



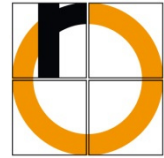
Prozedurale Programmierung

Felder

Hochschule Rosenheim - University of Applied Sciences

WS 2018/19

Prof. Dr. F.J. Schmitt



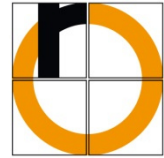
Problem

- Speicherung von 1000 Adressen
- muss man dafür wirklich 1000 Variablen einzeln anlegen?
- wie soll man diese z.B. in einer Schleife verarbeiten?
 - ⊞ man hat ja keinen gemeinsamen Namen



Überblick

- eindimensionale Felder
- mehrdimensionale Felder
- konstante Felder
- Felder als Parameter



Motivation

- Bisher:
 - ⊞ Verwendung von einfachen **Variablen** oder **Konstanten**, in denen jeweils nur **ein Wert** abgespeichert wird

- häufiges Auftreten von gleichartigen Daten in der Datenverarbeitung

- Einführung von **Feldern**
 - ⊞ um **mehrere Daten des selben Typs** zu speichern
 - ⊞ Können auch als Gruppe oder Aneinanderreihung von mehreren Variablen desselben Typs verstanden werden, die unter einem **gemeinsamen Namen** und **Index** referenziert werden können



Eindimensionale Felder (1)

➤ Definition:

`Feldtyp feldname[Feldlaenge]`

- ⊕ `Feldtyp` gibt den Datentyp für alle Feldelemente an
- ⊕ `Feldlaenge` legt die Anzahl der Elemente des Feldes fest
 - ⊕ muss zur **Übersetzungszeit** bekannt sein
 - ⊕ während Programmablauf nicht veränderbar!
- ⊕ `feldname` steht für die Adresse, an der das Feld im Speicher liegt (alle Elemente liegen hintereinander)

➤ Beispiel:

```
long zahlen[10];
```

- ⊕ 10 Variablen vom Typ `long` werden angefordert
- ⊕ $10 * 32 \text{ Bit} = 320 \text{ Bit}$ (40 Byte) verbraucht



Eindimensionale Felder (2)

- Zugriff auf die einzelnen Feldelemente
 - ⊞ Jedes Element hat einen **eindeutigen Index**
 - ⊞ Nummerierung der Feldindizes **beginnt** immer bei **0**!
 - ⊞ letztes Element eines Feldes der Länge 10 hat den Index 9

- Beispiel:

- ⊞ Feldelement mit Index 3 (viertes Element) auf 27 setzen

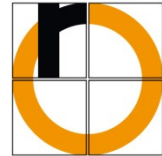
```
zahlen[3] = 27;
```

- ⊞ Speicherbelegung

long	long	long	27	long	long	long	long	long	long
0	1	2	3	4	5	6	7	8	9

Eindimensionale Felder (3)

Beispiel



```
int main(void)
{
    long zahlen[10]; // Definition von 10 long Werten
    long i;

    // Eingabe
    for(i = 0; i < 10; i++)
    {
        printf("%2ld.te Zahl: ", i+1);
        scanf("%ld",&zahlen[i]);
    }

    // Ausgabe rückwärts
    for(i = 9; i >= 0; i--)
        printf("%2ld.te Zahl: %ld\n", i+1, zahlen[i]);
}
```



Eindimensionale Felder (4)

- Feldelemente können wie einfache Variablen benutzt werden,
 - ⊞ nur anderer Variablenname: Feldname plus Index

- **Vorsicht: Beim Lesen oder Schreiben außerhalb der Feldgrenzen** können schwere Fehler passieren
 - ⊞ Programm stürzt ab, da vom Betriebssystem ein illegaler Speicherzugriff auf einen Speicherbereich, der dem Programm nicht zugeordnet ist, erkannt wird

 - ⊞ andere Daten des Programms oder sogar ein Teil des Programms wird direkt überschrieben
(nicht sofort erkennbar – kein Kompilier- bzw. Laufzeitfehler)



Eindimensionale Felder (5)

- Wie lautet der Startwert und die Bedingung bei aufsteigenden Schleifen?

Startwert = 0 und Bedingung $<$ Feldlänge

- Wie lautet der Startwert und die Bedingung bei absteigenden Schleifen?

Startwert = Feldlänge - 1 und Bedingung ≥ 0

- Welches Problem entsteht wenn Feldlänge in einem Programm verändert werden soll?

Abändern sämtlicher Grenzen im Programm schwierig
 \Rightarrow Definition der Feldlänge mit `#define`-Anweisung



Eindimensionale Felder (6)

```
#define ZAHLEN_LEN 5

int main(void)
{
    long zahlen[ZAHLEN_LEN];
    long i;

    // Eingabe
    for(i = 0; i < ZAHLEN_LEN; i++)
    {
        printf("%2ld.te Zahl: ", i+1);
        scanf("%ld", &zahlen[i]);
    }

    // Ausgabe in verkehrter Reihenfolge
    for(i = ZAHLEN_LEN - 1; i >= 0; i--)
        printf("%2ld.te Zahl: %ld\n", i+1, zahlen[i]);
}
```



Felder von Strukturen (1)

Dienen der Verwaltung von mehreren Strukturen

```
#define ADRESSE_NAME_LEN      30
#define ADRESSE_ORT_LEN      20
#define ADRESSE_STRASSE_LEN  50
#define ADRESSBUCH_LEN      100

struct Adresse_s
{
    char name[ADRESSE_NAME_LEN] ;
    long plz;
    char ort[ADRESSE_ORT_LEN] ;
    char strasse[ADRESSE_STRASSE_LEN] ;
    long nummer;
};

struct Adresse_s Adressbuch[ADRESSBUCH_LEN] ;
```



Felder von Strukturen (2)

Initialisierung

```
struct Adresse_s KleinesAdressBuch[3] =  
{  
    { "Josef Maier", 83714, "Miesbach", "Hauptstrasse", 25 },  
    { "Ludwig Huber", 83119, "Obing", "Rosenstrasse", 13 },  
    { "Karl Ganz", 83209, "Prien", "Tulpenstrasse", 8 }  
};
```

⊞ Teilinitialisierung möglich



Felder von Strukturen (3)

➤ Zugriff auf die einzelnen Elemente:

⊞ Strukturen

```
KleinesAdressBuch[0]  
KleinesAdressBuch[1]  
KleinesAdressBuch[2]
```

⊞ Attribute der Strukturen

```
KleinesAdressBuch[0].nummer = 28;
```

```
strcpy(KleinesAdressBuch[0].name, "Fritz Walter");
```

```
KleinesAdressBuch[0].name[0] = 'J';
```



Felder von Strukturen (4)

```
struct Adresse_s
{
    char name[ADRESSE_NAME_LEN] ;
    long plz;
    char ort[ADRESSE_ORT_LEN] ;
    char strasse[ADRESSE_STRASSE_LEN] ;
    long nummer;
};

struct Adresse_s Adressbuch[ADRESSBUCH_LEN] ;
```

Größe der Attribute wurde bei der Definition der Struktur fest codiert und kann während der Programmlaufzeit nicht verändert werden!



Felder von Strukturen (5)

➤ Implementierung mit Zeigern

```
struct Adresse_s  
{  
    char *name;  
    long plz;  
    char *ort;  
    char *strasse;  
    long nummer;  
};
```

```
struct Adresse_s Adressbuch[100];
```

- ⊞ Zeiger sind flexibler – jedoch aufwendigere Verwaltung (dynamische Speicherverwaltung, folgt in späterem Kapitel)
- ⊞ Felder sind starr – jedoch einfacher zu verwalten



Beispiel: Komplexe Strukturen

Kartei

Name	Vorname	0
<input type="text"/>	<input type="text"/>	1

Adresse

Straße	
PLZ	Ort

Tel

Vorwahl	Nummer
---------	--------

Umsatz

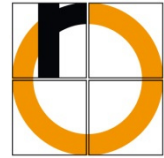
<input type="text"/>

Diagram illustrating a complex structure (Kartei) with multiple overlapping forms. The forms are labeled: Name, Vorname, Adresse, Tel, and Umsatz. The structure shows a sequence of forms, with the first form (Name/Vorname) being the base, and subsequent forms (Adresse, Tel, Umsatz) being stacked on top, indicating a hierarchical or sequential relationship. The forms are numbered 0, 1, and 99, suggesting a range of forms or a specific sequence.



Aufgabe (2)

- Legen Sie ein Feld zur Speicherung von 20 Städten an und initialisieren Sie die ersten 5 Einträge (Stadt, Land, Einwohnerzahl):
 - ⊞ Wien, A, 1700000
 - ⊞ Graz, A, 255000
 - ⊞ Berlin, D, 3430000
 - ⊞ Zuerich, CH, 365000
 - ⊞ Kopenhagen, DK, 1168000



Aufgabe (3)

- Schreiben Sie eine Funktion, welche die Attribute einer Stadt in einer Zeile ausgibt und folgendem Prototypen entspricht:

```
void ausgebenStadt(const struct Stadt_s *stadt)
```



Mehrdimensionale Felder (1)

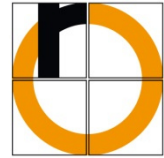
- C unterstützt auch mehrdimensionale Felder
 - ⊞ Jedes Feldelement hat **zwei oder mehr Indizes**
- Beispiel:

```
long matrix[3][4];
```

matrix

0	long	long	long	long
1	long	long	long	long
2	long	long	long	long
	0	1	2	3

Zweidimensionales Feld mit 3
mal 4 Werten



Mehrdimensionale Felder (2)

- Zugriff auf die einzelnen Elemente
 - ⊞ Ähnlich wie eindimensionale Felder
 - ⊞ Element der ersten Zeile und der zweiten Spalte soll auf 27 gesetzt werden

```
matrix[0][1] = 27;
```

- Sind in C **eindimensionale Felder**, bei denen **jedes Feldelement** wieder ein **Feld** ist
- Werden **zeilenweise abgespeichert**



Mehrdimensionale Felder (3)

➤ Beispiel:

```
long feld2D[2][4];
```

feld2D

0	long	long	long	long
1	long	long	long	long
	0	1	2	3

Umsetzung:
Zwei Felder mit 4 Elementen
von Typ `long`
werden hintereinander als Block
zu 8 Elementen angelegt

➤ Speicherbelegung:

long	long	long	long	long	long	long	long
0	1	2	3	4	5	6	7

➤ **Vorsicht: Falsche Indizierung !**

(z.B. `feld2D[0][4]` wäre 1. Element in der 2. Zeile)



Mehrdimensionale Felder (4)

- Beispiel dreidimensionales Feld

```
long field3D[2][3][4];
```

- Beachte:

- ✚ Zu groß gewählte Feldindizes führen bei mehrdimensionalen Feldern schnell zu Speicherproblemen

```
long field3D[1000][1000][1000];
```

- ✚ Benötigt nahezu 4 Gigabyte Speicher
- ✚ Auswahl von Datenstrukturen ist bei hohen Datenmengen daher gut zu überlegen

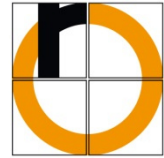


Mehrdimensionale Felder (5)

```
#define MATRIX_LEN 3
int main(void)
{
    long matrix[MATRIX_LEN][MATRIX_LEN];
    long i, j;

    // Eingabe
    for(i = 0; i < MATRIX_LEN; i++)
    {
        for(j = 0; j < MATRIX_LEN; j++)
        {
            printf("Element (%2ld, %2ld): ", i, j);
            scanf("%ld", &matrix[i][j]);
        }
    }

    // Ausgabe
    for(i = 0; i < MATRIX_LEN; i++)
    {
        for(j = 0; j < MATRIX_LEN; j++)
            printf("%ld ", matrix[i][j]);
        printf("\n");
    }
}
```



Initialisierung von Feldern (1)

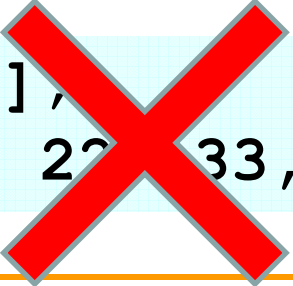
- Für jedes Element muss der Reihe nach ein Wert angegeben werden

```
long zahlen[5] = {11, 22, 33, 44, 55};
```

- Überspringen von Elementen ist **nicht** möglich
 - ⊞ Werden zu wenig Werte angegeben, so werden diese den ersten Elementen zugewiesen und die restlichen werden auf 0 gesetzt

- diese Initialisierung funktioniert nur direkt bei der Definition

```
long zahlen[5],  
zahlen = {11, 22, 33, 44, 55};
```





Initialisierung von Feldern (2)

➤ Beispiele:

```
long zahlen[5] = {0};
```

- ⊞ gesamtes Feld wird auf 0 gesetzt, da Teilinitialisierung stattfindet

```
long primzahlen[] = {2,3,5,7,11,13,17,19};
```

- ⊞ Feldlänge kann offen gelassen werden: Compiler ermittelt Feldlänge selbständig

```
sizeof(primzahlen);
```

- ⊞ liefert die Größe des Feldes in Byte
- ⊞ Vorsicht: funktioniert nur, wenn Feld komplett spezifiziert wurde
 - ⊞ problematisch z.B. bei Feldern als Funktionsparameter



Initialisierung von Feldern (3)

```
#include <stdio.h>

int main(void)
{
    long primzahlen[] = {2,3,5,7,11,13,17,19};

    // Ermittlung der Feldlänge
    const long PRIMZAHLEN_LEN = sizeof(primzahlen) / sizeof(long);

    // Ausgabe
    for(i = 0; i < PRIMZAHLEN_LEN; i++)
        printf("%ld\n ", primzahlen[i]);
}
```

Feldlänge (Anzahl Elemente) wird zur Übersetzungszeit ermittelt:

Division der durch das Feld belegten Bytes durch die Anzahl der Bytes pro Element



Initialisierung von Feldern (4)

- Analoges Vorgehen bei mehrdimensionalen Feldern

```
long field[3][3] =  
{  
    {1, 2, 3},  
    {4, 5, 6},  
    {7, 8, 9}  
};
```



Konstante Felder

- Sind die **Elemente** eines Feldes **Konstanten**, so sollte das Schlüsselwort `const` bei der Definition des Feldes angegeben werden:

```
const long primzahlen[] = {2,3,5,7,11,13,17,19};
```

- Compiler gibt Fehler aus, falls Schreibzugriff auf Feld versucht wird



Felder als Parameter (1)

- Wie werden Parameter an Funktionen übergeben?
 - ⊞ Standard: in Form von Kopien (call by value)
 - ⊞ „Ausnahme“: Bei Feldern wird **Anfangsadresse** übergeben
 - ⊞ Grund:
Das Erzeugen von Kopien von Feldern würde zu einem zu hohen Speicherverbrauch und Geschwindigkeitsverlust führen
 - ⊞ tatsächlich enthält der Feldname ohne [] die Adresse des Feldanfangs



Felder als Parameter (2) – Beispiel

```
...  
int main(void)  
{ ...  
    AusgabeFeld(primzahlen, 8); // Funktionsaufruf  
    ...  
}  
  
void AusgabeFeld(long feld[], long len)  
{  
    int i;  
    for(i = 0; i < len; i++)  
        printf("%ld\n ", feld[i]);  
  
    feld[0] = 1; // überschreibt primzahlen[0] in main()  
}
```

Eindimensionales Feld beliebiger Länge



Felder als Parameter (3)

➤ Erläuterungen zum Beispiel:

- ⌘ Feld `primzahlen` wird im Original an die Funktion `AusgabeFeld` übergeben
- ⌘ Name `primzahlen` steht genau genommen für die Adresse, an der das Feld im Speicher steht
(Synonym für die Adresse des Feldes – Zeiger)
- ⌘ diese Adresse wird dann (äquivalent zu „normalen“ Parametern) als call-by-value (also in Kopie) übergeben
- ⌘ Beim Zugriff auf die Daten, die an dieser Adresse stehen, werden die Originalelemente verwendet



Felder als Parameter (4)

➤ Mehrdimensionale Felder:

- ⊞ alle Größen bis auf die linkeste müssen angegeben werden
- ⊞ Daten werden zur Berechnung des Speicherortes benötigt

```
int main(void)
{
    ...
    int mainFeld_1[3][4];
    int mainFeld_2[3][4][5];
    ...
}

void AusgabeFeld2D(int feld[][4], long len)
{
    ...
}

void AusgabeFeld3D(int feld[][4][5], long len)
{
    ...
}
```




Aufgabe

- Schreiben Sie ein C-Programm, welches in einem Feld von ganzen Zahlen bestimmt, ob eine bestimmte ganze Zahl enthalten ist
 - ⊞ Rückgabewert Index, falls Wert enthalten
 - ⊞ Rückgabewert -1, falls Wert nicht enthalten
 - ⊞ Prototyp:

```
long sequentielleSuche(long feld[], long len, long wert)
```



Zusammenfassung

- Feldgröße muss zur Übersetzungszeit bekannt sein
- keine Überprüfung der Grenzen zur Laufzeit
- mehrdimensionale Felder
 - ⊞ werden zeilenweise gespeichert
 - ⊞ Index rechts außen läuft am schnellsten hoch
- Initialisierung
 - ⊞ mit {...}
 - ⊞ mit for Schleife
- Parameterübergabe an Funktionen
 - ⊞ Übergabe der Adresse des Felds
 - ⊞ das Feld wird „im Original“ übergeben
 - ⊞ alle Dimensionen bis auf die am weitesten links stehende müssen angegeben werden