Praxis der Programmierung

Arrays, Pointer, Parameterbergabe

Institut für Informatik und Computational Science
Universität Potsdam

Henning Bordihn

Einige Folien gehen auf A. Terzibaschian zurück.

Arrays (Felder/Vectoren)

Arrays: Motivation

- **Gegeben:** Durchschnittstempeaturen der Monate der letzten fünf Jahre

- **Aufgabe:** Berechnung von Durchschnittstemperaturen für Jahre, Monate, Jahreszeiten, ...

- *naiver Ansatz:*
  - `double`-Variable für jeden Monat (12 x 5 Variablen);
  - manuelle Berechnung

```
double t_2007_01 = 3.7;
double t_2007_02 = 4.2;
…
double t_2012_01 = 4.1;
```

```
double avg_january =
        (t_2007_01 + t_2008_01 + … + t_2012_01)/5.0;
```

⇝ Schleife zur Summenbildung ??? ⇝ keine gute Idee!!!

Arrays als besserer Ansatz

- Arrays fassen mehrere Variablen unter einem Namen zusammen
- Zugriff auf Werte über den gemeinsamen Namen + Nummer (Index)
- ähnlich zu Listen in Python
- Werte müssen einheitlichen Datentyp haben
- Größe des Arrays (Anzahl der Elemente) unveränderbar

$\rightsquigarrow$ Datentyp und Größe bei der Deklaration der Array-Variablen festlegen

Array-Definition und -Zugriff

- **Definition:** *Typname Arrayname* `[`*Größe*`];`

  Beispiele: `int ar [5];`
  `double temperaturen[12*5];`

- **Zufriff:** auf das i-te Array-Element: *Arrayname*`[i]`

  Beispiel: `ar[3]`

- Indizierung beginnt bei 0
  $\rightsquigarrow$ letztes Element: Länge des Arrays minus 1

`temperaturen`:

(Werte werden im Speicher direkt hintereinander abgelegt.)

Lösung des motivierenden Problems

- Beispiel: Durchschnitt aller Werte der letzten fünf Jahre:

```
double avg = 0;
int i;
for (i = 0; i < 12*5; ++i) {
   avg += temperaturen[i];
}
avg /= i;
```

- `for`-Schleifen oft auch zur Initialisierung von Arrays:

```
int i, ar[5];
for (i = 0; i < 5; ++i) {
   ar[i] = i+1;
}
```

$\rightsquigarrow$ `ar[0] = 1, ar[1] = 2, ar[2] = 3, ar[3] = 4 , ar[4] = 5`

Mehrdimensionale Arrays

Arrays können komplexe Daten speichern ... auch Arrays

- Verwendung mehrfacher eckiger Klammern
- Elemente sind selbst Arrays (eine Dimension niedriger)
- Beispiel: `int ar [3][4];`

`ar[0][0]`	`ar[0][1]`	`ar[0][2]`	`ar[0][3]`
`ar[1][0]`	`ar[1][1]`	`ar[1][2]`	`ar[1][3]`
`ar[2][0]`	`ar[2][1]`	`ar[2][2]`	`ar[2][3]`

Initialisierungslisten

- Elemente in geschweiften Klammern, durch Komma getrennt
- nur direkt bei der Definition

  $\rightsquigarrow$ `int ar [4] = {1, 2, 3, 4};`
- Größe in eckigen Klammern darf fehlen (*offenes Array*)

  `int ar [] = {1, 2, 3, 4};`
- fehlende Elemente werden mit `0` aufgefüllt:

  `int ar [4] = {1, 2};` $\rightsquigarrow$ `ar[0]=1; ar[1]=2; ar[2]=0; ar[3]=0;`
- *mehrdimensional*: Initialisierungsliste von Initialisierungslisten $\{\{\ldots\},\ldots,\{\ldots\}\}$

Was können Arrays in C nicht?

Ändern oder Bestimmen ihrer Größe

```
void func(int n) {
    int array[n];
    // Fehler: n dynamisch
}
```

```
int array[2048];
…
for(i = 0;i<#array??;++i) {
}
```

```
int array[10];
…
array[12] = ???
// Fehler zur Laufzeit:
// Größe nicht änderbar
```

⇝ Pointer!!!

Pointer (Zeiger)

Vier Kennzeichen einer Variablen

- Datentyp
- Variablenname
- Wert
- Adresse im Arbeitspeicher (Primärspeicher)

In **C**: Zugriff über Namen oder Adresse auf den Wert

Adressen von Variablen

- Arbeitsspeicher ist in (gleich große) *Speicherzellen* eingeteilt (z.B. je 1 Byte)
- Speicherzellen sind durchnummeriert (Hexadezimalzahlen)
- **Adresse** einer Variablen ist *Nummer* der Speicherzelle, in der ihr Speicherplatz beginnt.

Variablen und ihre Adressen

Variable `v`
speichert den Wert eines
bestimmten Datentyps
an einer Speicheradresse

Deklaration: `int v;`

`v` hat die Adresse `&v`

`&` - **Adressoperator**
ermittelt die Adresse / die Referenz
(den Pointer) auf `v`

Variablen und ihre Adressen

Variable `v`
speichert den Wert eines
bestimmten Datentyps
an einer Speicheradresse

Deklaration: `int v;`

`v` hat die Adresse `&v`

`&` - **Adressoperator**
ermittelt die Adresse / die Referenz
(den Pointer) auf `v`

Pointer `p`
speichert die Adresse von `v`,
an der ihr Wert abgelegt wird

Dekalration: `int *p;`

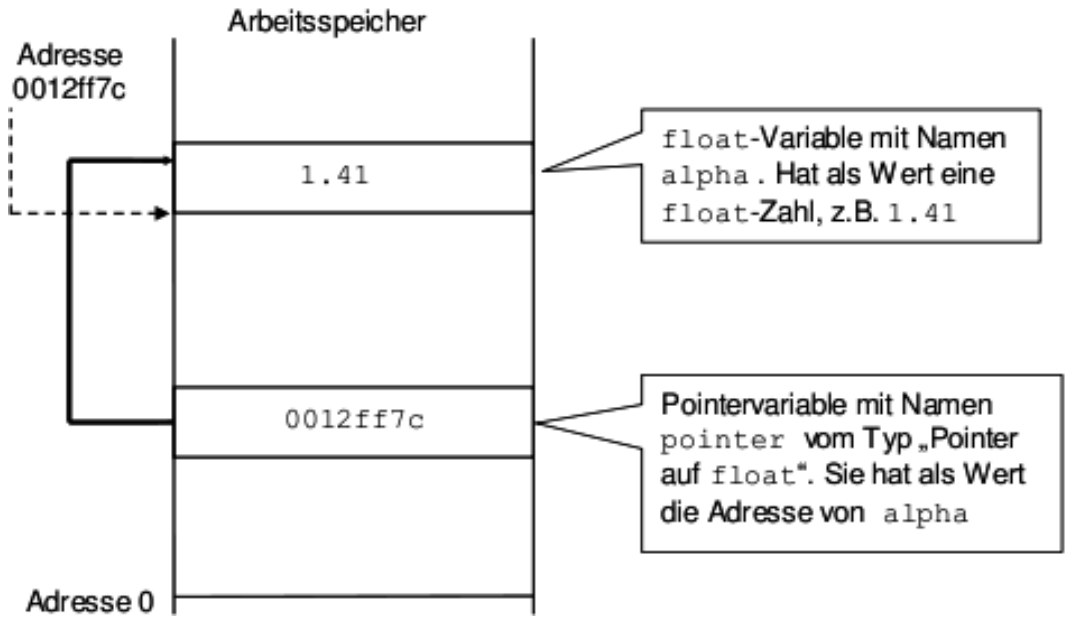
`p` *zeigt* auf Speicherplatz mit Wert `*p`

`*` - **Inhaltsoperator**
ermittelt den Wert an Speicherplatz `p`
(Dereferenzierungsoperator)

Pointer

- **Pointer:** ist Variable

  Typ: Pointer auf einen bestimmten Datentyp (z.B. float)
  Wert: $Adresse$ einer Variablen



Pointervariablen

- Definition als Pointer auf *Datentyp*
  $\rightsquigarrow$ Pointertyp und Datentyp des Speicherobjekts sind gekoppelt!

- *Typname* `*` *Pointername*;
  $\rightsquigarrow$ Datentyp: Pointer auf *Typname*

- Beispiel: `float * pointer1;`
  $\rightsquigarrow$ Pointer auf `float` mit Namen `pointer1`

- Vorsicht:

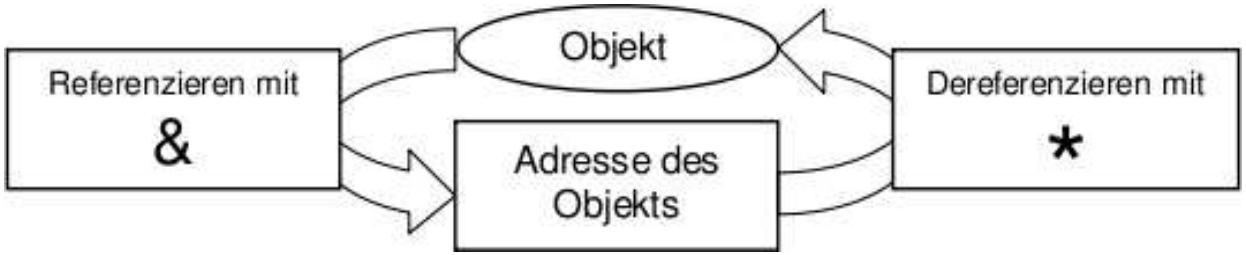
Definition	entspricht
`int * pointer, alpha;`	`int * pointer;` `int alpha;`
`int * pointer1, * pointer2;`	`int * pointer1;` `int * pointer2;`

Wertzuweisung an einen Pointer

1. **Adressoperator**
   - Adressoperator **&** liefert Pointer auf Speicherplatz einer Variablen
   - &$Variable$ $\rightsquigarrow$ Pointer auf $Datentyp$ von $Variable$
   - kann an Variable vom Typ Pointer auf $Datentyp$ zugewiesen werden:
     `pointer = &alpha;`
2. **Wert einer anderen Pointervariablen**
   - `pointer2 = pointer1;`
   - Übergabe der Adresse von einer Pointer-Variablen an eine weitere
   - nur bei Pointern auf den gleichen Typ!
3. **Konstante NULL**
   - vordefinierter Pointer, zeigt auf Adresse `0` / nie auf ein Speicherobjekt
   - darf nie dereferenziert werden ($\rightsquigarrow$ $Null\text{-}Pointer\text{-}Laufzeitfehler$)

Dereferenzieren

- **Inhaltsoperator** $*$: $*Pointer$
- $*$ und & sind invers: `*&alpha` ist äquivalent zu `alpha`



- `*pointer =` $Wert$; erlaubt
  $\rightsquigarrow$ nicht bevor `pointer` gültige Adresse speichert!

Rückblick: Benutzereingaben mit `scanf`

- formatiertes Einlesen eines Wertes mit `scanf`
- Formatelemente wie bei `printf`
- Beispiele: Einlesen einer ganzen Zahl und Speichern auf `int n`:
  `scanf(%d, &n);`

  Einlesen einer Gleitkommazahl und Speichern auf `double x`:
  `scanf(%f, &x)`

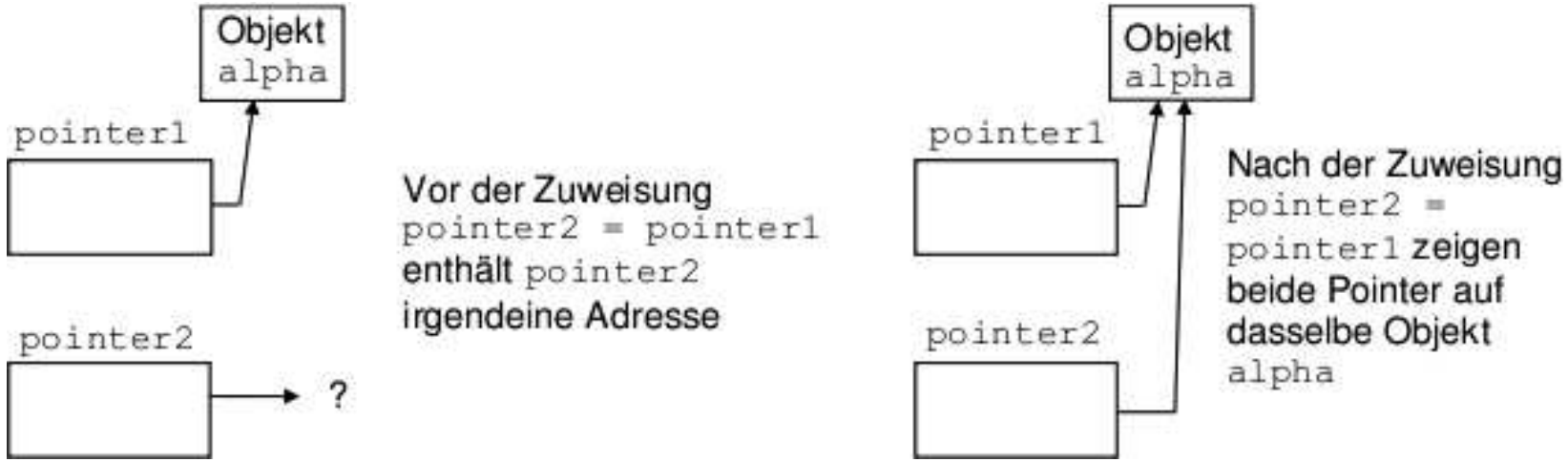
  Hinweis: Das Zeichen `&` ist nötig, da die Speicheradresse der Variablen angegeben werden muss (und kein Zugriff auf den Wert der Variablen erfolgt)

  $\rightsquigarrow$ Pointer können als Parameter an Funktionen übergeben werden!!!

Arbeit mit Pointern

- nicht initialisierte Variablen haben *irgendeinen* Wert!

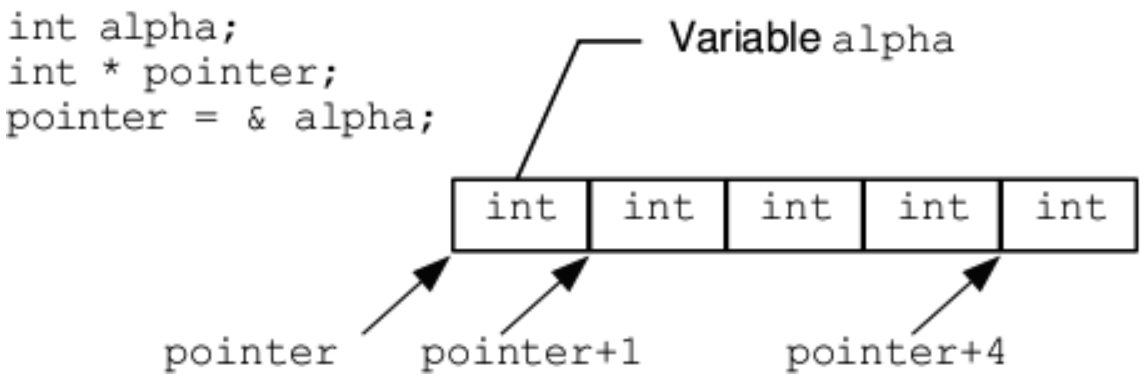
$\rightsquigarrow$ nicht initialisierte Pointer verweisen auf *irgendeine* Adresse! (*Fehlerquelle!!!*)



Pointerarithmetik

- Vergleich (== und !=)
- Addition und Subtraktion (einer ganzen Zahl n)

  Verschieben des Zeigers um n Speicherobjekte des Typs, auf den der Pointer zeigt

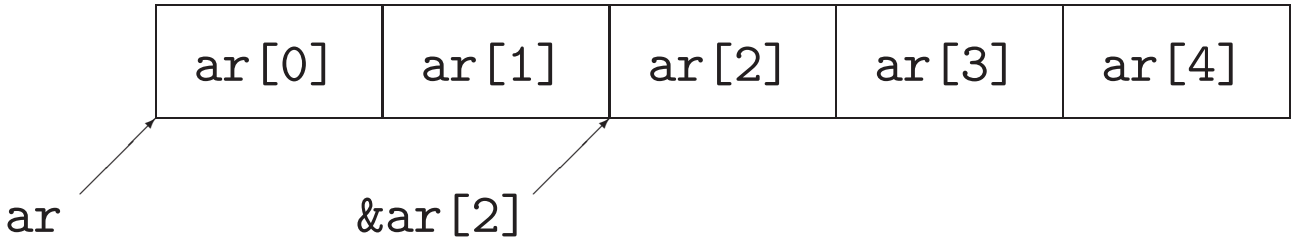


Pointer auf `void`

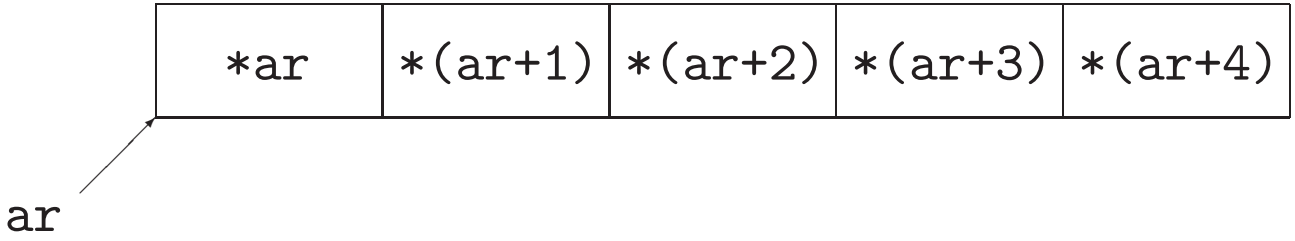
- `void *` *Pointername*`;`
- **untypisierter Pointer**: Datentyp steht nicht fest
- darf nicht dereferenziert werden (zeigt nie auf Speicherobjekte)
- kann in jeden Pointertyp umgewandelt werden (Zuweisung)
  $\rightsquigarrow$ kein Verlust an Information oder Genauigkeit
- für die Zuweisung von Pointern auf anderen Typ verwenden

Arrays und Pointer

- Name des Arrays ist konstanter Zeiger auf das erste Array-Element



- `ar[i]` ist äquivalent zu `*(ar+i)`



- Sei `ptr` ein Pointer. Dann ist `*(ptr + i)` äquivalent zu `ptr[i]`.

Pointer: Pros und Cons

- Pointer erlauben hardwarenahes Programmieren
- Pointer erlauben dynamische Datenstrukturen
- Pointer sind häufig Quelle von schwer auffindbaren Fehlern, z.B.:

```
float v;
float *p = &v;
p[0] = 0.5;
p[1] = 3.14;  // ???
```

- Pointer können zu Datenverlust führen
  - Schreiben von Werten an nicht referenzierte Adressen
  - versehentliches Überschreiben von Werten, ...

Variablen und Funktionen

Rückblick: Blöcke und Variablen

- In Blöcken definierte Variablen sind nur innerhalb dieses Blockes sichtbar.
  - $\rightsquigarrow$ auch in enthaltenen Blöcken
  - $\rightsquigarrow$ aber **nicht** in umfassenden Blöcken
- In Blöcken definierte Variablen *verdecken* gleichnamige Variablen von umfassenden Blöcken.
- In Blöcken definierte Variablen werden beim Verlassen des Blockes wieder ungültig.
  - $\rightsquigarrow$ überdeckte Variablen werden wieder sichtbar
- Funktionsrümpfe sind Blöcke!!! Lokal sind:
  - Variablen, die im Rumpf deklariert werden,
  - Parameter der Funktion

Parameterübergabe mit call-by-value

- Vereinbarung einer lokalen Variablen:
  Name und Typ des formalen Parameters
- *Kopie* des aktuellen Parameterwertes als Wert dieser lokalen Variablen
- Im Funktionsrumpf wird immer die lokale Variable verwendet.
  $\rightsquigarrow$ **kein Einfluss auf aktuellen Parameter**
  $\rightsquigarrow$ keine Seiteneffekte
- Ausdruck des aktuellen Parameters wird nur einmal ausgewertet
- verwendet in: C, C++, Java, C#, PASCAL, ...

call-by-value - Beispiel

```
int var1 = 1;
int var2 = 2;
```

Aufgabe: Tausch der Werte von `var1` und `var2`
mit einer Funktion `swap(int m, int n)`

```
void swap(int m, int n) {
   int h = m;
   m = n;
   n = h;
   }
```

$\rightsquigarrow$ Aufruf `swap(var1, var2)` ohne Effekt! (*Warum?*)

Andere Methoden der Parameterübergabe — nicht in C

- **call-by-value-result**
  - zunächst wie call-by-value:
    - lokale Variable mit Wert des aktuellen Parameters initialisiert
    - keine Änderung am aktuellen Parameter
  - am Ende (z.B. bei `return;`):

    eine weitere automatische Wertzuweisung

    $\rightsquigarrow$ Wert der lokalen Variablen $\longrightarrow$ aktueller Parameter

- **call-by-name**
  - textuelle Übergabe des aktuellen Bezeichners
  - Im Funktionsrumpf wird jedes Vorkommen des formalen Parameters textuell ersetzt.

    $\rightsquigarrow$ Seiteneffekt: kann aktuellen Partameter beeinflussen
  - Ausdruck des aktuellen Parameters wird mehrfach ausgewertet
  - kann zu Konflikten mit globalen Variablen führen

    (falls Variablen des Funktionsrumpfes einen Bezeichner verwenden, der Teil eines aktuellen Parameters ist)

- **call-by-reference**
  - Vereinbarung von lokalen Variablen für alle Variablen in den aktuellen Parametern
  - referenzieren die Speicheradressen der Variablen in den aktuellen Parametern
  - Zuweisungen beeinflussen die Werte der aktuellen Parameter direkt
  - **in C**:
    Call-by-Reference-Semantik durch Übergabe von Pointern als Parameter

```
void swap(int *m, int *n) {
    int h = *m;
    *m = *n;
    *n = h;
}
```

Arrays und Pointer als Parameter

- **Problem:** Größe kann nicht ermittelt werden
  $\rightsquigarrow$ muss vom Programmierer übergeben werden

- **Beispiel:** Summe der Arrayelemente

```
int sum(int[] p, int n) {
   int s = 0, i = 0;
   for(i=0; i<n; ++i) {
      s += p[i];
   }
   return s;
```

```
int sum(int *p, int n) {
   int s = 0, i = 0;
   for(i=0; i<n; ++i) {
      s += *(p++);
   }
   return s;
```