Vorlesung

Systemnahe Programmierung in C

—

Grundlagen der systemnahen Programmierung in C

Sommer 2010

SPiC

A Überblick über die Lehrveranstaltung

A.1 Thema: Systemnahe Programmierung in C

- Grundlagen (GSPiC und SPiC)
  - ➤ Grundzüge von Systemarchitekturen
  - ➤ Einführung in die Programmiersprache C
  - ➤ Mikrocontroller-Überblick, AVR-Architektur
  - ➤ Programmiersprache C: Zeiger, Felder, Strukturen
  - ➤ Interruptverarbeitung und Nebenläufigkeit in Programmen
  - ➤ Betriebssystemunterstützung zur Programmausführung
  - ➤ Speichermodelle: virtuelle Adressräume / physikalischer Speicher
- Vertiefung (nur SPiC)
  - ➤ Programmiersprache C: weitere Vertiefung
  - ➤ Systemschnittstelle UNIX/Linux: Dateisystem und Prozesse

A.2 Aufbau der Lehrveranstaltung

1 Vorlesung

- Überblick über grundlegende Konzepte von systemnaher (= Betriebssystem-naher oder Hardware-naher) Programmierung
- Einführung in die Programmiersprache C
- C-Programmierung "auf der nackten Hardware" (am Beispiel AVR-µC)
  - ➤ Abbildung Speicher ↔ Sprachkonstrukte
  - ➤ Interrupts
  - ➤ Nebenläufigkeit
- C-Programmierung "auf einem Betriebssystem" (am Beispiel Linux)
  - ➤ Gegensatz µC-Umgebung - Betriebssystem
  - ➤ Betriebssystem als Ausführungsumgebung für Programme
  - ➤ Abstraktionen und Dienste eines Betriebssystems

2 Übungen

- Praktische Umsetzung des Vorlesungsstoffs anhand von einigen kleinen Programmieraufgaben
- Tafelübungen:
  - ➤ Hinweise zur Durchführung der Übungsaufgaben
  - ➤ erste Anleitung
  - ➤ Besprechung von Lösungen
  - ➤ Betreuung bei der Bearbeitung der Programmieraufgaben am Rechner
- Rechnerübungen:
  - ➤ selbstständige Programmierung
    - – auf einer kleinen Mikrocontroller-Plattform
    - – unter Linux
  - ➤ Umgang mit Entwicklungswerkzeugen und Betriebssystemen zur Programmentwicklung
    - – Editor, Compiler, AVR-Studio, Windows, Linux
  - ➤ Hilfestellung bei Problemen durch Übungsbetreuer

B Organisatorisches

- Zwei Varianten
- **GSPiC** Grundlagen der systemnahen Programmierung in C
  - Vorlesung + Übung
  - 2 SWS / 2,5 ECTS
  - Studienfächer: EEI
- **SPiC** Systemnahe Programmierung in C
  - Vorlesung + Übung
  - 4 SWS / 5 ECTS
  - Studienfächer: **Mechatronik**, Mathematik, Technomathematik
    - Mathematik, Technomathematik:
      Alternativ kann auch Systemprogrammierung (SysProg) belegt werden!
- Vorlesungen 1 und 2 gemeinsam

B.1 Semesterüberblick GSPiC

KW	Mo	Di	Mi	Do	Fr	Themen / Kapitel
16	19.04.	20.04. VL 1	21.04. VL 2	22.04.	23.04.	*Organisatorisches, Sprachüberblick, Datentypen, Operatoren A, B, C, D.1 – D.5*
17	26.04.	27.04.	28.04. VL 3	29.04. A1 (Blink)	30.04.	*Programmaufbau, Kontrollstrukturen D.6 – D.7*
18	03.05.	04.05.	05.05. VL 4	06.05. A2 (Snake)	07.05.	*Funktionen, Programmstruktur und Module D.8 – D.9*
19	10.05.	11.05.	12.05.	13.05.	14.05.	
20	17.05.	18.05.	19.05. VL 5	20.05.	21.05.	*Module, Übersetzen und Binden, Mikrocontroller-Programmierung D.9 – D.10, E*
21	24.05. *Pfingsten/Berg*	25.05.	26.05.	27.05. A3 (LED)	28.05.	
22	31.05.	01.06.	02.06. VL 6	03.06.	04.06.	*Zeiger und Felder F.1 – F.11*

B.1 Semesterüberblick GSPiC (Fortsetzung)

KW	Mo	Di	Mi	Do	Fr	Themen / Kapitel
23	07.06.	08.06.	09.06.	10.06.	11.06.	*Verbund- und Aufzählungstypen* *F.12 – F.16*
			VL 7	A4 (pLED)		
24	14.06.	15.06.	16.06.	17.06.	18.06.	
25	21.06.	22.06.	23.06.	24.06.	25.06.	*Nebenläufigkeit, Betriebssysteme* *G*
			VL 8	A5 (Ampel)		
26	28.06.	29.06.	30.06.	01.07.	02.07.	*Speicherorganisation, Stapelaufbau* *H*
			VL 9			
27	05.07.	06.07.	07.07.	08.07.	09.07.	*Dateisysteme* *I*
			VL 10	A6 (PrintDir)		
28	12.07.	13.07.	14.07.	15.07.	16.07.	
				Wiederholung		
29	19.07.	20.07.	21.07.	22.07.	23.07.	

B.2 Semesterüberblick SPiC

KW	Mo	Di	Mi	Do	Fr	Themen / Kapitel
16	19.04.	20.04.	21.04.	22.04.	23.04.	*Organisatorisches, Sprachüberblick, Datentypen, Operatoren* *A, B, C, D.1 – D.5*
	VL1	VL2				
17	26.04.	27.04.	28.04.	29.04.	30.04.	*Programmaufbau, Kontrollstrukturen* *D.6 – D.7*
		VL3		A1 (Blink)		
18	03.05.	04.05.	05.05.	06.05.	07.05.	*Funktionen, Programmstruktur und Module* *D.8 – D.9*
		VL4		A2 (Snake)		
19	10.05.	11.05.	12.05.	13.05.	14.05.	*Module, Übersetzen und Binden, Mikrocontroller-Programmierung* *D.9 – D.10, E*
		VL5		A3 (LED)		
20	17.05.	18.05.	19.05.	20.05.	21.05.	*Zeiger und Felder* *F.1 – F.11*
		VL6				
21	24.05.	25.05.	26.05.	27.05.	28.05.	
	Pfingsten			A4 (pLED)		
22	31.05.	01.06.	02.06.	03.06.	04.06.	*Verbund- und Aufzählungstypen* *F.12 – F.16*
		VL7		A5 (Ampel)		

B.2 Semesterüberblick SPiC (Fortsetzung)

KW	Mo	Di	Mi	Do	Fr	Themen / Kapitel
23	07.06.	08.06.	09.06.	10.06.	11.06.	*Nebenläufigkeit, Betriebssysteme* *G*
		VL8		A6(PrintDir)		
24	14.06.	15.06.	16.06.	17.06.	18.06.	*Speicherorganisation, Stapelaufbau* *H*
		VL9		A7 (fish)		
25	21.06.	22.06.	23.06.	24.06.	25.06.	*Dateisysteme* *I*
		VL10		A8 (tqsh)		
26	28.06.	29.06.	30.06.	01.07.	02.07.	*Prozesse, Prozesszustände, Prozesswechsel, Prozesserzeugung, Ausführen von Programmen* *J.1 – J.6*
		VL11		A9 (find_max)		
27	05.07.	06.07.	07.07.	08.07.	09.07.	*Signale* *J.7*
28	12.07.	13.07.	14.07.	15.07.	16.07.	*Threads, Koordinierung* *J.8 – J.9*
		VL13		Wdh		
29	19.07.	20.07.	21.07.	22.07.	23.07.	*pthreads, Threads in Java* *J.10 – J.11*

B.3 Vorlesungsbetrieb

- Dozenten
  - Jürgen Kleinöder (SPiC, GSPiC)
  - Daniel Lohmann (GSPiC, SPiC)
- SPiC-Webseite: **`www4.informatik.uni-erlangen.de/Lehre/SS10/V_SPIC/`**
- GSPiC-Webseite: **`www4.informatik.uni-erlangen.de/Lehre/SS10/V_GSPIC/`**

B.4 Vorlesungsskript

- Vorlesungsfolien
  - im pdf-Format auf der Webseite
  - Gutscheinverkauf zum Bezug von Folienkopien, Schutzgebühr 1 EUR
    - ➤ Kopien werden jeweils vor der Vorlesung ausgegeben

B.5 Literatur

- Literatur
  - zu der Programmiersprache C
    - ➤ Peter A. Darnell, Philip E. Margolis:
      C: A Software Engineering Approach, 3. Edition, Springer, 1996.
    - ➤ Karlheinz Zeiner:
      Programmieren lernen mit C, 2. Auflage, Carl Hanser, 1996.
    - ➤ B. W. Kernighan, D. M. Ritchie:
      Programmieren in C, 2. Auflage, Carl Hanser, 1990.

B.6 Übungen

- Beginn: Do. 29.04.2009 (Übungswoche jeweils Do. - Mi.)
- Tafelübungen (teilweise auch "am Rechner")
  - ➤ Erläuterung zur Benutzung der Rechnerumgebung
  - ➤ Anleitung zu den Aufgabenstellungen
  - ➤ Besprechung der Lösungen
- Rechnerübungen Raum 01.155 Informatik-Hochhaus
  - ➤ selbstständige Bearbeitung von Aufgaben
  - ➤ Übungsleiter stehen bei Fragen und Problemen zur Verfügung
- Verantwortlich
  - ➤ Moritz Strübe (SPiC, GSPiC), Wanja Hofer (GSPiC, SPiC)
  - ➤ Übungsbetreuer: Rainer Müller, Tobias Scharpff, Martin Klüpfel, Fabian Festerra, Franziska Bertelshofer, Christian Schlumberger, Sebastian Schinabeck

B.6 Übungen (2)

- Übungsbetrieb für **SPiC**
  (Mechatronik-Bachelor, 2. Semester, Mathematik, Technomathematik)
  - 9 Gruppen zur Auswahl
    - ➤ Dauer einer Übung 90 Minuten
    - ➤ maximale Teilnehmerzahl 12 Personen
  - Termine **Tafelübung** (Änderungen vorbehalten):
    Mo. 14-16, 16-18
    Di. 12-14, 16-18
    Mi. 10-12, 14-16
    Do. 14-16, 16-18
    Fr. 10-12
  - Termine **Rechnerübung**:
    Mo. 12 - 14
    Di. 14 - 16
    Mi. 08 - 10, 14 - 16
    Do. 10 - 12, 16 - 18

B.6 Übungen (3)

- Übungsbetrieb für **GSPiC**
  (EEI-Bachelor, 2. Semester)
  - 8 Gruppen zur Auswahl
    - ➤ Dauer einer Übung 90 Minuten
    - ➤ maximale Teilnehmerzahl 12 Personen
  - Termine (Änderungen vorbehalten):

    Mo. 10-12, 12-14
    Di. 14-16, 16-18
    Mi. 14-16
    Do. 8-10, 12-14
    Fr. 14-16
  - **Tafel-** und **Rechnerübung** im Wechsel

B.6 Übungen (4)

- Anmeldung zu den Tafelübungen
  - ➤ heute (Di, 20.4.) ab 16:00 (nach der Vorlesung)
  - ➤ über Web-Anmeldesystem "waffel"
  - ➤ Link auf der Übungs-Webseite
  - ➤ Bei der Anmeldung Auswahl des Tafelübungstermins

B.7 Programmieraufgaben

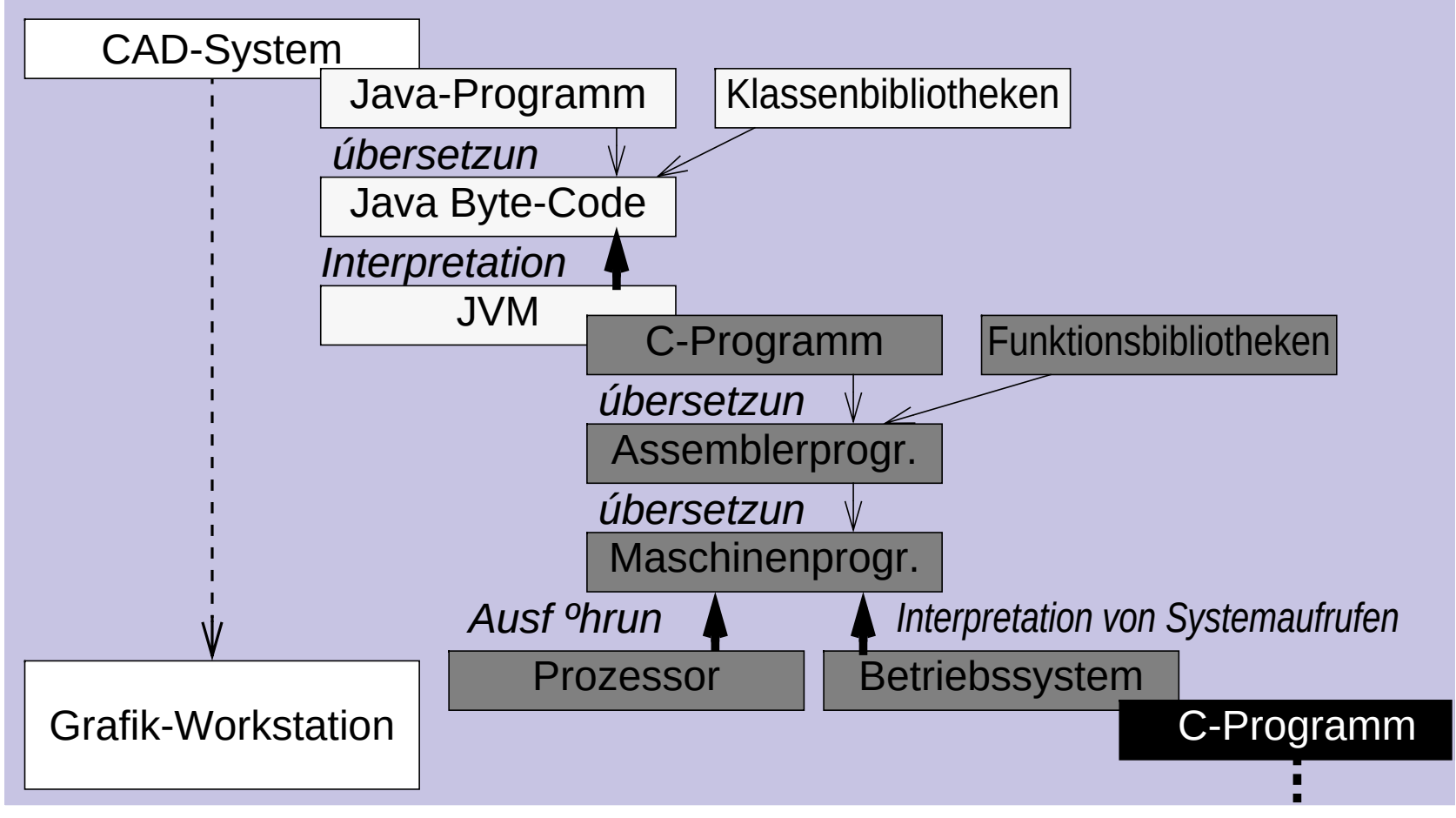
- Programmieraufgaben teilweise alleine, teilweise in 2er-Gruppen zu bearbeiten
- Lösungsaufgaben mit Abgabeskript am Rechner abgeben
- Lösung wird durch Skripte überprüft;
  zusätzlich korrigieren wir die Abgaben und geben sie zurück;
  außerdem geben wir Hinweise auf typische Fehler
  in der Vorlesung und den Tafelübungen.
- ★ abgegebene Aufgaben werden bepunktet
  durch die Punkte auf Übungsaufgaben können bis zu
  10 % Bonuspunkte
  bei der Prüfungsklausur erarbeitet werden
  - ➤ Voraussetzung: abgegebene Aufgaben müssen jederzeit in den Tafelübungen vorgestellt werden können
    (impliziert Anwesenheit!)

B.8 Prüfung

- Prüfung (Klausur)
  - ➤ Termin voraussichtlich Ende Juli/Anfang August
    Dauer GSPiC: 60 min, SPiC: 90 min
  - ➤ Inhalt: Fragen zum Vorlesungsstoff + Programmieraufgabe

C Systemarchitekturen

- Große Diskrepanz zwischen Anwendungsproblem und dem Ablauf der Lösung auf einer Hardware



C.1 Softwareschichten

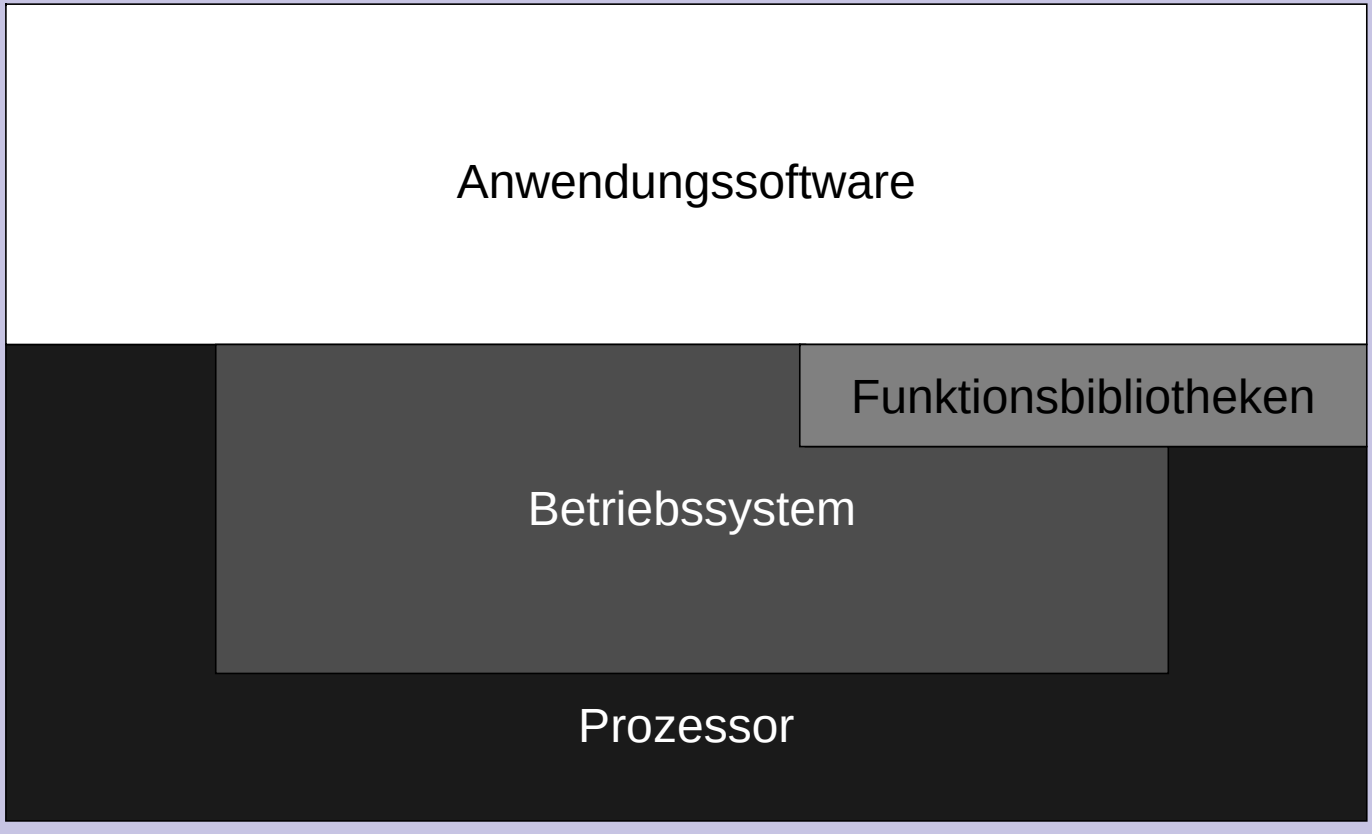
- Anwendungs-/Problemorientierte Darstellungen
  - ◆ Modelle
    - ➤ Matlab/Simulink
    - ➤ UML
  - ◆ Programmiersprachen / höhere Abstraktionsebenen
    - ➤ Software-Komponenten
    - ➤ Java, C#, C++, Tcl/TK
- Softwarewerkzeuge konvertieren / generieren
  - ➤ Matlab/Simulink ➔ C
- Ausführungsumgebungen unterstützen / konvertieren / interpretieren
  - ➤ Enterprise Java Beans
  - ➤ JVM oder .NET

➥ Ziel: durch Prozessor ausführbarer Maschinencode

C.1 Softwareschichten (2)

- verschiedene Ausführungsmodelle für Maschinencode
  - vollständig durch den Prozessor ausführbar
    - alle Funktionen müssen vollständig durch die Werkzeuge in direkt ausführbaren Maschinencode umgewandelt worden sein
    - keinerlei weitere Unterstützung zur Laufzeit erforderlich
    - kann so in ROM oder EPROM gespeichert werden
    - z. B. Steuerung einer Waschmaschine
  - zusätzliche Unterstützung zur Ausführungszeit erforderlich
    - "darunter liegende" Softwareschicht realisiert Dienste: Betriebssystem
    - z. B. Daten in Datei speichern, Daten über Internet übertragen
    - Realisierung: partielle Interpretation
      bestimmte Maschinencodes werden nicht direkt vom Prozessor ausgeführt sondern stoßen die Abarbeitung von Betriebssystemfunktionen an

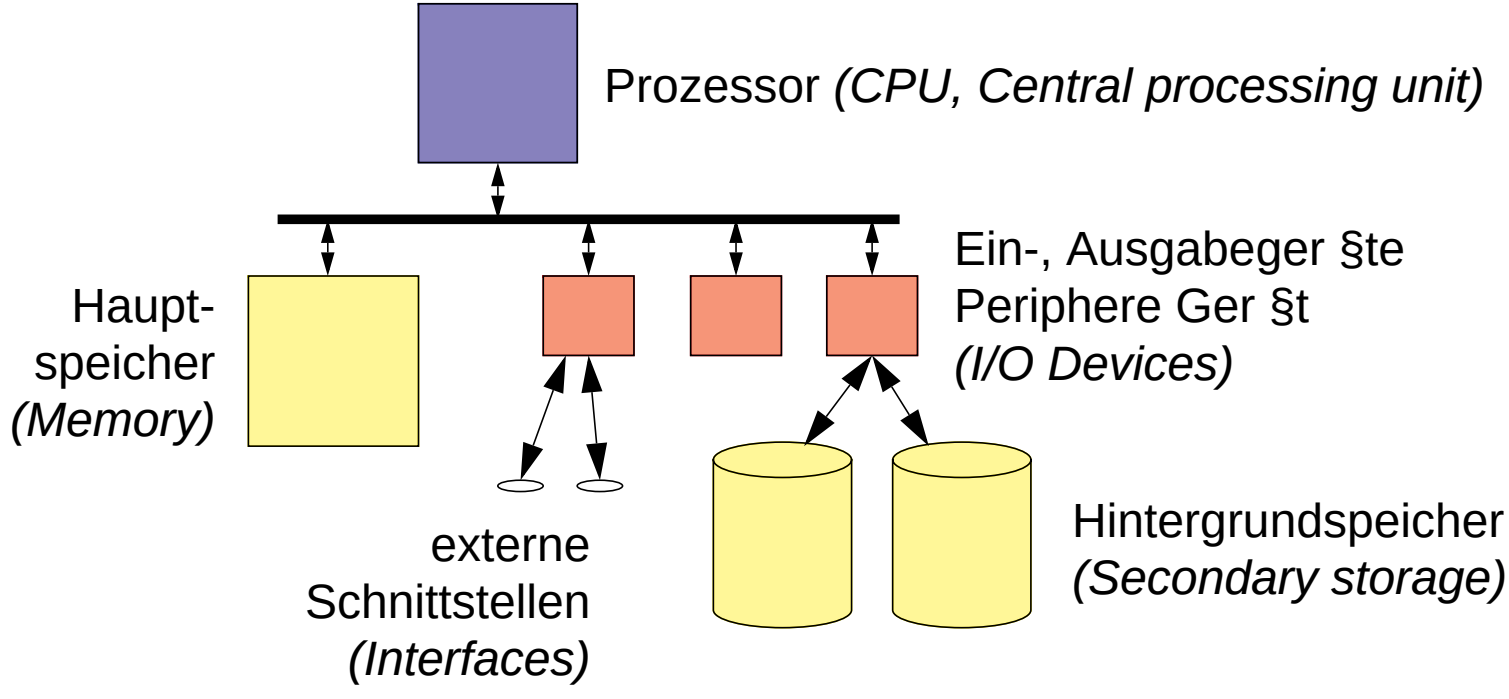
C.1 Softwareschichten (3)



C.2 Was sind Betriebssysteme?

- DIN 44300
  - „...die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften der Rechenanlage die **Basis der möglichen Betriebsarten** des digitalen Rechensystems bilden und die insbesondere die **Abwicklung von Programmen steuern und überwachen.**“
- Andy Tanenbaum
  - „...eine Software-Schicht ..., die alle Teile des Systems verwaltet und dem Benutzer eine Schnittstelle oder eine *virtuelle Maschine* anbietet, die einfacher zu verstehen und zu programmieren ist [als die nackte Hardware].“
- ★ Zusammenfassung:
  - Software zur Verwaltung und Virtualisierung der Hardwarekomponenten (Betriebsmittel)
  - Programm zur Steuerung und Überwachung anderer Programme

1 Verwaltung von Betriebsmitteln



1 Verwaltung von Betriebsmittel (2)

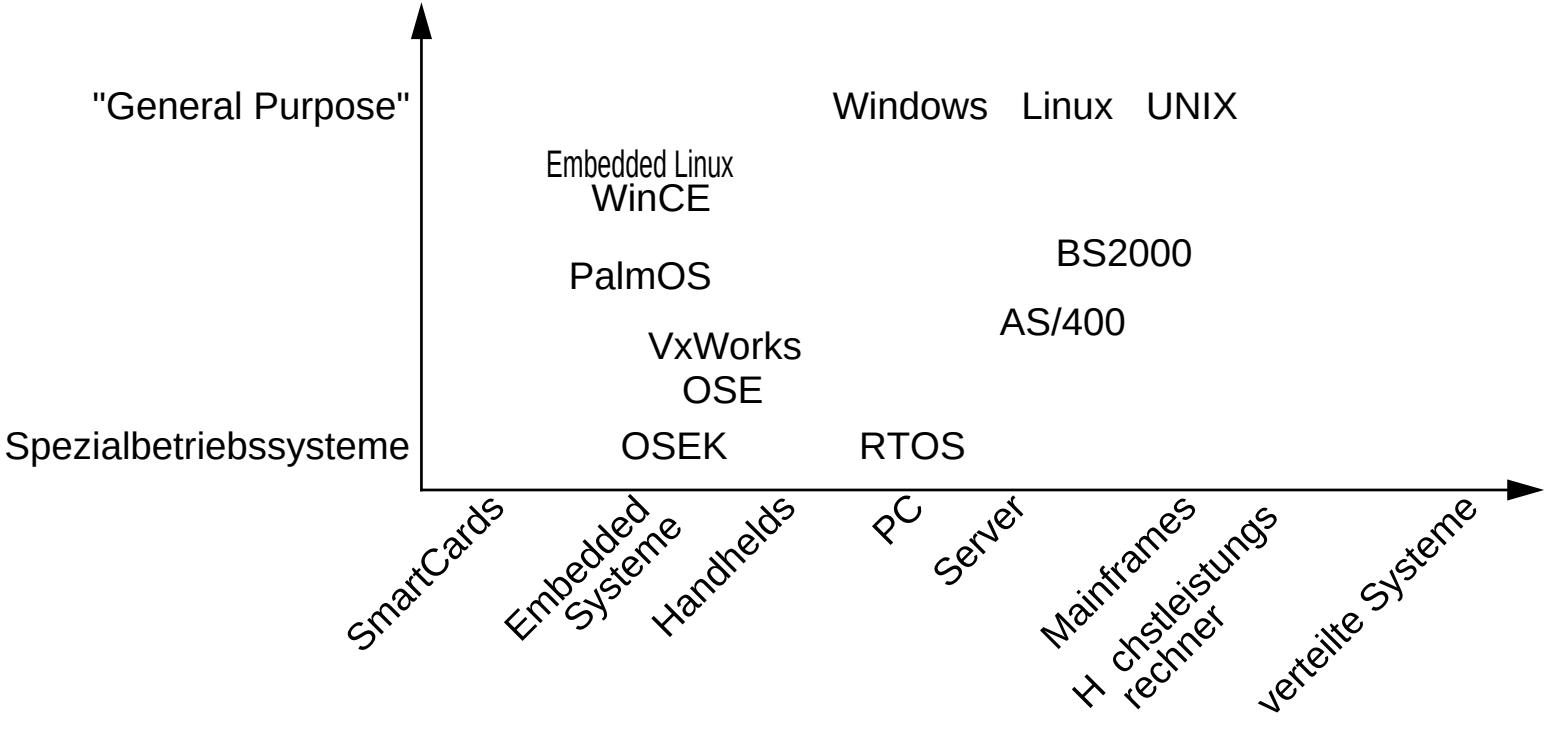
- Resultierende Aufgaben
  - Multiplexen von Betriebsmitteln für mehrere Benutzer bzw. Anwendungen
  - Schaffung von Schutzumgebungen
  - Bereitstellen von Abstraktionen zur besseren Handhabbarkeit der Betriebsmittel

- Ermöglichen einer koordinierten gemeinsamen Nutzung von Betriebsmitteln, klassifizierbar in
  - aktive, zeitlich aufteilbare (Prozessor)
  - passive, nur exklusiv nutzbare (periphere Geräte, z.B. Drucker u.Ä.)
  - passive, räumlich aufteilbare (Speicher, Plattenspeicher u.Ä.)

- Unterstützung bei der Fehlererholung

2 Klassifikation von Betriebssystemen

- Unterschiedliche Klassfikationskriterien
  - ➤ Zielplattform
  - ➤ Einsatzzweck, Funktionalität



2 Klassifikation von Betriebssystemen (2)

- Wenigen "General Purpose"- und Mainframe/Höchstleistungsrechner-Betriebssystemen steht eine Vielzahl kleiner und kleinster Spezialbetriebssysteme gegenüber:

  C51, C166, C251, CMX RTOS, C-Smart/Raven, eCos, eRTOS, Embos, Ercos, Euros Plus, Hi Ross, Hynet-OS, LynxOS, MicroX/OS-II, Nucleus, OS-9, OSE, OSEK Flex, OSEK Turbo, OSEK Plus, OSEKtime, Precise/MQX, Precise/RTCS, proOSEK, pSOS, PXROS, QNX, Realos, RTMOSxx, Real Time Architect, ThreadX, RTA, RTX51, RTX251, RTX166, RTXC, Softune, SSXS RTOS, VRTX, VxWorks, …

➥ Einsatzbereich: Eingebettete Systeme, häufig Echtzeit-Betriebssysteme, über 50% proprietäre (in-house) Lösungen

- Alternative Klassifikation: nach Architektur

3 Betriebssystemarchitekturen

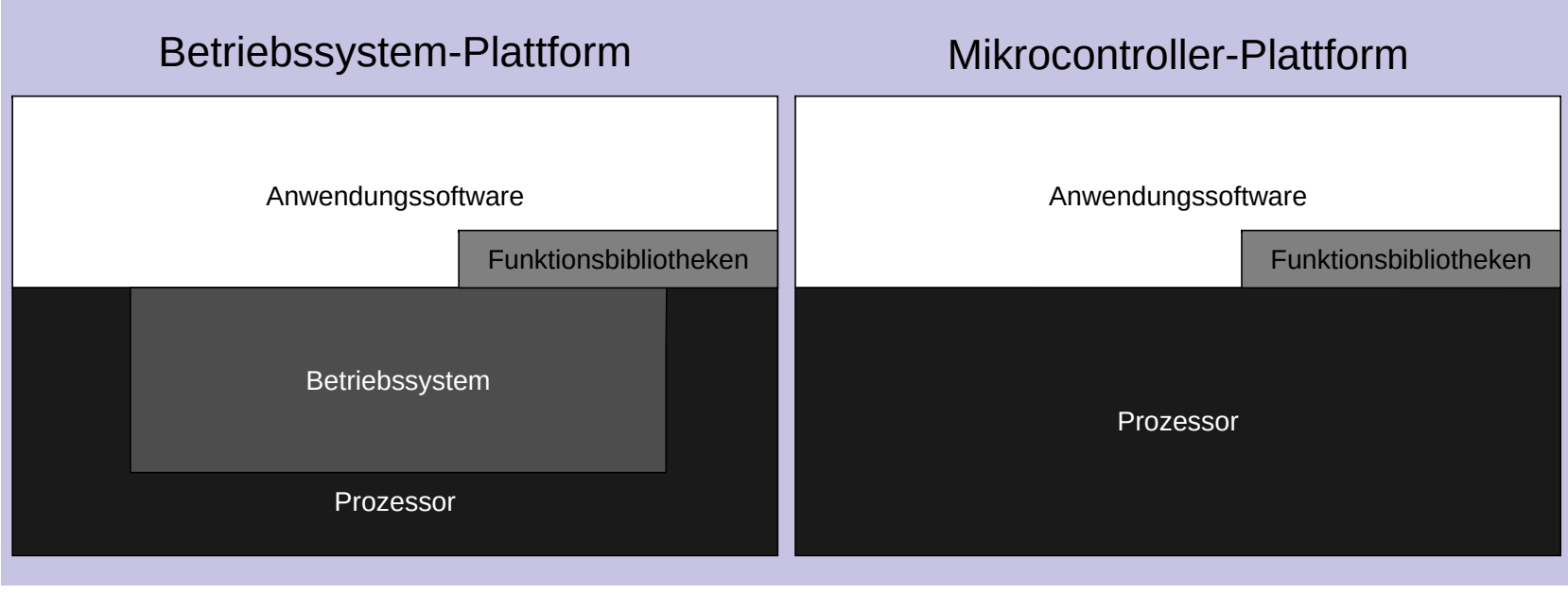
- Umfang zehntausende bis mehrere Millionen Befehlszeilen
  - Strukturierung hilfreich
- Verschiedene Strukturkonzepte
  - monolithische Systeme
  - geschichtete Systeme
  - Minimalkerne
  - Laufzeitbibliotheken (minimal, vor allem im Embedded-Bereich)
- Unterschiedliche Schutzkonzepte
  - ➤ kein Schutz
  - ➤ Schutz des Betriebssystems
  - ➤ Schutz von Betriebssystem und Anwendungen untereinander
  - ➤ feingranularer Schutz auch innerhalb von Anwendungen

4 Betriebssystemkomponenten

- Speicherverwaltung
  - Wer darf wann welche Information wohin im Speicher ablegen?
- Prozessverwaltung
  - Wann darf welche Aufgabe bearbeitet werden?
- Dateisystem
  - Speicherung und Schutz von Langzeitdaten
- Ein/Ausgabe
  - Kommunikation mit der "Außenwelt" (Benutzer/Rechner)

C.3 Mikrocontroller vs. Betriebssystem-Plattform

- Entscheidende Unterschiede:
  - Betriebssystem-Unterstützung entfällt



  - Prozessor bietet in der Regel weniger / andere Funktionalität
    - ➤ kein virtueller Speicher
    - ➤ kein Speicherschutz
    - ➤ einfachere Peripherie-Ansteuerung

D Einführung in die Programmiersprache C

D.1 C vs. Java

- Java: objektorientierte Sprache
  - ➤ zentrale Frage: aus welchen Dingen besteht das Problem
  - ➤ Gliederung der Problemlösung in Klassen und Objekte
  - ➤ Hierarchiebildung: Vererbung auf Klassen, Teil-Ganze-Beziehungen
  - ➤ Ablauf: Interaktion zwischen Objekten
- C: imperative / prozedurale Sprache
  - ➤ zentrale Frage: welche Aktivitäten sind zur Lösung des Problems auszuführen
  - ➤ Gliederung der Problemlösung in Funktionen
  - ➤ Hierarchiebildung: Untergliederung einer Funktion in Teilfunktionen
  - ➤ Ablauf: Ausführung von Funktionen

D.1 C vs. Java

1 C hat nicht

- Klassen und Vererbung
- Objekte
- umfangreiche Klassenbibliotheken

2 C hat

- Zeiger und Zeigerarithmetik
- Präprozessor
- Funktionsbibliotheken

D.2 Sprachüberblick

1 Erstes Beispiel (C-Programm unter Linux)

- Die Datei **`hello.c`** enthält die folgenden Zeilen:

```
#include <stdio.h>
/* say "hello, world" */
int main()
{
    printf("hello, world\n"); return 0;
}
```

- Die Datei wird mit dem Kommando **`cc`** übersetzt:

```
% cc hello.c
```
(C-Compiler)

oder

```
% gcc hello.c
```
(GNU-C-Compiler)

es entsteht eine Datei **`a.out`**, die das ausführbare Programm enthält.

➤ ausführbares Programm liegt in Form von Maschinencode des Zielprozessors vor (kein Byte- oder Zwischencode)!

1 Erstes Beispiel (2)

- Mit der Option **`-o`** kann der Name der Ausgabedatei auch geändert werden – z. B.

```
% cc -o hello hello.c
```

- Das Programm wird durch Aufruf der Ausgabedatei ausgeführt:

```
% ./hello
hello, world
%
```

- Kommandos werden so in einem Fenster mit UNIX/Linux-Kommandointerpreter (Shell) eingegeben
  - ➤ es gibt auch integrierte Entwicklungsumgebungen (z. B. Eclipse)

2 Erstes Beispiel (C-Programm für AVR-Mikrocontroller)

- Die Datei **`red.c`** enthält die folgenden Zeilen:

```
/* switch red led on */
#include <led.h>
void main()
{
    sb_led_on(RED0); while(1);
}
```

- Die Datei wird mit dem Kommando **`avr-gcc`** übersetzt:

```
% avr-gcc -o red.elf -ffreestanding -mmcu=atmega32 ... red.c
```

  - ➤ im Gegensatz zur Übersetzung eines Programms für Linux muss hier
    – die Zielplattform angegeben werden *(Cross-Compilation)*
    – Angaben über Bibliotheken und include-Dateien angegeben werden (...)

- Vereinfachung über "Makefile"
  - ➤ Details in der Übung

2 Erstes Beispiel (C-Programm für AVR-Mikrocontroller) (2)

- In der Datei **`red.elf`** liegt der ausführbare Programmcode für den Mikrocontroller vor
- dieser muss anschliessend auf den Mikrocontroller geladen werden
  - ➤ Mikrocontroller über USB-Schnittstelle an Entwicklungs-PC anschließen
  - ➤ Programm zum Übertragen des Codes (*Flashen*) starten
- Weitere Details in den Übungen!

3 Aufbau eines C-Programms

- frei formulierbar **- *Zwischenräume*** *(Leerstellen, Tabulatoren, Newline und Kommentare*) werden i. a. ignoriert - sind aber zur eindeutigen Trennung direkt benachbarter Worte erforderlich
- ***Kommentar*** wird durch `/*` und `*/` geklammert
  keine Schachtelung möglich
- ***Identifier*** (Variablennamen, Marken, Funktionsnamen, ...) sind aus Buchstaben, gefolgt von Ziffern oder Buchstaben aufgebaut
  - "_" gilt hierbei auch als Buchstabe
  - Schlüsselwörter wie `if`, `else`, `while`, usw. können nicht als *Identifier* verwendet werden
  - ***Identifier*** müssen vor ihrer ersten Verwendung ***deklariert*** werden
- Anweisungen werden generell durch `;` abgeschlossen

4 Allgemeine Form eines C-Programms:

```
/* globale Variablen */
     ...

/* Hauptprogramm */
main(...)
{
     /* lokale Variablen */
          ...
     /* Anweisungen */
          ...
}

/* Unterprogramm 1 */
function1(...)
{
     /* lokale Variablen */
          ...
     /* Anweisungen */
          ...
}

/* Unterprogramm n */
functionN(...)
{
     /* lokale Variablen */
          ...
     /* Anweisungen */
          ...
}
```

5 Wie ein C-Programm nicht aussehen sollte:

```
#define o define
#o ___o write
#o ooo (unsigned)
#o o_o_ 1
#o _o_ char
#o _oo goto
#o _oo_ read
#o o_o for
#o o_ main
#o o__ if
#o oo_ 0
#o _o(_,__,___)(void)___o(_,__,ooo(___))
#o __o(o_o_<<((o_o_<<(o_o_<<o_o_))+(o_o_<<o_o_)))
+(o_o_<<(o_o_<<(o_o_<<o_o_)))
o_(){_o_ _=oo_,__,___,____[__o];_oo _______;_____:___=__o-o_
_______:
_o(o_o_,____,__=(_-o_o_<___?_-
o_o_:___));o_o(;__;_o(o_o_,"\b",o_o_),__--);
_o(o_o_," ",o_o_);o__(--___)_oo
_______;_o(o_o_,"\n",o_o_);______:o__(_=_oo_(
oo_,____,__o))_oo _____;}
```

sieht eher wie Morse-Code aus, ist aber ein **gültiges** C-Programm.

D.3 Datentypen

- Datentypen
  - ➤ Konstanten
  - ➤ Variablen

- ◆ Ganze Zahlen
- ◆ Fließkommazahlen
- ◆ Zeichen
- ◆ Zeichenketten

1 Was ist ein Datentyp?

- Menge von Werten
  +
  Menge von Operationen auf den Werten

  - **Literale** — Darstellung für einen konkreten Wert (`2`, `3.14`, `'a'`)
  - **Variablen** — Namen für Speicherplätze, die einen Wert aufnehmen können

  ➥ Literale 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)

2 Standardtypen in C

- Eine Reihe häufig benötigter Datentypen ist in C vordefiniert

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

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

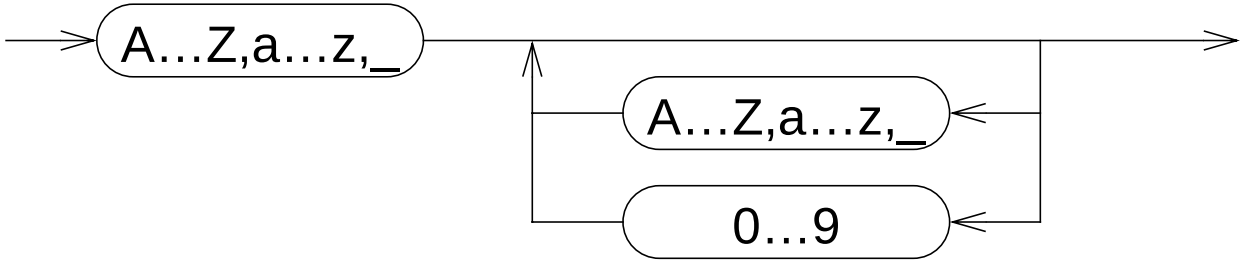
`const`

legt fest, dass der Inhalt einer Variable des Datentyps nicht verändert werden darf

3 Variablen

- Variablen haben:
  - **Namen** (Bezeichner)
  - Typ
  - zugeordneten Speicherbereich für einen Wert des Typs
    Inhalt des Speichers (= **aktueller Wert** der Variablen) ist veränderbar!
  - **Lebensdauer**
    wann wird der Speicherplatz angelegt und wann freigegeben

- Bezeichner

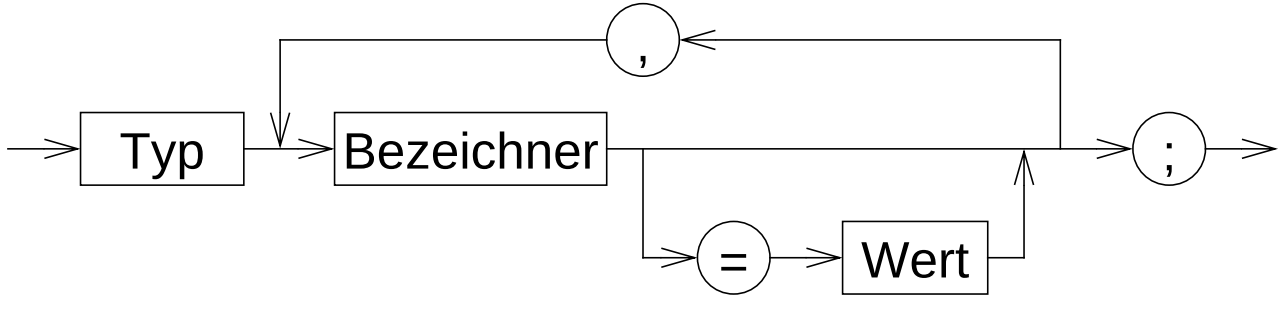


(Buchstabe oder _ ,
evtl. gefolgt von beliebig vielen Buchstaben, Ziffern oder _)

3 Variablen (2)

- Typ und Bezeichner werden durch eine **Variablen-Deklaration** festgelegt (= dem Compiler bekannt gemacht)
  - 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



3 Variablen (3)

- Variablen-Definition: Beispiele

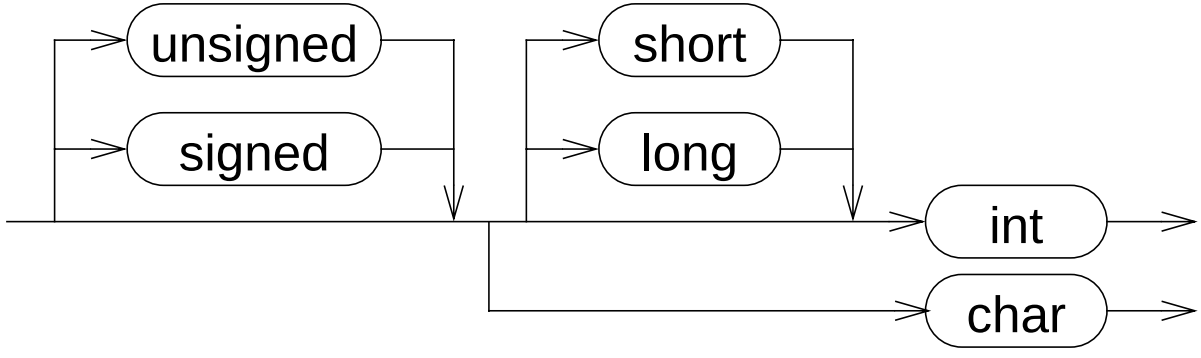
```
int a1;
float a, b, c, dis;
int anzahl_zeilen = 5;
const char Trennzeichen = ':';
```

  - Position im Programm:
    - nach jeder "`{`"
    - außerhalb von Funktionen
    - neuere C-Standards und der GNU-C-Compiler erlauben Definitionen an beliebiger Stelle im Programmcode: Variable ab der Stelle gültig

- Wert kann bei der Definition initialisiert werden
- Wert ist durch Wertzuweisung und spezielle Operatoren veränderbar
- Lebensdauer ergibt sich aus der Programmstruktur

4 Ganze Zahlen

- Definition

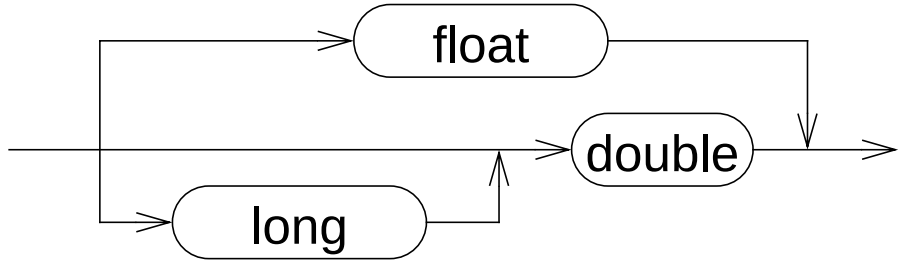


- Speicherbedarf: (**`char`**) ≤ (**`short int`**) ≤ (**`int`**) ≤ (**`long int`**)
- Speicherbedarf(**`int`**): meist 16 oder 32 Bit
- Literale (Beispiele):

```
42, -117
035          (oktal = 29_10)
0x10         (hexadezimal = 16_10)
0x1d         (hexadezimal = 29_10)
```

5 Fließkommazahlen

- Definition



- Speicherbedarf(**`float`**) ≤ Speicherbedarf(**`double`**) ≤ Speicherbedarf(**`long double`**)
- Speicherbedarf(**`float`**): 32 Bit
- Literale (Beispiele):
  - normale Dezimalpunkt-Schreibweise

    `3.14`, **`-2.718`**, **`368.345, 0.003`**

    `1.0` aber nicht einfach **`1`** (wäre ein **`int`**-Literal!)
  - 10er-Potenz Schreibweise ($368.345 = 3.68345 \cdot 10^{2}$, $0.003 = 3.0 \cdot 10^{-3}$)

    `3.68345e2`, **`3.0e-3`**

6 Zeichen

- Bezeichnung: **`char`**
- Speicherbedarf: 1 Byte
- Repräsentation: ASCII-Code
  zählt damit zu den ganzen Zahlen
- Werte: Zeichen durch `' '` geklammert
  - Beispiele: **`'a'`**, **`'X'`**
  - Sonderzeichen werden durch **Escape-Sequenzen** beschrieben

Tabulator:	**`'\t'`**	Backslash:	**`'\\'`**
Zeilentrenner:	**`'\n'`**	Backspace:	**`'\b'`**
Apostroph:	**`'\''`**		

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

7 Zeichenketten (Strings)

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

```
const char *Mitteilung = "Dies ist eine Mitteilung\n";
```

D.4 Ausdrücke

- Ausdruck = gültige Kombination von
  Operatoren, **Werten** 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.5 Operatoren

1 Zuweisungsoperator =

- Zuweisung eines Werts an eine Variable

- Beispiel:

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

2 Arithmetische Operatoren

- für alle **`int`** und **`float`** Werte erlaubt

`+`	Addition
`-`	Subtraktion
`*`	Mulitiplikation
`/`	Division
`%`	Rest bei Division, (modulo)
unäres `-`	negatives Vorzeichen (z. B. **`-3`**)
unäres `+`	positives Vorzeichen (z. B. **`+3`**)

- Beispiel:

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

3 spezielle Zuweisungsoperatoren

- Verkürzte Schreibweise für Operationen auf einer Variablen

  a *op*= b ≡ a = a *op* b

  mit ***op*** ∈ **{ +, −, *, /, %, <<, >>, &, ^, | }**

- Beispiele:

```
int a = -8;

a += 24;                /* -> a: 16 */
a /= 2;                 /* -> a: 8 */
```

4 Vergleichsoperatoren

`<`	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 ) { …
```

5 Logische Operatoren

➥ Verknüpfung von Wahrheitswerten (wahr / falsch)

"nicht"

!	
f	w
w	f

"und"

&&	f	w
f	f	f
w	f	w

"oder"

||	f	w
f	f	w
w	w	w

◆ Wahrheitswerte (Boole'sche Werte) werden in C generell durch int-Werte dargestellt:

➤ Operanden in einem Ausdruck:
Operand = 0: falsch
Operand ≠ 0: wahr

➤ Ergebnis eines Ausdrucks:
falsch: 0
wahr: 1

5 Logische Operatoren (2)

- Beispiel:

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

1 *und* 0 → 0

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

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

0 → Gesamtergebnis=*falsch*

wird nicht ausgewertet → (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"

^	f	w
f	f	w
w	w	f

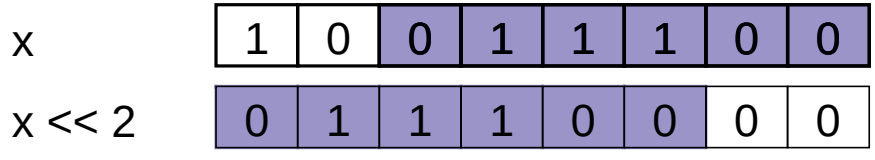
- Beispiele:

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

7 Logische Shiftoperatoren

- Bits werden im Wort verschoben
  - `<<` Links-Shift
  - `>>` Rechts-Shift

- Beispiel:



7 Inkrement / Dekrement Operatoren

`++`	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 */
```

8 Bedingte Bewertung

```
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;
```

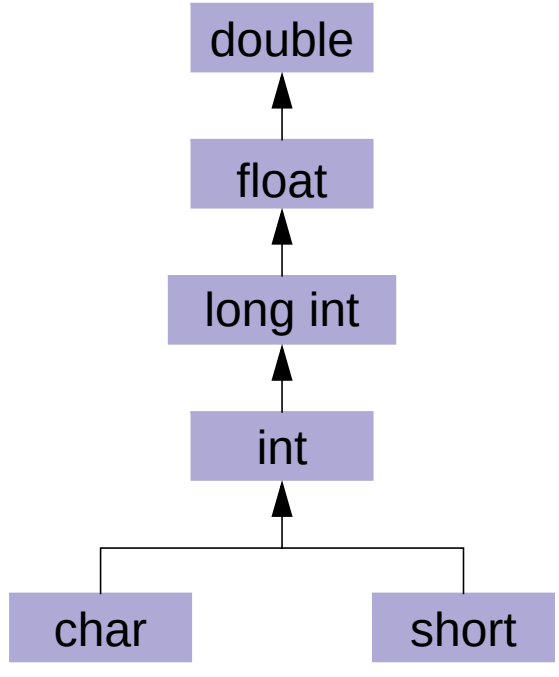
9 Komma-Operator

,

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

10 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)

11 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

D.6 Einfacher Programmaufbau

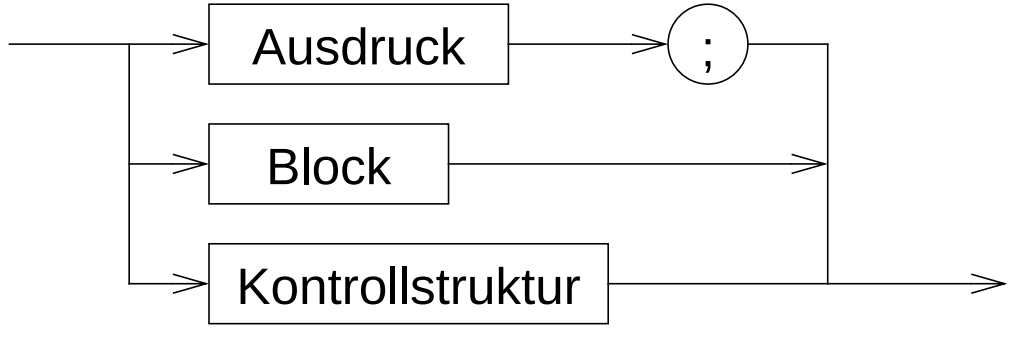
- Struktur eines C-Hauptprogramms
- Anweisungen und Blöcke
- Einfache Ein-/Ausgabe
- C-Präprozessor

1 Struktur eines C-Hauptprogramms

```
int main()
{
    Variablendefinitionen
    Anweisungen
    return 0;
}
```

2 Anweisungen

Anweisung:



3 Blöcke

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

```
int 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 */
    }
    ...
```

4 Einfache Ein-/Ausgabe

- Jeder Prozess (jedes laufende Programm) bekommt unter Linux von der Shell als Voreinstellung drei Ein-/Ausgabekanäle:

  stdin als Standardeingabe

  stdout als Standardausgabe

  stderr Fehlerausgabe

- Die Kanäle **stdin**, **stdout** und **stderr** sind in UNIX auf der Kommandozeile umlenkbar:

```
% prog < EingabeDatei > AusgabeDatei
```

4 Einfache Ein-/Ausgabe (2)

- Für die Sprache C existieren folgende primitive Ein-/Ausgabefunktionen für die Kanäle **stdin** und **stdout**:

  getchar zeichenweise Eingabe

  putchar zeichenweise Ausgabe

  scanf formatierte Eingabe

  printf formatierte Ausgabe

- folgende Funktionen ermöglichen Ein-/Ausgabe auf beliebige Kanäle (z. B. auch **stderr**)

  getc, **putc**, **fscanf**, **fprintf**

5 Einzelzeichen E/A

- ***getchar( ), getc( )*** ein Zeichen lesen
  - Beispiel:

```
int c;
c = getchar();
```

```
int c;
c = getc(stdin);
```

- ***putchar( ), putc( )*** ein Zeichen schreiben
  - Beispiel:

```
char c = 'a';
putchar(c);
```

```
char c = 'a';
putc(c, stdout);
```

- Beispiel:

```
#include <stdio.h>

/*
 * kopiere Eingabe auf Ausgabe
 */
main()
{
    int c;
    while ( (c = getchar()) != EOF )
    {
        putchar(c);
    }
}
```

6 Formatierte Ausgabe

- Aufruf: **`printf (`** *format*`,` *arg*`)`
- ***printf*** konvertiert, formatiert und gibt die **Werte *(arg)*** unter der Kontrolle des Formatstrings ***format*** aus
  - die Anzahl der Werte (*arg*) ist abhängig vom Formatstring
- sowohl für ***format***, wie für ***arg*** sind Ausdrücke zulässig
- ***format*** ist vom Typ **Zeichenkette *(string)***
- ***arg*** muss dem durch das zugehörige **Formatelement** beschriebenen Typ entsprechen

6 Formatierte Ausgabe (2)

- die Zeichenkette ***format*** ist aufgebaut aus:
  - ➥ **einfachem Ausgabetext**, der unverändert ausgegeben wird
  - ➥ **Formatelementen**, die Position und Konvertierung der zugeordneten ***Werte*** beschreiben

- Beispiele für **Formatelemente**:

Zeichenkette:	**`%[-][`*`min`*`][.`*`max`*`]s`**
Zeichen:	**`%[+][-][`*`n`*`]c`**
Ganze Zahl:	**`%[+][-][`*`n`*`][l]d`**
Gleitkommazahl:	**`%[+][-][`*`n`*`][.`*`n`*`]f`**

`[ ]` *bedeutet optional*

- Beispiel:

```
printf("a = %d, b = %d, a+b = %d", a, b, a+b);
```

7 C-Präprozessor — Kurzüberblick

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

`#define`	Definition von Makros
`#include`	Einfügen von anderen Dateien

8 C-Präprozessor — Makrodefinitionen

- 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 Präprozessor im nachfolgenden Text der C-Quelle alle Vorkommen von ***Makroname*** durch ***Ersatztext*** ersetzt
- Beispiel:

```
#define EOF -1
```

9 C-Präprozessor — Einfügen von Dateien

- **`#include`** fügt den Inhalt einer anderen Datei in eine C-Quelldatei ein
- Syntax:

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

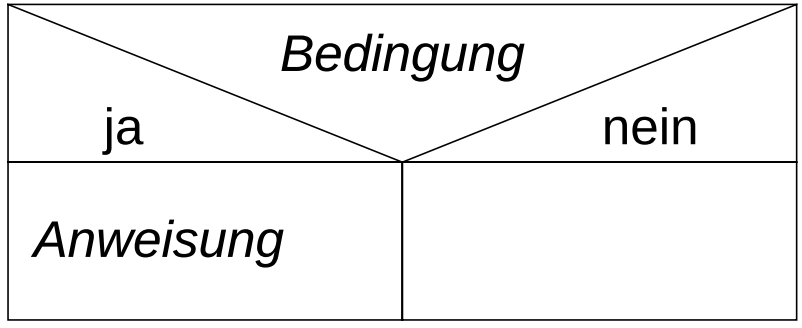
- mit **`#include`** werden *Header*-Dateien mit Daten, die für mehrere Quelldateien benötigt werden einkopiert
  - ➤ Deklaration von Funktionen, Strukturen, externen Variablen
  - ➤ Definition von Makros
- wird **Dateiname** durch **`< >`** geklammert, wird eine **Standard-Header-Datei** einkopiert
- wird **Dateiname** durch **`" "`** geklammert, wird eine Header-Datei des Benutzers einkopiert (vereinfacht dargestellt!)

D.7 Kontrollstrukturen

Kontrolle des Programmablaufs in Abhängigkeit von dem Ergebnis von Ausdrücken

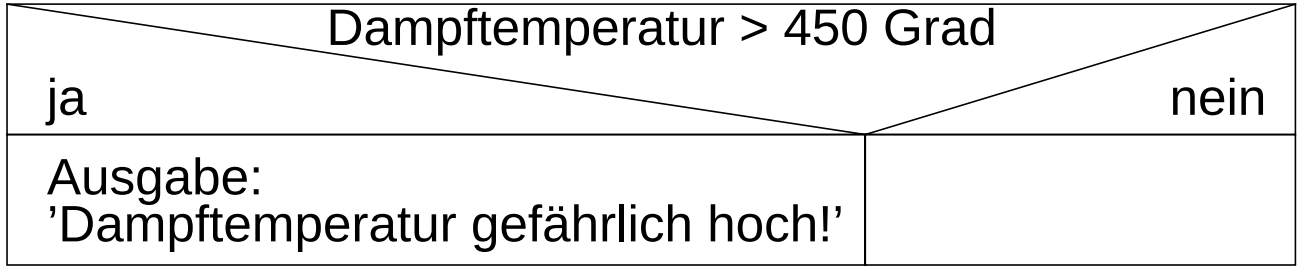
- Bedingte Anweisung
  - einfache Verzweigung
  - mehrfache Verzweigung
- Fallunterscheidung
- Schleifen
  - abweisende Schleife
  - nicht abweisende Schleife
  - Laufanweisung
  - Schleifensteuerung

1 Bedingte Anweisung



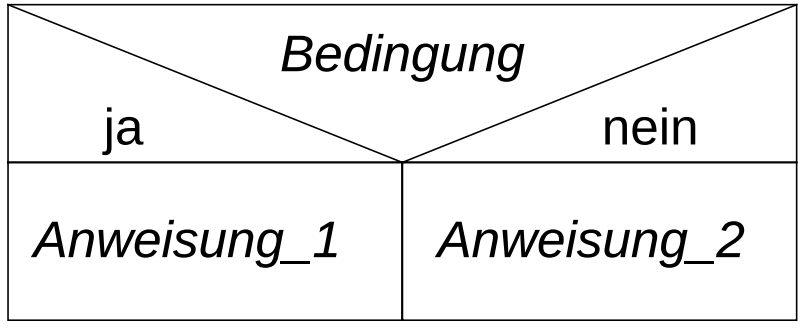

```
if ( Bedingung )
    Anweisung
```

- Beispiel:



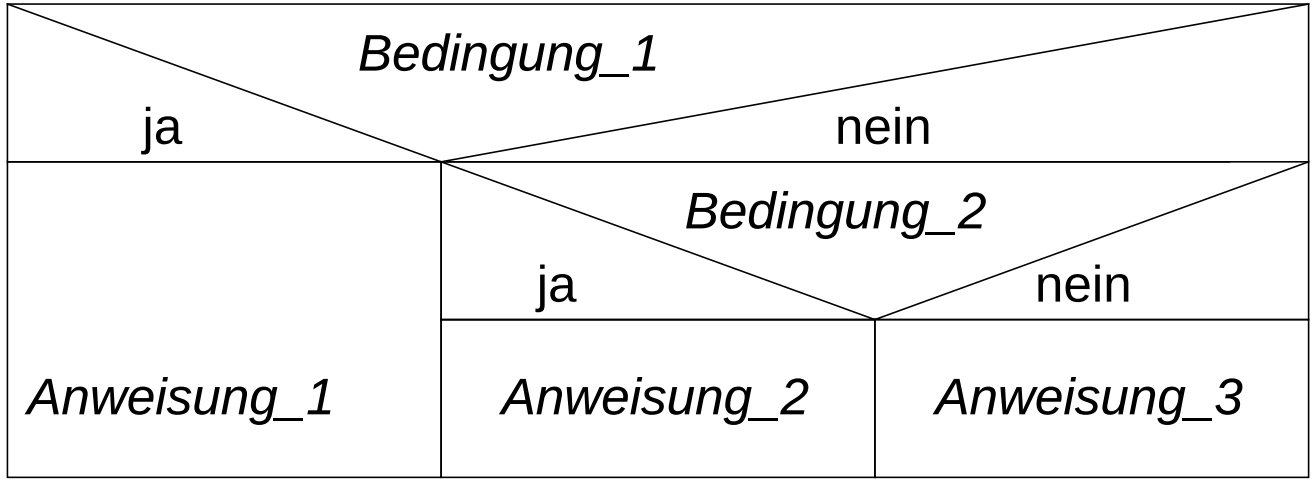

```
if (temp >= 450.0)
  printf("Dampftemperatur gefaehrlich hoch!\n");
```

1 Bedingte Anweisung einfache Verzweigung




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

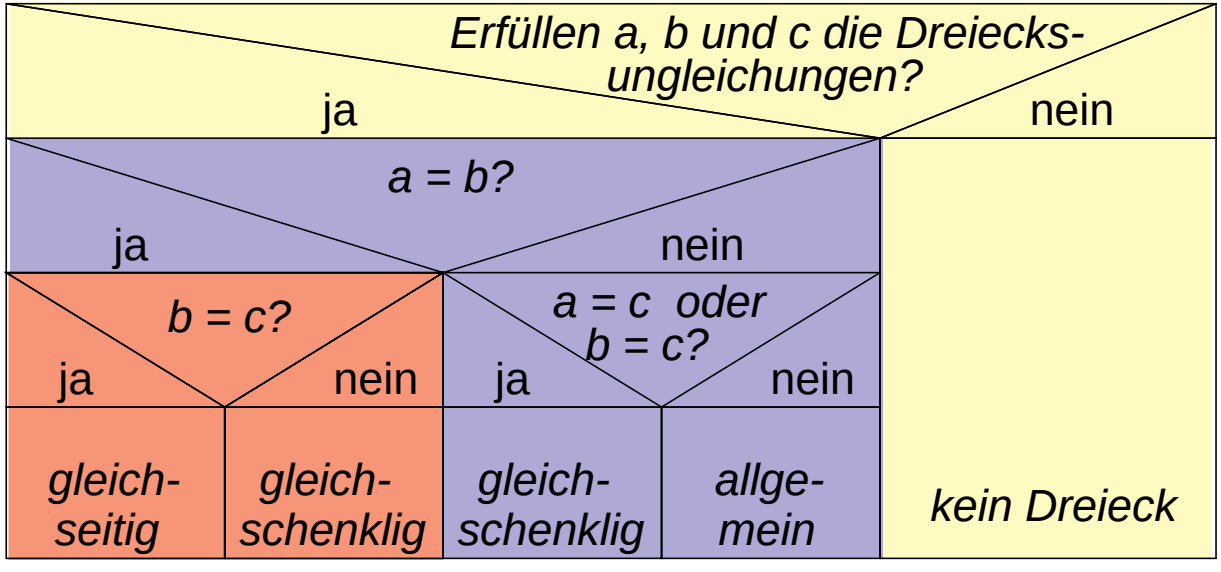
1 Bedingte Anweisung mehrfache Verzweigung




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

1 Bedingte Anweisung mehrfache Verzweigung (2)

- Beispiel: Eigenschaften von Dreiecken — Struktogramm



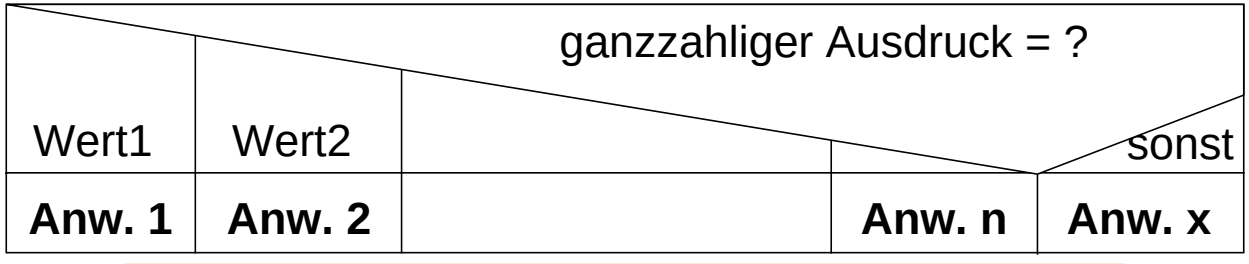
1 Bedingte Anweisung mehrfache Verzweigung (3)

- 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");
```

2 Fallunterscheidung

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




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

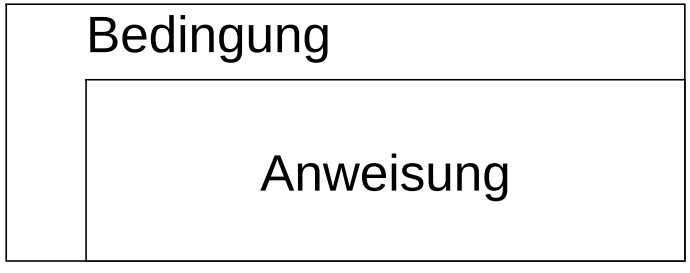
2 Fallunterscheidung — Beispiel

```
#include <stdio.h>

int main()
{
  int 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);
}
```

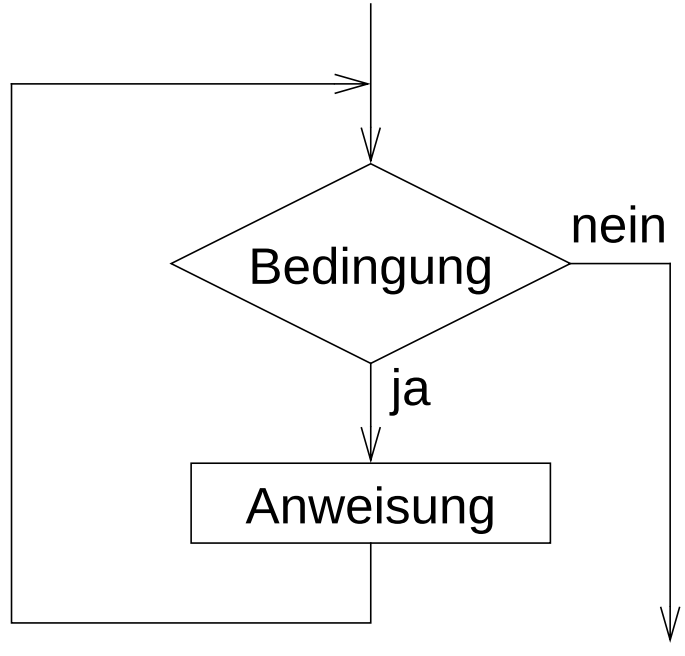
3 Schleifen

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

4 abweisende Schleife

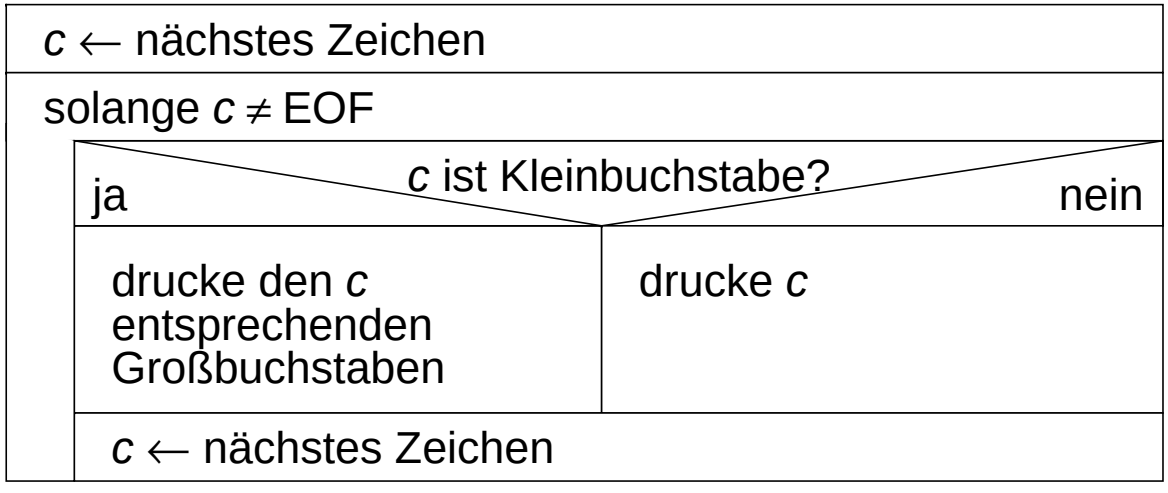



```
while ( Bedingung )
    Anweisung
```



4 abweisende Schleife (2)

- Beispiel: Umwandlung von Klein- in Großbuchstaben

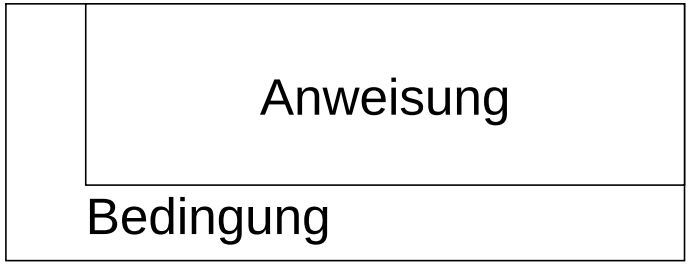



```
int c;
c = getchar();
while ( c != EOF ) {
    if ( c >= 'a' && c <= 'z' )
        putchar(c+'A'-'a');
    else
        putchar(c);
    c = getchar();
}
```

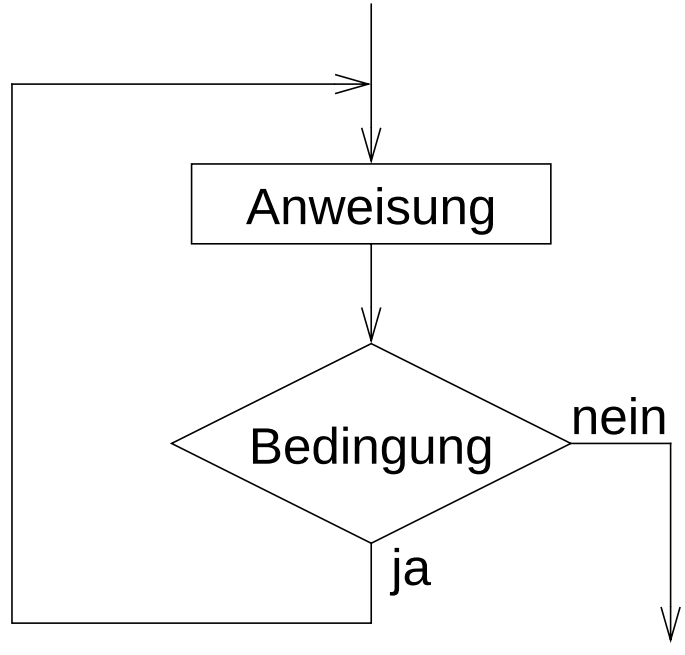
➤ abgekürzte Schreibweise

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

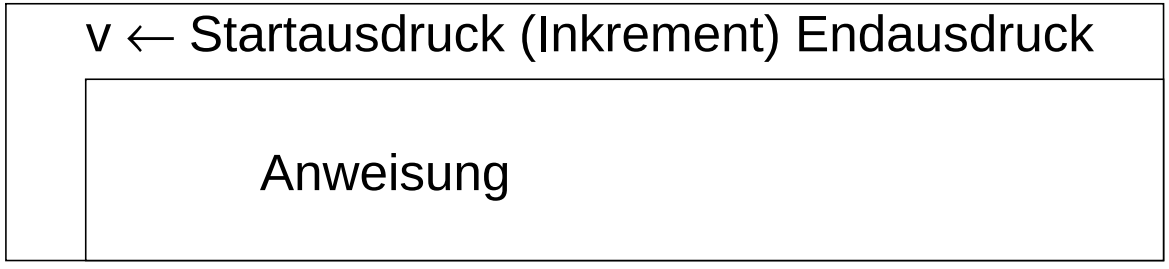
5 nicht-abweisende Schleife




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



6 Laufanweisung




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

allgemein:

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

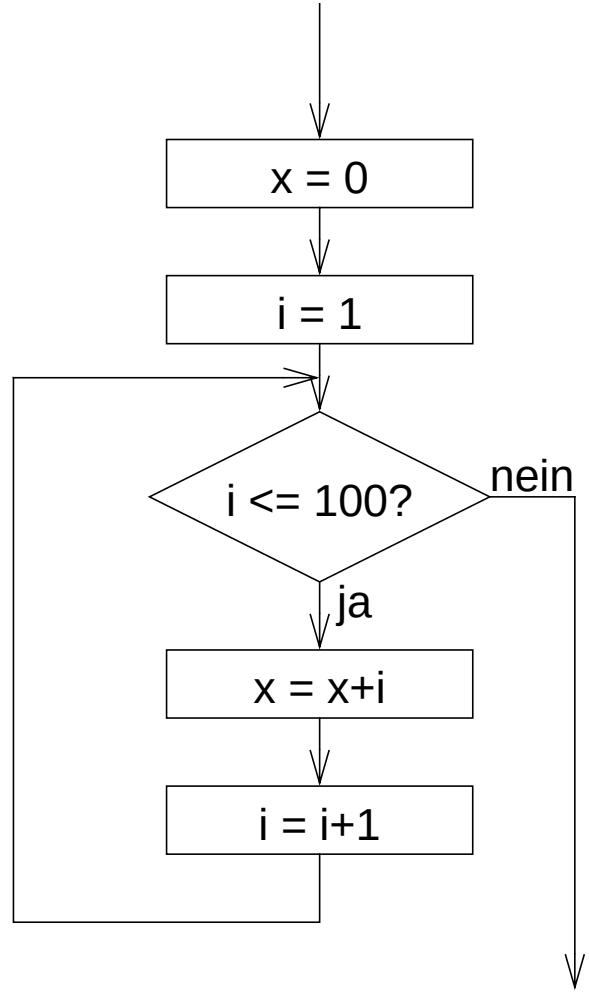
```
Ausdruck_1;
while (Ausdruck_2) {
    Anweisung
    Ausdruck_3;
}
```

6 Laufanweisung (2)

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

$x \leftarrow 0$	
$i \leftarrow 1(1)100$	
	$x \leftarrow x + i$

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



7 Schleifensteuerung

- 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

D.8 Funktionen

1 Überblick

- **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)

1 Überblick (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)

2 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);
}
```

```
int main()
{
     double wert;

     printf("Berechnung des Sinus von ");
     scanf("%lf", &wert);
     printf("sin(%lf) = %lf\n",
               wert, sinus(wert));
     return(0);
}
```

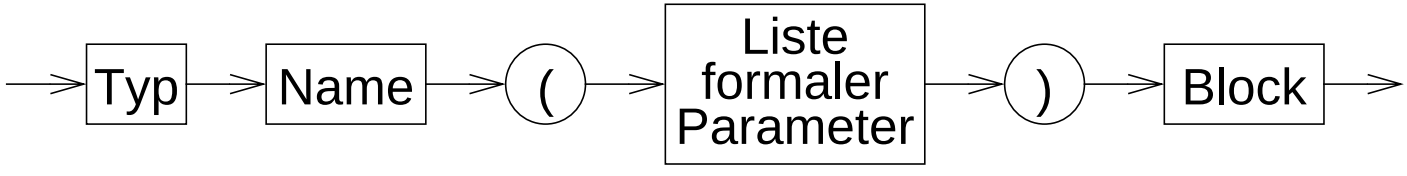
- beliebige Verwendung von **`sinus`** in Ausdrücken:

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

3 Funktionsdefinition

- Schnittstelle (Typ, Name, Parameter) und die Implementierung
  - Beispiel:

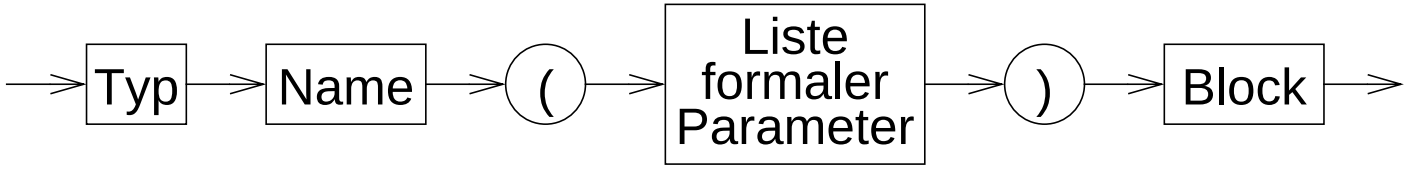
```
int addition ( int a, int b ) {
        int ergebnis;
        ergebnis = a + b;
        return ergebnis;
}
```



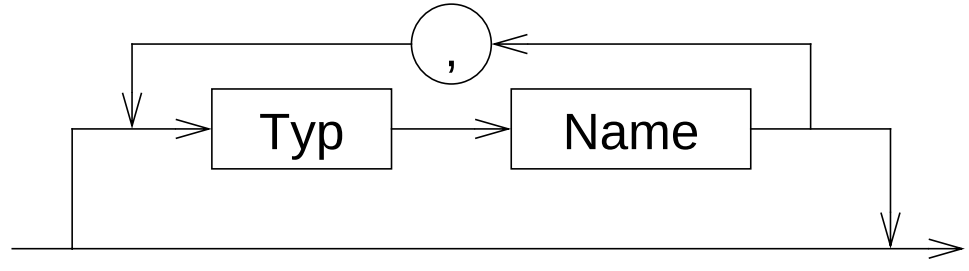
- **Typ**
  - Typ des Werts, der am Ende der Funktion als Wert zurückgegeben wird
  - beliebiger Typ
  - **`void`** = kein Rückgabewert

- **Name**
  - beliebiger Bezeichner, kein Schlüsselwort

3 Funktionsdefinition (2)

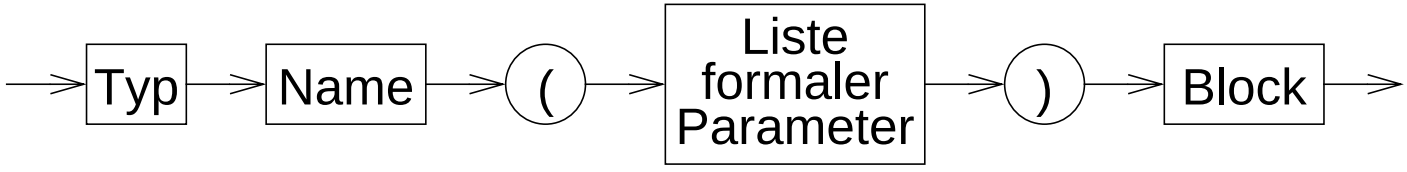


- **Liste formaler Parameter**



  - **Typ**: beliebiger Typ
  - **Name**:
    beliebiger Bezeichner
  - die formalen Parameter stehen innerhalb der Funktion 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

3 Funktionsdefinition (3)



Block

- beliebiger Block
- zusätzliche Anweisung

```
return ( Ausdruck );
```

oder

```
return;
```

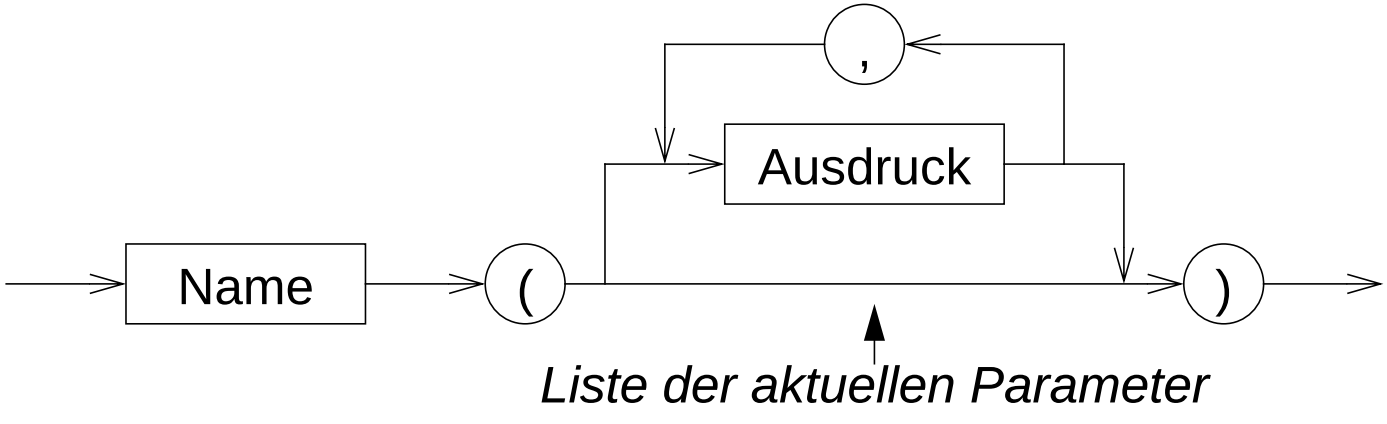
bei **`void`**-Funktionen

- ➤ 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

4 Funktionsaufruf

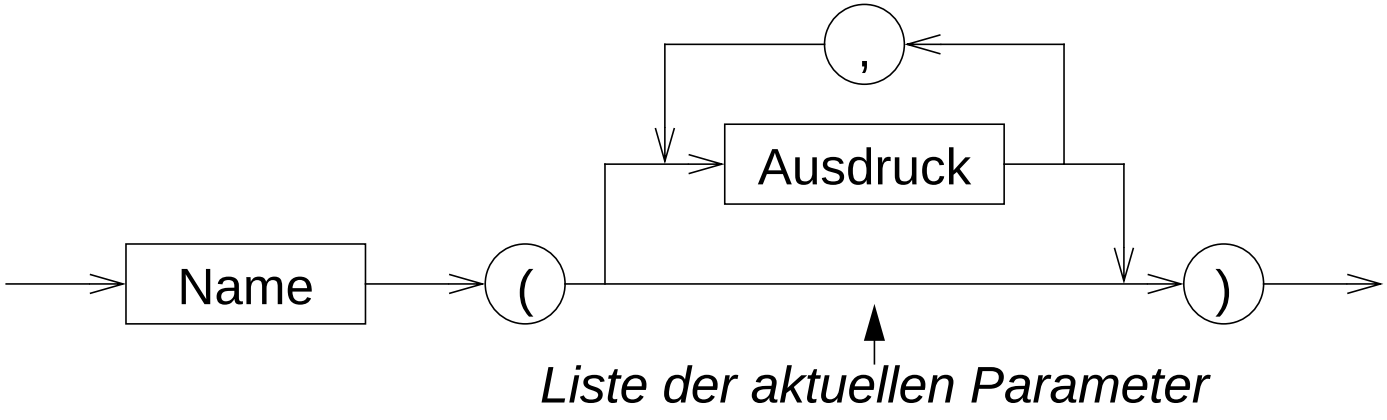
- Aufruf einer Funktion aus dem Ablauf einer anderen Funktion
  - Beispiel:

```
int main ( ) {
        int summe;
        summe = addition(3,4);
        ...
```



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

4 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

5 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;
```

6 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
  - ➤ Ergebnis vom Typ int - wird an die Shell zurückgeliefert (in Kommandoprozeduren z. B. abfragbar)
- 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) );
}
```

6 Regeln (2)

- Funktionen müssen **deklariert** sein, bevor sie aufgerufen werden
  - = Rückgabetyp und Parametertypen müssen dem Compiler 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**

6 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);
```

7 Funktionsdeklarationen — Beispiel

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

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

int main()
{
     double wert;

     printf("Berechnung des Sinus von ");
     scanf("%lf", &wert);
     printf("sin(%lf) = %lf\n",
          wert, sinus(wert));
     return(0);
}
```

```
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);
}
```

8 Parameterübergabe an Funktionen

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

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

8 Parameterübergabe an Funktionen (2)

call by reference

- 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

D.9 Programmstruktur & Module

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

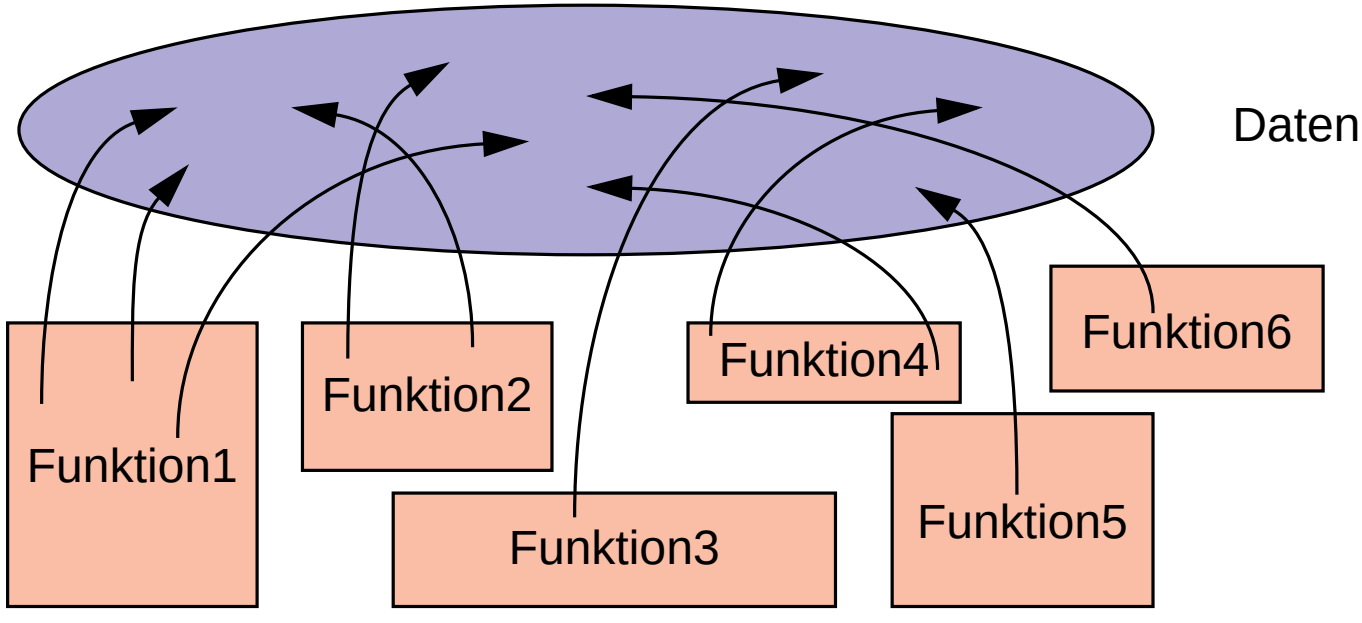
2 Top-down Entwurf

- 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

2 Top-down Entwurf (2)

- Problem:
  Gliederung betrifft nur die Aktivitäten, nicht die Struktur der Daten

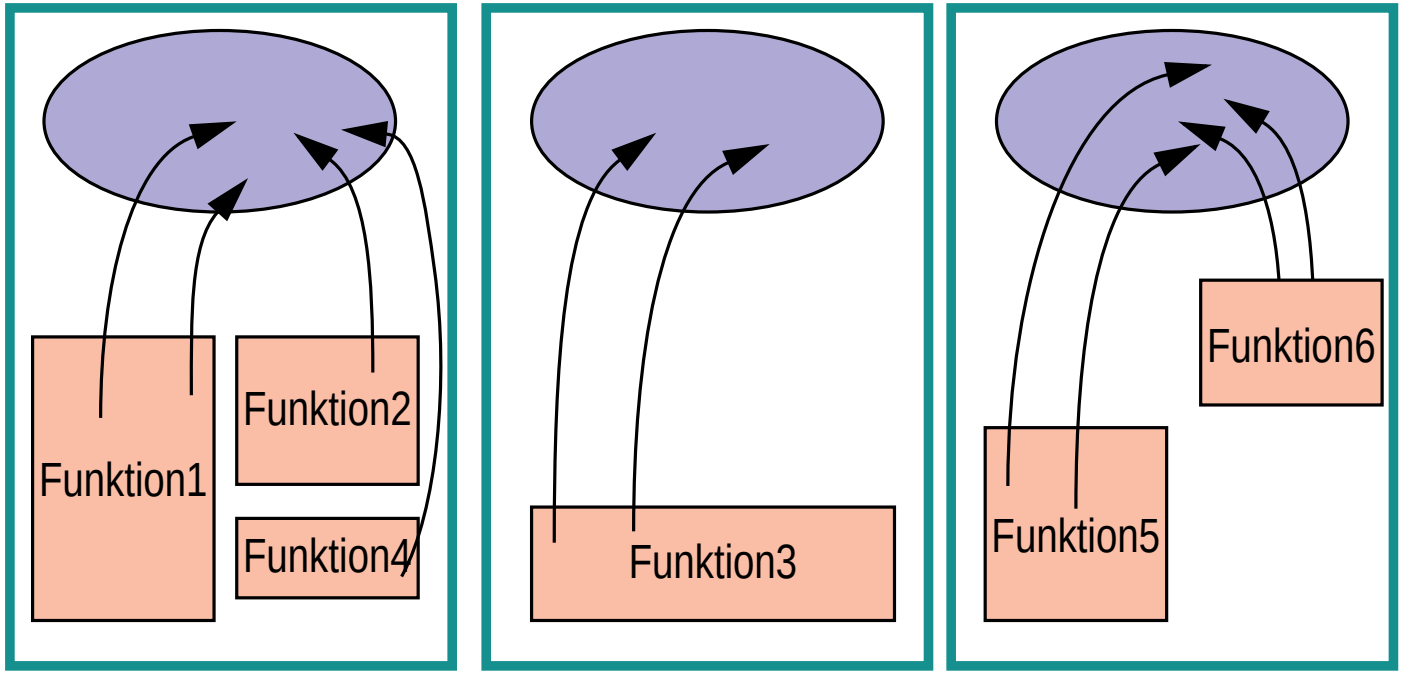
- Gefahr:
  Sehr viele Funktionen arbeiten "wild" auf einer Unmenge schlecht strukturierter Daten



2 Top-down Entwurf (3) Modul-Bildung

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

  ➥ **Modul**



3 Module in C

- 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 -c main.c          (erzeugt Datei main.o)
% cc -c f1.c            (erzeugt Datei f1.o)
% cc -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 -o prog main.o f1.o f2.o f3.o f4.c f5.c
```

3 Module in 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 Präprozessors in C-Quelldateien einkopiert
  - ◆ der Name einer *Header*-Datei endet immer auf **`.h`**

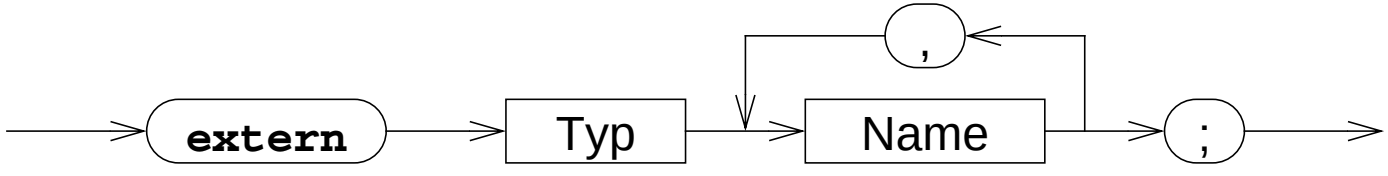
4 Gültigkeit von Namen

- 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 Varibalen in umgebenden Blöcken

5 Globale Variablen

Gültig im gesamten Programm

- 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;
```

5 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!!!**

5 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

6 Einschränkung der Gültigkeit auf ein Modul

- 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**

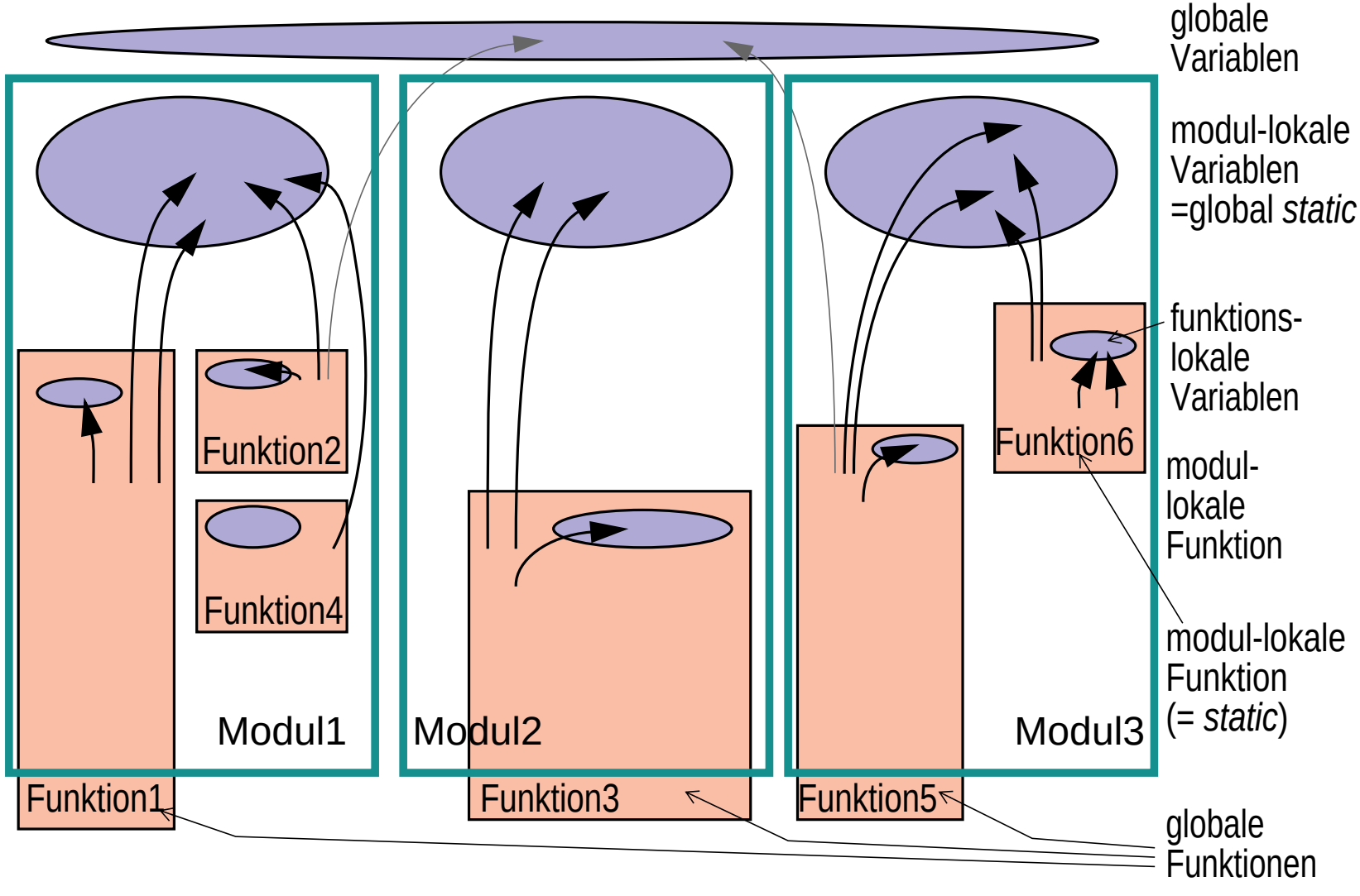
- Hilfsfunktionen innerhalb eines Moduls, die nur von den Modulfunktionen benötigt werden, sollten immer *static* definiert werden
  - ➤ sie werden dadurch nicht Bestandteil der Modulschnittstelle (= des "Vertrags" mit den Modulbenutzern)

!!! das Schlüsselwort **static** gibt es auch bei lokalen Variablen (mit anderer Bedeutung! - zur Unterschiedung ist das hier beschriebene *static* immer kursiv geschrieben)

7 Lokale Variablen

- 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

8 Gültigkeitsbereiche — Übersicht



9 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

9 Lebensdauer von Variablen (2)

auto-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)

9 Lebensdauer von Variablen (2)

static-Variablen

- 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!)
  - ➤ erfolgt keine explizite Initialisierung, wird automatisch mit 0 vorbelegt

10 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;;
        ...
        }
}
```

10 Getrennte Übersetzung — Beispiel (2)

- *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));
}
```

10 Getrennte Übersetzung — Beispiel (3)

- 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);
}
```

D.10Vom C-Quellcode zum laufenden Programm

1 Übersetzen - Objektmodule

- 1. Schritt: Präprozessor
  - entfernt Kommentare, wertet Präprozessoranweisungen aus
    - ➤ fügt include-Dateien ein
    - ➤ expandiert Makros
    - ➤ entfernt Makro-abhängige Code-Abschnitte (*conditional code*) Beispiel:

```
#define DEBUG
...
#ifdef DEBUG
    printf("Zwischenergebnis = %d\n", wert);
#endif DEBUG
```

  - Zwischenergebnis kann mit `cc -P datei.c` als `datei.i` erzeugt werden oder mit `cc -E datei.c` ausgegeben werden

1 Übersetzen - Objektmodule (2)

- 2. Schritt: Compilieren
  - übersetzt C-Code in Assembler
  - Zwischenergebnis kann mit `cc -S datei.c` als `datei.s` erzeugt werden
- 3. Schritt: Assemblieren
  - assembliert Assembler-Code, erzeugt Maschinencode (Objekt-Datei)
  - standardisiertes Objekt-Dateifomat: ELF (Executable and Linking Format) (vereinfachte Darstellung) - in nicht-UNIX-Systemen andere Formate
    - ➤ Maschinencode
    - ➤ Informationen über Variablen mit Lebensdauer static (ggf. Initialisierungswerte)
    - ➤ Symboltabelle: wo stehen welche globale Variablen und Funktionen
    - ➤ Relokierungsinformation: wo werden welche "nicht gefundenen" globalen Variablen bzw. Funktionen referenziert
  - Zwischenergebnis kann mit `cc -c datei.c` als `datei.o` erzeugt werden

2 Binden und Bibliotheken

- 4. Schritt: Binden
  - Programm **`ld`** : (*linker*), erzeugt ausführbare Datei (*executable file*)
    - ➤ ebenfalls ELF-Format (früher a.out-Format oder COFF)
  - Objekt-Dateien (.o-Dateien) werden zusammengebunden
    - ➤ noch nicht abgesättigte Referenzen auf globale Variablen und Funktionen in anderen Objekt-Dateien werden gebunden (Relokation)
  - nach fehlenden Funktionen wird in Bibliotheken gesucht
  - **statisch binden**
    - ➤ alle fehlenden Funktionen werden aus Bibliotheken genommen und in die ausführbare Datei einkopiert
- Ergebnis: ein ausführbares Programm

3 Laden des ausführbaren Programms

- Windows oder Linux
  - Erzeugen einer Umgebung zur Ausführung des Programms:
    - ➥ Prozess
  - **dynamisch binden**
    - ➤ Funktionen, die von mehreren Programmen gemeinsam genutzt werden (*shared libraries, dll*), werden erst beim Start des Programms hinzugeladen

- Mikrocontroller
  - Übertragen des statisch gebundenen Programms in den Programmspeicher des Mikrocontrollers
    - ➥ Flashen

E Mikrocontroller-Programmierung

E.1 Überblick

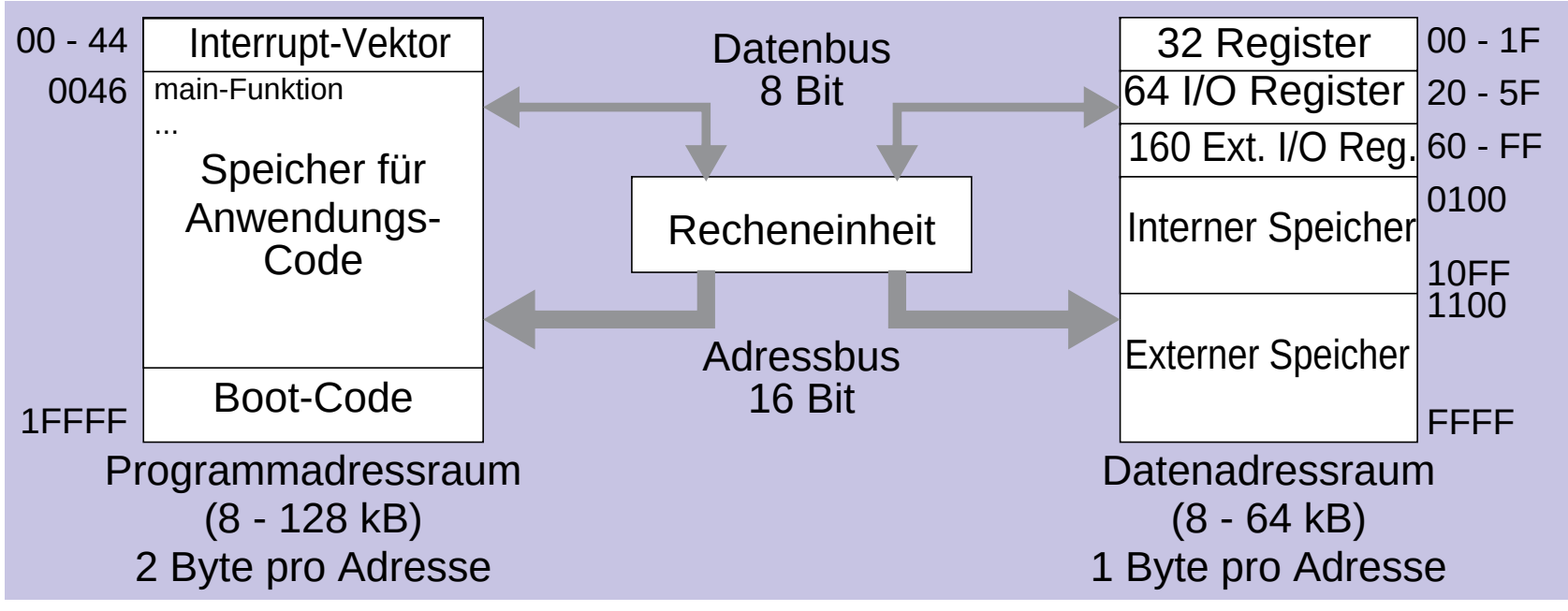
- Mikrocontroller-Umgebung
  - Prozessor am Beispiel AVR-Mikrocontroller
  - Speicher
  - Peripherie
- Programmausführung
  - Programm laden
  - starten
  - Fehler zur Laufzeit
- Interrupts
  - Grundkonzepte
  - Nebenläufigkeit
  - Umgang mit Nebenläufigkeit

1 Mikrocontroller-Umgebung

- Programm läuft "nackt" auf der Hardware
  - ➥ Compiler und Binder müssen ein vollständiges Programm erzeugen
    - ➤ keine Betriebssystemunterstützung zur Laufzeit
  - ◆ Funktionalität muss entweder vom Anwender programmiert werden oder in Form von Funktionsbibliotheken zum Programm dazugebunden werden
  - ◆ Umgang mit "lästigen Programmierdetails" (z. B. bestimmte Bits setzen) wird durch Makros erleichtert
- Es wird genau ein Programm ausgeführt
  - ➤ Programm kann zur Laufzeit "niemanden stören"
  - ➤ Fehler betreffen nur das Programm selbst
  - ➤ keine Schutzmechanismen notwendig
    - ➥ ABER: Fehler ohne direkte Auswirkung werden leichter übersehen

E.2 Beispiel: AVR-Mikrocontroller (ATmega-Serie)

1 Architektur



- Getrennter Speicher für Programm (Flash-Speicher) und Daten (SRAM)
  (Harvard-Architektur — im Gegensatz zur von-Neumann-Architektur beim PC)
- Register des Prozessors und Register für Ein-/Ausgabe-Schnittstellen sind in Adressbereich des Datenspeichers eingebettet

1 Architektur(2)

- Peripherie-Bausteine werden über I/O-Register angesprochen bzw. gesteuert (Befehle werden in den Bits eines I/O-Registers kodiert)
  - ➤ mehrere Timer
    (Zähler, deren Geschwindigkeit einstellbar ist und die bei einem bestimmten Wert einen Interrupt auslösen)
  - ➤ Ports
    (Gruppen von jeweils 8 Anschlüssen, die auf 0V oder V_{cc} gesetzt, bzw. deren Zustand abgefragt werden kann)
  - ➤ Output Compare Modulator (OCM)
    (zur Pulsweitenmodulation)
  - ➤ Serial Peripheral Interface (SPI)
  - ➤ Synchrone/Asynchrone serielle Schnittstelle (USART)
  - ➤ Analog-Comparator
  - ➤ A/D-Wandler
  - ➤ EEPROM
    (zur Speicherung von Konfigurationsdaten)

2 In Speicher eingebettete Register (memory mapped registers)

- Je nach Prozessor sind Zugriffe auf I/O-Register auf zwei Arten realisiert:
  - spezielle Befehle (z. B. in, out bei x86)
  - in den Adressraum eingebettet, Zugriff mittels Speicheroperationen
- Bei den meisten Mikrocontrollern sind die Register in den Speicheradressraum eingebettet
- Zugriffe auf die entsprechende Adresse werden auf das entsprechende Register umgeleitet

➥ sehr einfacher und komfortabler Zugriff

3 I/O-Ports

- Ein I/O-Port ist eine Gruppe von meist 8 Anschluss-Pins und dient zum Anschluss von digitalen Peripheriegeräten
- Die Pins können entweder als Eingang oder als Ausgang dienen
  - als Ausgang konfiguriert, kann man festlegen, ob sie eine logische "1" oder eine logische "0" darstellen sollen
    - ➤ Ausgang wird dann entsprechend auf V_{CC} oder GND gesetzt
  - als Eingang konfiguriert, kann man den Zustand abfragen
- Manche I/O Pins können dazu genutzt werden, einen Interrupt (IRQ = Interrupt Request) auszulösen (externe Interrupt-Quelle)
- Die meisten Pins können alternativ eine Spezialfunktion übernehmen, da sie einem integrierten Gerät als Ein- oder Ausgabe dienen können
  - z. B. dienen die Pins 0 und 1 von Port E (ATmega128) entweder als allgemeine I/O-Ports oder als RxD und TxD der seriellen Schnittstelle

4 Programme laden

- generell bei Mikrocontrollern mehrere Möglichkeiten
- Programm ist schon da (ROM)
- Bootloader-Programm ist da, liest Anwendung über serielle Schnittstelle ein und speichert sie im Programmspeicher ab
- spezielle Hardware-Schnittstelle
  - "jemand anderes" kann auf Speicher zugreifen
  - Beispiel: JTAG
    spezielle Hardware-Komponente im AVR-Chip, die Zugriff auf die Speicher hat und mit der man über spezielle PINs kommunizieren kann

5 Programm starten

- Reset bewirkt Ausführung des Befehls an Adresse 0x0000
  - ➤ dort steht ein Sprungbefehl auf die Speicheradresse einer start-Funktion, die nach einer Initialisierungsphase die main-Funktion aufruft
  - ➤ alternativ: Sprungbefehl auf Adresse des Bootloader-Programms, Bootloader lädt Anwendung, initialisiert die Umgebung und springt dann auf main-Adresse

6 Fehler zur Laufzeit

- Zugriff auf ungültige Adresse
  - ◆ es passiert nichts:
    - ➤ Schreiben geht in's Leere
    - ➤ Lesen ergibt zufälliges Ergebnis
- ungültige Operation auf nur-lesbare / nur-schreibbare Register/Speicher
  - ➤ hat keine Auswirkung

E.3 Interrupts

1 Motivation

- An einer Peripherie-Schnittstelle tritt ein Ereignis auf
  - ➤ Spannung wird angelegt
  - ➤ Zähler ist abgelaufen
  - ➤ Gerät hat Aufgabe erledigt (z. B. serielle Schnittstelle hat Byte übertragen, A/D-Wandler hat neuen Wert vorliegen)
  - ➤ Gerät hat Daten für die Anwendung bereit stehen (z. B. serielle Schnittstelle hat Byte empfangen)

? wie bekommt das Programm das mit?

- ➤ Zustand der Schnittstelle regelmäßig überprüfen (= ***Polling***)
- ➤ Schnittstelle meldet sich von sich aus beim Prozessor und unterbricht den Programmablauf (= ***Interrupt***)

1 Motivation (2)

- Polling vs. Interrupts: Vor und Nachteile
  - Polling
    - \+ Pollen erfolgt **synchron** zum Programmablauf, Programm ist in dem Moment auf das Ereignis vorbereitet
    - – Pollen erfolgt explizit im Programm und meistens umsonst — Rechenzeit wird verschwendet
    - – Polling-Funktionalität ist in den normalen Pragrammablauf eingestreut — und hat mit der "eigentlichen" Funktionalität dort meist nichts zu tun

1 Motivation (3)

- Polling vs. Interrupts: Vor und Nachteile
  - Interrrupts
    - \+ Interrupts melden sich nur, wenn tatsächlich etwas zu erledigen ist
    - \+ Interrupt-Bearbeitung ist in einer Funktion kompakt zusammengefasst
    - – Interrupts unterbrechen den Programmablauf irgendwo (**asynchron**), sie könnten in dem Augenblick stören
      - ➥ durch die Interrupt-Bearbeitungentsteht **Nebenläufigkeit**

2 Implementierung

- typischerweise mehrere Interrupt-Quellen
- Interrupt-Vektor
  - Speicherbereich (Tabelle), der für jeden Interrupt Informationen zur Bearbeitung enthält
    - ➤ Maschinenbefehl
      (typischerweise ein Sprungbefehl auf eine Adresse, an der eine Bearbeitungsfunktion (***Interrupt-Handler***) steht)
      oder
    - ➤ Adresse einer Bearbeitungsfunktion
  - feste Position im Speicher — ist im Prozessorhandbuch nachzulesen
- Maskieren von Interrupts
  - ➤ Bit im Prozessor-Statusregister schaltet den Empfang aller Interrupts ab
  - ➤ zwischenzeitlich eintreffende Interrupts werden gepuffert (nur einer!)
  - ➤ die Erzeugung einzelner Interrupts kann am jeweiligen Gerät unterbunden werden

3 Ablauf auf Hardware-Ebene

① Gerät löst Interrupt aus,
Ablauf des Anwendungsprogramms wird unmittelbar unterbrochen

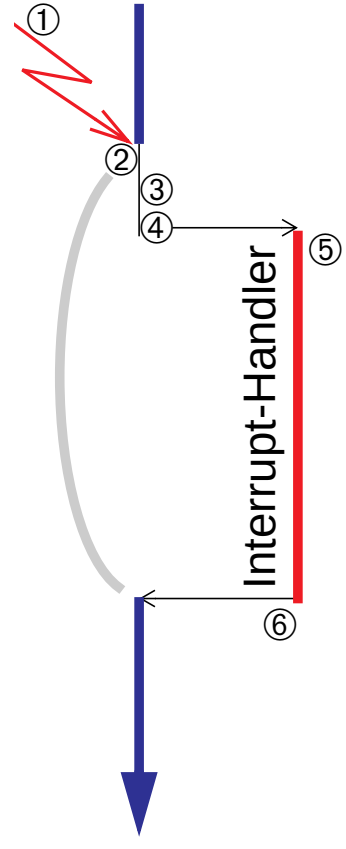
② weitere Interrupts werden deaktiviert

③ aktuelle Position im Programm wird gesichert

④ Eintrag im Interrupt-Vektor ermitteln

⑤ Befehl wird ausgeführt bzw. Funktion aufrufen (= Sprung in den Interrupt-Handler)

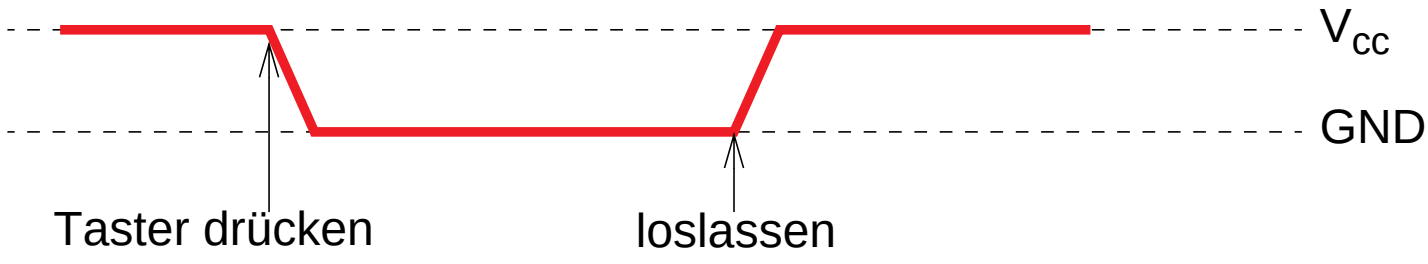
⑥ am Ende der Bearbeitungsfunktion bewirkt ein Befehl "Return from Interrupt" die Fortsetzung des Anwendungsprogramms und die Reaktivierung der Interrupts



! Der Interrupt-Handler muss alle Register, die er ändert am Anfang sichern und vor dem Rücksprung wieder herstellen!

4 Pegel- und Flanken-gesteuerte Interrupts

- Beispiel: Signal eines (idealisierten) Tasters



- Flanken-gesteuert
  - ➤ Interupt wird durch die Flanke (= Wechsel des Pegels) ausgelöst
  - ➤ welche Flanke einen Interrupt auslöst kann bei manchen Prozessoren konfiguriert werden
- Pegel-gesteuert
  - ➤ solange ein bestimmter Pegel anliegt (hier Pegel = GND) wird immer wieder ein Interrupt ausgelöst

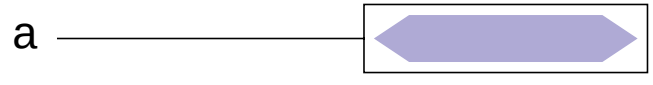
F Zeiger, Felder und Strukturen in C

F.1 Zeiger(-Variablen)

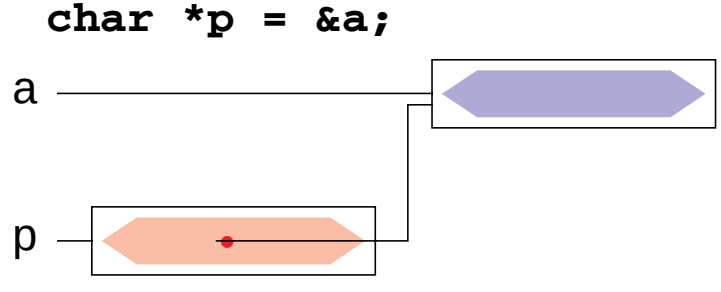
1 Einordnung

- **Literal:**
  Bezeichnung für einen Wert

- **Variable:**
  Bezeichnung eines Datenobjekts



- **Zeiger-Variable (Pointer):**
  Bezeichnung einer Referenz auf ein Datenobjekt



2 Überblick

- Eine Zeigervariable ***(pointer)*** enthält als Wert einen Verweis auf den Inhalt einer anderen Variablen
  - ➥ *der Zeiger verweist auf die Variable*
- Über diese Adresse kann man **indirekt** auf die andere Variable zugreifen
- Daraus resultiert die große Bedeutung von Zeigern in C
  - ➥ Funktionen können ihre Argumente verändern ***(call-by-reference)***
  - ➥ dynamische Speicherverwaltung
  - ➥ effizientere Programme
- Aber auch Nachteile!
  - ➥ Programmstruktur wird unübersichtlicher (welche Funktion kann auf welche Variable zugreifen?)
  - ➥ häufigste Fehlerquelle bei C-Programmen

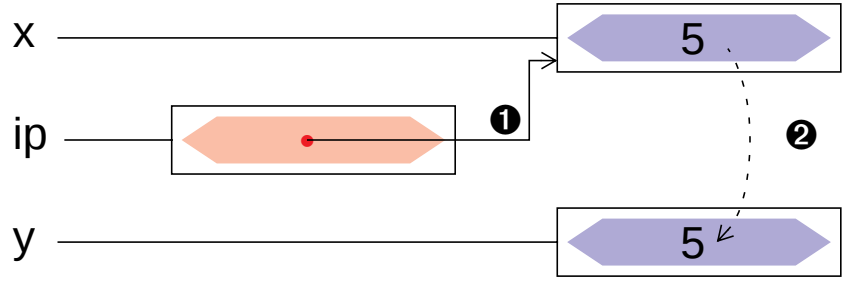
3 Definition von Zeigervariablen

- Syntax:

```
Typ *Name ;
```

4 Beispiele

```
int x = 5;

int *ip;

int y;

ip = &x; ❶

y = *ip; ❷
```



5 Adressoperatoren

- Adressoperator `&`

  `&x` der unäre Adress-Operator liefert eine Referenz auf den Inhalt der Variablen (des Objekts) `x`

- Verweisoperator `*`

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

6 Zeiger als Funktionsargumente

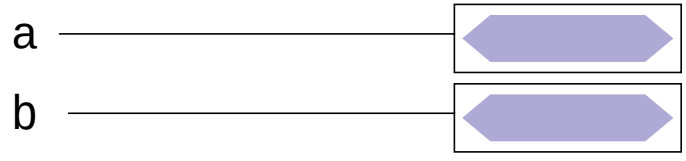
- Parameter werden in C *by-value* übergeben
- die aufgerufene Funktion kann den aktuellen Parameter beim Aufrufer nicht verändern
- auch Zeiger werden *by-value* übergeben, d. h. die Funktion erhält lediglich eine Kopie des Adressverweises
- über diesen Verweis kann die Funktion jedoch mit Hilfe des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

  ➥ *call-by-reference*

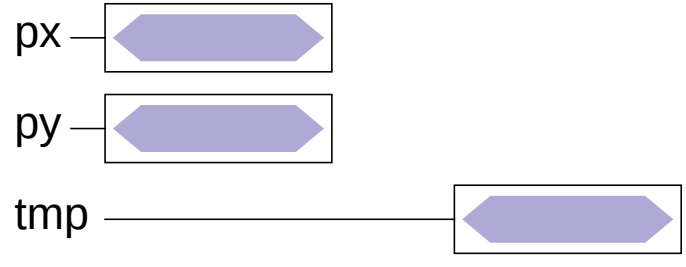
6 … Zeiger als Funktionsargumente (2)

- Beispiel:

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



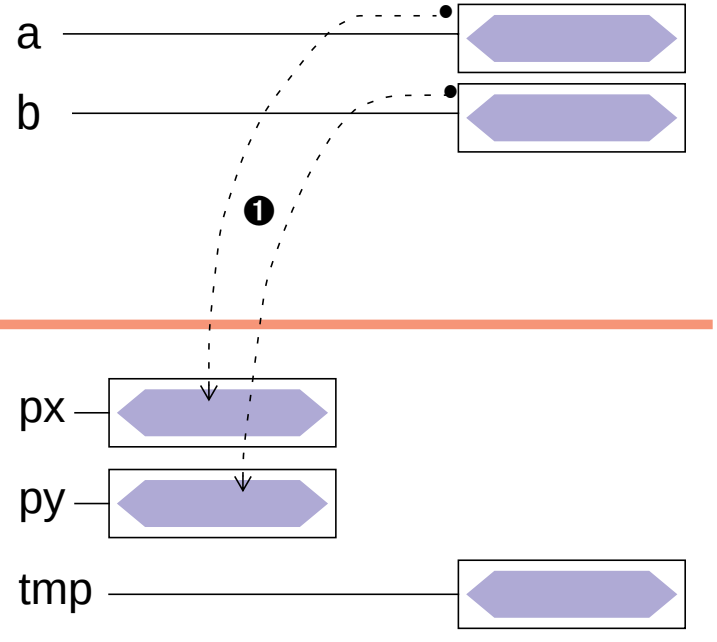

```
void swap (int *px, int *py)
{
    int tmp;

    tmp = *px;
    *px = *py;
    *py = tmp;

}
```



6 Zeiger als Funktionsargumente (2)

- Beispiel:

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

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

    tmp = *px;
    *px = *py;
    *py = tmp;

}
```

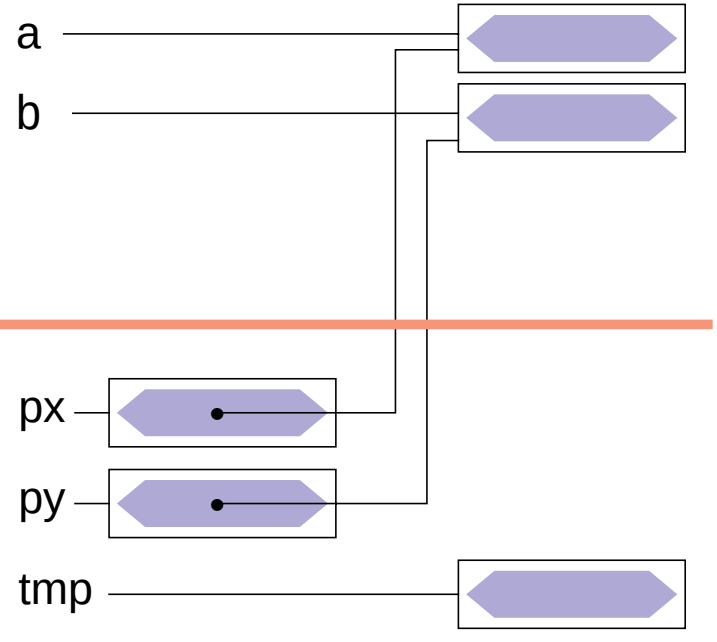


6 Zeiger als Funktionsargumente (2)

- Beispiel:

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

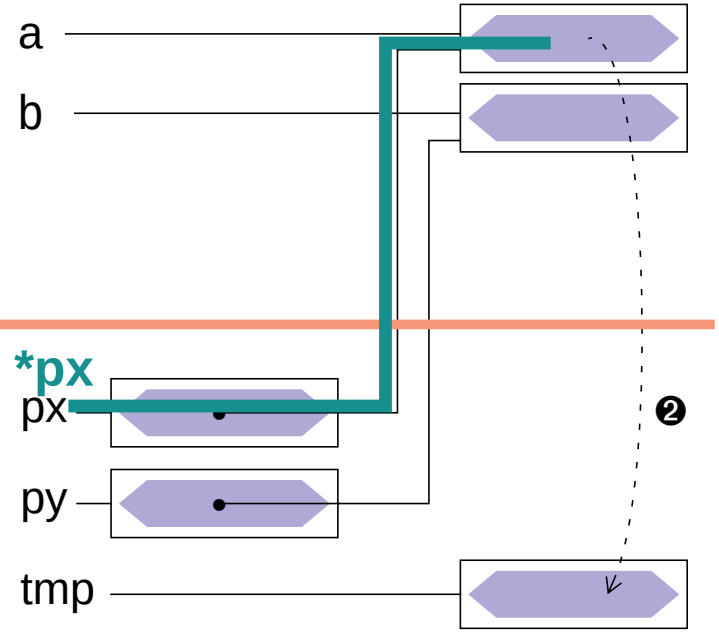
```
void swap (int *px, int *py)
{
    int tmp;

    tmp = *px;
    *px = *py;
    *py = tmp;

}
```



6 Zeiger als Funktionsargumente (2)

- Beispiel:

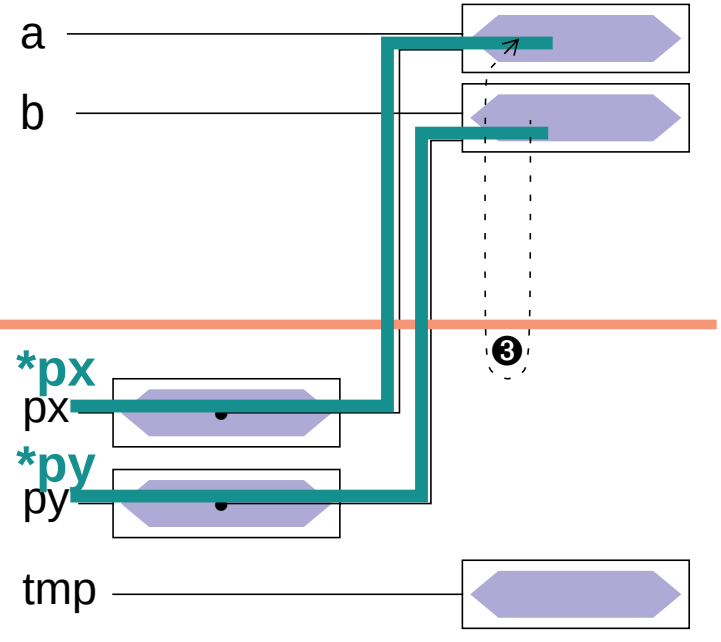
```
void swap (int *, int *);
int main() {
    int a, b;
    ...
    swap(&a, &b);
    ...
}

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

    tmp = *px; ❷
    *px = *py;
    *py = tmp;

}
```



6 Zeiger als Funktionsargumente (2)

- Beispiel:

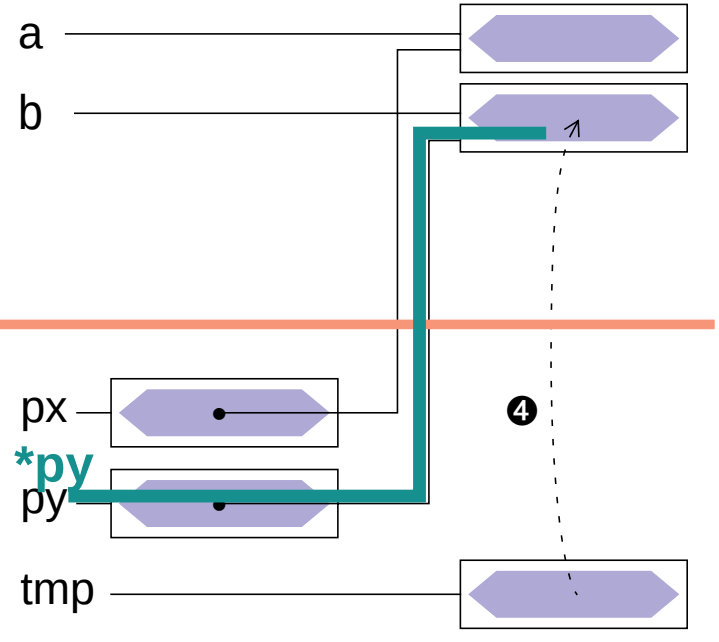
```
void swap (int *, int *);
int main() {
    int a, b;
    ...
    swap(&a, &b);
    ...
}

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

    tmp = *px;
    *px = *py; ❸
    *py = tmp;

}
```



6 Zeiger als Funktionsargumente (2)

- Beispiel:

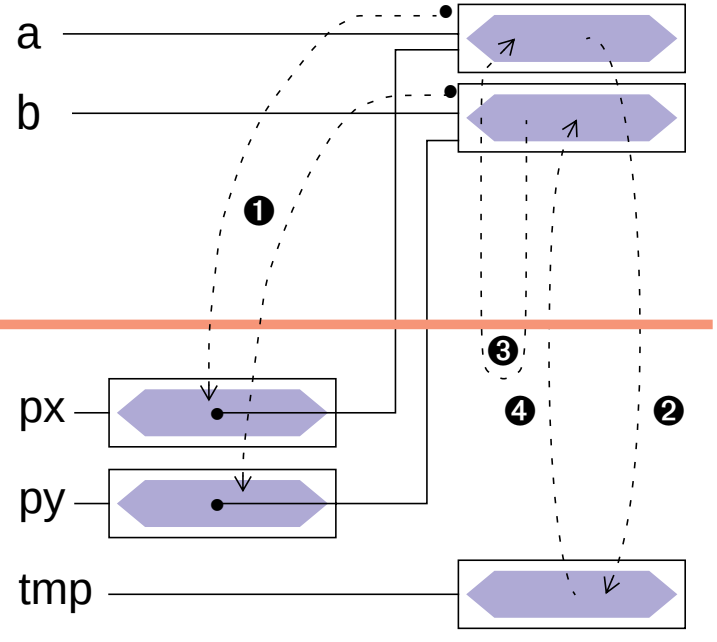
```
void swap (int *, int *);
int main() {
    int a, b;
    ...
    swap(&a, &b);
    ...
}

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

    tmp = *px;
    *px = *py;
    *py = tmp; ❹

}
```



6 Zeiger als Funktionsargumente (2)

- Beispiel:

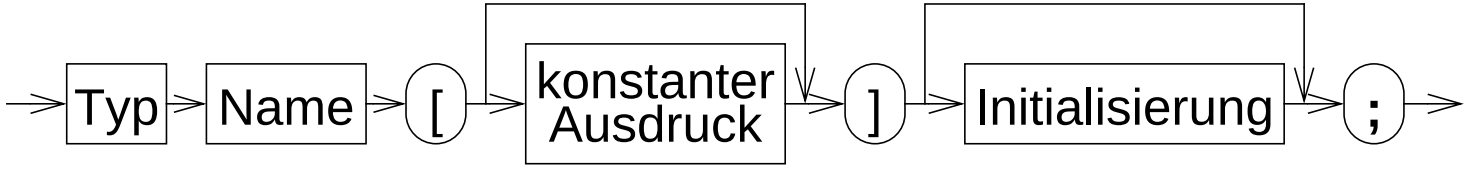
```
void swap (int *, int *);
int main() {
    int a, b;
    ...
    swap(&a, &b); ❶
    ...
}

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

    tmp = *px; ❷
    *px = *py; ❸
    *py = tmp; ❹

}
```



F.2 Felder

1 Eindimensionale Felder

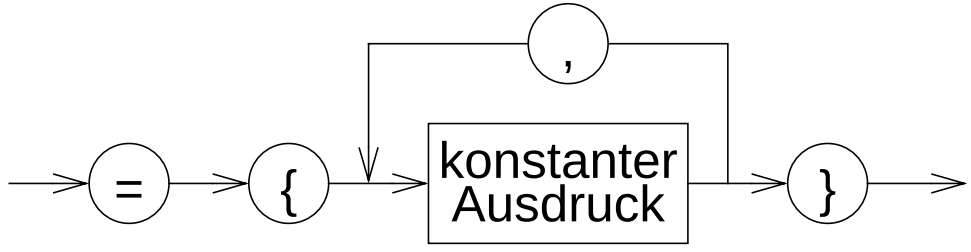
- eine Reihe von Daten desselben Typs kann zu einem **Feld** zusammengefasst werden
- bei der Definition wird die Anzahl der Feldelemente angegeben, die Anzahl ist konstant!
- der Zugriff auf die Elemente erfolgt durch **Indizierung**, beginnend bei Null
- Definition eines Feldes



- Beispiele:

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

2 Initialisierung eines Feldes



- Ein Feld kann durch eine Liste von konstanten Ausdrücken, die durch Komma getrennt sind, initialisiert werden

```
int prim[4] = {2, 3, 5, 7};
char name[5] = {'O', '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[] = {'O', 't', 't', 'o', '\0'};
```

- werden zu wenig Initialisierungskonstanten angegeben, so werden die restlichen Elemente mit 0 initialisiert

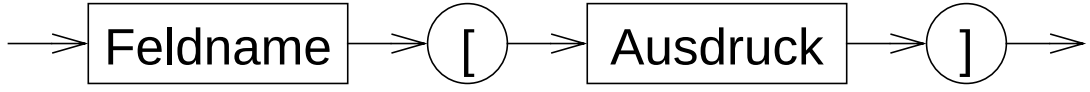
2 … Initialisierung eines Feldes (2)

- Felder des Typs ***char*** können auch durch String-Literale initialisiert werden

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

3 Zugriffe auf Feldelemente

- Indizierung:



wobei: 0 <= Wert(Ausdruck) < Feldgröße

- Beispiele:

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

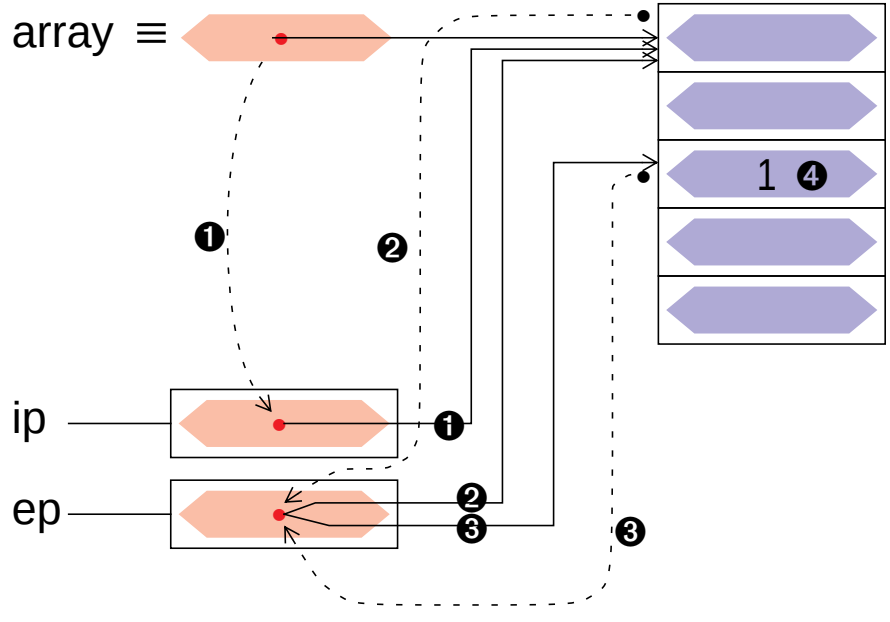
- Beispiel Vektoraddition:

```
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]);
```

F.3 Zeiger und Felder

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

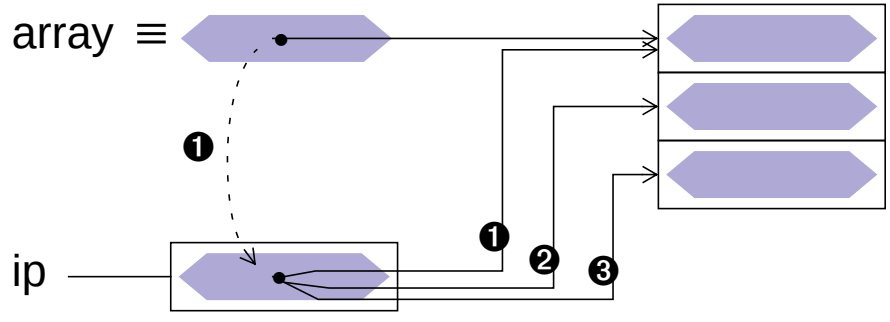
```
int array[5];

int *ip = array; ❶

int *ep;
ep = &array[0]; ❷

ep = &array[2]; ❸

*ep = 1; ❹
```



1 Arithmetik mit Adressen

- `++` -Operator: Inkrement = nächstes Objekt

```
int array[3];
int *ip = array; ❶

ip++; ❷
ip++; ❸
```

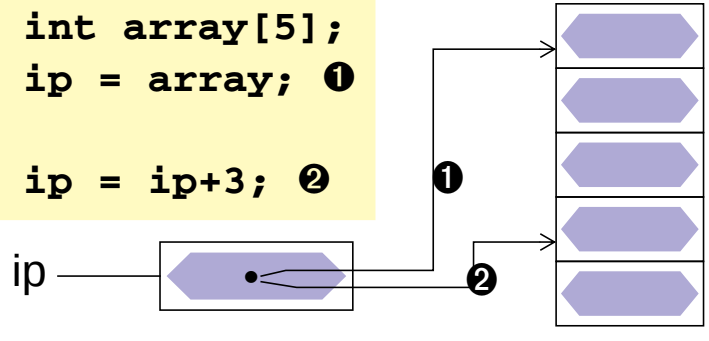


- `--` -Operator: Dekrement = vorheriges Objekt

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

  Dabei wird immer die Größe des Objekttyps berücksichtigt!

```
int array[5];
ip = array; ❶

ip = ip+3; ❷
```



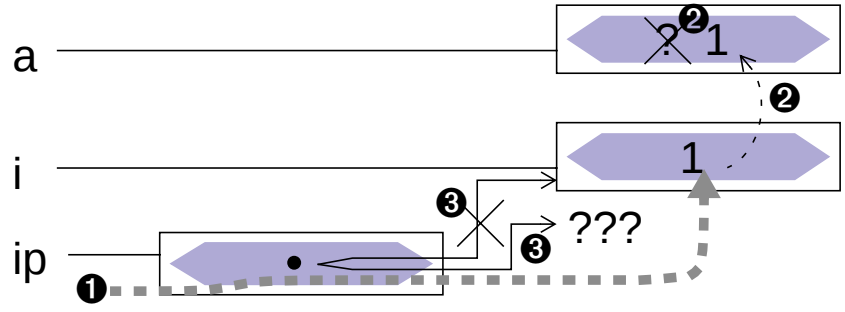
!!! **Achtung:** Assoziativität der Operatoren beachten !!

2 Vorrangregeln bei Operatoren

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

3 Beispiele

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

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

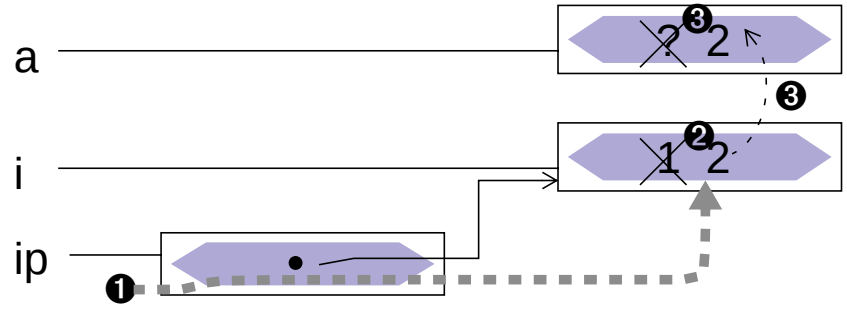


3 Beispiele (2)

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

a = ++*ip;
```

(1) a = ++*ip;
❶*ip
❷++ → *ip

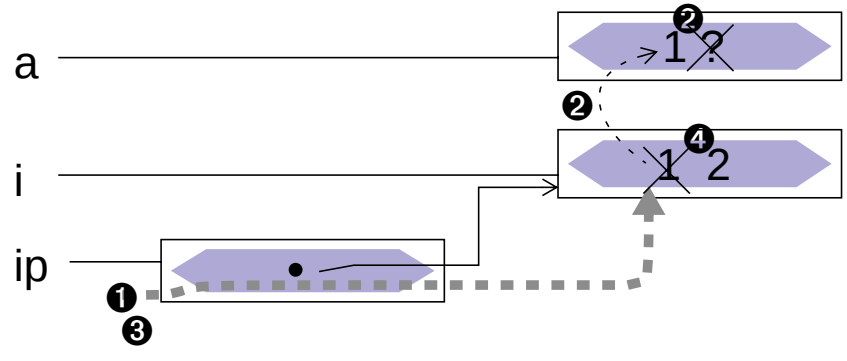
(2) a = ++*ip;
❸




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

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

(1) a = (*ip)++;
❷ ❶

(2) a = (*ip)++;
*ip❸
❹++ → *ip



4 Zeigerarithmetik und Felder

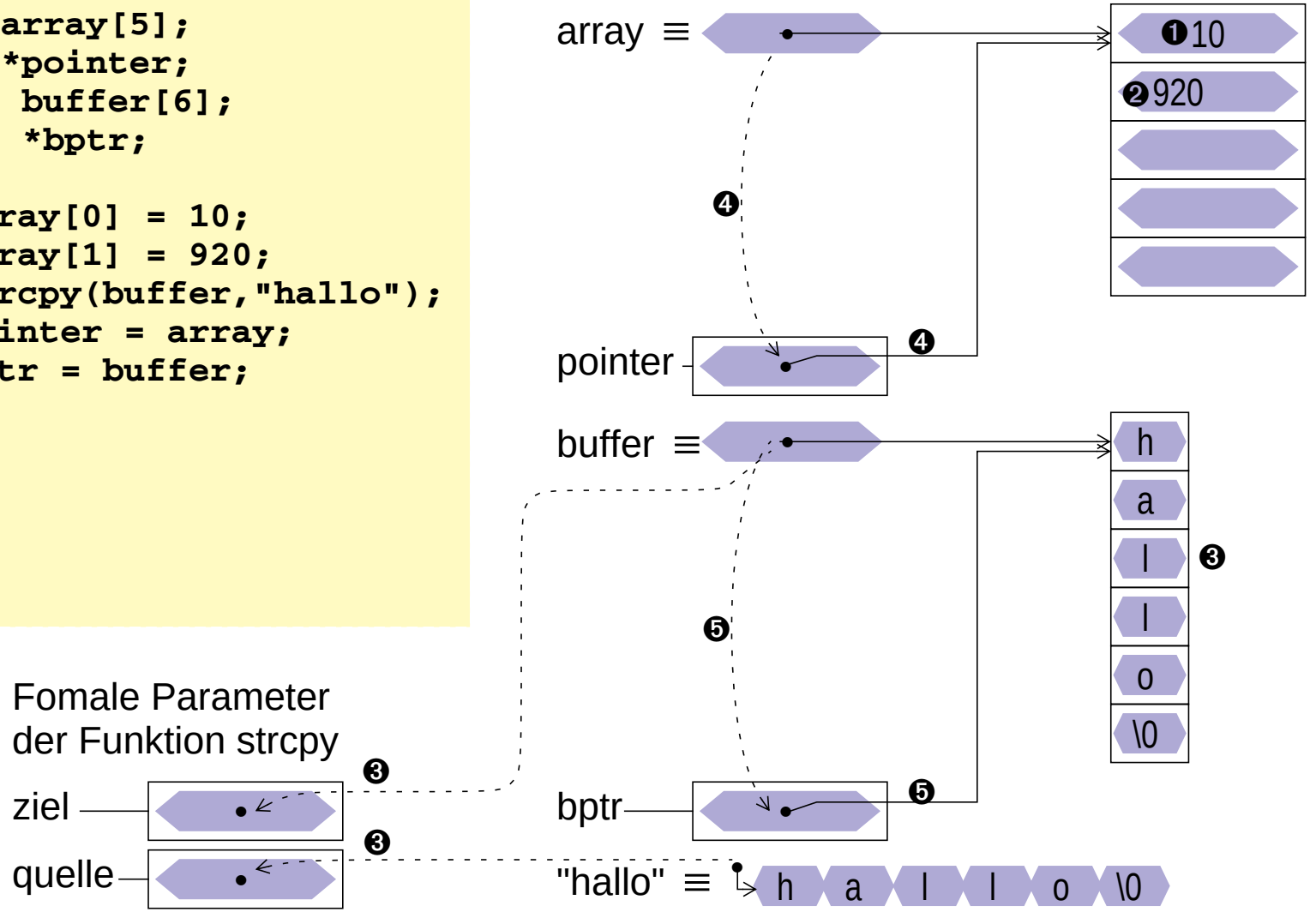
- Ein Feldname ist eine Konstante, für die Adresse des Feldanfangs
  - ➥ Feldname ist ein ganz normaler Zeiger
    - ➤ Operatoren für Zeiger anwendbar (**`*`**, **`[]`**)
  - ➥ aber keine Variable ➔ keine Modifikationen erlaubt
    - ➤ keine Zuweisung, kein **`++`**, **`--`**, **`+=`**, …

- es gilt:

```
int array[5]; /* → array ist Konstante für den Wert &array[0] */
int *ip = array;/* ≡ int *ip = &array[0] */
int *ep;

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

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

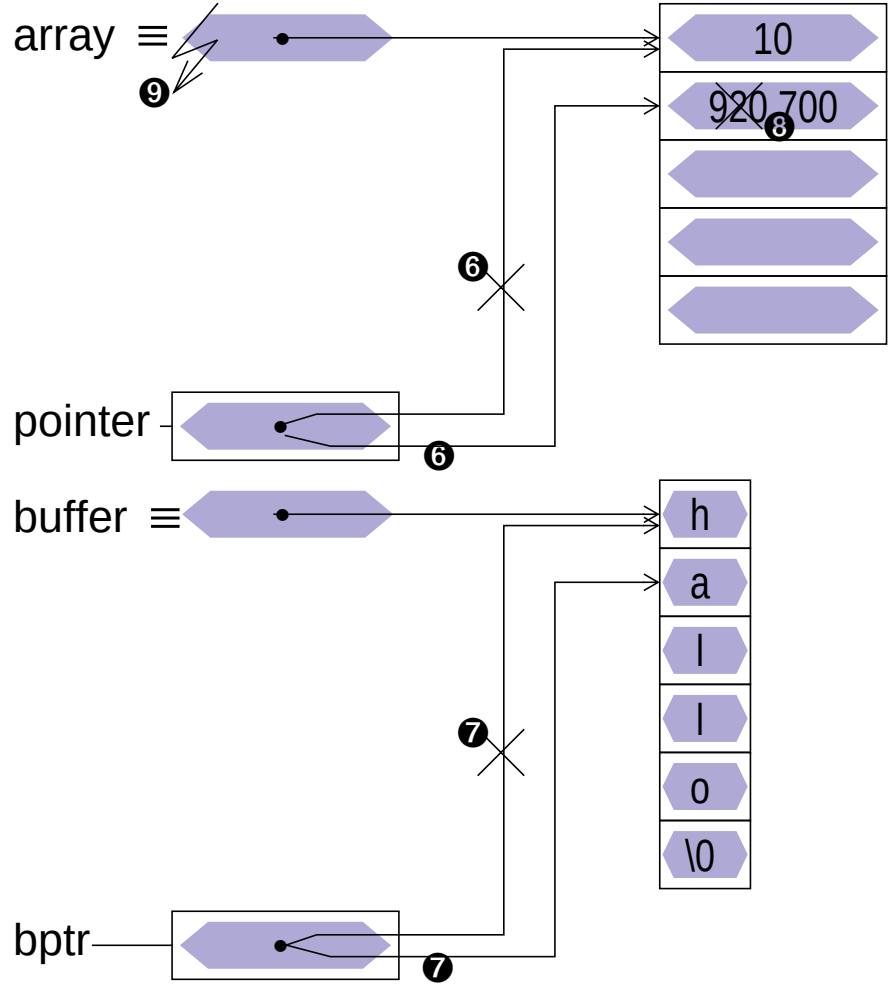
4 Zeigerarithmetik und Felder

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

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



4 Zeigerarithmetik und Felder

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

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

❻ pointer++;
❼ bptr++;
❽ *pointer = 700;

❾ array++;
```



5 Vergleichsoperatoren und Adressen

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

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

F.4 Eindimensionale Felder als Funktionsparameter

- ganze Felder können in C **nicht *by-value*** übergeben werden
- wird einer Funktion ein Feldname als Parameter übergeben, wird damit der Zeiger auf das erste Element "by value" übergeben
  - ➥ die Funktion kann über den formalen Parameter (=Kopie des Zeigers) in gleicher Weise wie der Aufrufer auf die Feldelemente zugreifen (und diese verändern!)
- bei der Deklaration des formalen Parameters wird die Feldgröße weggelassen
  - ➤ die Feldgröße ist automatisch durch den aktuellen Parameter gegeben
  - ➤ die Funktion kennt die Feldgröße damit nicht
  - ➤ ggf. ist die Feldgröße über einen weiteren **`int`**-Parameter der Funktion explizit mitzuteilen
  - ➤ die Länge von Zeichenketten in **`char`**-Feldern kann normalerweise durch Suche nach dem **`\0`**-Zeichen bestimmt werden

F.4 Eindimensionale Felder als Funktionsparameter (2)

- wird ein Feldparameter als **`const`** deklariert, können die Feldelemente innerhalb der Funktion nicht verändert werden

- Funktionsaufruf und Deklaration der formalen Parameter am Beispiel eines **`int`**-Feldes:

```
int a, b;
int feld[20];
func(a, feld, b);
…
int func(int p1, int p2[], int p3);
oder:
int func(int p1, int *p2, int p3);
```

- die Parameter-Deklarationen **`int p2[]`** und **`int *p2`** sind vollkommen äquivalent!

  ➤ im Unterschied zu einer Variablendefinition!!!

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

F.5 Dynamische Speicherverwaltung

- Felder können (mit einer Ausnahme im C99-Standard) nur mit statischer Größe definiert werden
- Wird die Größe eines Feldes erst zur Laufzeit des Programm bekannt, kann der benötigte Speicherbereich dynamisch vom Betriebssystem angefordert werden: Funktion **`malloc`**
  - ➤ Ergebnis: Zeiger auf den Anfang des Speicherbereichs
  - ➤ Zeiger kann danach wie ein Feld verwendet werden (`[]`-Operator)
- **`void *malloc(size_t size)`**

```
int *feld;
int groesse;
...
feld = (int *) malloc(groesse * sizeof(int));
if (feld == NULL) {
        perror("malloc feld");
        exit(1);
}
for (i=0; i<groesse; i++) { feld[i] = 8; }
...
```

cast-Operator

sizeof-Operator

F.5 Dynamische Speicherverwaltung (2)

- Dynamisch angeforderte Speicherbereiche können mit der **`free`**-Funktion wieder freigegeben werden
- **`void free(void *ptr)`**

```
double *dfeld;
int groesse;
...
dfeld = (double *) malloc(groesse * sizeof(double));
...
free(dfeld);
```

- die Schnittstellen der Funktionen sind in in der include-Datei stdlib.h definiert

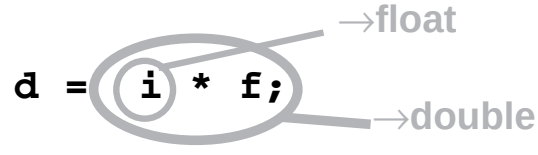
  `#include <stdlib.h>`

F.6 Explizite Typumwandlung — Cast-Operator

- C enthält Regeln für eine automatische Konvertierung unterschiedlicher Typen in einem Ausdruck (vgl. Abschnitt D.5.10)

Beispiel:

```
int i = 5;
float f = 0.2;
double d;
```



- In manchen Fällen wird eine explizite Typumwandlung benötigt (vor allem zur Umwandlung von Zeigern)
  - Syntax:

(Typ) Variable

Beispiele:

```
(int) a
(float) b
```

```
(int *) a
(char *) a
```

  - Beispiel:

```
feld = (int *) malloc(groesse * sizeof(int));
```

malloc liefert Ergebnis vom Typ (void *)

cast-Operator macht daraus den Typ (int *)

F.7 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

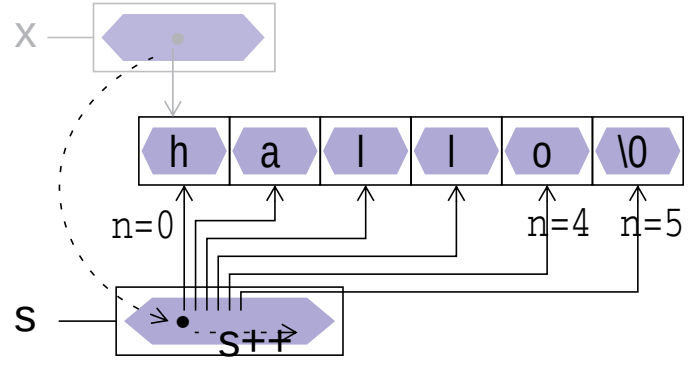
- Das Ergebnis ist vom Typ **`size_t`** (≡ **`int`**)
  (**`#include <stddef.h>`**!)
- Beispiel:

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

F.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(const char *s)
{
    int n;
    for (n=0; *s != '\0'; s++)
        n++;
    return(n);
}
```



F.9 Felder von Zeigern

- Auch von Zeigern können Felder gebildet werden
- Deklaration

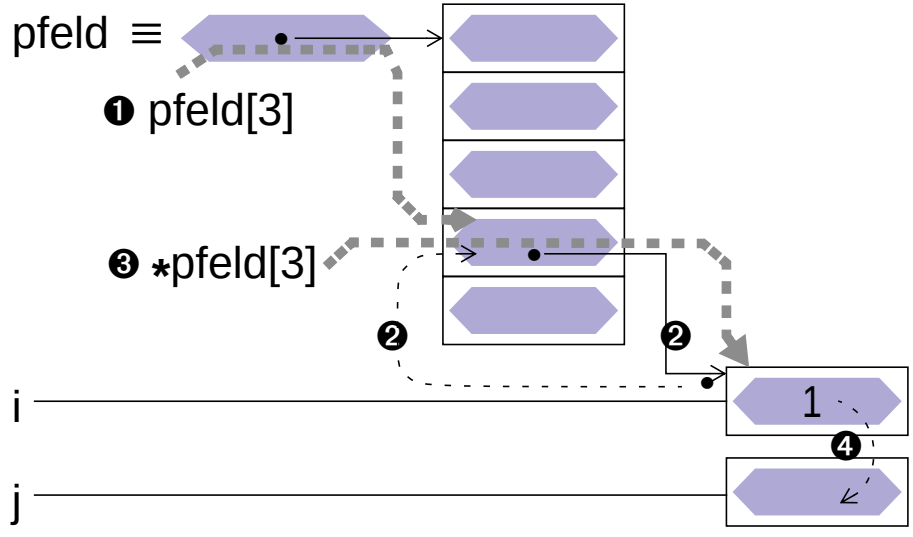
```
int *pfeld[5];
int i = 1
int j;
```

- Zugriffe auf einen Zeiger des Feldes

```
pfeld[3] = &i; ❷
❶
```

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

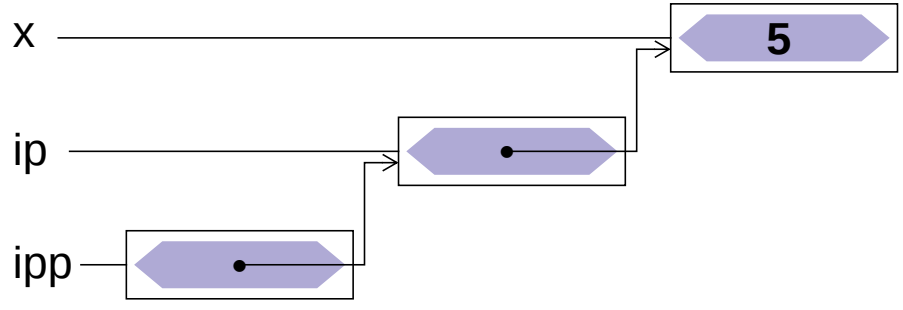
```
j = *pfeld[3]; ❹
    ❶
  ❸
❹
```



F.10 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)

F.11 Zeiger auf Funktionen

- Datentyp: Zeiger auf Funktion
  - Variablendef.: <*Rückgabetyp*> **(*<*Variablenname*>**)(**<*Parameter*>**);**

```
void (*fptr)(int, char*);

void test1(int a, char *s) { printf("1: %d %s\n", a, s); }
void test2(int a, char *s) { printf("2: %d %s\n", a, s); }

fptr = test1;

fptr(42,"hallo");

fptr = test2;

fptr(42,"hallo");
```

F.12 Strukturen

1 Motivation

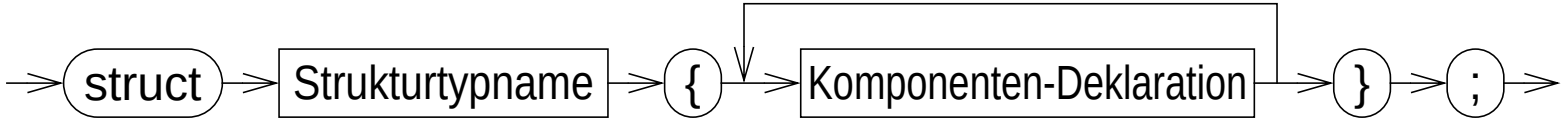
- Felder fassen Daten eines einheitlichen Typs zusammen
  - ➤ ungeeignet für gemeinsame Handhabung von Daten unterschiedlichen Typs
- Beispiel: Daten eines Studenten

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

- Möglichkeiten der Repräsentation in einem Programm
  - ◆ einzelne Variablen ➔ sehr umständlich, keine "Abstraktion"
  - ◆ Datenstrukturen

2 Deklaration eines Strukturtyps

- 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)

2 Deklaration eines Strukturtyps (2)

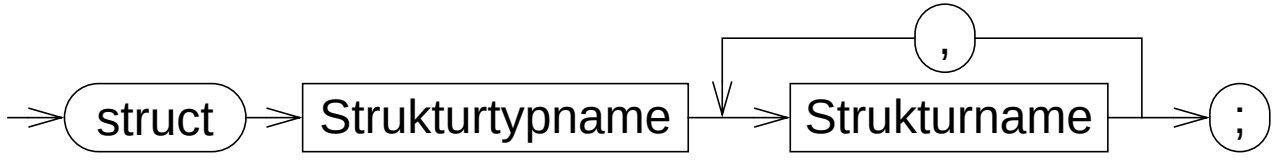
- Beispiele

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

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

3 Definition einer Struktur

- 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

4 Zugriff auf Strukturkomponenten

- `.`-Operator
  - `x.y` ≡ Zugriff auf die Komponente `y` der Struktur `x`
  - `x.y` verhält sich wie eine normale Variable vom Typ der Strukturkomponenten `y` der Struktur `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';
}
```

5 Initialisieren von Strukturen

- Strukturen können — wie Variablen und Felder — bei der Definition initialisiert werden

- Beispiele

```
struct student stud1 = {
    "Meier", "Hans", "24.01.1970", 1533180, 5, 'n'
};

struct komplex c1 = {1.2, 0.8}, c2 = {0.5, 0.33};
```

!!! **Vorsicht**
bei Zugriffen auf eine Struktur werden die Komponenten durch die Komponentennamen identifiziert,
bei der Initialisierung jedoch nur durch die Postion

➥ potentielle Fehlerquelle bei Änderungen der Strukturtyp-Deklaration

6 Strukturen als Funktionsparameter

- Strukturen können wie normale Variablen an Funktionen übergeben werden
  - ◆ Übergabesemantik: **call by value**
    - ➤ Funktion erhält eine Kopie der Struktur
    - ➤ auch wenn die Struktur ein Feld enthält, wird dieses komplett kopiert!
    - !!! Unterschied zur direkten Übergabe eines Feldes
- Strukturen können auch Ergebnis einer Funktion sein
  - ➤ Möglichkeit mehrere Werte im Rückgabeparameter zu transportieren
- Beispiel

```
struct komplex komp_add(struct komplex x, struct komplex y) {
    struct komplex ergebnis;
    ergebnis.re = x.re + y.re;
    ergebnis.im = x.im + y.im;
    return(ergebnis);
}
```

7 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

7 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';
```

8 Felder von Strukturen

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

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

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

9 Strukturen in Strukturen

- 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') {
    …
```

10 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

F.13 Verbundstrukturen — Unions

- In einer Struktur liegen die einzelnen Komponenten hintereinander, in einem Verbund liegen sie übereinander
  - ➤ die gleichen Speicherzellen können unterschiedlich angesprochen werden
  - ➤ Beispiel: ein int-Wert (4 Byte) und die einzelnen Bytes des int-Werts

```
union intbytes {
    int ivalue;
    char bvalue[4];
} u1;
...
u1.ivalue = 259000;
printf("Wert=%d, Byte0=%d, Byte1=%d, Byte2=%d, Byte3=%d\n",
   u1.ivalue, u1.bvalue[0], u1.bvalue[1], u1.bvalue[2], u1.bvalue[3]);
```

- Einsatz nur in sehr speziellen Fällen sinnvoll
  konkretes Wissen über die Speicherorganisation unbedingt erforderlich!

F.14 Bitfelder

- Bitfelder sind Strukturkomponenten bei denen die Zahl der für die Speicherung verwendeten Bits festgelegt werden kann.
- Anwendung z. B. um auf einzelnen Bits eines Registers zuzugreifen

```
struct cregister {
    unsigned int protection : 1;
    unsigned int interrupt_mask : 3;
    unsigned int enable_read : 1;
    unsigned int enable_write : 1;
    unsigned int pedding : 2;
    unsigned int address : 8;
};
```

F.14 Bitfelder (2)

- Struktur mit Bitfeld kann ihrerseits Teil einer Union sein
  - ➤ Zugriff auf Register als Ganzes und auf die einzelnen Bits

```
union cregister {
    unsigned short all;
    struct bits {
        unsigned int protection : 1;
        unsigned int interrupt_mask : 3;
        unsigned int enable_read : 1;
        unsigned int enable_write : 1;
        unsigned int pedding : 2;
        unsigned int address : 8;
    };
};
```

- Adresse und Aufbau eines Registers steht üblicherweise in der Hardwarebeschreibung.

F.14 Bitfelder (3)

- Beispiel:
  - ➤ Adresse auf Register anlegen,
    Registerinhalt sichern
    Bits verändern
    Registerinhalt wieder herstellen

```
union cregister *creg;
unsigned short oldvale;
creg = (union cregister *)0x2400; /* Addr. aus Manual */
oldvalue = creg->all;             /* Wert sichern */
creg->bits.protection = 0;
creg->bits.enable_read = 1;
creg->bits.address = 0x40;
...
creg->all = oldvalue;             /* Wert restaurieren */
```

F.15 Typedef

- Typedef erlaubt die Definition neuer Typen
  - ➤ neuer Typ kann danach wie die Standardtypen (int, char, ...) genutzt werden
- Beispiele:

```
typdef int Laenge;
Laenge l = 5;
Laenge *pl; Laenge fl[20];
```

```
typdef struct student Student;
Student s; Student *ps1;
s.matrnr = 1234567; ps1 = &s; ps1->best = 'n';
```

```
typdef struct student *Studptr;
Studptr ps2;
ps2 = &s; ps2->best = 'n';
```

F.16 Enumerations

- Enumerations sind Datentypen, für die explizit angegeben wird, welche Werte (symbolische Namen) sie annehmen können
  - ➤ interne Darstellung als int (Werte beginnend ab 0 oder explizit angebbar)
- Beispiele:

```
enum led {
  RED0, YELLOW0, GREEN0, BLUE0, RED1, YELLOW1, GREEN1, BLUE1
};
enum led signal;
signal = GREEN0;
```

```
typedef enum {
  BUTTON0 = 4, BUTTON1 = 8
} BUTTON;
BUTTON b = BUTTON0;
```

- Effekt auch mit `#define` - Makros erreichbar
  - ➤ Vorteil von enum: Compiler kennt den Typ und ***könnte*** Ausdrücke prüfen (Nachteil: die meisten Compiler tun es nicht ⇒ `enum` ≡ `int`)

G Nebenläufigkeit

G.1 Überblick

- Definition von Nebenläufigkeit:
  zwei Programmausführungen sind nebenläufig, wenn für zwei einzelne Befehle a und b aus beiden Ausführungen nicht feststeht, ob a oder b tatsächlich zuerst ausgeführt wird

- Nebenläufigkeit tritt auf
  - ➤ bei Interrupts
  - ➤ bei parallelen Abläufen (gleichzeitige Ausführung von Code in einem Mehrprozessorsystem mit Zugriff auf den gleichen Speicher)
  - ➤ bei quasi-parallelen Abläufen (wenn ein Betriebssystem verschiedenen Prozesse den Prozessor jeweils für einen Zeitraum zuteilt und ihn nach Ablauf der Zeit wieder entzieht)

- Problem:
  - ➤ was passiert, wenn die nebenläufigen Ausführungen auf die gleichen Daten im Speicher zugreifen?

G.2 Nebenläufigkeit durch Interrupts

- Interrupts unterbrechen Anwendungsprogramme "irgendwo"
- Interrupts haben Zugriff auf den gleichen Speicher
- Szenario:
  - ➤ eine Lichtschranke soll Fahrzeuge zählen und alle 10 Sekunden soll der Wert ausgegeben werden

```
static int a;

void main(void) {
  long i;
  while(1) {
    for (i=0; i<2000000; i++)
    /* Zählen dauert 10 Sek. */;
    print(a);
    a=0;
  }
}
```

```
/* Lichtschranken-
   Interrupt */
void count(void) {
        a++;
}
```

G.2 Nebenläufigkeit durch Interrupts (2)

- Auf C-Ebene führt die Interrupt-Behandlung nur einen Befehl aus: a++
  - ➤ nur scheinbar ein Befehl
  - ➤ auf Maschinencode-Ebene (Bsp. AVR) sieht die Sache anders aus

```
...
    print(a);
    a=0;
...
```

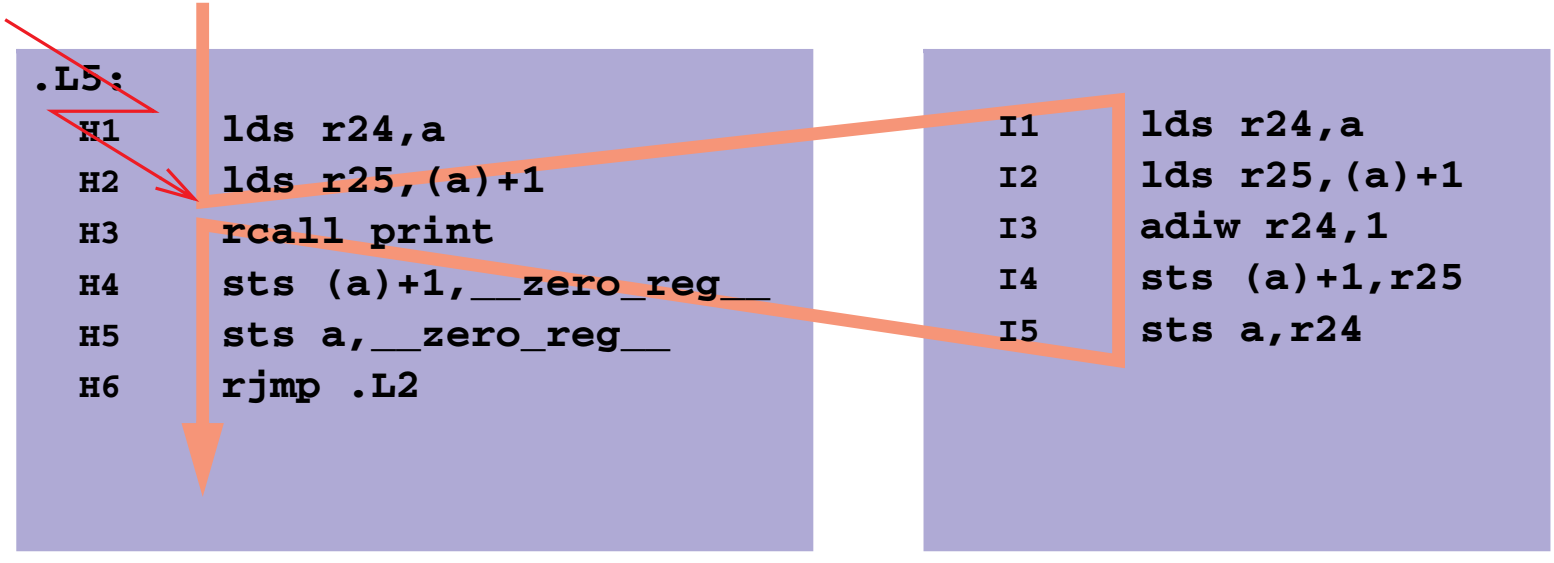
```
void count(void) {
        a++;
}
```

```
...
.L5:
        lds r24,a
        lds r25,(a)+1
        rcall print
        sts (a)+1,__zero_reg__
        sts a,__zero_reg__
        rjmp .L2
...
```

```
...
        lds r24,a
        lds r25,(a)+1
        adiw r24,1
        sts (a)+1,r25
        sts a,r24
...
```

G.2 Nebenläufigkeit durch Interrupts (3)

- Annahme1: Interrupt trifft folgendermaßen ein:



  - ➤ Folge: ein Fahrzeug wird nicht gezählt

- Details des Szenarios zeigen mehrere Problemstellen:
  - ➤ int-Wert wird in zwei Schritten in zwei Register geladen (long: 4 Register)
  - ➤ Operationen erfolgen in Registern, danach wird in Speicher zurückgeschrieben

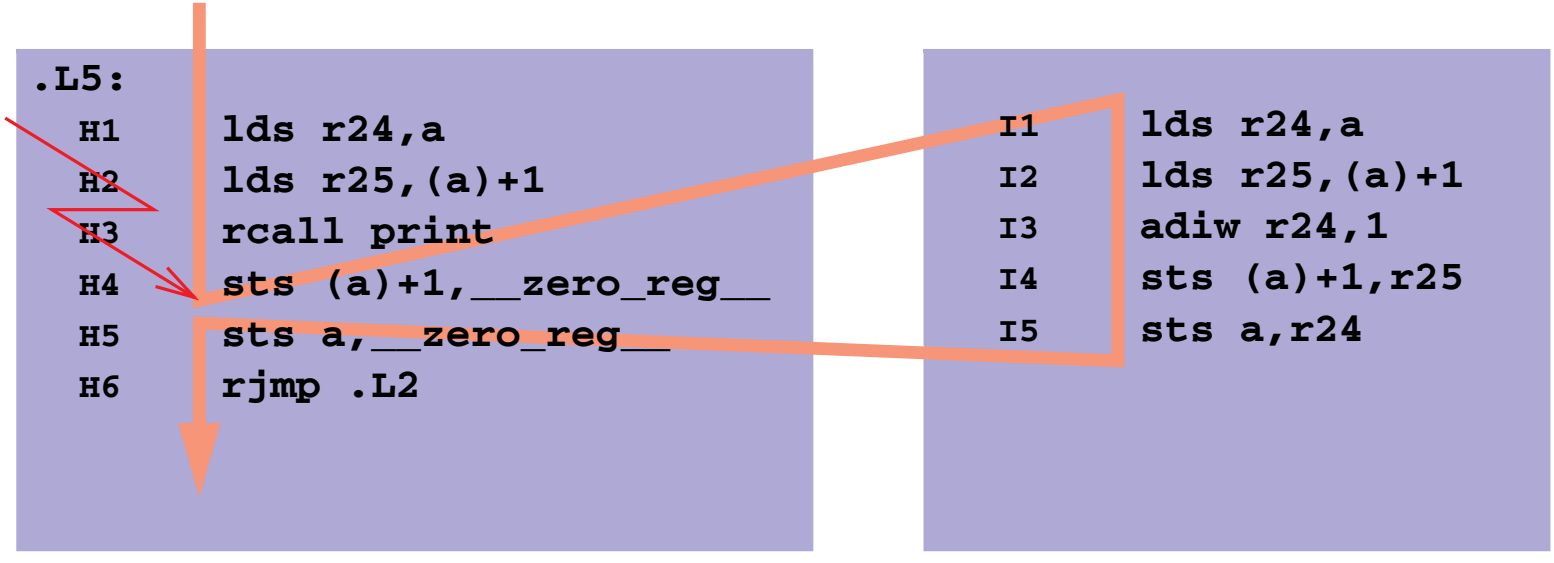
G.2 Nebenläufigkeit durch Interrupts (3)

- Skizze zu Annahme 1, a habe initial den Wert 5

Code-zeile	Variable **a**		Prozessor-Register		gesicherte Registerinhalte		Ausgabe von print
	oberes Byte	unteres Byte	**r25**	**r24**	**r25**	**r24**	
initial	**00**	**05**					
H1		05		**05**			
H2	00		**00**				
INT			00	05	**00**	**05**	
I1				**05**			
I2			**00**	05			
I3			**00**	**06**			
I4	**00**		00				
I5		**06**		06			
ret			**00**	**05**	00	05	
H3			00	05			**5**
H4	**00**						
H5		**00**					

G.2 Nebenläufigkeit durch Interrupts (4)

- Annahme 2: Interrupt trifft folgendermaßen ein:



➤ Folge: möglicherweise werden 255 Fahrzeuge zuviel gezählt

- Variable a ist auf 2 Register verteilt →a = 0 nicht atomar
  zuerst wird obere Hälfte auf 0 gesetzt
- falls a++ im Interrupt-Handler a zufällig von 255 auf 256 zählt
  → Bitüberlauf vom "unteren" in's "obere" Register
- nach Interrupt wird nur noch untere Hälfte auf 0 gesetzt → a = 256

G.2 Nebenläufigkeit durch Interrupts (4)

- Skizze zu Annahme 2, a habe initial den Wert 255

Code-zeile	Variable **a**		Prozessor-Register		gesicherte Registerinhalte		Ausgabe von print
	oberes Byte	unteres Byte	**r25**	**r24**	**r25**	**r24**	
initial	**00**	**ff**					
H1		ff →		**ff**			
H2	00 →		**00**				
H3			00	ff			**255**
H4	**00**						
INT			00	ff →	**00**	**ff**	
I1		ff →		**ff**			
I2	00 →		**00**	ff			
I3			**01**	**00**			
I4	**01** ←		01				
I5		**00** ←		00			
ret			**00**	**ff**	00	ff	
H5	01	**00**					

G.2 Nebenläufigkeit durch Interrupts (5)

- weiteres Problem bei Zugriff auf globale Variablen:
  - AVR stellt 32 Register zur Vefügung
  - Compiler optimiert Code und vermeidet Speicherzugriffe wenn möglich
    - ➤ Variablen werden möglichst in Registern gehalten
  - Registerinhalte werden bei Interrupt gesichert und am Ende restauriert
    - ➤ Änderungen der Interrupt-Funktion an einer Variablen gehen beim Restaurieren der Register wieder verloren

- Lösung für dieses Problem:
  - Compiler muss Variablen vor jedem Zugriff aus dem Speicher laden und anschließend zurückschreiben
    - ➤ Attribut `volatile`

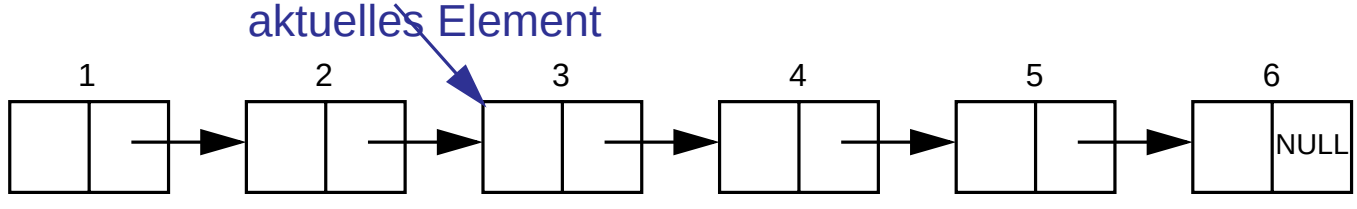
```
volatile int a;
```

  - Nachteil: Code wird umfangreicher und langsamer
    - ➤ nur einsetzen wo unbedingt notwendig!

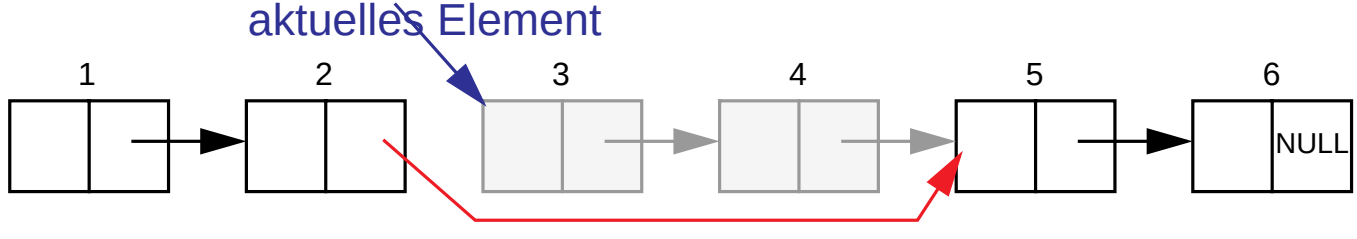
G.3 Nebenläufigkeitsprobleme allgemein

- Zugriff auf gemeinsame Daten ist bei nebenläufigen Ausführungen generell kritisch
  - ➤ selbst bei einfachen Variablen (siehe vorheriges Beispiel)
  - ➤ Problem bei komplexeren Datenstrukturen
    (z. B. Einketten einer Struktur in verkettete Liste)
    noch gravierender: Datenstruktur kann völlig zerstört werden

- Beispiel: Programm läuft durch eine verkettete Liste



  - ◆ Interrupt-Handler oder parallel laufendes Programm entfernt Elemente 3 und 4 und gibt den Speicher dieser Elemente frei



G.4 Umgang mit Nebenläufigkeitsproblemen

- Gemeinsame Daten möglichst vermeiden
  - ➤ Interrupt-Funktionen sollten weitgehend auf eigenen Daten arbeiten
  - ➤ Parallele Abläufe sollten ebenfalls möglichst eigene Datenbereiche haben
- Kommunikation zwischen Anwendungsabläufen erfordert aber oft gemeinsame Daten
  - ➤ solche Daten sollten deutlich hervorgehoben werden
    z. B. durch entsprechenden Namen

```
volatile int INT_zaehler;
```

    - betrifft nur globale Variablen
    - lokale Variablen sind unkritisch
      (nur in der jeweiligen Funktion sichtbar)
  - ➤ Zugriff auf solche Daten sollte in der Anwendung möglichst begrenzt sein
    (z. B. nur in bestimmten Funktionen,
    gemeinsames Modul mit Interrupt-Handlern, vgl. Kap. D.9-3)

G.4 Umgang mit Nebenläufigkeitsproblemen (2)

- Zugriffskonflikte mit Interrupt-Handlern verhindern
  - das Programm muss vor kritischen Zugriffen auf gemeinsame Daten Interrupts sperren
    - ➤ Beispiel AVR:
      Funktionen `cli()` (blockiert alle Interrupts)
      und `sei()` (erlaubt Interrupts)
  - Problem: Interrupt-Verluste bei Interrupt-Sperren
    - ➤ trifft ein Interrupt während der Sperre ein, wird im zugehörigen Register das entsprechende Bit gesetzt
    - ➤ treffen weitere Interrupts ein, geht diese Information verloren

➥ Zeitraum von Interruptsperren muss möglichst kurz bleiben!

- ➤ es kann sinnvoll sein, nur Interrupts des Geräts zu sperren, dessen Handler auch auf die kritischen Daten zugreift
  (hängt von Details der Hardware ab!)

G.4 Umgang mit Nebenläufigkeitsproblemen (3)

- Warten auf einen Interrupt
  - Häufiges Szenario: im Programm soll auf ein bestimmtes Ereignis gewartet werden, das durch einen Interrupt signalisiert wird
    - ➤ Warten erfolgt meist passiv (Sleep-Modus des Prozessors)
  - Problem: Abfrage ob Ereignis bereits eingetreten ist, ist ein kritischer Zugriff auf gemeinsame Daten mit der Interrupt-Behandlung

```
volatile static uint8_t event = 0;
ISR(INT2_vect) { event = 1; }

void main(void) {
        sleep_enable();

        while(1) {
                while(event == 0) { /* Warte auf Ereignis */
                        sleep_cpu();
                }
                /* bearbeite Ereignis */
                ...
```

  - Synchronisation erforderlich?

G.4 Umgang mit Nebenläufigkeitsproblemen (4)

- ... Warten auf einen Interrupt
  - Was passiert, wenn der Interrupt an dieser Stelle eintrifft?

```
volatile static uint8_t event = 0;
ISR(INT2_vect) { event = 1; }

void main(void) {
        sleep_enable();

        while(1) {

                while(event == 0) { /* Warte auf Ereignis */
                                                  Interrupt!
                        sleep_cpu();

                }

                /* bearbeite Ereignis */
                ...
```

  - ➥ Lost-wakeup-Problem

G.4 Umgang mit Nebenläufigkeitsproblemen (5)

- ... Warten auf einen Interrupt
  - kritischer Abschnitt

```
volatile static uint8_t event = 0;
ISR(INT2_vect) { event = 1; }

void main(void) {
        sleep_enable();

        while(1) {
                kritischer Abschnitt
                while(event == 0) { /* Warte auf Ereignis */
                        sleep_cpu();
                }

                /* bearbeite Ereignis */
                ...
```

Interrupt!

  - können hier Interruptsperren helfen?

G.4 Umgang mit Nebenläufigkeitsproblemen (6)

- ... Warten auf einen Interrupt
  - Problem: Interruptsperre muss vor dem `sleep_cpu()` aufgehoben werden

```
volatile static uint8_t event = 0;
ISR(INT2_vect) { event = 1; }

void main(void) {
        sleep_enable();

        while(1) {
                cli();
                while(event == 0) {
                        sei();
                        sleep_cpu();

                }

                /* bearbeite Ereignis */
                ...
```

kritischer Abschnitt

Interrupt!

  - ➤ aber Interrupt darf nicht zwischen `sei()` und `sleep_cpu()` kommen
  - ➤ Lösung: `sei()` und die Folgeanweisung werden atomar ausgeführt

G.4 Umgang mit Nebenläufigkeitsproblemen (7)

- Einseitige Synchronisation
  - Besonderheit bei Nebenläufigkeit durch Interrupts:
    - ➤ der Interrupt kann den normalen Programmablauf unterbrechen
    - ➤ aber nicht umgekehrt
    - ➥ die Interruptbehandlung wird nie unterbrochen (höchstens durch Interrupts mit höherer Prioriät)
- Mehrseitige Synchronisation
  - Standardsituation bei parallelen Abläufen (z. B. auf Mehrkern-Prozessor)
    - ➤ Interruptsperren helfen hier nicht
  - Lösungen
    - ➤ spezielle atomare Maschinenbefehle (z. B. test-and-set oder compare-and-swap bei Intel-Architekturen)
    - ➤ Software-Synchronisation (lock-Variablen, Semaphore, etc.)
    - ➤ Kommunikation mittels Nachrichten statt gemeinsamer Daten

H Programme, Prozesse und Speicher

- **Programm:** Folge von Anweisungen
  (hinterlegt beispielsweise als ausführbare Datei auf dem Hintergrundspeicher)

- **Prozess:** Betriebssystemkonzept
  - ➤ Programm, das sich in Ausführung befindet, und seine Daten
    (Beachte: ein Programm kann sich mehrfach in Ausführung befinden)
  - ➤ eine konkrete Ausführungsumgebung für ein Programm
    Speicher, Rechte, Verwaltungsinformation (verbrauchte Rechenzeit,...),...

- jeder Prozess bekommt einen eigenen virtuellen Adressraum zur Verfügung gestellt
  - ➤ eigener (virtueller) Speicherbereich von 0 bis 2 GB (oder mehr bis 4 GB)
  - ➤ Datenbereichevon verschiedenen Prozessen und Betriebssystem sind gegeneinander geschützt
  - ➤ Datentransfer zwischen Prozessen nur durch Vermittlung des Betriebssystems möglich

H.1 Speicherorganisation eines Prozesses

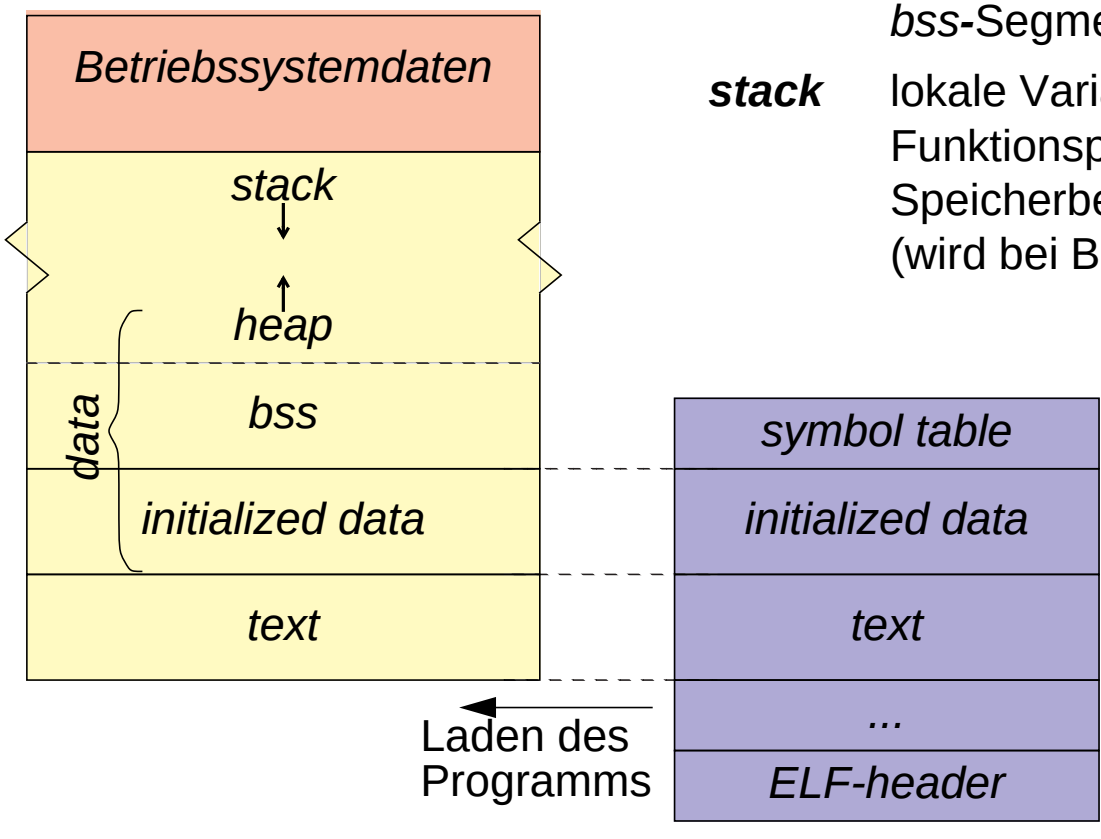
text Programmkode

data globale und static Variablen

bss nicht initialisierte globale und *static* Variablen (wird vor der Vergabe an den Prozess mit 0 vorbelegt)

heap dynamische Erweiterungen des *bss*-Segments (***sbrk(2)***, ***malloc(3)***)

stack lokale Variablen, Funktionsparameter, Speicherbereiche für Registerinhalte, (wird bei Bedarf dynamisch erweitert)

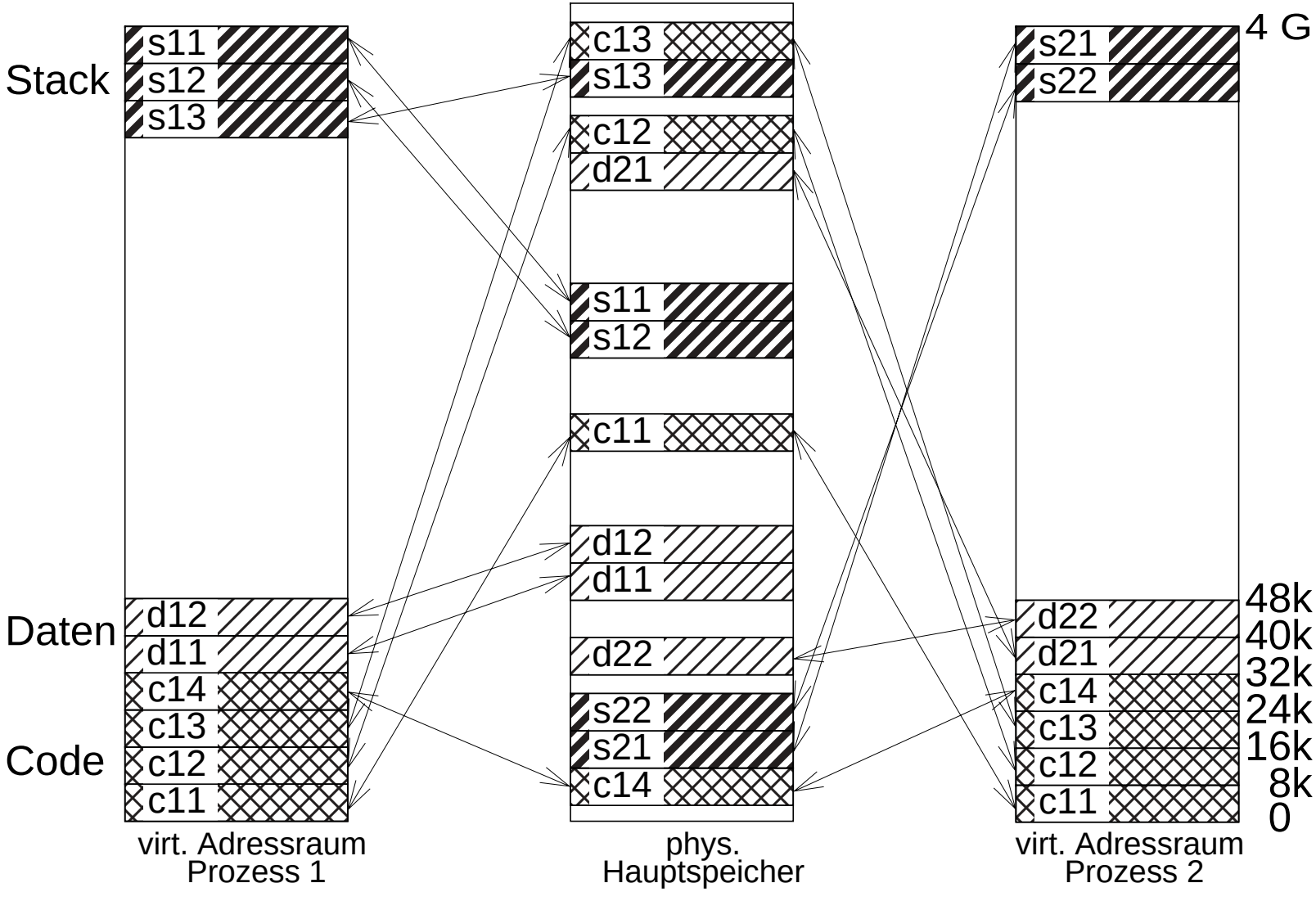


Aufbau eines Programms (ELF-Format)

H.1 Speicherorganisation eines Prozesses (2)

- Abbildung des virtuellen Adressraums in den realen Hauptspeicher durch Seitenadressierung (**Paging)**
  - ➤ Adreßraum ist in kleine (4 oder 8 kB) Stücke unterteilt (**Seiten**)
  - ➤ jede Seite wird über eine Tabelle in ein entsprechendes Stück des Hauptspeichers (**Kachel**) abgebildet
  - ➤ bei jedem Speicherzugriff wird die virtuelle Adresse in die entsprechende physikalische Adresse umgerechnet
    (spezielle Hardware: Memory Management Unit - MMU)
  - ➤ zu jeder Seite sind Zugriffsrechte vermerkt (nur lesen, lesen+schreiben, Maschinenbefehle ausführen)
  - ➤ eine Seite kann bei Speichermangel von Betriebssystem auf Festplatte ausgelagert werden und bei Bedarf automatisch wieder eingelagert werden

H.1 Speicherorganisation eines Prozesses(3)



H.2 Stackaufbau eines Prozesses

1 Prinzip

- für jede Funktion wird ein ***Stack-Frame*** angelegt, in dem
  - lokale Variablen der Funktion
  - Aufrufparameter an weitere Funktionen
  - Registerbelegung der Funktion während des Aufrufs weiterer Funktionen

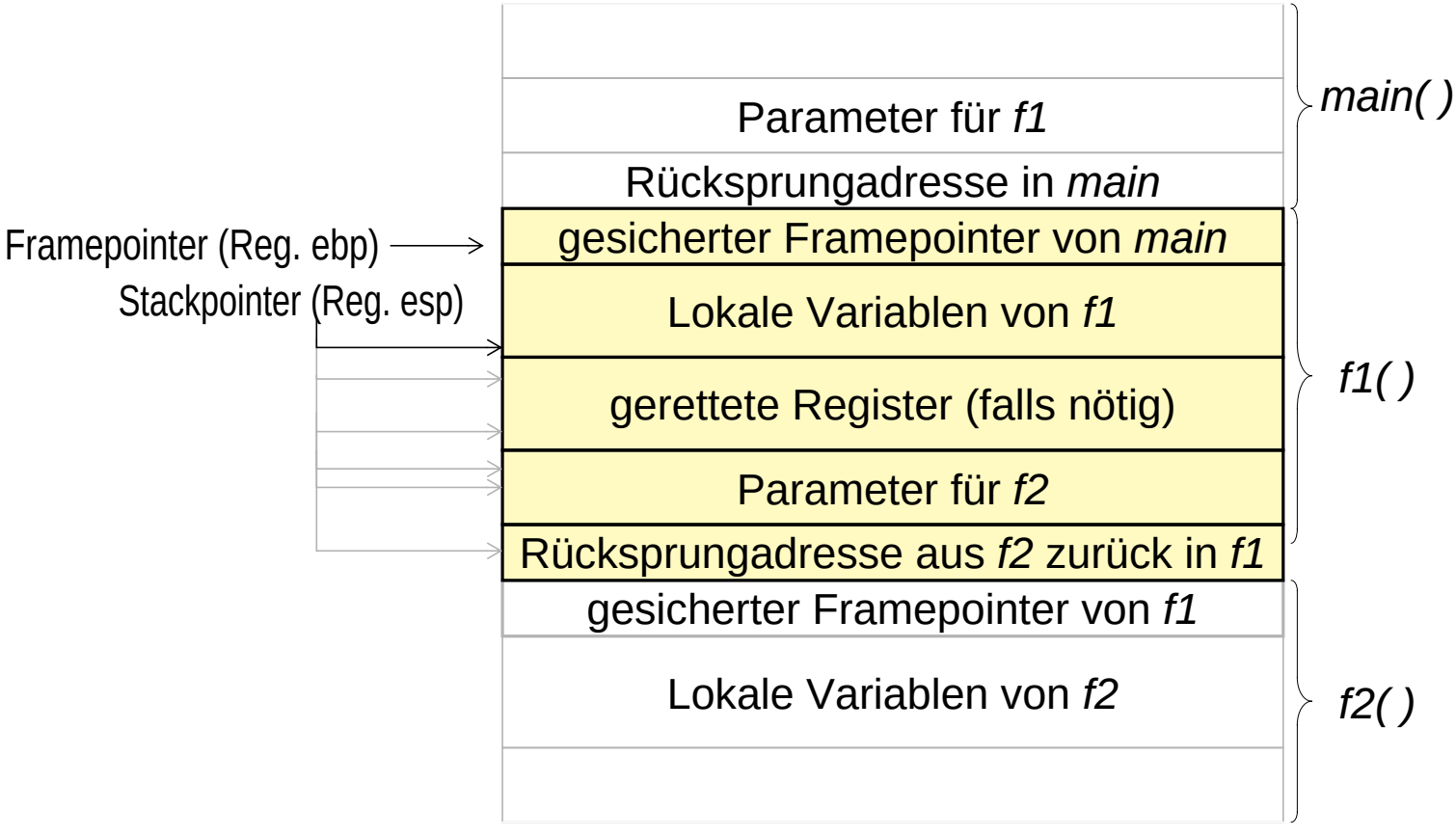
  gespeichert werden

- Stackorganisation ist abhängig von
  - Prozessor
  - Compiler und
  - Betriebssystem

- Beispiele aus einem UNIX auf Intel-Prozessor (typisch für CISC)
  - RISC-Prozessoren mit Registerfiles gehen teilweise anders vor!

2 Beispiel

- Aufbau eines **Stack-Frames** (Funktionen *main( ), f1( ), f2( )*)

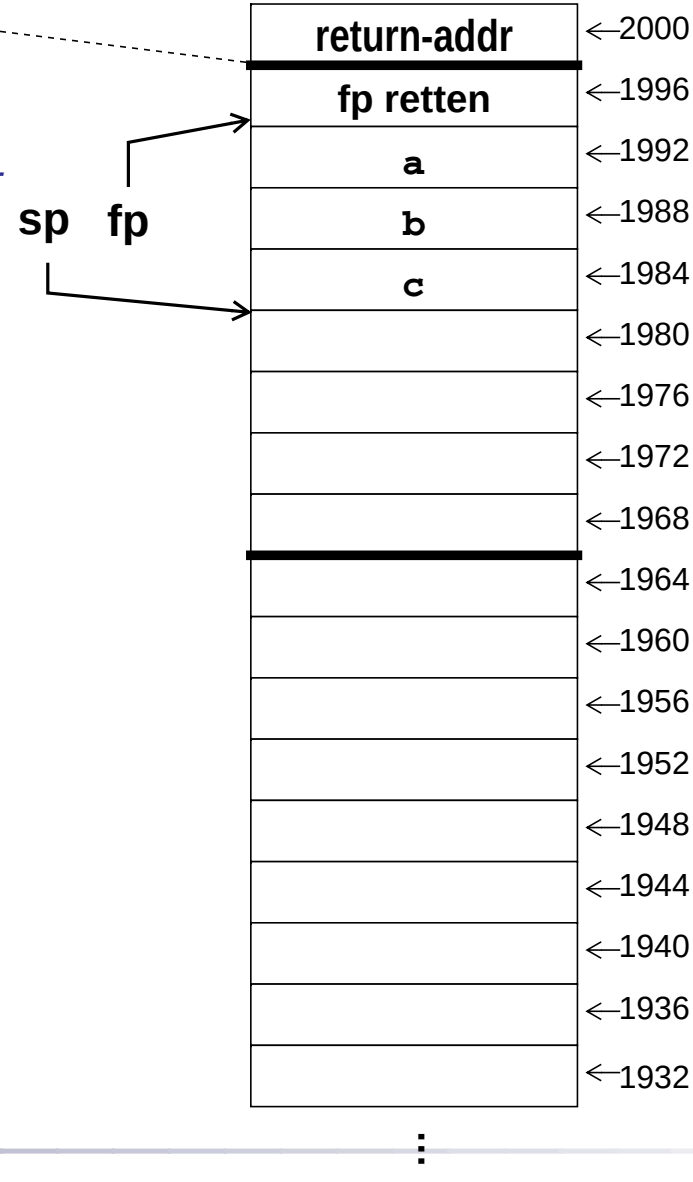


➤ Achtung: architekturabängige Optimierungen können zu Padding (Füllbytes) führen!

2 Stack mehrerer Funktionsaufrufe

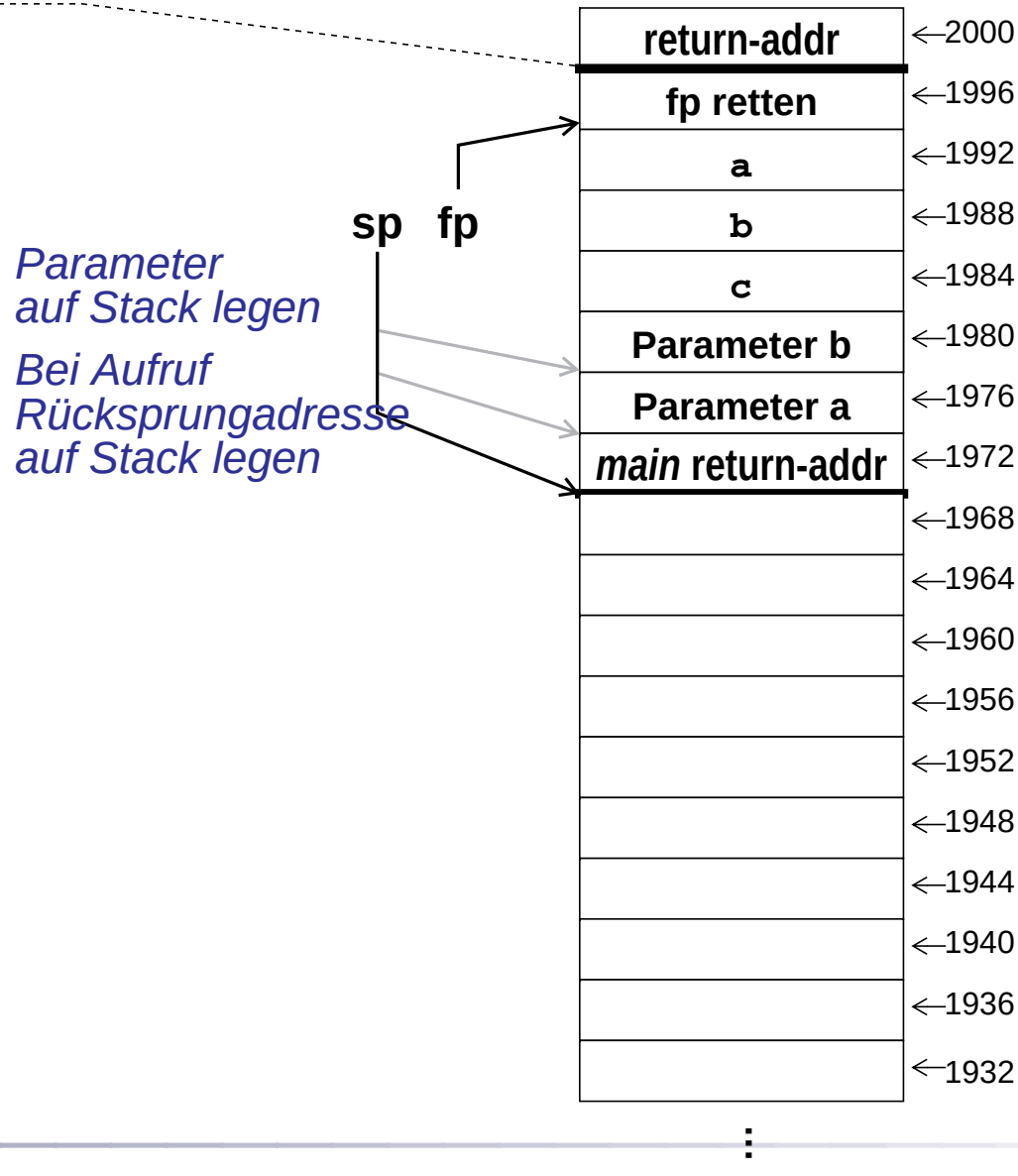
```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

Stack-Frame für main erstellen
&a = fp-4
&b = fp-8
&c = fp-12



2 Stack mehrerer Funktionsaufrufe

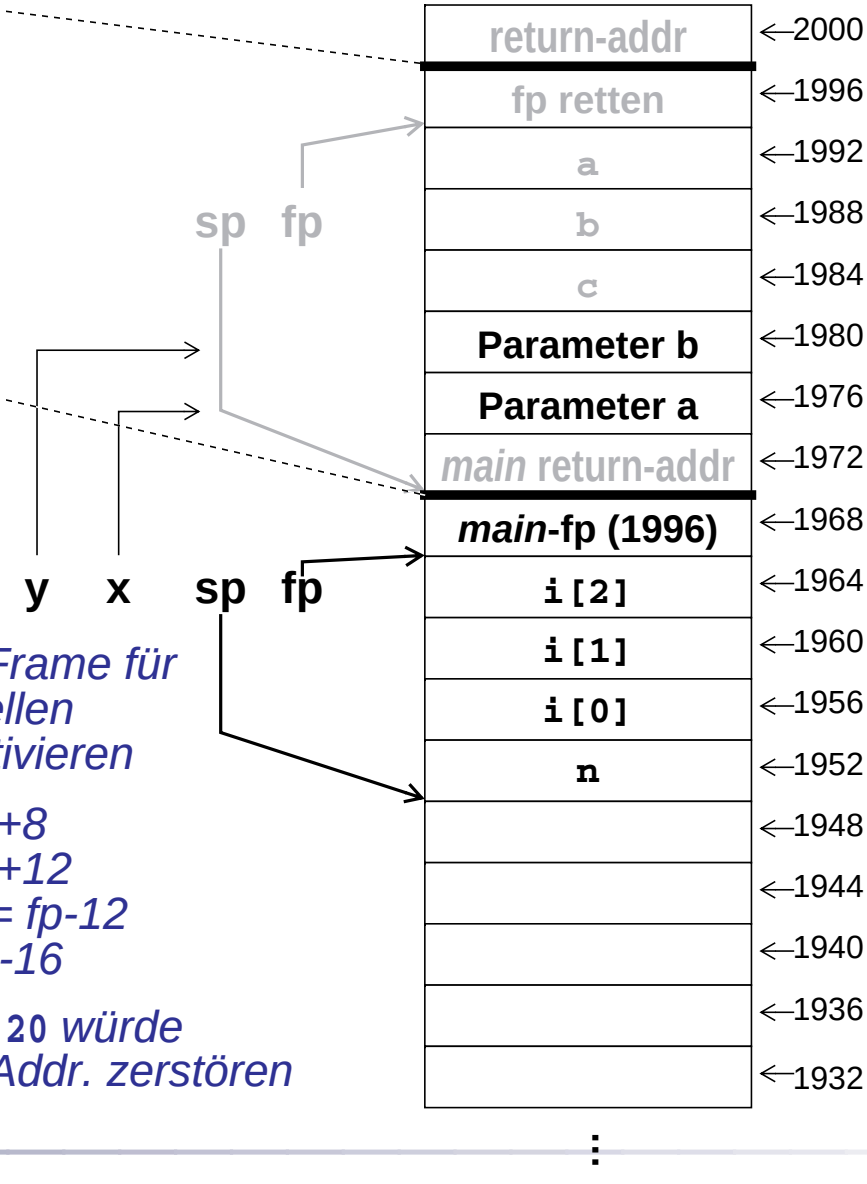
```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```



2 Stack mehrerer Funktionsaufrufe

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

```
int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}
```



Stack-Frame für f1 erstellen und aktivieren

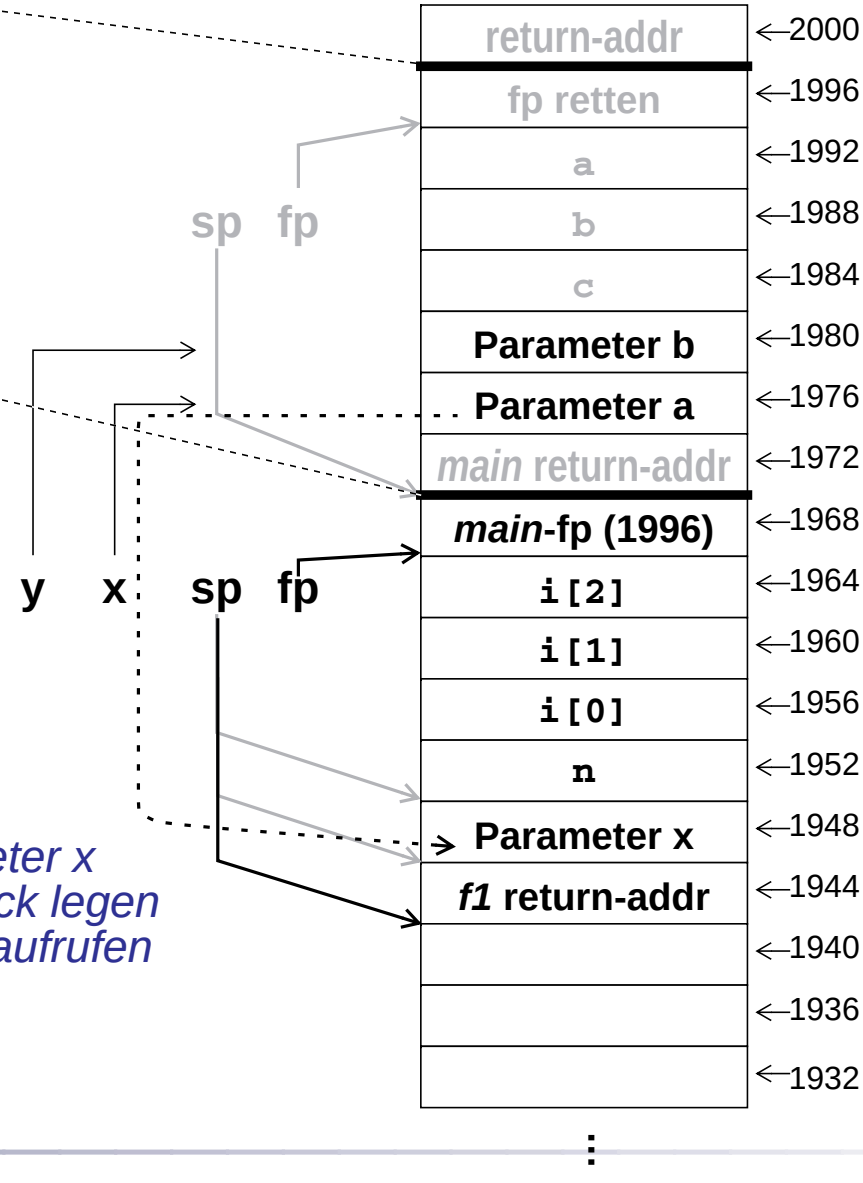
&x = fp+8
&y = fp+12
&(i[0]) = fp-12
&n = fp-16

`i[4] = 20` *würde return-Addr. zerstören*

2 Stack mehrerer Funktionsaufrufe

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

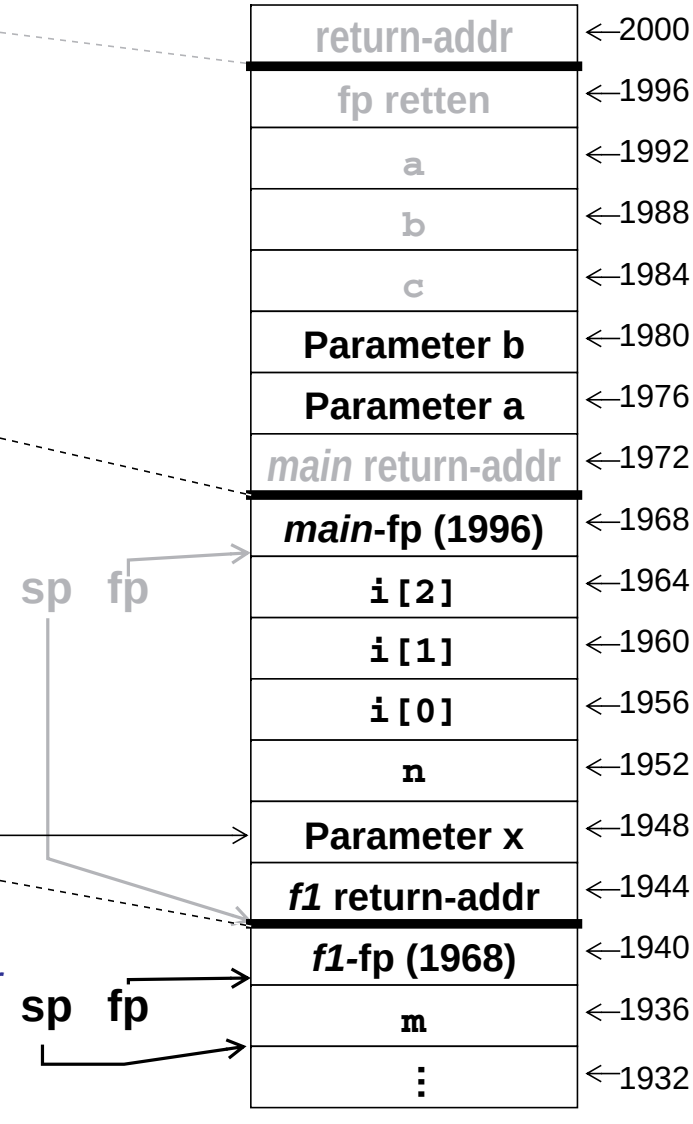
```
int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}
```



Parameter x auf Stack legen und f2 aufrufen

2 Stack mehrerer Funktionsaufrufe

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}


int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}

int f2(int z) {
        int m;

        m = 100;

        return(z+1);
}
```



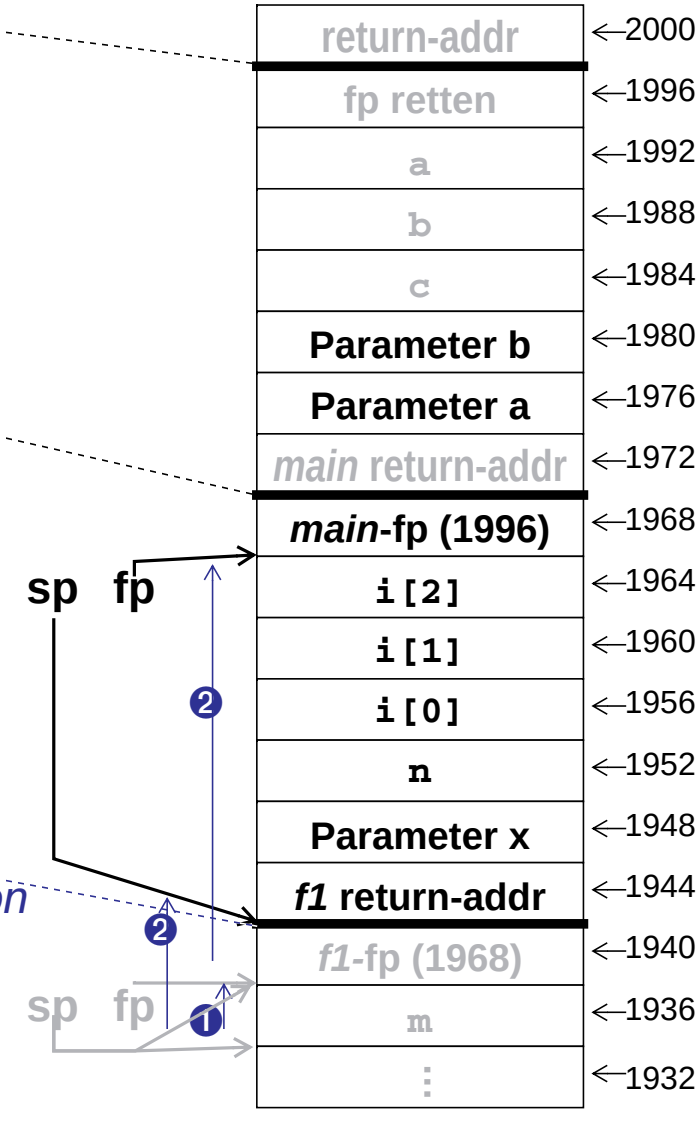
Stack-Frame für f2 erstellen und aktivieren

2 Stack mehrerer Funktionsaufrufe

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

```
int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}
```

```
int f2(int z) {
        int m;

        m = 100;

        return(z+1);
}
```



Stack-Frame von f2 abräumen

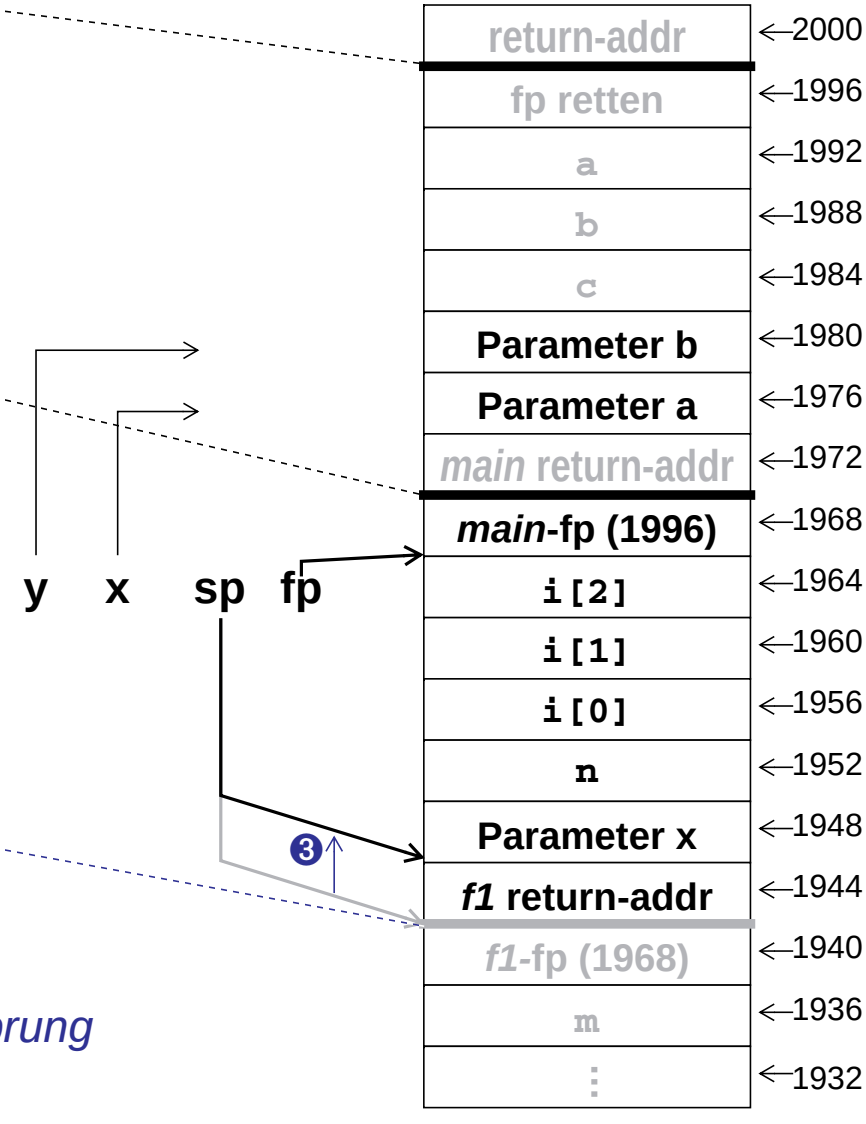
❶ *sp = fp*
❷ *fp = pop(sp)*

2 Stack mehrerer Funktionsaufrufe

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

```
int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}
```

```
int f2(int z) {
        int m;

        m = 100;

        return(z+1);
}
```

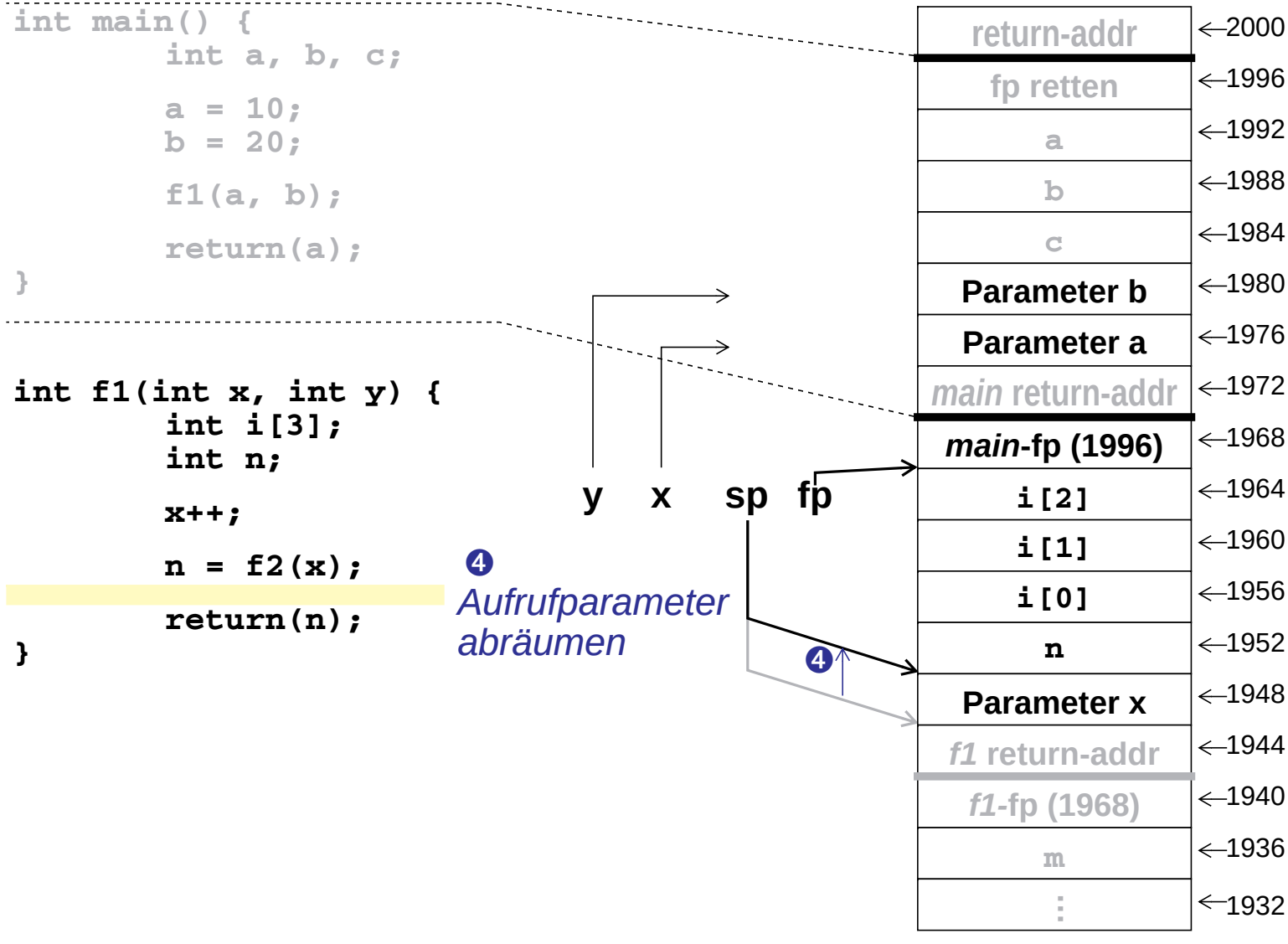


2 Stack mehrerer Funktionsaufrufe

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

```
int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}
```

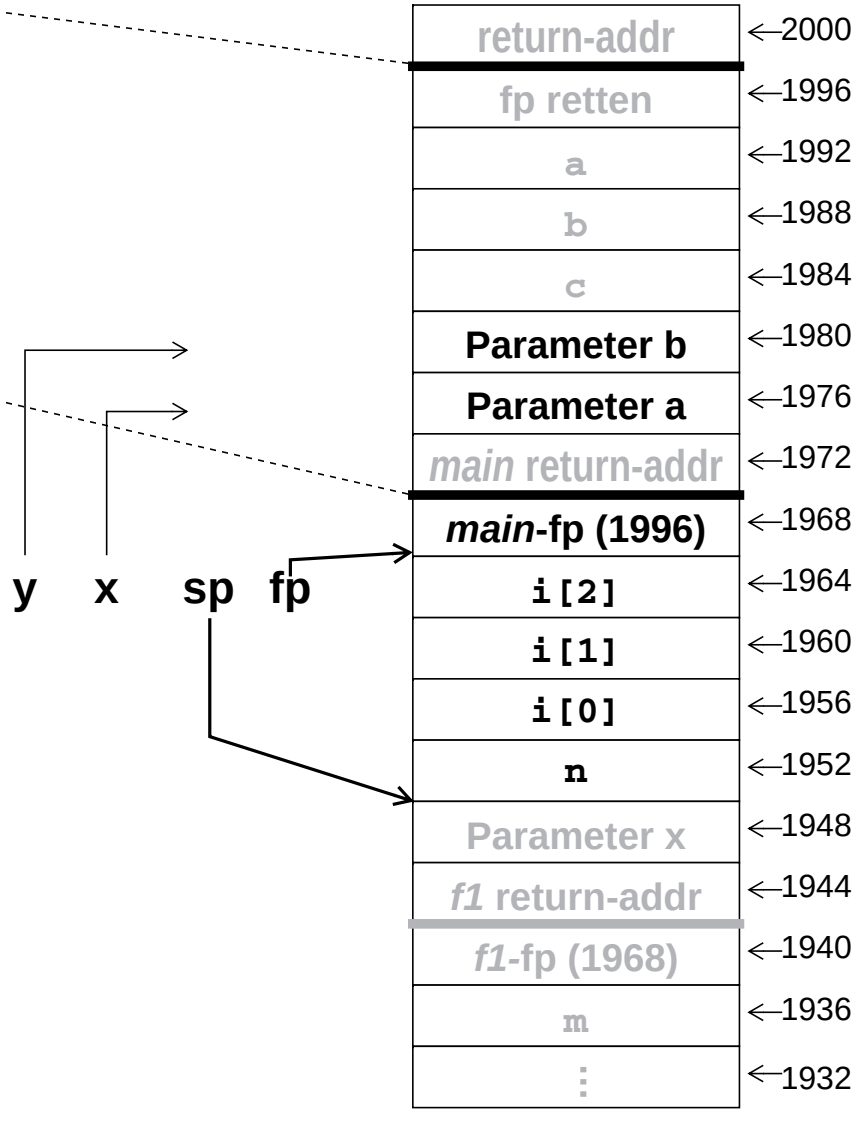
❹ *Aufrufparameter abräumen*



2 Stack mehrerer Funktionsaufrufe

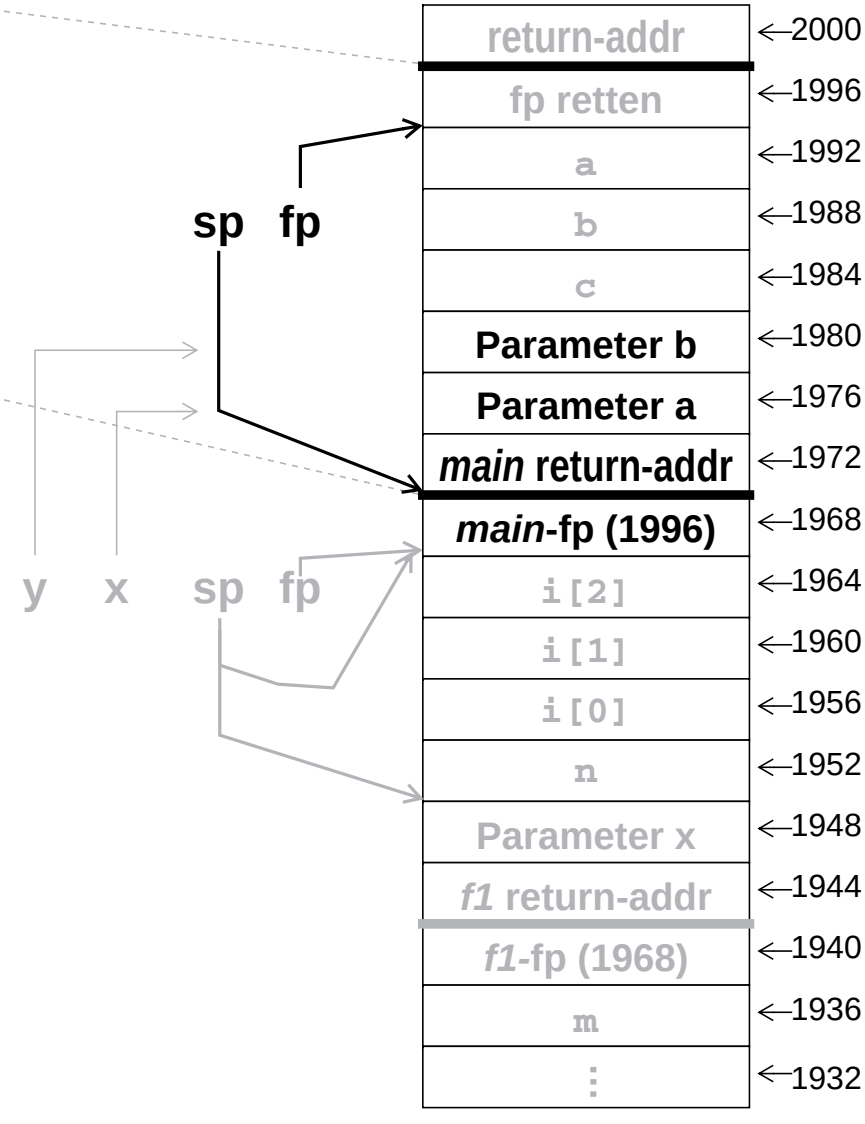
```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

```
int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}
```



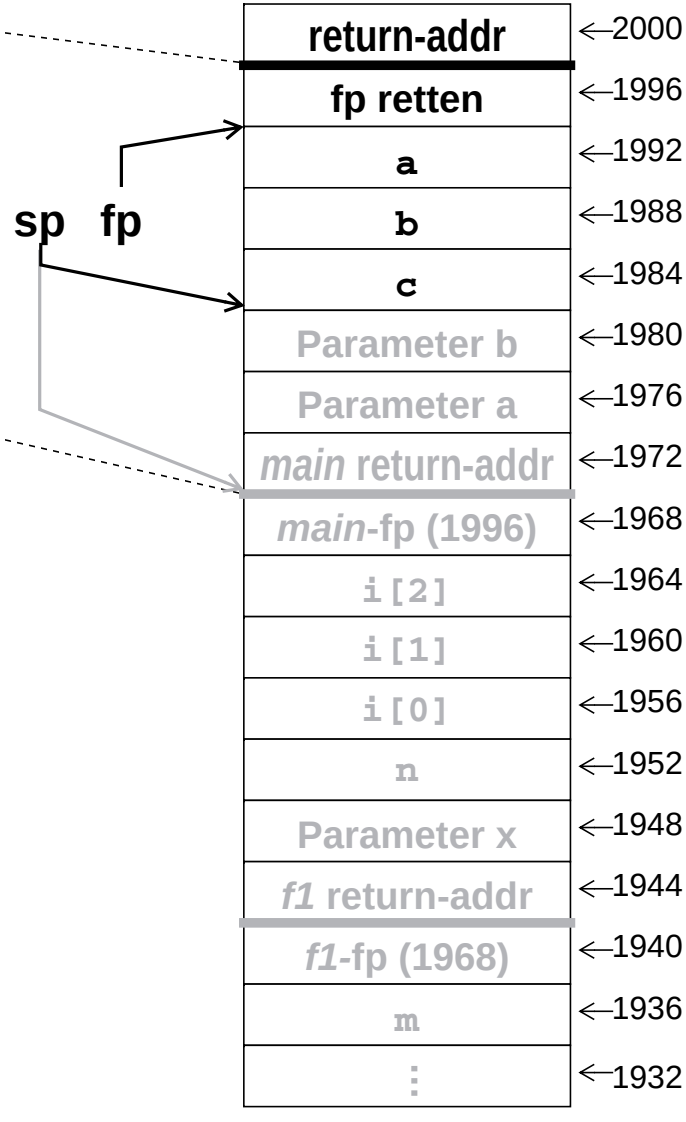
2 Stack mehrerer Funktionsaufrufe

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}


int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}
```



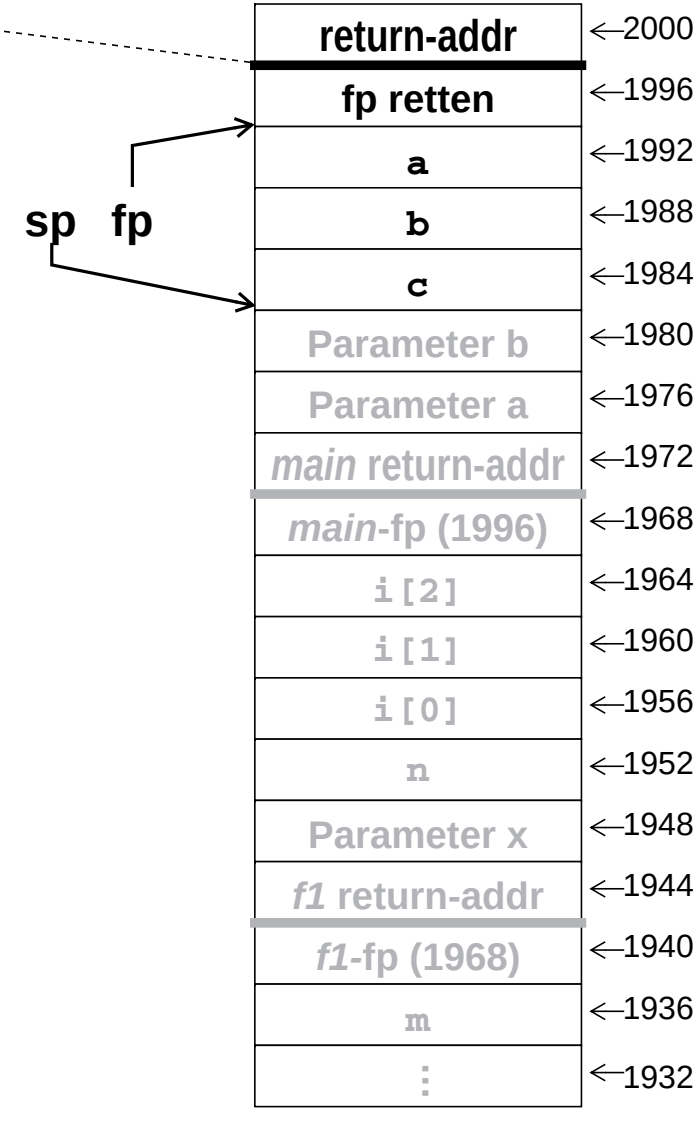
2 Stack mehrerer Funktionsaufrufe

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}


int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}
```



2 Stack mehrerer Funktionsaufrufe

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

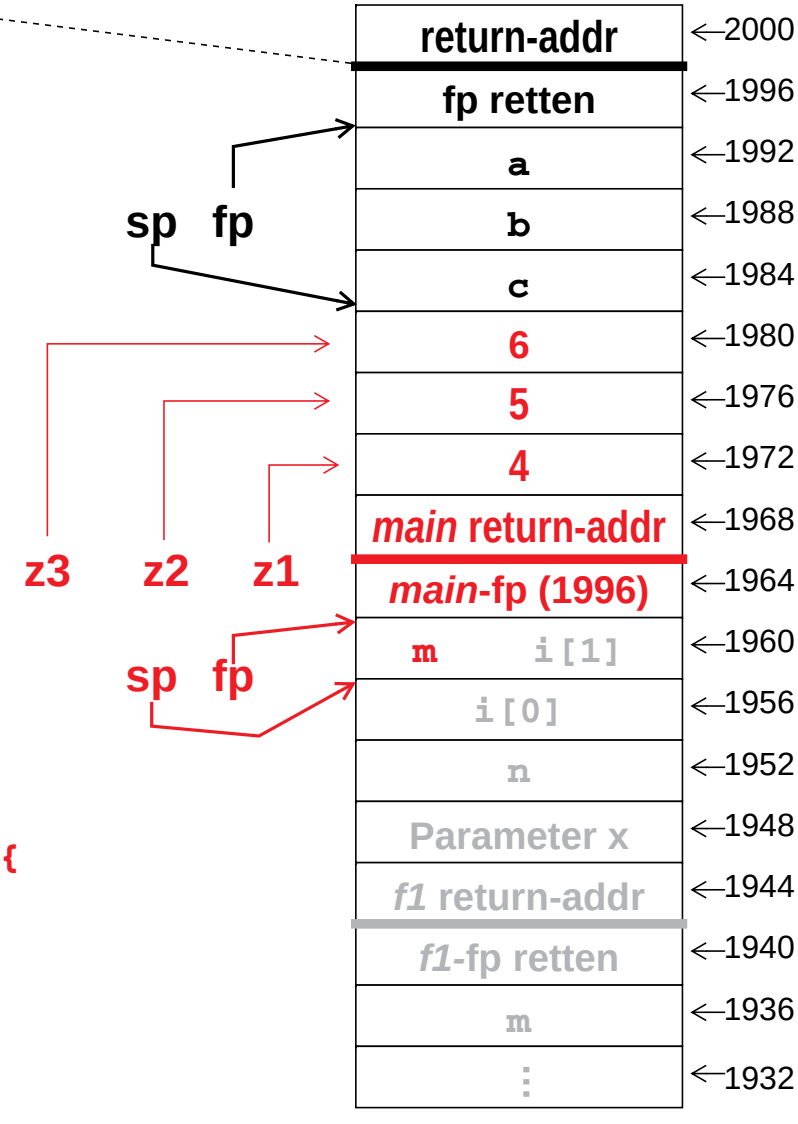


2 Stack mehrerer Funktionsaufrufe

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        f3(4,5,6);
}
```

was wäre, wenn man nach f1 jetzt eine Funktion f3 aufrufen würde?

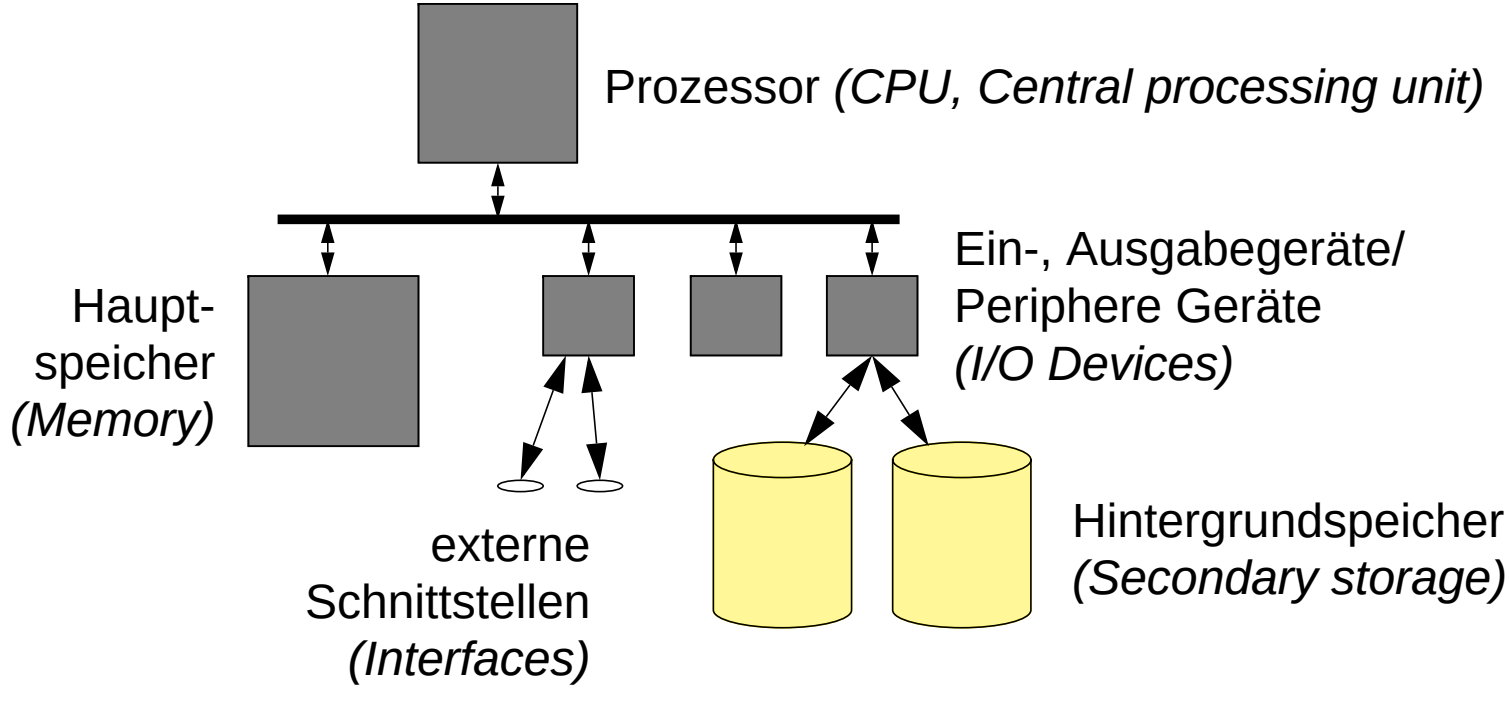
```
int f3(int z1, int z2, int z3) {
        int m;

        return(m);
}
```



I Dateisysteme

I.1 Allgemeine Konzepte

- Einordnung

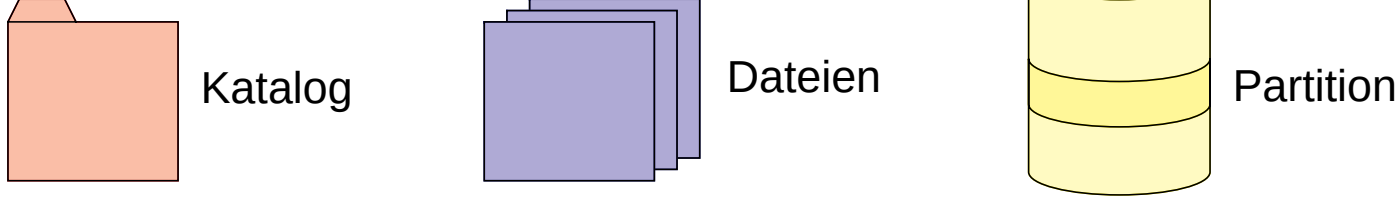


I.2 Allgemeine Konzepte (2)

- Dateisysteme speichern Daten und Programme persistent in Dateien
  - Betriebssystemabstraktion zur Nutzung von Hintergrundspeichern (z.B. Platten, CD-ROM, Bandlaufwerke)
    - Benutzer muss sich nicht um die Ansteuerungen verschiedener Speichermedien kümmern
    - einheitliche Sicht auf den Hintergrundspeicher
- Dateisysteme bestehen aus
  - Dateien *(Files)*
  - Katalogen *(Directories)*
  - Partitionen *(Partitions)*

I.2 Allgemeine Konzepte (3)

- Datei
  - speichert Daten oder Programme
- Katalog / Verzeichnis (*Directory*)
  - erlaubt Benennung der Dateien
  - enthält Zusatzinformationen zu Dateien
- Partitionen
  - eine Menge von Katalogen und deren Dateien
  - sie dienen zum physischen oder logischen Trennen von Dateimengen.



I.3 Ein-/Ausgabe in C-Programmen

1 Überblick

- E-/A-Funktionalität nicht Teil der Programmiersprache
- Realisisierung 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

2 Standard Ein-/Ausgabe

- Jedes C-Programm erhält beim Start automatisch 3 E-/A-Kanäle:
  - **`stdin`** Standardeingabe
    - normalerweise mit der Tastatur verbunden, Umlenkung durch **`<`**
    - Dateiende (**`EOF`**) wird durch Eingabe von **`CTRL-D`** am Zeilenanfang signalisiert
  - **`stdout`** Standardausgabe
    - normalerweise mit dem Bildschirm (bzw. dem Fenster, in dem das Programm gestartet wurde) verbunden, Umlenkung durch **`>`**
  - **`stderr`** Ausgabekanal für Fehlermeldungen
    - normalerweise ebenfalls mit Bildschirm verbunden
- automatische Pufferung
  - Eingabe von der Tastatur wird normalerweise vom Betriebssystem zeilenweise zwischengespeichert und erst bei einem **`NEWLINE`**-Zeichen (**`'\n'`**) an das Programm übergeben!

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

3 Öffnen und Schließen von Dateien (2)

- Beispiel:

```
#include <stdio.h>

int main() {
        FILE *eingabe;
        char dateiname[256];

        printf("Dateiname: ");
        scanf("%s\n", dateiname);

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

        ... /* Programm kann jetzt von eingabe lesen */
        ... /* z. B. mit c = getc(eingabe) */
```

- Schließen eines E/A-Kanals

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

➤ schließt E/A-Kanal **`fp`**

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

4 Zeichenweise Lesen und Schreiben (2)

- Beispiel: copy-Programm

Teil 1: Dateien öffnen

```
#include <stdio.h>

int main() {
        FILE *quelle;
        FILE *ziel;
        char quelldatei[256], zieldatei[256];
        int c;                  /* gerade kopiertes Zeichen */

        printf("Quelldatei und Zieldatei eingeben: ");
        scanf("%s %s\n", quelldatei, zieldatei);

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

        if ((ziel = fopen(zieldatei, "w")) == NULL) {
                perror(zieldatei);/* Fehlermeldung ausgeben */
                exit(1);        /* Programm abbrechen */
        }

        /* ... */
```

4 Zeichenweise Lesen und Schreiben (3)

… Beispiel: copy-Programm
— Fortsetzung

Teil 2: kopieren

```
        /* ... */

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

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

5 Formatierte Ausgabe — Funktionen

- 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)

5 Formatierte Ausgabe — Formatangaben

- 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

5 Formatierte Eingabe — Funktionen

- Bibliotheksfunktionen — Prototypen (Schnittstelle)

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

  - Die Funktionen lesen Zeichen von **`stdin`** (**`scanf`**), **`fp`** (**`fscanf`**) bzw. aus dem **`char`**-Feld **`s`**.
  - **`format`** gibt an, welche Daten hiervon extrahiert und in welchen Datentyp konvertiert werden sollen
  - Die folgenden Parameter sind Zeiger auf Variablen der passenden Datentypen (bzw. **`char`**-Felder bei Format **`%s`**), in die die Resultate eingetragen werden
  - relativ komplexe Funktionalität, hier nur Kurzüberblick
    für Details siehe Manual-Seiten

5 Formatierte Eingabe — Bearbeitung der Eingabe-Daten

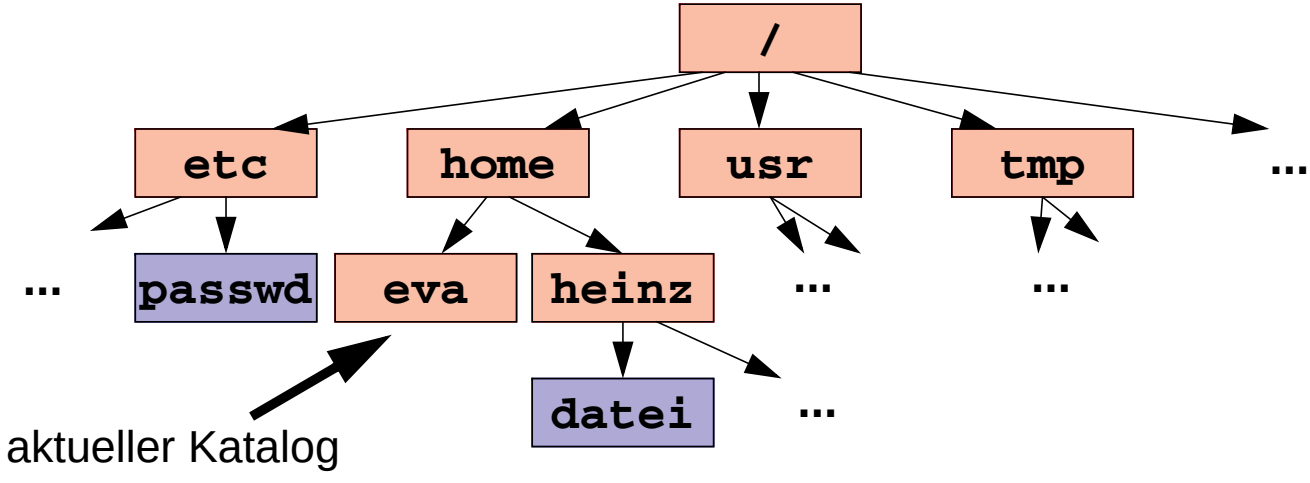
- *White space* (Space, Tabulator oder Newline \n) bildet jeweils die Grenze zwischen Daten, die interpretiert werden
  - ➤ *white space* wird in beliebiger Menge einfach überlesen
  - ➤ Ausnahme: bei Format-Anweisung **`%c`** wird auch *white space* eingelesen
- Alle anderen Daten in der Eingabe müssen 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
- Detail siehe Manual-Seite `(man scanf)`

I.4 Dateisystem am Beispiel Linux/UNIX

- Datei
  - ◆ einfache, unstrukturierte Folge von Bytes
  - ◆ beliebiger Inhalt; für das Betriebssystem ist der Inhalt transparent
  - ◆ dynamisch erweiterbar
- Katalog
  - ◆ baumförmig strukturiert
    - Knoten des Baums sind Kataloge
    - Blätter des Baums sind Verweise auf Dateien
  - ◆ jedem UNIX-Prozess ist zu jeder Zeit ein aktueller Katalog *(Current working directory)* zugeordnet
- Partitionen
  - ➤ jede Partition enthält einen eigenen Dateibaum
  - ➤ Bäume der Partitionen werden durch "mounten" zu einem homogenen Dateibaum zusammengebaut (Grenzen für Anwender nicht sichtbar!)

1 Pfadnamen

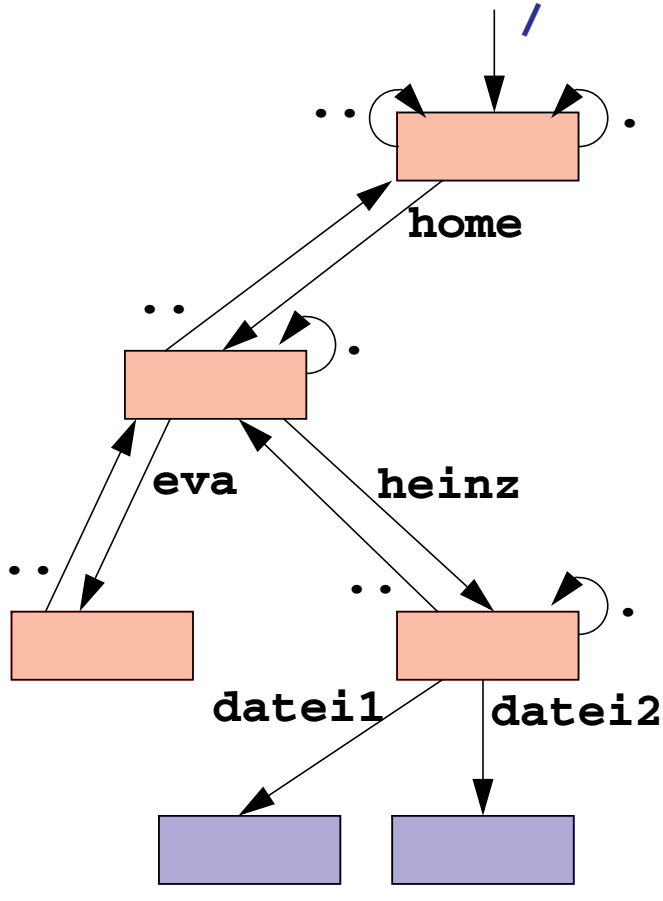
- Baumstruktur



- Pfade
  - z.B. „**`/home/heinz/datei`**“, „**`/tmp`**“, „**`../heinz/datei`**“
  - „**`/`**“ ist Trennsymbol *(Slash)*; beginnender „**`/`**“ bezeichnet Wurzelkatalog; sonst Beginn implizit mit dem aktuellem Katalog

1 Pfadnamen (2)

- Eigentliche Baumstruktur



- benannt sind nicht Dateien und Kataloge, sondern die Verbindungen (*Links*) zwischen ihnen
  - Kataloge und Dateien können auf verschiedenen Pfaden erreichbar sein
    z. B. **`../heinz/datei1`** und **`/home/heinz/datei1`**
  - Jeder Katalog enthält
    - einen Verweis auf sich selbst (**`.`**) und
    - einen Verweis auf den darüberliegenden Katalog im Baum (**`..`**)
    - Verweise auf Dateien

2 Programmierschnittstelle für Kataloge

- Kataloge verwalten
  - Erzeugen

```
int mkdir( const char *path, mode_t mode );
```

  - Löschen

```
int rmdir( const char *path );
```

- Kataloge lesen (Schnittstelle der C-Bibliothek)
  - ➤ Katalog öffnen:
    `DIR *opendir( const char *path );`
  - ➤ Katalogeinträge lesen:
    `struct dirent *readdir( DIR *dirp );`
  - ➤ Katalog schließen:
    `int closedir( DIR *dirp );`

2 Kataloge (2): opendir / closedir

- Funktionsschnittstelle:

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *dirname);

int closedir(DIR *dirp);
```

- Argument von opendir
  - **dirname**: Verzeichnisname
- Rückgabewert: Zeiger auf Datenstruktur vom Typ **DIR** oder **NULL**

2 Kataloge (3): readdir

- Funktionsschnittstelle:

```
#include <sys/types.h>
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
```

- Argumente
  - **dirp**: Zeiger auf **DIR**-Datenstruktur
- Rückgabewert: Zeiger auf Datenstruktur vom Typ **struct dirent** oder **NULL** wenn fertig oder Fehler (**errno** vorher auf 0 setzen!)
- Probleme: Der Speicher für **struct dirent** wird von der Funktion readdir beim nächsten Aufruf wieder verwendet!
  - ➤ wenn Daten aus der Struktur (z. B. der Dateiname) länger benötigt werden, reicht es nicht, sich den zurückgegebenen Zeiger zu merken sondern es müssen die benötigten Daten kopiert werden

2 Kataloge (4): struct dirent

- Definition unter Linux (/usr/include/bits/dirent.h)

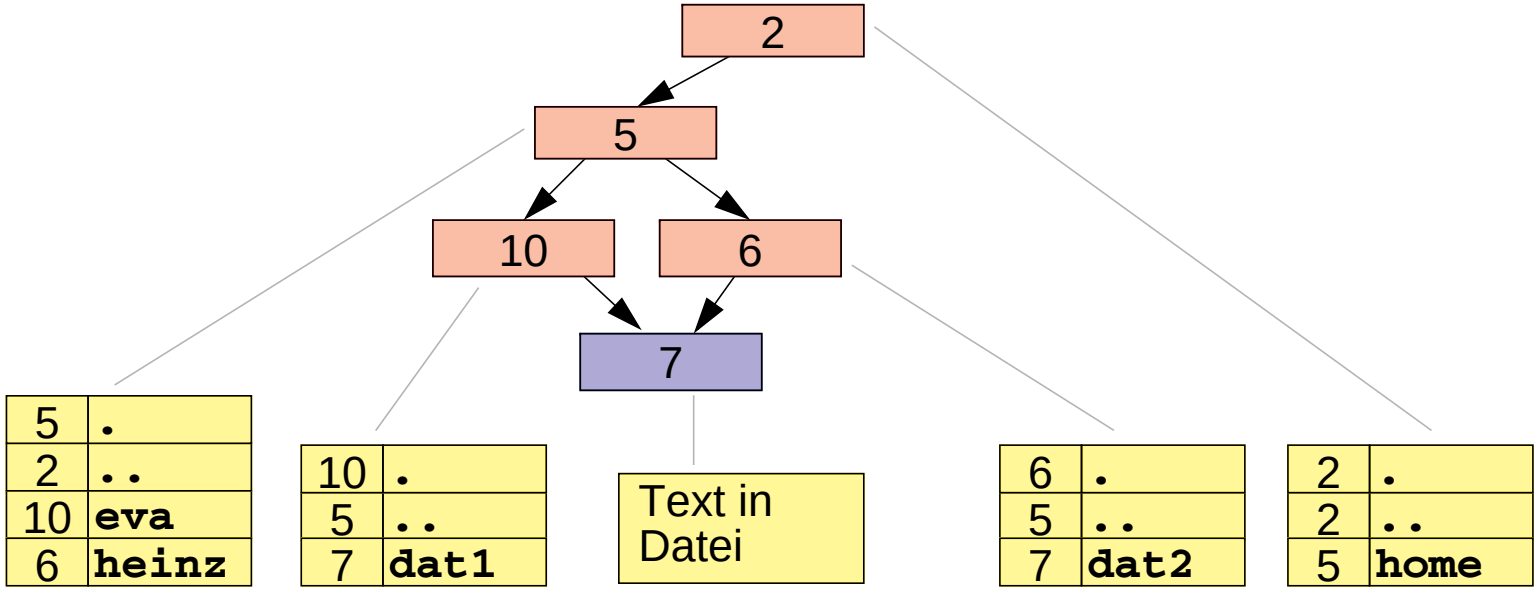
```
struct dirent {
    __ino_t d_ino;
    __off_t d_off;
    unsigned short int d_reclen;
    unsigned char d_type;
    char d_name[256];
};
```

3 Programmierschnittstelle für Dateien

- siehe C-Ein/Ausgabe (Schnittstelle der C-Bibliothek)
- C-Funktionen (fopen, printf, scanf, getchar, fputs, fclose, ...) verbergen die "eigentliche" Systemschnittstelle und bieten mehr "Komfort"
  - ➤ Systemschnittstelle: open, close, read, write

4 Inodes

- Attribute (Zugriffsrechte, Eigentümer, etc.) einer Datei und Ortsinformation über ihren Inhalt werden in **Inodes** gehalten
  - Inodes werden pro Partition numeriert *(Inode number)*
- Kataloge enthalten lediglich Paare von Namen und Inode-Nummern
  - Kataloge bilden einen hierarchischen Namensraum über einem eigentlich flachen Namensraum (durchnumerierte Dateien)



4 Inodes (2)

- Inhalt eines Inode
  - Dateityp: Katalog, normale Datei, Spezialdatei (z.B. Gerät)
  - Eigentümer und Gruppe
  - Zugriffsrechte
  - Zugriffszeiten: letzte Änderung *(mtime)*, letzer Zugriff *(atime)*, letzte Änderung des Inodes *(ctime)*
  - Anzahl der Hard links auf den Inode
  - Dateigröße (in Bytes)
  - Adressen der Datenblöcke des Datei- oder Kataloginhalts

5 Inodes — Programmierschnittstelle: stat / lstat

- liefert Datei-Attribute aus dem Inode
- Funktionsschnittstelle:

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

- Argumente:
  - **path**: Dateiname
  - **buf**: Zeiger auf Puffer, in den Inode-Informationen eingetragen werden
- Rückgabewert: 0 wenn OK, -1 wenn Fehler
- Beispiel:

```
struct stat buf;
stat("/etc/passwd", &buf); /* Fehlerabfrage ... */
printf("Inode-Nummer: %d\n", buf.st_ino);
```