



Systemprogrammierung

Angewandte Informatik, FH Flensburg

Prof. Dr. Tepper

Inhalt der Veranstaltung:

- Einführung in C und C++
- Nebenläufigkeit (Prozesse und Threads)
- Scheduling, Realzeitverhalten, Zeitmanagement
- Synchronisation (Mutex, Semaphor, Monitor, Signal, ...)
- Kommunikation (Queues, Pipes, Shared Memory, ...)

Systemprogrammierung:

- **system**abhängige Programmierung
- Verwendung von **System**aufrufen
- i.A. nicht portabel
- **system** calls fast immer aus C/C++, da aktuelle OSs in C geschrieben sind
- insbesondere erforderlich für nebenläufige Anwendungen:
(Serverapplikationen, Simulatoren, Prozesssteuerungen, Realzeitsysteme, ...)

Einführung in C/C++:

- C: sehr alte Sprache (Ende 60er),
Unix ist in C geschrieben
- ANSI C 1998
- C++: OOP-Erweiterung, ab 1979
 - Klassen
 - Mehrfachvererbung
 - Exceptions
 - Templates
 - String-Klasse
 - Überladung
- C++11: autom. Typen, threading
- aktueller Standard: C++14 (2015)
- in Planung: C++17, C++20

C++ ist
abwärtskompatibel zu C

C - Standardtypen:

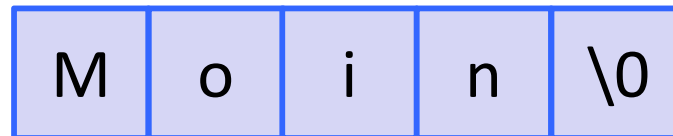
- Ganzzahltypen:
`char, int, short, long`
- alle Ganzzahltypen auch als `unsigned`, z.B.:
`unsigned short`
- Realzahltypen:
`float, double, long double`
- kein Typ für `bool`
(`int` mit `≠ 0 -> true, 0 -> false`)
- kein Typ für Zeichenketten (`char`-arrays)
- die Wertebereiche sind nicht in der Sprache definiert (siehe `limits.h` bzw. `sizeof()` verwenden)

C - Zeichenketten:

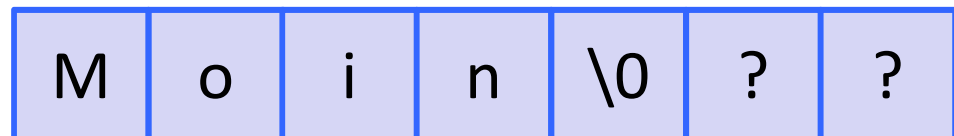
- Zeichenketten (strings) sind **char-arrays** mit '**\0**' als Endmarke (**NTBS**: null terminated byte string)
- diverse string-Funktionen (printf, strcpy, ...) erwarten als letztes Zeichen ein '**\0**'

```
char str01[] = "Moin";
```

```
char str02[] = {'M', 'o', 'i', 'n', '\0'};
```



```
char str03[7] = "Moin";
```



C – Input/Output:

- diverse I/O-Funktionen deklariert in `<stdio.h>`
(siehe z.B. <http://en.cppreference.com/w/c/io>)
- hier nur beispielhaft `printf` und `scanf`:
Formatbeschreiber für ganzzahlige Typen:
 - `c` char
 - `d` int
 - `u` unsigned int
 - `o` unsigned int oktale Ausgabe
 - `x` unsigned int hex Ausgabe, Kleinbuchstaben
 - `X` unsigned int hex Ausgabe, Großbuchstaben
 - `p` pointer/Adresse Format systemabhängig

für long-Typen zusätzlich mit Vorsatz `l`, z.B.: `ld`

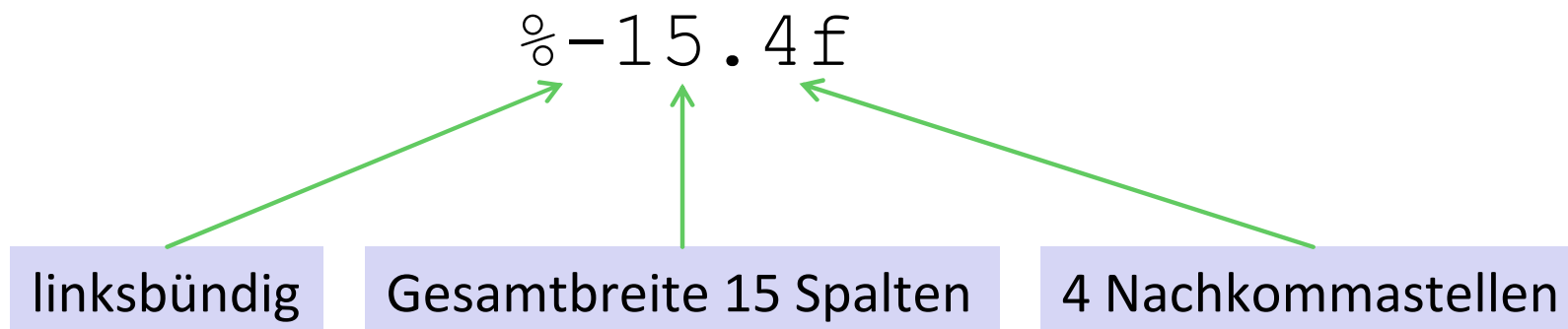
Formatbeschreiber für float-Typen:

- f float oder double
- e oder E float oder double wiss. Schreibweise
- g oder G float oder double wählt f- oder e-Format

Formatbeschreiber für strings:

- s array of chars Ausgabe bis \0

Die Formatbeschreiber können zusätzlich Angaben zur Spaltenbreite, Anzahl der Nachkommastellen und links-/rechtsbündig enthalten, z.B.:



Beispiel zu printf:

```
#include <stdio.h>    // für printf
#include <stdlib.h>    // für EXIT_SUCCESS

int main(void)
{
    int i = 42;
    double x = 42.42;
    char s[] = "Fritzzzzzzz";

    printf("i: %d, %-8o, %4x\n", i, i, i);
    printf("x: %-8.3f, %7.1e, %g\n", x, x, x);
    printf("Moin %s, Moin %10.5s", s, s);

    return EXIT_SUCCESS; // EXIT_SUCCESS = 0
}
```

Ausgabe:

```
i: 42, 52          ,    2a
x: 42.420   ,  4.2e+01, 42.42
Moin Fritzzzzzzz, Moin          Fritz
```

Beispiel zu scanf:

```
char c;  
int i;  
float x;  
double d;  
char str[100];
```

```
puts("char eingeben:");
```

```
scanf("%c", &c);
```

```
puts("int, float, double eingeben:");
```

```
scanf("%d %f %lf", &i, &x, &d);
```

```
puts("string eingeben, max. 99 Zeichen:");
```

```
scanf("%s", str);
```

```
puts("Es wurde eingegeben:");
```

```
printf("%c %d %f %lf %s", c, i, x, d, str);
```

Ein-/Ausgaben:

char eingeben:

a

int, float, double eingeben:

1 2.22 3.33

string eingeben, max. 99 Zeichen:

abcdefg

Es wurde eingegeben:

a 1 2.220000 3.330000 abcdefg

Adressen

kein Adressoperator, str ist ein Zeiger

Zeiger:

Eine Zeigervariable enthält die Adresse einer Variablen, eines Objekts oder einer Funktion

➡ "ein Zeiger zeigt auf ein Speicherobjekt"

Verwendung:

- als Funktionsparameter („Rückgabeparameter“)
- Zugriff auf dynamische Datenstrukturen
- Zugriff auf Elemente von Feldern oder Strukturen
- Realisierung polymorpher Aufrufe (C++)

Normalerweise wird mit Bezeichnern auf Speicherobjekte zugegriffen:

```
float x;
```

```
x = 1.234f;
```

```
float y;
```

```
int a, b;
```

```
a = 42;
```

Adresse

Bezeichner

0815

1.234

x

4711

???

y

A042

42

a

F12B

???

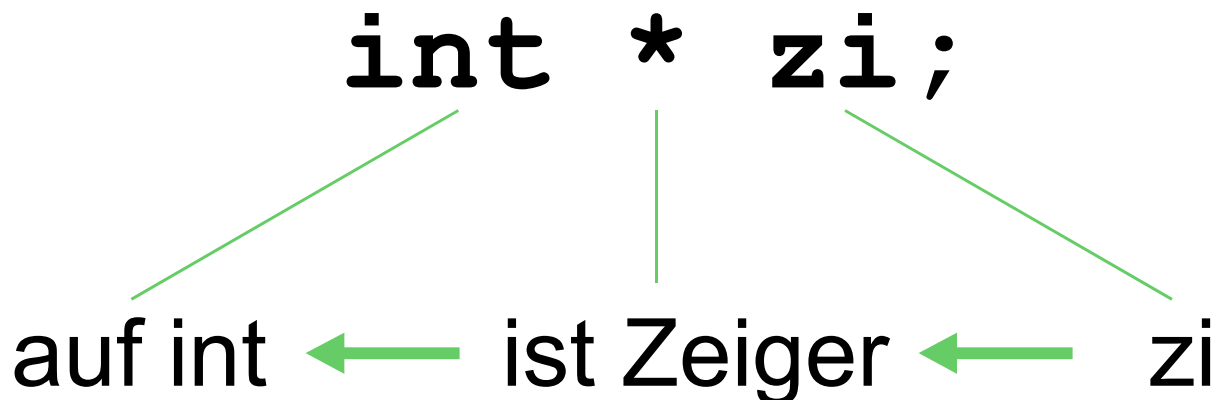
b

Zeiger bieten eine weitere Zugriffsmöglichkeit (auch, wenn keine Bezeichner verfügbar sind).

Deklaration:

Zeiger sind **typisiert**. Bei der Deklaration wird festgelegt, auf welche Art von Objekten ein Zeiger verweisen kann:

int * zi;



auf int ← ist Zeiger ← zi

zi kann nur
auf int-Objekte
zeigen und ist
nicht initialisiert!

Adressoperator:

Der Operator **&** liefert die Adresse eines Datenobjekts oder einer Funktion. Diese Adressen können Zeigervariablen zugewiesen werden (sofern typkompatibel):

```
int * zi;
```

```
zi = &a;
```

```
zi = &x;
```

Zeiger auf float
(s.S.12)

1704

A042

zi

A042

42

a

Dereferenzierungsoperator (indirection operator):

angewandt auf einen Zeiger beschreibt der Operator ***** den Wert des Objekts, auf das der Zeiger zeigt (auch als Lvalue zu verwenden):



b = *zi;



***zi = 2*b;**



Hinweise:

- die Verwendung (Dereferenzierung) nicht initialisierter Zeiger ist unbedingt zu vermeiden

→ *liefert Zufallswert,
Programmabbruch, wenn als Lvalue verwendet*

- der konstante Zeigerwert **NULL** ist kompatibel zu allen Zeigertypen.

Sein Wert ist 'Adresse 0' , was i.A. interpretiert wird als:

→ *Zeiger zeigt auf nichts*





- Typecasting ist möglich, Kompatibilität zum Typ **void*** ist immer gegeben:

→ `float* zf = (float*) zi;`
`void* zv = zi;`

- Folgende Schreibweisen der Deklaration sind identisch:

int* zi; int * zi; int *zi;

Mögliche Interpretationen:

int*	zi;	int	*zi;
			
Typ:	Variable zi	Typ:	'dereferenzierte
'Zeiger auf int'		'int'	Variable' zi

Achtung: mehrere Variablen in einer Deklaration:

int* zj, zk;

→ nur **zj** ist *Zeiger auf int*, **zk** ist ein *int*

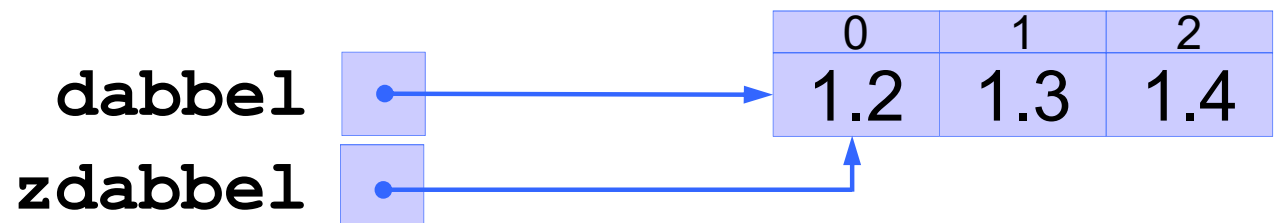
Zeiger und arrays:

- der Feldbezeichner (**dabbel**) ohne Subscript-Operator [] ist ein Zeiger auf das erste Feldelement
- der Wert des Feldbezeichners ist konstant (kein Lvalue)
- der sizeof() Operator liefert die Länge (in byte) des gesamten Feldes, nicht die Länge des Zeigers

```
double dabbel[] = {1.2, 1.3, 1.4};  
double* zdabbel = dabbel;
```

```
cout << *zdabbel << endl;  
cout << sizeof(dabbel) << endl;  
cout << sizeof(zdabbel) << endl;
```

Ausgabe:
1.2
24
8



Zeigeroperationen:

- increment und decrement: der Wert wird entsprechend der Größe des referenzierten Objekts geändert, z.B.:

zdabbel++;  zeigt jetzt auf **dabbel[1]**

- Addition:

- Zeiger plus / minus Integer ergibt einen Zeigerwert
- Zeiger plus Zeiger ist nicht erlaubt

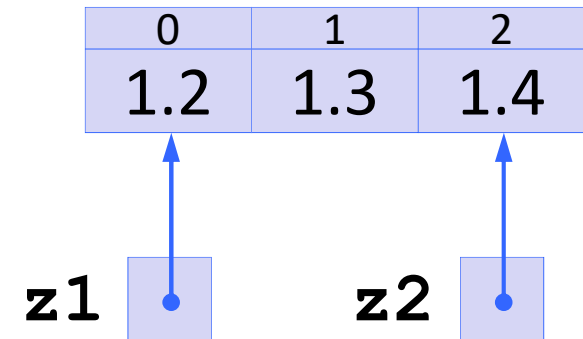
zdabbel = zdabbel + 2;

 zeigt jetzt hinter das array!

- Subtraktion:

- Zeiger minus Zeiger ist erlaubt, falls beide Zeiger auf Elemente des selben arrays zeigen. Das Ergebnis ist kein Zeiger, sondern die ganzzahlige (Index-)Differenz der Feldelemente!

```
double* z1 = dabbel;  
double* z2 = &dabbel[2];  
  
cout << z2 - z1 << endl;
```



→ das Ergebnis ist 2

– Vergleiche:

- die Operatoren `==` und `!=` können auf alle Zeiger angewendet werden
- verweisen zwei Zeiger auf dasselbe Feld, sind zusätzlich die Operatoren `<`, `>`, `<=` und `>=` definiert

– Subscript Operator `[]`:

- sei `exp1` ein Zeigerausdruck und `exp2` ein ganzzahliger Ausdruck, dann gilt:

! $exp1[exp2] \equiv exp2[exp1] \equiv *(exp1 + exp2)$



der erste Index muss immer 0 sein

Zeigerdeklarationen mit const:

- konstanter Zeiger (***const** modifier):
der *Wert des Zeigers* kann nicht geändert werden,
Initialisierung bei der Deklaration erforderlich:

```
char c = 'A';
```

```
char *const p1 = &c;
```

- Zeiger auf konstantes Objekt (**const** modifier):
der *Wert des Objekts*, auf das der Zeiger zeigt kann nicht *mit Hilfe des Zeigers* geändert werden:

```
const char* p2 = &c;
```

oder →

```
char const* p2 = &c;
```

~~```
*p2 = 'B';
```~~

```
c = 'B';
```

← nicht möglich

← o.k.

- konstanter Zeiger auf konstantes Objekt:

```
const char *const p3 = &c;
```

oder  `char const *const p4 = &c;`

```
p3 = p4;
```

```
*p3 = 'B';
```

```
c = 'B';
```

 nicht möglich

 o.k.

- Zeiger und const Funktionsargumente:

```
void abc(int* zi, const float* zf);
```

- in der Funktion können **zi** und **zf** nicht geändert werden, da sie by-value übergeben werden
- in der Funktion kann **\*zf** nicht als Lvalue verwendet werden, da **zf** als *Zeiger auf const Objekt* deklariert ist

## Zeiger auf Strukturen:

(in C sind `structs`: „Klassen ohne Methoden“)

Existiert ein *Bezeichner* für eine Struktur, so wird mit dem Punktoperator `.` auf Strukturelemente zugegriffen:

```
struct Str
{
 double x;
 char c;
}; // <- Semikolon!!
```

```
struct Str st;
```

in C ist dies der Typ!!!

```
st.x = 1.23f;
```

Existiert ein *Zeiger* auf eine Struktur, so wird mit dem Pfeiloperator `->` auf Strukturelemente zugegriffen:

```
struct Str
{
 double x;
 char c;
};
```

```
struct Str st;
```

```
struct Str* pst = &st;
```

```
pst->c = 'a';
```

## Dynamischer Speicher:

- der sog. **heap** (Halde, free store, Freispeicher) kann zur Laufzeit mit **malloc**, **calloc** und **realloc** belegt werden (Allokation, allokieren, allozieren, *malloc = memory allocation*)
- Objekte im heap haben keinen Bezeichner, sie sind nur über Zeiger ansprechbar
- wird der Speicher nicht mehr benötigt, muss er ausdrücklich mit **free** freigegeben werden (Deallokation, deallokieren, dislozieren)

### Hinweis:

In C++ werden diese Funktionen durch die Operatoren **new** und **delete** ersetzt.



```
void* malloc(unsigned m);
```

- reserviert **m-Bytes** im heap (zusammenhängend)
- liefert einen Zeiger vom Typ **void\*** auf den Anfang des reservierten Bereichs
- **void\*** ist ein „universeller“ Zeigertyp, er ist zu allen anderen Zeigertypen kompatibel
- liefert **NULL** (Wert/Adresse 0), falls der Speicher erschöpft ist
- der Speicher wird **nicht initialisiert**

*calloc = core allocation*

```
void* calloc(unsigned n, unsigned m);
```

- reserviert **n-mal** einen Bereich von **m-Bytes**
- der reservierte Bereich wird **mit 0 initialisiert**

*realloc = reallocation*

```
void* realloc(void* z, unsigned m);
```

- **verändert die Größe** eines mit **malloc** oder **calloc** reservierten Bereichs
- **z** ist Zeiger auf den „alten“ Speicherbereich
- **m** ist die neu angeforderte Größe in Bytes
- return-Wert ist die evtl. neue Adresse

```
void free(void* z);
```

- **deallokiert** einen mit **malloc**, **calloc** oder **realloc** reservierten Bereich
- **z** ist Zeiger auf den Bereich
- **keine Fehlermeldung** bei falscher Verwendung

static oder auto (stack)

heap

```
int* zi;
```

zi



```
char* zc;
```

zc



zi



int

```
zi = malloc(sizeof(int));
```

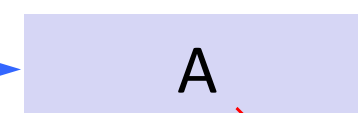
```
*zi = 42;
```

zi



42

zc



A

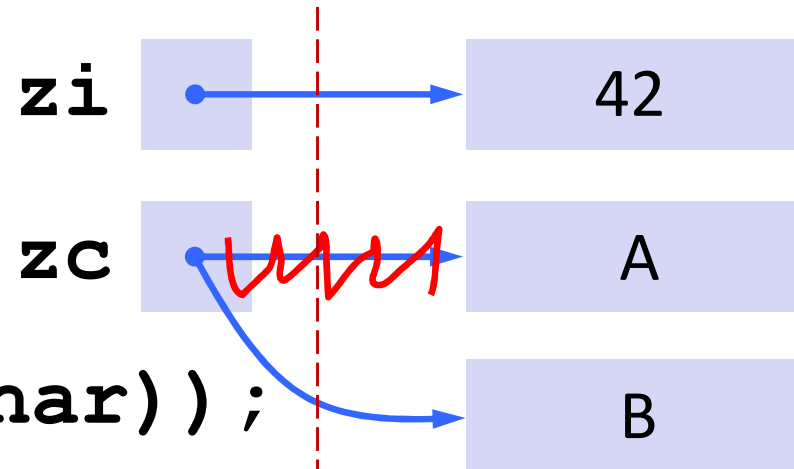
char

```
zc = malloc(sizeof(char));
```

```
*zc = 'A';
```

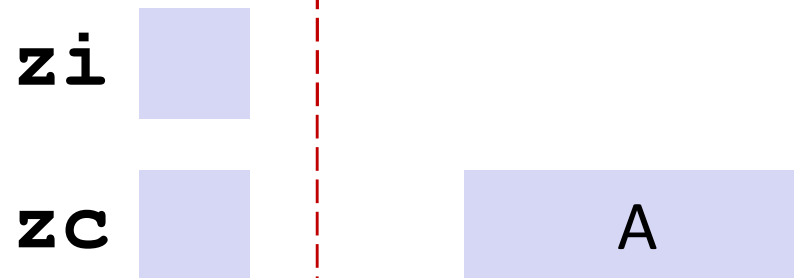
Auf den Speicher mit Inhalt 'A'  
kann nicht mehr zugegriffen werden!

```
zc = malloc(sizeof(char));
*zc = 'B';
```



```
free(zi);
```

```
free(zc);
```



Es wird NICHT zi und zc „entfernt“!

```
zi = NULL;
zc = NULL; } „Vorsichtsmaßnahme“
```