Praxis der Programmierung

Arrays, Pointerarithmetik, Konstanten, Makros

Institut für Informatik und Computational Science
Universität Potsdam

Henning Bordihn

Einige Folien gehen auf A. Terzibaschian zurück.

Arrays (Felder/Vektoren)

Arrays: Motivation

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

```
double t_2011_01 = 3.7;
double t_2011_02 = 4.1;
  ...
double t_2015_12 = 4.2;
```

- **Aufgabe:** Berechnung von Durchschnittstemperaturen für Jahre, Monate, Jahreszeiten, ...
- *naiver Ansatz:*
  - `double`-Variable für jeden Monat (12 x 5 Variablen);
  - manuelle Berechnung

```
double avg_january =
        (t_2011_01 + t_2012_01 + t_2013_01 + t_2014_01 + t_2015_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};`
- fehlende Elemente werden mit `0` aufgefüllt:

  `int ar [4] = {1, 2};` $\rightsquigarrow$ `ar[0]=1; ar[1]=2; ar[2]=0; ar[3]=0;`
- Größe in eckigen Klammern darf fehlen (*offenes Array*)

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

Was können Arrays in C nicht?

Ändern oder direktes Abfragen ihrer Größe

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

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

```
(int) (sizeof(array)/sizeof(int))
```

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

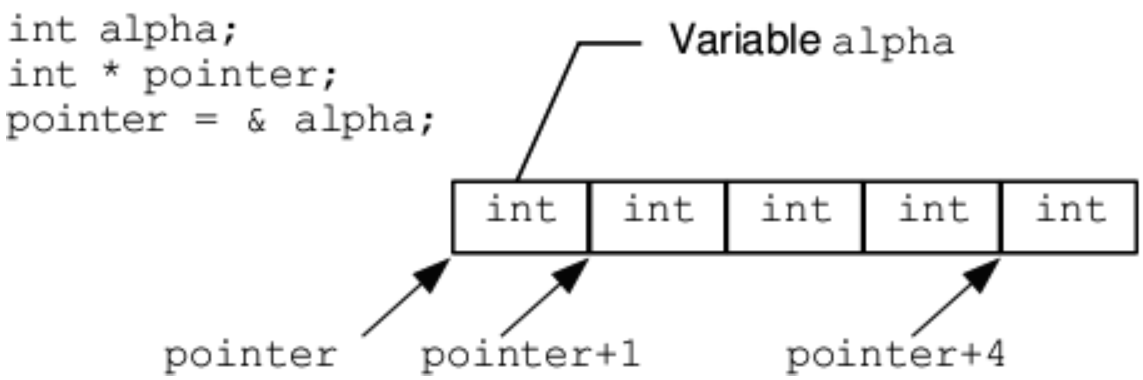
⇝ Pointer!!! ⟶ Vorlesung „Dynamische Speicherverwaltung“

Pointer und Arrays

Pointerarithmetik

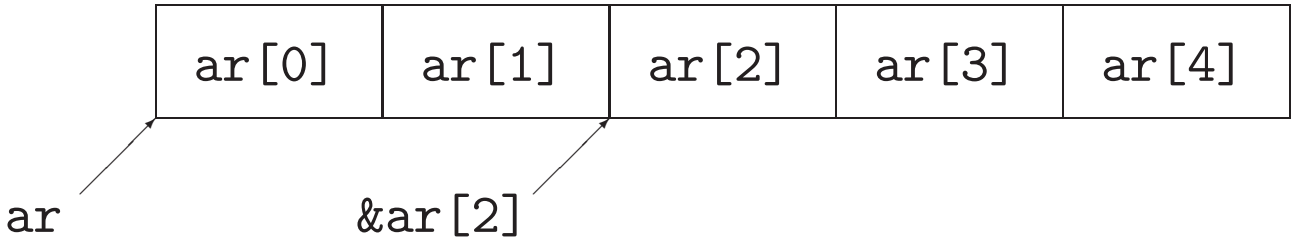
- Vergleich (== und !=) für Pointer desselben Typs
- Addition und Subtraktion (einer ganzen Zahl n)

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

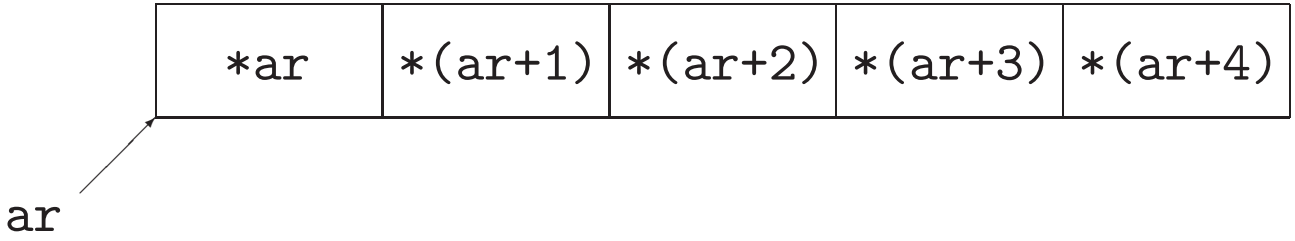


Arrays und Pointer

- 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]`.

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;
}
```

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

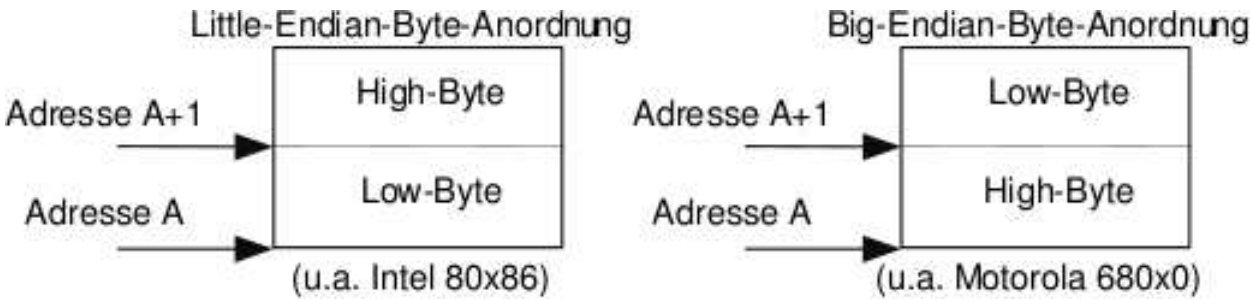
Verwendung von `void`-Pointern — Beispiel

Abgreifen des ersten Bytes eines `int`-Wertes (als `unsigned char`):

```
int zahl = n;
int * pointer1 = &zahl;
unsigned char * pointer2;   // soll auf dieselbe Adresse wie pointer1 zeigen!

// pointer2  = pointer1;    unmoeglich (verschiedene Typen)

void * dummy;

dummy = pointer1;           // korrekt bei
pointer2 = dummy;           // Little-Endian-Byte-Anordnung
```

Byte-Anordnungen

... abhängig von der Rechnerarchitektur:



Bei High-Endian-Byte-Anordnung müssen Pointer mit Hilfe der Pointerarithmetik zum gewünschten Byte bewegt werden.

Pointer: Pros und Cons

- Pointer erlauben hardwarenahes Programmieren
- Pointer erlauben dynamische Datenstrukturen
  $\longrightarrow$ Vorlesung „Dynamische Speicherverwaltung“
- 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, ...

Konstanten

Konstanten

- Literale (vordefinierte Konstanten elementarer Typen)
- Variablen, die mit Typattribut `const` definiert sind
  - `const` *Datentyp Bezeichner* `=` *Initialisierung*`;`
  - Variable nach Initialisierung schreibgeschützt
  - *Beispiel:* `const double PI = 3.1415927;`
  - *Konvention:* Bezeichner aus Großbuchstaben
  - für Variablen, Pointer, Parameter

$\rightsquigarrow$ Schutz vor unbeabsichtigten Änderungen

$\rightsquigarrow$ "Dingen einen Namen geben"
(z.B. PI, ROT, GELB, MONTAG, DIENSTAG, ...)
$\rightsquigarrow$ vereinfacht Lesbarkeit und Wartung des Quellcodes

Konstanten (2)

- Aufzählungstypen
- Definition mit Schlüsselwort `enum` und "Mengenschreibweise"
  $\rightsquigarrow$ Definition eines neuen Datentyps mit endlich vielen konstanten Werten
- Deklaration von Variablen dieses Typs mit Schlüsselwort `enum`

```
enum ausbildung {
       HAUPTSCHULE,
       REALSCHULE,
       ABITUR,
       BERUFSAUSBILDUNG,
       BACHELOR,
       MASTER
};
```

```
int akademiker (enum ausbildung a) {
     if (a == BACHELOR
      || a == MASTER)
               return 1;
     else
               return 0;
}
```

Aufzählungstypen

- Typ- und Variablendefinition können zusammen erfolgen:

```
enum boolean {FALSE, TRUE} b;
```

- Vereinbarungen ohne *Etikett*:

```
enum {FALSE, TRUE} b;
```

  $\rightsquigarrow$ Variablendefinition muss hier erfolgen; kein Typname vereinbart

- in **C**: keine Typprüfung durch den Compiler:

```
b = TRUE;    // o.k., typgerechte Verwendung
b = 5;       // weder Compiler- noch Laufzeitfehler !!!
```

Konstanten (3)

- symbolische Konstanten
  - `#define` $Name$ $Wert$
  - Präprozessor ersetzt $textuell$ alle Vorkommen von $Name$ durch $Wert$
  - *Beispiel:* `#define PI 3.1415927`
  - *Konvention:* $Name$ aus Großbuchstaben

$\rightsquigarrow$ keine Typprüfung durch den Compiler

Makros

- Verallgemeinerung des Vorgehens bei der Definition symbolischer Konstanten
- Textuelle Einsetzung funktioniert mit jeder Zeichenkette.
- Vermeidung lästiger Code-Wiederholungen durch Makro-Definition
- Beispiel: `printf("\n")` als Anweisung für den Zeilenvorschub

```
#include <stdio.h>
#define zv printf("\n")

zv;
printf("Hallo Praeprozessor!");
zv;
```

Makros — Beispiel

```
#include <stdio.h>
#define zv printf("\n")
#define printar for (i=0;i<4;++i) printf("%d : %d\t", i, ar[i])

int main() {
     zv;
     int ar[4] = {10,20,30,40};
     printar;
     zv;
     int i;
     for(i=1;i<4;++i)
          ar[i]++;
     printar;
     zv;
     return 0;
}
```

Soll das Ausgabeformat für das Array `ar` *verändert werden, braucht nur die Makrodefinition angepasst werden.*