

Vorlesung

Einführung in die Programmierung für Ingenieure

Sommersemester 1998

(10299)

Vorlesung

Einführung in die Programmierung für Ingenieure

Sommersemester 1998

(10299)

Einführung in die Programmierung für Ingenieure (C)

Inhaltsverzeichnis

A	Organisatorisches	A.1
A.1	Vorlesung	A.1
A.2	Vorlesung (2)	A.2
A.3	Übungen	A.4
B	Übersicht	B.1
B.1	Grundlagen	B.1
B.2	erste Schritte zum Programmieren	B.2
B.3	Ablaufkontrolle	B.2
B.4	Funktionen und Programmstruktur	B.3
B.5	Felder und Zeiger	B.3
B.6	Strukturen etc.	B.4

C	Grundlagen	C.5
C.1	Programmierung	C.6
C.2	Programm	C.7
C.3	Beispiel (1) — Polynomberechnung	C.8
C.4	Beispiel (1) — Programm.	C.9
C.5	Beispiel (2) — Nullstellen einer Funktion	C.10
C.6	Beispiel (2) — Struktogramm.	C.11
C.7	Beispiel (2) — Programm.	C.12
C.8	Programmerstellung und -ausführung	C.13
	1 Vorgehensweise	C.13
	2 einfaches Beispiel	C.14
	3 Übersetzung und Ausführung der Beispielprogramme.	C.15
C.9	Klassischer Universalrechner	C.16
C.10	Datendarstellung im Rechner	C.17
C.11	Dateien und Dateisystem.	C.18
	1 Allgemeines.	C.18
	2 Struktur des Dateisystems.	C.19
	3 Pfade im Dateisystem	C.20
C.12	Benutzerumgebung	C.21
C.13	Programmierumgebung	C.22
C.14	Begriffe.	C.24

D	Datentypen, einfache Programmstruktur, Operatoren, Ausdrücke	D.1
D.1	Datentypen	D.2
D.2	Standardtypen in C	D.3
D.3	Variablen	D.5
D.4	Ganze Zahlen	D.8
D.5	Fließkommazahlen	D.10
D.6	Zeichen	D.13
D.7	Zeichenketten	D.15
D.8	Aufbau eines C-Programms	D.16
D.9	Ausdrücke	D.18
D.10	Operatoren	D.19
	1 Zuweisungsoperator =	D.19
	2 Arithmetische Operatoren	D.20
	3 spezielle Zuweisungsoperatoren	D.21
	4 Vergleichsoperatoren	D.22
	5 Logische Operatoren	D.23
	6 Bitweise logische Operatoren	D.25
	7 Logische Shiftoperatoren	D.26
	8 Bedingte Bewertung	D.28
	9 Komma-Operator	D.29
D.11	Typumwandlung in Ausdrücken	D.30
D.12	Vorrangregeln bei Operatoren	D.31

E	Ablaufkontrolle	E.1
E.1	Struktur eines C-Hauptprogramms	E.2
E.2	Anweisungen	E.2
E.3	Blöcke	E.3
E.4	Kontrollstrukturen	E.4
	1 Bedingte Anweisung	E.5
	2 Fallunterscheidung	E.10
	3 Schleifen	E.12
	4 abweisende Schleife	E.12
	5 nicht-abweisende Schleife	E.14
	6 Schleifen — Bsp. Sinusberechnung	E.15
	7 Laufanweisung	E.17
	8 Schleifensteuerung	E.20
F	Funktionen und Programmstruktur	F.1
F.1	Funktionen	F.2
F.2	Funktionsdefinition	F.5
F.3	Funktionsaufruf	F.8
F.4	Funktionen — Beispiel	F.10
F.5	Funktionen — Regeln	F.11
F.6	Funktionsdeklarationen — Beispiel	F.14
F.7	Parameterübergabe an Funktionen	F.15
	1 call by value	F.15
	2 call by reference	F.16

F.8	C-Preprozessor	F.17
	1 Kurzüberblick	F.17
	2 Makrodefinitionen	F.18
F.9	Programmstruktur & Module	F.20
	1 Softwaredesign	F.20
	2 Top-down Entwurf	F.21
	3 Module in C	F.24
F.10	Gültigkeit von Namen.	F.26
	1 Gültigkeit im gesamten Programm	
	Globale Variablen F.27	
	2 Gültigkeit nur im Modul	F.30
	3 Lokale Variablen	F.31
	4 Übersicht	F.32
F.11	Lebensdauer von Variablen.	F.33
	1 auto-Variablen	F.34
	2 static-Variablen	F.35
F.12	Wertaustausch zwischen Funktionen	F.36
F.13	Getrennte Übersetzung von Programmteilen — Beispiel	F.37

G	Felder und Strukturen	G.1
G.1	Eindimensionale Felder	G.1
G.2	Initialisierung eines Feldes	G.2
G.3	Zugriffe auf Feldelemente	G.4
G.4	Mehrdimensionale Felder	G.5
	1 Zugriffe auf Feldelemente	G.6
	2 Initialisierung eines mehrdimensionalen Feldes	G.7
G.5	Eindimensionale Felder als Funktionsparameter	G.8
	1 Beispiele	G.9
G.6	Strukturen	G.11
	1 Motivation	G.11
	2 Deklaration eines Strukturtyps	G.12
	3 Definition einer Struktur	G.14
	4 Zugriff auf Strukturkomponenten	G.15
	5 Initialisieren von Strukturen	G.16
	6 Strukturen als Funktionsparameter	G.17
	7 Felder von Strukturen	G.18
	8 Strukturen in Strukturen	G.19
	9 Compilerabhängige Eigenschaften	G.20

H	Zeiger(-Variablen)	H.1
H.1	Einordnung	H.1
H.2	Überblick	H.2
	1 Definition von Zeigervariablen	H.3
	2 Beispiele	H.3
H.3	Adreßoperatoren	H.4
	1 Adreßoperator &	H.4
	2 Verweisoperator *	H.4
	3 Der Operator []	H.4
H.4	Zeiger als Funktionsargumente	H.5
H.5	Zeiger und Felder (1)	H.7
H.6	Arithmetik mit Adressen	H.8
	1 Vorrangregeln bei Operatoren	H.9
	2 Beispiele	H.9
	3 Zeigerarithmetik und Felder	H.11
	4 Vergleichsoperatoren und Adressen	H.14
H.7	Felder als Funktionsparameter	H.15
H.8	Zeiger, Felder und Zeichenketten	H.16
H.9	Beispiel: dynamische Speicherverwaltung	H.22
H.10	Felder von Zeigern	H.25
H.11	Argumente aus der Kommandozeile	H.27
	1 Datenaufbau	H.28
	2 Zugriff Beispiel: Ausgeben aller Argumente (1)	H.29
H.12	Zeiger auf Zeiger	H.31
H.13	sizeof-Operator	H.32
H.14	Explizite Typumwandlung — Cast-Operator	H.33

H.15	Zeiger auf Strukturen	H.34
H.16	Rekursive Strukturen	H.36
H.17	Zusammenfassung	H.40

I Ein-/Ausgabe I.1

I.1	Überblick	I.1
I.2	Standard Ein-/Ausgabe	I.2
I.3	Öffnen und Schließen von Dateien	I.4
I.4	Zeichenweise Lesen und Schreiben	I.6
I.5	Zeilenweise Lesen und Schreiben	I.9
I.6	Kopieren beliebiger Blöcke	I.11
I.7	Formatierte Ausgabe	I.12
	1 Bibliotheksfunktionen — Prototypen (Schnittstelle)	I.12
	2 Formatangaben	I.13
I.8	Formatierte Eingabe	I.14
	1 Bibliotheksfunktionen — Prototypen (Schnittstelle)	I.14
	2 Bearbeitung der Eingabe-Daten	I.15
	3 Format-Anweisungen	I.16

J Verschiedenes J.1

J.1	Hashing	J.1
-----	-------------------	-----

A Organisatorisches

- Dozent
 - ◆ Dr.-Ing. Jürgen Kleinöder, Informatik 4 (Lehrstuhl für Betriebssysteme)
 - ◆ kleinoeder@informatik.uni-erlangen.de

A.1 Vorlesung

- Einführung in die Programmierung für Ingenieure — C
 - ◆ Grundlagen der Programmiersprachen,
Darstellung von Daten im Programm,
Kontrolle des Programmablaufs, Zeiger, Strukturen,
Ein-/Ausgabe, ...
 - ◆ für Studierende aller ingenieurwiss. Fachrichtungen
- Termin: [Mi. von 12.15 bis 13.45 im H9](#) (2 SWS)

A.2 Vorlesung (2)

■ Skript

◆ kein Skript im eigentlichen Sinne

◆ Folien

- werden im WWW zur Verfügung gestellt und können selbst ausgedruckt werden
- wir verteilen Kopien vor der Vorlesung
(Gutschein für komplettes Skript: 3 DM)

◆ weitergehende Informationen zum Nachlesen findet man am besten in der Literatur

■ URL zur Vorlesung

◆ http://www4.informatik.uni-erlangen.de/Lehre/SS98/V_C/

◆ hier findet man Termine, Folien zum Ausdrucken und evt. Zusatzinformationen

A.2 Vorlesung (3)

■ Literatur

- ◆ B. W. Kernighan, D. M. Ritchie
Programmieren in C, 2. Ausgabe
Carl Hanser, München, Wien; Prentice Hall,
London, 1990
- ◆ Martin Lowes, Augustin Paulik
Programmieren mit C, 3. Auflage
Teubner, Stuttgart, 1995

■ Schein

- ◆ Klausur Ende Juli
- ◆ Voraussetzung für die Klausurteilnahme:
Zugangsvoraussetzung für die Klausur ist die Bearbeitung und
Abgabe der Übungsaufgaben!

A.3 Übungen

- Verantwortlich für die Übung
 - ◆ Dipl.-Inf. Jörg Barner,
 - ◆ Dipl.-Inf. Bernd Schnitzer
 - ◆ Dipl.-Inf. Martin Steckermeier

- Übungsinhalt:
 - ◆ Programmieren einfacher Übungsaufgaben zum Einüben des Vorlesungsstoffs
 - ◆ Tafelübungen
 - Vorbereitung und Besprechung der Übungsaufgaben
 - Diskussion von Problemen
 - ◆ Rechnerübungen
 - selbständige Bearbeitung der Programmieraufgaben an den CIP-Workstations der Informatik
 - Übungsleiter steht für Fragen zur Verfügung

A.3 Übungen (2)

■ mögliche Termine:

◆ Tafelübungen:

Mo: 10:00 - 11:00, 11:00 - 12:00, 12:00 - 13:00, 13:00 - 14:00

Di: 08:00 - 09:00, 09:00 - 10:00, 10:00 - 11:00, 11:00 - 12:00

Di: 16:00 - 17:00

◆ Rechnerübungen:

Mo: 10:00 - 11:00, 11:00 - 12:00

Di: 08:00 - 09:00, 09:00 - 10:00, 10:00 - 11:00, 11:00 - 12:00

■ Anmeldung zur Übung

◆ „login: can“ an allen CIP-Workstations der Informatik

Raum 01.155 im blauen Hochhaus

Mittwoch 6. Mai 15:00 Uhr bis Freitag, 8. Mai 18:00 Uhr

B Übersicht

B.1 Grundlagen

- Programmieren & Programme
- Grundkonzepte eines Rechners
- Datendarstellung im Rechner
- Dateien
- Programmierumgebung

B.2 erste Schritte zum Programmieren

- Datentypen
- Variablen
- Zahlen & Zeichen
- Aufbau eines C-Programms
- Ausdrücke & Operatoren

B.3 Ablaufkontrolle

- Anweisungen & Blöcke
- Kontrollstrukturen (if-Anweisung, Schleifen)

B.4 Funktionen und Programmstruktur

- Funktionen
- Parameterübergabe
- Globale und lokale Variablen
- Preprozessor

B.5 Felder und Zeiger

- Speicherorganisation in einem Programm
- Felder
- Zeiger

B.6 Strukturen etc.

- Strukturen
- Komplexe Datenstrukturen (Listen, Bäume, ...)
- Suchverfahren
- ...

C Grundlagen

- Programmierung
- Programmerstellung und -ausführung
- Organisation eines Rechners
 - ◆ Klassischer Universalrechner
 - ◆ Datendarstellung im Rechner
 - ◆ Dateien
 - ◆ Benutzerumgebung
 - ◆ Programmierumgebung
- Begriffe

C.1 Programmierung

- Einsatz von Rechenanlagen
 - ◆ Informationen speichern, strukturieren, sortieren, zugänglich machen
 - ◆ Rechnen an sich ist eher selten

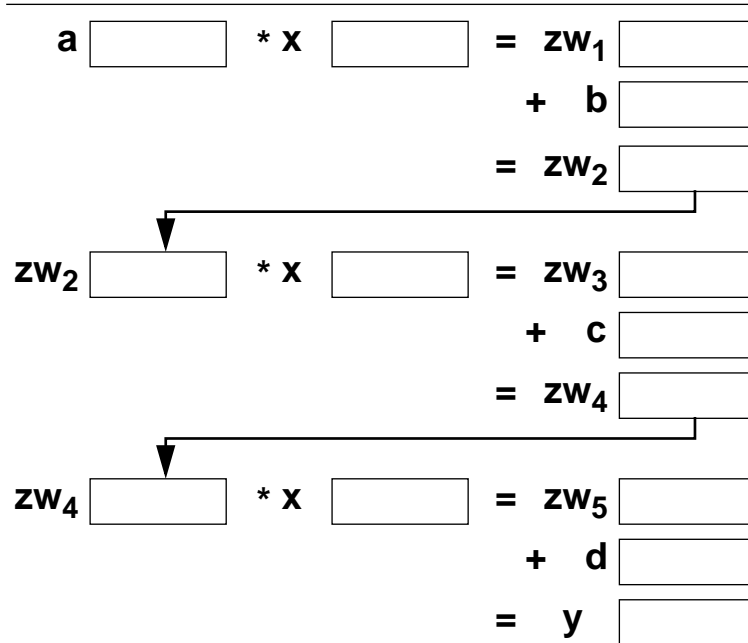
- Problemlösung mit Rechenanlagen
 - ◆ Programmierung

C.2 Programm

- Daten: Abstrakte Darstellung des problemrelevanten Teils der realen Welt
- Algorithmus: Schrittfolge von Operationen auf den Daten, die als allgemeines Verfahren die Lösung einer Klasse von Problemen zum Ziel hat
 - endliche Beschreibung
 - eindeutige Operationsfolge
es steht stets fest, welches der nächste Schritt ist
 - Problemklasse: z. B. größter gemeinsamer Teil von zwei beliebigen (!) natürlichen Zahlen

C.3 Beispiel (1) — Polynomberechnung

- Problemklasse: Berechnung von $f(x) = ax^3 + bx^2 + cx + d$
- Daten: Parameter $a, b, c, d, zw_{1,2,3,4} \in \mathbb{R}$
- Algorithmus: Horner Schema



Schritte:

1. $a * x \rightarrow zw_1$
2. $zw_1 + b \rightarrow zw_2$
3. $zw_2 * x \rightarrow zw_3$
4. $zw_3 + c \rightarrow zw_4$
5. $zw_4 * x \rightarrow zw_5$
6. $zw_5 + d \rightarrow y$

C.4 Beispiel (1) — Programm

```
main()
{
    /* Beschreibung der Daten */
    float x;                /* Argument zu dem Pol. berechnet wird */
    float y;                /* Polynomwert */

    float a = 11, b = 7, c = 5, d = 3;
    float zw1, zw2, zw3, zw4, zw5;    /* Zwischenergebnisse */

    /* Einlesen des Arguments */
    printf("Argument = ");
    scanf("%f", &x);

    /* Kontrollausgabe der Eingabedaten */
    printf("Der Polynomwert soll fuer das");
    printf(" Argument %f berechnet werden.\n", x);

    /* Berechnung des Polynomwertes mit Hornerschema */
    zw1 = a * x;
    zw2 = zw1 + b;
    zw3 = zw2 * x;
    zw4 = zw3 + c;
    zw5 = zw4 * x;
    y = zw5 + d;

    /* Ausgabe des Ergebnisses */
    printf("Ergebnis: f(%f) = %f\n", x, y);
}
```

C.5 Beispiel (2) — Nullstellen einer Funktion

■ Problemklasse: Was sind die reellen Nullstellen der Funktion $f: \mathbb{R} \rightarrow \mathbb{R}$ mit $f(x) = ax^2 + bx + c$

■ Daten: Parameter $a, b, c \in \mathbb{R}$, Diskriminante D

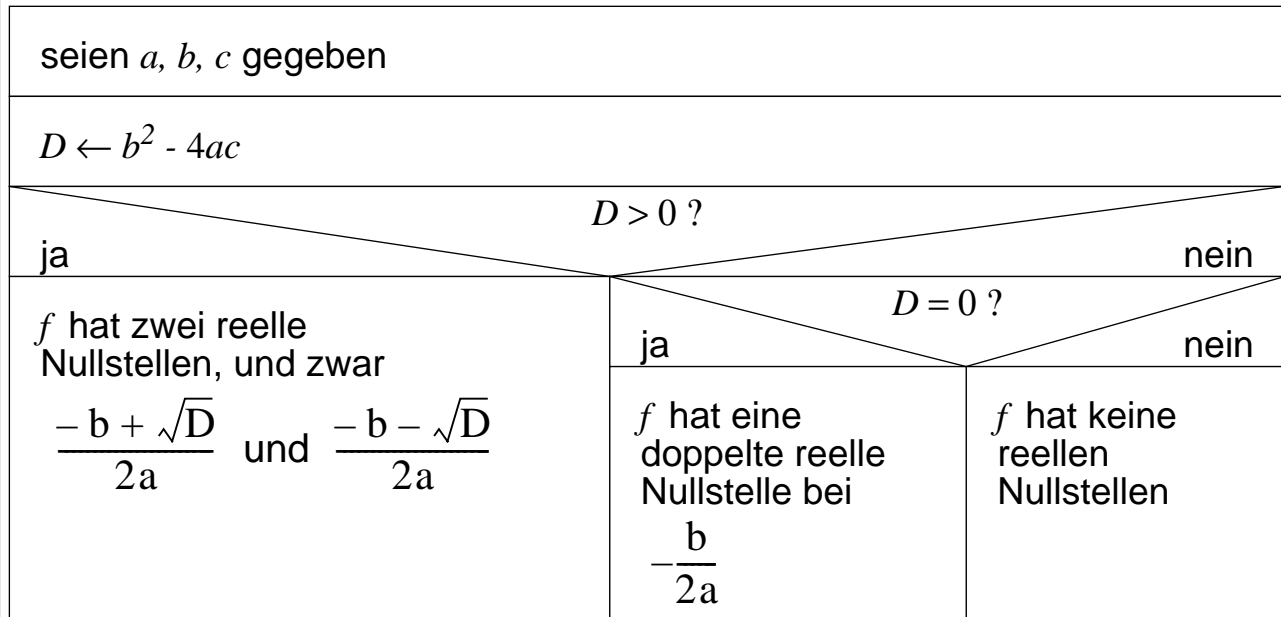
■ Algorithmus:

1. seien a, b, c gegeben
2. berechne die Diskriminante D zu $b^2 - 4ac$
3. wenn $D > 0$ weiter mit Schritt 7
4. wenn $D = 0$ weiter mit Schritt 9
5. f hat keine reellen Nullstellen
6. fertig
7. f hat zwei reelle Nullstellen, und zwar $\frac{-b + \sqrt{D}}{2a}$ und $\frac{-b - \sqrt{D}}{2a}$
8. fertig
9. f hat eine doppelte reelle Nullstelle bei $-\frac{b}{2a}$
10. fertig

C.6 Beispiel (2) — Struktogramm

- einfache Dokumentation des Programmentwurfs

Nullstellen — bestimmt reelle Nullstellen von $f(x) = ax^2 + bx + c$



C.7 Beispiel (2) — Programm

```
#include <stdio.h>
#include <math.h>

main ()
{
    float a, b, c;           /* Parameter in a*x^2+b*x+c */
    float dis;              /* Diskriminante */

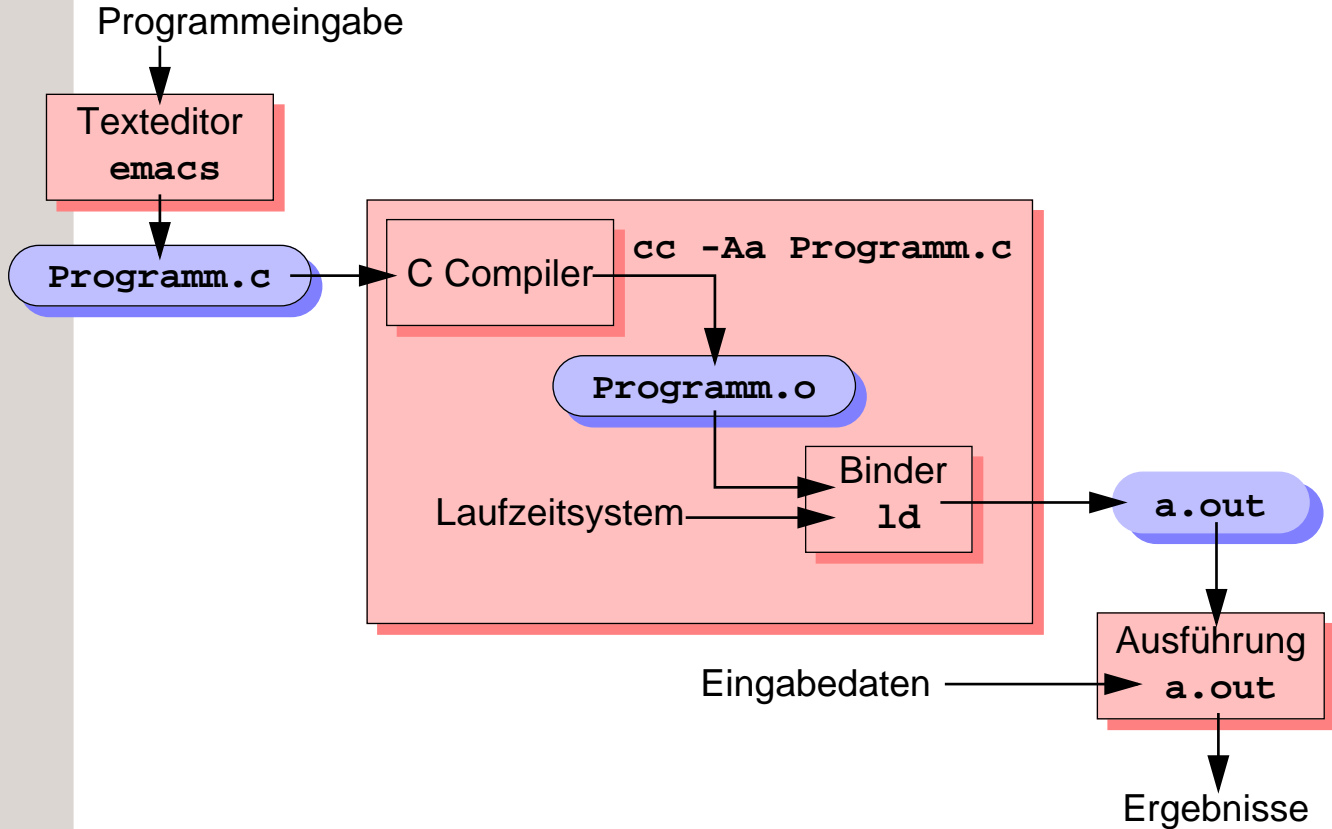
    /* Initialisierung: Einlesen der Parameter */
    printf("Nullstellen von f(x)=a*x^2+b*x+c\n");
    printf("a=");
    scanf("%f", &a);
    printf("b=");
    scanf("%f", &b);
    printf("c=");
    scanf("%f", &c);

    dis = b*b - 4.0 * a * c; /* Berechnung der Diskriminante */

    /* Berechnung der Nullstellen */
    if ( dis == 0.0 )        /* doppelte Nullstelle */
        printf("Doppelte Nullstelle bei %f\n", -b/(2.0*a));
    else if ( dis > 0.0 ) { /* zwei reelle Nullstellen */
        printf("1. Nullstelle bei %f\n", (sqrt(dis)-b)/(2.0*a));
        printf("2. Nullstelle bei %f\n", (-sqrt(dis)-b)/(2.0*a));
    }
    else                    /* keine reelle Nullstelle */
        printf("Keine reelle Nullstelle\n");
}
```

C.8 Programmierstellung und -ausführung

1 Vorgehensweise



■ Programm

```
main()
{
    printf("hello world\n");
}
```

■ Übersetzung und Ausführung

```
faii06a% cc hello.c
faii06a% a.out
hello world
faii06a%
```

3 Übersetzung und Ausführung der Beispielprogramme

■ Ausführung des Beispiels 'Polynom'

```
faii06a% cc -Aa polynom.c
```

```
faii06a% a.out
```

```
Argument = 4
```

Der Polynomwert soll fuer das Argument 4.000000 berechnet werden.

```
Ergebnis: f(4.000000) = 839.000000
```

```
faii06a%
```

■ Ausführung des Beispiels 'Nullstellen'

```
faii06a% cc -Aa Nullstellen.c -lm
```

```
faii06a% a.out
```

```
Nullstellen von  $f(x)=a*x^2+b*x+c$ 
```

```
a=1
```

```
b=2
```

```
c=-3
```

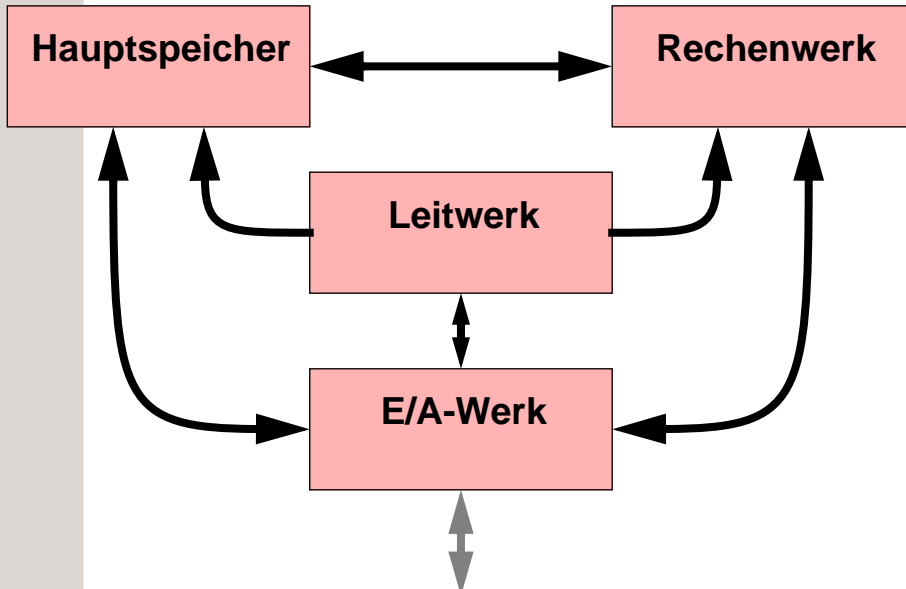
```
1. Nullstelle bei 1.000000
```

```
2. Nullstelle bei -3.000000
```

```
faii06a%
```

C.9 Klassischer Universalrechner

- *Burks, Goldstine und von Neumann (Princeton 1946/47)*



Rechenwerk:
Rechenoperationen und
logische Verknüpfungen

Hauptspeicher:
Programm und Daten

Leitwerk:
steuert Programmablauf

E/A-Werk:
Ein-/Ausgabe von Daten
und Programmen



C.10 Datendarstellung im Rechner

- Alle Informationen (Daten, Texte, Programmanweisungen) werden im Speicher durch Bitfolgen dargestellt
 - ◆ **Bit** = Speicherstelle mit Wert 0 oder 1
 - ◆ Beispiele: Zahl 2 = 10, Zahl 5 = 101, Buchstabe a = Zahl 97 = 1100 0001
- 8 Bit bilden ein **Byte**
- Für ganze Zahlen werden meist 32 Bit vorgesehen (ein **Wort**)
- Zeichen (Buchstaben) werden durch Zahlen kodiert
- Ein Wort ist normalerweise die Einheit, mit der das Rechenwerk operiert
- Das Rechenwerk führt entsprechend den Anweisungen des Programms arithmetische oder logische Verknüpfungen von Worten durch
- Die Worte auf denen das Rechenwerk operiert, werden in **Registern** des Rechenwerks zwischengespeichert (für schnelleren Zugriff)

1 Allgemeines

- Dateien sind abstrakte Gebilde zur Speicherung von Daten auf einem Hintergrundspeicher (Festplatte, Diskette)
 - ◆ Verbergen die reale Struktur (Sektoren, Zylinder, etc.)
 - ◆ Haben einen Namen (Zeichenkette)
 - ◆ UNIX verwaltet Zugriffsrechte
(welcher Benutzer darf lesen, schreiben
oder ein Programm in der Datei ausführen)

■ Hierarchische Strukturierung des Dateisystems

◆ reguläre Dateien

- Können beliebige Daten speichern (Texte, Zahlen, Programme, ...)
- Einfache Bedienung (öffnen, lesen, schreiben)

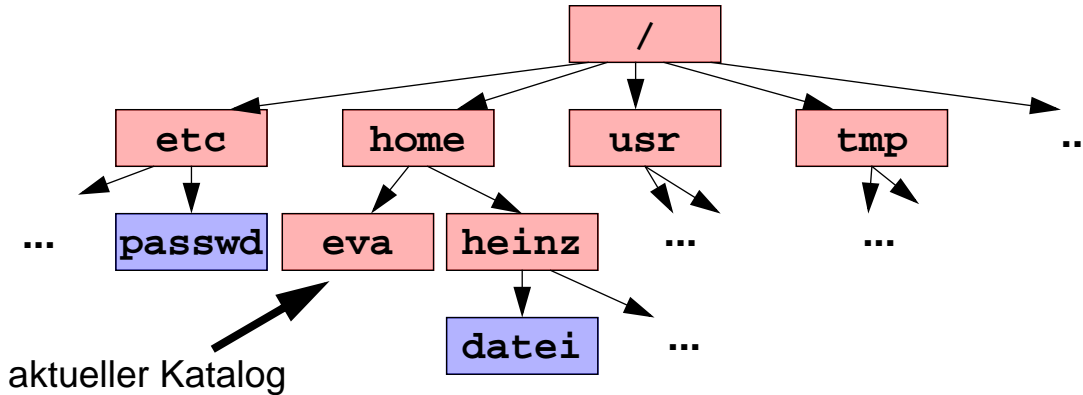
◆ Dateikataloge (*Directories*)

- Enthalten reguläre Dateien oder weitere Kataloge

➔ Baumförmige Struktur des Dateisystems

◆ Dateien werden durch Aneinanderreihung der Katalognamen und des Dateinamens eindeutig benannt

■ Baumstruktur & Pfade im Dateibaum



◆ z. B. `/home/heinz/datei` `/tmp` `../heinz/datei`

◆ `/` ist Trennsymbol (*Slash*) zwischen Pfadkomponenten
Pfade, die mit `/` beginnen, starten im Wurzelkatalog (*root*)
sonst Beginn implizit im aktuellem Katalog

◆ jeder Katalog enthält Verweise auf sich selbst (`.`) und auf den darüberliegenden Katalog (`..`) — in Pfadnamen verwendbar!

◆ aktueller Katalog kann mit Kommando `cd` gewechselt werden
z. B. `cd ../heinz`

C.12 Benutzerumgebung

- Das Betriebssystem (UNIX) und eine Reihe von Dienstprogrammen (*Tools*) stellen eine Umgebung bereit, die von der eigentlichen Rechnerarchitektur abstrahiert
- Jedem Benutzer ist ein eigener Dateikatalog zugeordnet (*Home-Directory*)
- Nach dem *Login* wird automatisch eine Fensteroberfläche mit mehreren Fenstern gestartet (*X-Windows*)
 - ◆ Weitere Fenster sind über Menüs mit der Maus zu starten
- Jedes Fenster entspricht einem virtuellen Terminal
 - ◆ Kommandointerpreter (Shell) zur Eingabe von Kommandos (C-Compiler, Dateikatalog auflisten, ...)
 - ◆ Editor (Emacs) zur Eingabe von Programmen und Texten
 - ◆ ...

C.12 Benutzerumgebung (2)

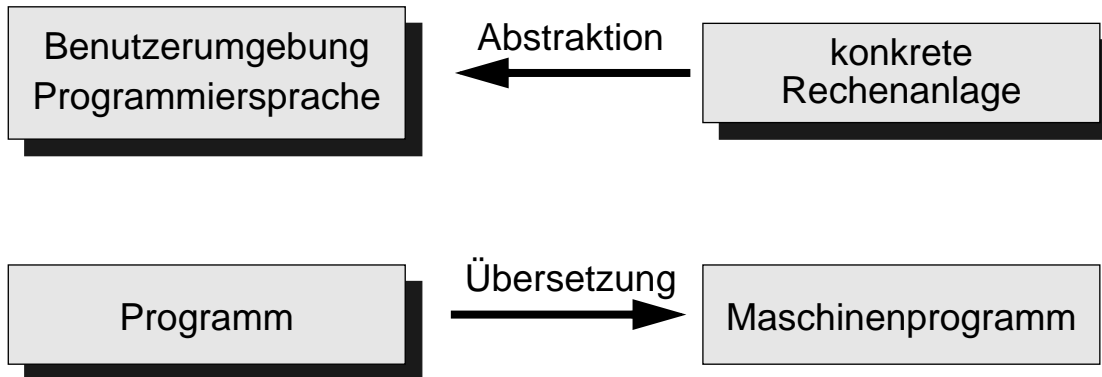
- Beschreibung über die Benutzerumgebung und alle Kommandos sind online auf dem Rechner abrufbar
 - ◆ Integriert in World-Wide-Web (WWW)
 - ◆ Tool: *Mosaic* (über Menü starten, dann selbsterklärend)

C.13 Programmierumgebung

- Programmiersprache bietet Sicht auf eine abstrakte Rechenanlage
 - ◆ Statt Bits, Bytes und Worten:
 Ganze und reelle Zahlen, Zeichen
 - ◆ Statt einfachen Abfragen und direkten Sprüngen an Adressen:
 if - then - else, Schleifen, ...

C.13 Programmierumgebung (2)

- Abbildung Programmiersprache → Maschinensprache
- ◆ Interpreter: Ein Dienstprogramm interpretiert das Programm des Anwenders
- ◆ Compiler: Ein Dienstprogramm übersetzt das Anwenderprogramm in Maschinensprache



C.14 Begriffe

Programm:

Eine in einer Programmiersprache abgefaßte Verarbeitungsvorschrift mit einer (evtl. impliziten) Vereinbarung der Daten

Maschinensprache:

Programmiersprache, die nur Anweisungen zuläßt, die von einer bestimmten Rechenanlage (unmittelbar) ausgeführt werden können

Maschinenprogramm:

Ein in Maschinensprache abgefaßtes Programm

Maschinenorientierte Programmiersprache:

Programmiersprache, deren Anweisungen ähnliche/gleiche Struktur haben wie die der Maschinensprache einer bestimmten Rechenanlage

Maschinenorientiertes Programm:

Ein in maschinenorientierter Programmiersprache abgefaßtes Programm


Höhere / problemorientierte Programmiersprache:

Programmiersprache, die dazu dient, Programme eines bestimmten Anwendungsbereiches unabhängig von einer bestimmten Rechenanlage abzufassen, und die dem Anwendungsbereich besonders angemessen ist (FORTRAN, PASCAL, COBOL, C, C++, ...).

C - Programm:

Ein in der Programmiersprache C abgefaßtes Programm

D Datentypen, einfache Programmstruktur, Operatoren, Ausdrücke

- Datentypen 
 - Konstanten
 - Variablen
 - ◆ Ganze Zahlen
 - ◆ Fließkommazahlen
 - ◆ Zeichen
 - ◆ Zeichenketten
-
- Aufbau eines C-Programms — erste Hinweise
 - ◆ Hauptprogramm
 - ◆ Bedingte Anweisung
-
- Ausdrücke
 - ◆ Operatoren
 - ◆ Typumwandlung in Ausdrücken
 - ◆ Vorrangregeln bei Operatoren

D.1 Datentypen

- Menge von Werten
+
Menge von Operationen auf den Werten
 - ◆ **Konstanten** stellen einen Wert dar
 - ◆ **Variablen** sind Namen für Speicherplätze, die einen Wert aufnehmen können
 - ↳ Konstanten und Variablen besitzen einen **Typ**

- Datentypen legen fest:
 - ◆ Repräsentation der Werte im Rechner
 - ◆ Größe des Speicherplatzes für Variablen
 - ◆ erlaubte Operationen

- Festlegung des Datentyps
 - ◆ implizit durch Verwendung und Schreibweise (Zahlen, Zeichen)
 - ◆ explizit durch **Deklaration** (Variablen)

D.2 Standardtypen in C

- Eine Reihe häufig benötigter Datentypen ist in C vordefiniert

<code>char</code>	Zeichen (im ASCII-Code dargestellt, 8 Bit)
<code>int</code>	ganze Zahl (16 oder 32 Bit)
<code>float</code>	Gleitkommazahl (32 Bit) etwa auf 6 Stellen genau
<code>double</code>	doppelt genaue Gleitkommazahl (64 Bit) etwa auf 12 Stellen genau
<code>void</code>	ohne Wert

D.2 Standardtypen in C (2)

- Die Bedeutung der Basistypen kann durch vorangestellte **Typ-Modifier** verändert werden

short, long

legt für den Datentyp `int` die Darstellungsbreite (i. a. 16 oder 32 Bit) fest.

Das Schlüsselwort `int` kann auch weggelassen werden

long double

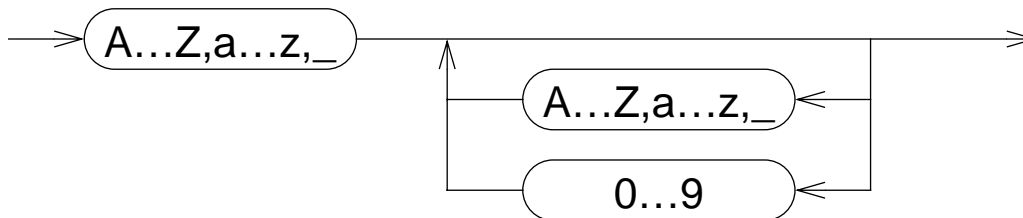
`double`-Wert mit erweiterter Genauigkeit (je nach Implementierung) – mindestens so genau wie `double`

signed, unsigned

legt für die Datentypen `char`, `short`, `long` und `int` fest, ob das erste Bit als Vorzeichenbit interpretiert wird oder nicht

D.3 Variablen

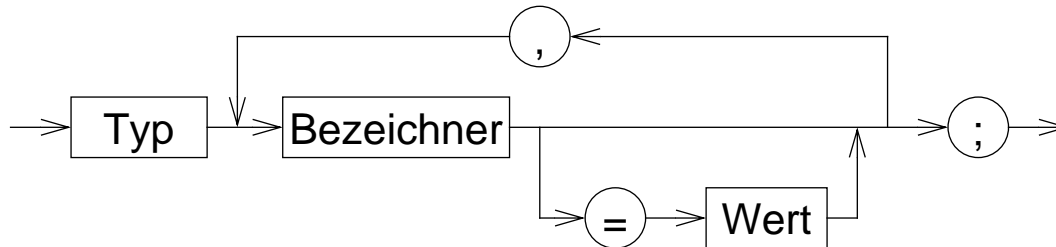
- Variablen besitzen
 - ◆ **Namen** (Bezeichner)
 - ◆ Typ
 - ◆ zugeordneten Speicherbereich für einen Wert des Typs
Inhalt des Speichers (= **aktueller Wert** der Variablen) ist veränderbar!
 - ◆ **Lebensdauer**
- Bezeichner



(Buchstabe oder `_`,
evtl. gefolgt von beliebig vielen Buchstaben, Ziffern oder `_`)

D.3 Variablen (2)

- Typ und Bezeichner werden durch eine **Variablen-Deklaration** festgelegt
 - ◆ reine Deklarationen werden erst in einem späteren Kapitel benötigt
 - ◆ vorerst beschränken wir uns auf Deklarationen in **Variablen-Definitionen**
- eine **Variablen-Definition** deklariert eine Variable und reserviert den benötigten Speicherbereich



D.3 Variablen (3)

■ Variablen-Definition: Beispiele

```
int a1;  
float a, b, c, dis;  
int anzahl_zeilen=5;  
char Trennzeichen;
```

◆ Position im Programm:

- nach jeder "{"
- außerhalb von Funktionen

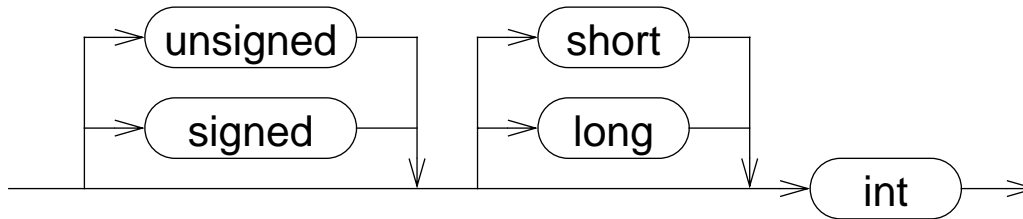
■ Wert kann bei der Definition initialisiert werden

■ Wert ist durch Wertzuweisung und spezielle Operatoren veränderbar

■ Lebensdauer ergibt sich aus der Programmstruktur

D.4 Ganze Zahlen

■ Definition



■ Speicherbedarf(short int) ≤ Speicherbedarf(int) ≤ Speicherbedarf(long int)

■ Speicherbedarf(int): heute meist 32 Bit

■ Konstanten (Beispiele):

42, -117

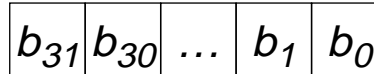
035 (oktal = 29₁₀)

0x10 (hexadezimal = 16₁₀)

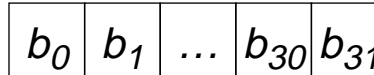
0x1d (hexadezimal = 29₁₀)

D.4 Ganze Zahlen (2)

- Repräsentation (meistens):
(Motorola, HP, SPARC, ...)



bei Intel-Prozessoren:



- Wertebereich:

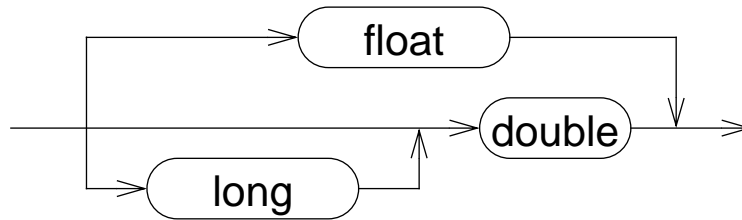
unsigned: $0 \dots 2^{32} - 1$

signed: $-2^{31} \dots 2^{31} - 1$

b_{31} ist Vorzeichenbit

D.5 Fließkommazahlen

■ Definition



■ Speicherbedarf(float) ≤ Speicherbedarf(double) ≤ Speicherbedarf(long double)

■ Speicherbedarf(float): 32 Bit

■ Konstanten (Beispiele):

◆ normale Dezimalpunkt-Schreibweise

3.14, -2.718, 368.345, 0.003

1.0

aber nicht einfach 1 (wäre eine int-Konstante!)

◆ 10er-Potenz Schreibweise ($368.345 = 3.68345 \cdot 10^2$, $0.003 = 3.0 \cdot 10^{-3}$)

3.68345e2, 3.0e-3

D.5 Fließkommazahlen (2)

- Repräsentation: durch Mantisse m und Exponent e
 - ◆ Wert = $m \cdot 2^e$
- Genauigkeit
 - ◆ hängt von der Anzahl der Bits für die Mantisse ab
- Wertebereich
 - ◆ hängt von der Anzahl der Bits für den Exponent ab

D.5 Fließkommazahlen (3)

- Repräsentation abhängig vom jeweiligen Prozessor
(häufig verwendetes Format: IEEE-Standard)

	Mant. (Bit)	Exp. (Bit)	kleinste darstellbare positive Zahl größte darstellbare Zahl prozentuale Genauigkeit
float	24	8	$1.175494351 \cdot 10^{-38}$ $3.402823466 \cdot 10^{+38}$ $1.192092896 \cdot 10^{-07}$
double	53	11	$2.2250738585072014 \cdot 10^{-308}$ $1.7976931348623157 \cdot 10^{+308}$ $2.2204460492503131 \cdot 10^{-16}$
long double 80 Bit (Intel-Architektur)	65	15	$3.3621031431120935062627 \cdot 10^{-4932}$ $1.1897314953572317650213 \cdot 10^{+4932}$ $1.0842021724855044340075 \cdot 10^{-19}$
long double 128 Bit	113	15	$3.36210314311209350626267781732175260 \cdot 10^{-4932}$ $1.189731495357231765085759326628007016 \cdot 10^{+4932}$ $1.925929944387235853055977942584927319 \cdot 10^{-34}$

D.6 Zeichen

- Bezeichnung: `char`
- Speicherbedarf: 1 Byte
- Repräsentation: ASCII-Code
zählt damit zu den ganzen Zahlen
- Konstanten: Zeichen durch `' '` geklammert
- ◆ Beispiele: `'a'`, `'x'`
- ◆ Sonderzeichen werden durch **Escape-Sequenzen** beschrieben
 - Tabulator: `'\t'` Backslash: `'\\'`
 - Zeilentrenner: `'\n'` Backspace: `'\b'`
 - Apostroph: `'\''`

D.6 Zeichen (2)

American Standard Code for Information Interchange (ASCII)

NUL 00	SOH 01	STX 02	ETX 03	EOT 04	ENQ 05	ACK 06	BEL 07
BS 08	HT 09	NL 0A	VT 0B	NP 0C	CR 0D	SO 0E	SI 0F
DLE 10	DC1 11	DC2 12	DC3 13	DC4 14	NAK 15	SYN 16	ETB 17
CAN 18	EM 19	SUB 1A	ESC 1B	FS 1C	GS 1D	RS 1E	US 1F
SP 20	! 21	" 22	# 23	\$ 24	% 25	& 26	' 27
(28) 29	* 2A	+ 2B	, 2C	- 2D	. 2E	/ 2F
0 30	1 31	2 32	3 33	4 34	5 35	6 36	7 37
8 38	9 39	: 3A	; 3B	< 3C	= 3D	> 3E	? 3F
@ 40	A 41	B 42	C 43	D 44	E 45	F 46	G 47
H 48	I 49	J 4A	K 4B	L 4C	M 4D	N 4E	O 4F
P 50	Q 51	R 52	S 53	T 54	U 55	V 56	W 57
X 58	Y 59	Z 5A	[5B	\ 5C] 5D	^ 5E	_ 5F
` 60	a 61	b 62	c 63	d 64	e 65	f 66	g 67
h 68	i 69	j 6A	k 6B	l 6C	m 6D	n 6E	o 6F
p 70	q 71	r 72	s 73	t 74	u 75	v 76	w 77
x 78	y 79	z 7A	{ 7B	 7C	} 7D	~ 7E	DEL 7F

D.7 Zeichenketten

- Bezeichnung: `char *`
- Speicherbedarf: (Länge + 1) Bytes
- Repräsentation: Folge von Einzelzeichen,
letztes Zeichen: 0-Byte (ASCII-Wert 0)
- Werte: alle endlichen Folgen von `char`-Werten
- Konstanten: Zeichenkette durch " " geklammert
 - ◆ Beispiel: `"Dies ist eine Zeichenkette"`
 - ◆ Sonderzeichen wie bei `char`, " wird durch `\` dargestellt
- Beispiel für eine Definition einer Zeichenkette:
`char *Mitteilung = "Dies ist eine Mitteilung\n";`

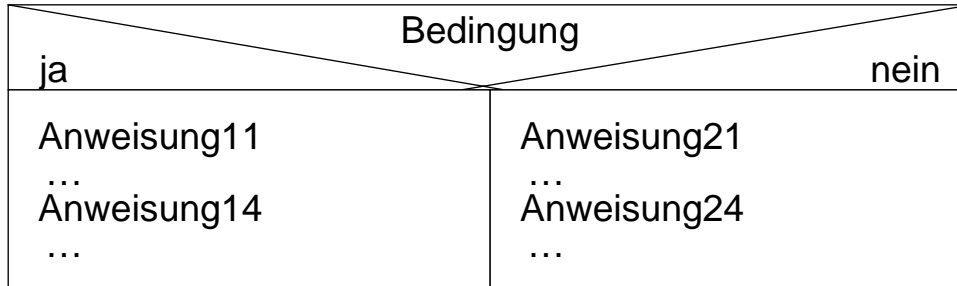
D.8 Aufbau eines C-Programms

- frei formulierbar - **Zwischenräume** (*Leerstellen, Tabulatoren, Zeilenvorschub und Kommentare*) werden i. a. ignoriert - sind aber zur eindeutigen Trennung direkt benachbarter Worte erforderlich
- **Kommentar** wird durch `/*` und `*/` geklammert, keine Schachtelung möglich
- Hauptprogramm

```
main()  
{  
    Variablendefinitionen  
    Anweisungen  
}
```
- Anweisungen werden generell durch `;` abgeschlossen

D.8 Aufbau eines C-Programms (2)

Bedingte Anweisung



```
if ( Bedingung ) {  
    Anweisung1  
    ...  
    Anweisung14  
    ...  
} else {  
    Anweisung21  
    ...  
    Anweisung24  
    ...  
}
```

■ Den `else`-Teil kann man auch weglassen!

D.9 Ausdrücke

- Ausdruck = gültige Kombination von
Operatoren, Konstanten und Variablen
- Reihenfolge der Auswertung
 - ◆ Die Vorrangregeln für Operatoren legen die Reihenfolge fest, in der Ausdrücke abgearbeitet werden
 - ◆ Geben die Vorrangregeln keine eindeutige Aussage, ist die Reihenfolge undefiniert
 - ◆ Mit Klammern () können die Vorrangregeln überstimmt werden
 - ◆ Es bleibt dem Compiler freigestellt, Teilausdrücke in möglichst effizienter Folge auszuwerten

D.10 Operatoren

1 Zuweisungsoperator =

➔ Zuweisung eines Werts an eine Variable

■ Beispiel:

```
int a;  
a = 20;
```

→ für alle `int` und `float` Werte erlaubt

<code>+</code>	Addition
<code>-</code>	Subtraktion
<code>*</code>	Multiplication
<code>/</code>	Division
<code>%</code>	Rest bei Division, (modulo)
unäres <code>-</code>	negatives Vorzeichen (z. B. <code>-3</code>)
unäres <code>+</code>	positives Vorzeichen (z. B. <code>+3</code>)

■ Beispiel:

```
a = -5 + 7 * 20 - 8;
```

→ Verkürzte Schreibweise für Operationen auf einer Variablen

$a \text{ op} = b \equiv a = a \text{ op } b$

mit $\text{op} \in \{+, -, *, /, \%, \ll, \gg, \&, \wedge, |\}$

■ Beispiele:

```
a = -8;
```

```
a += 24;
```

```
a /= 2;
```

```
/* -> a: 16 */
```

```
/* -> a: 8 */
```

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

■ **Beachte!** Ergebnistyp `int`: wahr (true) = 1
 falsch (false) = 0

■ **Beispiele:**

```
a > 3
a <= 5
a == 0
if ( a >= 3 ) { ...
```


■ Beispiel:

`a = 5; b = 3; c = 7;`

`a > b && a > c`

$\underbrace{a > b}_{1} \text{ und } \underbrace{a > c}_{0}$
 $\underbrace{\hspace{10em}}_0$

■ Die Bewertung solcher Ausdrücke wird abgebrochen, sobald das Ergebnis feststeht!

`(a > c) && ((d=a) > b)`

$\underbrace{(a > c)}_0$ $\underbrace{((d=a) > b)}$ wird nicht ausgewertet

↓
Gesamtergebnis=*falsch* → (d=a) wird nicht ausgeführt

6 Bitweise logische Operatoren

➔ Operation auf jedem Bit einzeln (Bit 1 = wahr, Bit 0 = falsch)

"nicht"

~

"und"

&

"oder"

|

Antivalenz
"exklusives oder"

\wedge	f	w
f	f	w
w	w	f

■ Beispiele:

x

1	0	0	1	1	1	0	0
---	---	---	---	---	---	---	---

~x

0	1	1	0	0	0	1	1
---	---	---	---	---	---	---	---

7

0	0	0	0	0	1	1	1
---	---	---	---	---	---	---	---

x | 7

1	0	0	1	1	1	1	1
---	---	---	---	---	---	---	---

x & 7

0	0	0	0	0	1	0	0
---	---	---	---	---	---	---	---

x ^ 7

1	0	0	1	1	0	1	1
---	---	---	---	---	---	---	---

➔ Bits werden im Wort verschoben

<< Links-Shift

>> Rechts-Shift

■ Beispiel:

x	1	0	0	1	1	1	0	0
x << 2	0	1	1	1	0	0	0	0

`++` inkrement
`--` dekrement

- **linksseitiger Operator:** `++x` bzw. `--x`
 - es wird der Inhalt von `x` inkrementiert bzw. dekrementiert
 - das Resultat wird als Ergebnis geliefert

- **rechtsseitiger Operator:** `x++` bzw. `x--`
 - es wird der Inhalt von `x` als Ergebnis geliefert
 - anschließend wird `x` inkrementiert bzw. dekrementiert.

- **Beispiele:**

```
a = 10;  
b = a++;            /* -> b: 10 und a: 11 */  
c = ++a;            /* -> c: 12 und a: 12 */
```

A ? B : C

→ der Operator dient zur Formulierung von Bedingungen in Ausdrücken

■ zuerst wird Ausdruck **A** bewertet

■ ist **A ungleich 0**, so hat der gesamte Ausdruck als Wert den Wert des Ausdrucks **B**,

■ sonst den Wert des Ausdrucks **C**

■ Beispiel:

```
c = a>b ? a : b;
```

```
/* z = max(a,b) */
```

besser:

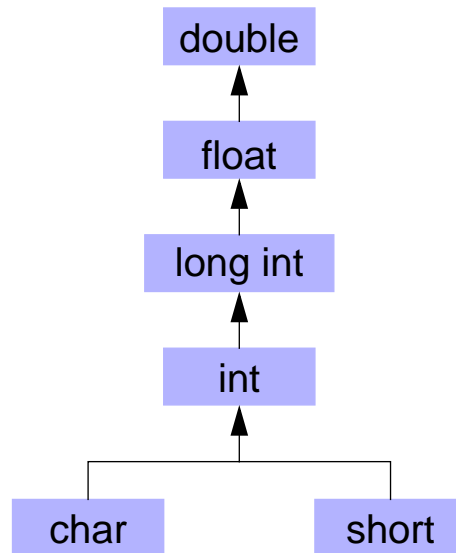
```
c = (a>b) ? a : b;
```

,

- der Komma-Operator erlaubt die Aneinanderreihung mehrerer Ausdrücke
- ein so gebildeter Ausdruck hat als Wert den Wert des letzten Teilausdrucks

D.11 Typumwandlung in Ausdrücken

- Enthält ein Ausdruck Operanden unterschiedlichen Typs, erfolgt eine automatische Umwandlung in den Typ des in der **Hierarchie der Typen** am höchsten stehenden Operanden. (*Arithmetische Umwandlungen*)



Hierarchie der Typen (Auszug)

D.12 Vorrangregeln bei Operatoren

Operatorklasse	Operatoren	Assoziativität
unär	! ~ ++ -- + -	von rechts nach links
multiplikativ	* / %	von links nach rechts
additiv	+ -	von links nach rechts
shift	<< >>	von links nach rechts
relational	< <= > >=	von links nach rechts
Gleichheit	== !=	von links nach rechts
bitweise	&	von links nach rechts
bitweise	^	von links nach rechts
bitweise	 	von links nach rechts
logisch	&&	von links nach rechts
logisch	 	von links nach rechts
Bedingte Bewertung	?:	von rechts nach links
Zuweisung	= op=	von rechts nach links
Reihung	,	von links nach rechts

E Ablaufkontrolle

- Struktur eines C-Hauptprogramms
- Anweisungen und Blöcke
- Bedingte Anweisung
 - ◆ einfache Verzweigung
 - ◆ mehrfache Verzweigung
- Fallunterscheidung
- Schleifen
 - ◆ abweisende Schleife
 - ◆ nicht abweisende Schleife
 - ◆ Laufanweisung
 - ◆ Schleifensteuerung

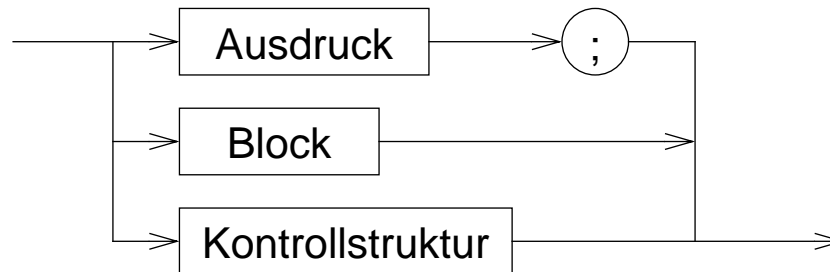
E.1 Struktur eines C-Hauptprogramms

E.1 Struktur eines C-Hauptprogramms

```
main()  
{  
    Variablendefinitionen  
    Anweisungen  
}
```

E.2 Anweisungen

Anweisung:

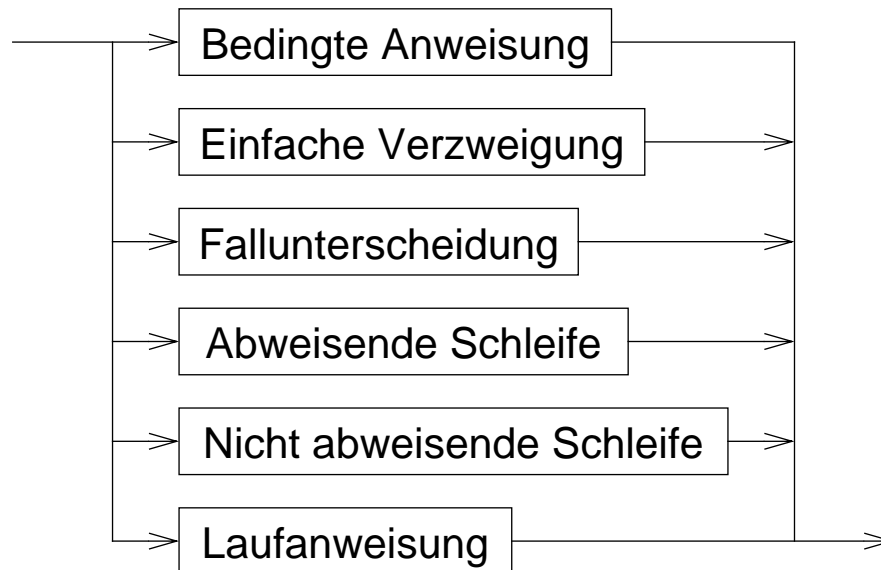


- Zusammenfassung mehrerer Anweisungen
- Lokale Variablendefinitionen → Hilfsvariablen
- Schaffung neuer Sichtbarkeitsbereiche (**Scopes**) für Variablen
- ◆ bei Namensgleichheit ist immer die Variable des innersten Blocks sichtbar

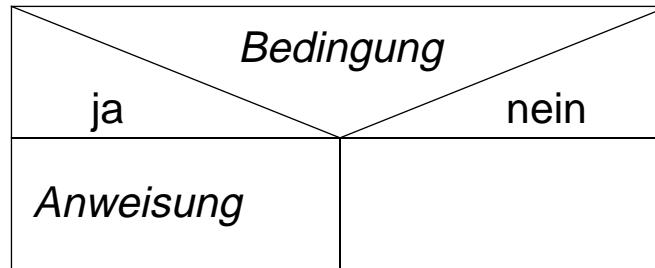
```
main()
{
    int x, y, z;
    x = 1;
    {
        int a, b, c;
        a = x+1;
        {
            int a, x;
            x = 2;
            a = 3;
        }
        /* a: 2, x: 1 */
    }
}
```

- Kontrolle des Programmablaufs in Abhängigkeit von dem Ergebnis von Ausdrücken

Kontrollstruktur:

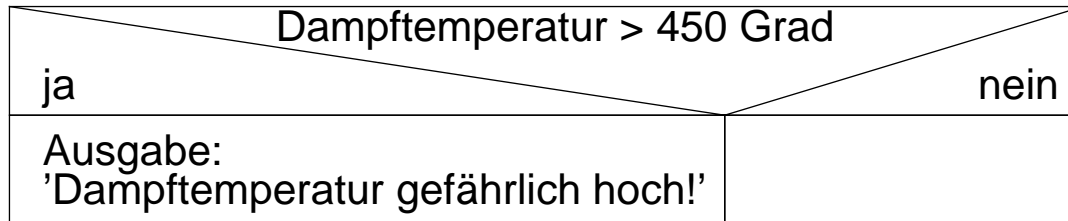


1 Bedingte Anweisung



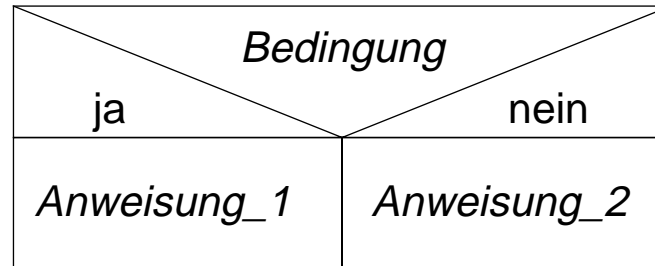
```
if ( Bedingung )  
    Anweisung
```

■ Beispiel:



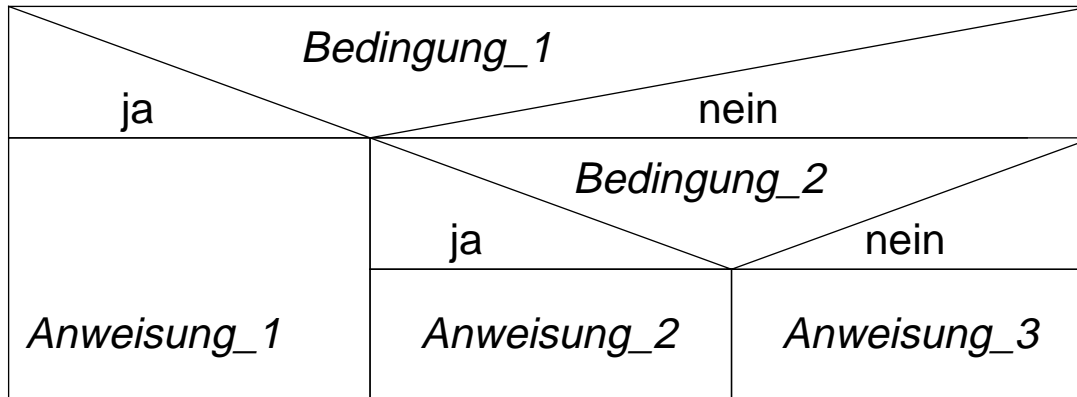
```
if (temp >= 450.0)  
    printf("Dampftemperatur gefaehrlich hoch!\n");
```

1 Bedingte Anweisung einfache Verzweigung



```
if ( Bedingung )  
    Anweisung_1  
else  
    Anweisung_2
```

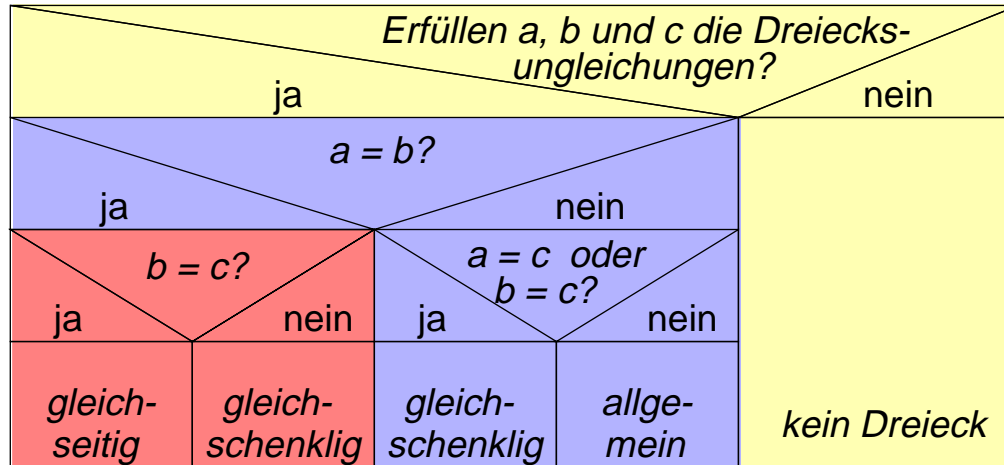
1 Bedingte Anweisung mehrfache Verzweigung



```
if ( Bedingung )  
    Anweisung_1  
else if ( Bedingung_2 )  
    Anweisung_2  
else  
    Anweisung_3
```

1 Bedingte Anweisung mehrfache Verzweigung (2)

■ Beispiel: Eigenschaften von Dreiecken — Struktogramm



1 Bedingte Anweisung mehrfache Verzweigung (3)

- Beispiel: Eigenschaften von Dreiecken — Programm

```
printf("Die Seitenlaengen %f, %f und %f bilden ", a, b, c);
```

```
if ( a < b+c && b < a+c && c < a+b )  
    if ( a == b )  
        if ( b == c )  
            printf("ein gleichseitiges");  
        else  
            printf("ein gleichschenkliges");  
    else  
        if ( a==c || b == c )  
            printf("ein gleichschenkliges");  
        else  
            printf("ein allgemeines");  
else  
    printf("kein");  
printf(" Dreieck");
```

- Mehrfachverzweigung = Kaskade von if-Anweisungen
- verschiedene Fälle in Abhängigkeit von einem ganzzahligen Ausdruck

ganzzahliger Ausdruck = ?				
Wert1	Wert2			sonst
Anw. 1	Anw. 2		Anw. n	Anw. x

```
switch ( Ausdruck ) {  
    case Wert_1:  
        Anweisung_1  
        break;  
    case Wert_2:  
        Anweisung_2  
        break;  
    ..  
    case Wert_n:  
        Anweisung_n  
        break;  
    default:  
        Anweisung_x  
}
```

```
#include <stdio.h>

main()
{
    char zeichen;
    int i;
    int ziffern, leer, sonstige;

    ziffern = leer = sonstige = 0;

    while ((zeichen = getchar()) != EOF)
        switch (zeichen) {
            case '0':
            case '1':
            case '2':
            case '3':
            case '4':
            case '5':
            case '6':
            case '7':
            case '8':
            case '9':                ziffern++;
                                    break;

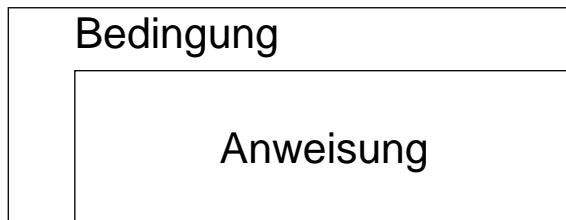
            case ' ':
            case '\n':
            case '\t':                leer++;
                                    break;

            default:                 sonstige++;
        }

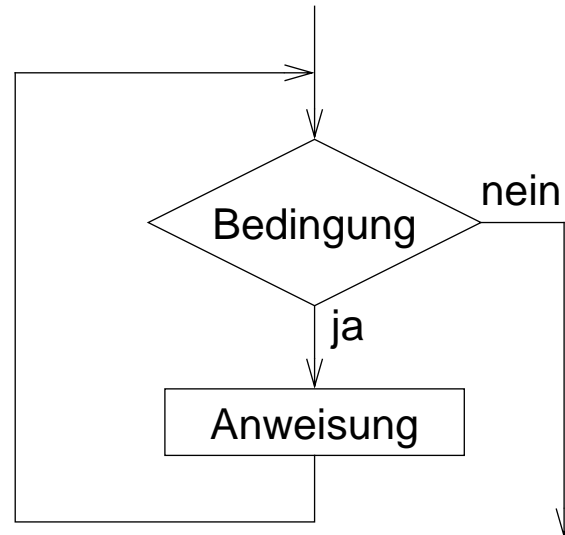
    printf("Zahl der Ziffern = %d\n", ziffern);
    printf("Zahl der Leerzeichen = %d\n", leer);
    printf("Zahl sonstiger Zeichen = %d\n", sonstige);
}
```

- Wiederholte Ausführung von Anweisungen in Abhängigkeit von dem Ergebnis eines Ausdrucks

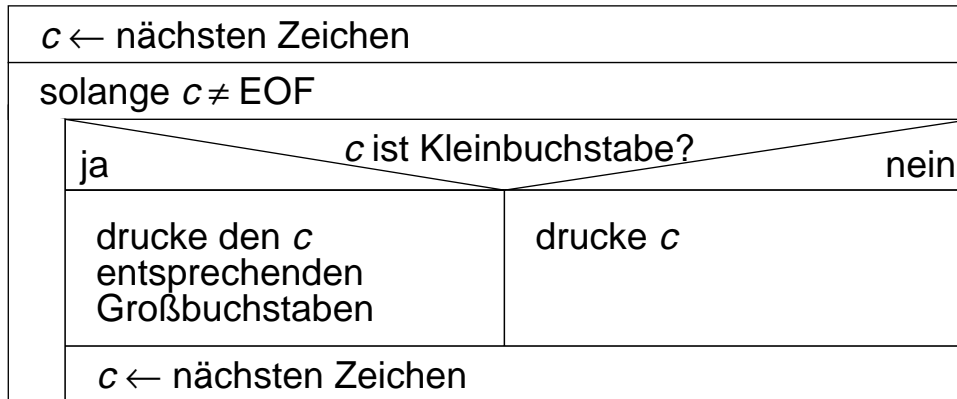
4 abweisende Schleife



```
while ( Bedingung )  
    Anweisung
```



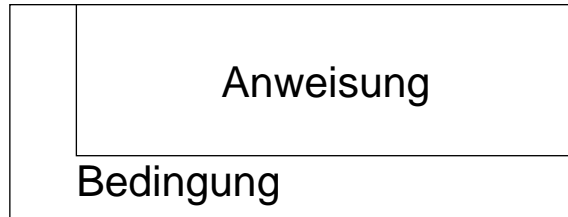
■ Beispiel: Umwandlung von Klein- in Großbuchstaben



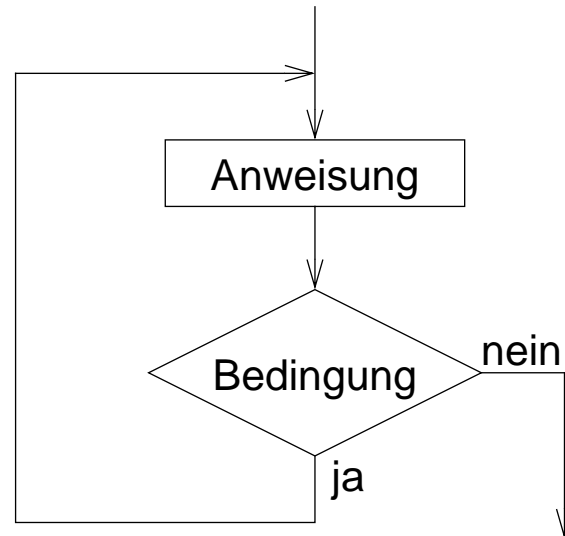
```
char c;  
c = getchar();  
while ( c != EOF ) {  
    if ( c >= 'a' && c <= 'z' )  
        putchar(c+'A'-'a');  
    else  
        putchar(c);  
    c = getchar();  
}
```

► abgekürzte Schreibweise

```
while ( (c = getchar()) != EOF )  
    if ( c >= 'a' && c <= 'z' )  
        putchar(c+'A'-'a');  
    else  
        putchar(c);
```



```
do  
  Anweisung  
while ( Bedingung )
```



6 Schleifen — Bsp. Sinusberechnung

■ Taylor-Reihe:
$$\sin x = \sum_{i=0}^n (-1)^i \frac{x^{2i+1}}{(2i+1)!} + r_{n+1}(x)$$

wobei: $|r_{n+1}(x)| \leq |R_n(x)|$ mit $R_n(x) = (-1)^n \frac{x^{2n+1}}{(2n+1)!}$

Also:
$$\sin x = \sum_{i=0}^n R_i(x) + r_{n+1}(x)$$

mit
$$R_n(x) = x \cdot \left(-\frac{x^2}{2 \cdot 3}\right) \cdot \left(-\frac{x^2}{4 \cdot 5}\right) \cdots \left(-\frac{x^2}{(2n) \cdot (2n+1)}\right)$$

Eingabe x, ε
$Rest \leftarrow x$ $k \leftarrow 0$ $sinus \leftarrow 0$
solange $ Rest > \varepsilon$
$sinus \leftarrow sinus + Rest$ $k \leftarrow k + 2$
$Rest \leftarrow -\frac{x^2}{k(k+1)} Rest$
Ausgabe $sinus$

6 Schleifen — Bsp. Sinusberechnung (2) [E.4](#) Kontrollstrukturen

■ Programm

```
#include <stdio.h>
#include <math.h>

main()
{
    double x;                /* Argument */
    double sinus;           /* Summenwert */
    double x_quadrat;       /* x*x */
    double rest;            /* Summenglied */
    double eps;             /* Genauigkeit */
    int k;                  /* Nenner */

    printf("Berechnung des Sinus von ");
    scanf("%lf", &x);
    printf("Die Genauigkeit soll sein: ");
    scanf("%lf", &eps);

    k = 0;
    sinus = 0.0;
    rest = x;
    x_quadrat = x*x;

    while ( fabs(rest) > eps ) {
        sinus += rest;
        k += 2;
        rest *= -x_quadrat/(k*(k+1));
    }

    printf("sin(%lf) = %lf +- %lf\n", x, sinus, fabs(rest));
}
```

$v \leftarrow$ Startausdruck (Inkrement) Endausdruck

Anweisung

```
for (v = Startausdruck; v <= Endausdruck; v += Inkrement)
    Anweisung
```

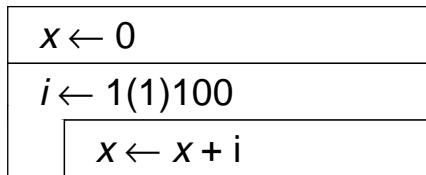
allgemein:

```
for (Ausdruck_1; Ausdruck_2; Ausdruck_3)
    Anweisung
```

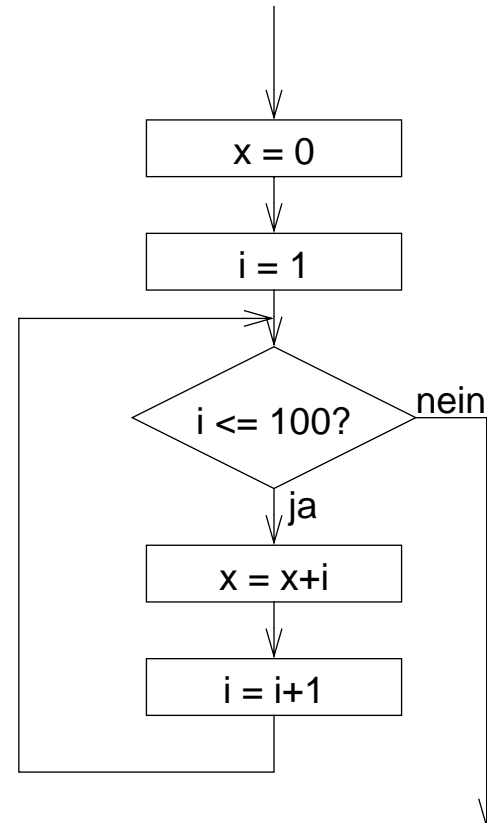
```
Ausdruck_1;
while (Ausdruck_2) {
    Anweisung
    Ausdruck_3;
}
```

7 Laufanweisung (2)

■ Beispiel: Berechne $x = \sum_{i=1}^{100} i$



```
x = 0;  
for ( i=1; i<=100; i++)  
    x += i;
```



■ Negativbeispiele

$$\blacklozenge x = \sum_{i=1}^{100} i$$

```
for ( x = 0, i = 1; i <= 100; x += i++ );
```

◆ Sinusberechnung

```
x_quadrat = x*x;
```

```
for ( i = 0, sinus = 0.0, rest = x;  
      fabs(rest) > eps;  
      i += 2, rest *= -x_quadrat/(i+(i+1) )  
      sinus += rest;
```

■ break

- ◆ bricht die umgebende Schleife bzw. `switch`-Anweisung ab

```
char c;  
  
do {  
    if ( ( c = getchar() ) == EOF ) break;  
    putchar(c);  
}  
while ( c != '\n' );
```

■ continue

- ◆ bricht den aktuellen **Schleifendurchlauf** ab
- ◆ setzt das Programm mit der Ausführung des Schleifenkopfes fort

F Funktionen und Programmstruktur

- Funktionen
- Gültigkeitsbereiche und Lebensdauer von Variablen
- Getrennt Übersetzung von Programmteilen
- C-Preprozessor
 - ◆ Include-Dateien
 - ◆ Makros

F.1 Funktionen

- **Funktion =**
Programmstück (Block), das mit einem **Namen** versehen ist und dem zum Ablauf **Parameter** übergeben werden können
- Funktionen sind die elementaren Bausteine für Programme
 - ↳ gliedern umfangreiche, schwer überblickbare Aufgaben in kleine Komponenten
 - ↳ erlauben die Wiederverwendung von Programmkomponenten
 - ↳ verbergen Implementierungsdetails vor anderen Programmteilen (**Black-Box-Prinzip**)

F.1 Funktionen (2)

- ↳ Funktionen dienen der Abstraktion
- Name und Parameter abstrahieren
 - vom tatsächlichen Programmstück
 - von der Darstellung und Verwendung von Daten
- Verwendung
 - ◆ mehrmals benötigte Programmstücke können durch Angabe des Funktionsnamens aufgerufen werden
 - ◆ Schrittweise Abstraktion
(**Top-Down-** und **Bottom-Up-**Entwurf)

F.1 Funktionen (3) — Beispiel Sinusberechnung

```
#include <stdio.h>
#include <math.h>

double sinus (double x)
{
    double summe;
    double x_quadrat;
    double rest;
    int k;

    k = 0;
    summe = 0.0;
    rest = x;
    x_quadrat = x*x;

    while ( fabs(rest) > 1e-9 ) {
        summe += rest;
        k += 2;
        rest *= -x_quadrat/(k*(k+1));
    }
    return(summe);
}
```

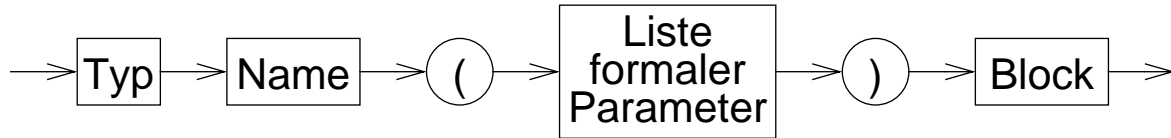
```
main()
{
    double wert;

    printf("Berechnung des Sinus von ");
    scanf("%lf", &x);
    printf("sin(%lf) = %lf\n",
           wert, sinus(wert));
}
```

- beliebige Verwendung von `sinus` in Ausdrücken:

```
y = exp(tau*t) * sinus(f*t);
```

F.2 Funktionsdefinition



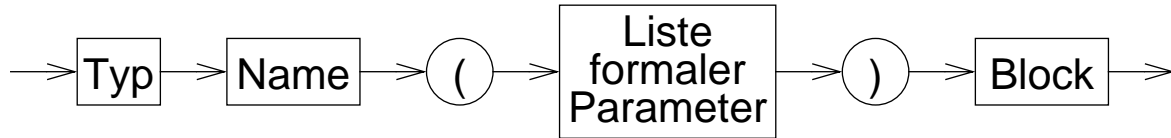
■ Typ

- ◆ Typ des Werts, der am Ende der Funktion als Wert zurückgegeben wird
- ◆ beliebiger Typ
- ◆ `void` = kein Rückgabewert

■ Name

- ◆ beliebiger Bezeichner, kein Schlüsselwort

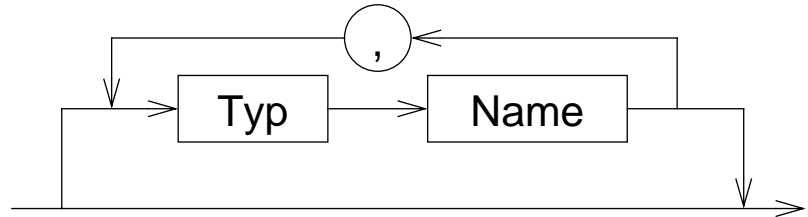
F.2 Funktionsdefinition (2)



■ Liste formaler Parameter

◆ **Typ**: beliebiger Typ

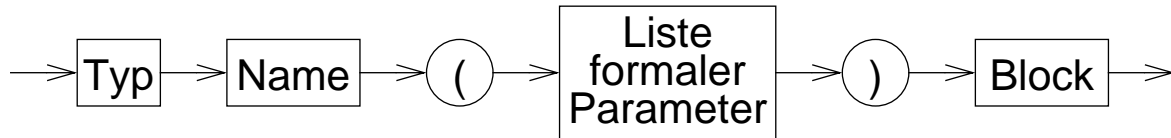
◆ **Name**:
beliebiger Bezeichner



◆ die formalen Parameter stehen für die Werte, die beim Aufruf an die Funktion übergeben wurden (= **aktuelle Parameter**)

◆ die formalen Parameter verhalten sich wie Variablen, die im **Funktionsrumpf** definiert sind und mit den aktuellen Parametern vorbelegt werden

F.2 Funktionsdefinition (3)



■ Block

- ◆ beliebiger Block
- ◆ zusätzliche Anweisung

```
return ( Ausdruck );
```

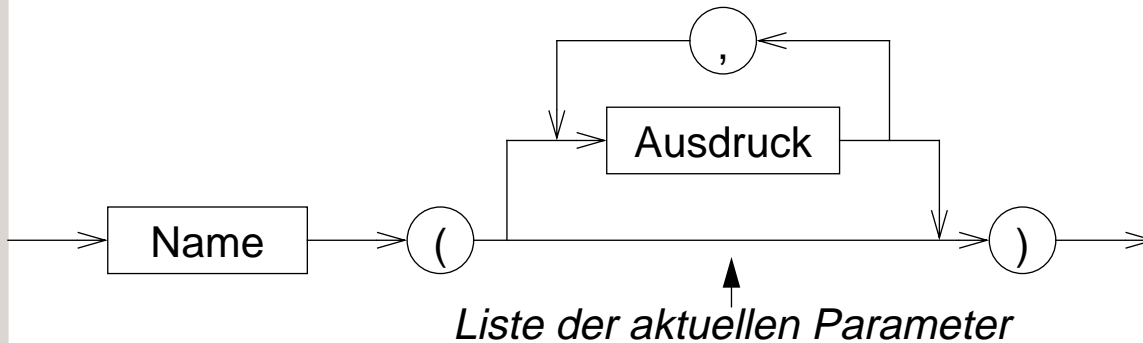
oder

```
return;
```

bei `void`-Funktionen

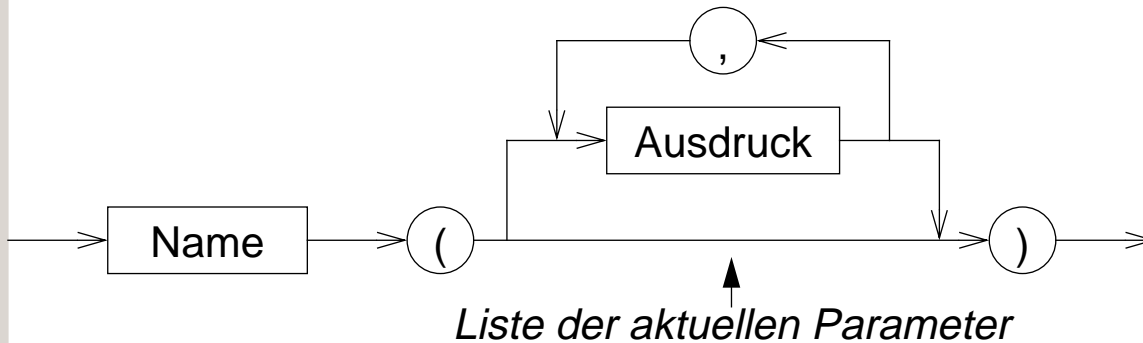
- Rückkehr aus der Funktion: das Programm wird nach dem Funktionsaufruf fortgesetzt
- der Typ des Ausdrucks muß mit dem Typ der Funktion übereinstimmen
- die Klammern können auch weggelassen werden

F.3 Funktionsaufruf



- Jeder Funktionsaufruf ist ein Ausdruck
- `void`-Funktionen können keine Teilausdrücke sein
 - ◆ wie Prozeduren in anderen Sprachen (z. B. Pascal)

F.3 Funktionsaufruf (2)



- Die Ausdrücke in der Parameterliste werden ausgewertet, **bevor** in die Funktion gesprungen wird
➔ **aktuelle Parameter**
- Anzahl und Typen der Ausdrücke in der Liste der aktuellen Parameter müssen mit denen der formalen Parameter in der Funktionsdefinition übereinstimmen
- Die Auswertungsreihenfolge der Parameterausdrücke ist **nicht** festgelegt

F.4 Funktionen — Beispiel

```
float power (float b, int e)
{
    float prod = 1.0;
    int i;

    for (i=1; i <= e; i++)
        prod *= b;
    return(prod);
}
```

```
float x, y;

y = power(2+x,4)+3;
```

≡

```
float x, y, power;
{
    float b = 2+x;
    int e = 4;
    float prod = 1.0;
    int i;

    for (i=1; i <= e; i++)
        prod *= b;
    power = prod;
}
y=power+3;
```

F.5 Funktionen — Regeln

- Funktionen werden global definiert
 - ↳ keine lokalen Funktionen/Prozeduren wie z. B. in Pascal
- `main()` ist eine normale Funktion, die aber automatisch als erste beim Programmstart aufgerufen wird
- rekursive Funktionsaufrufe sind zulässig
 - ↳ eine Funktion darf sich selbst aufrufen (z. B. zur Fakultätsberechnung)

```
fakultaet(int n)
{
    if ( n == 1 )
        return(1);
    else
        return( n * fakultaet(n-1) );
}
```

F.5 Funktionen — Regeln (2)

- Funktionen müssen **deklariert** sein, bevor sie aufgerufen werden
 - = Rückgabetyt und Parametertypen müssen bekannt sein
 - ◆ durch eine Funktionsdefinition ist die Funktion automatisch auch deklariert

- wurde eine verwendete Funktion vor ihrer Verwendung nicht deklariert, wird automatisch angenommen
 - Funktionswert vom Typ `int`
 - 1 Parameter vom Typ `int`
 - ↪ **schlechter Programmierstil → fehleranfällig**

F.5 Funktionen — Regeln (2)

■ Funktionsdeklaration

◆ soll eine Funktion vor ihrer Definition verwendet werden, kann sie durch eine **Deklaration** bekannt gemacht werden

◆ Syntax:

```
Typ Name ( Liste formaler Parameter );
```

➤ Parameternamen können weggelassen werden, die Parametertypen müssen aber angegeben werden!

◆ Beispiel:

```
double sinus(double);
```

F.6 Funktionsdeklarationen — Beispiel

```
#include <stdio.h>
#include <math.h>

double sinus(double);
/* oder: double sinus(double x); */

main()
{
    double wert;

    printf("Berechnung des Sinus von ");
    scanf("%lf", &wert);
    printf("sin(%lf) = %lf\n",
           wert, sinus(wert));
}
```

```
double sinus (double x)
{
    double summe;
    double x_quadrat;
    double rest;
    int k;

    k = 0;
    summe = 0.0;
    rest = x;
    x_quadrat = x*x;

    while ( fabs(rest) > 1e-9 ) {
        summe += rest;
        k += 2;
        rest *= -x_quadrat/(k*(k+1));
    }
    return(summe);
}
```

F.7 Parameterübergabe an Funktionen

- allgemein in Programmiersprachen vor allem zwei Varianten:
 - call by value
 - call by reference

1 call by value

- Normalfall in C
- Es wird eine Kopie des aktuellen Parameters an die Funktion übergeben
 - ➔ die Funktion kann den Übergabeparameter durch Zugriff auf den formalen Parameter lesen
 - ➔ die Funktion kann den Wert des formalen Parameters (also die Kopie!) ändern, ohne daß dies Auswirkungen auf den Wert des aktuellen Parameters beim Aufrufer hat
 - ➔ die Funktion kann über einen Parameter dem Aufrufer keine Ergebnisse mitteilen

- In C nur indirekt mit Hilfe von Zeigern realisierbar
- Der Übergabeparameter ist eine Variable und die aufgerufene Funktion erhält die Speicheradresse dieser Variablen
 - ➔ die Funktion kann den Übergabeparameter durch Zugriff auf den formalen Parameter lesen
 - ➔ wenn die Funktion den Wert des formalen Parameters verändert, ändert sie den Inhalt der Speicherzelle des aktuellen Parameters
 - ➔ auch der Wert der Variablen (aktueller Parameter) beim Aufrufer der Funktion ändert sich dadurch

1 Kurzüberblick

- bevor eine C-Quelle dem C-Compiler übergeben wird, wird sie durch einen Makro-Preprozessor bearbeitet
- Anweisungen an den Preprozessor werden durch ein #-Zeichen am Anfang der Zeile gekennzeichnet
- die Syntax von Preprozessoranweisungen ist unabhängig vom Rest der Sprache
- Preprozessoranweisungen werden nicht durch ; abgeschlossen!
- wichtigste Funktionen:

<code>#define</code>	Definition von Makros
<code>#include</code>	Einfügen von anderen Dateien

- Makros ermöglichen einfache textuelle Ersetzungen (parametrierbare Makros werden später behandelt)
- ein Makro wird durch die `#define`-Anweisung definiert
- Syntax:

```
#define Makroname Ersatztext
```

- eine Makrodefinition bewirkt, daß der Preprozessor im nachfolgenden Text der C-Quelle alle Vorkommen von ***Makroname*** durch ***Ersatztext*** ersetzt
- Beispiel:

```
#define EOF -1
```

■ `#include` fügt den Inhalt einer anderen Datei in eine C-Quelldatei ein

■ Syntax:

```
#include <Dateiname >  
oder  
#include "Dateiname "
```

■ mit `#include` werden *Header*-Dateien mit Daten, die für mehrere Quelldateien benötigt werden einkopiert

- Deklaration von Funktionen, Strukturen, externen Variablen
- Definition von Makros

■ wird **Dateiname** durch `< >` geklammert, wird eine **Standard-Header-Datei** einkopiert

■ wird **Dateiname** durch `" "` geklammert, wird eine Header-Datei des Benutzers einkopiert (vereinfacht dargestellt!)

1 Softwaredesign

- Grundsätzliche Überlegungen über die Struktur eines Programms **vor** Beginn der Programmierung
- Verschiedene Design-Methoden
 - ◆ Top-down Entwurf / Prozedurale Programmierung
 - traditionelle Methode
 - bis Mitte der 80er Jahre fast ausschließlich verwendet
 - an Programmiersprachen wie Fortran, Cobol, Pascal oder C orientiert
 - ◆ Objekt-orientierter Entwurf
 - moderne, sehr aktuelle Methode
 - Ziel: Bewältigung sehr komplexer Probleme
 - auf Programmiersprachen wie C++, Smalltalk oder Java ausgerichtet

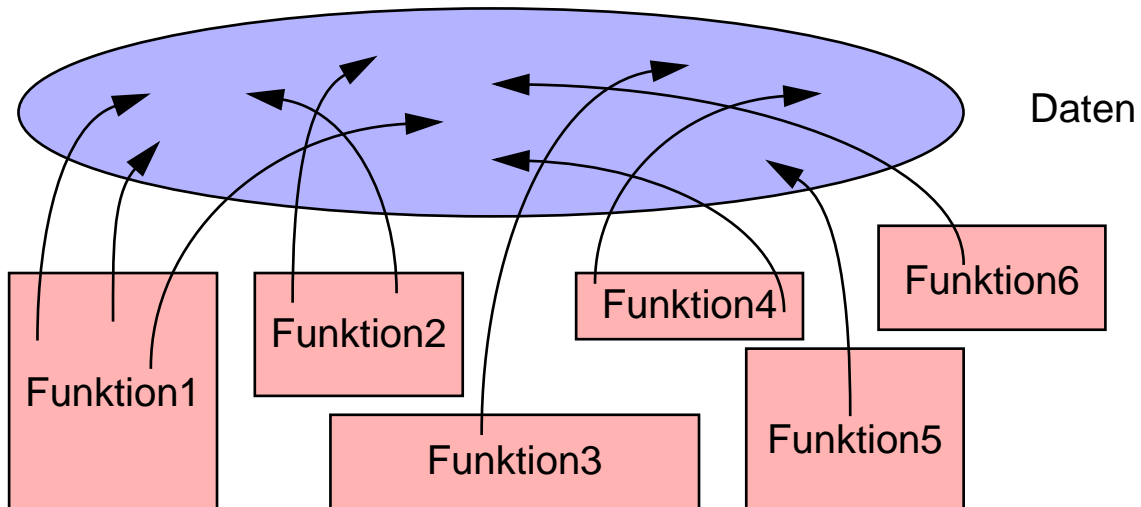
■ Zentrale Fragestellung

◆ was ist zu tun?

◆ in welche Teilaufgaben läßt sich die Aufgabe untergliedern?

- Beispiel: Rechnung für Kunden ausgeben
 - Rechnungspositionen zusammenstellen
 - Lieferungsposten einlesen
 - Preis für Produkt ermitteln
 - Mehrwertsteuer ermitteln
 - Rechnungspositionen addieren
 - Positionen formatiert ausdrucken

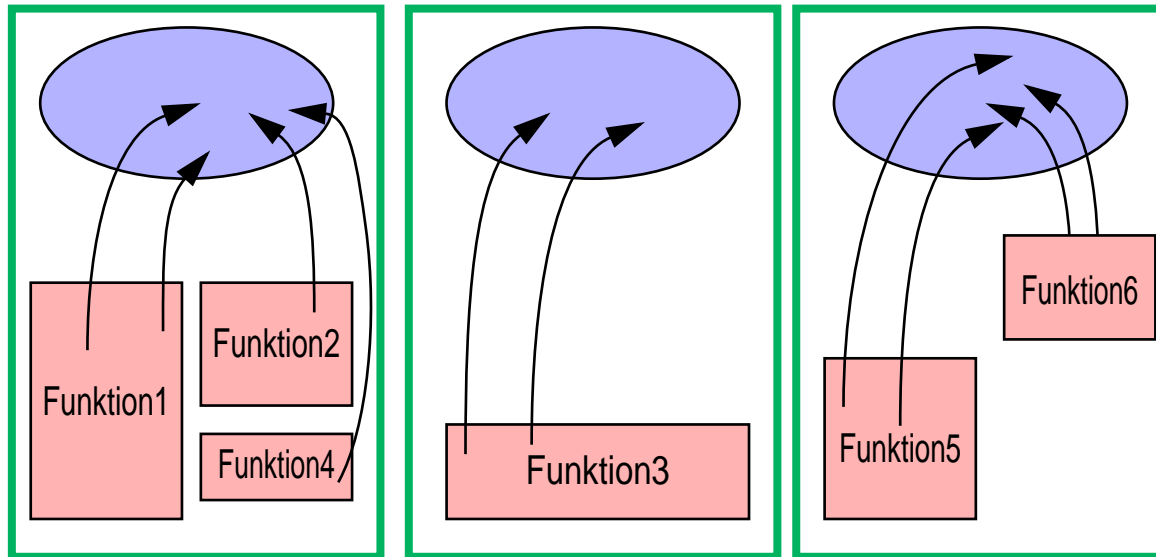
- Problem:
Gliederung betrifft nur die Aktivitäten, nicht die Struktur der Daten
- Gefahr:
Sehr viele Funktionen arbeiten "wild" auf einer Unmenge schlecht strukturierter Daten



2 Top-down Entwurf (3) Modul-Bildung

- Lösung:
Gliederung von Datenbeständen zusammen mit Funktionen, die darauf operieren

➔ **Modul**



- Teile eines C-Programms können auf mehrere `.c`-Dateien (C-Quelldateien) verteilt werden
- Logisch zusammengehörende Daten und die darauf operierenden Funktionen sollten jeweils zusammengefaßt werden
 - ➔ **Modul**
- Jede C-Quelldatei kann separat übersetzt werden (Option `-c`)
 - Zwischenergebnis der Übersetzung wird in einer `.o`-Datei abgelegt

```
% cc -Aa -c main.c           (erzeugt Datei main.o)
% cc -Aa -c f1.c             (erzeugt Datei f1.o)
% cc -Aa -c f2.c f3.c       (erzeugt f2.o und f3.o)
```

- Das Kommando `cc` kann mehrere `.c`-Dateien übersetzen und das Ergebnis — zusammen mit `.o`-Dateien — binden:

```
% cc -Aa -o prog main.o f1.o f2.o f3.o f4.c f5.c
```

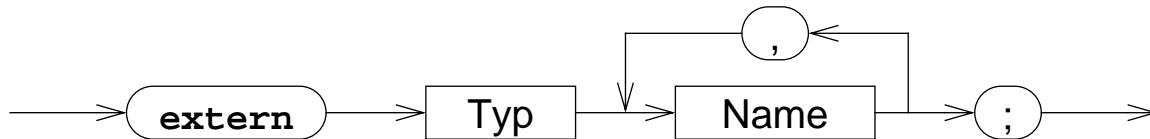
- !!! **.c-Quelldateien auf keinen Fall mit Hilfe der `#include`-Anweisung in andere Quelldateien einkopieren**
- Bevor eine Funktion aus einem anderen Modul aufgerufen werden kann, muß sie **deklariert** werden
 - Parameter und Rückgabewerte müssen bekannt gemacht werden
- Makrodefinitionen und Deklarationen, die in mehreren Quelldateien eines Programms benötigt werden, werden zu **Header-Dateien** zusammengefaßt
 - ◆ *Header-Dateien* werden mit der `#include`-Anweisung des Preprozessors in C-Quelldateien einkopiert
 - ◆ der Name einer *Header-Datei* endet immer auf **.h**

F.10 Gültigkeit von Namen

- Gültigkeitsregeln legen fest, welche Namen (Variablen und Funktionen) wo im Programm bekannt sind
- Mehrere Stufen
 1. Global im gesamten Programm
(über Modul- und Funktionsgrenzen hinweg)
 2. Global in einem Modul
(auch über Funktionsgrenzen hinweg)
 3. Lokal innerhalb einer Funktion
 4. Lokal innerhalb eines Blocks
- Überdeckung bei Namensgleichheit
 - eine lokale Variable innerhalb einer Funktion überdeckt gleichnamige globale Variablen
 - eine lokale Variable innerhalb eines Blocks überdeckt gleichnamige globale Variablen und gleichnamige lokale Variablen in umgebenden Blöcken

1 Gültigkeit im gesamten Programm Globale Variablen

- Globale Variablen werden außerhalb von Funktionen definiert
- Globale Variablen sind ab der Definition in der gesamten Datei zugreifbar
- Globale Variablen, die in anderen Modulen **definiert** wurden, müssen vor dem ersten Zugriff bekanntgemacht werden
(**extern-Deklaration** = Typ und Name bekanntmachen)



- Beispiele:

```
extern int a, b;  
extern char c;
```

1 Gültigkeit im gesamten Programm

Globale Variablen (2)

■ Probleme mit globalen Variablen

- ◆ Zusammenhang zwischen Daten und darauf operierendem Programmcode geht verloren
- ◆ Funktionen können Variablen ändern, ohne daß der Aufrufer dies erwartet (Seiteneffekte)
- ◆ Programme sind schwer zu pflegen, weil bei Änderungen der Variablen erst alle Programmteile, die sie nutzen gesucht werden müssen

➔ **globale Variablen möglichst vermeiden!!!**

1 Gültigkeit im gesamten Programm

Globale Funktionen

- Funktionen sind generell global
(es sei denn, die Erreichbarkeit wird explizit auf das Modul begrenzt)
- Funktionen aus anderen Modulen müssen ebenfalls vor dem ersten Aufruf **deklariert** werden
(= Typ, Name und Parametertypen bekanntmachen)
- Das Schlüsselwort **extern** ist bei einer Funktionsdeklaration nicht notwendig
- Beispiele:

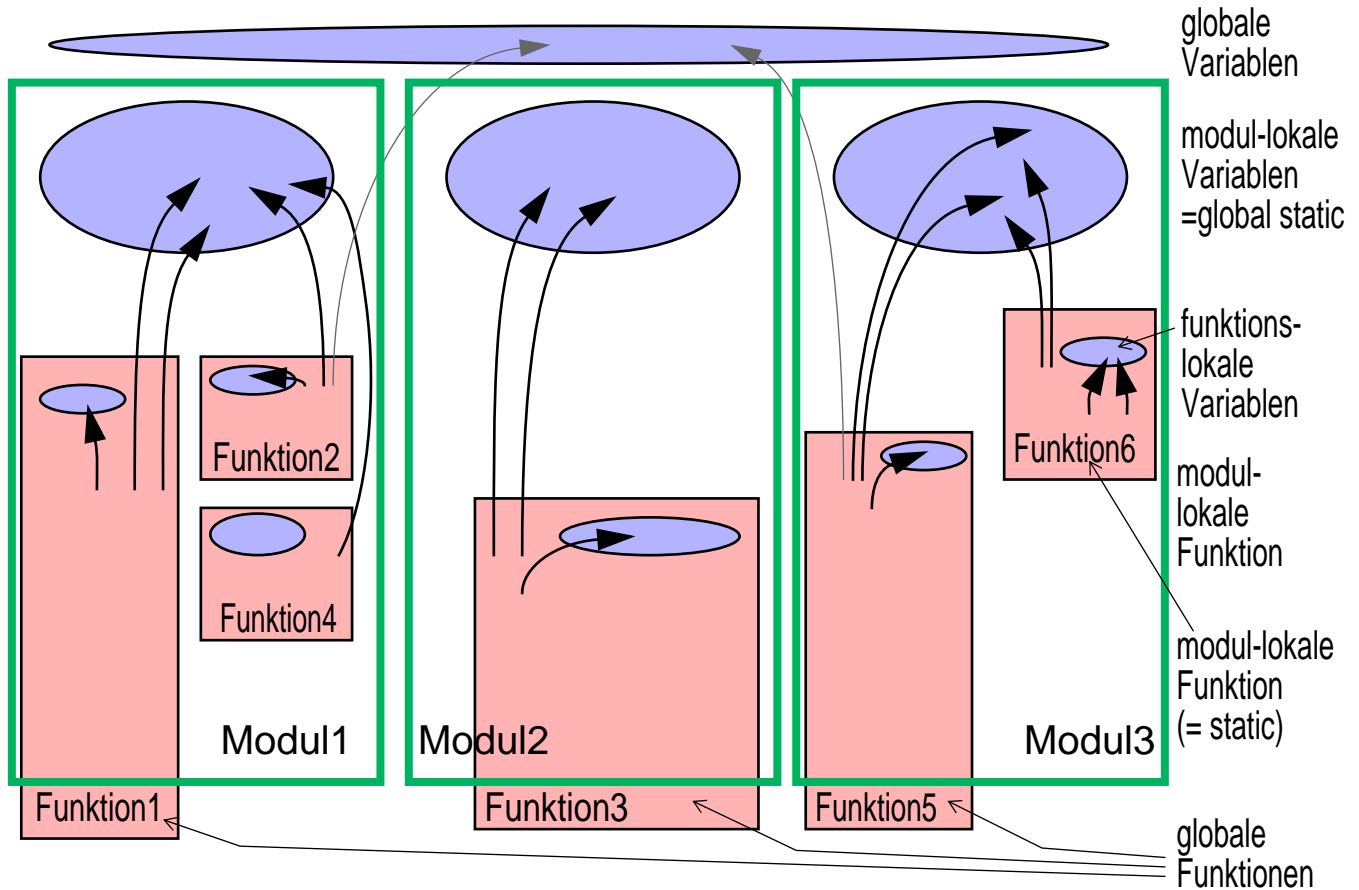
```
double sinus(double);  
float power(float, int);
```
- Globale Funktionen (und soweit vorhanden die globalen Daten) bilden die äußere Schnittstelle eines Moduls
 - "vertragliche" Zusicherung an den Benutzer des Moduls

- Zugriff auf eine globale Variable oder Funktion kann auf das Modul (= die Datei) beschränkt werden, in der sie definiert wurde
 - Schlüsselwort `static` vor die Definition setzen



- `extern`-Deklarationen in anderen Modulen sind nicht möglich
- Die `static`-Variablen bilden zusammen den Zustand eines Moduls, die Funktionen des Moduls operieren auf diesem Zustand
- Hilfsfunktionen innerhalb eines Moduls, die nur von den Modulfunktionen benötigt werden, sollten immer `static` definiert werden
 - sie werden dadurch nicht Bestandteil der Modulschnittstelle (= des "Vertrags" mit den Modulbenutzern)
- !!! das Schlüsselwort `static` gibt es auch bei lokalen Variablen (mit anderer Bedeutung!)

- Variablen, die innerhalb einer Funktion oder eines Blocks definiert werden, sind lokale Variablen
- bei Namensgleichheit zu globalen Variablen oder lokalen Variablen eines umgebenden Blocks gilt die jeweils letzte Definition
- lokale Variablen sind außerhalb des Blocks, in dem sie definiert wurden, nicht zugreifbar und haben dort keinen Einfluß auf die Zugreifbarkeit von Variablen



F.11 Lebensdauer von Variablen

- Die Lebensdauer einer Variablen bestimmt, wie lange der Speicherplatz für die Variable aufgehoben wird
- Zwei Arten
 - ◆ Speicherplatz bleibt für die gesamte Programmausführungszeit reserviert
 - statische (`static`) Variablen
 - ◆ Speicherplatz wird bei Betreten eines Blocks reserviert und danach wieder freigegeben
 - dynamische (`automatic`) Variablen

- Alle lokalen Variablen sind automatic-Variablen
 - der Speicher wird bei Betreten des Blocks / der Funktion reserviert und bei Verlassen wieder freigegeben
 - ➔ der Wert einer lokalen Variablen ist beim nächsten Betreten des Blocks nicht mehr sicher verfügbar!
- Lokale auto-Variablen können durch beliebige Ausdrücke initialisiert werden
 - die Initialisierung wird bei jedem Eintritt in den Block wiederholt
 - !!! wird eine auto-Variable nicht initialisiert, ist ihr Wert vor der ersten Zuweisung undefiniert (= irgendwas)**

- Der Speicher für alle globalen Variablen ist generell von Programmstart bis Programmende reserviert
- Lokale Variablen erhalten bei Definition mit dem Schlüsselwort `static` eine **Lebensdauer über die gesamte Programmausführung** hinweg
 - ➔ der Inhalt bleibt bei Verlassen des Blocks erhalten und ist bei einem erneuten Eintreten in den Block noch verfügbar
- !!! Das Schlüsselwort `static` hat bei globalen Variablen eine völlig andere Bedeutung (Einschränkung des Zugriffs auf das Modul)
- Static-Variablen können durch beliebige konstante Ausdrücke initialisiert werden
 - die Initialisierung wird nur einmal beim Programmstart vorgenommen (auch bei lokalen Variablen!)
 - wird eine static-Variable nicht explizit initialisiert, ist sie automatisch mit 0 vorbelegt

F.12 Wertaustausch zwischen Funktionen

Mechanismus	Aufrufer → Funktion	Funktion → Aufrufer
Parameter	ja	mit Hilfe von Zeigern
Funktionswert	nein	ja
globale Variablen	ja	ja

■ Verwendung globaler Variablen?

- ◆ Variablen, die von vielen Funktionen verwendet werden und/oder oft als Parameter übergeben werden müßten
 - Menge der Funktionen muß überschaubar bleiben
→ Zugriff auf Modul begrenzen (globale static-Variablen)
 - **sonst sehr schlechter Programmierstil**
- ◆ Variablen, die keiner Funktion als Variable oder Parameter fest zugeordnet werden können
 - Modul suchen, dem die Variable zugeordnet werden kann!!!
- ◆ Variablen, deren Lebensdauer nicht beschränkt sein darf, die aber nicht in `main()` deklariert werden sollen
 - in zugehöriger Funktion lokal-static definieren

F.13 Getrennte Übersetzung von Programmteilen — Beispiel

■ Hauptprogramm (Datei `fplot.c`)

```
#include "trig.h"
#define INTERVALL 0.01

/*
 * Funktionswerte ausgeben
 */
int main(void)
{
    char c;
    double i;

    printf("Funktion (Sin, Cos, Tan, cOt)? ");
    scanf("%x", &c);

    switch (c) {
        ...
        case 'T':
            for (i=-PI/2; i < PI/2; i+=INTERVALL)
                printf("%lf %lf\n", i, tan(i));
            break;;
        ...
    }
}
```

■ *Header-Datei* (Datei `trig.h`)

```
#include <stdio.h>
#define PI 3.1415926535897932
double tan(double), cot(double);
double cos(double), sin(double);
```

■ Trigonometrische Funktionen
(Datei `trigfunc.c`)

```
#include "trig.h"

double tan(double x) {
    return(sin(x)/cos(x));
}

double cot(double x) {
    return(cos(x)/sin(x));
}

double cos(double x) {
    return(sin(PI/2-x));
}
```

...

■ Trigonometrische Funktionen — Fortsetzung
(Datei `trigfunc.c`)

...

```
double sin (double x)
{
    double summe;
    double x_quadrat;
    double rest;
    int k;

    k = 0;
    summe = 0.0;
    rest = x;
    x_quadrat = x*x;

    while ( fabs(rest) > 1e-9 ) {
        summe += rest;
        k += 2;
        rest *= -x_quadrat/(k*(k+1));
    }
    return(summe);
}
```

G Felder und Strukturen

G.1 Eindimensionale Felder

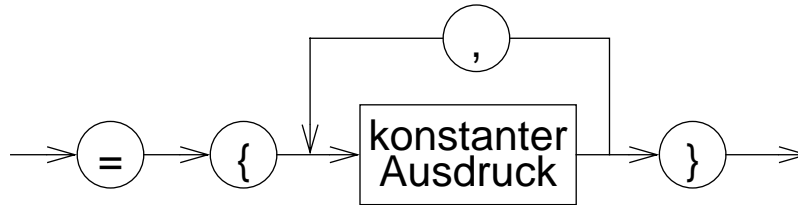
- eine Reihe von Daten desselben Typs kann zu einem **Feld** zusammengefaßt werden
- bei der Definition wird die Anzahl der Feldelemente angegeben, die Anzahl ist konstant!
- der Zugriff auf die Elemente erfolgt durch **Indizierung**, beginnend bei Null
- Definition eines Feldes



- Beispiele:

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

G.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

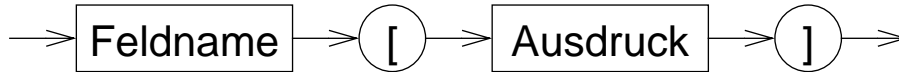
G.2 ... Initialisierung eines Feldes (2)

- Felder der Speicherklasse *automatic* (lokale Felder) können in altem K&R-C nicht initialisiert werden
ANSI-C erlaubt die Initialisierung von Feldern
- Felder des Typs ***char*** können auch durch String-Konstanten initialisiert werden

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

G.3 Zugriffe auf Feldelemente

- Indizierung:



wobei: $0 \leq \text{Wert}(\text{Ausdruck}) < \text{Feldgröße}$

- Beispiele:

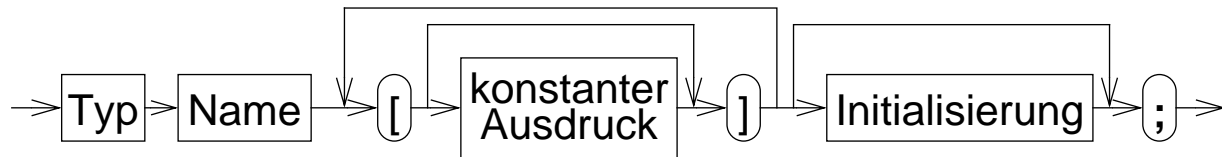
```
prim[0] == '2'  
prim[1] == '3'  
name[1] == 't'  
name[4] == '\0'
```

- 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]);
```

G.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];
```

■ 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;
```

2 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!

G.5 Eindimensionale Felder als Funktionsparameter

- ganze Felder können in C **nicht *by-value*** übergeben werden
- wird einer Funktion der Feldname als Parameter übergeben, kann sie 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
 - 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
- wird ein Feldparameter als `const` deklariert, können die Feldelemente innerhalb der Funktion nicht verändert werden (ANSI)

■ Bestimmung der Länge einer Zeichenkette (*String*)

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

■ 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" */
```

1 Motivation

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

- Beispiel: Daten eines Studenten

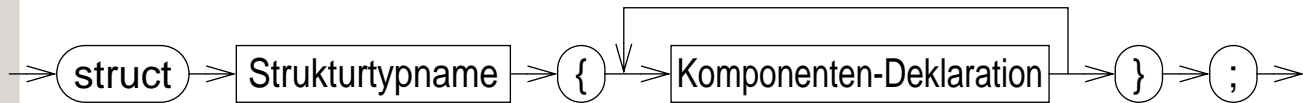
– Nachname	<code>char nachname[25];</code>
– Vorname	<code>char vorname[25];</code>
– Geburtsdatum	<code>char gebdatum[11];</code>
– Matrikelnummer	<code>int matrnr;</code>
– Übungsgruppennummer	<code>short gruppe;</code>
– Schein bestanden	<code>char best;</code>

- Möglichkeiten der Repräsentation in einem Programm

- ◆ einzelne Variablen → sehr umständlich, keine "Abstraktion"
- ◆ Datenstrukturen

- Durch eine Strukturtyp-Deklaration wird dem Compiler der Aufbau einer Datenstruktur unter einem Namen bekanntgemacht
 - ➔ deklariert einen neuen Datentyp (wie `int` oder `float`)

- Syntax



- **Strukturtypname**

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

- **Komponenten-Deklaration**

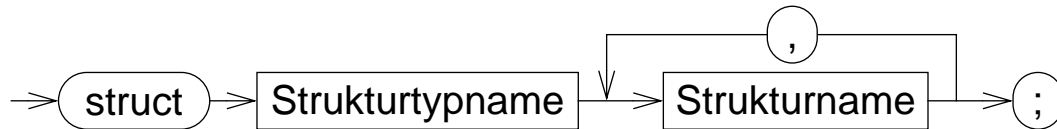
- ◆ entspricht normaler Variablen-Definition, aber keine Initialisierung!
- ◆ in verschiedenen Strukturen dürfen die gleichen Komponentennamen verwendet werden (eigener Namensraum pro Strukturtyp)

■ Beispiele

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

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

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



- Beispiele

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

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

■ .-Operator

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

■ Beispiele

```
struct komplex c1, c2, c3;
...
c3.re = c1.re + c2.re;
c3.im = c1.im + c2.im;

struct student stud1;
...
if (stud1.matrnr < 1500000) {
    stud1.best = 'y';
}
```

- 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

- 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);  
}
```

- 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';
    }
}
```

- Die Komponenten einer Struktur können wieder Strukturen sein
- Beispiel

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

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

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

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

■ Funktionsparameter

- ◆ alte C-Compiler erlauben die Übergabe ganzer Strukturen als Funktionsparameter oder -rückgabewert nicht
 - ▶ dann nur Zeiger auf Strukturen möglich

■ Zuweisungen

- ◆ moderne C-Compiler erlauben die Zuweisung kompletter Strukturen
- ◆ Beispiel

```
struct komplex c1, c2;  
c2 = c1;
```

- ◆ wenn der Compiler dies nicht erlaubt, komponentenweise zuweisen

```
c2.re = c1.re; c2.im = c1.im;
```


■ Namen von Strukturkomponenten

- ◆ alte C-Compiler erlauben nicht gleiche Komponentennamen in unterschiedlichen Strukturen

H Zeiger(-Variablen)

H.1 Einordnung

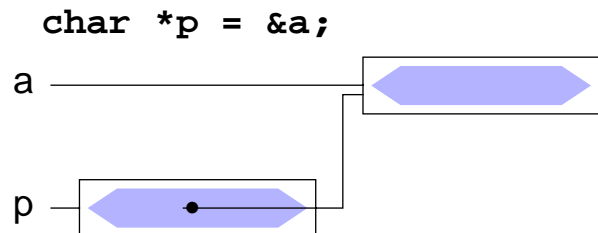
- **Konstante:**
Bezeichnung für einen Wert

'a' ≡ 

- **Variable:**
Bezeichnung einer Speicherstelle,
die einen Wert enthält



- **Zeiger-Variable (Pointer):**
Bezeichnung einer Speicherstelle,
die als Wert die Adresse einer
Speicherstelle enthält,
die dann schließlich einen Wert
enthält



H.2 Überblick

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

1 Definition von Zeigervariablen

■ Syntax:

```
Typ *Name ;
```

2 Beispiele

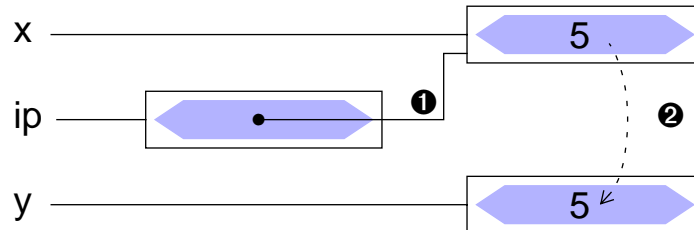
```
int x = 5;
```

```
int *ip;
```

```
int y;
```

```
ip = &x; ①
```

```
y = *ip; ②
```



H.3 Adreßoperatoren

1 Adreßoperator &

&x der unäre Adreß-Operator liefert die Adresse
der Variablen (des Objekts) **x**

2 Verweisoperator *

x** der unäre Verweisoperator ** ermöglicht den Zugriff auf den
Inhalt der Variablen (des Objekts), auf die der Zeiger **x** verweist

3 Der Operator []

x[y] **x** ist ein Zeiger auf das erste einer Reihe von Objekten
(z. B. ein Feld)
 y ist ein Index (*offset*), der bestimmt, auf welches der Objekte
 zugegriffen wird
 – der Zugriff erfolgt durch eine implizite Addition,
 bei der die Größe des Objekts berücksichtigt wird

H.4 Zeiger als Funktionsargumente

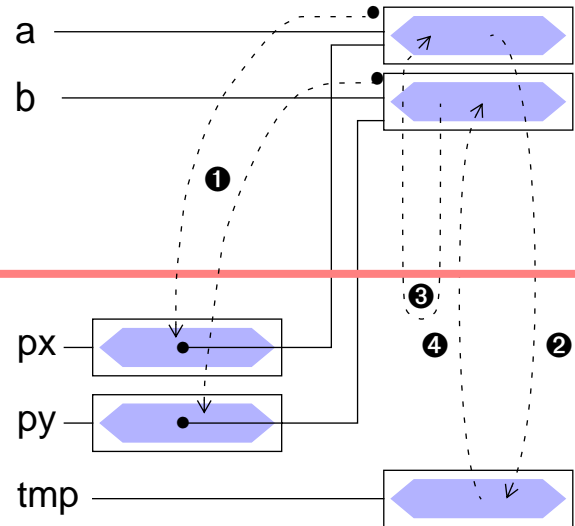
- Parameter werden in C *by-value* übergeben
- die aufgerufene Funktion kann den aktuellen Parameter beim Aufrufer nicht verändern
- auch Zeiger werden *by-value* übergeben, d. h. die Funktion erhält lediglich eine Kopie des Adreßverweises
- über diesen Verweis kann die Funktion jedoch mit Hilfe des *-Operators auf die zugehörige Variable zugreifen und sie verändern
↳ *call-by-reference*

H.4 ... Zeiger als Funktionsargumente (2)

■ Beispiel:

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

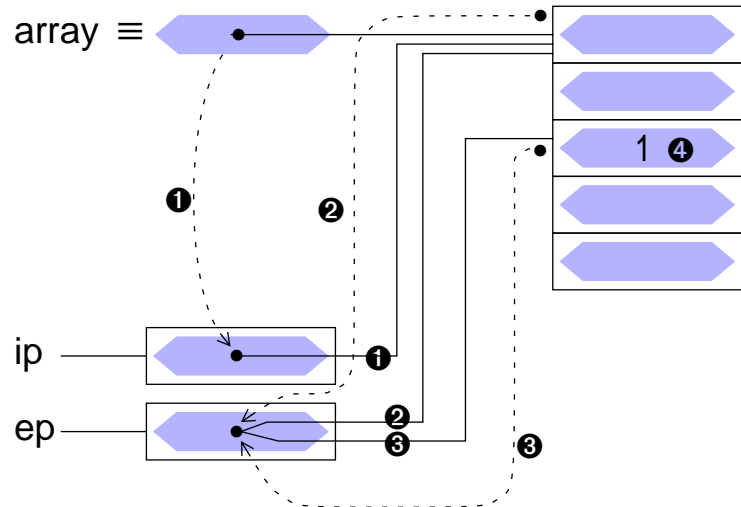
```
void swap (int *px, int *py)  
{  
  int tmp;  
  
  tmp = *px; ②  
  *px = *py; ③  
  *py = tmp; ④  
  
}
```



H.5 Zeiger und Felder (1)

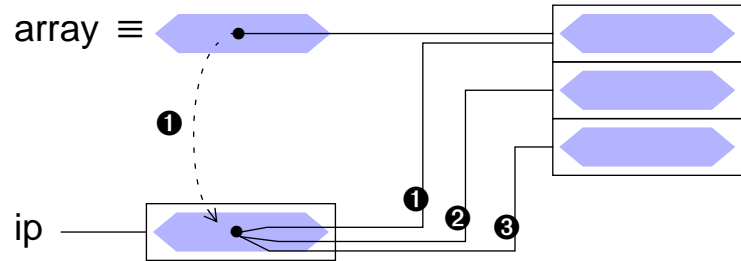
- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

```
int array[5];  
  
int *ip = array; ①  
  
int *ep;  
ep = &array[0]; ②  
  
ep = &array[2]; ③  
  
*ep = 1; ④
```



■ ++ -Operator: Inkrement = nächstes Objekt

```
int array[3];  
int *ip = array; ❶  
  
ip++; ❷  
ip++; ❸
```

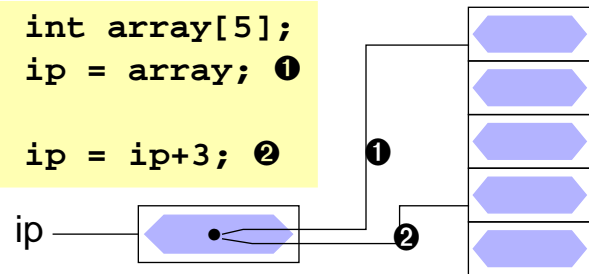


■ -- -Operator: Dekrement = vorheriges Objekt

■ +, -
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 !!

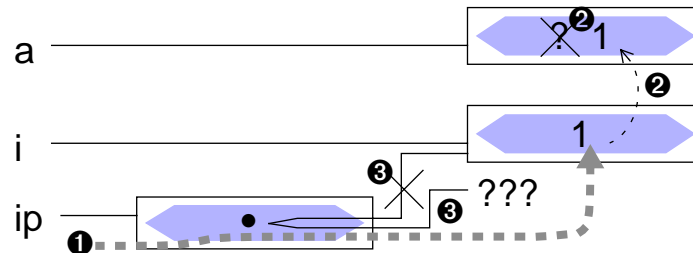
1 Vorrangregeln bei Operatoren

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

2 Beispiele

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

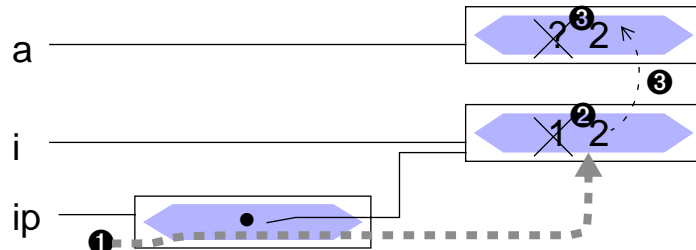
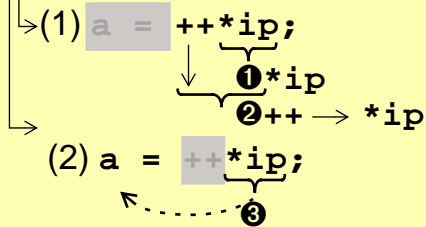
a = *ip++;
  ↳ (1) a = *ip++;
     ↳ (2) a = *ip++;
     ↳ (3) a = *ip++;
```



2 Beispiele (2)

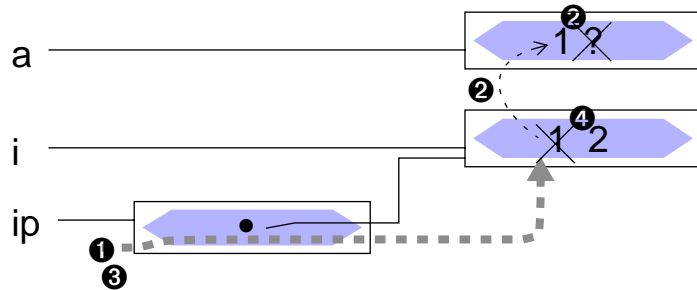
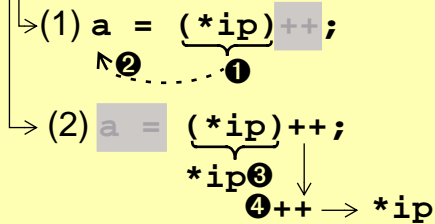
```
int a, i, *ip;
i = 1;
ip = &i;
```

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



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

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



3 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;

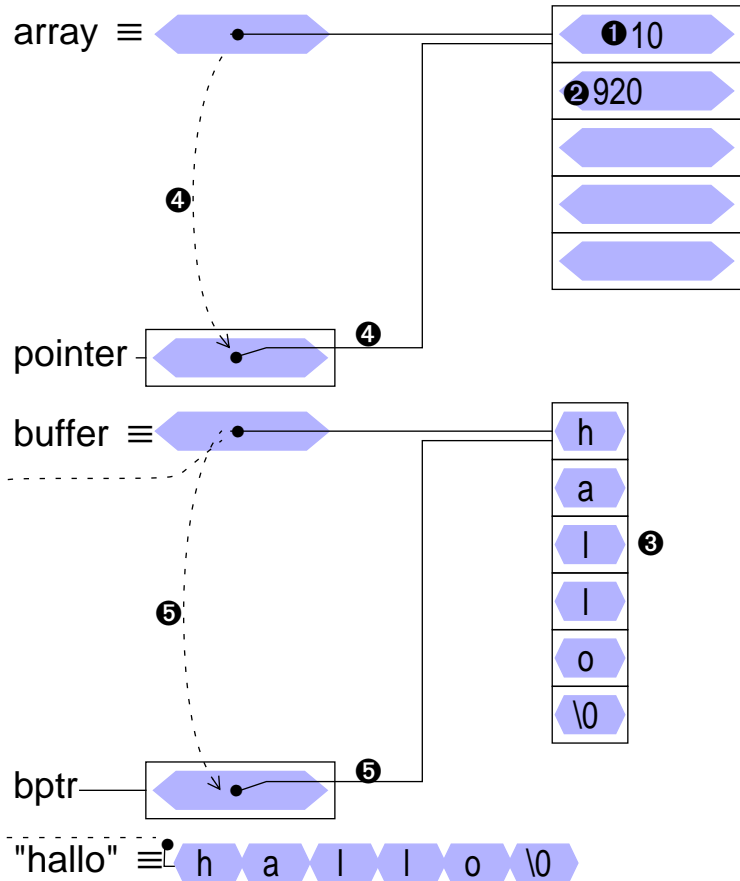
/* Folgende Zuweisungen sind äquivalent */
array[i] = 1;
ip[i] = 1;
*(ip+i) = 1;      /* Vorrang! */
*(array+i) = 1;

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

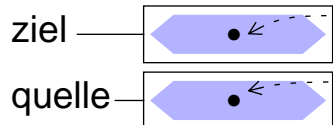
3 Zeigerarithmetik und Felder

```
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;



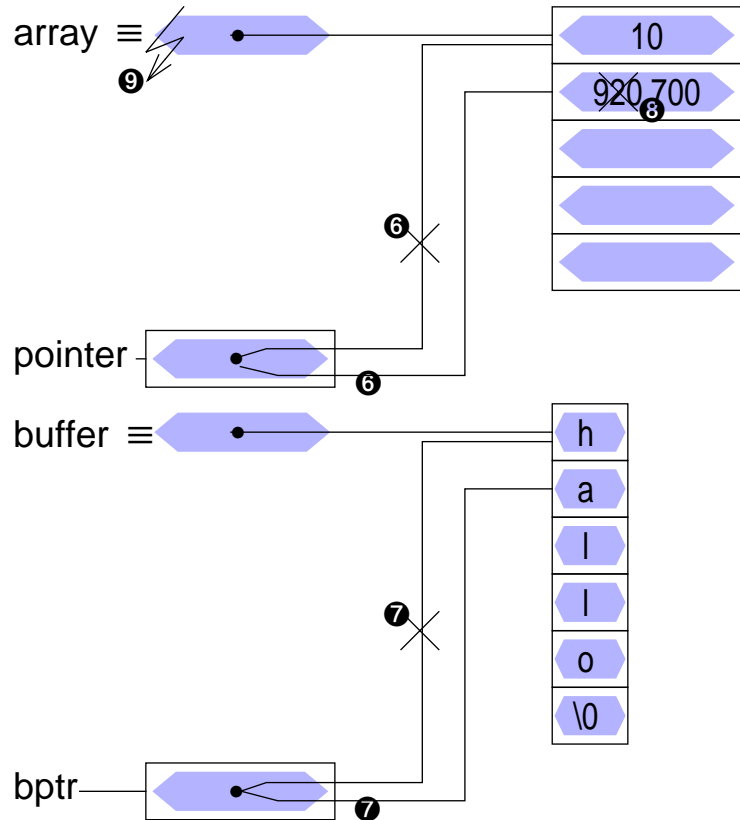
Fomale Parameter
der Funktion strcpy



3 Zeigerarithmetik und Felder

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

```
① array[0] = 10;  
② array[1] = 920;  
③ strcpy(buffer, "hallo");  
④ pointer = array;  
⑤ bptr = buffer;  
  
⑥ pointer++;  
⑦ bptr++;  
⑧ *pointer = 700;  
  
⑨ array++;
```



4 Vergleichsoperatoren und Adressen H.6 Arithmetik mit Adressen

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

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

H.7 Felder als Funktionsparameter

- 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);
```

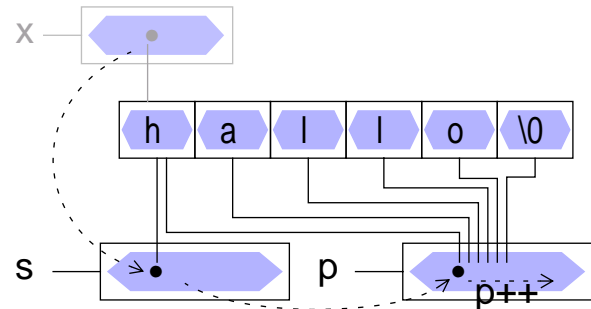
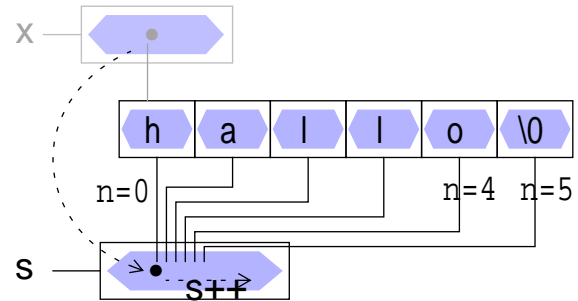
- Der Feldname ist eine Konstante Adresse, die als Argument eines Funktionsaufrufs *by-value* übergeben wird (= Funktion erhält Kopie!)
- Diese Kopie kann von der aufgerufenen Funktion – unabhängig von der Deklaration des formalen Parameters – wie eine Zeigervariable verwendet werden
- die Parameter-Deklarationen `int p2[]` und `int *p2` sind vollkommen äquivalent!

H.8 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);
}
```




H.8 ... 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";
```

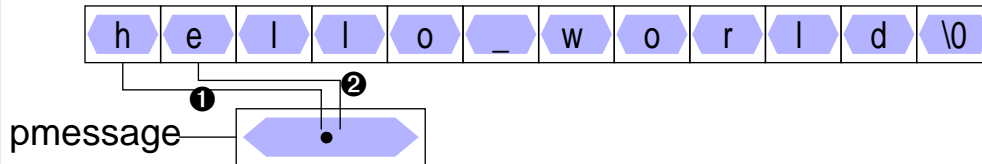


amessage ≡ 

H.8 ... Zeiger, Felder und Zeichenketten (3)

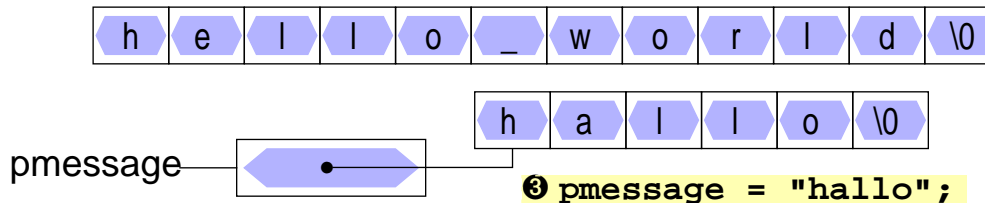
- wird eine Zeichenkette zur Initialisierung eines `char`-Zeigers verwendet, ist der Zeiger eine Variable, die mit der Anfangsadresse der Zeichenkette initialisiert wird

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



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

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

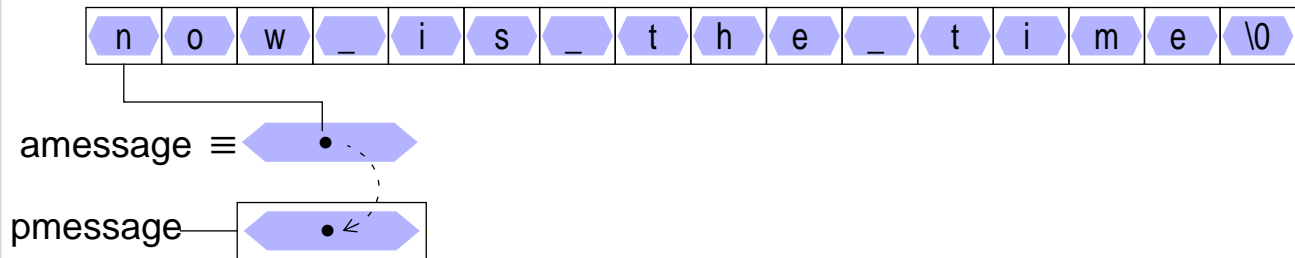


H.8 ... Zeiger, Felder und Zeichenketten (4)

- 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

H.8 ... Zeiger, Felder und Zeichenketten (5)

■ 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++ )
        ;
}
```

H.8 ... Zeiger, Felder und Zeichenketten (6)

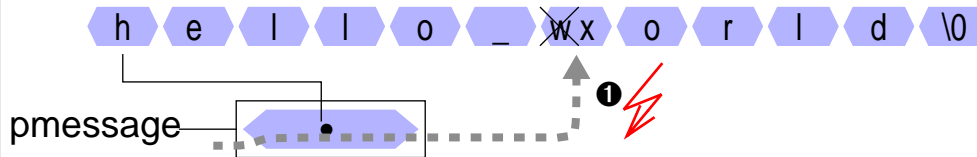
- in ANSI-C können Zeichenketten in nicht-modifizierbaren Speicherbereichen angelegt werden (je nach Compiler)

↳ 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!

H.9 Beispiel: dynamische Speicherverwaltung

- die folgenden zwei Funktionen verwalten einen globalen, zusammenhängenden Speicherbereich,
 - aus dem freier Speicher angefordert werden kann (***alloc***)
 - und in den freigegebener Speicher wieder integriert wird (***free***)
- diese Version einer dynamischen Speicherverwaltung ist rudimentär und soll lediglich einen Eindruck von einer möglichen Realisierung geben
- in der Regel werden für diese Aufgabenstellung die Funktionen ***malloc()*** und ***free()*** aus der C-Bibliothek verwendet

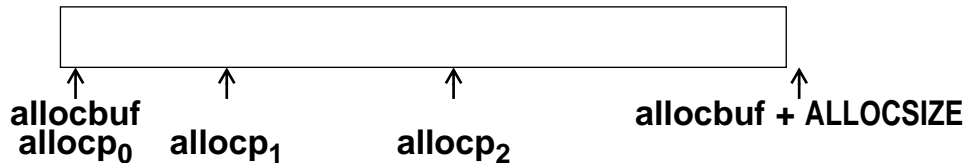
H.9 ... Beispiel: dynamische Speicherverwaltung (2)

■ Globale Definitionen

```
#define NULL      (char *)0
#define ALLOCSIZE 1000
static char allocbuf[ALLOCSIZE]
static char *allocp = allocbuf;
```

■ Anfordern von Speicher

```
char *alloc(int n)
{
    if (allocp+n <= allocbuf+ALLOCSIZE) {
        allocp += n;
        return (allocp - n);
    } else
        return (NULL);
}
```



H.9 ... Beispiel: dynamische Speicherverwaltung (3)

■ Freigabe von Speicher

```
void free(char *p)
{
    if (p >= allocbuf &&
        p < allocbuf+ALLOCSIZE)
        allocp = p;
}
```

H.10 Felder von Zeigern

■ Auch von Zeigern können Felder gebildet werden

■ Deklaration

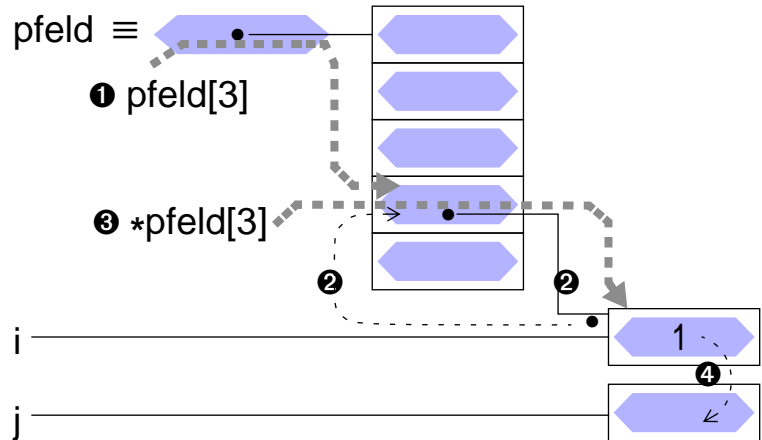
```
int *pfeld[5];  
int i = 1  
int j;
```

■ Zugriffe auf einen Zeiger des Feldes

```
pfeld[3] = &i; ②
```

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

```
j = *pfeld[3]; ④
```

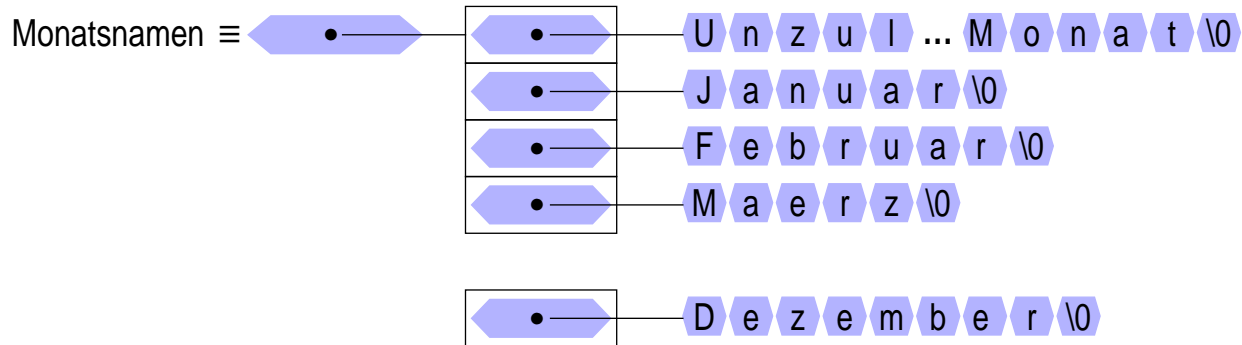


H.10 Felder von Zeigern (2)

- 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] );
}
```



H.11 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[])
{
    ...
}
```

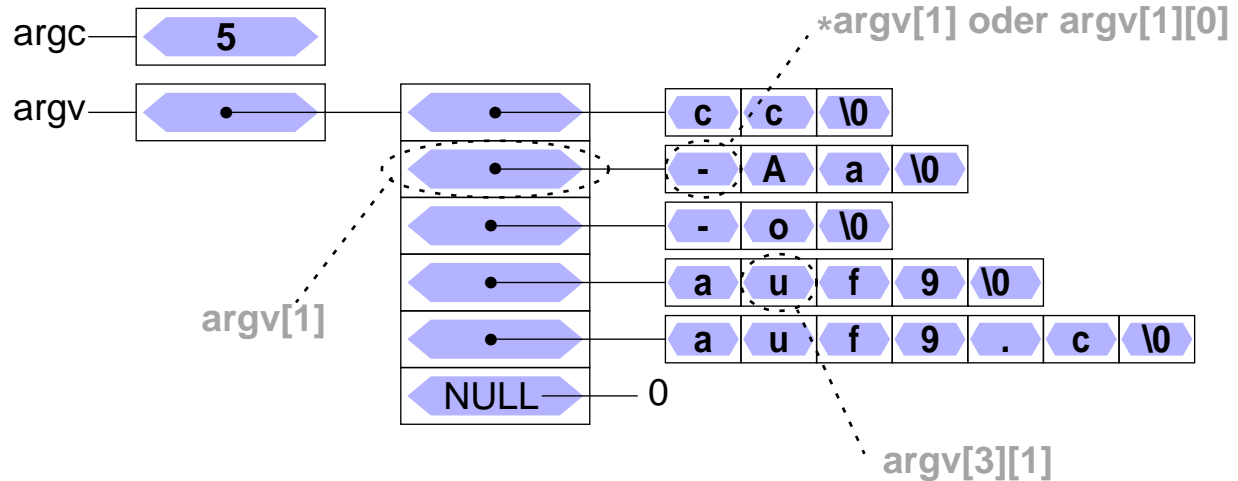
oder

```
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]`)

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

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



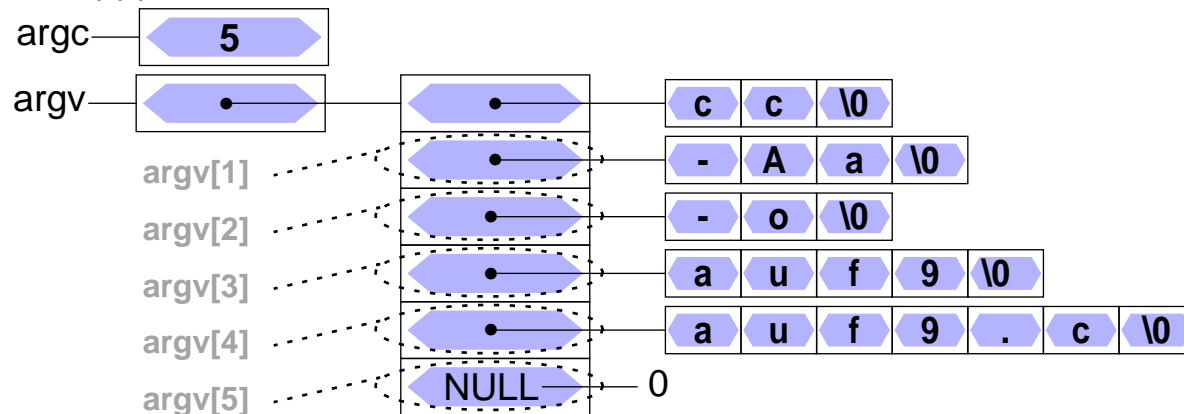
2 Zugriff

Beispiel: Ausgeben aller Argumente (1)

- das folgende Programmstück gibt alle Argumente der Kommandozeile aus (außer dem Kommandonamen)

```
int  
main (int argc, char *argv[])  
{  
    int i;  
    for ( i=1; i<argc; i++) {  
        printf("%s%c", argv[i],  
            (i < argc-1) ? ' ':'\n' );  
    }  
    ...  
}
```

1. Version



2 Zugriff

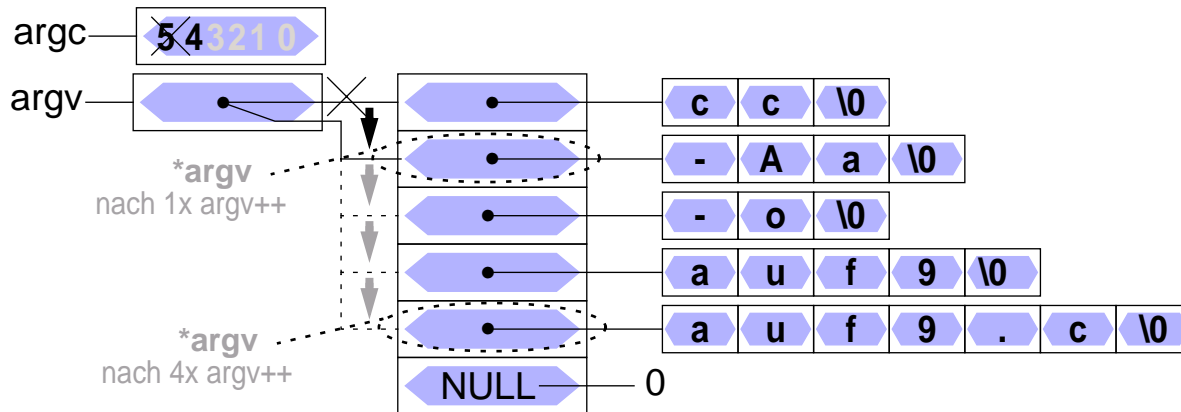
Beispiel: Ausgeben aller Argumente (2)

- das folgende Programmstück gibt alle Argumente der Kommandozeile aus (außer dem Kommandonamen)

```
int  
main (int argc, char **argv)  
{  
    while (--argc > 0) {  
        argv++;  
        printf("%s%c", *argv, (argc>1) ? ' ' : '\n' );  
    }  
    ...  
}
```

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

2. Version

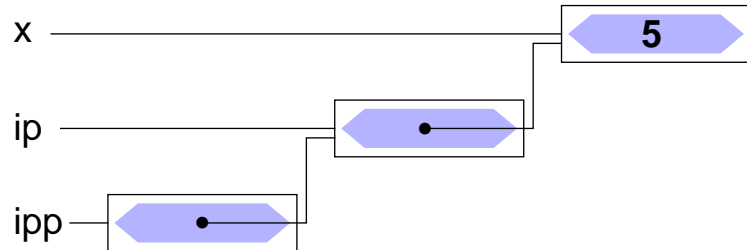


H.12 Zeiger auf Zeiger

- 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)

H.13 sizeof-Operator

- In manchen Fällen ist es notwendig, die Größe (in Byte) einer Variablen oder Struktur zu ermitteln
 - z. B. zum Anfordern von Speicher für ein Feld (→ malloc)

- Syntax:

`sizeof x` liefert die Größe des Objekts `x` in Bytes

`sizeof (Typ)` liefert die Größe eines Objekts vom Typ `Typ` in Bytes

- Das Ergebnis ist vom Typ `size_t` (\equiv `int`)
(`#include <stddef.h>!`)

- Beispiel:

```
int a; size_t b;  
b = sizeof a;          /*  $\Rightarrow$  b = 2 oder b = 4 */  
b = sizeof(double);   /*  $\Rightarrow$  b = 8 */
```

H.14 Explizite Typumwandlung — Cast-Operator

- C enthält Regeln für eine automatische Konvertierung unterschiedlicher Typen in einem Ausdruck (vgl. Abschnitt D.11)

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:

```
char *malloc(int);      /* Funktion zum dynamischen Anfordern von Speicher */
int *array;             /* Zeiger auf Anfang eines dyn. anzufordernden Feldes */
int i, n = 20;

array = (int *)malloc(n * sizeof(int)); /* Feld mit n Integer-Werten */
/* dynamisch anfordern */
for (i=0; i<n; i++) array[i] = 1; /* alle Feld-Werte auf 1 setzen */
```

H.15 Zeiger auf Strukturen

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

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

- Besondere Bedeutung zum Aufbau
 - ➔ rekursiver Strukturen

H.15 Zeiger auf Strukturen (2)

- Zugriff auf Strukturkomponenten über einen Zeiger
- Bekannte Vorgehensweise
 - *-Operator liefert die Struktur
 - .-Operator zum Zugriff auf Komponente
 - Operatorenvorrang beachten

↳ `(*pstud).best = 'n';`

unleserlich!

- Syntaktische Verschönerung

↳ `->-Operator`

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

H.16 Rekursive Strukturen

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

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

falsch!

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

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

➔ Programmieren rekursiver Datenstrukturen

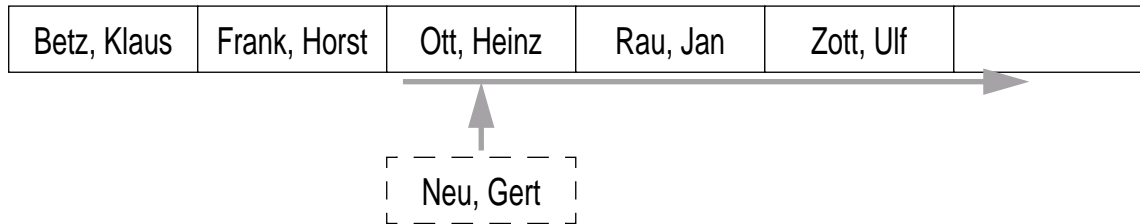
H.16 Rekursive Strukturen (2)

■ Problem:

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

Lösung 1: Feld

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



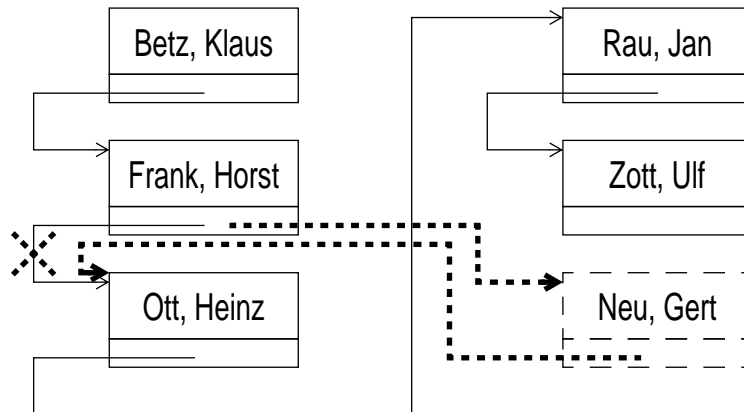
H.16 Rekursive Strukturen (3)

■ Problem:

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

Lösung 2: verkettete Liste von dynamisch angeforderten Strukturen

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



H.16 Rekursive Strukturen (4)

■ Realisierung von Lösung 2 (Skizze):

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

int eintrag_lesen(struct student *);

/* erstes Listen-Element anfordern */
akt_eintrag = (struct eintrag *)malloc(sizeof (struct eintrag));
stud_liste = &leer; /* Listenanfang auf leeres Element setzen (dadurch
                    Vermeidung von Sonderbehandlung für Listenanfang) */

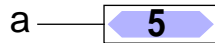
while (eintrag_lesen(&akt_eintrag->stud) != EOF ) {
    einfuege_pos = ... /* Eintrag, hinter dem einzufügen ist irgendwie suchen */
    akt_eintrag->naechster = einfuege_pos->naechster;
    einfuege_pos->naechster = akt_eintrag;

    /* neues Listen-Element anfordern */
    akt_eintrag = (struct eintrag *)malloc(sizeof (struct eintrag));
}
```

H.17 Zusammenfassung

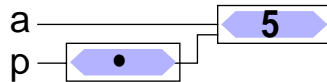
■ Variable

```
int a;
```



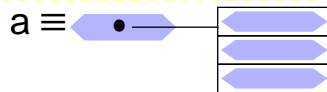
■ Zeiger

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



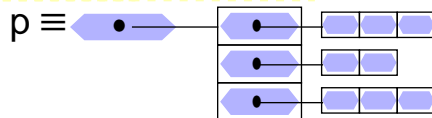
■ Feld

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



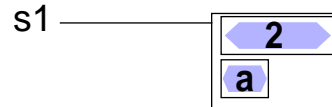
■ Feld von Zeigern

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



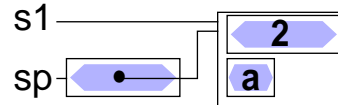
■ Struktur

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



■ Zeiger auf Struktur

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



■ Feld von Strukturen

```
sa ≡
```

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



I Ein-/Ausgabe

I.1 Überblick

- 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

■ 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

I.2 Standard Ein-/Ausgabe (2)

■ 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!

I.3 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`

- Prototyp:

```
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

I.3 Öffnen und Schließen von Dateien (2)

■ Beispiel:

```
#include <stdio.h>

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`

I.4 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` übergeben Zeichen
- geben gleichzeitig das geschriebene Zeichen als Ergebnis zurück

1.4 Zeichenweise Lesen und Schreiben (2)

■ Beispiel: copy-Programm

Aufruf: `copy Quelldatei Zieldatei`

```
#include <stdio.h>
```

Teil 1: Aufrufargumente
auswerten

```
main(int argc, char *argv[]) {  
    FILE *quelle;  
    FILE *ziel;  
    int c;                /* gerade kopiertes Zeichen */  
  
    if (argc < 3) {  
        fprintf(stderr, "Aufruf: copy Quelle Ziel");  
        exit(1);        /* Programm abbrechen */  
    }  
  
    if ((quelle = fopen(argv[1], "r")) == NULL) {  
        perror(argv[1]); /* Fehlermeldung ausgeben */  
        exit(1);        /* Programm abbrechen */  
    }  
  
    if ((ziel = fopen(argv[2], "w")) == NULL) {  
        perror(argv[2]); /* Fehlermeldung ausgeben */  
        exit(1);        /* Programm abbrechen */  
    }  
  
    /* ... */  
}
```

I.4 Zeichenweise Lesen und Schreiben (2)

... Beispiel: copy-Programm
— Fortsetzung

```
/* ... */  
  
while ( (c = getc(quelle)) != EOF ) {  
    putc(c, ziel);  
}  
  
fclose(quelle);  
fclose(ziel);  
}
```

Teil 2: kopieren

I.5 Zeilenweise Lesen und Schreiben

■ Lesen einer Zeile von einem Dateikanal

```
char *gets(char *s)
```

- liest Zeichen von der Standardeingabe in das Feld `s` bis entweder `'\n'` oder `EOF` gelesen wurde
- `'\n'` wird entfernt und `s` wird mit `'\0'` abgeschlossen
- gibt bei `EOF` oder Fehler `NULL` zurück, sonst `s`

!!! die Länge des Feldes `s` wird nicht überprüft

- ➔ wenn bis zu einem `'\n'` zu viele Zeichen kommen, wird unkontrolliert Speicher überschrieben
 - besonders gefährlich, wenn die Standardeingabe auf eine Internetverbindung umgelenkt ist
 - durch trickreiche Eingabe kann Programm mißbraucht werden
 - Angriffspunkt für Hacker

!!! Funktion `gets()` NICHT BENUTZEN

I.5 Zeilenweise Lesen und Schreiben (2)

■ 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

1.5 Zeilenweise Lesen und Schreiben (3)

... Beispiel: copy-Programm aus 1.4 (erster Teil identisch)
— Fortsetzung mit zeilenweise Kopieren

```
/* ... */ Teil 2: kopieren  
  
{ /* neuen Block beginnen, um Feld definieren zu können */  
  
#define BUFSIZE 1024 /* Größe des Zeilenpufferspeichers */  
  
    char buffer[BUFSIZE]; /* Feld für Zeilenpuffer anlegen */  
  
    while ( fgets(buffer, BUFSIZE, quelle) != EOF ) {  
        fputs(buffer, ziel);  
    }  
  
    fclose(quelle);  
    fclose(ziel);  
}
```

1.6 Kopieren beliebiger Blöcke

■ Funktionen `fread` und `fwrite` (bei Bedarf siehe Manual)

I.7 Formatierte Ausgabe

1 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
-> potentielle Sicherheitsprobleme!)
 - `snprintf` arbeitet analog, schreibt aber maximal nur `n` Zeichen
(`n` sollte natürlich nicht größer als die Feldgröße sein)

- 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` oder `double` Parameter wird als Fließkommazahl (z. B. 271.456789) ausgegeben

`%e` `float` oder `double` 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

1 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

- *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

`%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 %2d %3d %5c %s %d",  
          &a, &b, &c, s1, s2, &d);
```

Eingabe: 12 1234567 sowas hmm

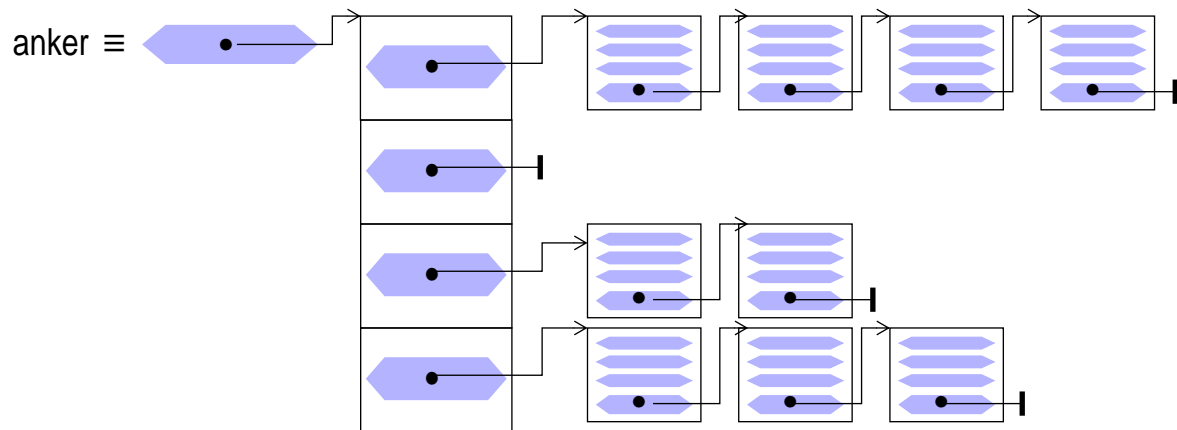
Ergebnis: n=5, a=12, b=12, c=345

s1="67 soX", s2="was"

J Verschiedenes

J.1 Hashing

- Feld von Anker für verkettete Listen
- Listenelemente werden abhängig von ihrem Inhalt in eine der Listen eingetragen



J.1 Hashing (2)

- Index in das Anker-Feld wird aus dem Inhalt des einzutragenden Listenelements ermittelt
 - ◆ Kriterium, nach dem Listenelemente hinterher gesucht werden, nutzen
 - erster Buchstabe des Nachnamens
 - Geburtsdatum
 - Matrikelnummer
 - Übungsgruppe
- Eintragen von Listenelementen
 - einfach am Anfang der entsprechenden Liste eintragen
- Suchen von Listenelementen
 - entsprechende Liste durchlaufen und Elemente einzeln abprüfen

J.1 Hashing (3)

■ Beispiel

- Feld mit 32 Anker
- Code an Codeskizze aus H.16 angelehnt

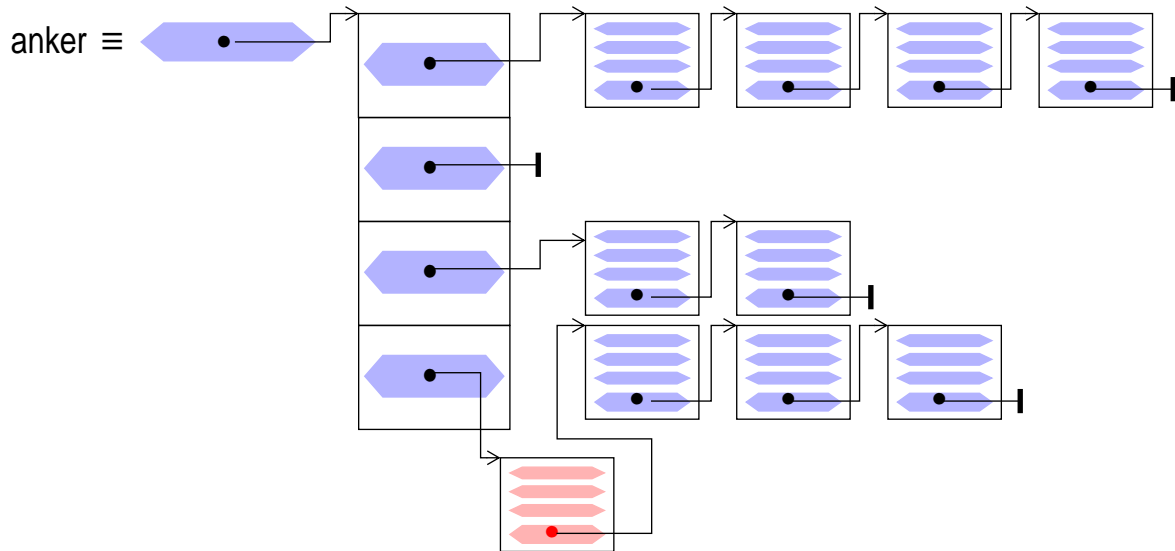
```
struct eintrag *anker[32];
struct eintrag *akt_liste;
...
/* Index aus erstem Buchstaben des Nachnamens ermitteln */
index = akt_eintrag->stud.nachname[0] % 32;
akt_liste = anker[index];

/* akt_liste zeigt jetzt auf die Liste,
   in die akt_eintrag eingetragen wird */
akt_eintrag->naechster = akt_liste;
akt_liste = akt_eintrag;
```

J.1 Hashing (4)

■ ... Beispiel

► Resultat



Z Animationen



C.4 Beispiel (1) — Programm

```
main()
{
    /* Beschreibung der Daten */
    float x;                /* Argument zu dem Pol. berechnet wird */
    float y;                /* Polynomwert */

    float a = 11, b = 7, c = 5, d = 3;
    float zw1, zw2, zw3, zw4, zw5;    /* Zwischenergebnisse */
}
```

C.4 Beispiel (1) — Programm

```
main()
{
    /* Beschreibung der Daten */
    float x;                /* Argument zu dem Pol. berechnet wird */
    float y;                /* Polynomwert */

    float a = 11, b = 7, c = 5, d = 3;
    float zw1, zw2, zw3, zw4, zw5;    /* Zwischenergebnisse */

    /* Einlesen des Arguments */
    printf("Argument = ");
    scanf("%f", &x);
}
```

C.4 Beispiel (1) — Programm

```
main()
{
    /* Beschreibung der Daten */
    float x;                /* Argument zu dem Pol. berechnet wird */
    float y;                /* Polynomwert */

    float a = 11, b = 7, c = 5, d = 3;
    float zw1, zw2, zw3, zw4, zw5;    /* Zwischenergebnisse */

    /* Einlesen des Arguments */
    printf("Argument = ");
    scanf("%f", &x);

    /* Kontrollausgabe der Eingabedaten */
    printf("Der Polynomwert soll fuer das");
    printf(" Argument %f berechnet werden.\n", x);
}
```

C.4 Beispiel (1) — Programm

```
main()
{
    /* Beschreibung der Daten */
    float x;                /* Argument zu dem Pol. berechnet wird */
    float y;                /* Polynomwert */

    float a = 11, b = 7, c = 5, d = 3;
    float zw1, zw2, zw3, zw4, zw5;    /* Zwischenergebnisse */

    /* Einlesen des Arguments */
    printf("Argument = ");
    scanf("%f", &x);

    /* Kontrollausgabe der Eingabedaten */
    printf("Der Polynomwert soll fuer das");
    printf(" Argument %f berechnet werden.\n", x);

    /* Berechnung des Polynomwertes mit Hornerschema */
    zw1 = a * x;
    zw2 = zw1 + b;
    zw3 = zw2 * x;
    zw4 = zw3 + c;
    zw5 = zw4 * x;
    y = zw5 + d;
}
```

C.4 Beispiel (1) — Programm

```
main()
{
    /* Beschreibung der Daten */
    float x;                /* Argument zu dem Pol. berechnet wird */
    float y;                /* Polynomwert */

    float a = 11, b = 7, c = 5, d = 3;
    float zw1, zw2, zw3, zw4, zw5;    /* Zwischenergebnisse */

    /* Einlesen des Arguments */
    printf("Argument = ");
    scanf("%f", &x);

    /* Kontrollausgabe der Eingabedaten */
    printf("Der Polynomwert soll fuer das");
    printf(" Argument %f berechnet werden.\n", x);

    /* Berechnung des Polynomwertes mit Hornerschema */
    zw1 = a * x;
    zw2 = zw1 + b;
    zw3 = zw2 * x;
    zw4 = zw3 + c;
    zw5 = zw4 * x;
    y = zw5 + d;

    /* Ausgabe des Ergebnisses */
    printf("Ergebnis: f(%f) = %f\n", x, y);
}
```

C.7 Beispiel (2) — Programm

```
#include <stdio.h>
#include <math.h>

main ()
{
```

C.7 Beispiel (2) — Programm

```
#include <stdio.h>
#include <math.h>

main ()
{
    float a, b, c;
    float dis;
    /* Parameter in a*x^2+b*x+c */
    /* Diskriminante */
```

C.7 Beispiel (2) — Programm

```
#include <stdio.h>
#include <math.h>

main ()
{
    float a, b, c;           /* Parameter in  $a*x^2+b*x+c$  */
    float dis;              /* Diskriminante */

    /* Initialisierung: Einlesen der Parameter */
    printf("Nullstellen von  $f(x)=a*x^2+b*x+c$ \n");
    printf("a=");
    scanf("%f", &a);
    printf("b=");
    scanf("%f", &b);
    printf("c=");
    scanf("%f", &c);
```

C.7 Beispiel (2) — Programm

```
#include <stdio.h>
#include <math.h>

main ()
{
    float a, b, c;           /* Parameter in a*x^2+b*x+c */
    float dis;              /* Diskriminante */

    /* Initialisierung: Einlesen der Parameter */
    printf("Nullstellen von f(x)=a*x^2+b*x+c\n");
    printf("a=");
    scanf("%f", &a);
    printf("b=");
    scanf("%f", &b);
    printf("c=");
    scanf("%f", &c);

    dis = b*b - 4.0 * a * c; /* Berechnung der Diskriminante */
}
```

C.7 Beispiel (2) — Programm

```
#include <stdio.h>
#include <math.h>

main ()
{
    float a, b, c;           /* Parameter in  $a*x^2+b*x+c$  */
    float dis;              /* Diskriminante */

    /* Initialisierung: Einlesen der Parameter */
    printf("Nullstellen von  $f(x)=a*x^2+b*x+c$ \n");
    printf("a=");
    scanf("%f", &a);
    printf("b=");
    scanf("%f", &b);
    printf("c=");
    scanf("%f", &c);

    dis = b*b - 4.0 * a * c; /* Berechnung der Diskriminante */

    /* Berechnung der Nullstellen */
    if ( dis == 0.0 )        /* doppelte Nullstelle */
        printf("Doppelte Nullstelle bei %f\n", -b/(2.0*a));
}
```

C.7 Beispiel (2) — Programm

```
#include <stdio.h>
#include <math.h>

main ()
{
    float a, b, c;           /* Parameter in  $a*x^2+b*x+c$  */
    float dis;              /* Diskriminante */

    /* Initialisierung: Einlesen der Parameter */
    printf("Nullstellen von  $f(x)=a*x^2+b*x+c$ \n");
    printf("a=");
    scanf("%f", &a);
    printf("b=");
    scanf("%f", &b);
    printf("c=");
    scanf("%f", &c);

    dis = b*b - 4.0 * a * c; /* Berechnung der Diskriminante */

    /* Berechnung der Nullstellen */
    if ( dis == 0.0 )        /* doppelte Nullstelle */
        printf("Doppelte Nullstelle bei %f\n", -b/(2.0*a));
    else if ( dis > 0.0 ) { /* zwei reelle Nullstellen */
        printf("1. Nullstelle bei %f\n", (sqrt(dis)-b)/(2.0*a));
        printf("2. Nullstelle bei %f\n", (-sqrt(dis)-b)/(2.0*a));
    }
}
```

C.7 Beispiel (2) — Programm

```
#include <stdio.h>
#include <math.h>

main ()
{
    float a, b, c;           /* Parameter in a*x^2+b*x+c */
    float dis;              /* Diskriminante */

    /* Initialisierung: Einlesen der Parameter */
    printf("Nullstellen von f(x)=a*x^2+b*x+c\n");
    printf("a=");
    scanf("%f", &a);
    printf("b=");
    scanf("%f", &b);
    printf("c=");
    scanf("%f", &c);

    dis = b*b - 4.0 * a * c; /* Berechnung der Diskriminante */

    /* Berechnung der Nullstellen */
    if ( dis == 0.0 )        /* doppelte Nullstelle */
        printf("Doppelte Nullstelle bei %f\n", -b/(2.0*a));
    else if ( dis > 0.0 ) { /* zwei reelle Nullstellen */
        printf("1. Nullstelle bei %f\n", (sqrt(dis)-b)/(2.0*a));
        printf("2. Nullstelle bei %f\n", (-sqrt(dis)-b)/(2.0*a));
    }
    else                    /* keine reelle Nullstelle */
        printf("Keine reelle Nullstelle\n");
}
```

C.7 Beispiel (2) — Programm

```
#include <stdio.h>
#include <math.h>

main ()
{
    float a, b, c;           /* Parameter in a*x^2+b*x+c */
    float dis;              /* Diskriminante */

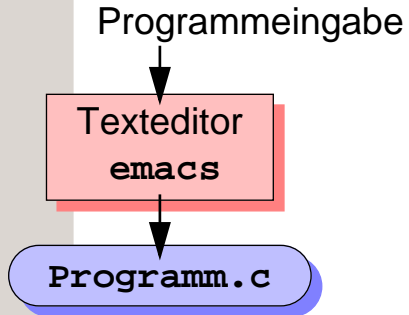
    /* Initialisierung: Einlesen der Parameter */
    printf("Nullstellen von f(x)=a*x^2+b*x+c\n");
    printf("a=");
    scanf("%f", &a);
    printf("b=");
    scanf("%f", &b);
    printf("c=");
    scanf("%f", &c);

    dis = b*b - 4.0 * a * c; /* Berechnung der Diskriminante */

    /* Berechnung der Nullstellen */
    if ( dis == 0.0 )        /* doppelte Nullstelle */
        printf("Doppelte Nullstelle bei %f\n", -b/(2.0*a));
    else if ( dis > 0.0 ) { /* zwei reelle Nullstellen */
        printf("1. Nullstelle bei %f\n", (sqrt(dis)-b)/(2.0*a));
        printf("2. Nullstelle bei %f\n", (-sqrt(dis)-b)/(2.0*a));
    }
    else                    /* keine reelle Nullstelle */
        printf("Keine reelle Nullstelle\n");
}
```

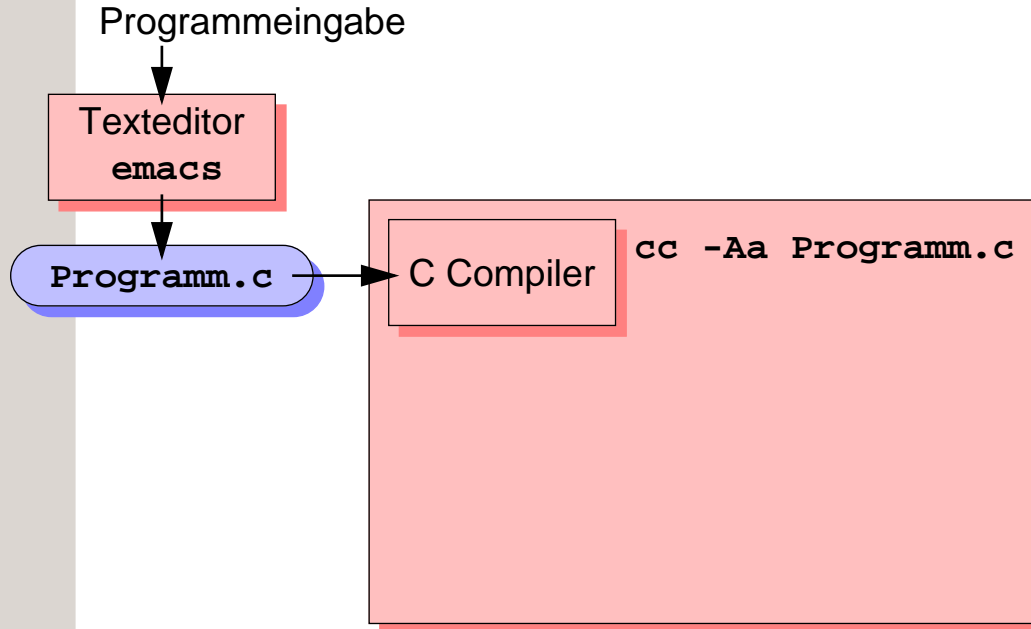
C.8 Programmerstellung und -ausführung

1 Vorgehensweise



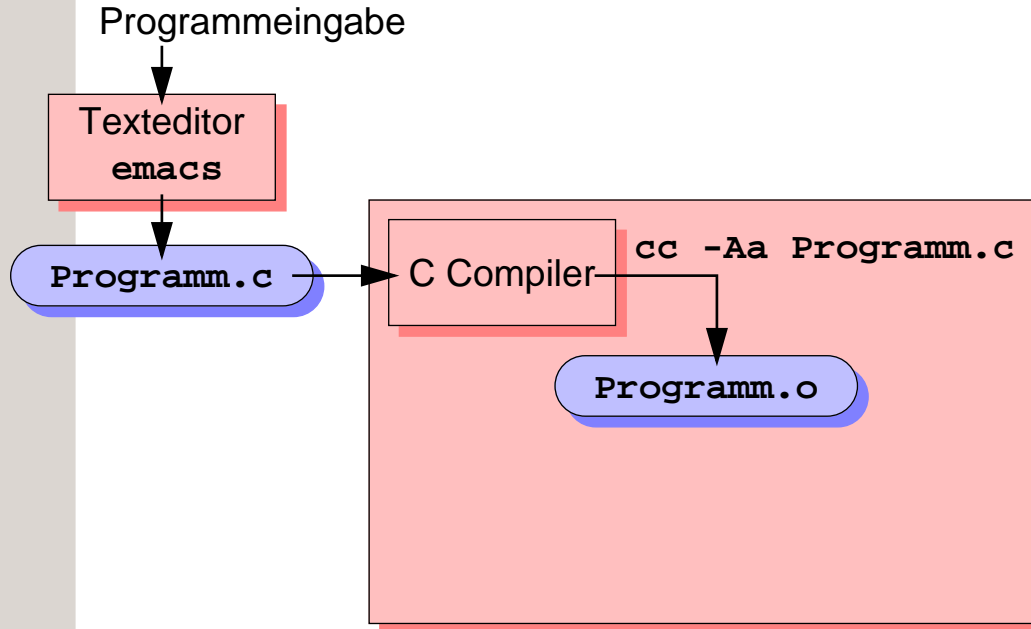
C.8 Programmierstellung und -ausführung

1 Vorgehensweise



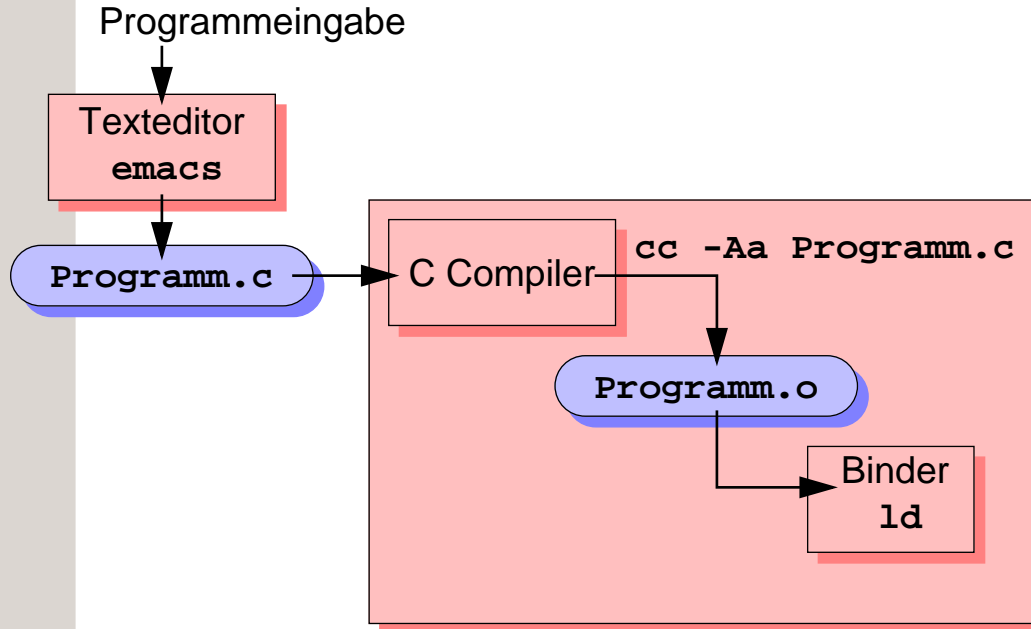
C.8 Programmerstellung und -ausführung

1 Vorgehensweise



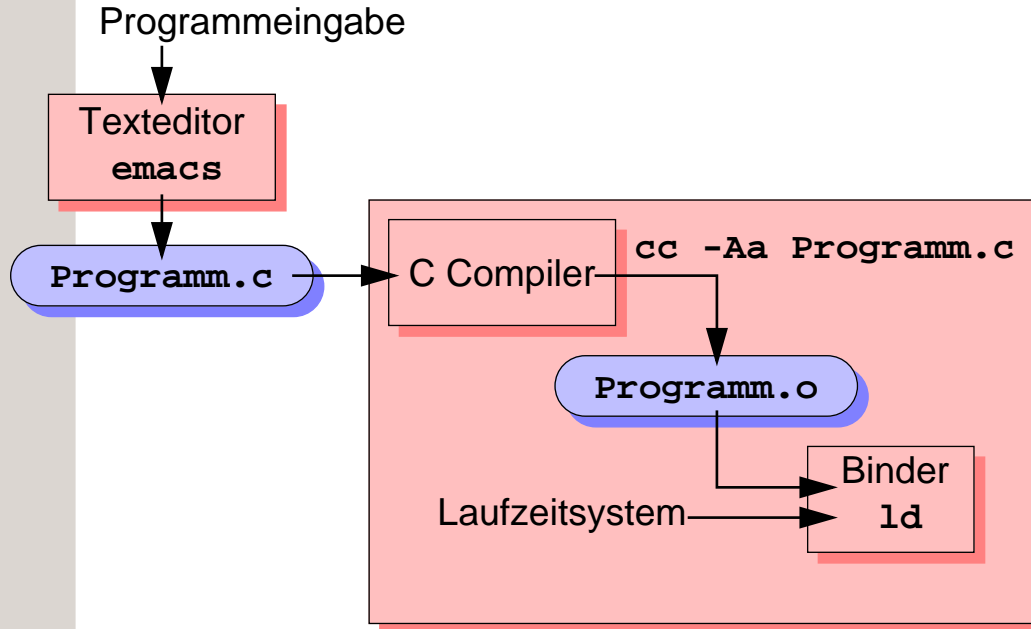
C.8 Programmierstellung und -ausführung

1 Vorgehensweise



C.8 Programmerstellung und -ausführung

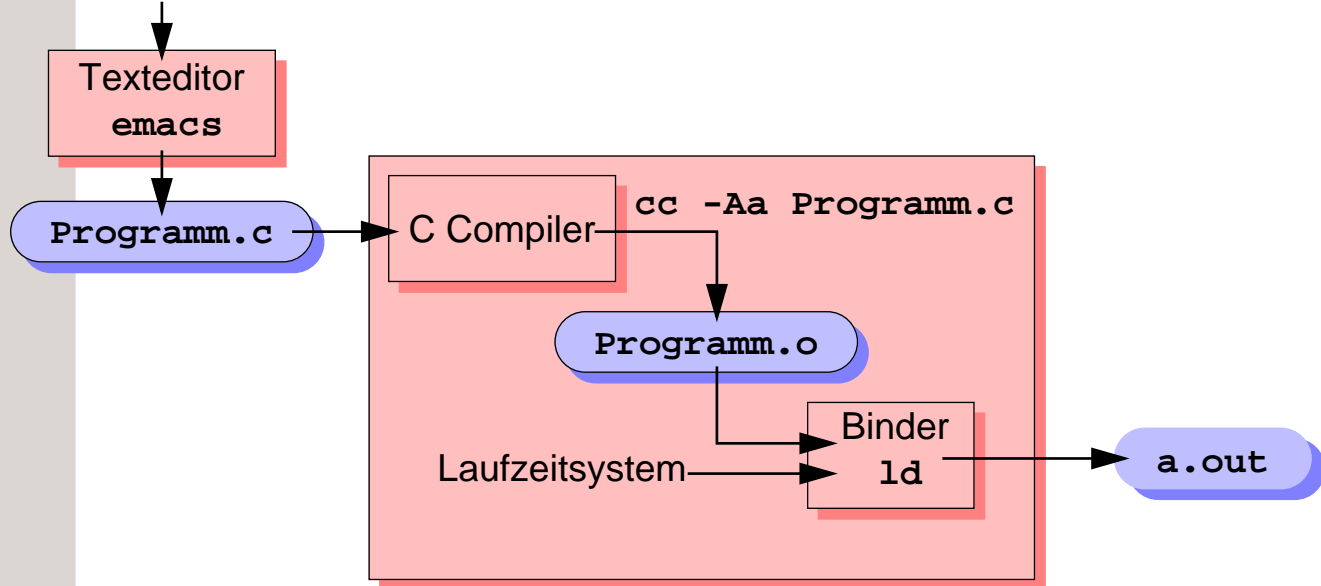
1 Vorgehensweise



C.8 Programmerstellung und -ausführung

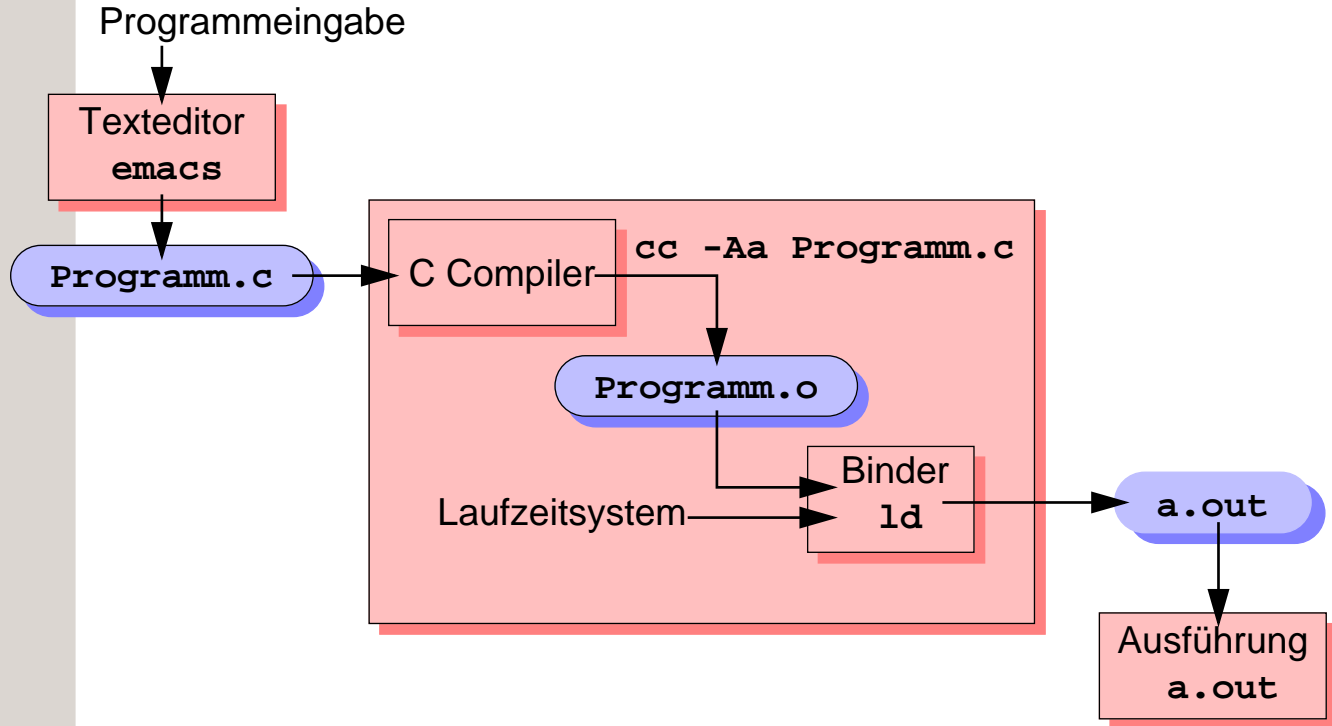
1 Vorgehensweise

Programmeingabe



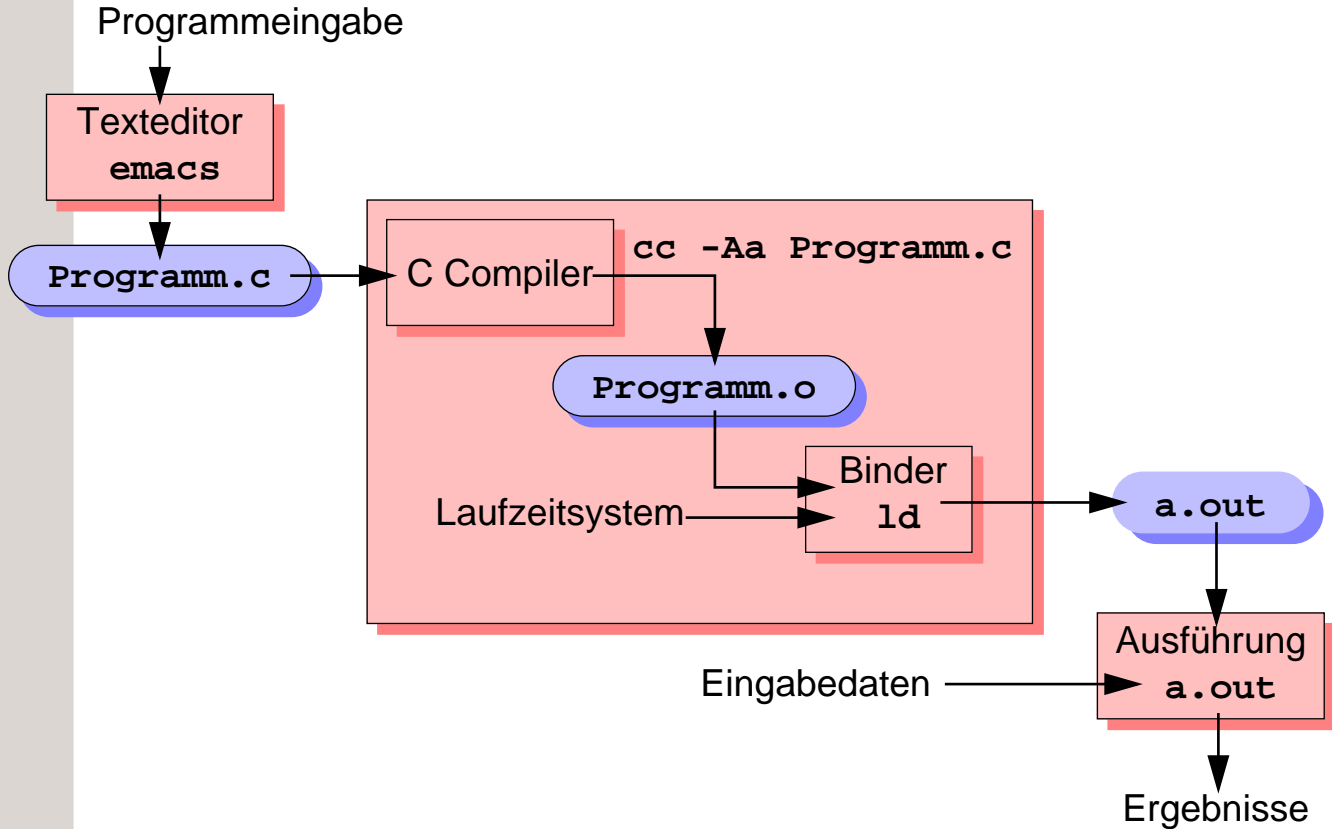
C.8 Programmierstellung und -ausführung

1 Vorgehensweise



C.8 Programmierstellung und -ausführung

1 Vorgehensweise



3 Übersetzung und Ausführung der Beispielprogramme

■ Ausführung des Beispiels 'Polynom'

```
faii06a% cc -Aa polynom.c
```

```
faii06a% a.out
```

```
Argument =
```

3 Übersetzung und Ausführung der Beispielprogramme

■ Ausführung des Beispiels 'Polynom'

```
faii06a% cc -Aa polynom.c
```

```
faii06a% a.out
```

```
Argument = 4
```

3 Übersetzung und Ausführung der Beispielprogramme

■ Ausführung des Beispiels 'Polynom'

```
faii06a% cc -Aa polynom.c
```

```
faii06a% a.out
```

```
Argument = 4
```

```
Der Polynomwert soll fuer das Argument 4.000000 berechnet werden.
```

```
Ergebnis: f(4.000000) = 839.000000
```

```
faii06a%
```

3 Übersetzung und Ausführung der Beispielprogramme

■ Ausführung des Beispiels 'Polynom'

```
faii06a% cc -Aa polynom.c
```

```
faii06a% a.out
```

```
Argument = 4
```

```
Der Polynomwert soll fuer das Argument 4.000000 berechnet werden.
```

```
Ergebnis: f(4.000000) = 839.000000
```

```
faii06a%
```

■ Ausführung des Beispiels 'Nullstellen'

```
faii06a% cc -Aa Nullstellen.c -lm
```

```
faii06a% a.out
```

```
Nullstellen von  $f(x)=a*x^2+b*x+c$ 
```

```
a=
```

3 Übersetzung und Ausführung der Beispielprogramme

■ Ausführung des Beispiels 'Polynom'

```
faii06a% cc -Aa polynom.c
```

```
faii06a% a.out
```

```
Argument = 4
```

Der Polynomwert soll fuer das Argument 4.000000 berechnet werden.

```
Ergebnis: f(4.000000) = 839.000000
```

```
faii06a%
```

■ Ausführung des Beispiels 'Nullstellen'

```
faii06a% cc -Aa Nullstellen.c -lm
```

```
faii06a% a.out
```

```
Nullstellen von  $f(x)=a*x^2+b*x+c$ 
```

```
a=1
```

```
b=
```

3 Übersetzung und Ausführung der Beispielprogramme

■ Ausführung des Beispiels 'Polynom'

```
faii06a% cc -Aa polynom.c
```

```
faii06a% a.out
```

```
Argument = 4
```

Der Polynomwert soll fuer das Argument 4.000000 berechnet werden.

```
Ergebnis: f(4.000000) = 839.000000
```

```
faii06a%
```

■ Ausführung des Beispiels 'Nullstellen'

```
faii06a% cc -Aa Nullstellen.c -lm
```

```
faii06a% a.out
```

```
Nullstellen von  $f(x)=a*x^2+b*x+c$ 
```

```
a=1
```

```
b=2
```

```
c=-3
```

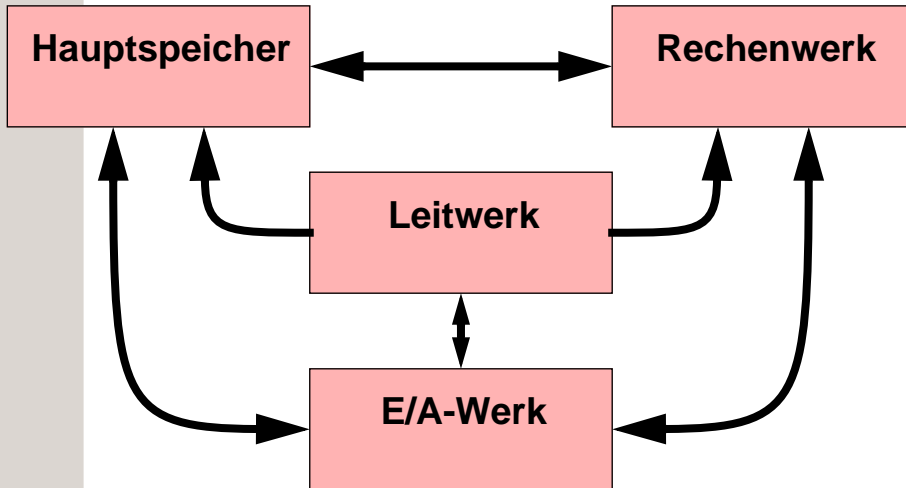
```
1. Nullstelle bei 1.000000
```

```
2. Nullstelle bei -3.000000
```

```
faii06a%
```

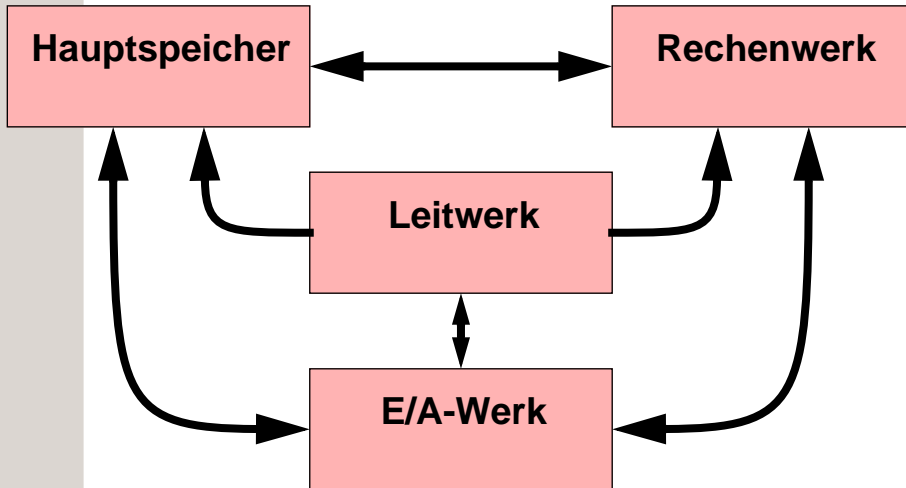
C.9 Klassischer Universalrechner

- *Burks, Goldstine und von Neumann (Princeton 1946/47)*



C.9 Klassischer Universalrechner

- *Burks, Goldstine und von Neumann (Princeton 1946/47)*



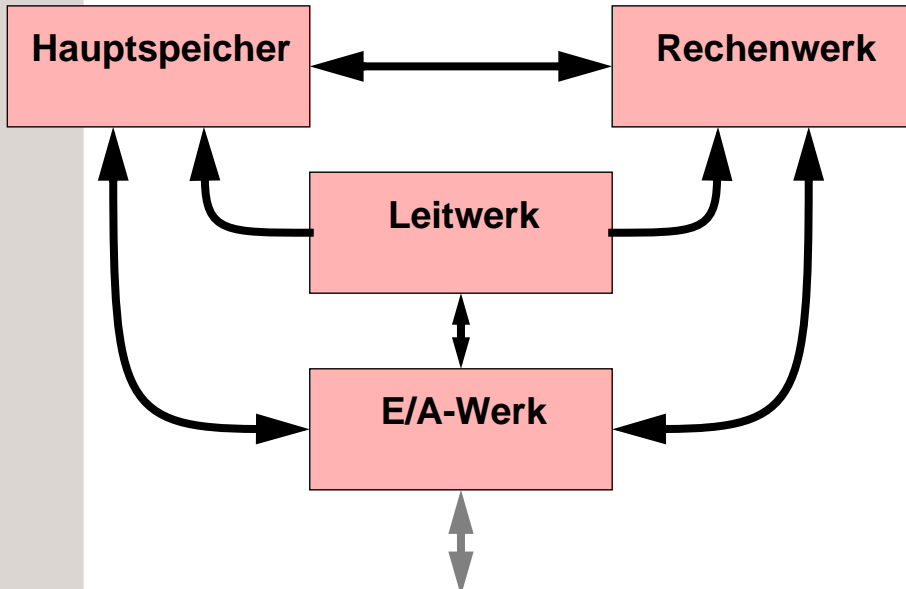
Rechenwerk:
Rechenoperationen und
logische Verknüpfungen

Hauptspeicher:
Programm und Daten

Leitwerk:
steuert Programmablauf

C.9 Klassischer Universalrechner

- *Burks, Goldstine und von Neumann (Princeton 1946/47)*



Rechenwerk:
Rechenoperationen und
logische Verknüpfungen

Hauptspeicher:
Programm und Daten

Leitwerk:
steuert Programmablauf

E/A-Werk:
Ein-/Ausgabe von Daten
und Programmen



C.10 Datendarstellung im Rechner

- Alle Informationen (Daten, Texte, Programmanweisungen) werden im Speicher durch Bitfolgen dargestellt
 - ◆ Bit = Speicherstelle mit Wert 0 oder 1
 - ◆ Beispiele: Zahl 2 = 10, Zahl 5 = 101, Buchstabe a = Zahl 97 = 1100 0001
- 8 Bit bilden ein Byte

C.10 Datendarstellung im Rechner

- Alle Informationen (Daten, Texte, Programmanweisungen) werden im Speicher durch Bitfolgen dargestellt
 - ◆ **Bit** = Speicherstelle mit Wert 0 oder 1
 - ◆ Beispiele: Zahl 2 = 10, Zahl 5 = 101, Buchstabe a = Zahl 97 = 1100 0001
- 8 Bit bilden ein **Byte**
- Für ganze Zahlen werden meist 32 Bit vorgesehen (ein **Wort**)
- Zeichen (Buchstaben) werden durch Zahlen kodiert
- Ein Wort ist normalerweise die Einheit, mit der das Rechenwerk operiert

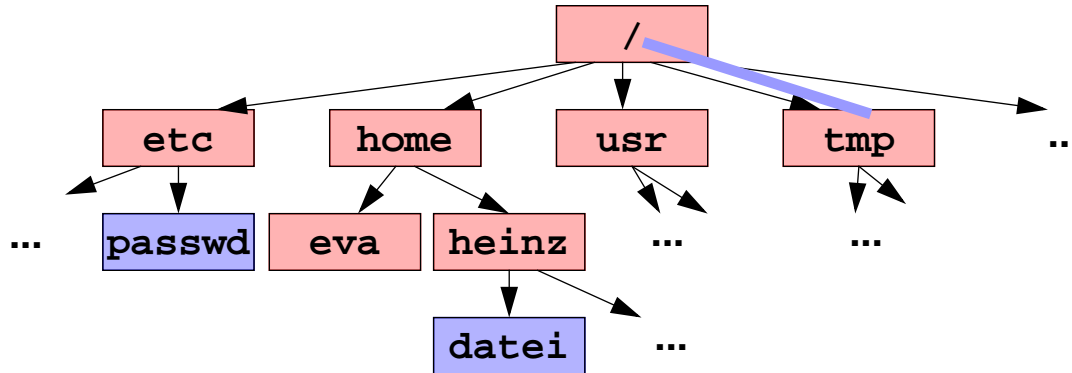
C.10 Datendarstellung im Rechner

- Alle Informationen (Daten, Texte, Programmanweisungen) werden im Speicher durch Bitfolgen dargestellt
 - ◆ **Bit** = Speicherstelle mit Wert 0 oder 1
 - ◆ Beispiele: Zahl 2 = 10, Zahl 5 = 101, Buchstabe a = Zahl 97 = 1100 0001
- 8 Bit bilden ein **Byte**
- Für ganze Zahlen werden meist 32 Bit vorgesehen (ein **Wort**)
- Zeichen (Buchstaben) werden durch Zahlen kodiert
- Ein Wort ist normalerweise die Einheit, mit der das Rechenwerk operiert
- Das Rechenwerk führt entsprechend den Anweisungen des Programms arithmetische oder logische Verknüpfungen von Worten durch

C.10 Datendarstellung im Rechner

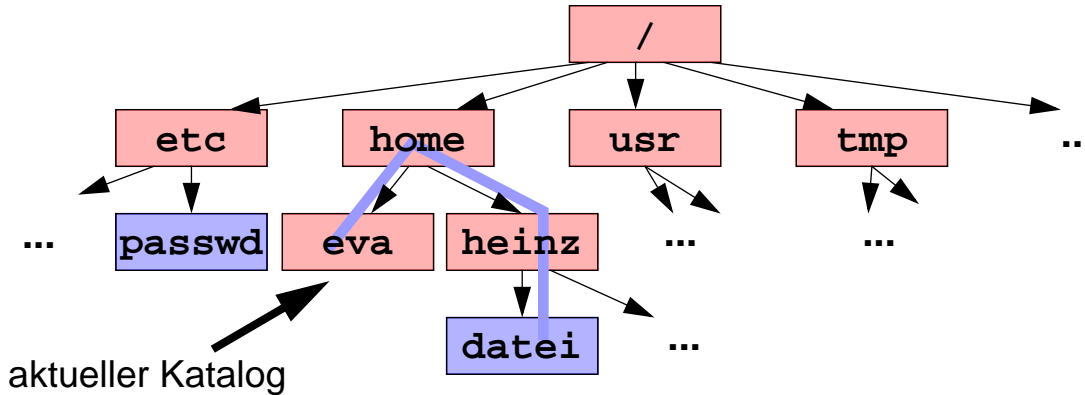
- Alle Informationen (Daten, Texte, Programmanweisungen) werden im Speicher durch Bitfolgen dargestellt
 - ◆ **Bit** = Speicherstelle mit Wert 0 oder 1
 - ◆ Beispiele: Zahl 2 = 10, Zahl 5 = 101, Buchstabe a = Zahl 97 = 1100 0001
- 8 Bit bilden ein **Byte**
- Für ganze Zahlen werden meist 32 Bit vorgesehen (ein **Wort**)
- Zeichen (Buchstaben) werden durch Zahlen kodiert
- Ein Wort ist normalerweise die Einheit, mit der das Rechenwerk operiert
- Das Rechenwerk führt entsprechend den Anweisungen des Programms arithmetische oder logische Verknüpfungen von Worten durch
- Die Worte auf denen das Rechenwerk operiert, werden in **Registern** des Rechenwerks zwischengespeichert (für schnelleren Zugriff)

■ Baumstruktur & Pfade im Dateibaum



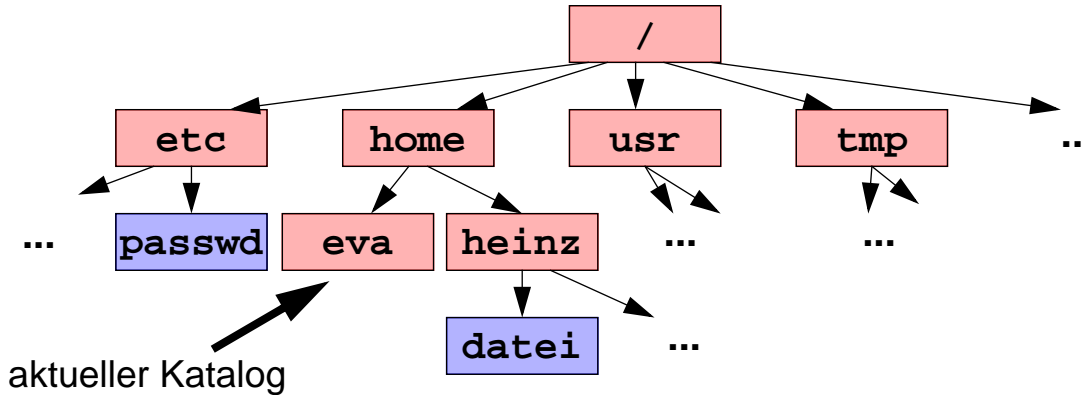
◆ z. B. `/home/heinz/datei` `/tmp`

■ Baumstruktur & Pfade im Dateibaum



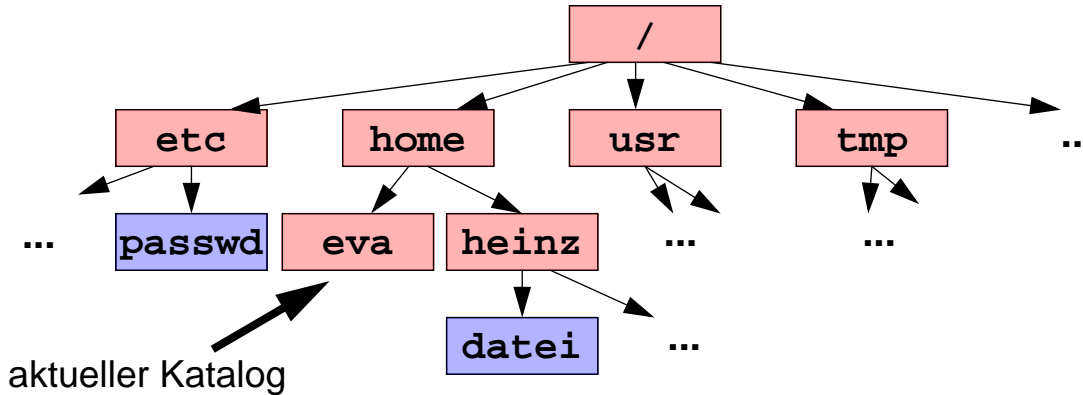
◆ z. B. `/home/heinz/datei` `/tmp` `../heinz/datei`

■ Baumstruktur & Pfade im Dateibaum



- ◆ z. B. `/home/heinz/datei` `/tmp` `../heinz/datei`
- ◆ `/` ist Trennsymbol (*Slash*) zwischen Pfadkomponenten
Pfade, die mit `/` beginnen, starten im Wurzelkatalog (*root*)
sonst Beginn implizit im aktuellem Katalog

■ Baumstruktur & Pfade im Dateibaum

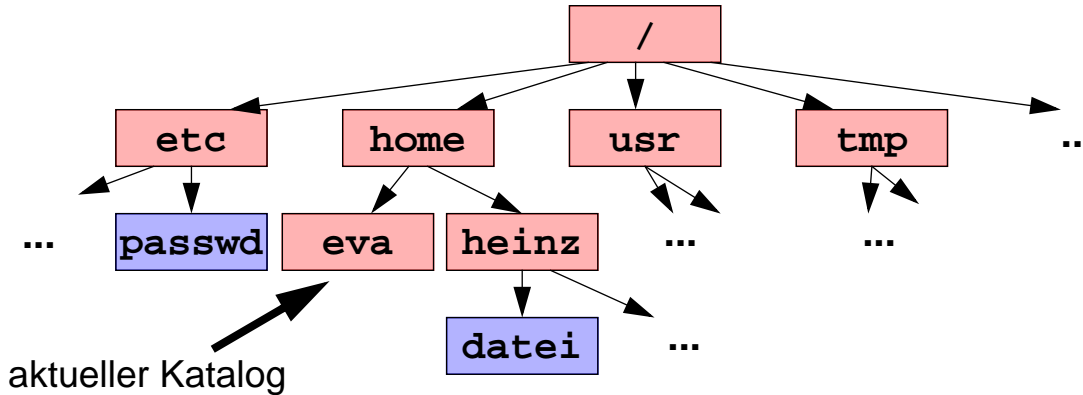


◆ z. B. `/home/heinz/datei` `/tmp` `../heinz/datei`

◆ `/` ist Trennsymbol (*Slash*) zwischen Pfadkomponenten
Pfade, die mit `/` beginnen, starten im Wurzelkatalog (*root*)
sonst Beginn implizit im aktuellem Katalog

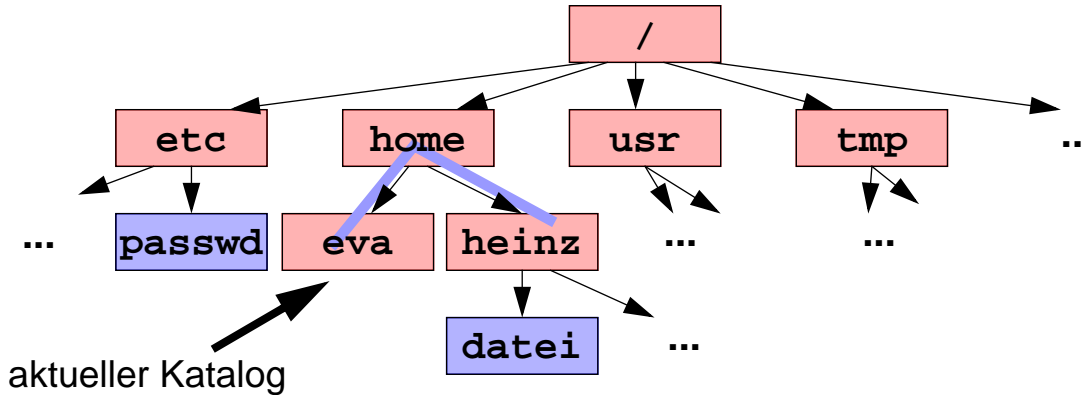
◆ jeder Katalog enthält Verweise auf sich selbst (`.`) und auf den darüberliegenden Katalog (`..`) — in Pfadnamen verwendbar!

■ Baumstruktur & Pfade im Dateibaum



- ◆ z. B. `/home/heinz/datei` `/tmp` `../heinz/datei`
- ◆ `/` ist Trennsymbol (*Slash*) zwischen Pfadkomponenten
Pfade, die mit `/` beginnen, starten im Wurzelkatalog (*root*)
sonst Beginn implizit im aktuellem Katalog
- ◆ jeder Katalog enthält Verweise auf sich selbst (`.`) und auf den darüberliegenden Katalog (`..`) — in Pfadnamen verwendbar!
- ◆ aktueller Katalog kann mit Kommando `cd` gewechselt werden

■ Baumstruktur & Pfade im Dateibaum



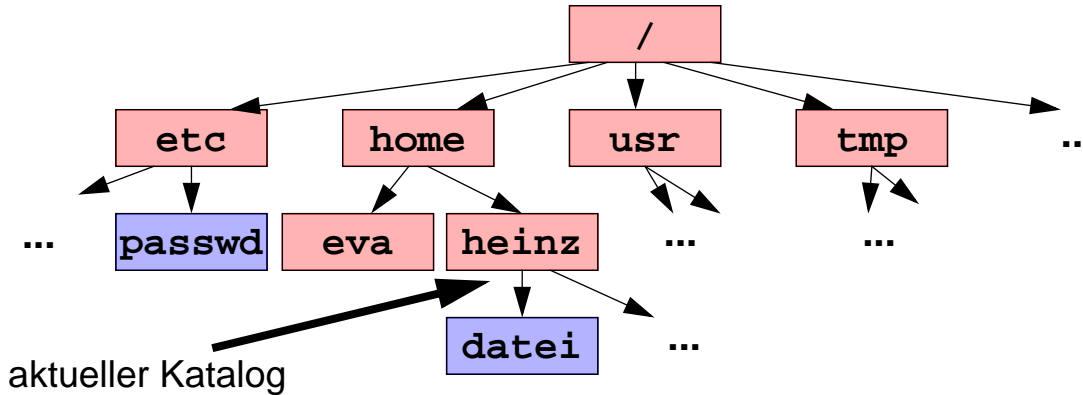
◆ z. B. `/home/heinz/datei` `/tmp` `../heinz/datei`

◆ `/` ist Trennsymbol (*Slash*) zwischen Pfadkomponenten
Pfade, die mit `/` beginnen, starten im Wurzelkatalog (*root*)
sonst Beginn implizit im aktuellem Katalog

◆ jeder Katalog enthält Verweise auf sich selbst (`.`) und auf den darüberliegenden Katalog (`..`) — in Pfadnamen verwendbar!

◆ aktueller Katalog kann mit Kommando `cd` gewechselt werden
z. B. `cd ../heinz`

■ Baumstruktur & Pfade im Dateibaum



- ◆ z. B. `/home/heinz/datei` `/tmp` `../heinz/datei`
- ◆ `/` ist Trennsymbol (*Slash*) zwischen Pfadkomponenten
Pfade, die mit `/` beginnen, starten im Wurzelkatalog (*root*)
sonst Beginn implizit im aktuellem Katalog
- ◆ jeder Katalog enthält Verweise auf sich selbst (`.`) und auf den darüberliegenden Katalog (`..`) — in Pfadnamen verwendbar!
- ◆ aktueller Katalog kann mit Kommando `cd` gewechselt werden
z. B. `cd ../heinz`

C.12 Benutzerumgebung

- Das Betriebssystem (UNIX) und eine Reihe von Dienstprogrammen (*Tools*) stellen eine Umgebung bereit, die von der eigentlichen Rechnerarchitektur abstrahiert
- Jedem Benutzer ist ein eigener Dateikatalog zugeordnet (*Home-Directory*)
- Nach dem *Login* wird automatisch eine Fensteroberfläche mit mehreren Fenstern gestartet (*X-Windows*)
 - ◆ Weitere Fenster sind über Menüs mit der Maus zu starten
- Jedes Fenster entspricht einem virtuellen Terminal
 - ◆ Kommandointerpreter (Shell) zur Eingabe von Kommandos (C-Compiler, Dateikatalog auflisten, ...)
 - ◆ Editor (Emacs) zur Eingabe von Programmen und Texten
 - ◆ ...

C.12 Benutzerumgebung (2)

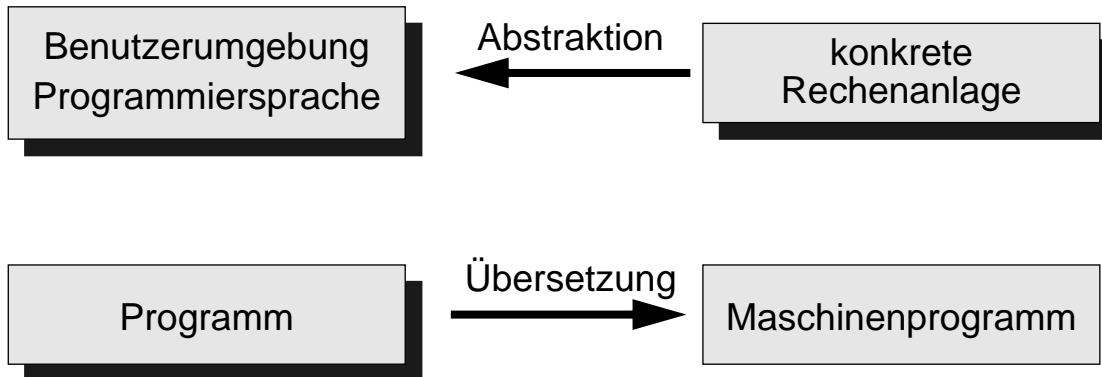
- Beschreibung über die Benutzerumgebung und alle Kommandos sind online auf dem Rechner abrufbar
 - ◆ Integriert in World-Wide-Web (WWW)
 - ◆ Tool: *Mosaic* (über Menü starten, dann selbsterklärend)

C.13 Programmierumgebung

- Programmiersprache bietet Sicht auf eine abstrakte Rechenanlage
 - ◆ Statt Bits, Bytes und Worten:
 Ganze und reelle Zahlen, Zeichen
 - ◆ Statt einfachen Abfragen und direkten Sprüngen an Adressen:
 if - then - else, Schleifen, ...

C.13 Programmierumgebung (2)

- Abbildung Programmiersprache → Maschinensprache
- ◆ Interpreter: Ein Dienstprogramm interpretiert das Programm des Anwenders
- ◆ Compiler: Ein Dienstprogramm übersetzt das Anwenderprogramm in Maschinensprache



C.14 Begriffe

Programm:

Eine in einer Programmiersprache abgefaßte Verarbeitungsvorschrift mit einer (evtl. impliziten) Vereinbarung der Daten

Maschinensprache:

Programmiersprache, die nur Anweisungen zuläßt, die von einer bestimmten Rechenanlage (unmittelbar) ausgeführt werden können

Maschinenprogramm:

Ein in Maschinensprache abgefaßtes Programm

Maschinenorientierte Programmiersprache:

Programmiersprache, deren Anweisungen ähnliche/gleiche Struktur haben wie die der Maschinensprache einer bestimmten Rechenanlage

Maschinenorientiertes Programm:

Ein in maschinenorientierter Programmiersprache abgefaßtes Programm

Höhere / problemorientierte Programmiersprache:

Programmiersprache, die dazu dient, Programme eines bestimmten Anwendungsbereiches unabhängig von einer bestimmten Rechenanlage abzufassen, und die dem Anwendungsbereich besonders angemessen ist (FORTRAN, PASCAL, COBOL, C, C++, ...).

C - Programm:

Ein in der Programmiersprache C abgefaßtes Programm

C.4 Beispiel (1) — Programm

```
main()
{
    /* Beschreibung der Daten */
    float x;                /* Argument zu dem Pol. berechnet wird */
    float y;                /* Polynomwert */

    float a = 11, b = 7, c = 5, d = 3;
    float zw1, zw2, zw3, zw4, zw5;    /* Zwischenergebnisse */
}
```

C.4 Beispiel (1) — Programm

```
main()
{
    /* Beschreibung der Daten */
    float x;                /* Argument zu dem Pol. berechnet wird */
    float y;                /* Polynomwert */

    float a = 11, b = 7, c = 5, d = 3;
    float zw1, zw2, zw3, zw4, zw5;    /* Zwischenergebnisse */

    /* Einlesen des Arguments */
    printf("Argument = ");
    scanf("%f", &x);
}
```

C.4 Beispiel (1) — Programm

```
main()
{
    /* Beschreibung der Daten */
    float x;                /* Argument zu dem Pol. berechnet wird */
    float y;                /* Polynomwert */

    float a = 11, b = 7, c = 5, d = 3;
    float zw1, zw2, zw3, zw4, zw5;    /* Zwischenergebnisse */

    /* Einlesen des Arguments */
    printf("Argument = ");
    scanf("%f", &x);

    /* Kontrollausgabe der Eingabedaten */
    printf("Der Polynomwert soll fuer das");
    printf(" Argument %f berechnet werden.\n", x);
}
```

C.4 Beispiel (1) — Programm

```
main()
{
    /* Beschreibung der Daten */
    float x;                /* Argument zu dem Pol. berechnet wird */
    float y;                /* Polynomwert */

    float a = 11, b = 7, c = 5, d = 3;
    float zw1, zw2, zw3, zw4, zw5;    /* Zwischenergebnisse */

    /* Einlesen des Arguments */
    printf("Argument = ");
    scanf("%f", &x);

    /* Kontrollausgabe der Eingabedaten */
    printf("Der Polynomwert soll fuer das");
    printf(" Argument %f berechnet werden.\n", x);

    /* Berechnung des Polynomwertes mit Hornerschema */
    zw1 = a * x;
    zw2 = zw1 + b;
    zw3 = zw2 * x;
    zw4 = zw3 + c;
    zw5 = zw4 * x;
    y = zw5 + d;
}
```

C.4 Beispiel (1) — Programm

```
main()
{
    /* Beschreibung der Daten */
    float x;                /* Argument zu dem Pol. berechnet wird */
    float y;                /* Polynomwert */

    float a = 11, b = 7, c = 5, d = 3;
    float zw1, zw2, zw3, zw4, zw5;    /* Zwischenergebnisse */

    /* Einlesen des Arguments */
    printf("Argument = ");
    scanf("%f", &x);

    /* Kontrollausgabe der Eingabedaten */
    printf("Der Polynomwert soll fuer das");
    printf(" Argument %f berechnet werden.\n", x);

    /* Berechnung des Polynomwertes mit Hornerschema */
    zw1 = a * x;
    zw2 = zw1 + b;
    zw3 = zw2 * x;
    zw4 = zw3 + c;
    zw5 = zw4 * x;
    y = zw5 + d;

    /* Ausgabe des Ergebnisses */
    printf("Ergebnis: f(%f) = %f\n", x, y);
}
```


1 Definition von Zeigervariablen

■ Syntax:

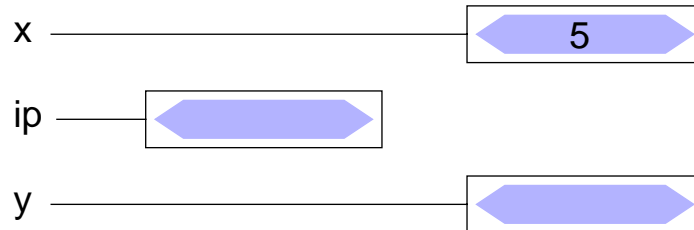
```
Typ *Name ;
```

2 Beispiele

```
int x = 5;
```

```
int *ip;
```

```
int y;
```



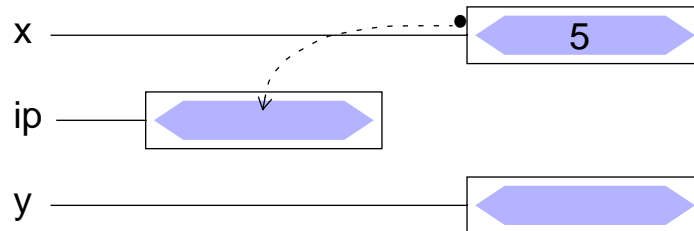
1 Definition von Zeigervariablen

■ Syntax:

```
Typ *Name ;
```

2 Beispiele

```
int x = 5;  
  
int *ip;  
  
int y;  
  
ip = &x; ❶
```



1 Definition von Zeigervariablen

■ Syntax:

```
Typ *Name ;
```

2 Beispiele

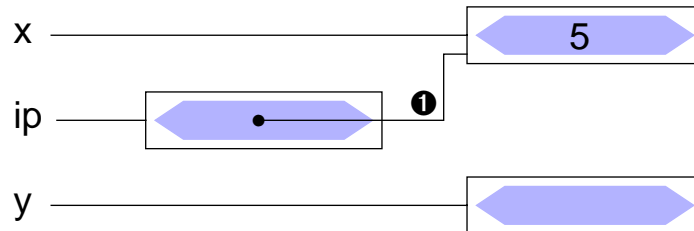
```
int x = 5;
```

```
int *ip;
```

```
int y;
```

```
ip = &x; ①
```

```
y = *ip; ②
```



1 Definition von Zeigervariablen

■ Syntax:

```
Typ *Name ;
```

2 Beispiele

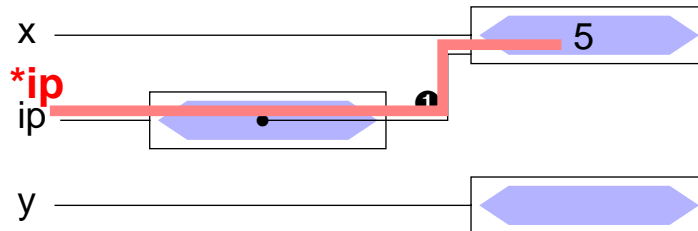
```
int x = 5;
```

```
int *ip;
```

```
int y;
```

```
ip = &x; ①
```

```
y = *ip; ②
```



1 Definition von Zeigervariablen

■ Syntax:

```
Typ *Name ;
```

2 Beispiele

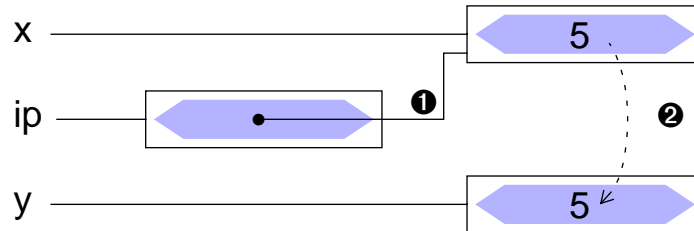
```
int x = 5;
```

```
int *ip;
```

```
int y;
```

```
ip = &x; ①
```

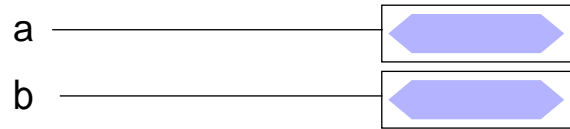
```
y = *ip; ②
```



H.4 ... Zeiger als Funktionsargumente (2)

■ Beispiel:

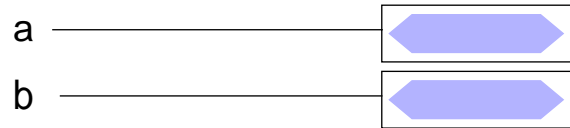
```
main(void) {  
  int a, b;  
  void swap (int *, int *);  
  ...  
  swap(&a, &b);  
}
```



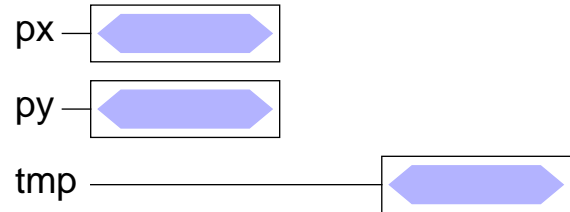
H.4 ... Zeiger als Funktionsargumente (2)

■ Beispiel:

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



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

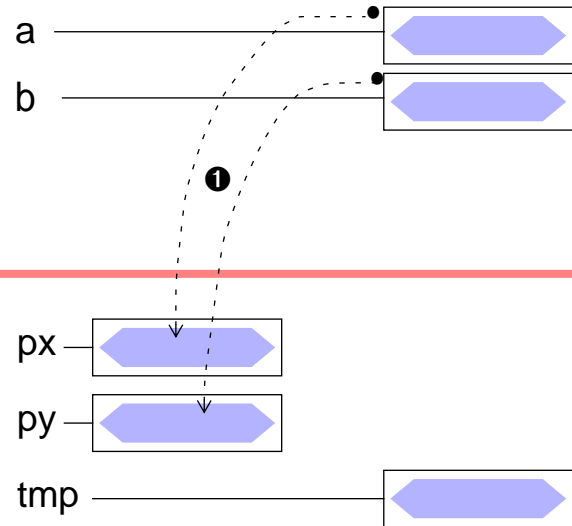


H.4 ... Zeiger als Funktionsargumente (2)

■ Beispiel:

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

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

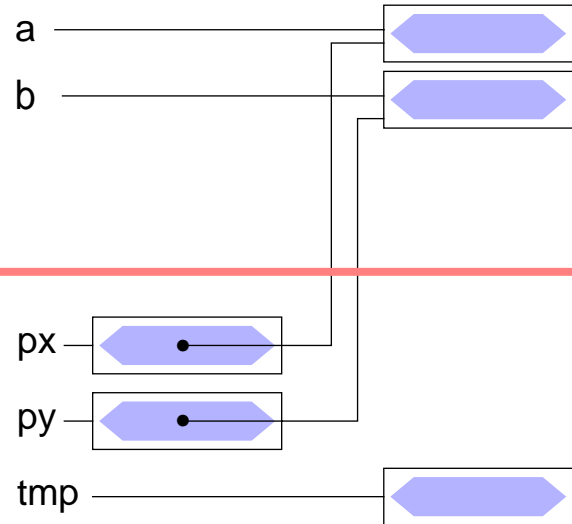


H.4 ... Zeiger als Funktionsargumente (2)

■ Beispiel:

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

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

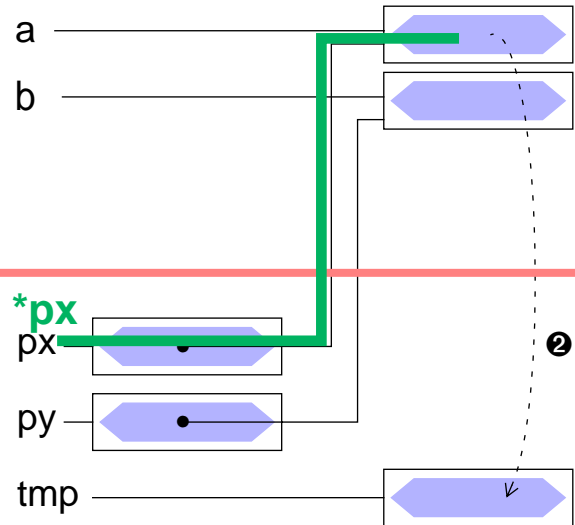


H.4 ... Zeiger als Funktionsargumente (2)

■ 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;  
    *py = tmp;  
  
}
```

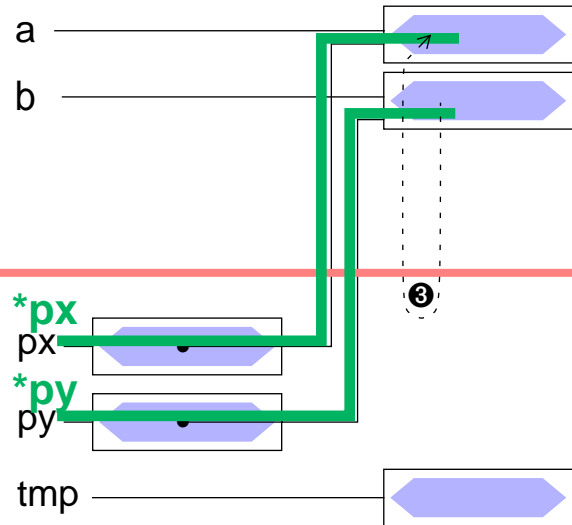


H.4 ... Zeiger als Funktionsargumente (2)

■ Beispiel:

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

```
void swap (int *px, int *py)  
{  
  int tmp;  
  
  tmp = *px;  
  *px = *py; ③  
  *py = tmp;  
}
```

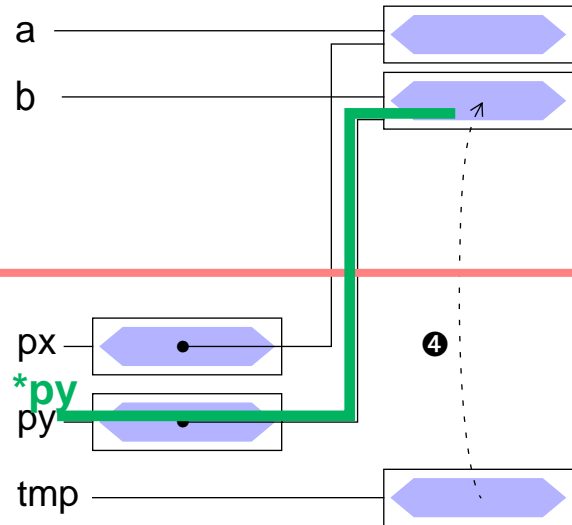


H.4 ... Zeiger als Funktionsargumente (2)

■ 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  
  
}
```

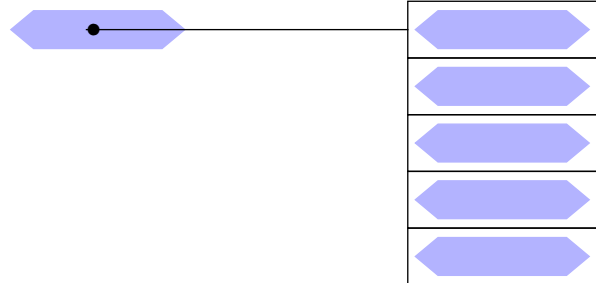


H.5 Zeiger und Felder (1)

- 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];
```

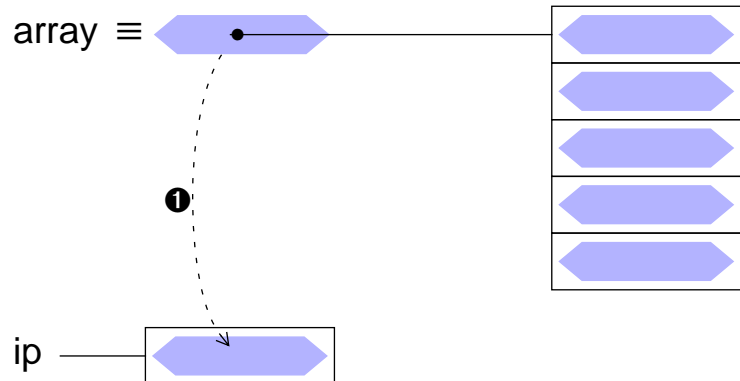
array ≡



H.5 Zeiger und Felder (1)

- 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:

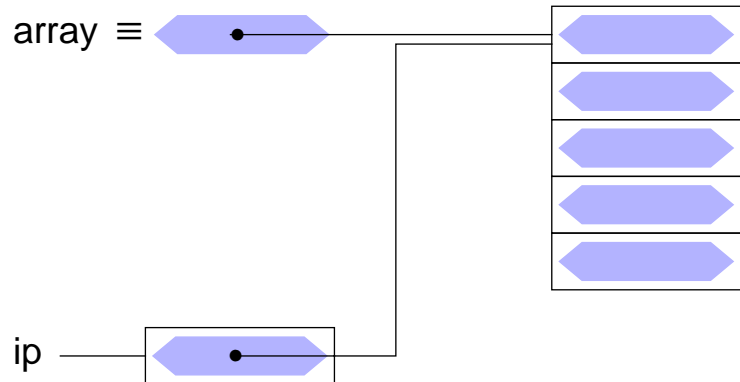
```
int array[5];  
  
int *ip = array; ❶
```



H.5 Zeiger und Felder (1)

- 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:

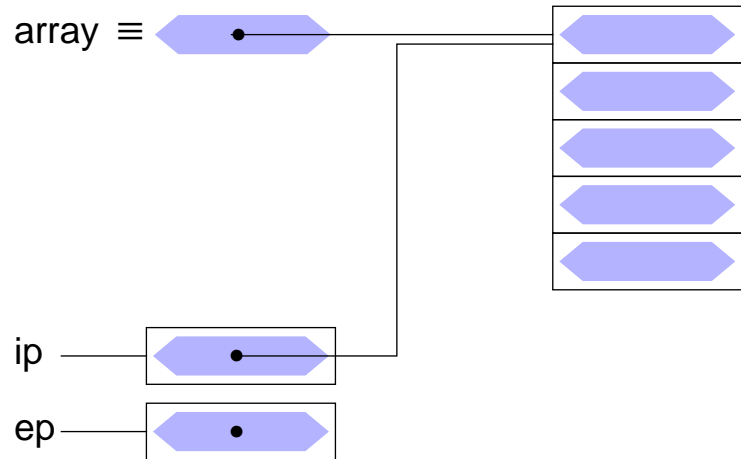
```
int array[5];  
  
int *ip = array; ❶
```



H.5 Zeiger und Felder (1)

- 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:

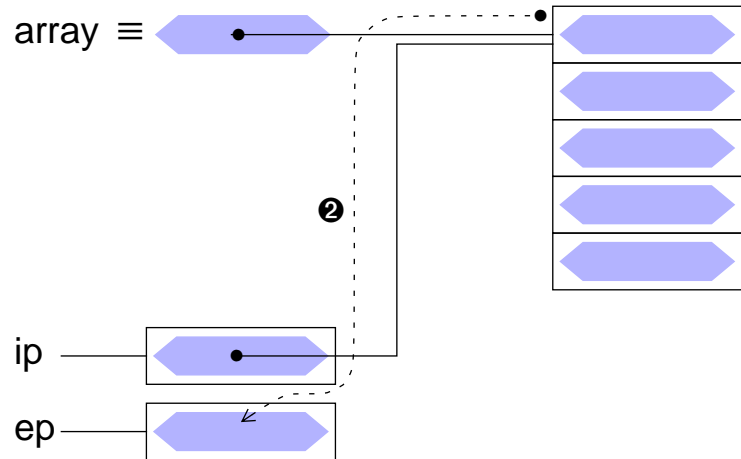
```
int array[5];  
  
int *ip = array;  
  
int *ep;
```



H.5 Zeiger und Felder (1)

- 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:

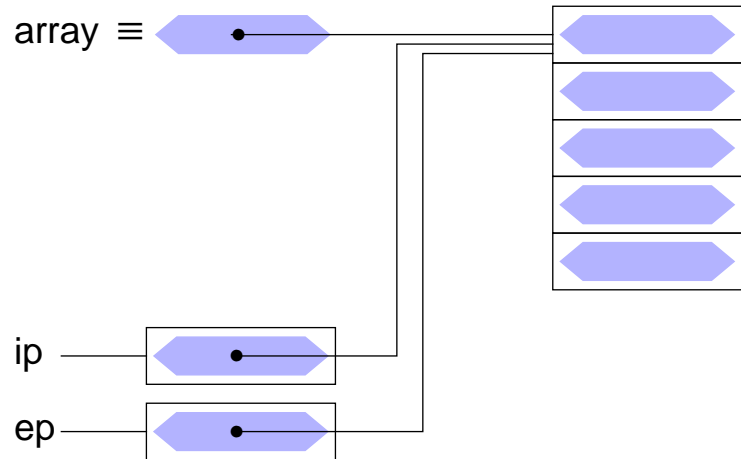
```
int array[5];  
  
int *ip = array;  
  
int *ep;  
ep = &array[0]; ②
```



H.5 Zeiger und Felder (1)

- 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:

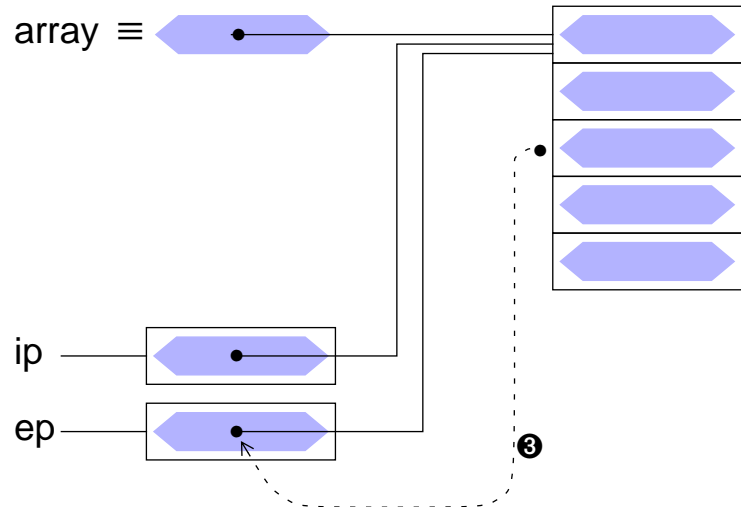
```
int array[5];  
  
int *ip = array;  
  
int *ep;  
ep = &array[0]; ②
```



H.5 Zeiger und Felder (1)

- 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:

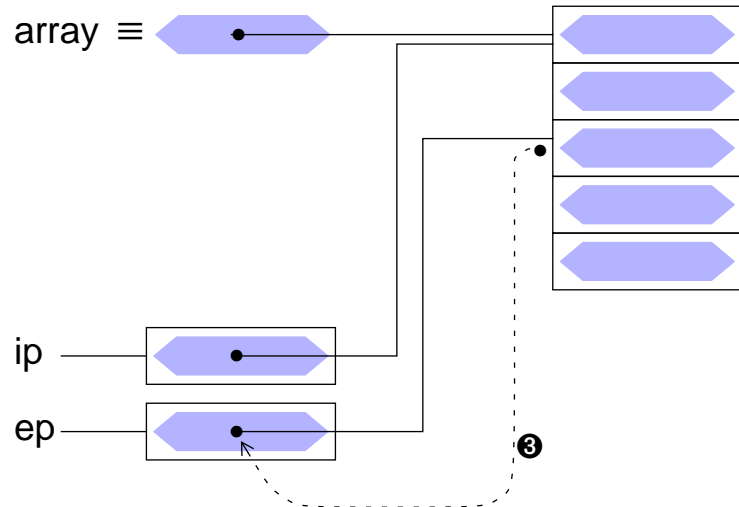
```
int array[5];  
  
int *ip = array;  
  
int *ep;  
ep = &array[0];  
  
ep = &array[2]; ③
```



H.5 Zeiger und Felder (1)

- 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:

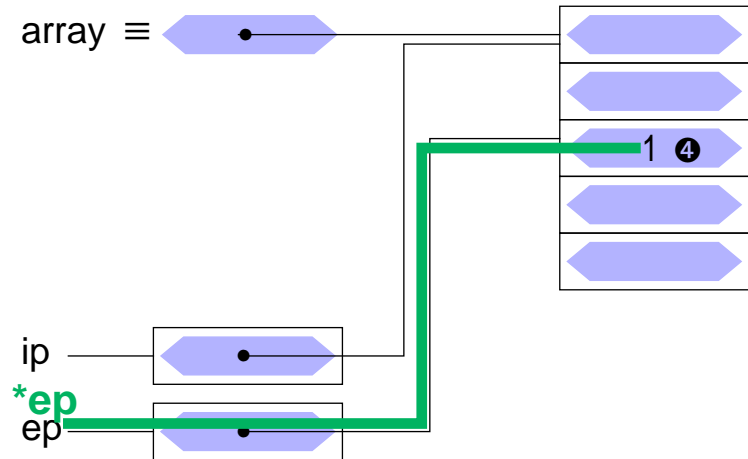
```
int array[5];  
  
int *ip = array;  
  
int *ep;  
ep = &array[0];  
  
ep = &array[2]; ③
```



H.5 Zeiger und Felder (1)

- 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:

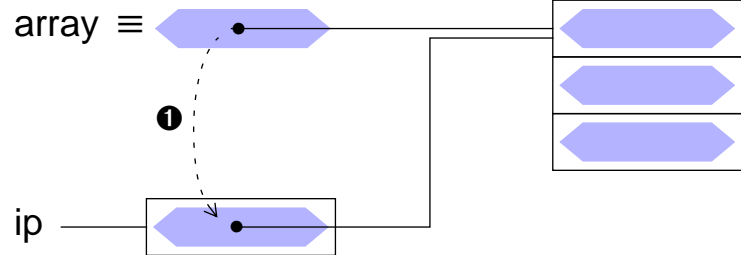
```
int array[5];  
  
int *ip = array;  
  
int *ep;  
ep = &array[0];  
  
ep = &array[2];  
  
*ep = 1; ④
```



H.6 Arithmetik mit Adressen

■ ++ -Operator: Inkrement = nächstes Objekt

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

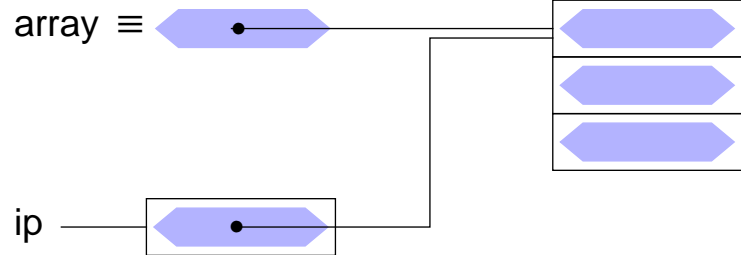


H.6 Arithmetik mit Adressen

■ ++ -Operator: Inkrement = nächstes Objekt

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

```
ip++; ②
```

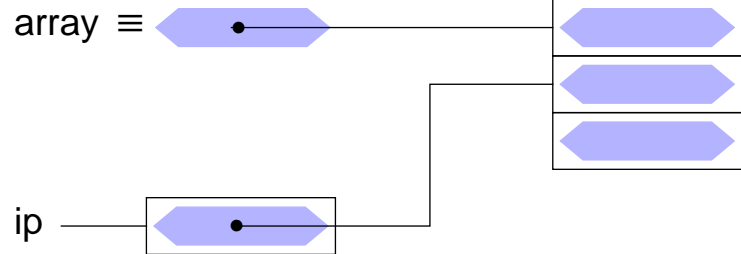


H.6 Arithmetik mit Adressen

■ ++ -Operator: Inkrement = nächstes Objekt

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

```
ip++; ②
```

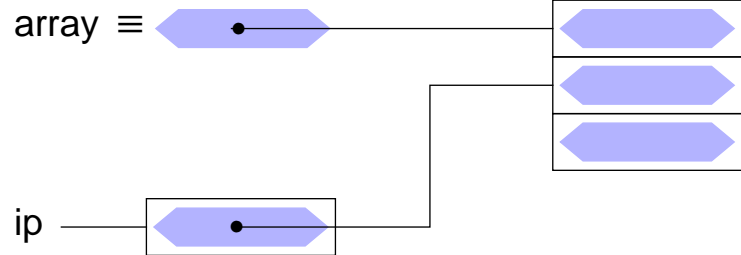


H.6 Arithmetik mit Adressen

■ ++ -Operator: Inkrement = nächstes Objekt

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

```
ip++;  
ip++; ③
```

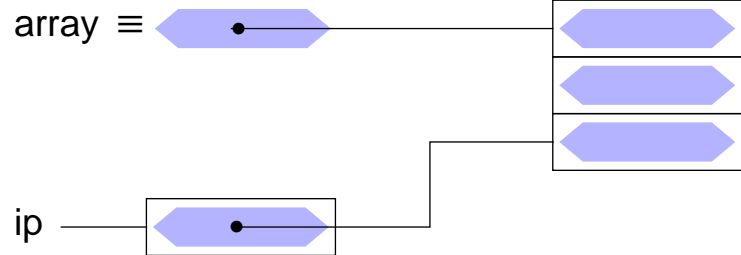


H.6 Arithmetik mit Adressen

■ ++ -Operator: Inkrement = nächstes Objekt

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

```
ip++;  
ip++; ③
```

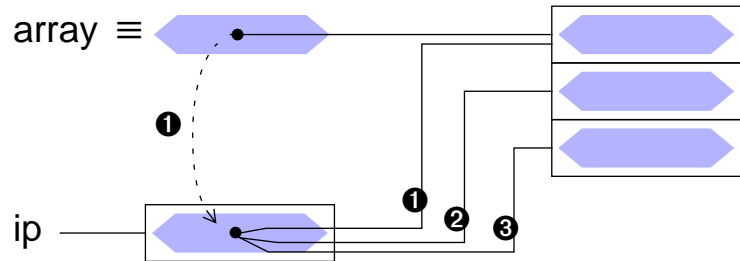


H.6 Arithmetik mit Adressen

- ++ -Operator: Inkrement = nächstes Objekt

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

```
ip++; ❷  
ip++; ❸
```



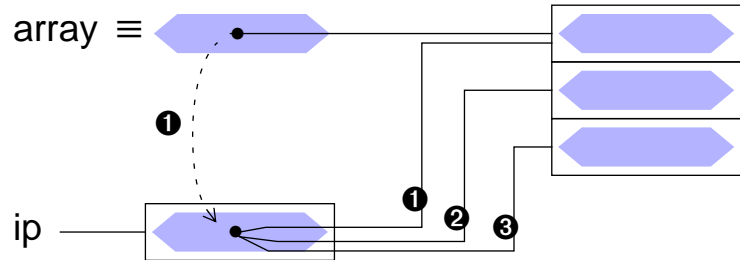
- -- -Operator: Dekrement = vorheriges Objekt

H.6 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.

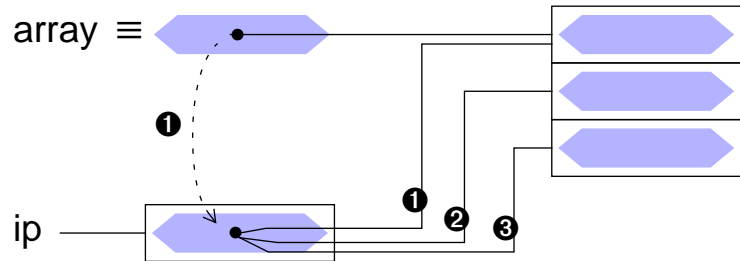
Dabei wird immer die Größe des
Objektyps berücksichtigt!

H.6 Arithmetik mit Adressen

- ++ -Operator: Inkrement = nächstes Objekt

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

```
ip++; ❷  
ip++; ❸
```

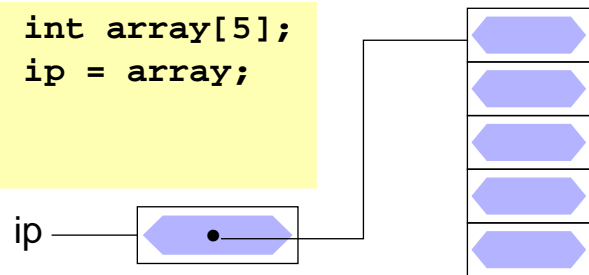


- -- -Operator: Dekrement = vorheriges Objekt

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

Dabei wird immer die Größe des
Objektyps berücksichtigt!

```
int array[5];  
ip = array;
```

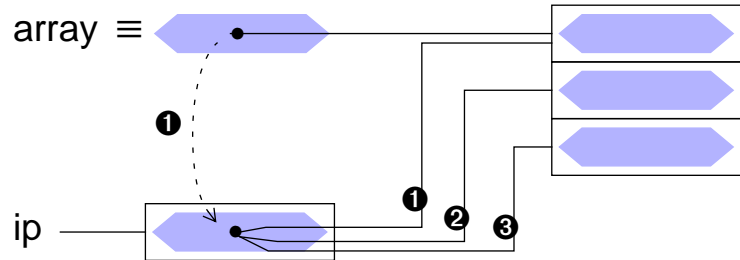


H.6 Arithmetik mit Adressen

- ++ -Operator: Inkrement = nächstes Objekt

```
int array[3];  
int *ip = array; ①
```

```
ip++; ②  
ip++; ③
```

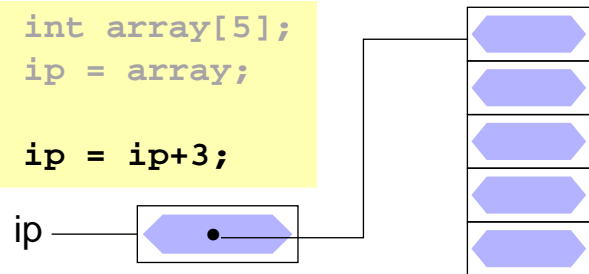


- -- -Operator: Dekrement = vorheriges Objekt

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

Dabei wird immer die Größe des
Objektyps berücksichtigt!

```
int array[5];  
ip = array;  
  
ip = ip+3;
```

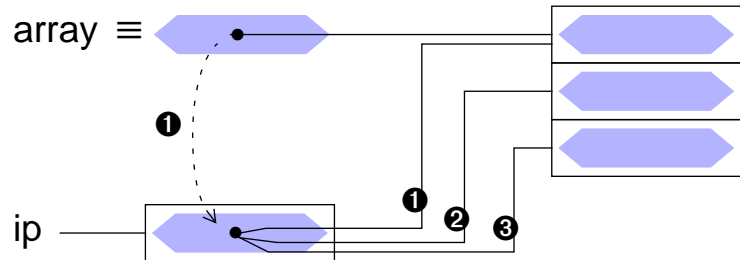


H.6 Arithmetik mit Adressen

- ++ -Operator: Inkrement = nächstes Objekt

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

```
ip++; ❷  
ip++; ❸
```

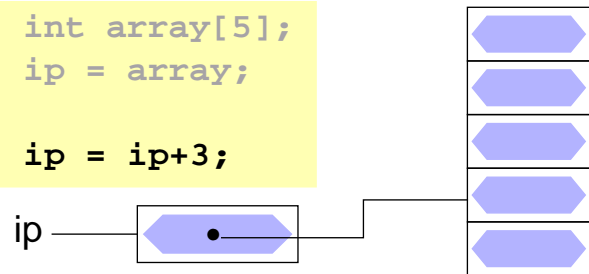


- -- -Operator: Dekrement = vorheriges Objekt

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

Dabei wird immer die Größe des
Objektyps berücksichtigt!

```
int array[5];  
ip = array;  
  
ip = ip+3;
```

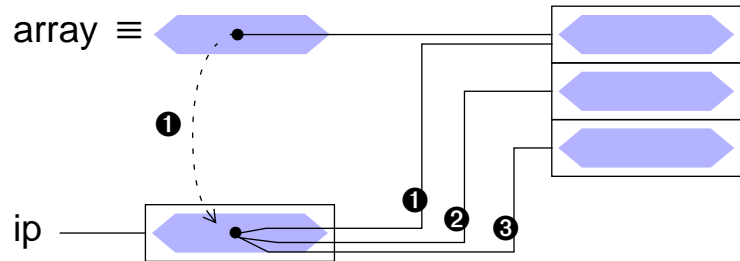


H.6 Arithmetik mit Adressen

- ++ -Operator: Inkrement = nächstes Objekt

```
int array[3];  
int *ip = array; ①
```

```
ip++; ②  
ip++; ③
```

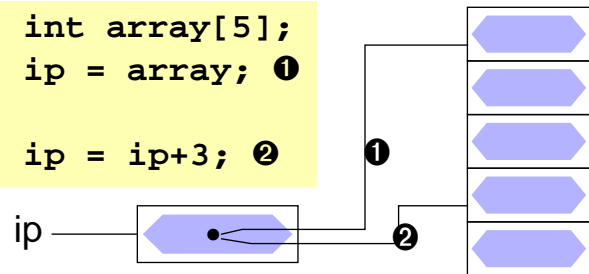


- -- -Operator: Dekrement = vorheriges Objekt

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

Dabei wird immer die Größe des
Objektyps berücksichtigt!

```
int array[5];  
ip = array; ①  
  
ip = ip+3; ②
```



!!! Achtung: Assoziativität der Operatoren beachten !!

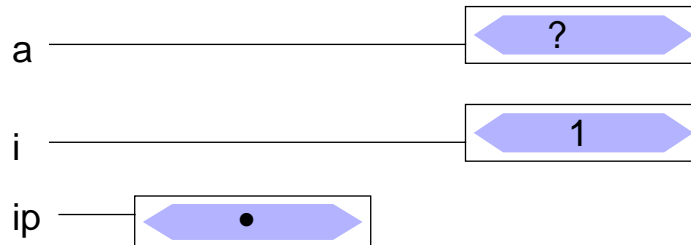
Operatorklasse	Operatoren	Assoziativität
primär	() Funktionsaufruf []	von links nach rechts
unär	! ~ ++ -- + - * &	von rechts nach links
multiplikativ	* / %	von links nach rechts
...		

1 Vorrangregeln bei Operatoren

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

2 Beispiele

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

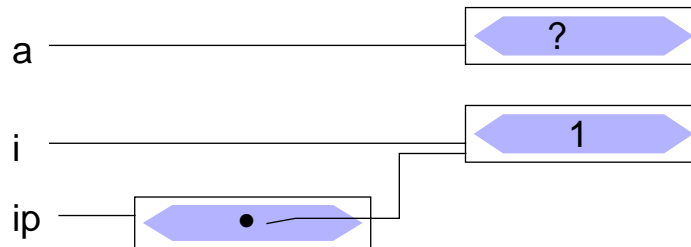


1 Vorrangregeln bei Operatoren

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

2 Beispiele

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

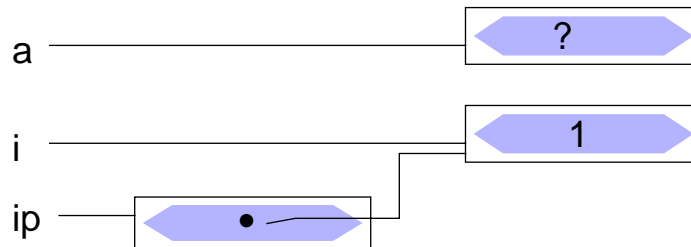


1 Vorrangregeln bei Operatoren

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

2 Beispiele

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



1 Vorrangregeln bei Operatoren

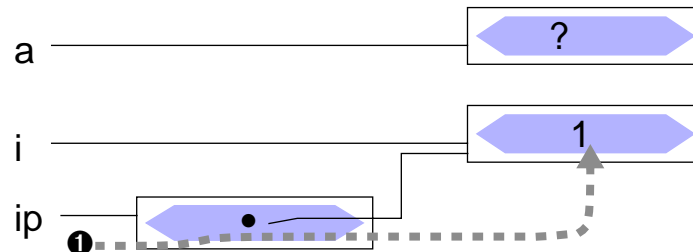
Operatorklasse	Operatoren	Assoziativität
primär	() Funktionsaufruf []	von links nach rechts
unär	! ~ ++ -- + - * &	von rechts nach links
multiplikativ	* / %	von links nach rechts
...		

2 Beispiele

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

a = *ip ;
```

*(Note: In the original image, a bracket under '*ip' in the last line is labeled with a circled '1', indicating the order of operations.)*



1 Vorrangregeln bei Operatoren

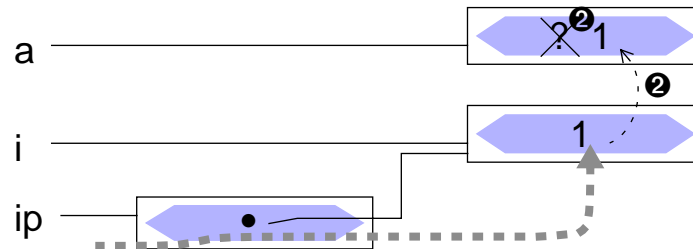
Operatorklasse	Operatoren	Assoziativität
primär	() Funktionsaufruf []	von links nach rechts
unär	! ~ ++ -- + - * &	von rechts nach links
multiplikativ	* / %	von links nach rechts
...		

2 Beispiele

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

a = *ip ;
```

*(Note: In the original image, a bracket underlines `*ip` and a dashed arrow with a circled 2 points from it to the `ip` variable in the line above.)*

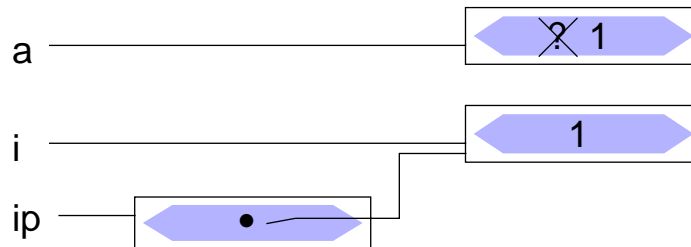


1 Vorrangregeln bei Operatoren

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

2 Beispiele

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



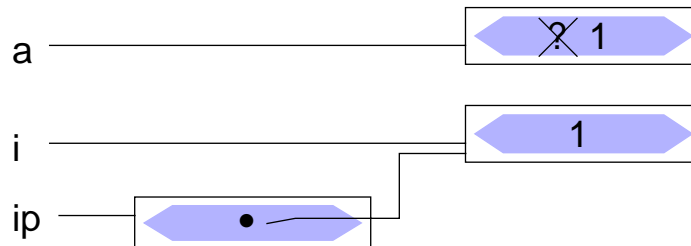
1 Vorrangregeln bei Operatoren

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

2 Beispiele

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

```
    *ip++;
```



1 Vorrangregeln bei Operatoren

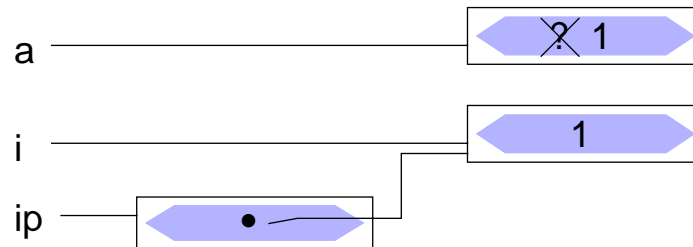
Operatorklasse	Operatoren	Assoziativität
primär	() Funktionsaufruf []	von links nach rechts
unär	! ~ ++ -- + - * &	von rechts nach links
multiplikativ	* / %	von links nach rechts
...		

2 Beispiele

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

ip++;
```

③



1 Vorrangregeln bei Operatoren

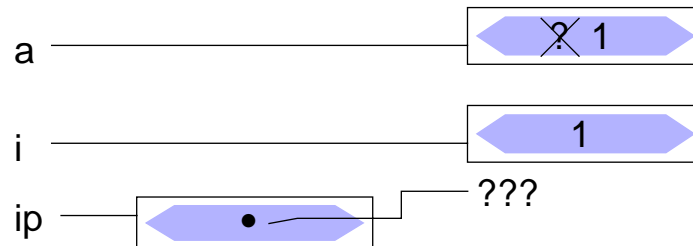
Operatorklasse	Operatoren	Assoziativität
primär	() Funktionsaufruf []	von links nach rechts
unär	! ~ ++ -- + - * &	von rechts nach links
multiplikativ	* / %	von links nach rechts
...		

2 Beispiele

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

ip++;
```

③

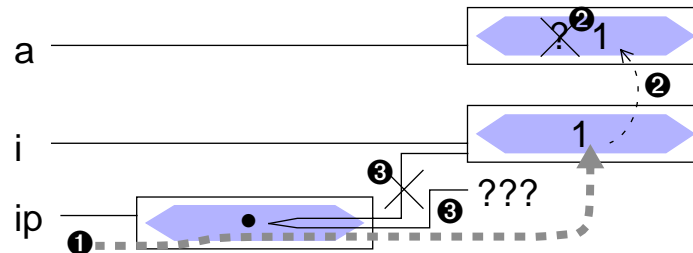


1 Vorrangregeln bei Operatoren

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

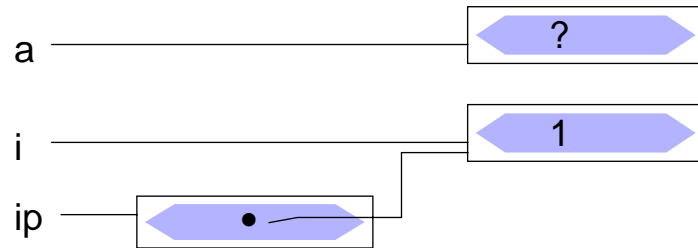
2 Beispiele

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



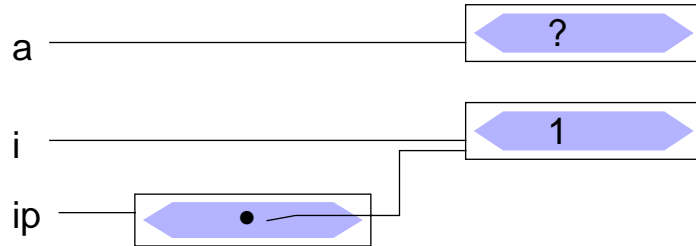
2 Beispiele (2)

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



2 Beispiele (2)

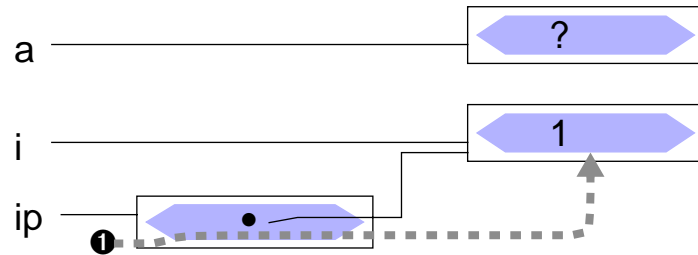
```
int a, i, *ip;  
i = 1;  
ip = &i;  
  
a = ++*ip;
```



2 Beispiele (2)

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

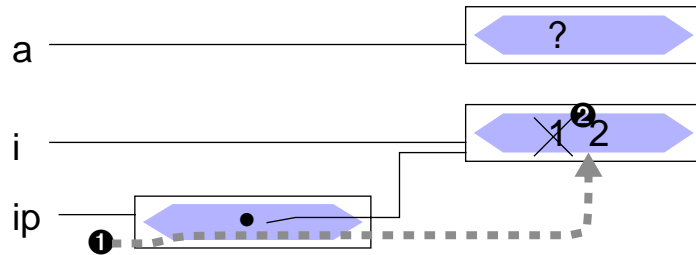
```
+++*ip;  
  1 *ip
```



2 Beispiele (2)

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

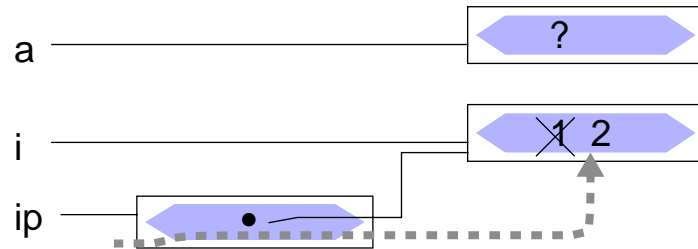
```
++*ip;  
  ↓  
  ① *ip  
  ② ++ → *ip
```



2 Beispiele (2)

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

a = ++*ip;
```

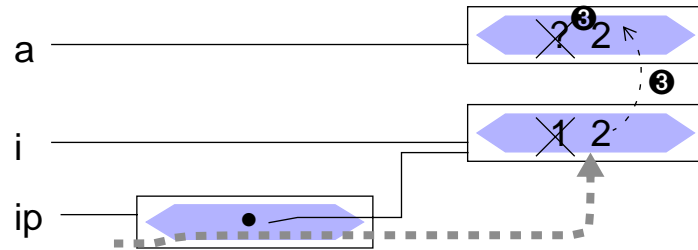


2 Beispiele (2)

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

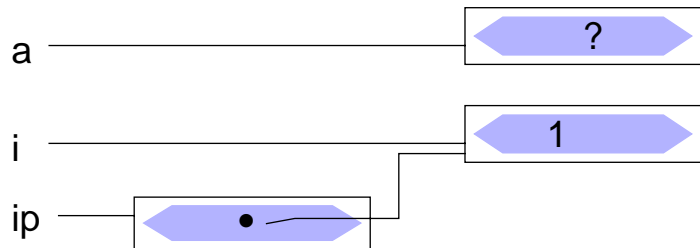
```
a =    *ip;
```

Diagram illustrating the dereferencing operation: a dashed arrow points from the asterisk in `*ip` to the variable `ip`, which is marked with a circled 3. Another dashed arrow points from the `ip` variable to the memory location containing the value 1, also marked with a circled 3.



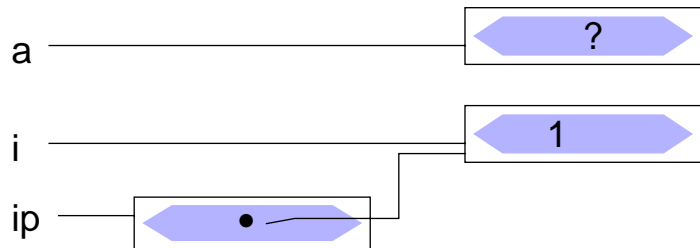
2 Beispiele (2)

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



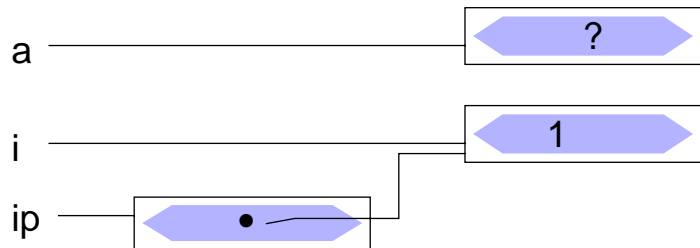
2 Beispiele (2)

```
int a, i, *ip;  
i = 1;  
ip = &i;  
  
a = (*ip)++;
```



2 Beispiele (2)

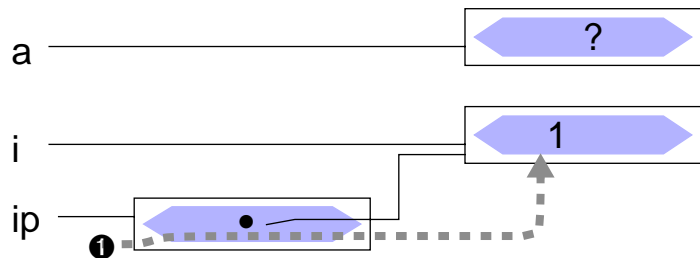
```
int a, i, *ip;  
i = 1;  
ip = &i;  
  
a = (*ip);
```



2 Beispiele (2)

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

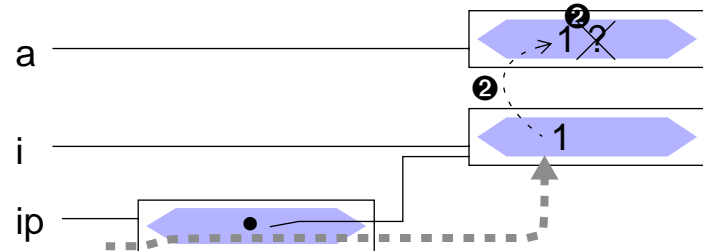
The code snippet is displayed on a yellow background. In the final line, `a = (*ip);`, the expression `(*ip)` is underlined with a bracket, and a circled '1' is placed below the bracket to indicate the value being dereferenced.



2 Beispiele (2)

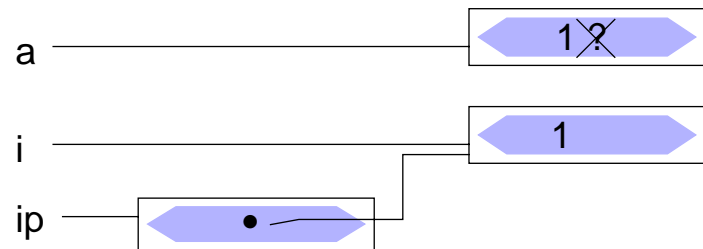
```
int a, i, *ip;  
i = 1;  
ip = &i;
```

```
a = (*ip) ;  
  ↗ ②
```



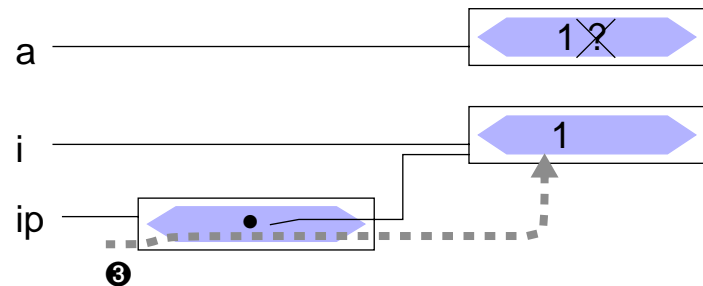
2 Beispiele (2)

```
int a, i, *ip;  
i = 1;  
ip = &i;  
  
a = (*ip)++;
```



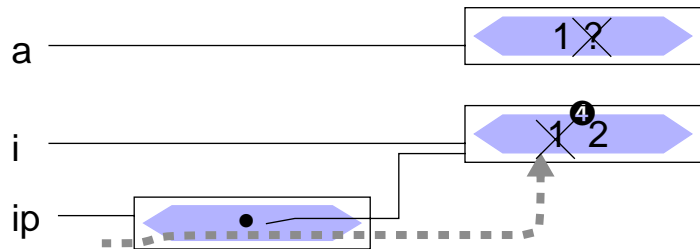
```
int a, i, *ip;  
i = 1;  
ip = &i;
```

```
  (*ip)++;  
  *ip③
```



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

```
  (*ip)++;  
  *ip③  
    ④++ → *ip
```

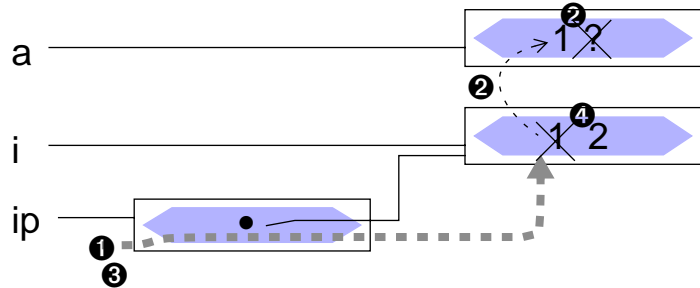


```
int a, i, *ip;  
i = 1;  
ip = &i;
```

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

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

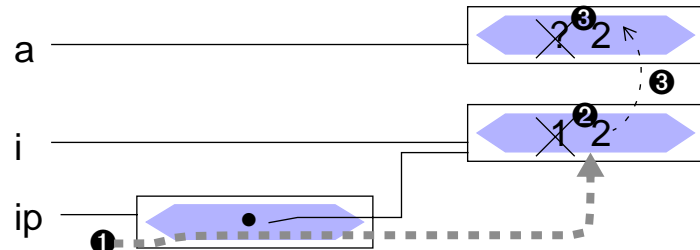
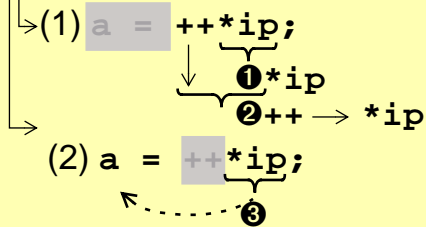
(2) a = (*ip)++;



2 Beispiele (2)

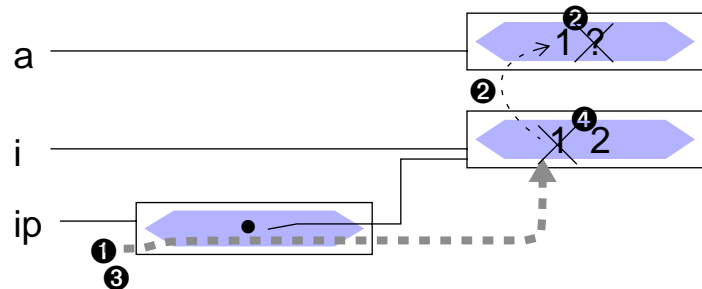
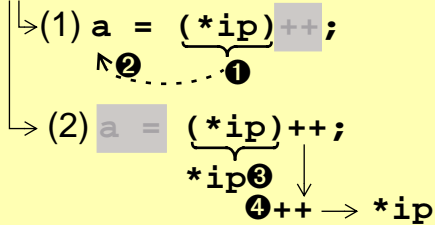
```
int a, i, *ip;
i = 1;
ip = &i;
```

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



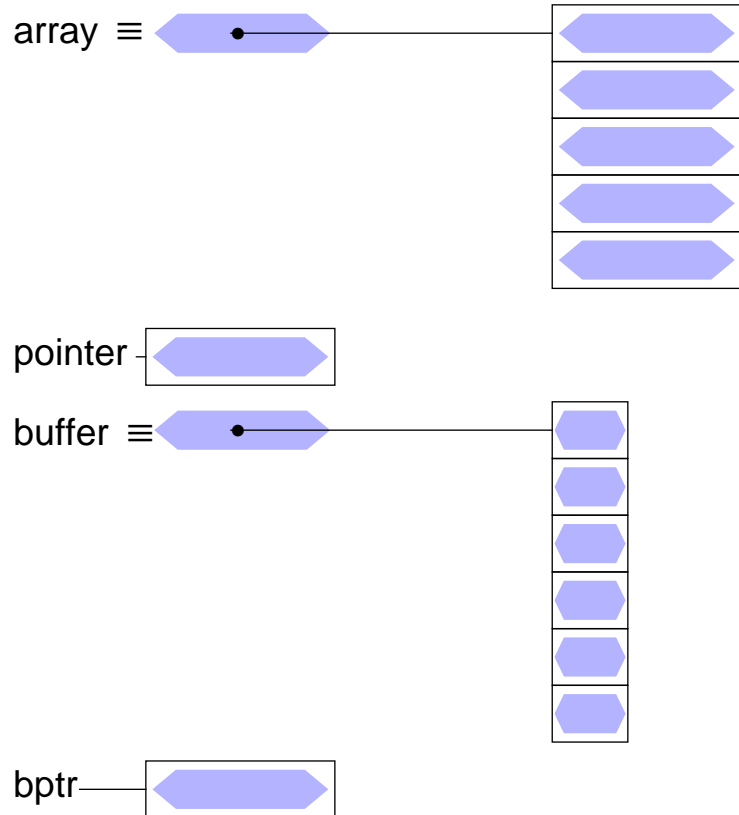
```
int a, i, *ip;
i = 1;
ip = &i;
```

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



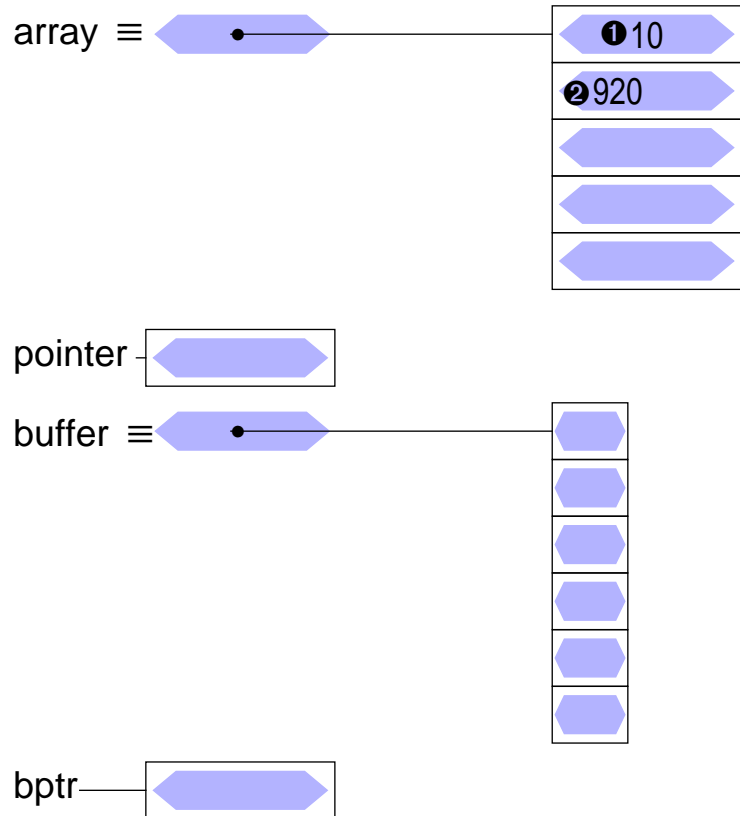
2 Zeigerarithmetik und Felder

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



3 Zeigerarithmetik und Felder

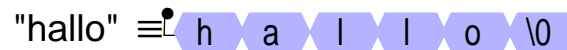
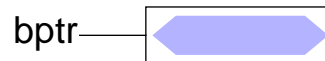
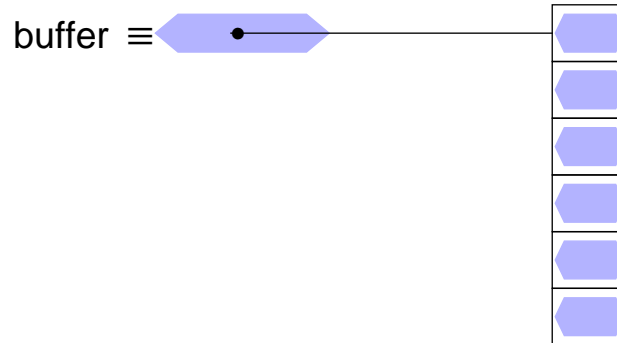
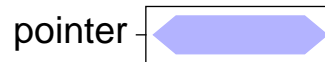
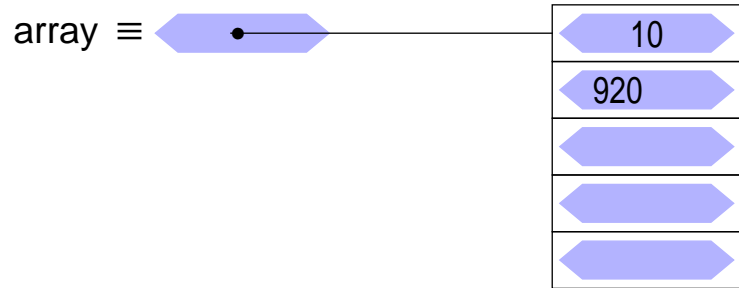
```
int array[5];  
int *pointer;  
char buffer[6];  
char *bptr;  
  
❶ array[0] = 10;  
❷ array[1] = 920;
```



3 Zeigerarithmetik und Felder

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

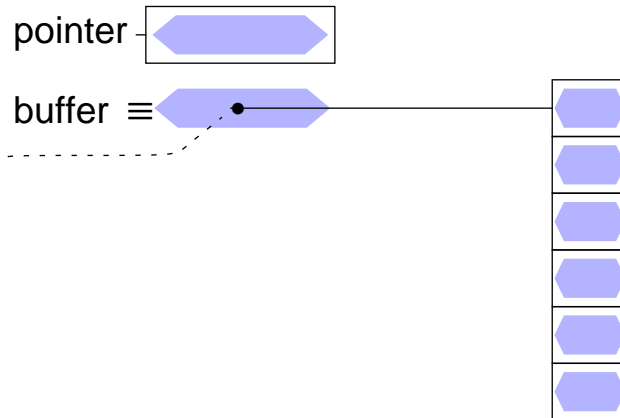
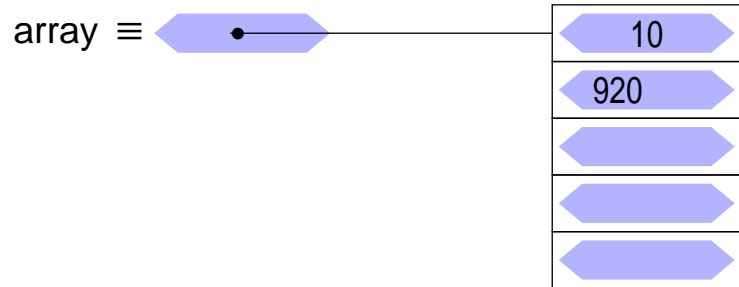
```
① array[0] = 10;  
② array[1] = 920;  
③ strcpy(buffer, "hallo");
```



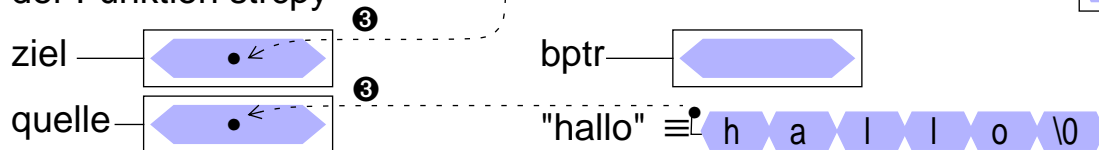
3 Zeigerarithmetik und Felder

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

- ① array[0] = 10;
- ② array[1] = 920;
- ③ strcpy(buffer, "hallo");



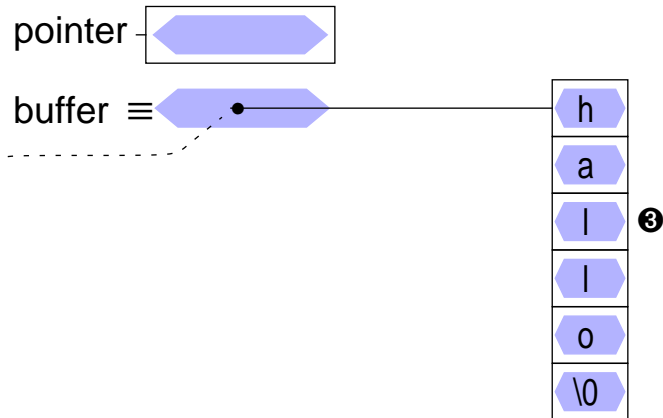
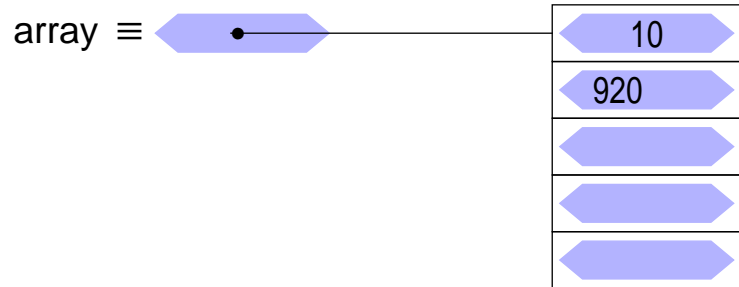
Formale Parameter
der Funktion strcpy



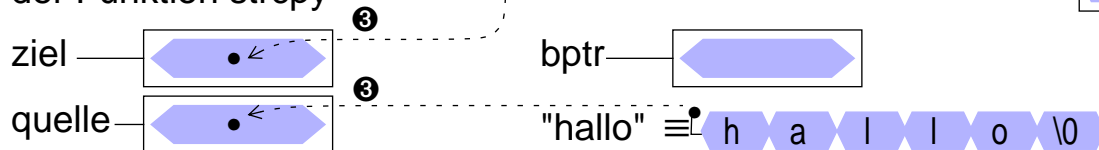
3 Zeigerarithmetik und Felder

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

- ① array[0] = 10;
- ② array[1] = 920;
- ③ strcpy(buffer, "hallo");



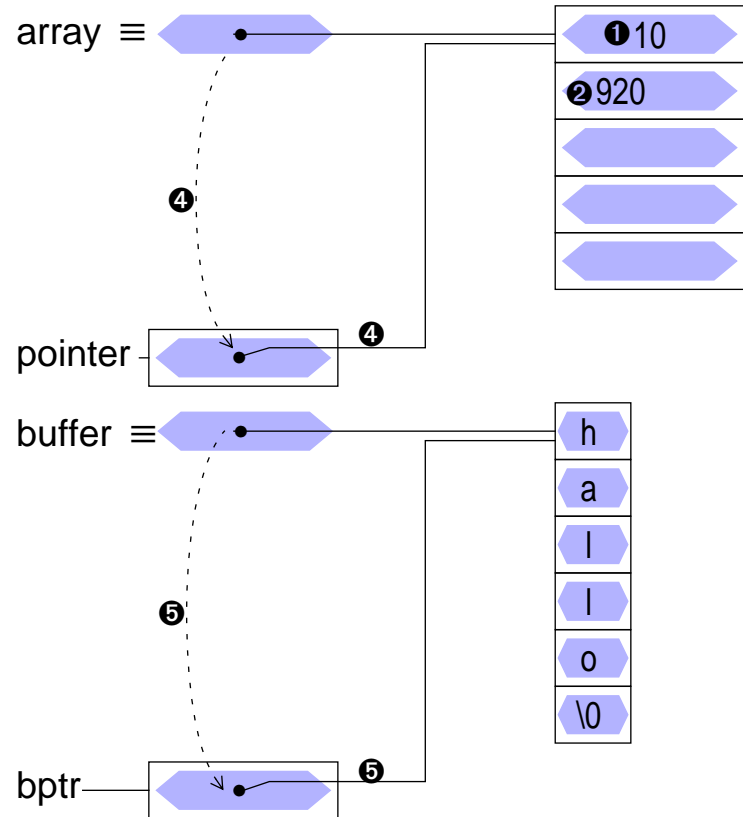
Formale Parameter
der Funktion strcpy



3 Zeigerarithmetik und Felder

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

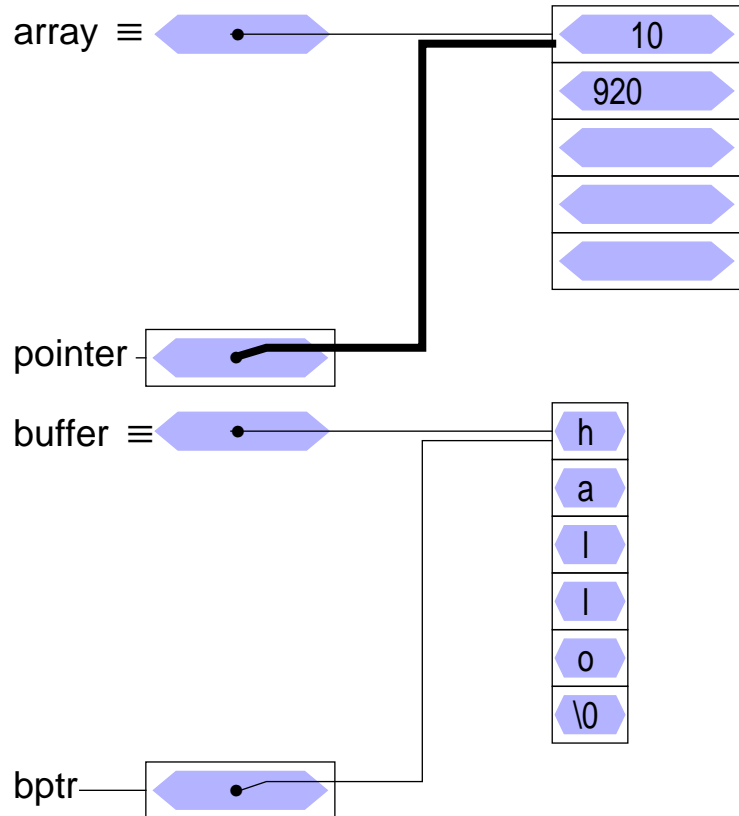
```
① array[0] = 10;  
② array[1] = 920;  
③ strcpy(buffer, "hallo");  
④ pointer = array;  
⑤ bptr = buffer;
```



3 Zeigerarithmetik und Felder

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

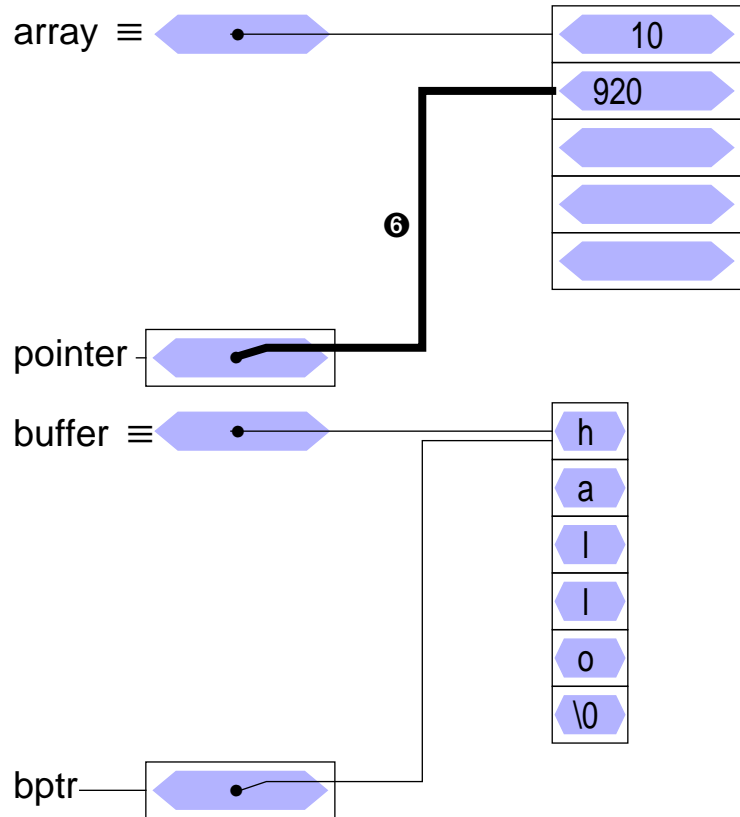
- ① array[0] = 10;
- ② array[1] = 920;
- ③ strcpy(buffer, "hallo");
- ④ pointer = array;
- ⑤ bptr = buffer;
- ⑥ pointer++;



3 Zeigerarithmetik und Felder

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

- ① array[0] = 10;
- ② array[1] = 920;
- ③ strcpy(buffer, "hallo");
- ④ pointer = array;
- ⑤ bptr = buffer;
- ⑥ pointer++;

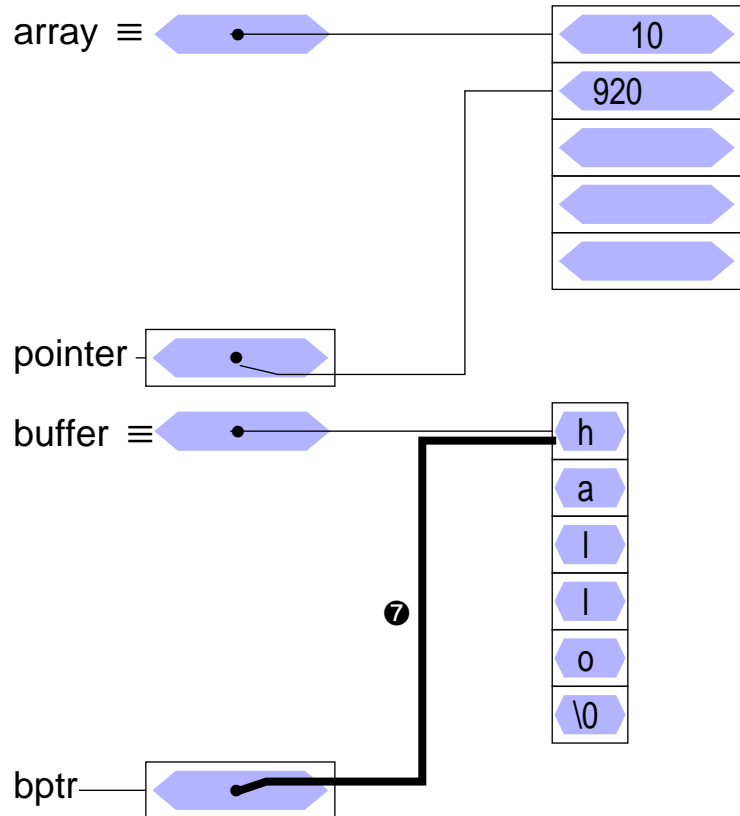


3 Zeigerarithmetik und Felder

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

- ① array[0] = 10;
- ② array[1] = 920;
- ③ strcpy(buffer, "hallo");
- ④ pointer = array;
- ⑤ bptr = buffer;

- ⑥ pointer++;
- ⑦ bptr++;

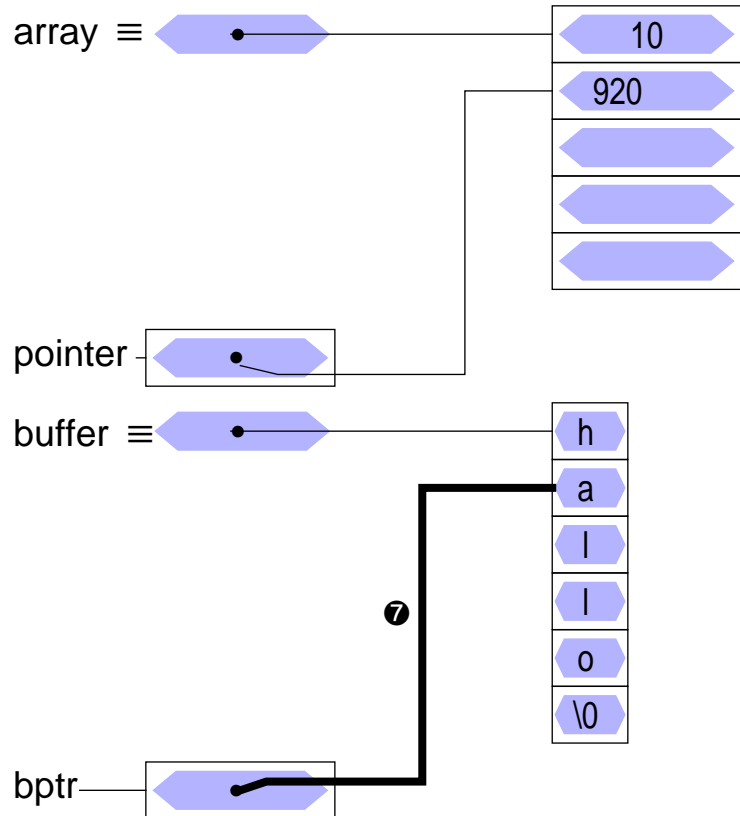


3 Zeigerarithmetik und Felder

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

- ① array[0] = 10;
- ② array[1] = 920;
- ③ strcpy(buffer, "hallo");
- ④ pointer = array;
- ⑤ bptr = buffer;

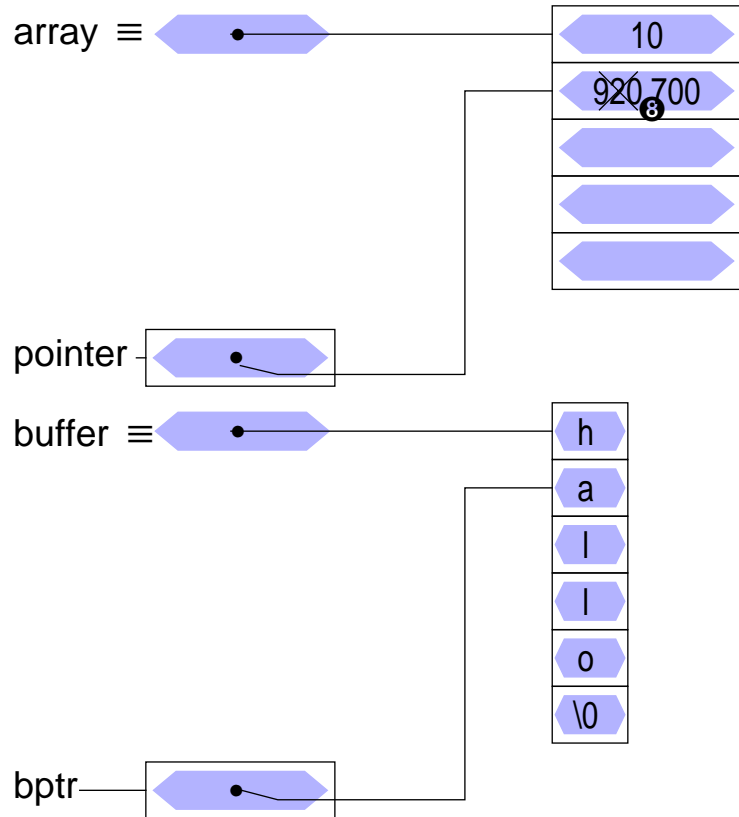
- ⑥ pointer++;
- ⑦ bptr++;



3 Zeigerarithmetik und Felder

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

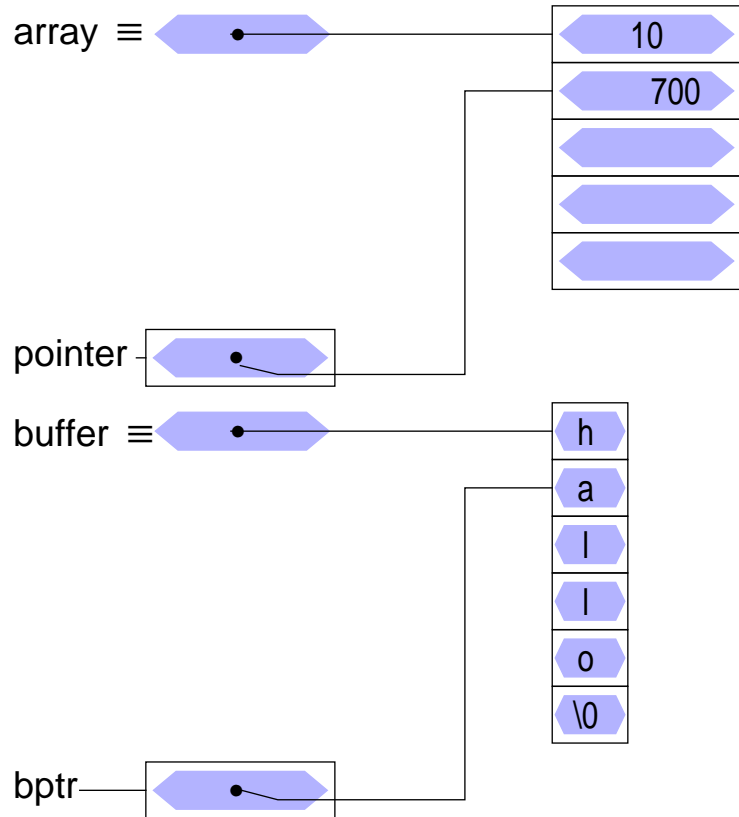
```
① array[0] = 10;  
② array[1] = 920;  
③ strcpy(buffer, "hallo");  
④ pointer = array;  
⑤ bptr = buffer;  
  
⑥ pointer++;  
⑦ bptr++;  
⑧ *pointer = 700;
```



3 Zeigerarithmetik und Felder

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

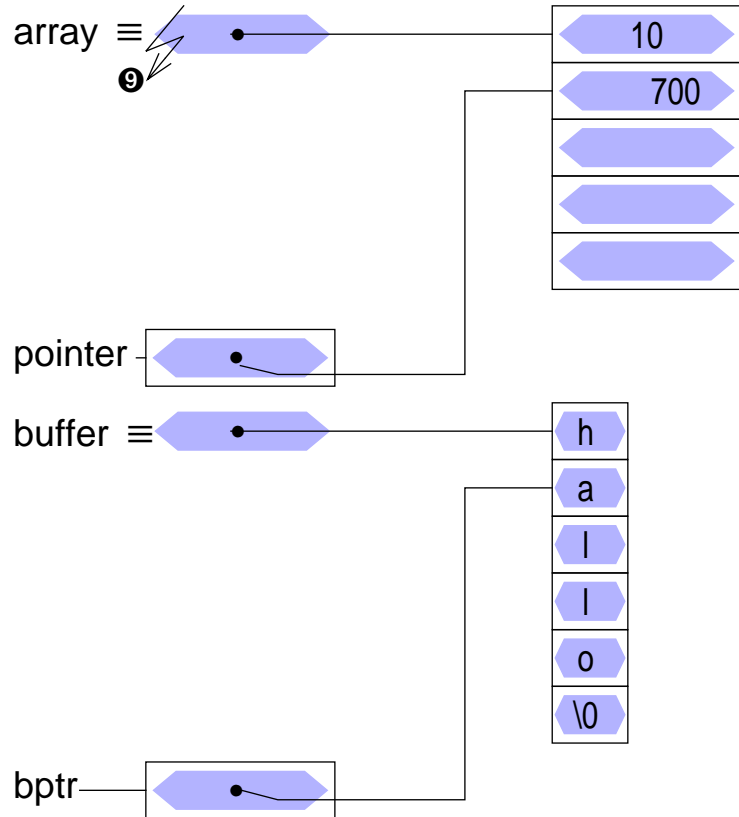
```
① array[0] = 10;  
② array[1] = 700;  
③ strcpy(buffer, "hallo");  
④ pointer = array;  
⑤ bptr = buffer;  
  
⑥ pointer++;  
⑦ bptr++;  
⑧ *pointer = 700;  
  
⑨ array++;
```



3 Zeigerarithmetik und Felder

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

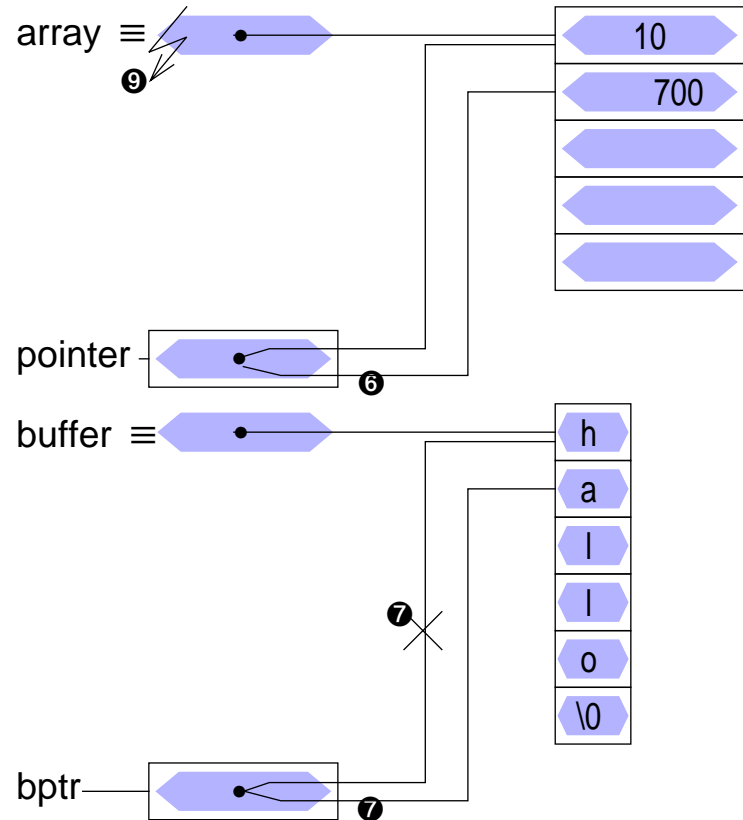
```
① array[0] = 10;  
② array[1] = 920;  
③ strcpy(buffer, "hallo");  
④ pointer = array;  
⑤ bptr = buffer;  
  
⑥ pointer++;  
⑦ bptr++;  
⑧ *pointer = 700;  
  
⑨ array++;
```



3 Zeigerarithmetik und Felder

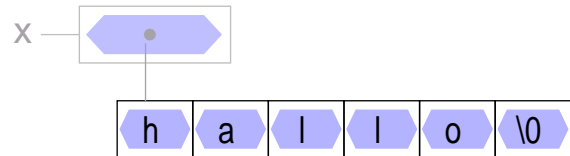
```
int array[5];  
int *pointer;  
char buffer[6];  
char *bptr;
```

```
① array[0] = 10;  
② array[1] = 920;  
③ strcpy(buffer, "hallo");  
④ pointer = array;  
⑤ bptr = buffer;  
  
⑥ pointer++;  
⑦ bptr++;  
⑧ *pointer = 700;  
  
⑨ array++;
```



H.8 Zeiger, Felder und Zeichenketten

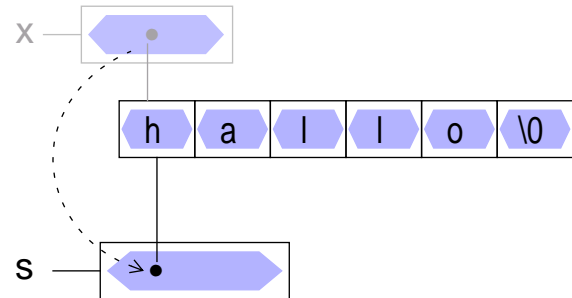
- Zeichenketten sind Felder von Einzelzeichen (`char`), die in der internen Darstellung durch ein `'\0'`-Zeichen abgeschlossen sind
- Beispiel: Länge eines Strings ermitteln — Aufruf `strlen(x)`;



H.8 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)`;

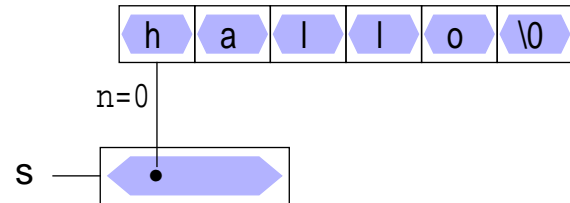
```
/* 1. Version */
int strlen(char *s)
{
    int n;
    for (n=0; *s != '\0'; s++)
        n++;
    return(n);
}
```



H.8 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)`;

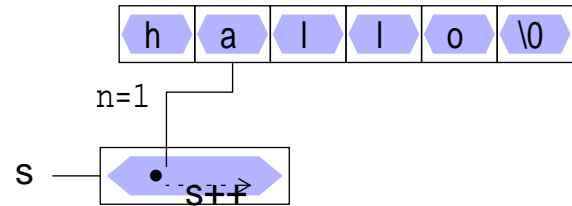
```
/* 1. Version */
int strlen(char *s)
{
    int n;
    for (n=0; *s != '\0'; s++)
        n++;
    return(n);
}
```



H.8 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)`;

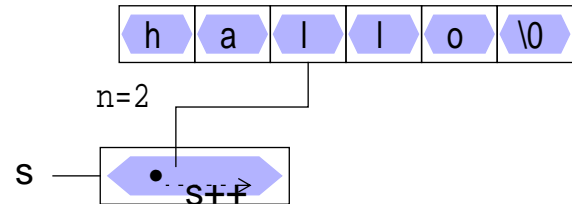
```
/* 1. Version */
int strlen(char *s)
{
    int n;
    for (n=0; *s != '\0'; s++)
        n++;
    return(n);
}
```



H.8 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)`;

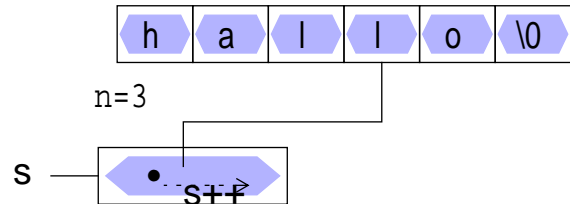
```
/* 1. Version */
int strlen(char *s)
{
    int n;
    for (n=0; *s != '\0'; s++)
        n++;
    return(n);
}
```



H.8 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)`;

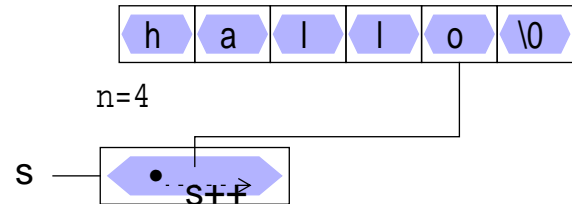
```
/* 1. Version */
int strlen(char *s)
{
    int n;
    for (n=0; *s != '\0'; s++)
        n++;
    return(n);
}
```



H.8 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)`;

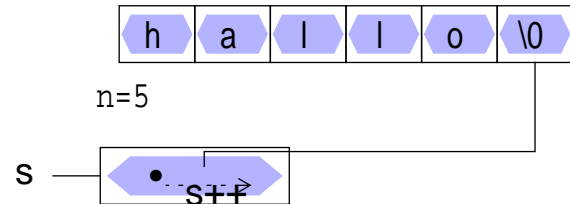
```
/* 1. Version */
int strlen(char *s)
{
    int n;
    for (n=0; *s != '\0'; s++)
        n++;
    return(n);
}
```



H.8 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)`;

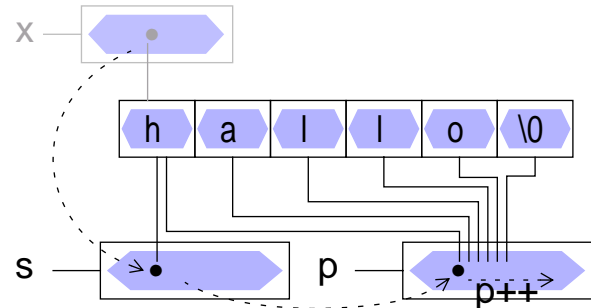
```
/* 1. Version */
int strlen(char *s)
{
    int n;
    for (n=0; *s != '\0'; s++)
        n++;
    return(n);
}
```



H.8 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)`;

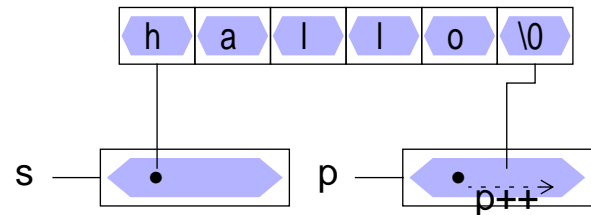
```
/* 2. Version */
int strlen(char *s)
{
    char *p = s;
    while (*p != '\0')
        p++;
    return(p-s);
}
```



H.8 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)`;

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

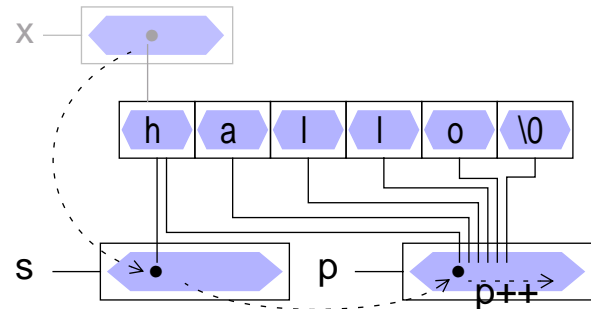
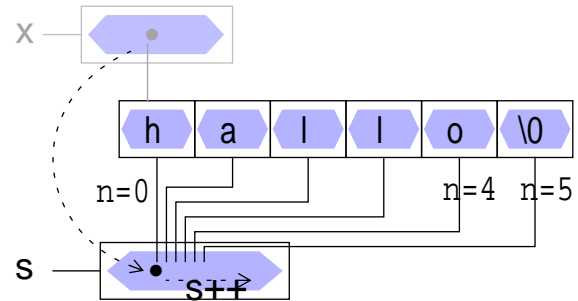


H.8 Zeiger, Felder und Zeichenketten

- 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);
}
```

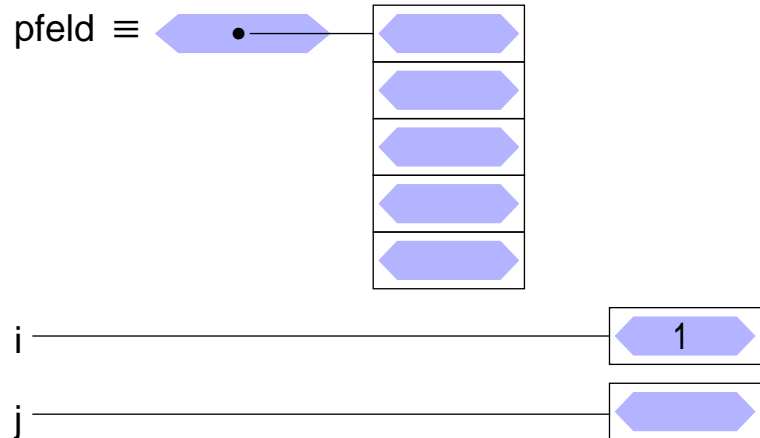
```
/* 2. Version */
int strlen(char *s)
{
    char *p = s;
    while (*p != '\0')
        p++;
    return(p-s);
}
```



H.10 Felder von Zeigern

- Auch von Zeigern können Felder gebildet werden
- Deklaration

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



H.10 Felder von Zeigern

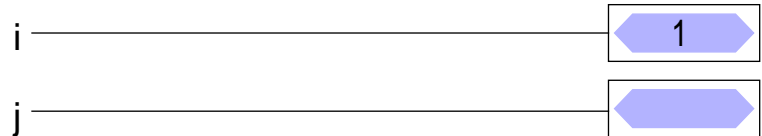
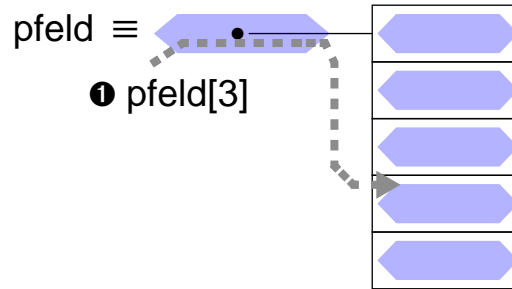
■ Auch von Zeigern können Felder gebildet werden

■ Deklaration

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

■ Zugriffe auf einen Zeiger des Feldes

```
pfeld[3] = &i;  
①
```



H.10 Felder von Zeigern

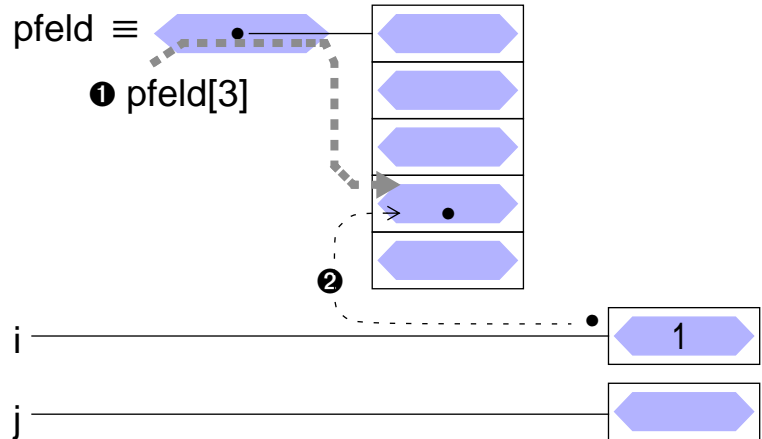
■ Auch von Zeigern können Felder gebildet werden

■ Deklaration

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

■ Zugriffe auf einen Zeiger des Feldes

```
pfeld[3] = &i; ②  
①
```



H.10 Felder von Zeigern

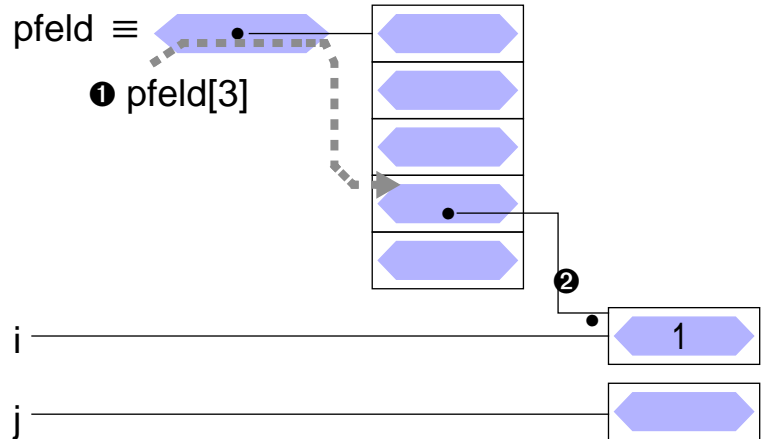
■ Auch von Zeigern können Felder gebildet werden

■ Deklaration

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

■ Zugriffe auf einen Zeiger des Feldes

```
pfeld[3] = &i; ②  
①
```



H.10 Felder von Zeigern

- Auch von Zeigern können Felder gebildet werden

- Deklaration

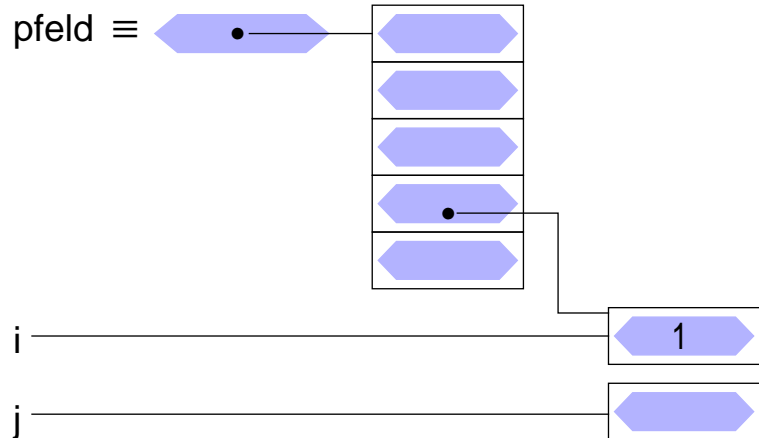
```
int *pfeld[5];  
int i = 1  
int j;
```

- Zugriffe auf einen Zeiger des Feldes

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

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

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



H.10 Felder von Zeigern

■ Auch von Zeigern können Felder gebildet werden

■ Deklaration

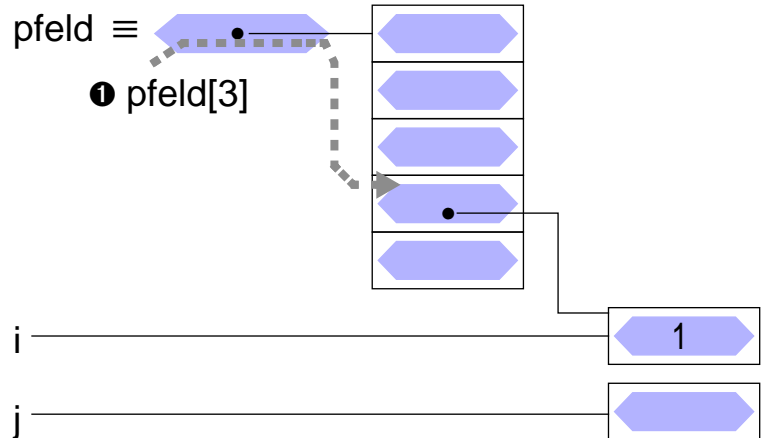
```
int *pfeld[5];  
int i = 1  
int j;
```

■ Zugriffe auf einen Zeiger des Feldes

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

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

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



H.10 Felder von Zeigern

■ Auch von Zeigern können Felder gebildet werden

■ Deklaration

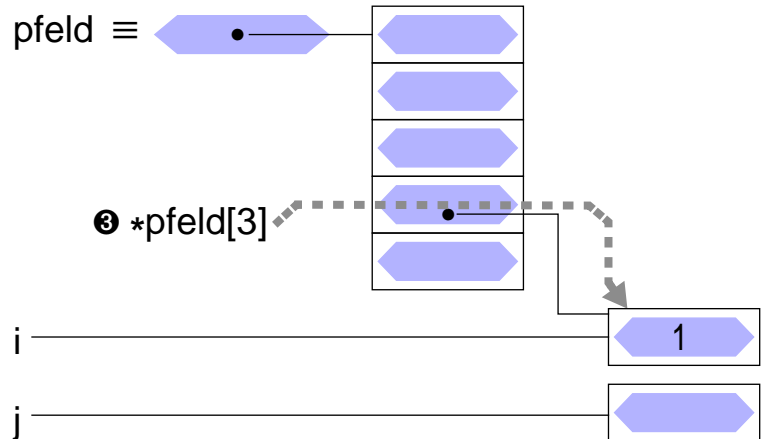
```
int *pfeld[5];  
int i = 1  
int j;
```

■ Zugriffe auf einen Zeiger des Feldes

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

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

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



H.10 Felder von Zeigern

■ Auch von Zeigern können Felder gebildet werden

■ Deklaration

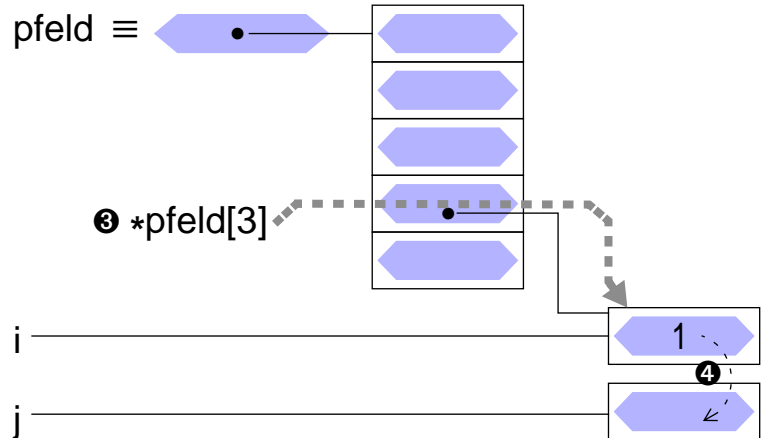
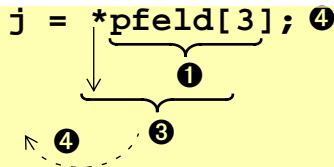
```
int *pfeld[5];  
int i = 1  
int j;
```

■ Zugriffe auf einen Zeiger des Feldes

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

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

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



H.10 Felder von Zeigern

■ Auch von Zeigern können Felder gebildet werden

■ Deklaration

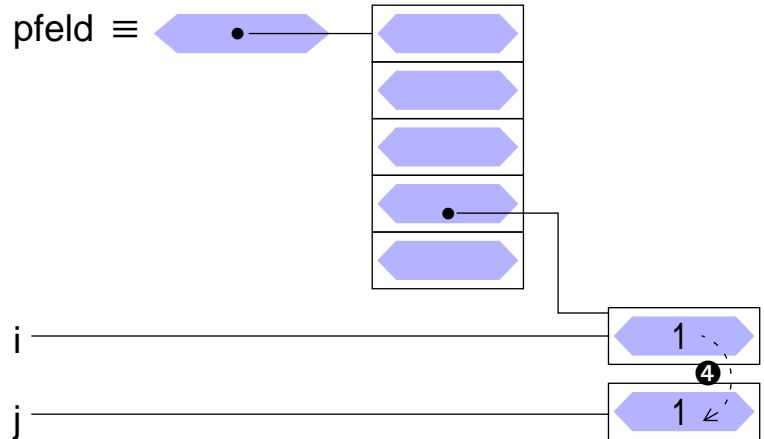
```
int *pfeld[5];  
int i = 1  
int j;
```

■ Zugriffe auf einen Zeiger des Feldes

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

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

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



H.10 Felder von Zeigern

■ Auch von Zeigern können Felder gebildet werden

■ Deklaration

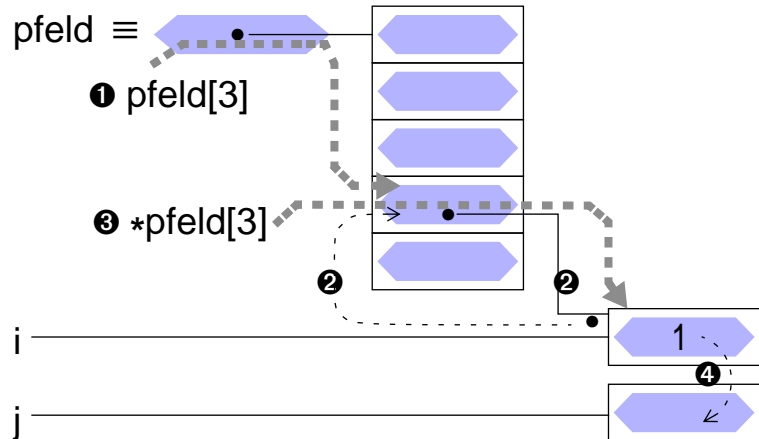
```
int *pfeld[5];  
int i = 1  
int j;
```

■ Zugriffe auf einen Zeiger des Feldes

```
pfeld[3] = &i; ②
```

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

```
j = *pfeld[3]; ④
```



H.11 Argumente aus der Kommandozeile



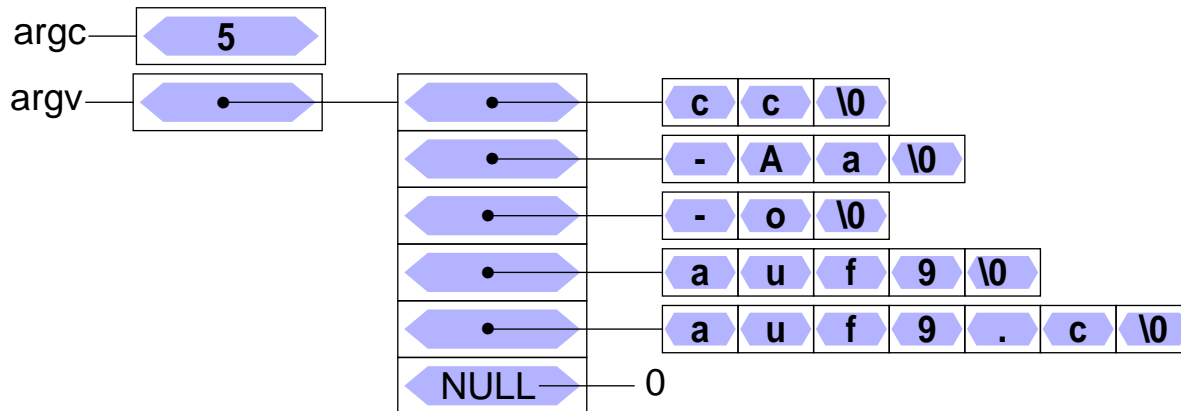
2 Zugriff

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



2 Zugriff

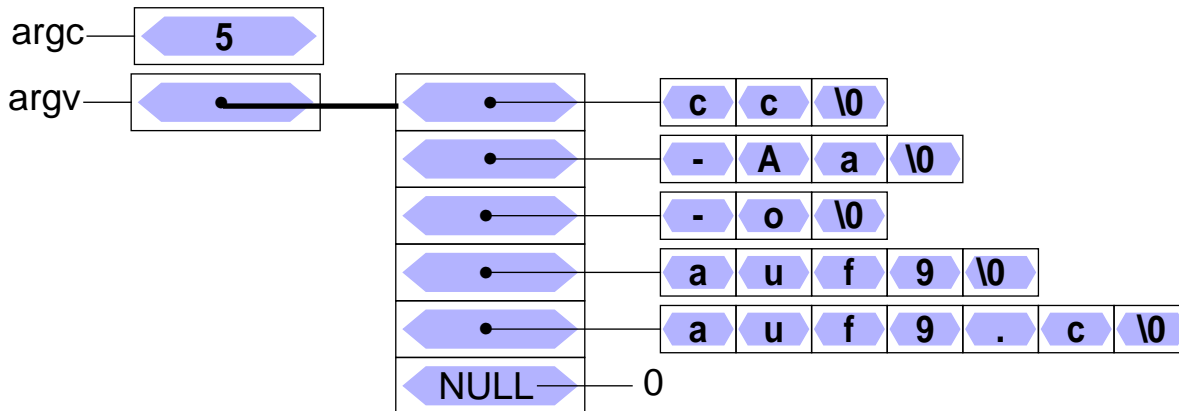
Beispiel: Ausgeben aller Argumente (2)

- das folgende Programmstück gibt alle Argumente der Kommandozeile aus (außer dem Kommandonamen)

```
int  
main (int argc, char **argv)  
{  
    while (--argc > 0) {  
        argv++;  
        printf("%s%c", *argv, (argc>1) ? ' ' : '\n' );  
    }  
    ...  
}
```

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

2. Version



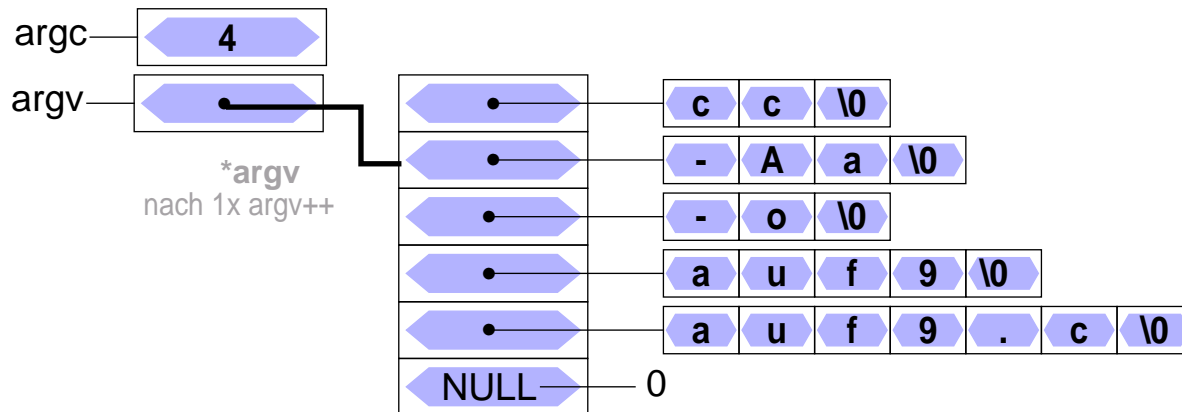
2 Zugriff

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



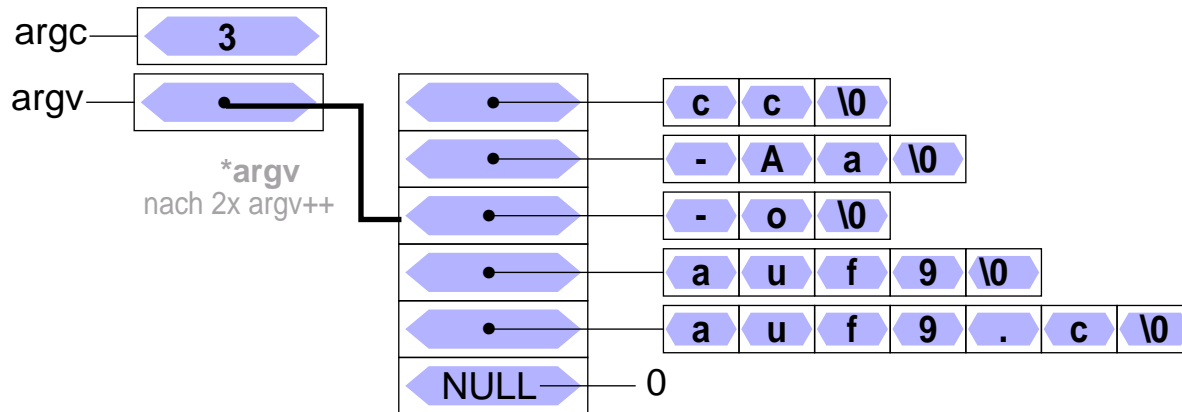
2 Zugriff

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



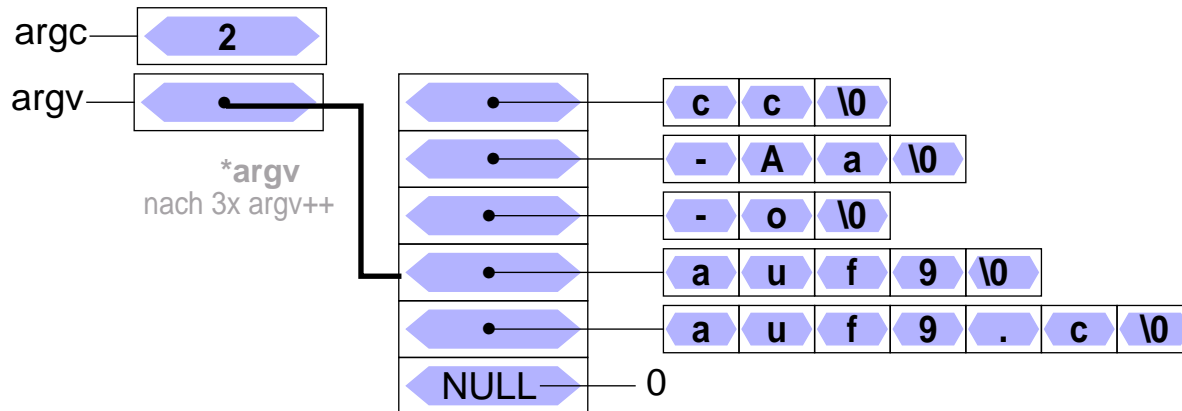
2 Zugriff

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



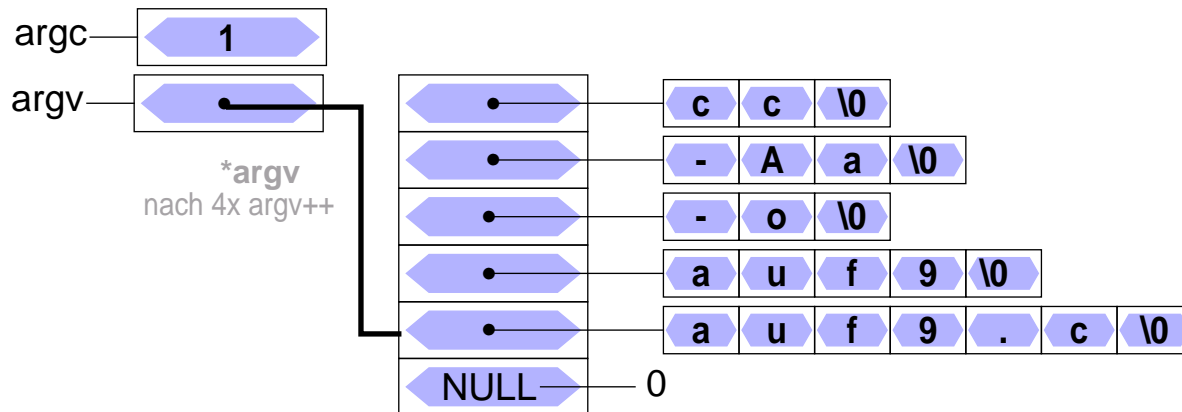
2 Zugriff

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



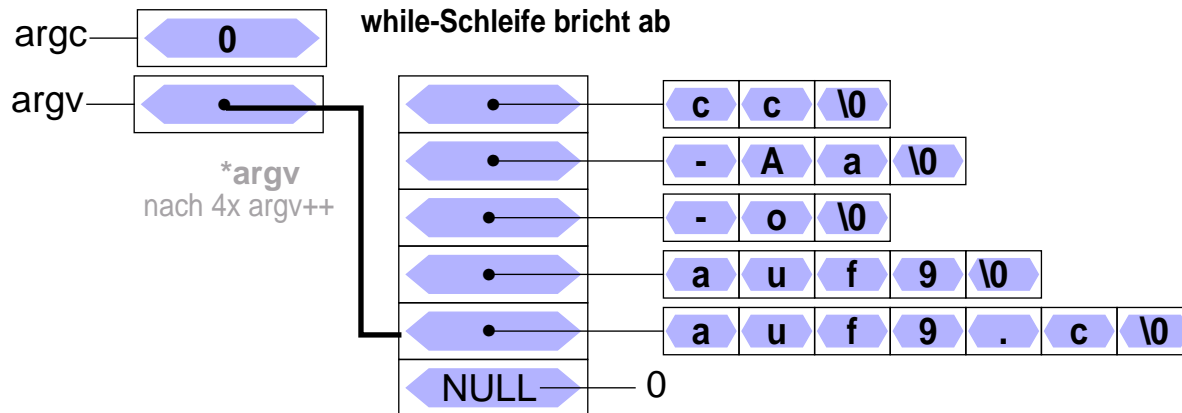
2 Zugriff

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



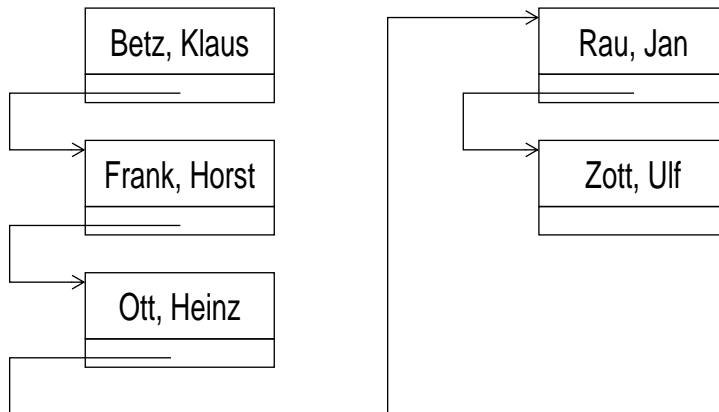
H.16 Rekursive Strukturen (3)

■ Problem:

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

Lösung 2: verkettete Liste von dynamisch angeforderten Strukturen

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



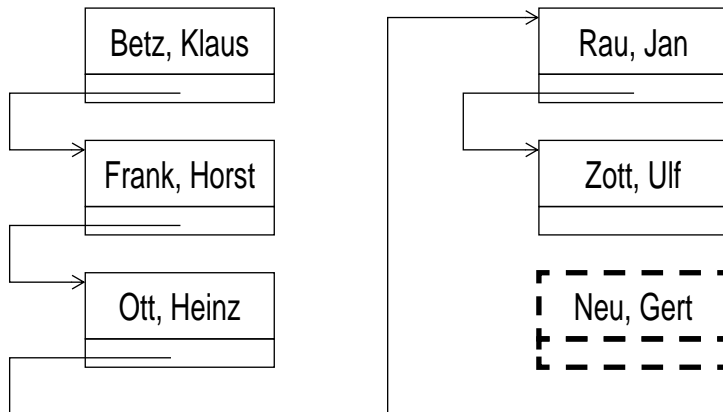
H.16 Rekursive Strukturen (3)

■ Problem:

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

Lösung 2: verkettete Liste von dynamisch angeforderten Strukturen

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



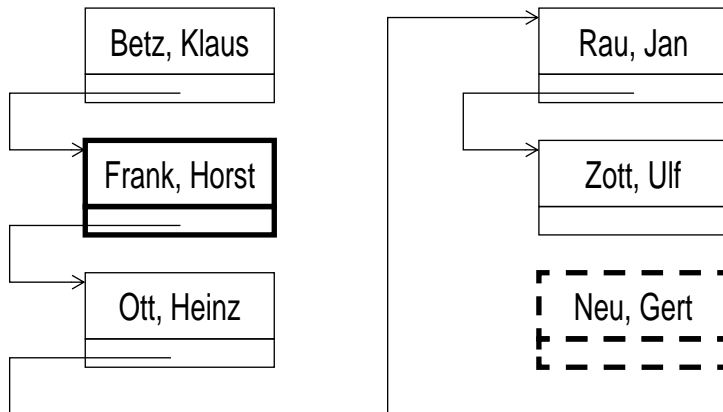
H.16 Rekursive Strukturen (3)

■ Problem:

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

Lösung 2: verkettete Liste von dynamisch angeforderten Strukturen

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



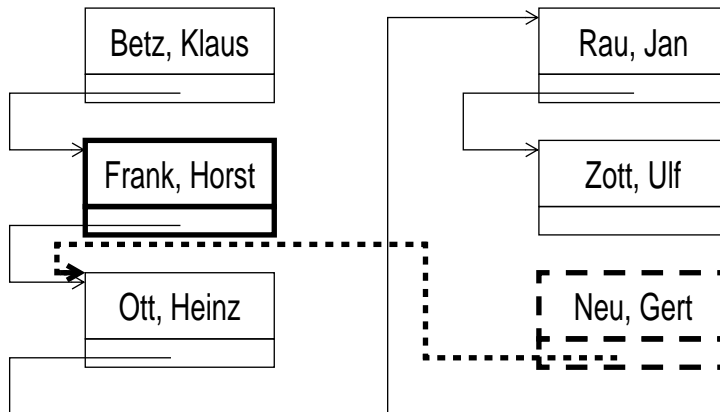
H.16 Rekursive Strukturen (3)

■ Problem:

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

Lösung 2: verkettete Liste von dynamisch angeforderten Strukturen

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



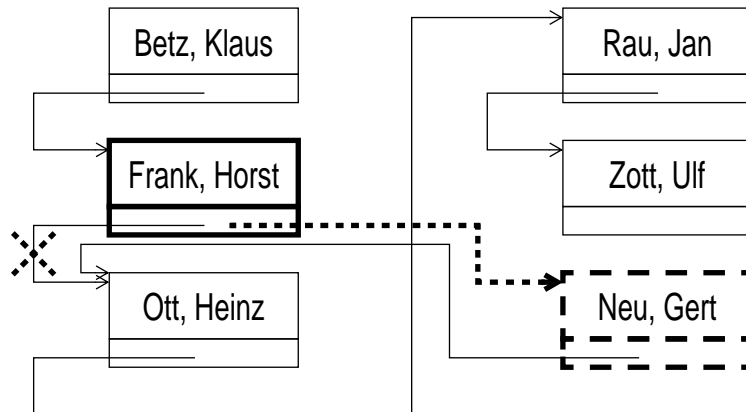
H.16 Rekursive Strukturen (3)

■ Problem:

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

Lösung 2: verkettete Liste von dynamisch angeforderten Strukturen

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



H.16 Rekursive Strukturen (3)

■ Problem:

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

Lösung 2: verkettete Liste von dynamisch angeforderten Strukturen

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

