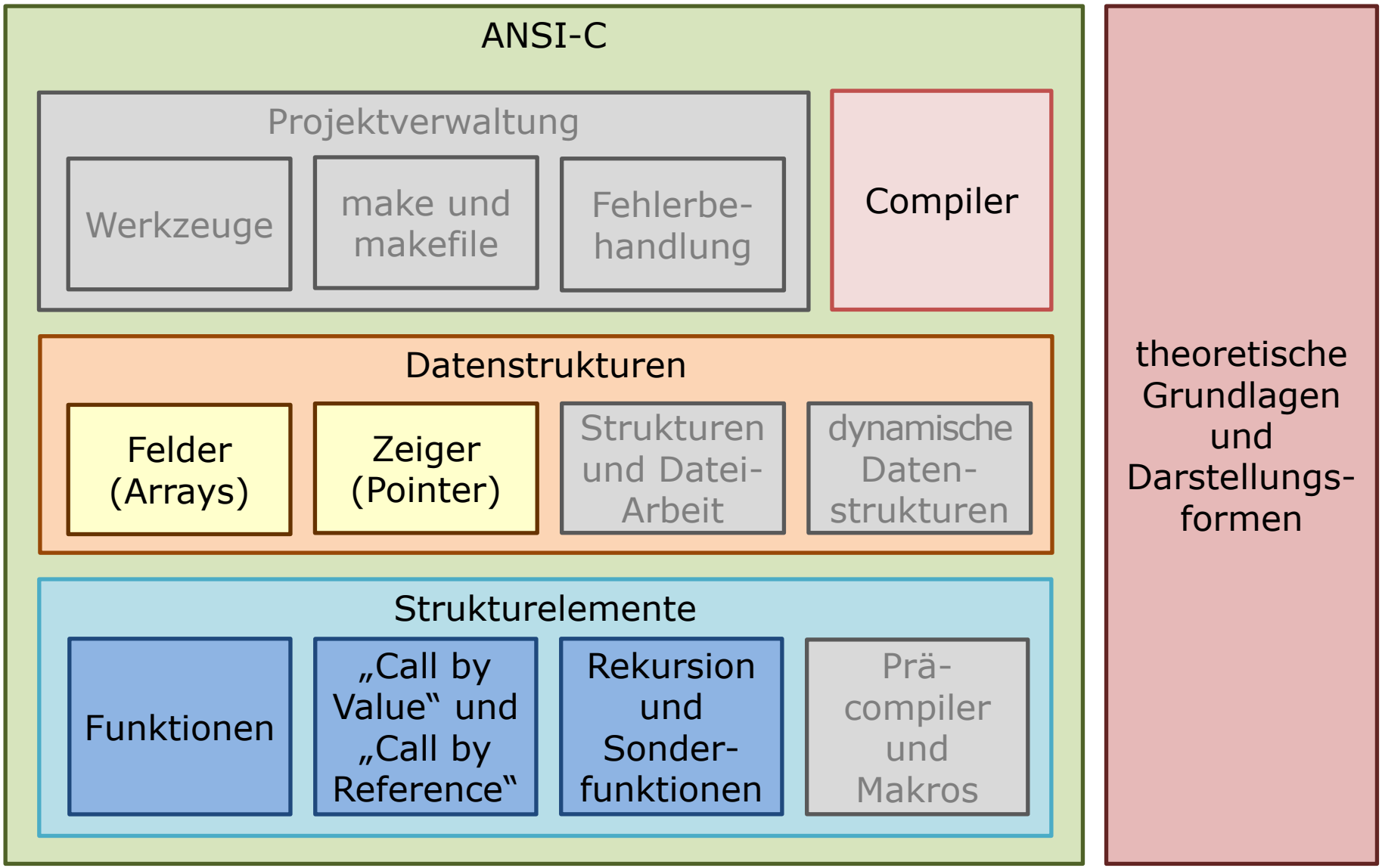
BERUFSAKADEMIE SACHSEN
STAATLICHE STUDIENAKADEMIE
DRESDEN
UNIVERSITY OF COOPERATIVE EDUCATION

Imperative Programmierung

Felder und Zeiger

Prof. Dr.-Ing. Tenshi Hara
tenshi.hara@ba-sachsen.de

Aufbau der Lehrveranstaltung



Feld (Vektor, Array)

- Aneinanderreihung von Elementen gleichen Typs
  (elementare Datentypen, Strukturen, selbst definierte Datentypen, ...)
- Feld-Elemente stehen im Speicher lückenlos hintereinander
  (mit aufsteigenden Adressen)
- Zugriff auf einzelne Elemente eines Feldes erfolgt über Index
  (beginnt in C bei `0`): `<Variable> "[" <Integer Number> "]"`
- keine Sprachelemente in C zur Verarbeitung von kompletten Feldern
  (eckige Klammern sind nur Syntaxelemente für Arbeit mit den Indizes)
- Arbeit mit Feldern erfolgt mit Standardfunktionen oder elementweise

Arbeit mit Feldern

- Name des Feldes ist die Anfangsadresse des Feldes, also die Adresse des ersten Elementes (mit Index 0): `feld` ≙ `&feld[0]`
- wird Anfangsadresse des Feldes benötigt (z.B. bei Parametern im Funktionenaufruf), kann einfach Name des Feldes angegeben werden
- Beispiel: Feld ganzer Zahlen

Index	0	1	2	3	4	5	6	7
Inhalt	12	7	-3	8	-13	0	5	1

Vereinbarung von Feldern und Zugriff

```
int feld1[10];      /* Vereinbarung */

// …
```

Elementanzahl muss bei Compilierung bekannt sein.

Es kann ein ganzzahliger aber konstanter Ausdruck angegeben werden.

Ab C99 kann die Anzahl der Feldelemente während der Laufzeit ermittelt werden, kann aber danach nicht mehr verändert werden (semidynamisch).

```
hilf= feld1[i];     /* Inhalt des Feldelementes */
                    /* wird kopiert             */

//…
```

Feldelemente können durch beliebigen ganzzahligen Ausdruck adressiert werden, auch durch Variablen.

Feld-Vereinbarung

- Beispiele für Feldvereinbarungen

```
#define ANZ 10
int feld1[ANZ];     // zufällige Anfangswerte
int feld2[] = { 12, 7, -3, 8, -13, 0, 5, 1 }; // 8 Elemente
float feld3[15];
int k = 8;
double feld4[k];    // nicht bis C90; erst ab C99 erlaubt!
```

- Länge des Feldes ist mit Vereinbarung festgelegt; kann nachträglich nicht mehr verändert werden!

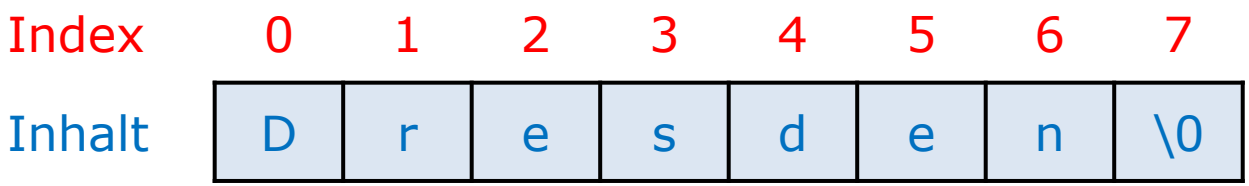
- dynamische Speicherverwaltung kann mit den Funktionen `malloc`, `calloc` und `free` realisiert werden

INDIZES

- Zugriff auf die Feldelemente erfolgt über den Index
- Index kann ein beliebiger ganzzahliger Ausdruck sein
- keine Überwachung der Überschreitung von Feldgrenzen
  (weder bei Compilierung noch bei Ausführung wird geprüft, ob über die vereinbarten Grenzen hinaus zugegriffen wird → Speicherschutzproblem)
- Index kann größer sein als Anzahl der Elemente oder auch negativ
  (widerspricht nicht den Syntaxregeln)
- Verantwortung für die Arbeit mit Feldindizes liegt beim Programmierer!

Zeichenkette (String, Text)

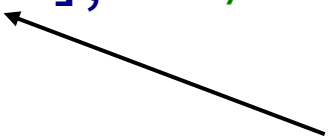
- Zeichenketten (Texte, Strings) sind in C Felder einzelner Zeichen mit dem Ende-Kennzeichen `\0` (Endlimiter) als letztem Zeichen



- Speicherplatz für das Ende-Kennzeichen muss berücksichtigt werden
- mangels Sprachelementen für Feld-Arbeit muss Verarbeitung von Texten in C mittels dafür vorgesehener Standardfunktionen (oder in eigenen Programmstücken mit Schleifen) elementweise erfolgen

Vereinbarung von Texten

```
char text1[80+1];     /* Vereinbarung */
```



Anzahl maximal benötigter Einzelzeichen zzgl. Speicherplatz für die binäre Null

- binäre Null (\0) entspricht nicht dem Zeichen „0“ (ASCII 48, 0x30, 060)
- bei binärer Null sind alle Bits mit 0 belegt

```
char zeichen = 0;     /* ohne Delimiter und ohne \ */
char zeichen = '\0'; /* mit Delimiter aber mit \ */
```

- Datentyp char wird in C wie ein ganzzahliger Datentyp behandelt

Zeiger (Pointer)

Zeiger (Pointer) (1/3)

- Arbeit mit Pointern macht C-Programme effizienter (Arbeit mit Adressen)
- Pointer-Arbeit nur möglich, wenn entsprechende Daten (worauf Pointer zeigen) vorhanden sind
- Pointer selbst belegen auch Speicherplatz
  (so viel, wie zur Speicherung einer Adresse nötig ist)
- Pointer-Typ muss dem Typ dessen, worauf gezeigt wird, entsprechen
- mit Pointern kann gerechnet werden (Zeiger-Arithmetik)
- alles was mit Pointern realisiert werden kann, kann auch mit anderen Mitteln (Felder, Indizes, …) realisiert werden

Zeiger (Pointer) (2/3)

```
int *ptrInt;      // Vereinbarung eines Zeigers,
                  // der auf int-Daten zeigt
int feld[10];
```

Der Stern bei der Vereinbarung deklariert die Vereinbarung eines Pointers.

Der Inhalt des Pointers ist zufällig, kann aber bei der Vereinbarung einen Anfangswert erhalten.

```
// …

ptrInt = feld;    /* Pointer erhält Adresse des */
                  /* Feldes (des 1. Elementes)  */
// …              /* -- gleichbedeutend mit:    */
                  /*         ptrInt = &feld[0]; */
```

Irgendeine Zuweisung dieser Art muss immer vor der Arbeit mit Zeigern vorhanden sein.

Zeiger (Pointer) (3/3)

- Beispiele für Pointer-Vereinbarungen

```
int *ptrInt;            // zufälliger Anfangswert
int *ptrFeld[5];        // Feld von fünf Zeigern
char text[80+1];        // String mit maximal 80 Zeichen
char *ptrChar = text; // Anfangsinitialisierung
```

- Name eines Feldes ist die Adresse auf das erste Element
  (wie ein Zeiger auf dieses Element)

- Name eines Feldes darf nicht auf linker Seite einer Zuweisung stehen
  (ist nämlich ein konstanter Zeiger)

Zeiger-Arithmetik (1/2)

- Zeigervariablen enthalten Adressen, mit denen gerechnet werden kann

```
zeiger++; bzw. ++zeiger;  /* Inkrement */

zeiger--; bzw. --zeiger;  /* Dekrement */
```

- Zeiger verweist auf Element mit der als datentyp-abhängigen Offset von der Startadresse berechneten Adresse; insbesondere
  - bei Inkrement: Adresse des nächsten Elements
  - bei Dekrement: Adresse des vorherigen Elements
- Es werden so viele Byte weiter gezählt, wie es dem Typ der Zeigervariablen entspricht
- Verantwortung liegt beim Programmierer, dass nach allen Operationen die Zeiger auf gültige Daten zeigen

Zeiger-Arithmetik (2/2)

```
Zeiger + (/* ganzzahliger Ausdruck */)
```

```
Zeiger - (/* ganzzahliger Ausdruck */)
```

→ die ganze Zahl (Auswertung des ganzzahligen Ausdrucks) wird mit der Anzahl Byte entsprechend des Typs der Zeigervariablen multipliziert

```
Anzahl = Zeiger1 - Zeiger2;
```

→ Anzahl der Elemente zwischen den Zeigern, wenn Adresse `*Zeiger1` vor Adresse `*Zeiger2` liegt (`Zeiger1` enthält größere Zahl)

Sinn von Zeigern

effizientere Arbeit und dynamische Datenstrukturen sind der Hauptgrund für die Verwendung von Pointern in C

- indirekte Berechnung der Zieladresse:

```
i++;                /* Indexvariable erhöhen  */
feld[i] = ...;      /* Zugriff auf das Element */
-------------------------------------------------
/* interne Berechnung anhand Bytebreite des Datentyps:
   ==> feld + i * (Anzahl Byte entsprechend Datentyp) */
```

Anfangsadresse des Feldes

berechnete Adresse des Elementes

- Zieladresse mittels Zeiger:

```
ptrFeld++;          /* Zeigervariable erhöhen */

*ptrFeld= ...;      /* Zugriff auf den Inhalt */
```

→ Zeigervariable enthält bereits nach der Erhöhung die richtige Adresse