Einführung in die Programmierung

Arrays, Zeiger, Strings

Arvid Terzibaschian

Arrays

Arrays: Motivation

- Gegeben:
  - monatliche Durchschnittstemperaturen der letzten 5 Jahre
- Gesucht:
  - Mittelwerte für Jahre, Monate, Jahreszeiten, …
- Wie modelliert man das?
  - Variable für jeden Messwert!?

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

  - Berechnung per Hand!?

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

- keine gute Idee!

Arrays: Motivation

> Gegeben: Temperaturtabelle
> Gesucht: Mittelwerte für Jahre, Monate, …
> **Wie modelliert man das?**

- Arrays fassen mehrere Variablen eines Datentyps zusammen in einer Array-Variable

```
float temperaturen[12*5];
```

Datentyp — Array-Name — Anzahl der Elemente

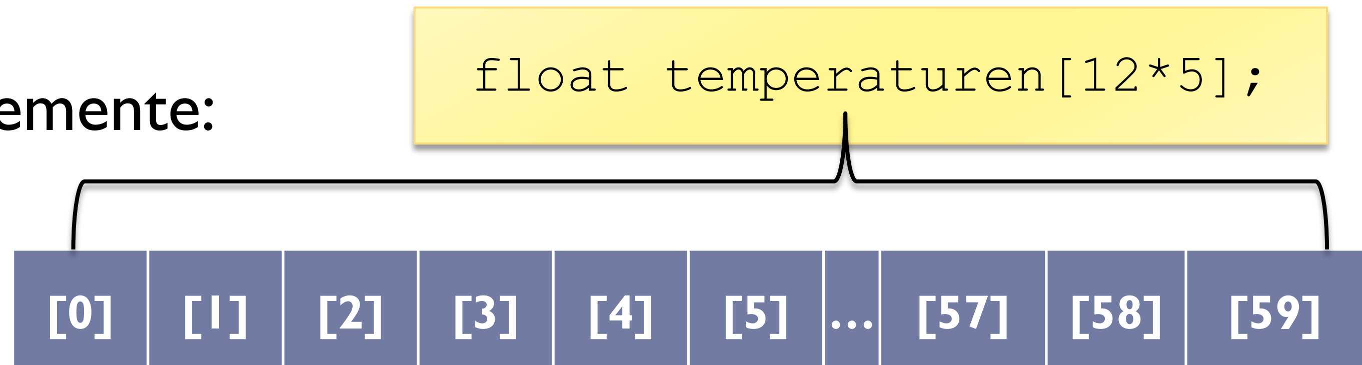
  - Datentyp: alle Elemente haben gleichen Typ
  - Name: gleiche Regeln wie für Variablen
  - Größe: Anzahl der Elemente,
    - darf nicht dynamisch verändert werden!

Arrays: Motivation

Gegeben: Temperaturtabelle
Gesucht: Mittelwerte für Jahre, Monate, …
Wie modelliert man das?

- Arrays modellieren Tabellen

- Einzelne Elemente:



  - Indizes beginnen bei 0 [entspricht „Entfernung zum Anfang“]
  - letztes Element = #Elemente – 1
  - alle Elemente auch direkt hintereinander im Speicher
  - Zugriff per Variable möglich

Arrays: Motivation

Gegeben: Temperaturtabelle
Gesucht: Mittelwerte für Jahre,
Monate, …
Wie modelliert man das?

```
float temperaturen[12*5];
```

- Arrays modellieren Tabellen

- Beispiel: Durchschnitt aller Elemente

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

- Arrays speichern viele komplexe Daten wie z.B. Matritzen, Vektoren, Bilder und Texte

Arbeiten mit Arrays

```
float t[60];
```

- Schreiben und Lesen von Arrays:
  - Setzen eines Wertes

```
t[2] = 1.4;
```

  - Lesen eines Wertes

```
double v = t[2];
```

- Initialisieren von Arrays:

```
float t[5] = {1.0, 0.2, 3.4, 1.8, 2.7 };
```

  - direkt mit {} oder per Schleife
  - uninitialisierte Arrays enthalten zufällige (nicht-definierte) Werte!

```
float t[5];
for(i = 0;i<5;++i)
      t[i] = 0;
```

Was können Arrays **in C** nicht?

- Arrays können ihre Größe nicht zur Laufzeit bestimmen oder ändern:

```
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
```

Aber: für alle diese Probleme gibt es Lösungen auch in C!

Zeiger

Variablen, Speicher, Adressen

Zeiger (Pointer) und C

- Zeiger: Fluch und Segen

  - Zeiger ermöglichen effizientes, hardwarenahes Programmieren
  - Zeiger sind in kaum einer anderen Sprache so präsent
  - Zeiger sind nicht immer einfach zu handhaben
  - Zeiger können zu Speicherverlust und schwer auffindbaren Fehlern führen
  - Zeiger ermöglichen komplexe Datenstrukturen
  - Zeiger ermöglichen dynamische Datenstrukturen

Zeiger (Pointer) und Variablen

- Variable
  - ist **Wert** eines bestimmten Datentyps
- Deklaration:

```
int v;
```

- Variable **hat** Speicheradresse

```
&v
```

- & - Adressoperator ermittelt Adresse (Referenz) von v
  - diese Adresse ist ein Zeiger

- Zeiger
  - ist **Adresse (Referenz)** einer bestimmten Variable
- Deklaration:

```
int *p;
```

- Zeiger **referenziert** Variable

```
*p
```

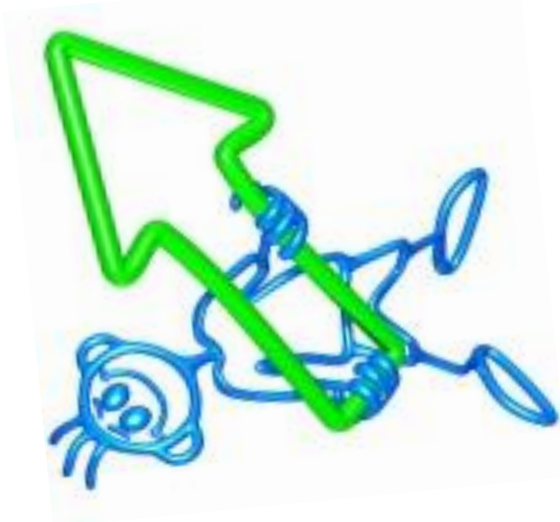
- * - Dereferenzierungsoperator ermittelt Wert der Variablen
  - Wert kann wie Variable gelesen und geschrieben werden

Zeiger und Variablen und Speicher

```
float *p;

p = &v;

*p -= 10;
```



Programmspeicher

Adresse	Wert
(NULL)	
0x0001	
0x0002	
0x0003	v 89.2
0x0004	
0x0005	
0x0006	

```
float v = 98.2;

v += 1;
```

Ausdruck	v	&v	p	*p
Wert	89.2	0x0003	0x0003	89.2

Zeiger (Pointer)

```
float v;
float *p;
```

- Zeiger sind Adressen
  - von Variablen
  - Zuweisung einer Adresse p = &v;
  - spezielle Adresse für ungültige Zeiger:
    - p = NULL; *markiert* p als „ungültig“..

```
if(p == NULL) …
```

- Zeiger können dereferenziert werden
  - *p ist Variable
  - *p = 224; entspricht üblicher Zuweisung einer Variablen an Adresse p
- Zeiger sind typisiert
  - sind Adresse eines bestimmten Datentyps
- Zeiger können an Funktionen übergeben werden
  - erlauben Call-By-Reference (~ by-Pointer)-Semantik

Zeiger und Parameter

- Zeiger können Parameter von Funktionen sein
- Betrachte 2 Funktionen:

```
void inc1(int *pI) {
    *pI += 1;
}
```

```
void inc2(int i) {
    i += 1;
}
```

- was ist der Unterschied zwischen…?

```
int k = 0;
inc1(&k);
```

```
int k = 0;
inc2(k);
```

- Call-by-Reference (~Pointer) vs. Call-by-Value!
  - vgl. auch `printf("%d",i)` vs `scanf("%d",&i)`

Zeiger und Arrays

```
float *p
```

- Dereferenzierung eines Zeigers p auch mit Arraysemantik:

```
*p = 2.14
```

=

```
p[0] = 2.14
```

- Warum?
  - **Array in C ist gleich Zeiger auf erstes Element!!!**

```
float v[3];

v[0] = 0.2;
v[1] = 0.5;
v[2] = 3.14;
```

=

```
float v[3];
float *p = v;
p[0] = 0.2;
p[1] = 0.5;
p[2] = 3.14;
```

- jeder! Zeiger kann wie Array benutzt werden
  - Fluch und Segen: Quelle vieler Fehler aber auch sehr nützliches Werkzeug für optimale Datenverarbeitung

Zeiger und Arrays

```
float *p
```

- Jeder Zeiger kann wie Array benutzt werden
  - D.h. auch Zeiger auf Variablen können wie Arrays benutzt werden

```
float v;
float *p = &v;
p[0] = 0.2;
p[1] = 0.5;   //??
p[2] = 3.14;  //??
p[3000] = 2;  //??
```

- eine der häufigsten Fehlerquellen:
  - entweder Speicherzugriffsfehler
  - oder (noch schlimmer) kein Fehler aber undefiniertes Verhalten
- beim Umgang mit Pointern und Arrays in C ist grundsätzlich der Programmierer verantwortlich!
  - Compiler und Laufzeitumgebung prüft nur unzureichend
  - Schreiben über Array-/Pointergrenzen ist klassische „Sicherheitslücke“ in Software

Zeiger, Arrays und Funktionen

- Arrays können wie Zeiger an Funktionen übergeben werden:
  - Zeiger und Array können hier synonym verwendet werden

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

=

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

- Problem: Größe des Arrays nicht bekannt!
  - Erinnere: Array = Adresse des ersten Elements = Zeiger auf erstes Element
  - weder Array noch Zeiger erlauben Abfrage der Anzahl der Elemente

Zeiger, Arrays und Funktionen

- Arrays können wie Zeiger an Funktionen übergeben werden:
  - Zeiger und Array können hier synonym verwendet werden

```
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[i];
    }
    return s;
}
```

- Größeninformation muss vom Entwickler gespeichert, d.h. manuell verwaltet werden!

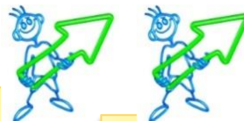
Zeigerarithmetik

```
float v[20];
float *p = v;
```

- Zeiger sind Adressen
- Zeigerarithmetik erlaubt es, einfach mit Adressen zu rechnen:

Erinnerung: Zeiger können wie Arrays behandelt werden!

- setze Zeiger auf nächstes Element: `p++ oder ++p`
- setze Zeiger auf vorheriges Element: `p-- oder --p`
- setze Zeiger n-Elemente vor: `p = p+n od. p += n;`
- setze Zeiger n-Elemente zurück: `p = p-n od. p -= n;`



`p = &v[2]`

`p++`

`p + 5`

Zeigerarithmetik vs. Arrays

- Summe eines Arrays

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

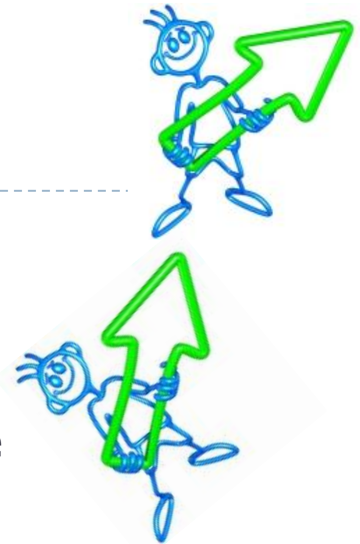
```
int sum(int *p,int n) {
    int s = 0,i=0;
    for(i = 0;i<n;++i) {
        s += *(p++);
    }
    return s;
}
```

- typischen 2 Arten zum durchlaufen eines Arrays:
  - Indexzugriff oder Zeigerinkrementierung

Zusammenfassung Zeiger

- Zeiger sind Adressen …
  - … von Variablen – mit &-Operator bestimmbar
  - … von Arraylementen
- Zeiger sind typisiert
  - Zeiger können dereferenziert werden
    - mit *-Operator
    - mit dem Array-Operator []
  - Wert der Zielvariablen kann gelesen oder geschrieben werden
- Zeiger (und damit auch Arrays) können Funktionen übergeben werden
  - erlauben Call-By-Reference-Semantik
- Zeiger erlauben Adressarithmetik
  - erlauben effizientes Zugreifen von Arrayelementen

Ausblick: Zeiger auf Zeiger



- Zeiger ist eine Adressvariable
  - Adressevariable ist im Speicher abgelegt
  - Speicherort von Adressvariable ist Zeigeradresse
- Prinzip ist:

  Zeiger auf Zeiger = Adresse einer Zeigervariable

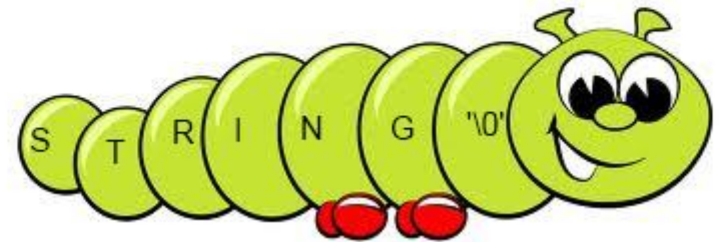
```
float v = 98.2;
float *p = &v;          // Zeiger auf v;
float **p2 = &p;        // Zeiger auf p [Adresse von Zeiger auf v]
```

- Es gilt: `*p2 == &v` und `**p2 == v;`
  - (nahezu) beliebige Verkettungstiefe erlaubt!
  - erlauben z.B. mehrdimensionale Arrays u.v.m.

Strings

Zeichenketten in C

Strings in C

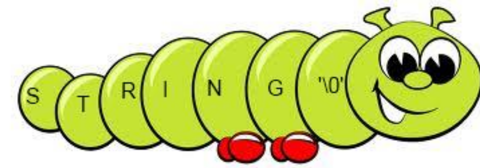


- Strings sind „Zeichenketten“
- statische Strings werden in C mit Anführungszeichen gekennzeichnet
  - bisher bei printf und scanf benutzt

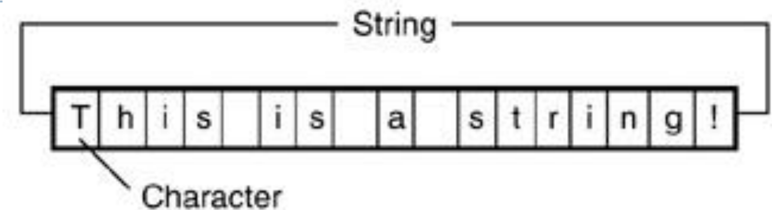
```
printf("Hallo, ich bin ein String!");
```

- Welchen Typ hat ein String?
- Wie kann man einen String in einer Variablen ablegen?
- Wie kann man Strings manipulieren?
- …

Strings in C



- Strings sind „Zeichenketten“



- Zeichen in C:

```
char ein_zeichen;
```

- Zeichenkette in C:

```
char zeichenkette[] = "ich bin eine";
```

  - Strings sind Character-Arrays!

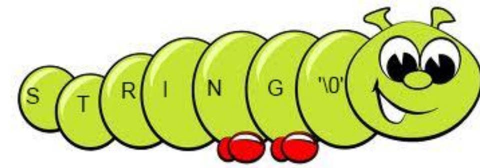
  Erinnere: Array = Zeiger auf erstes Element

  - String ist Zeiger auf ersten Character!

```
char *zeichenkette = "ich bin eine";
```

- Strings in C sind
  - Character-Arrays oder
  - Zeiger auf den ersten Buchstaben (Character)!

Strings in C



String
T h i s i s a s t r i n g !
Character

- Strings in C sind
  - Character-Arrays oder
  - Zeiger auf den ersten Buchstaben (Character)!
- Definition und Initialisierung von Strings
  - als klassisches Array

```
char s[] = {‘H‘,‘a‘,‘l‘,‘l‘,‘o‘,‘\0‘};
```

  - *besser*: per String-Syntax

```
char *s = “Hallo“;
```

- Ende von Strings muss mit Zeichen ‘\0‘ gekennzeichnet sein.
  - erlaubt Funktionen und Programmcode das Ende eines Strings zu ermitteln
  - auch „**nullterminierte Strings**“ genannt

Statische vs. dynamische Strings

```
char *str = "Hallo";
```

- statischer String
  - darf *nur gelesen* werden!
  - Daten “Hallo“ direkt im Binärcode abgelegt
    - str zeigt auf statischen Speicherblock
    - schreiben auf z.B. str[2000] kann u.U. Maschinencode während der Laufzeit verändern

```
char str[1024];
```

- dynamischer String
  - kann *verändert* werden!
  - Speicherplatz wird im Programmspeicher reserviert
  - für Strings dynamischer Länge:
    - Tools für dynamische Speicherverwaltung

Strings benutzen

Stringfunktionen:
```
#include <string.h>
```

- auf einzelne Zeichen zugreifen
  - Zugriff wie per Array oder Zeiger:

```
char str[] = "Hallo";
char str2[1024];
```

```
char c1 = str[0]; // == 'H'
char c2 = str[1]; // == 'a'
char c3 = str[5]; // == '\0'
```

- Strings ausgeben:

```
printf(str);
```

oder

```
printf("Dies ist ein String %s",s);
```

  - mit %s können Strings in formatierte Ausgabe eingebaut werden
- Strings einlesen:

```
char str[1024];          // Platz für max. 1024 Zeichen
printf("Bitte geben Sie ihren Namen an:");
fgets(str,1024,stdin); // String mit max. Länge von 1024 lesen
                          stdin bedeutet: Vom Standard-Input
```

Strings benutzen

```
char str[] = "Hallo";
char str2[1024];
str2[0] = '\0';
```

- String kopieren:
  - Achtung: Zielstring muss genug Platz für Quellstring haben

```
// Kopiere 2. Argument IN str2
strcpy(str2,"Kopiere mich bitte");
```

- Strings verketten (konkatenieren):

```
// Hänge 2 Strings nacheinander ans Ende von str2
strcat(str2,"Endlich wieder ");
strcat(str2," zusammen! ");
```

- Strings "löschen"
  - Stringterminator setzen: str[0] = '\0';
- Länge eines Strings ermitteln

```
// ermittle die Länge (OHNE Stringterminator '\0')
int len = strlen("Kopiere mich bitte");
```

Strings benutzen

```
char str0[] = "Hallo";
char str1[] = "Hallo2";
char str3[1024];
str2[0] = '\0';
int n = 9984;
```

- Strings vergleichen
  - Achtung: strcmp gibt **0** aus, wenn die Strings gleich sind

```
// Wenn str0 und str1 gleich sind …
if(strcmp(str0,str1) == 0)
       …
```

- Zahl in String umwandeln
  - **s**printf arbeitet wie printf *aber* gibt Ergebnis in String aus

```
// Wandelt n in String um und schreibt Ergebnis in str3
sprintf(str3,"%d",n);
```

- String in Zahl umwandeln
  - **s**scanf arbeitet wie scanf *aber* liest Eingabe aus String

```
// Konvertiert String s in Double-Wert nach x
char *s = "2048.68";     float x;
sscanf(s,"%f",&x);
```

Vielen Dank!

- Bei Fragen einfach eine Mail an:
  - arvid@cs.uni-potsdam.de