

I) Auswertung von Ausdrücken (expressions) in C:

- ▶ von links → rechts
- ▶ * / vor + -

Beispiel:

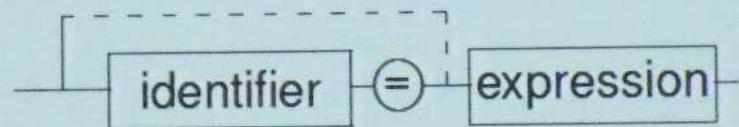
int a, b, c;	
a = 2; b = 7;	
c = a * 4 - b;	△ $(2 \cdot 4) - 7 = 1$
c = b / 2 * 4;	△ $(7/2) \cdot 4 = 3 \cdot 4 = 12$
c = 4 * b / 2;	△ $(4 \cdot 7)/2 = 28/2 = 14$

Klammern regeln Priorität:

c = (a + b) / (a - b);	△ $9/(-5) = -1$
c = a + (b + 1);	△ $2 + 8 = 10$

Im Zweifel: **lieber klammern!**

Syntaxdiagramm einer Zuweisung:



x + 1 = a * 7; ↴
y + 1 = x = a * 7 ↴



Bedingungen und Vergleiche

III) Aufstellen von Bedingungen (clauses) in C:

- ▶ Typisch durch Vergleich:

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

- ▶ Numerischer Wert einer logischen Aussage:

0	\triangleq	false
$\neq 0$	\triangleq	true

Verknüpfungen logischer Aussagen:

- ▶ logische **UND**-Verknüpfung – `&&`
- ▶ logische **ODER**-Verknüpfung – `||`



if-Anweisung

Bedingte Ausführung durch eine if-Abfrage:

```
if (a < 0)
    a = -a;      Ergebnis |a|
```

```
if (a == -1)
    a = 0;      Ergebnis {0   falls a = -1
                           a   sonst}
```

[besser: if (a == (-1))]
Klammern schaden kaum!



Allgemeine Syntax einer einfachen if-Abfrage:

```
if ( expression )
    statement
```

expression (int) $\neq 0 \Leftrightarrow$ statement wird ausgeführt

Beispiel:

```
if (n == 10)
    n = 0;
```

Vorsicht:

```
if (a < 5)
    a = a + 1;
    n = n + 1;
```

Hier bezieht sich if nur auf das nächste statement! Was immer der Wert von a ist, in jedem Fall wird $n=n+1;$ ausgeführt.

Abhife:

```
if (a < 5) {
    a = a + 1;
    n = n + 1; }
```



Häufig gemachter Fehler:

```
if (a = -1)
    a = 0;      Ergebnis a = 0
```

Erklärung:

- 1) $a = -1$; ist eine Zuweisung, kein Vergleich
- 2) Die Zuweisung liefert einen Wert zurück, nämlich -1
- 3) -1 ist ungleich 0 , also true
→ Die Bedingung ist immer erfüllt!

Priorität Vergleich **kleiner** als $+, -, *, /$

$$a + b < b + c \triangleq (a + b) < (b + c)$$

(\notin Klammern können kaum schaden \notin)

if-else-Anweisung

Verzweigungen im Programm:

```
int a = 1;  
if (a > 1)  
    a = a - 1;  
else  
    a = a + 1;      Ergebnis a = 2
```

Allgemeine Syntax der if-else-Kontrollstruktur:

```
if (expression) {  
    statement1  
} else {  
    statement2  
}
```

expression (int) $\neq 0 \Leftrightarrow$ statement1 wird ausgeführt
expression = 0 \Leftrightarrow statement2 wird ausgeführt.

while-Schleife

Wiederholte Ausführung von Anweisungen:

while (expression) {statements}

↔ Führe statements aus solange expression true (d.h. $\neq 0$)

Beispiel 1:

int a = 10;

while (a > 0)

a = a - 4; wird 3 mal ausgeführt $\Rightarrow a = -2$

Beispiel 2:

int a = 0;

while(a)

a = a + 7; wird nicht ausgeführt $\Rightarrow a = 0$

Beispiel 3:

int a = 1, b;

while(a)

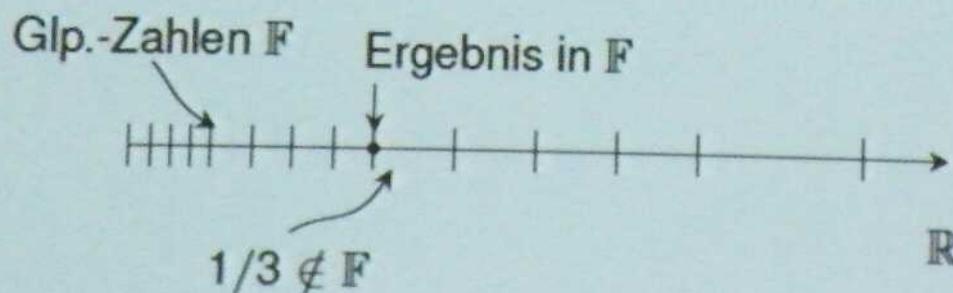
b = a + 3; Endlosschleife

Gleitpunkt Datentypen

float (single)	4 Byte	1 bit	23 bit	8 bit
double	8 Byte	1 bit	52 bit	11 bit
Darstellung		sign	mantissa	exponent
		(s)	$(m_1 m_2 \dots m_{23/52})$	$(e_1 e_2 \dots e_{8/11})$
Interpretation	$(-1)^s \cdot 1.m_1 m_2 \dots m_{23/52} \cdot 2^{(e)}$			

float $\triangleq \sim 6$ Dez. Genauigkeit; $-127 \leq e \leq 127 \rightarrow \text{Range } \pm 10^{\pm 38}$
double $\triangleq \sim 16$ Dez. Genauigkeit; $-1023 \leq e \leq 1023 \rightarrow \text{Range } \pm 10^{\pm 308}$

IEEE 754 Arithmetik Standard legt u.a. das Ergebnis jeder Gleitpunkt-Operation fest



Themenübersicht Vorlesung 3

- Gleitpunktarithmetik
- Typkonvertierung int, float, double
- do...while - Schleife
- Syntax: %f, %e, %lf, %le, %ld, %c, short int, long int, char, do...while

Gleitpunktarithmetik

Eine Glp-Operation nach IEEE 754 Standard hat kleinst-möglichen Fehler. Das gilt nur für eine einzelne Operation.

Bsp.:

float x = 1e6;	1 Million
x = x-0.01;	$\Rightarrow x = 1e6 !$
x = x-1e6;	$\Rightarrow x = 0 !$

basis = 10

Glp.: i.d.R. genähert, mit Dezimalpunkt
 int : exakt, ohne Dezimalpunkt

Glp.	+3.704	5e-3	= 0.005
	-4	4.5e+3	= 45
	.35	-.4e+1	= -4
	4e7		

Typkonvertierung int, float, double

```
int a;  
a = 3.5;  ⇒ a = 3
```

Vorsicht: Compiler meldet
keinen Fehler!

Bsp.: float f;
double d;
f = d = 0.1;

```
d = f = 0.1;
```

Test: printf("%e\n", d-0.1);

```
float a;  
a = 3;  ⇔ a = 3.0;
```

automatische Konversion

d=0.1 wird ausgeführt
Ergebnis double
wird an f zugewiesen
(Konversion nach single)

Ergebnis f=0.1 ist
single (nur 24 Bit)
wird an d zugewiesen. ungenau !

Typkonvertierung int, float, double

Formate	printf()	scanf()
float ohne Exp. (z.B.: 0.1000)	%f	"
float mit Exp. (z.B.: 1.000e-01)	%e	"
double ohne Exp.	%f	%lf
double mit Exp.	%e	%le
long int	%ld	"
int	%d	"
char (z.B. printf("%c",'a'));	%c	"

printf konvertiert float implizit in double
 ⇒ keine unterschiedlichen Formatierungszeichen nötig !

Automatische Typanpassung int → double

short int	→	float, double	ohne Fehler
long int	→	double	ohne Fehler
long int	→	float	mit Fehler

Bsp. $e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$ sehr gute Konvergenz für $|x| \leq 1$.

z.B.: $e = 1 + 1 + 0.5 + 0.16\bar{6} + 0.04\bar{1}\bar{6} + 0.008\bar{3} + \dots$
 $1/10! \sim 2.7e-7$, $1/20! \sim 4.1e-19$

Berechnung von e:

```

1 #include <stdio.h>
2 int main(void) {
3     float y = 1.0, /* y double => Error<=1e-7 */
4         t = 1.0, x = 1.0;
5     int i = 0;
6     while (t > 1e-7) {
7         i = i + 1;
8         t = t * x / i; /* Konversion int i -> float */
9         y = y + t;
10    }
11    printf("e ist (naeherungsweise): %f", y);
12    return(0);
13 }/* main */
```

Problem: 1e-7 absolute, keine relative Genauigkeit.

besser:

```
1 float y = 1.0, t = 1.0, x = 1.0, yold = 0.0;
2 int i = 0;
3 while (y != yold) {
4     i = i + 1;
5     t = t * x / i;
6     yold = y;
7     y = y + t;
8 }
```

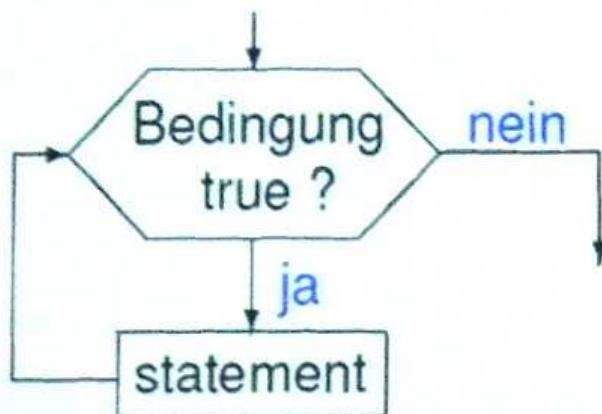
Summieren, bis sich Summenwert (glp-mäßig) nicht mehr ändert.

Vorteil: Programm identisch für double statt float.

while- vs. do...while - Schleife

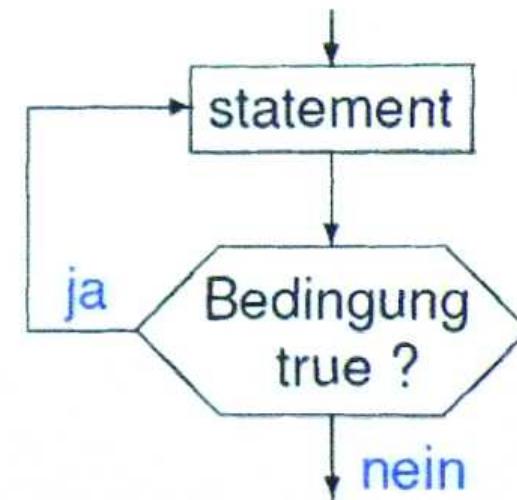
while-Schleife

while (Bedingung) statement



do...while - Schleife

do statement while (Bedingung)



Bsp. do {

```

        yold = y
        y = y-f(x)/fs(x);
    }while(y!=yold);
  
```

für $fs \triangleq f'$

Bsp. geg.: $n \in \mathbb{N}, n \geq 0$
ges.: $n!$ [= $\prod_{i=1}^n i$] , per Def. leeres Produkt := 1.

```
int n = ..., f = 1, i = 1;  
while ( i<n ) {  
    i = i+1;  
    f = f*i;  
}
```

korrekt für $n \geq 2$, und auch für $n \in \{0, 1\}$.

Funktionen in C

Beispiel:

```

Signature
(B) _____ (A)
double sqr(double x) {
    double y; }(C)
    y = x * x; }(D)
    return y; }(E)
}

```

Syntax:

- `sqr` – Name der Funktion
- (`A`) – Eingabeargument(e)
- (`B`) – Datentyp der Ausgabe
- (`C`) – (lokaler) Vereinbarungsteil
- (`D`) – (lokaler) Anweisungsteil
- (`E`) – Rückgabewert

```
Aufruf: double x, z;  
...  
z = sqr(x-1);
```

Semantik:

- 1) Berechnung des Wertes des Eingabearguments.
z.B. für $x=5 \Rightarrow x-1=4$
- 2) Einsetzen dieses Wertes in Eingabeargument der Funktion,
d.h. die Variable x hat innerhalb der Funktion den Wert 4
- 3) Ausführen des Anweisungsteils (mit lokalen Variablen)
- 4) Rückgabe des Funktionswertes (hier 16.0)

Nachtrag zum ersten Beispiel: (kürzer)

```
double sqr(double x) {  
    return (x*x);  
} /* sqr */
```

Regeln für Funktionsdefinition

I) Parameterübergabe durch **call-by-value**

Beispiel:

```
int factorial(int n) {  
    int res = 1;  
    do {  
        res = res*n;  
        n = n - 1;  
    } while (n > 0)  
    return (res);  
} /* factorial */
```

Aufruf:

```
n = 5;  
m = factorial(n);
```

Durch Aufruf wird der Wert von n nicht verändert!

- Alles innerhalb der Funktion Vereinbarte ist **lokal** (keine Verbindung zur Außenwelt).

Beispiel: #include <stdio.h>

```
double fabs(double y) {
    double x;
    if (y < 0) x = -y; else x = y;
    return x;
} /* fabs */

int main(void) {
    double x = -3.14;
    printf("%f\n", fabs(x));
    return 0;
} /* main */
```

2 mal Variable x: haben nichts miteinander zu tun!

```
double fabs(double x) { ... }
```

- III) Default Ergebnis-Typ einer Funktion ist int.
→ Das bedeutet, läßt man das double vor der Funktion fabs weg, ist das Programm syntaktisch korrekt und produziert irgendeinen Unsinn!

Beispiel:

```
int NearDoubleZero(double p, double q) {
    /* returns 1 if x^2 + p*x + q has */
    /* (nearly) a double zero */
    double discr, d;
    discr = p*p/4 - q;
    d = fabs(p*q);
    return (fabs(discr) < 1e-12 * d);
} /* NearDoubleZero */
```

(Für `fabs` wird `#include <math.h>` benötigt.)

→ Beobachten Sie die Fehlermeldung des Compilers, wenn Sie z.B. `NearDoublezero(1.0, 2.0)` aufrufen wollen.

Abgekürzte Notation

$a = a * 5$ \Leftrightarrow $a := 5;$
 $i = i + 2;$ \Leftrightarrow $i += 2;$
 $x = x/y;$ \Leftrightarrow $x /= y;$
 $Vol = Vol - 4.3;$ \Leftrightarrow $Vol -= 4.3;$

$x = x + 1;$ \Leftrightarrow $x += 1;$ \Leftrightarrow $x++;$
 $i = i - 1;$ \Leftrightarrow $i -= 1;$ \Leftrightarrow $i--;$

for-Schleife, Beispiele

```
Bsp.1   for (i=0; i<5; i++)
        printf("i = %d i*i = %d\n", i, i*i)
                druckt i,  $i^2$  für  $0 \leq i \leq 4$ 
```

Bsp.2 `for (i=5; i<3;) { ... }` Leere Schleife (tut gar nichts)

Bsp.3 `for (; ;)` Endlosschleife

Bsp.4 Fibonacci-Zahlen

```

a = b = 1;
for (i=0; i<10; i++) {
    c = a+b;
    a = b;
    b = c;
}

```

1 1 2 3 5 8 13

```
for (expr1; expr2; expr3)
    statement;

do
    expr1;
    while(expr2) {
        statement;
        expr3;
    }
```

- Arrays
 - Adressen
 - Zeiger
 - Makros
 - Syntax: A[...], A+..., *(A+...), *x, &x, sizeof(),
#define, A[...][...]

double x; → 8 Byte werden für x reserviert

`double A[3];` → $3 \cdot 8 = 24$ Byte werden für A reserviert

Zugriff A[0] A[1] A[2] ! Beginn mit Index 0

arbeiten wie mit Variablen

A[3] 4

Bsp. int a[3], k, i;
 for (i=0; i<3; i++)
 a[i] = i*i;
 a[3] = -2; Ke

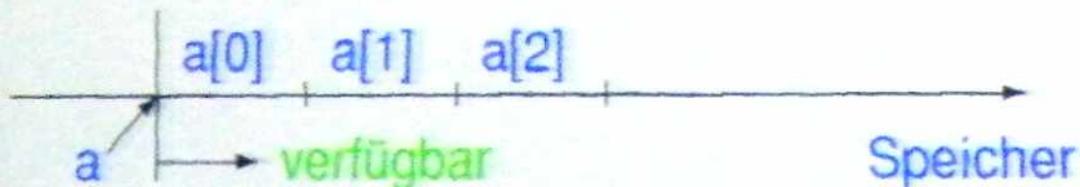
Keine Fehlermeldung !
Unvorhersehbares Resultat !

Arrays und alle Variablen werden nicht vorinitialisiert !

```
s = 0; /* nicht vergessen! */
for (i=0; i<3; i++)
    s+= a[i];
```

```
double a[3];  ⇒
```

- Speicher für 3 double (24 Byte) reservieren
 - Anfangsadresse → a
 - Typ a[i] ist double



a: **Zeiger** auf den Anfang des Speicherbereiches für a

`a[0]`: Inhalt der ersten 8 Byte ab `a` interpretiert als double

a[1] : Inhalt der Bytes 8...15 ab a

Adressen, Zeiger

Zugriff: Adresse &
Inhalt *

double x; x Variable vom Typ double \Rightarrow Inhalt double
 $\&x$ Adresse von x

double *x; x Zeiger auf double
 $*x$ Inhalt von x

a[2] = 7.0; • Addiere zur Adresse a den Wert
2*sizeof(a[0]) (=16) \Rightarrow Adresse von a[2]
• Speichere Bitmuster (8 Byte) von 7.0 dorthin

Mit Zeigern rechnen

```
double A[5];
```

A[2]	Inhalt(adr(A)+2·sizeof(*A))
A+2	<u>adr(A) + 2 · sizeof(*A)</u>
* (A+2)	Inhalt()

Vorsicht: * bindet enger als +, d.h.: *A+2 \Leftrightarrow (*A)+2 \Leftrightarrow A[0]+2

Bsp.

```
double A[5], s = 0.0, *pA;  
int i;  
pA = A;  
... /* Initialisierung von A */  
for(i=0; i<5; i++)  
    s+= A[i]; /* s =  $\sum_{i=0}^4 A[i]$  */  
    ⇔ s+= *(A+i);  
    ⇔ s+= *pA; pA++;  
    ⇔ s+= *pA; pA++;  
    ⇔ s+= *pA++;
```

Präprozessor

z.B. `#include<stdio.h>`
`#define MAX_N 100`

...

Erst arbeitet der Präprozessor und übergibt
dann das (veränderte, ergänzte) Programm an den Compiler.

Bsp. `#define` → textuelle Ersetzung

```
#define MAX_N 100
double A[MAX_N][MAX_N];      /* ≡ A[100][100] */
int i, j;
for (i=0; i<MAX_N; i++)
    for (j=0; j<MAX_N; j++)
        A[i][j] = 0.0;
```

Vorteil: Dimensionierung (`MAX_N`) nur an einer Stelle zu ändern

- Vorl.1
- Vorl.2
- Vorl.3
- Vorl.4
- Vorl.5
- Vorl.6
- Vorl.7
- Vorl.8
- Vorl.9
- Vorl.10
- Vorl.11
- Vorl.12

Themenübersicht Vorlesung 6

- Makros
- Pointer (Zeiger)
- Adressübergabe
- dynamische Felder
- Syntax: `exp()`, `scanf("%d", &j)`, `(int*)malloc(n)`,
`free()`

Präprozessormakros

Wiederholung: `#include <stdio.h>`
`#define MAX_N 100`

- Der Compiler sieht nur das (durch den Präprozessor) veränderte Programm!

Beispiel:

```
#define double int B, A
int main(void) {
    double [9];
    ...
}

int main(void) {
    int B, A[9];
    ...
}
```

Makros (cont'd)

Üblicherweise das zu ersetzende nur in Großbuchstaben!

Beispiel:

```
#define FOREVER for ( ; ; )
```

Anwendung: FOREVER {
 ...
 break;
 }

Vorsicht: 1 : 1 Ersetzung ohne syntaktische/sematische Prüfung!

Beispiel: #define SQR(x) x*x

...

z = 3 + exp(SQR(z));

→ z = 3 + exp(z*z);

...

a = SQR(n+1);

→ a = n + 1*n + 1;

[= $2n + 1$ statt $(n + 1)^2$]

→ #define: Argumente immer Klammern!

```
#define SQR(x) ((x)*(x))  
a = ((n+1)*(n+1))
```

Pointer (cont'd)

```
double x, *y;  
x = 5.0;           x hat den Wert 5.0  
y = &x;             y zeigt auf Adresse von x  
*y = -3.0;         x hat den Wert -3.0
```

Anwendung: Parameterübergabe (bei Funktionen)

call-by-value:

- Der Wert der Parameter wird berechnet und übergeben
- Rechnen wie mit lokaler Variable

Beispiel:

```
long int factorial(long int n) {  
    long int res = 1;  
    while (n)  
        res *= n--;  
    return (res);  
} /*factorial*/
```

Beispiel: swap(x, y) \Leftrightarrow $x \rightarrow y$

```
void swap(double x, double y) {  
    double z;  
    z = x;  
    x = y;  
    y = z;  
} /*swap*/
```

Aufruf: swap(a, d); oder swap(a+1, x*7)

Ändert nicht die Variablen im aufrufenden Programm!

Adressen übergeben!
(call-by-reference)

Adressübergabe

Beispiel: $\text{swap}(x, y) \Leftrightarrow x \leftrightarrow y$

```
void swap(double *x, double *y) {  
    double z;  
    z = *x;  
    *x = *y;  
    *y = z;  
} /*swap*/
```

Aufruf: `swap(&a, &d);`

[nicht etwa `swap(a, d); ...`]

Benutzereingabe

```
int i = 9;  
printf("i = %d      i*i = %d \n", i, i*i);
```

→ Ausgabe: i = 9 i*i = 81

→ printf ist ein C-Programm [stdio.h]

```
scanf("%d", &j);  
„liest j“ ein, d.h. j muß geändert werden
```

⇒ Parameter &j

```
double x,y; scanf(%lf %d %lf", &x, &i, &y);  
input 3.0 47 1
```

Beispiel: $x, y \in \mathbb{R}^n \quad x^T y = \sum_{i=1}^n x_i y_i \in \mathbb{R}$

```
#define N 10
```

```
double dot(int n, double *x, double *y) {
    double res = 0.0;
    while(n) {
        res += (*x++) * (*y++);
        n--;
    }
    return (res);
} /* dot */
```

```
int main(void) {
    int i;
    double p[N], q[N];
    for (i = 0; i < N; i++)
        scanf("%lf %lf", &p[i], q+i);
    printf("x^T*y = %f\n", dot(N, p, q));
    return 0;
} /* main */
```

Dynamische Felder

Bisher: → Feldlängen konstant

z.B. i) double A[100];

ii) #define NMAX 50
double A[NMAX][NMAX];

iii) int n = 20;
double A[n];

↓

Auch innerhalb einer Funktion Feldlänge konstant

Vorlesung 1

Dynamische Feldlängen: `pA = (int*) malloc(n);`

- Reserviere n Byte Speicher
- return (Anfangsadresse)
- als Zeiger auf int

Prototyp malloc in stdlib :

`void *malloc(size_t size);`

Freigabe: `free(pA);`

Themenübersicht Vorlesung 7

- Dynamische Felder
- Rekursion
- Sortieren (Mergesort)
- char-Datentyp
- Syntax: #include<stdlib.h>,
 (...*)malloc(sizeof(...)),
 exit(),NULL,if...else...,char,%c

Dynamische Felder

- Zeiger auf Variable bestimmten Typs:

```
int *a;
```

```
int *b;
```

```
a = b;      Warning: Suspicious pointer conversion
```

- void *

Pointer "auf alles" kann nicht dereferenziert werden.

Anwendung:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(void) {
4     int n;
5     double *A;
6     scanf("%d", &n);
7     A = (double*) malloc (n*sizeof(double));
8     if (A==NULL) {
9         printf("Error: no memory");
10        exit (1);
11    }
12    return (0);
13 } /* main */
```

Beispiel für void *

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 void *allocate (int n){
4     void *p;
5     p = malloc(n);
6     if ( p == NULL ){
7         printf("not enough memory");
8         exit(1);
9     }
10    return(p);
11 }/* allocate */
12
13 int main (void){
14     int *x, n ,i;
15     double *B;
16     scanf("%d", &n);
17     x = (int *) allocate (n*sizeof(int));
18     B = (double *) allocate (2*n*sizeof(*B));
19     for (i = 0; i < n; i++){
20         *x++ = 0;
21         scanf("%lf", B++);
22     };
23     x -= n; B -= n; /* Zeiger auf Anfang zurücksetzen */
24     return (0);
25 }/* main */

```

Rekursive Funktionen

“Funktion ruft sich selber auf”

Bsp.: $n! := \prod_{i=1}^n = 1 \cdot 2 \cdots (n-1) \cdot n.$

Rekursive Definition:

$$0! := 1$$

← Induktionsanfang

$$n > 0 \Rightarrow n! := n \cdot (n-1)! \quad \leftarrow \text{Induktionsschritt}$$

```
long int factorial(long int n){  
    if(n > 0)  
        return(n*factorial(n-1));  
    else  
        return(1);  
} /* factorial */
```

Vor. 7
factorial(3)

```
    ↴ factorial(2)
        ↴ factorial(1)
            ↴ factorial(0)
                ↴ 1 ↴
                1=1*1 ↴
                2=2*1 ↴
            3*2 ↴
```

factorial(-2) 

```
long int factorial(long int n) {
    if (n<0) {
        printf("factorial(n), n<0");
        exit();
    } ...
```

Performance: schlecht!

Rekursion nur wenn Realisierung kaum anders möglich.

Sortieren - merge sort

```

sort(n, x[0..n-1])
    Wähle k ≈ n/2
    k>1 : sort(k, x[0..k-1])
    n-k>1: sort(n-k, x[k..n-1])
    i1 = 0; i2 = k;
    for i = 0..n-1
        if x[i1] ≤ x[i2]
            y[i] = x[i1++]
        else
            y[i] = x[i2++]

```

Bsp.

2	4	-1	14	8	3	5
			-1	2	4	
			3	5	8	14
-1	2	3	4	5	8	14

Parameter:

- Länge des Feldes (# Elemente)
- Anfangsindex

lokales Feld y (free nicht vergessen!)

Rechenzeit (in Abhangigkeit von n)

I) for i = 0..n-2

$$x_j := \text{Min}(x[i..n-1])$$

$$x_i := x_j$$

Im Schritt i: n-i Vergleiche

$$\Rightarrow Rz \sim \sum_{i=0}^{n-2} (n - i) = \sum_{i=2}^n i = \frac{n(n+1)}{2} - 1 \sim n^2$$

II) Mergesort

Ansatz: Sei die Rz $\sim M(n)$

$$\begin{aligned} M(n) &\sim 2 \cdot M\left(\frac{n}{2}\right) + n \sim 2 \cdot \left(2 \cdot M\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n = 4M\left(\frac{n}{4}\right) + 2n \\ &\sim 4 \cdot \left(2 \cdot M\left(\frac{n}{8}\right) + \frac{n}{4}\right) + 2n = 8 \cdot M\left(\frac{n}{8}\right) + 3n = 2^3 \cdot M\left(\frac{n}{2^3}\right) + 3n \end{aligned}$$

$$k = \log_2 n \Rightarrow M(n) \sim 2^k \cdot M\left(\frac{n}{2^k}\right) + k \cdot n$$

$$= n \cdot M(1) + n \cdot \log n$$

$$\sim \underline{\underline{n \cdot \log n}}$$

Rechenzeit Methode I), II)

100 Mflops

n	I) $[n^2]$	II) $[n \cdot \log_2 n]$
10	1 μ sec	0.3 μ sec
1000	1 msec	99 μ sec
10^5	1.7 min	16 msec
10^6	2.8 h	0.2 sec
10^7	11.6 d	2.3 sec

Datentyp char

character, belegt 1 Byte, rechnen wie int

Bsp.:

```
char ch = 'p';
printf("%c %c %d %d \n", ch, ch+1, ch, ch+1);
output: p q 112 113
```

Bsp.:

```
int IsUpperCase(char ch) {
    return ('A' <= ch) && (ch <= 'Z');
} /* IsUpperCase */
```

ASCII (Kodierung von charaktern,
 American Standard Code for Information Interchange)
 'a'..'z', 'A'..'Z', '0'..'9' aufeinanderfolgend

Bsp.:

```
char LowerCase(char ch) {
    if IsUpperCase(ch)
        return (ch - 'A' + 'a')
    else
        return (ch);
} /* LowerCase */
```

- Vorl.1
- Vorl.2
- Vorl.3
- Vorl.4
- Vorl.5
- Vorl.6
- Vorl.7
- 8 Vorl.8**
- Vorl.9
- Vorl.10
- Vorl.11
- Vorl.12

87

Themenübersicht Vorlesung 8

- char-Datentyp
- make
- einfacher Taschenrechner
- switch-Anweisung
- Strings
- Syntax:
`#include<stdlib.h>,
(...*)malloc(sizeof(...)),
exit, NULL`

char-Datentyp (Fortsetzung)

```

void PrintBits (double d) {
    char *ch, i;
    unsigned char k;
    ch = ((char *)(&d))+7;
    /* ch zeigt auf das 8-te (letzte) byte der IEEE754-Darstellung von d.
       | 11 bit Exponent   | 52 bit Mantisse |
       |V|e|e|e|e|e|e|e|M|M|M|M|.....|M|M|M|
       63           ch->56      52|51           0| */
    for ( i=0; i<8; i++, ch--) { /* alle 8 byte absteigend durchgehen.*/
        for ( k=128; k>0; k>>=1 ){
            /* k hat zu Beginn der Schleife das bit-Muster 0...010000000,
               d.h. im 8-ten bit eine 1 und sonst Nullen. Nach jedem
               Schleifendurchlauf wird die 1 durch die bit-Operation k>>=1
               um eine Position nach rechts geschoben.*/
            printf("%d", ( k & (*ch)) != 0);
            /* Das & ist der bit-weise logische UND-Operator, d.h. k und
               (*ch) werden bit-weise verglichen. Das Ergebnis ist genau
               dann ungleich 0, wenn (*ch) an der Position, an der die 1
               in k steht ebenfalls eine 1 hat, d.h. im j-ten Schleifen-
               durchlauf wird das j-te bit von oben ausgegeben.*/
        }
        printf(" ");
    }
} /* PrintBits */

```

functions.h

```
double my_tan(double x);  
double my_cos(double x);
```

functions.c

```
#include <stdio.h>  
  
#include "functions.h"  
  
double my_tan(double x) {  
    double t, sum;  
    int k;  
    ...  
    return sum;  
}  
double my_cos(double x) {  
    ...  
}
```

main.c

```
#include <stdio.h>  
#include "functions.h"  
  
int main(void) {  
    double x, y;  
    x = 1.0;  
    y = my_tan(x);  
    ...  
}
```

make

Das Programm "make" ist ein Werkzeug zur Verwaltung mehrerer, miteinander in Beziehung stehender Dateien.

Die Steuerdatei "makefile" muss selbst erstellt werden.

Makefile

```
CFLAGS = -Wall -ansi -pedantic
```

```
CC = gcc
```

```
main: main.o functions.o
```

```
    CC $(CFLAGS) -o main main.o functions.o
```

```
main.o: main.c functions.h
```

```
    CC $(CFLAGS) -c main.c
```

```
functions.o: functions.c
```

```
    CC $(CFLAGS) -c functions.c
```

Analyse einfacher Ausdrücke

lexikalsche Analyse: Zerlegung eines Programms in Schlüsselwörter, Symbole; Elimination bedeutungsloser Zeichen, Kommentare

syntaktische Analyse: syntaktische Korrektheit, Zerlegung in syntaktische Einheiten

semantische Analyse: semantische Bearbeitung

Ein einfacher Taschenrechner

Formeln: aus $+, -, *, /, (,)$, integer

Eingabe: eine Formel pro Zeile,

Programmende: erstes Symbol 'q'

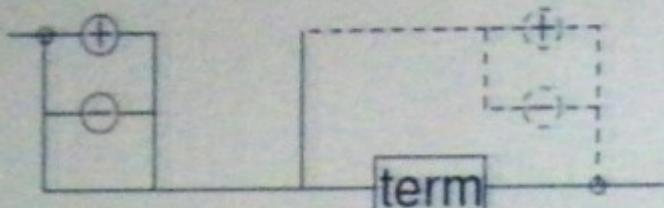
Ausgabe: jeweils der (numerische) Wert der Formel

Bps. $-4 * (13 + 1/2)$

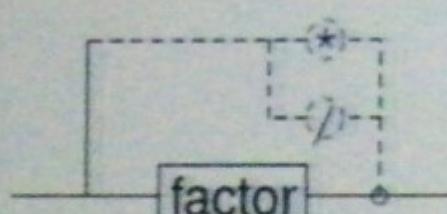
◦ – Schaltstellen

Bps. $-4 * (13 + 1/2)$

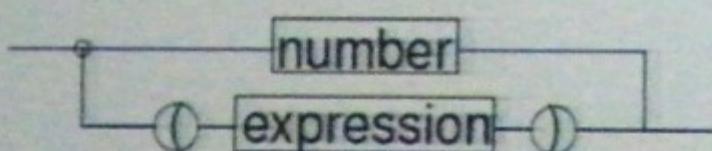
expression



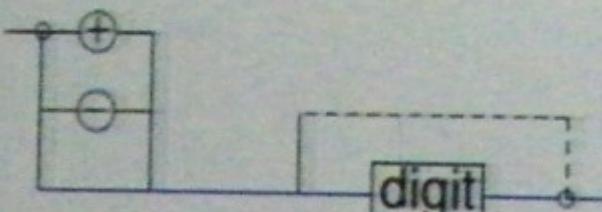
term



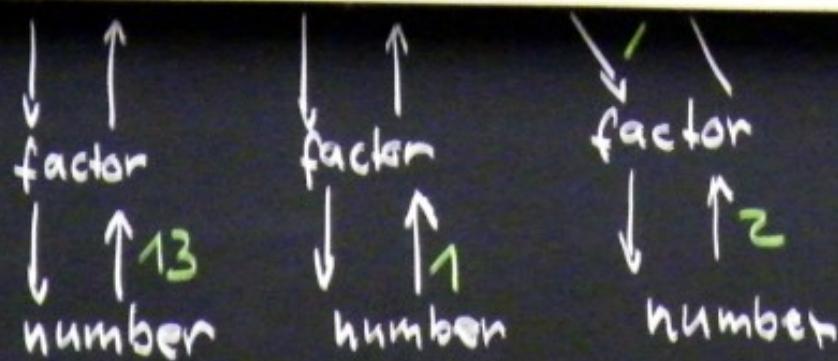
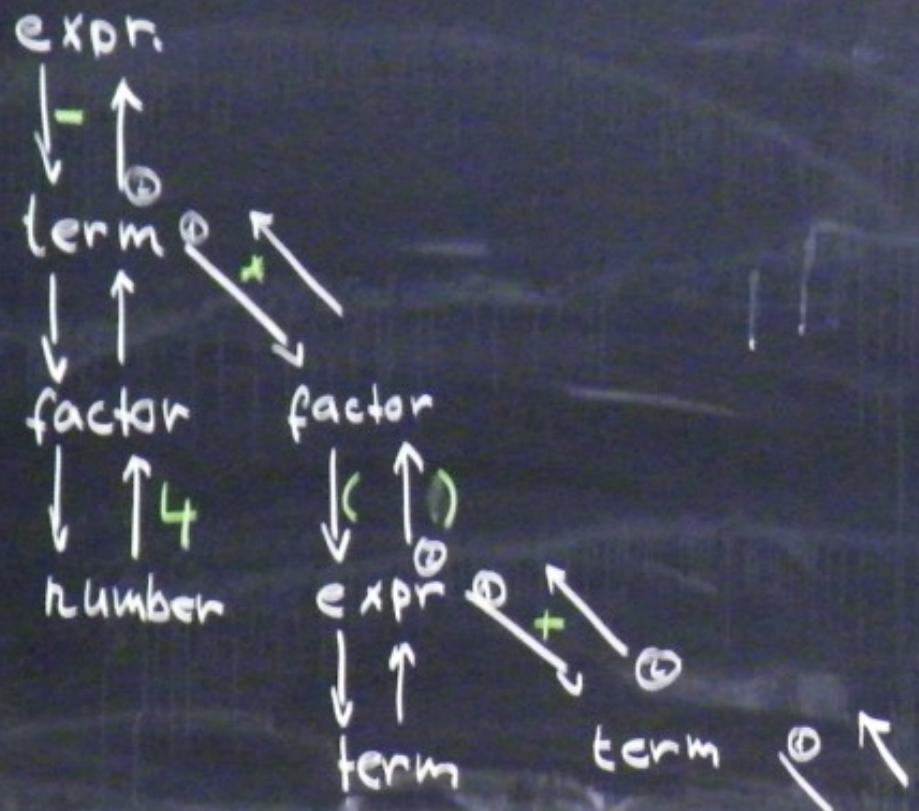
factor



number



Beispiel: $-4 * (13 + 1/2)$



switch-Anweisung

→ An allen "Schaltstellen" ist die Fortsetzung jeweils eindeutig durch den nächsten character bestimmt!

z.B. expression: + - sonst
 term: * / sonst
 factor: (sonst

```
switch (expression) {
    case constant expression: statement;
    case constant expression: statement;
    ...
    default: statement;
```

```
}
```

```
char ch;
switch (ch) {
    case '+': ...; break;
    case '-': ...; break;
    default: ...;
```

```
int n;
...
switch (n) {
    case 0: n++;
    case 1: n--;
}
```

$n = 1 \Rightarrow n = 0$
 sonst n unverändert

main (Taschenrechner)

- Lies nächstes character ch

- while ch != 'q'

 x = expression

 print x

 lies nächstes char ch

 ch = getchar(); <stdio.h>

 ch = getc(stdin); <stdio.h>

(→ "ruft auf")

main → expression → term → factor → expression → ...

⇒ Prototypen (header, *.h - Dateien)

d.h. nur die erste Zeile einer Funktion hinschreiben: dadurch wird der Funktionsname mit Parametern definiert, aber erst später der Funktionsrumpf angegeben.

Prototypen

Durch Prototypen wird Spezifikation und Implementierung getrennt
(rapid prototyping)

number "momentaner" char ch muß '+' , '-' oder digit sein!

⇒ input: "momentaner" char ch

output: Wert (der gelesenen "number")

neuer "momentaner" char ch

factor ch muß '+' , '-' , digit [für number] oder '(' sein

⇒ input: ch

output: berechneter Wert

ch

"momentaner" char ch ist immer input & output

⇒ Prototypen

```
double expression(char *ch);  
double term(char *ch);  
double factor(char *ch);  
double number(char *ch);
```

Bsp. factor

[ohne Kommentare]

```
double factor(char *ch) {  
    double d;  
  
    switch(*ch) {  
        case '(': *ch = getchar();  
                    d = expression(ch);  
                    *ch = getchar(); /*)  
                    break;  
        default: d = number(ch);  
    }  
    return (d);  
} /* factor */
```

*) geht davon aus, daß ')' folgt

→ Fehlerbehandlung!

Strings (Zeichenketten)

String = Array of char

```
char ch = 'p';           einzelnes Zeichen  
char str[] = "string";  Zeichenkette  
                        ↗ "... string Konstante"
```

char werden intern als **integer gespeichert** und
ggf. als **char interpretiert**

Bsp. int i = 115;
 printf("%d %c \n", i, i);
 ⇒ Ausgabe: 115 s

115 ist ASCII-code für s

Bsp. printf("%s", str);
 ⇒ Ausgabe: string

⇒ Spezielle Markierung des Endes eines strings durch die **Zahl 0**.

Themenübersicht Vorlesung 9

- Strings
- Dateiein-/ausgabe
- Kommandozeilenargumente
- strukturierte Datentypen
- Syntax: `sizeof(str), sprintf(), sscanf(),
file *fopen(char *filename, char *mode),
*mode:"r", "w", "a", stdin, stdout, tolower(),
fscanf(), fprintf(), fclose(),
struct...{...; ...};`

Strings

	<u>Speicher</u>	115	116	114	105	110	103	0
"string"		's'	't'	'r'	'i'	'n'	'g'	'\0'

⇒ sizeof(str) ist 7!

str[3] = 'o'; printf("%s", str); → "strong"

Bsp. #define BUFSIZE 100
char str[BUFSIZE];
str[0]='h';
str[1]='i';
str[2]='\0';
printf("%s \n", str); → "hi"

Bsp. void strcpy(char s1[], char s2[]){ /* kopiere s2 → s1 */

I) int i = 0;
while (s2[i]!='\0') {
 s1[i] = s2[i];
 i++;
};
s1[i] = '\0';

} /* strcpy */

II) int i = 0;
while (s2[i]) {
 s1[i] = s2[i];
 i++;
};
s1[i] = 0;

- Vorlesung
- III) `while (*s2){
 *s1 = *s2;
 s1++;
 s2++;
};
*s1 = 0;`
- IV) `while (*s2)
 *s1++ = *s2++;
*s1 = 0;`
- V) `while (*s1++ = *s2++);`

Bsp. `char *s;`

`s = (char *) malloc(3*sizeof(char));
s[0] = 'h'; s[1] = 'i'; s[2] = '\0';
printf("s = %s \n", s); → s = hi`

 Stringkonstante

`printf(format, arg1, arg2, ...)` Ausgabe → stdout
`sprintf(string, format, arg1, ...)` Ausgabe → string

Entsprechend `scanf, sscanf`

Bsp. `char str[] = "3.14e0";
double x;
sscanf(str, "%lf", &x);
printf("x = %10.3e \n", x); → _3.140e+00`

I/O von/in files

in stdio.h Datentyp FILE

interner (C-Programm) Filename ↔ externer (Betriebssystem) Filename

FILE *fopen(char *filename, char *mode);

-
- { "r" (nur) lesen
"w" (nur) schreiben
"a" anfügen (append)

Bsp. FILE *Daten;

Daten = fopen("Matrix.dat", "r");

Standardnamen: stdin input von Konsole
stdout output auf Konsole

printf(format, arg1, ...) ⇒ fprintf(stdout, format, arg1, ...)

scanf(format, arg1, ...) ⇒ fscanf(stdin, format, arg1, ...)

Beispiel: I/O von/in files

```
1 #include <stdio.h>
2 #include <ctype.h>
3 int main(void) {
4     FILE *in, *out;
5     char ch;
6     double x;
7     printf("I/O from/to file (y/n)? ");
8     ch=getchar();                                /* <=> fgetc(stdin); */
9     if( tolower(ch)=='y' ) {
10         in=fopen("Matrix.dat", "r");
11         out=fopen("Results.out", "w");
12     } else {
13         in=stdin;
14         out=stdout;
15     }/* if */
16     fscanf(in, "%lf", &x);
17     fprintf(out, "x=%f\n", x);
18     fclose(in);
19     fclose(out);                                /* auch, um Daten zu sichern! */
20     return (0);
21 }/* main */
```

Typische Anwendung: Filter

Eingabehilfe $\xrightarrow{\text{Programm}}$ Ausgabehilfe

Argumente aus der Kommandozeile:

int main(int argc, char *argv[])

argc : # Argumente, Trennung —

argv : array of strings argv[0]: Programmname

argv[i]: Argumente, $1 \leq i < \text{argc}$

Bsp.

```
int main (int argc, char *argv[]){
    int m,n;
    sscanf(argv[1], "%d", &m);
    sscanf(argv[2], "%d", &n);
    printf("m*n = %d \n", m*n);
    return 0;
}
```

Aufruf: mul -3 50 argc 3

 argv[0] "D:\gcc\mul.c"

Ausgabe: m*n = -150 argv[1] "-3"
 argv[2] "50"

strukturierte Datentypen

Array Daten gleichen Typs

```
int M[3];  
double A[NMAX][NMAX];
```

struct Daten unterschiedlichen Typs

Bsp.

```
struct sAdresse{  
    char *Strasse;  
    int Hausnummer;  
};  
struct sPerson{  
    char Name[20];  
    char Geschlecht;  
    int Alter;  
};  
struct sPerson P1 = {"Meyer",'m',23};
```

Typvereinbarungen

Variablen [P1] Vereinbarung und Initialisierung

- Vorl.1
- Vorl.2
- Vorl.3
- Vorl.4
- Vorl.5
- Vorl.6
- Vorl.7
- Vorl.8
- Vorl.9
- 10 Vorl.10
- Vorl.11
- Vorl.12

Themenübersicht Vorlesung 10

- strukturierte Datentypen
- Lineare Listen
- Hash-Tabellen
- Syntax: `struct...{...; ... ;};`
`(*mypointer).mycomponent,`
`mypointer->mycomponent,`

strukturierte Datentypen

Array Daten gleichen Typs

```
int M[3];
double A[NMAX] [NMAX];
```

struct Daten unterschiedlichen Typs

Bsp. struct sAdresse{

```
    char *Strasse;
    int Hausnummer;
};
```

Typvereinbarungen

```
struct sPerson{
    char Name[20];
    char Geschlecht;
    int Alter;
};
```



```
struct sPerson P1 = {"Meyer",'m',23};
```

Variablen [P1] Vereinbarung und Initialisierung



Komponentenzugriff

```
P1.Geschlecht = 'w';
printf("Alter = %d \n", P1.Alter);
```

Zeiger auf Strukturen

```
struct sPerson *Author = {"Lodge", 'm', 65};
printf("sex = %c", (*Author).Geschlecht);
```

einfacher: Author->Geschlecht

z.B. Author->Name = "Djerassi";

Bsp.

```
struct sPerson *PersonAlloc(void) {
    struct sPerson *P;
    P = (struct sPerson *)malloc(sizeof(struct sPerson));
    if (P==NULL) ....
    return(P);
} /* PersonAlloc */
```

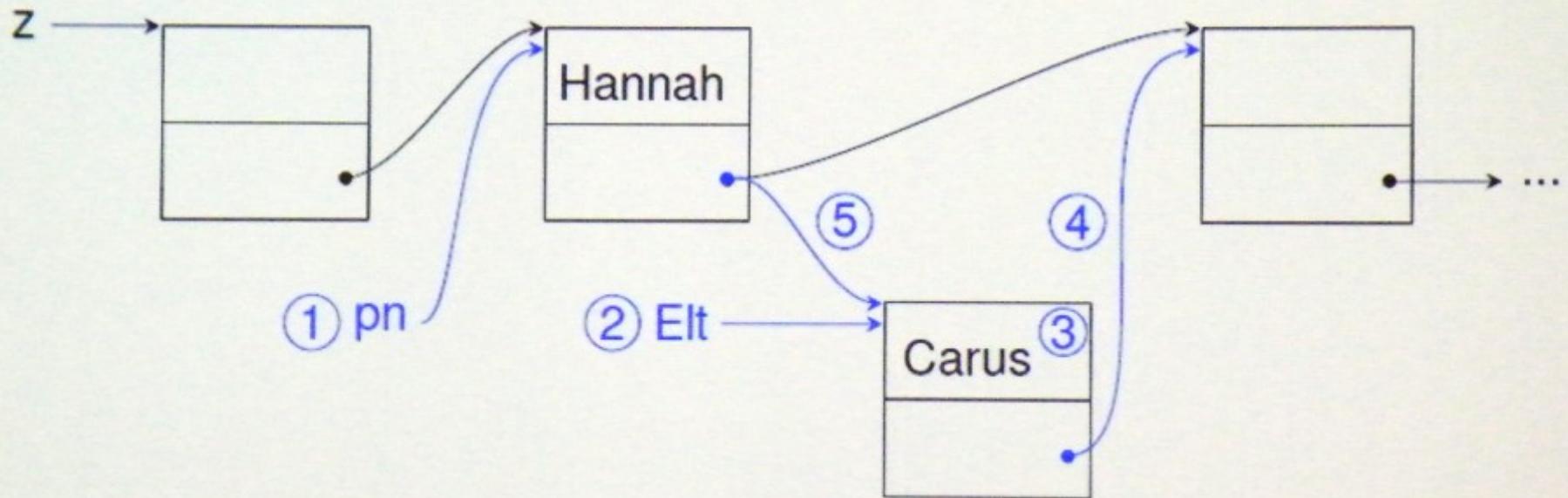


Lineare Listen

```
struct sNameList{
    char Name[20]
    struct sNamelist *succ; /* Zeiger auf Nachfolger */
}
struct sNameList *pn;
pn->Name = "Lodge";
pn->succ = NULL;
```

Bsp. Listenelement "Carus" nach "Hannah" einfügen.

```
struct sNameList *z, *pn, *Elt;
① pn = z->succ;
② Elt = (struct sNameList*)malloc(sizeof(*Elt));
if(Elt==NULL) ...
③ strcpy(Elt->Name, "Carus");
④ Elt->succ = pn->succ;
⑤ pn->succ = Elt;
```



```

void PersonInit(struct sPerson *P, char *Name,
                char sex, int age){

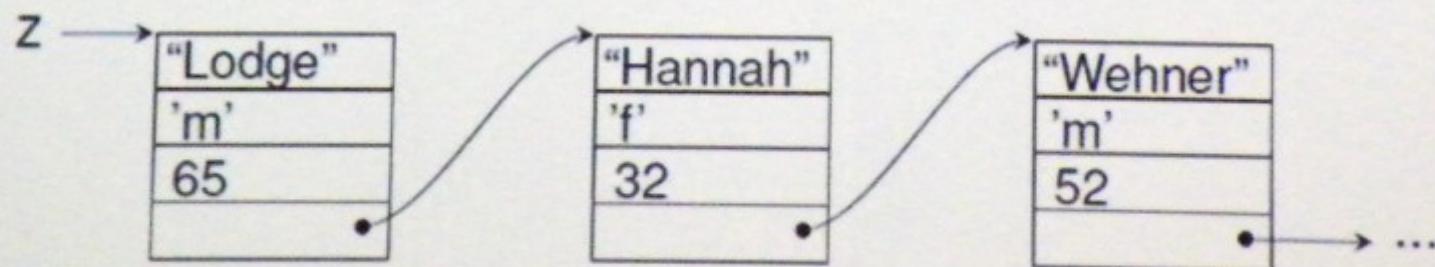
    strcpy(P->Name, Name);           /* string */
    P->Geschlecht = sex;            /* char */
    P->Alter = age;                /* int */

} /* PersonInit */

int main(void){
    struct sPerson *custodian;
    custodian = PersonAlloc();
    PersonInit(custodian, "Wehner", 'm', 52);
}

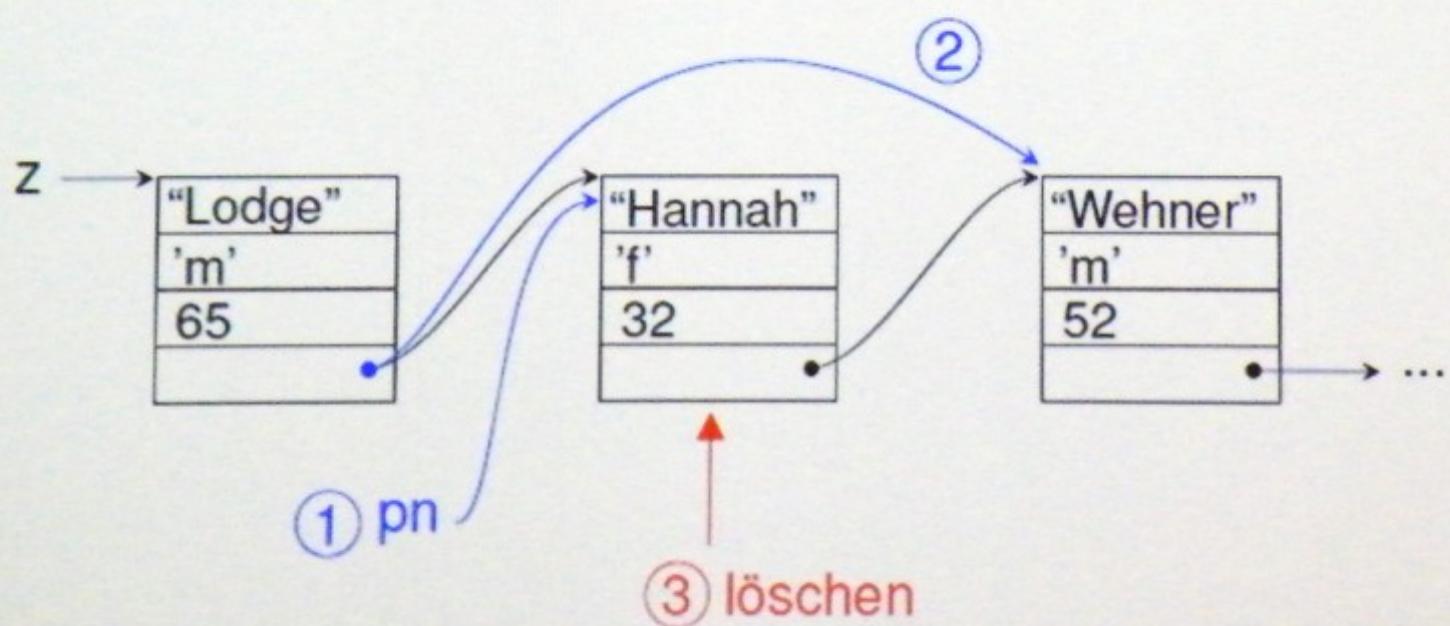
```

Lineare Listen



Zweites Element "Hannah" löschen

- ➊ `pn = z->succ;`
- ➋ `z->succ = pn->succ;`
- ➌ `free (pn);`



Hash-Tabelle

geg.: Menge von Namen

hash: {string} → $H \subseteq \mathbb{N}$

z.B. $H = \{0, 1, 2, \dots, 255\}$

hashtab: array von Zeigern auf NameList

hashtab[0] → □ → □ → □ → ...

hashtab[1] → □ → □ → ...

:

hashtab[255] → □ → □ → □ → □ → ...

struct sNamelist *hashtab[256] , 256 ≜ HASHSIZE

Bsp.

```
int hash(char *str){
    /* returns hashvalue for string str */
    int hashval;
    for(hashval=0; *str; hashval+=*str++);
    return(hashval % HASHSIZE);
} /* hash */
```

Bsp.

```
struct sNameList *Lookup(char *str){
    /* checks whether string str in hashtab */
    struct sNameList *pn;
    for(pn=hashtab[hash(str)]; pn!=NULL; pn=pn->succ) {
        if(strcmp(str, pn->Name)==0) {
            return(pn);    /* pn adress of str */
        }
    }
    return(NULL);          /* str not found */
} /* Lookup */
```

Themenübersicht Vorlesung 11

- Datentyp Matrix
- zweidimensionale Felder
- Syntax:

```
typedef  
    #ifdef...#else...#endif  
    #undef
```

Datentyp Matrix 1

```
typedef double *Matrix;
[           int      *z;           ]
```

Bsp.

```
Matrix MatAlloc(int m, int n){
    Matrix A;
    A = (Matrix)malloc(m*n*sizeof(double));
    if(A==NULL) ...
    return(A);
}
```

Zugriff: $A_{i,j} \triangleq A[i \cdot n + j]$

00		0(n - 1)
10	A	
m	(m - 1)0	(m - 1)(n - 1)

```
A = MatAlloc(5, 8);
...
free(A); /* nicht vergessen */
```

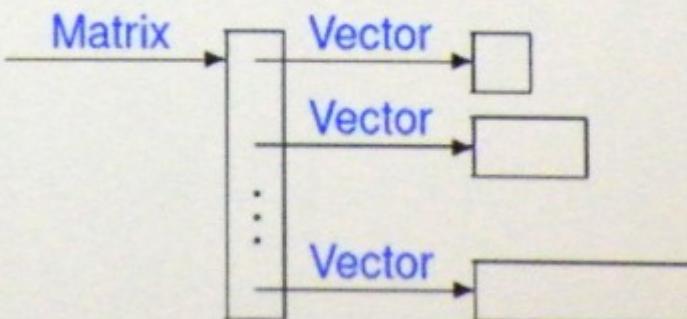
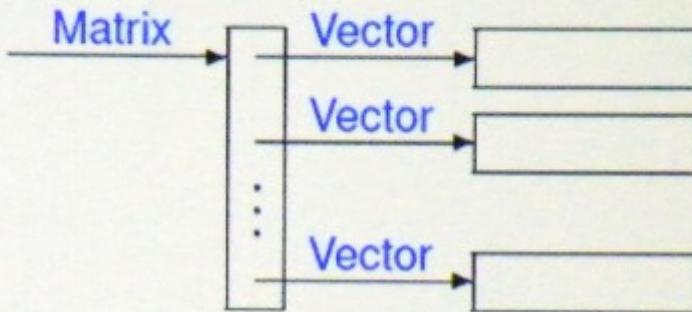
Datentyp Matrix 2

```
typedef double *Vector;
typedef Vector *Matrix;
```

```
1 Matrix MatAlloc(int m, int n){
2     int i;
3     Matrix B;
4     B = (Matrix) malloc(m*sizeof(Vector));
5     if(B==NULL){printf("error: not enough memory");return NULL;}
6     for (i = 0; i < m; i++) {
7         B[i] = (Vector) malloc(n*sizeof(double));
8         if(B[i]==NULL){printf("error: not enough memory");return NULL;}
9     }
10    return B;
11 }
```

MatFree : **erst** free(B[i]) für $0 \leq i \leq m - 1$
dann free(B)

auch △ - Matrizen :



Felder	Indizes ab 0
Vektoren, Matrizen	Indizes ab 1

Präprozessor Source $\xrightarrow{\text{Präprozessor}}$ mod. Source

Bsp. `#define ABS(x) ((x) < 0) ? (- (x)) : (x)`
 auch `a = (a < 3 ? 3 : a); max(a,3)`

`#define MAX(a,b) (((a) < (b)) ? (b) : (a))`

Bsp. `#ifdef LARGE`
 `#define DIM 1000`
 `#else`
 `#define DIM 10`
 `#endif`

Bsp. `#ifdef TRYMAIN1`
 `int main(void) { }`
 `... }`
 `#endif`

} der Code wird nur kompiliert, wenn vorher `#define TRYMAIN1`

Indexzugriff

```
#define N 10
```

...

```
double A[N][N]
```

 $A[0][0] \leftrightarrow A_{11}$

1. Lsg.: #define $A(i, j)$ $A[(i)-1][(j)-1]$
 $\Rightarrow A(1, 1) \rightarrow A[0][0]$

2. Lsg.:

00	01	02	03
10	11	12	13
20	21	22	23

2-dim Feld

11	12	13	14
21	22	23	24
31	32	33	34

Matrix

0	1	2	3
4	5	6	7
8	9	10	11

$n=4$
 $m=3$

zeilenweise
Speicherung

```
#define M 3
#define N 4
#define A(i, j) A[((i)-1)*N+(j)-1]

...
double A[M*N];
... A(i, j)
#undef A      /* nicht vergessen !! */
```