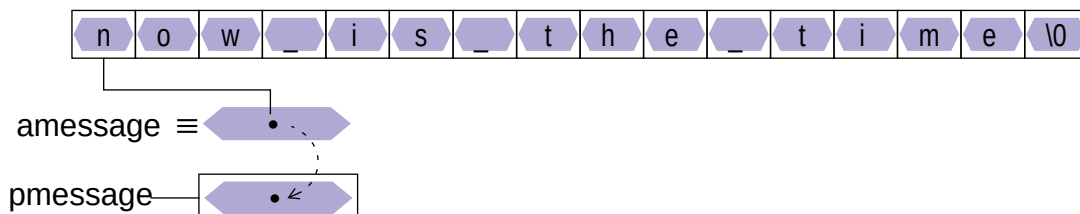
- der Speicherbereich von hello world darf aber nicht verändert werden
 - manche Compiler legen solche Zeichenketten in schreibgeschütztem Speicher an
 - ➔ Speicherschutzverletzung beim Zugriff
 - sonst funktioniert der Zugriff obwohl er nicht erlaubt ist
 - ➔ Programm funktioniert nur in manchen Umgebungen

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;
```

$d = (i * f);$

→float
→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 *)