APPLIED COMPUTER SCIENCES

FH | JOANNEUM
University of Applied Sciences

Systemnahe Programmierung
SS 2024

Unit 3

Helmut Lindner

Arrays und Zeichenketten

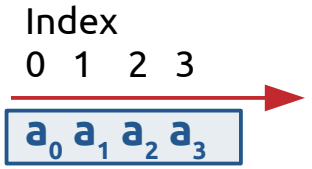
Arrays

Arrays in C bestehen aus aufeinander folgenden Elementen gleichen Typs im Speicher. Sie haben eine feste Dimension, die bei der Deklaration festgelegt wird.

```
int arr[20];
float matrix[4][4];
```

-> Mehrdimensional Zeilen/Spalten

Der Arrayindex beginnt bei 0 :



Initialisierung

```
int arr[] = {1,2,3,4,5}; → Compiler setzt die Länge automatisch
float matrix[3][4] = {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
```

Verwendung

```
a = arr[2];
matrix[0][i+1] = elem;
```

Arrays und Zeichenketten

Verwendung

Die Elemente eines Arrays werden mit ihrem Index angesprochen.

Beispiel: Initialisieren eines Arrays mit 0

```
int arr[10];

for (int i=0; i<10; i++)
    arr[i] = 0;
```

Das Array selbst ist eine Konstante, Zuweisung funktioniert nicht:

```
int arr1[2]={1,2};
int arr2[2]={3,4};

arr2 = arr1; → Geht nicht!
```

Arrays und Zeichenketten

Größe von Arrays

Für Arrays gibt es in C keine `len()` Methode oder ähnliches.

Um die Anzahl der Elemente in einem Array zu berechnen wird der `sizeof()` Operator verwendet.

```
int arr[20];
```

Anzahl der Elemente = `sizeof(arr)` ?
→ nein, berechnet den belegten Speicherplatz des Arrays

`sizeof(arr)` = Anzahl der Elemente `* sizeof(int)`

→ Anzahl der Elemente = **`sizeof(arr)/sizeof(int)`**

oder **`sizeof(arr)/sizeof(arr[0])`**

Achtung: Anzahl Elemente bedeutet Anzahl der reservierten Elemente.

Arrays und Zeichenketten

Verwendung

Arrays können als Parameter an eine Funktion übergeben werden:

```
void funktion(int arr[]){ ... }
```

ABER: ein Array als Funktionsparameter wird in einen Zeiger umgewandelt und verliert seine Längeninformation und `sizeof(arr)` liefert dann nicht mehr den belegten Speicherplatz des Arrays!

Also: Die Länge als Parameter mit übergeben:

```
void funktion(int arr[], int laenge){ ... }
```

Arrays können nicht als Rückgabetyp von Funktionen verwendet werden.

```
int arr[] funktion(){ ... } -> geht nicht!
```

Arrays und Zeichenketten

Arrays mit variable Länge

Seit C99 können Arrays mit variabler Größe angelegt werden.

```
int y=10;
int arr[y];
```

Es gelten jedoch folgende Einschränkungen:

- Die Größe kann nur einmal bei der Definition des Arrays festgelegt werden.
- Arrays mit variabler Länge können nur innerhalb von Funktionen angelegt werden.

Arrays und Zeichenketten

Arraygrenzen

Die Grenzen von Arrays werden zur Laufzeit nicht überprüft!

```
char arr[10]=“Luftballon“;
char arr1[5]=“Hallo“;

char c = arr[12];
char c = arr[-3];
```

Beide Statements werden vom Compiler akzeptiert und liefern zur Laufzeit völlig undefinierte Werte zurück.
Wenn der Index auf eine ungültige Speicheradresse verweist, dann wird das Programm abgebrochen (Segmentation fault).

Die ungeprüften Arraygrenzen in C werden oft für sog. buffer overflow attacks ausgenützt.

Arrays und Zeichenketten

Übung:

Berechnen sie die Anzahl der Elemente in einem Array
und geben sie jedes Element innerhalb einer Schleife einzeln aus.
Berechnen sie weiters auch den Minimal- und Maximalwert des Arrays.

Verwenden sie dafür folgendes `float` Array:

```
float weights[5]={1.34,4.567,56.55,33.44,782.00};
```

Damit die Validierung korrekt erfolgen kann, bitte folgendes Output Format verwenden:

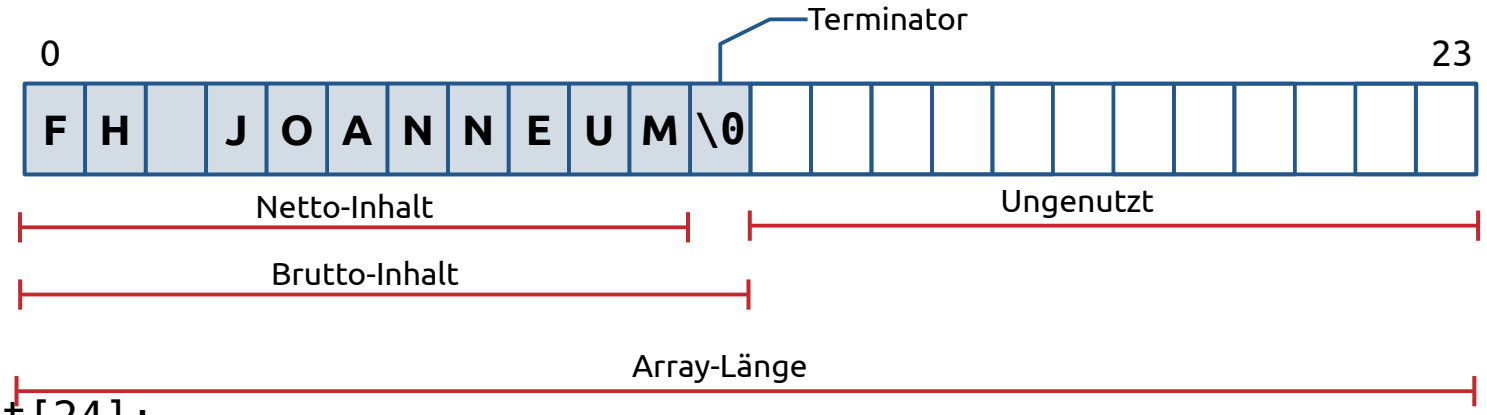
```
Anzahl: 5
1.34
4.57
...
Min: 1.34
Max: 782.00
```

Arrays und Zeichenketten

Zeichenketten/Strings

Strings sind Arrays vom Datentyp `char`, deren Ende mit `\0` markiert wird.

```
char text[24]="FH JOANNEUM";
```




```
char text[24];
text ="FH JOANNEUM"; → das geht nicht, weil Arrays ja konstant sind.
```

Strings können ansonsten wie Arrays verwendet werden.

z.B.: `text[10]='m'` oder `if (text[j] == 'H')`

Zeichenketten

Übung:

Schreiben sie eine Funktion `my_strlen(char string[])`, welche die Länge eines Strings ohne den `sizeof()` Operator zu verwenden, berechnet.
Verwenden sie dazu eine Schleife und testen sie die Funktion mit folgender Zeichenkette:

```
char text[100]="FH JOANNEUM";
```

Geben sie den String und die Länge so aus:

```
String: “FH JOANNEUM“ Länge:11
```

Zeichenketten (2)

Wichtige Stringfunktionen

```
#include <string.h>

strlen(s)          -> Länge von s, der Terminator wird nicht mitgerechnet!
strcpy(s1,s2)      -> kopiert s1 in den String s2
strcat(s1,s2)      -> fügt s2 am Ende von s1 an
strcmp(s1,s2)      -> vergleicht s1 mit s2, liefert:
                       0 ... s1 == s2
                      <0 ... s1 < s2
                      >0 ... s1 > s2
```

Zeichenketten (2)

Umwandlungsfunktionen

```
#include<stdlib.h>

atoi(string)   ... string -> int
atof(string)   ... string -> float
atol(string)   ... string -> long
```

Beispiele

```
int i = atoi(“334“);
float f = atof(“334.14159“);
```

Zeiger

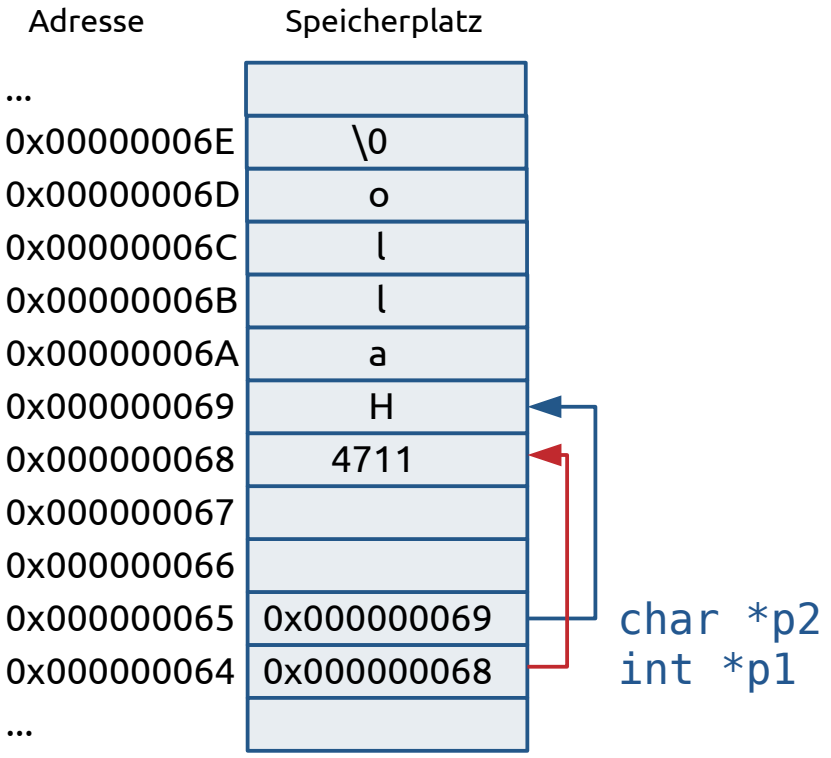
Zeiger

Zeiger sind Variable, die virtuelle Adressen enthalten, also auf eine andere Variable/Speicherzelle zeigen.
oder Allgemein: Ein Zeiger ist ein Objekt, das die Speicheradresse eines anderen Objektes enthält.

Deklaration: `<datentyp> *variable;`

```
int *p1;
char * p2;
```

`p1` ist ein Zeiger auf einen Speicherplatz vom Typ `int`

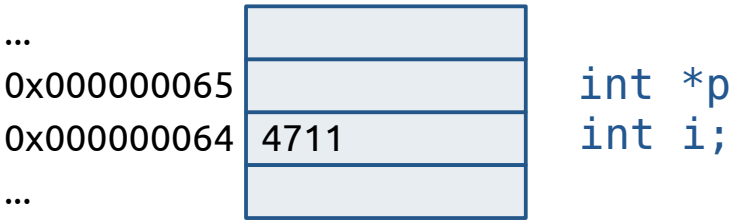


Zeiger

Operatoren

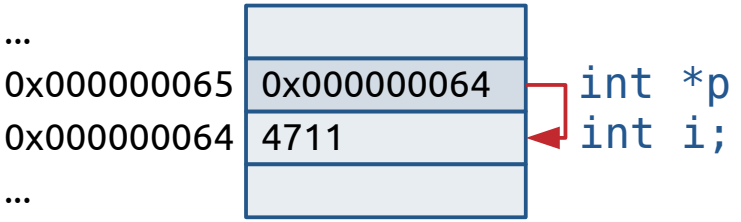
- Der Adressoperator `&` liefert die virtuelle Adresse einer Variable
- Der Dereferenzierungsoperator `*` ermöglicht den Zugriff auf den Inhalt einer Zeigervariable (auch Inhaltsoperator)

```
int i=4711;
int *p;
```



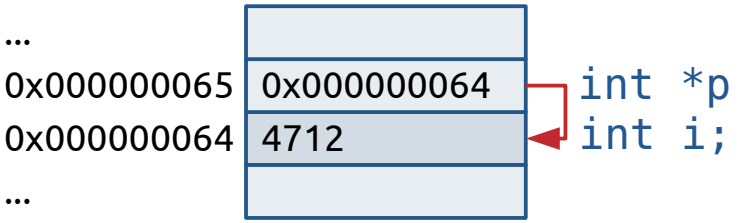

```
p = &i;
```

`p` = Adresse von `i`




```
*p =4712;
```

Setze den Inhalt des Speicherplatzes , auf den `p` zeigt, auf den Wert `4712`.



Zeiger

Zeiger verwenden

Zeiger können wie „normale“ Variable verwendet werden

- Einen Wert zuweisen – aber nur ein Zeiger vom gleichen Typ
- Dereferenzieren um den Wert zu lesen
- Dereferenzieren um den Wert zu schreiben

Beispiel

```
int main() {
    int i = 123;
    int *p;

    p = &i; // Adresse zuweisen

    *p = 345; // Dereferenzierung, Wert zuweisen

    printf("i=%d\n", i);
    printf("*p=%d\n", *p); // Dereferenzierung, Wert lesen
}
```

Zeiger

Zeiger verwenden

Zeiger ermöglichen Funktionsaufrufe mit Call By Reference, indem die Adresse einer Variable als Parameter übergeben wird.

Beispiel

```
void inc(int *zahl) {
    (*zahl)++;
}

int main() {
    int i = 1;
    inc(&i);
    printf("zahl=%d\n", i);
}
```

Zeiger

Übung

Schreiben sie eine Funktion `swap`, die zwei `int` Werte vertauscht.
Die Signatur sollte so aussehen: `void swap(int *a, int *b);`

Testen sie die Funktion für folgende Wertepaare und geben sie die Werte vor und nach der Vertauschung aus.

Wertepaare:

```
1 8
98172232 77
34 31092
```

Ausgabe:

```
a:1 b:8
a:8 b:1
...
```

Zeiger

Zeiger sind Typ-gebunden

```
int *p;
int **pp; // Zeiger auf einen Zeiger der auf einen int Speicherplatz zeigt
int i;
float f;

p=&i;
p=&f;     -> geht nicht, weil int * != float *
pp=&p;
pp=p;     -> geht nicht, weil int ** != int *
```

Casting von Zeigern

```
char c='A';
char *p;
int *i;

p = &c;
i =(int *)p;  -> Ok char * – int *

float f = (float *)p;  -> geht, aber nicht sinnvoll!
```

Zeiger

Typlose Zeiger

`void` Zeiger sind typlos und können von/zu jedem anderem Zeigertypen konvertiert werden. `void` Zeiger selbst können (ohne cast) nicht dereferenziert werden.
Deklaration: `void *p;`

```
int main() {
    int i=3;
    void *p;
    int *p1,*p2;

    p = &i;
    p1 = p;
    p2 = &i;

    printf(“Inhalt von p: %d\n“,*p);
}
```

Richtig:

```
printf("Inhalt von p: %d\n", *(int *)p);
```

Funktioniert das oben angeführte Programm? Warum nicht? Sind `p,p1,p2` gleich?

Zeiger

Ungültige Verwendung von Zeigern

1. Einen nicht initialisierten Zeiger dereferenzieren

```
int *p;
*p = 4711;
```

2. Zeiger zu Variablen die Out-of-Scope sind

```
int *verweis(int i) {
    int j = i;
    return &j;
}

int main() {
    int *p = verweis(100);
    *p = 4711;
}
```

Zeiger

Ungültige Verwendung von Zeigern

3. Einem Zeiger einen ungültigen Wert zuweisen

```
int *p;
p = 4711;
```

Zeiger

Null-Zeiger

Wenn ein Zeiger auf kein Objekt verweist, kann man ihm den Wert `NULL` zuweisen.

`NULL` ist ein Integer Literal vom Wert `0` und ist als Makro definiert:

```
#define NULL 0L
#define NULL (void *)0
```

Man könnte auch schreiben

```
int *p = 0;
```

Es empfiehlt sich aber immer das Makro zu verwenden:

```
int *p = NULL;
```

Viele Funktionen, deren Rückgabewert ein Zeiger ist, geben `NULL` zurück wenn ein Fehler aufgetreten ist und kennzeichnen so den ungültigen Rückgabewert.

Zeiger

const Zeiger

Wenn ein Zeiger als `const` deklariert wird, dann ist der Inhalt, auf den der Zeiger verweist schreibgeschützt, der Zeiger selbst nicht.

Beispiel:

```
void printString(const char *s) {
    while (*s != '\0') {
        putchar(*s);
        ++s;    // -> Zeiger wird verändert
    }
}
```

Zeiger

Zeiger - Array - String

Die Elemente von Arrays und Strings werden sequentiell hintereinander im Speicher abgelegt. Daher können auch Zeiger dazu verwendet werden, um auf Arrays und Strings zuzugreifen.

Der Variablenname eines Arrays ist ein unveränderbarer Zeiger, der auf das erste Elements das Arrays zeigt.

```
int arr[8]={1,2,3,4,5,6,7,8};
int *p;

p = arr ; <=> p = &arr [0];

arr = p → geht nicht!
```

Zeiger

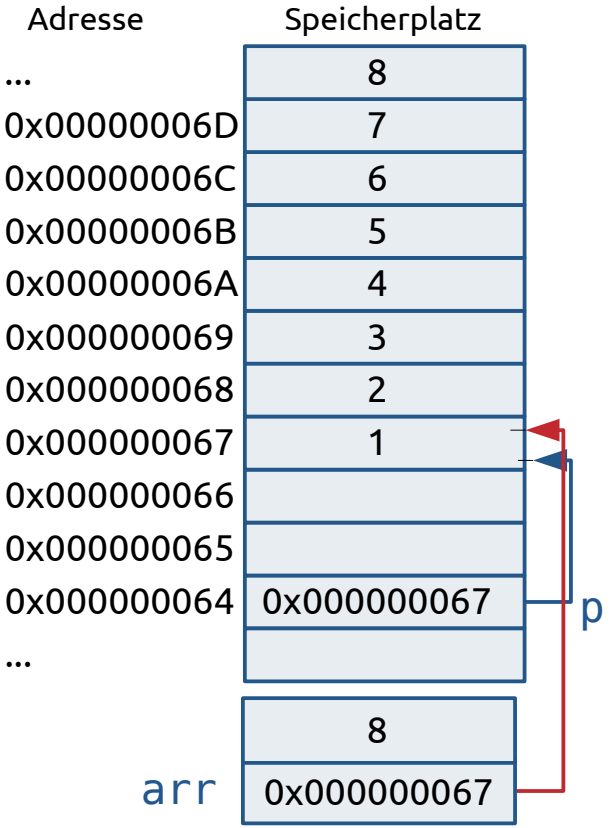
Zeiger - Array - String

Eine Array-Variable ist eine vom Compiler generierte, unveränderliche Beschreibung des den durch das Array belegten Speicherplatzes.
Wird ein Array durch einen Zeiger angesprochen ist z.B.: die Information über den belegten Speicherplatz nicht mehr verfügbar!

```
p = arr ;

sizeof(arr) = 8 * sizeof(int)

sizeof(p) → nicht definiert!
```



Zeiger

Zeiger - Array - String

Arrays als Parameter einer Funktionen werden implizit in einen Zeiger umgewandelt. Die Längeninformation geht verloren!

```
void funktion ( int arr[] ){
        int len = sizeof(arr)/sizeof(int);
        printf("Länge int Funktion: %d\n", len); ->nicht die Array Länge!
}
```

Länge des Arrays als Parameter mitgeben:

```
void funktion ( int arr[], int arrLen ){

    for ( i=0; i<arrLen;i++)
        arr[i]+=1;
}
```

Zeiger

Zeiger - Array - String

Jeder Zeiger kann wie ein Array verwendet werden:

```
char st[11]=“Hallo Welt!“;
char *p;

p=st;
for(int i=0; i<11; i++)
    printf(“Zeichen: %c\n“,p[i]);
```

Oder mit Zeigerarithmetik:

```
for(p=st; *p!='\0'; p++)
      printf("Zeichen: %c\n",*p);
```

Achtung: Die Länge von Speicherbereichen auf die ein Zeiger zeigt, muss immer selbst verwaltet werden.

Zeiger

Zeigerarithmetik

Zeiger sind Adressen.
Zeigerarithmetik erlaubt es mit Adressen zu rechnen:

Zeiger

```
char *p;
```

Zeiger auf nächstes Element setzen: `++p` oder `p++`
Zeiger auf vorheriges Element setzen: `--p` oder `p--`
Zeiger n Elemente nach vor setzen: `p=p+n` oder `p+=n`
Zeiger n Elemente nach zurück setzen: `p=p-n` oder `p-=n`

Beispiele:

```
c = *p++ -> c=*p; p=p+1
*str + i != *(str+i)

p ist int * : p+2 -> p + (2 * sizeof(int))
s ist char * : s+2 -> s +2  weil sizeof(char) == 1
```

Zeiger

Zeigerarithmetik - Array

Zeiger-Variante	Array-Variante
`*p` `*array`	`p[0]` `array[0]`
`*(p+n)` `*(array+n)`	`p[n]` `array[n]`
`*++p`	`p[n+1]`
`*p++`	`p[n]`

Zeiger

Zeigerarithmetik

`strlen()` Funktion mit Zeigerarithmetik

```
int my_strlen(char *string) {
    char *s;

    for (s = string; *s; ++s);
    return(s - string);
}
```

Zeiger

Zeiger auf Zeiger und Arrays

```
char *laender[] = {"\xd6sterreich", "Deutschland", "Schweiz",
                   "Tschechien", "Slowenien"};
char **pp = laender;

printf("%c\n", **laender);          Ö
printf("%c\n", *laender[0]);        Ö
printf("%c\n", *laender[4]);        S
printf("%c\n", *(laender[0]+2));    t
printf("%c\n", *(laender[0]+3));    e
printf("%s\n", laender[0]);         Österreich
printf("%s\n", *laender);           Österreich
printf("%s\n", (laender[0]+1));     sterreich
printf("%s\n", (laender[0]+2));     terreich
printf("%c\n", **pp);               Ö
printf("%c\n", *(*pp+2));           t
printf("%c\n", **(pp+2));           S
printf("%c\n", *(*(pp+2)+3));       w
printf("%s\n", *(pp+2)+3);          weiz
printf("%s\n", *pp+2);              terreich
```

Frage: was liefern diese Ausdrücke und warum?

Zeiger

Zeiger auf Funktionen (1)

`f` ist ein Zeiger auf eine Funktion, die zwei `int` Werte als Parameter hat und `int` zurückliefert.

```
int (*f)(int, int);
```

`f` ist ein Zeiger auf eine Funktion, die zwei `int` Werte als Parameter hat und `int *` zurückliefert.

```
int *((*f)(int, int));
```

Zuweisung:

```
f = add; // add ist eine Funktion die vorher definiert wurde zB
         // int add (int a, int b) { return a+ b; }
```

Aufruf:

```
int res = f(2,3);
```

Zeiger

Zeiger auf Funktionen (2)

```
#include <stdio.h>
#include <string.h>

int print(char *t) {
  printf("%s\n",t);
  return strlen(t);
}

int debug(char *t) {
  printf("debug: %s\n",t);
  return strlen(t);
}

int doit(int (*f)(char *), char *text) {
  printf("chars printed: %d\n",f(text));
}

int main() {
  int (*f)(char *);
  f = print;
  doit(f,"Hello World");

  f = debug;
  doit(f, "Fehler 735");
}
```

Zeiger

Übung

Die Funktion `qsort()` ist in der Standardlibrary vorhanden, um Arrays oder Speicherbereiche zu sortieren.
Schlagen sie die Definition von `qsort()` in der entsprechenden man-Page nach.

Schreiben sie ein Programm, das ein `int`-Array mit 10 Werten sortiert.
Verwenden sie dieses Array:

```
int arr [] ={10,286,888,104,22,399,58,38,1,478,6,99};
```

Verwenden sie die Funktion `qsort()` um das Array zuerst aufsteigend und dann absteigend zu sortieren. Definieren sie dazu zwei Vergleichsfunktionen `asc()` und `desc()` die dem Funktionsprototypen oben entsprechen und jeweils eine aufsteigende oder absteigende Sortierreihenfolge implementieren.
Geben sie die sortierten Arrays auf `stdout` aus.
Ausgabeformat:

```
Aufsteigend:
...
Absteigend:
...
```