

# Systemprogrammierung

## Teil 2: ANSI-C Daten Literale, Variablen, Typen

### ANSI-C Literale: Ganze Zahlen

Schreibweisen für ganze Zahlen (*Integers*):

- **dezimal**      1      23      456      7890
- **oktal**      01      023      045670
- **hexadezimal**      0x1      0x23      0x456      0x789a      0xbcdef0

Typ des Literals ist je nach Schreibweise der jeweils kleinste passende Typ:

- **dezimal**      `int, long int, unsigned long int`
- **oktal oder hexadezimal**      `int, unsigned int, long int, unsigned long int`
- mit **Suffix L**      `long int, unsigned long int`  
z.B. `12345L`
- mit **Suffix U**      `unsigned int, unsigned long int`  
z.B. `12345U`

*Nicht vergessen: der Compiler wandelt alle Schreibweisen in Binärzahlen!*

# Beispiel-Programm Zahlen-Literale

- Quellcode

```
#include <stdio.h>

int main(void)
{
    printf("%x\n", 12);
    printf("%d\n", 012);
    printf("%o\n", 0x12);
    return 0;
}
```

Konsolenausgabe  
des Programms:

c  
10  
22

*%x ist hexadezimaless Format  
%d ist dezimales Format  
%o ist oktales Format  
\n ist Zeilenwechsel*

## ANSI-C Literale: Gleitkomma-Zahlen

Schreibweisen für Gleitkomma-Zahlen (*Floating Point Numbers*):

- nur **dezimal** 1. .23 0.456 78.9 .789e2 789e-1  
*.789e2 steht für 0,789 · 10<sup>2</sup>*

Typ des Literals abhängig vom Suffix:

- ohne Suffix double
- mit **Suffix L** long double  
z.B. 1.2345L
- mit **Suffix F** float  
z.B. 1.2345F

*Nicht vergessen: Gleitkomma-Zahlen sind ungenau!*

*Auch bei Gleitkomma-Literalen wandelt der Compiler alle Schreibweisen in ein Binärformat (je nach Zielhardware z.B. IEEE 754)*

# Beispiel-Programm Gleitkomma-Literale

- Quellcode:

```
#include <stdio.h>

int main(void)
{
    printf("%g\n", (1e-30 + 1e30) - 1e30);
    printf("%g\n", 1e-30 + (1e30 - 1e30));
    printf("%f\n", 12.3456789);
    printf("%f\n", 1234567.89);
    printf("%e\n", 12.3456789);
    printf("%e\n", 1234567.89);
    return 0;
}
```

Konsolenausgabe  
des Programms:

```
0
1e-30
12,345679
1234567,890000
1.234568e+01
1.234568e+06
```

*Ausgabe bei %f und %e  
standardmäßig mit  
6 Nachkommastellen*

*%g ist Fest- oder Gleitkommaformat nach Bedarf  
%f ist Festkommaformat  
%e ist Gleitkommaformat*

## ANSI-C Literale: Einzelzeichen (1)

Schreibweisen für Einzelzeichen (*Characters*):

- in **Einfach-Hochkommas**

'a' 'A' '1' '.' ''	Buchstaben, Ziffern, Satzzeichen, Leerstelle, ...
'\0'	das <b>NULL</b> -Zeichen (Code-Nummer 0)
'\ooo'	Codenummer oktal (1 bis 3 Oktalziffern <i>o</i> )
'\xhh'	Codenummer hexadezimal (mindestens eine Hex-Ziffer <i>h</i> )
'\c'	Ersatzdarstellung für Steuerzeichen ( <i>c</i> ist a, b, f, n, r oder t)
'\''	das Einfach-Hochkomma
'\"'	das Doppel-Hochkomma
'\\'	der Backslash

*Der Compiler wandelt alle Schreibweisen  
in binäre Zeichencode-Nummern  
(je nach Plattform z.B. ASCII).*

Typ des Literals ist **char**

## ANSI-C Literale: Einzelzeichen (2)

---

- Bedeutung der Ersatzdarstellungen für Steuerzeichen:

'\a'	Alarm
'\b'	Rückschritt ( <i>Backspace</i> )
'\f'	Seitenvorschub ( <i>Formfeed</i> )
'\n'	Zeilenende ( <i>Newline</i> )
'\r'	Wagenrücklauf ( <i>Carriage-Return</i> )
'\t'	Horizontal-Tabulator
'\v'	Vertikal-Tabulator

*Nicht vergessen: der Compiler wandelt alle Schreibweisen in binäre Zeichencode-Nummern (je nach Plattform z.B. ASCII)*

## ANSI-C Literale: Zeichenketten

---

Schreibweise für Zeichenketten (**Strings**):

- in **Doppel-Hochkommas**

"Hallo"

" " *leerer String*

*zwischen den Doppelhochkommas sind alle Schreibweisen für Einzelzeichen erlaubt, wobei die Einfach-Hochkommas entfallen, z.B. "Hallo\n"*

- nur durch Zwischenraum (*Whitespace*) getrennte Zeichenketten fasst der Compiler zu einer Zeichenkette zusammen:

"Hal" "lo" *das gleiche wie "Hallo"*

Typ des Literals abhängig vom Präfix:

- ohne Präfix `char*`
- mit Präfix `L` `wchar_t*`  
z.B. `L"Hallo"`

# Beispiel-Programm Zeichen-Literale

- Quellcode:

```
#include <stdio.h>

int main(void)
{
    printf("%s\n", "Hallo");
    printf("%s\n", "Hal" "lo");
    printf("Hallo\n");
    printf("%c%c%c%c%c\n", 'H', 'a', 'l', 'l', 'o');
    return 0;
}
```

*%s ist Zeichenkettenausgabe  
%c ist Einzelzeichenausgabe*

Konsolenausgabe  
des Programms:

```
Hallo
Hallo
Hallo
Hallo
```

## ANSI-C Literale: Symbolische Konstanten

Der ANSI-C-Präprozessor erlaubt es, symbolische Namen für Literale zu vergeben.

- Definition einer symbolischen Konstanten:

```
#define Name Literal
```

*Präprozessor-Anweisungen sind Zeilen, die mit # beginnen*

*der Name sollte nur aus Großbuchstaben bestehen  
(und eventuell Ziffern und Unterstriche, allerdings nicht als erstes Zeichen)*

- Benutzung einer symbolischen Konstanten:

nach der Definition kann der Name anstelle des Literals geschrieben werden

*der Name wird beim Übersetzen vom Präprozessor durch das Literal ersetzt*

- Beispiel:

```
#define PI 3.14159265358979323846
```

## ANSI-C Literale: Vergleich mit Java

---

Schreibweise der Literale ist in ANSI-C und Java weitgehend gleich

Wichtige Unterschiede bei ANSI-C:

- es gibt ganze Zahlen ohne Vorzeichen
- der Zeichensatz ist nicht Unicode
- Verkettung von String-Literalen ohne +
- keine Literale `true` und `false`
- symbolische Namen für Literale

## ANSI-C Literale: Empfehlungen

---

### Zahlen-Literale:

- echte Zahlen immer dezimal schreiben
- Bitmuster immer oktal oder noch besser hexadezimal schreiben

### Zeichen-Literale:

- die oktale und hexadezimale Angabe von Code-Nummern (ausser `'\0'`) vermeiden  
*Es drohen sonst Überraschungen auf Rechnern mit verschiedenen Zeichencodes.*

### symbolische Konstanten:

- Literale in der Regel nur zum Initialisieren von Variablen verwenden, ansonsten symbolische Konstanten bevorzugen  
*Kommt ein bestimmtes Literal an mehreren Stellen vor, ist nicht erkennbar, ob zwischen diesen Stellen ein logischer Zusammenhang besteht*

# ANSI-C Variablen: Eigenschaften

---

**Variablen** dienen dazu, Werte im Hauptspeicher abzulegen und anzusprechen.

- eine Variable hat einen **Namen**:  
*Besteht aus Buchstaben, Ziffern und Unterstrichen.  
Darf nicht mit einer Ziffer beginnen und darf kein ANSI-C Schlüsselwort sein.*
- eine Variable hat einen **Typ**:  
*Legt fest, welche Art von Werten die Variable aufnehmen kann (z.B. nur ganze Zahlen).  
Legt fest, welche Operationen erlaubt sind (z.B. Addition usw.).*
- eine Variable hat einen **Wert**:  
*Steht in binärer Zahlendarstellung im Hauptspeicher.*
- eine Variable hat eine **Adresse**:  
*Die Anfangsadresse des Werts im Hauptspeicher.*
- eine Variable hat einen **Platzbedarf**:  
*Anzahl Bytes, die der Wert im Hauptspeicher belegt. Hängt vom Typ ab.*

# ANSI-C Variablen: Syntax

---

- **Variablen-Definition** legt **Typ** und **Name** fest:  
Erst nach ihrer Definition ist eine Variable benutzbar **Typ Name;**  
*Definition lokaler Variablen nur am Anfang eines { }-Blocks*
- **Wert**:  
definierter Anfangswert nur mit Initialisierung **Typ Name = Wert;**  
Wertänderung per Zuweisung **Name = Wert;**  
bei Konstanten Initialisierungspflicht und keine Zuweisung **const Typ Name = Wert;**
- **Adresse**:  
der Adressoperator liefert die Adresse einer Variablen **&Name**  
*i.d.R. müssen Variablen eine durch **sizeof** ( Typ ) teilbare Adresse haben ( Alignment )*
- **Platzbedarf**:  
der sizeof-Operator liefert den Platzbedarf einer Variablen **sizeof Name**  
bzw. den Platzbedarf eines Typs in Anzahl Byte. **sizeof ( Typ )**

# ANSI-C Datentypen: Übersicht

---

## Grundtypen (elementare Datentypen)

- Arithmetische Typen
  - Ganzzahlige Typen: `char`, `int`, ...
  - Gleitkommatypen: `float`, `double`, ...
- Anonymer Typ: `void`

*Einen logischen Typ (boolean) gibt es nicht!*

## Abgeleitete Typen

- Zeiger: `*`
- Felder: `[ ]`

## Benutzerdefinierte Typen

- Aufzählungen: `enum`
- Strukturen: `struct`, `union`

# ANSI-C Grundtypen: int

---

- Variablen-Definition:

```
int zahl = 123;  
short int zahl = 123;  
long int zahl = 123L;  
  
unsigned int bytefolge = 0xffffffffU;  
unsigned short int bytefolge = 0xffffU;  
unsigned long int bytefolge = 0xffffffffUL;
```

*Kurzschreibweise: hinter **short**, **long** und **unsigned** kann **int** weggelassen werden*
- Wert: ganze Zahl mit Vorzeichen  
mit Zusatz `unsigned` Bitmuster (ganze Zahl ohne Vorzeichen).
- Platzbedarf je nach Rechner bzw. Compiler:

```
sizeof (short) ≤ sizeof (int) ≤ sizeof (long)
```

typisch: `short` 2 Byte, `int` und `long` 4 Byte (ILP32-Rechner)  
`short` 2 Byte, `int` 4 Byte, `long` 8 Byte (LP64-Rechner)

*Zusatz **unsigned** ist ohne Einfluss auf den Platzbedarf*



# Beispiel-Programm int-Variablen

- Quellcode:

```
#include <stdio.h>

int main(void)
{
    int n = 0;
    int m = 1;

    /* print variable values */
    printf("n = %d\n", n);
    printf("m = %d\n", m);

    /* print variable addresses */
    printf("&n = %p\n", (void*) &n);
    printf("&m = %p\n", (void*) &m);

    /* print type and variable sizes */
    printf("sizeof (int) = %lu\n", (unsigned long) sizeof (int));
    printf("sizeof n = %lu\n", (unsigned long) sizeof n);

    return 0;
}
```

## Konsolenausgabe des Programms:

```
n = 0
m = 1
&n = 0x22efc4
&m = 0x22efc0
sizeof (int) = 4
sizeof n = 4
```

## ANSI-C Grundtypen: float und double

- Variablen-Definition:  
`float zahl = 3.14F;`  
`double zahl = 3.14;`  
`long double zahl = 3.14L;`
- Wert:  
bei `float` einfach genaue Gleitkommazahlen (*single precision*)  
bei `double` doppelt genaue Gleitkommazahlen (*double precision*)  
bei `long double` erweitert genaue Gleitkommazahlen (*extended precision*)
- Platzbedarf je nach Rechner bzw. Compiler:  
`sizeof (float) ≤ sizeof (double) ≤ sizeof (long double)`  
typisch: 4 Byte für `float`  
8 Byte für `double`  
16 Byte für `long double`

## ANSI-C Grundtypen: char

---

- Variablen-Definition: `char zeichen = 'a';`  
`signed char byte = -1;`  
`unsigned char byte = 0xff;`
- Wert:
  - bei `char` Einzelzeichen im Standard-Zeichensatz (normalerweise ASCII)
  - bei `signed char` ganze Zahl mit Vorzeichen
  - bei `unsigned char` Bitmuster (ganze Zahlen ohne Vorzeichen)
- Platzbedarf ist 1 Byte:
  - `1 == sizeof (char)`
  - `1 == sizeof (signed char) == sizeof (unsigned char)`

## ANSI-C Grundtypen: void

---

- Variablen-Definition:
  - entfällt — es gibt keine Variablen vom Typ `void`
- Wert:
  - entfällt
- Platzbedarf:
  - entfällt — `sizeof`-Operator auf `void` nicht anwendbar

Verwendung des Typs `void`:

- zur Definition abgeleiteter Typen
  - `void*` *Zeiger auf "irgendwas" (allgemeinster Zeigertyp)*
- bei Funktions-Definitionen
  - `void f(void);` *Funktion ohne Rückgabewert und ohne Parameter*

# ANSI-C Grundtypen: Vergleich mit Java

---

Grundtypen und Schreibweise der Variablen-Definition sind in ANSI-C und Java sehr ähnlich

Wichtige Unterschiede bei ANSI-C:

- kein Grundtyp `boolean`
- es gibt ganze Zahlen ohne Vorzeichen
- Platzbedarf und Speicheradresse von Variablen lassen sich mit Operatoren `sizeof` bzw. `&` ermitteln
- Platzbedarf und damit Wertebereiche der Grundtypen sind plattformabhängig

## ANSI-C Grundtypen: Empfehlungen

---

- vorzugsweise die Grundtypen `char`, `int`, `double` verwenden  
*Die anderen Grundtypen nur verwenden, wenn es einen zwingenden Grund gibt.*

- Zusatz `const` verwenden, wenn eine Variable ihren Wert nach der Initialisierung nicht mehr ändern soll:

```
const double pi = 3.14159265358979323846;
```

- Achtung:

Die Mischung unterschiedlich großer Zahltypen sowie von Zahltypen mit und ohne Vorzeichen kann zu überraschenden Ergebnissen führen.

```
double x = 8.5 + 1 / 2; // setzt x auf 8.5 statt 9
unsigned a = 1;
int b = -2;
if (a + b > 0) ... // Summe ist 4294967295 statt -1
```

## ANSI-C Abgeleitete Typen: Zeiger (1)

Zu jedem Typ kann ein Zeigertyp (*Pointertyp*) abgeleitet werden, indem man in der Variablen-Definition einen Stern **\*** vor den Variablen-Namen schreibt.

- **Variablen-Definition:** `Typ Name = Wert;`

`Typ *Zeigername_1 = &Name;`

`Typ **Zeigername_2 = &Zeigername_1;`

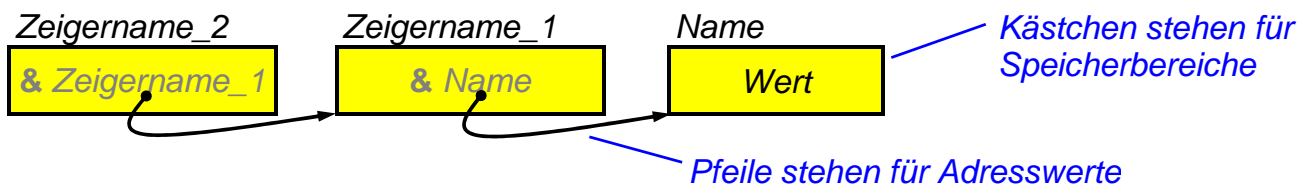
- **Wert:**

Die Adresse eines Speicherbereichs (*Wert 0 bedeutet, der Zeiger zeigt nirgendwohin*)

- **Platzbedarf** je nach Rechner bzw. Compiler:

`sizeof (int) ≤ sizeof (Typ*)` typisch: 4 Byte

- Grafische Darstellung:



## ANSI-C Abgeleitete Typen: Zeiger (2)

- Zeiger auf konstanten Wert:

`const Typ Name = Wert;`

`Typ *Zeigername = &Name; /* Fehler */`

`const Typ *Zeigername = &Name;`

*Der Wert einer Konstanten kann auch auf dem Umweg über Zeiger nicht geändert werden.*

- konstanter Zeiger:

`Typ Name = Wert;`

`Typ * const Zeigername = &Name;`

*Ein konstanter Zeiger zeigt während des ganzen Programmlaufs auf denselben Speicherbereich.*

- konstanter Zeiger auf konstanten Wert:

`const Typ * const Zeigername = &Name;`

- **Inhaltsoperator** `*` macht vom Zeiger adressierten Speicherbereich zugreifbar:

`*Zeigername` *Achtung: Programm-Absturz, wenn der Zeiger den Wert 0 hat*

Inhaltsoperator ist Gegenstück zum Adressoperator:

`*&Name` ist das gleiche wie `Name`

## ANSI-C Abgeleitete Typen: Zeiger (3)

### void-Pointer

- **Variablen-Definition:** *Typ Name = Wert;*  
`void *void_pointer = &Name;`
- **Wert:**  
Adresse eines Speicherbereichs beliebigen Typs (aber Inhalt nicht zugreifbar)
- **Platzbedarf:**  
wie andere Zeiger auch
- **Typecast-Operator** ( *T* ) wandelt einen void-Pointer in einen konkreten Pointer:  
*Typ \*typ\_pointer = ( Typ \* ) void\_pointer;*  
*Achtung: chaotische Laufzeitfehler, wenn der void-Pointer nicht auf einen Speicherbereich des angegebenen Typs zeigt*

## ANSI-C Abgeleitete Typen: Zeiger (4)

Verwendung von Zeigern z.B. bei dynamischer Speicher Verwaltung:

- die Funktion **malloc** reserviert Speicher für Werte eines Typs und liefert die Adresse des Speicherbereichs:

```
Typ *Zeigername = ( Typ* ) malloc( sizeof ( Typ ) );  
if ( Zeigername == NULL )  
{  
    ... /* Fehlerbehandlung */  
}
```

*Anzahl benötigte Bytes*

*malloc hat Rückgabetyt void\**

*malloc liefert die ungültige Adresse 0 (in ANSI-C als NULL geschrieben), wenn die angeforderte Menge Speicher nicht verfügbar ist.*

Achtung: malloc reserviert nur Speicher, initialisiert ihn aber nicht

- mit der Funktion **free** kann (und sollte!) per malloc reservierter Speicher irgendwann wieder freigegeben werden:

```
free ( Zeigername ) ;
```

`#include <stdlib.h>` erforderlich,  
damit malloc und free bekannt sind

# Beispiel-Programm Zeiger-Variable

- Quellcode:

```
#include <stdio.h>
int main(void)
{
    int n = 3082;
    int *p = &n;
    /* print pointer value */
    printf("p = %p\n", (void*) p);
    /* print pointer address */
    printf("&p = %p\n", (void*) &p);
    /* print pointer size */
    printf("sizeof p = %lu\n", (unsigned long) sizeof p);
    /* print dereferenced pointer value */
    printf("**p = %d\n", *p);
    return 0;
}
```

Konsolenausgabe  
des Programms:

```
p = 0x7fffcea7d8ec
&p = 0x7fffcea7d8e0
sizeof p = 8
*p = 3082
```

## ANSI-C Abgeleitete Typen: Felder (1)

Zu jedem Typ kann ein Feldtyp (Arraytyp) abgeleitet werden, indem man in der Variablen-Definition eine Feldgröße in Klammern **[ ]** angibt.

- **Variablen-Definition:** `Typ Feldname[Feldgröße] = {Wert_1, Wert_2, ...};`

*Die Feldgröße muss ein ganzzahliges Literal sein (oder eine symbolischer Name dafür).  
Die Feldgröße kann entfallen, wenn eine Initialisierung angegeben ist.*

- **Wert:** Folge von Werten gleichen Typs  
(Zugriff nur elementweise mit Indexoperator)
- **Platzbedarf:** `sizeof Feldname == Feldgröße * sizeof (Typ)`

- Grafische Darstellung: `Feldname[ ]`

<code>[0] = Wert_1</code>
<code>[1] = Wert_2</code>
<code>:</code>
<code>[Feldgröße - 1] = Wert_N</code>

## ANSI-C Abgeleitete Typen: Felder (2)

- **Indexoperator** [ ] macht die Feld-Elemente zugreifbar:

**Feldname[Index]**

*Der Index muss ganzzahlig sein und zwischen 0 und Feldgröße - 1 liegen.  
Indices außerhalb dieses Bereichs führen zu undefinierten Laufzeitfehlern!*

der Feldname ohne [ ] ist Kurzschreibweise für die Adresse des ersten Feldelements:

**Feldname** ist das gleiche wie **&Feldname[0]**

*Der Feldname ist also keine Name für den Speicherbereich des Felds,  
sondern ein Name für die Anfangsadresse des Felds!*

- der Indexoperator ist Kurzschreibweise für Inhaltsoperator und **Zeigerarithmetik**:

**Zeigername[Index]** ist das gleiche wie **\*(Zeigername + Index)**

Zeigerarithmetik arbeitet mit der Einheit **sizeof (Typ)**:

**Zeigername + Index** bedeutet **Adresse + Index \* sizeof (Typ)**

## ANSI-C Abgeleitete Typen: Felder (3)

Felder und dynamischer Speicherungsverwaltung:

- die Funktion **calloc** reserviert Speicher für ein Feld von Werten eines Typs und liefert die Adresse des Speicherbereichs:

```
Typ *Zeigername = (Typ*) calloc(Feldgröße, sizeof (Typ));  
if (Zeigername == NULL)  
{  
    ... /* Fehlerbehandlung */  
}
```

- calloc initialisiert den reservierten Speicher mit 0

*wird die Initialisierung nicht gebraucht, kann malloc verwendet werden:*

**Typ \*Zeigername = (Typ\*) malloc(Feldgröße \* sizeof (Typ));**

- Speicher auch bei calloc mit free wieder freigegeben:

**free (Zeigername) ;**

## Beispiel-Programm Feld-Variable

- Quellcode:

```
#include <stdio.h>
int main(void)
{
    int a[] = {3421, 3442, 3635, 3814};
    const int n = (int)(sizeof a / sizeof (int));
    int i;

    /* print array values and addresses */
    printf("a = %p\n", (void*) a);
    for (i = 0; i < n; ++i)
    {
        printf("%d: %p %d\n", i, (void*) &a[i], a[i]);
    }

    /* print array size */
    printf("sizeof a = %lu\n", (unsigned long) sizeof a);
    return 0;
}
```

Was gibt das Programm auf der Konsole aus?

## Beispiel-Programm Feld-Zeiger (1)

- Quellcode:

```
#include <stdio.h>
#include <stdlib.h> /* calloc, malloc, free, ... */
#include <stddef.h> /* NULL, size_t, ... */
int main(void)
{
    const int n = 4;
    int i;
    int *a = (int*) calloc((size_t) n, sizeof (int));
    if (a == NULL)
    {
        printf("Speicherreservierung fehlgeschlagen!\n");
        return 1;
    }

    a[0] = 3421;
    a[1] = 3442;
    a[2] = 3635;
    a[3] = 3814;
```

oder ohne Initialisierung mit 0:

`int *a = (int*) malloc(n * sizeof(int));`



## Beispiel-Programm Feld-Zeiger (2)

- Fortsetzung Quellcode:

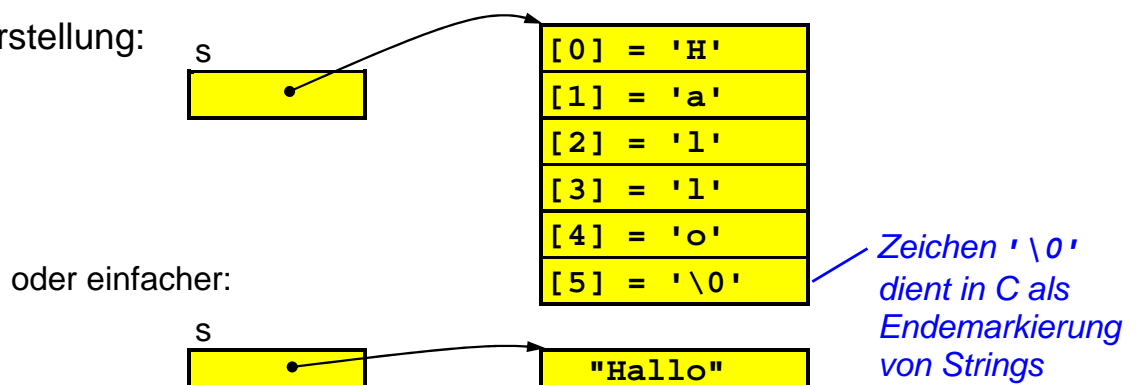
```
...
/* print array values and addresses */
printf("&a = %p\n", (void*)&a);
printf("a = %p\n", (void*)a);
for (i = 0; i < n; ++i)
{
    printf("%d: %p %d\n", i, (void*)&a[i], a[i]);
}
/* print array size */
printf("sizeof a = %lu\n", (unsigned long)sizeof a); /* pointer size */
printf("%d * sizeof *a = %lu\n", n, (unsigned long)n * sizeof *a);
free(a);
return 0;
}
```

## ANSI-C Abgeleitete Typen: String (1)

Ein String ist ein Feld von Einzelzeichen mit '\0' als letztem Zeichen.  
Strings werden über Zeiger-Variablen benutzt.

- Variablen-Definition:** `const char *s = "Hallo";` *const, weil String-Literal nicht änderbar!*
- Wert:** Anfangsadresse eines Strings (*d.h. die Adresse seines ersten Zeichens*)
- Platzbedarf:** `sizeof "Hallo" == 6` (*Anzahl Zeichen incl. '\0'*)  
`sizeof s == sizeof (char*)`

- Grafische Darstellung:



## ANSI-C Abgeleitete Typen: String (2)

String-Literale sind als Feld-Initialisierer verwendbar

- **Variablen-Definition:** `char s[] = "Hallo";`

*Kurzschreibweise für:*

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

- **Wert:** Folge der Zeichen (*Kopie des String-Literals einschließlich '\0'*)
- **Platzbedarf:** `sizeof s`  $\equiv$  6 (*Anzahl Zeichen einschl. '\0'*)
- Grafische Darstellung: s[ ]

[0]	=	'H'
[1]	=	'a'
[2]	=	'l'
[3]	=	'l'
[4]	=	'o'
[5]	=	'\0'

## ANSI-C Abgeleitete Typen: String (3)

- Manipulation von C-Strings mit Bibliotheks-Funktionen:

```
char *strcpy(char *s1, const char *s2);
```

*kopiert den String s2 in den Speicherbereich s1 und liefert s1 als Rückgabewert*

```
char *strcat(char *s1, const char *s2);
```

*hängt den String s2 an den String s1 an und liefert s1 als Rückgabewert*

```
int strcmp(const char *s1, const char *s2 );
```

*Vergleicht die Strings s1 und s2 und liefert 0, wenn die Strings gleich sind, eine Zahl größer 0 bei s1 > s2 bzw. eine Zahl kleiner 0 bei s1 < s2*

```
size_t strlen(const char *s);
```

*liefert die Länge des Strings s ohne '\0'*

*... /\* noch einige weitere str-Funktionen \*/*

## Beispiel-Programm String-Variablen (1)

- Quellcode:

Was gibt das Programm auf der Konsole aus?

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

int main(void)
{
    char a[] = "halli";
    const char *s = "hallo";
    char *t = NULL;

    /* compare, copy and concatenate strings */
    if (strcmp(a, s) < 0)
    {
        t = (char*) malloc(sizeof a + strlen(s));
        if (t == NULL) ... /* error handling */
        strcat(strcpy(t, a), s); /* or: strcpy(t, a); strcat(t, s); */
    }
    ...
}
```

*damit die ~~strxxx~~-Funktionen bekannt sind*

*strcpy und strcat  
allokieren keinen Speicher  
deshalb zuerst mit malloc  
genug Speicher reservieren*

## Beispiel-Programm String-Variablen (2)

- Fortsetzung Quellcode:

```
...
/* print string values and addresses */
printf("a = %p %s\ns = %p %s\nt = %p %s\n",
       (void*)a, a, (void*)s, s, (void*)t, t);

printf("sizeof a = %lu\n", (unsigned long) sizeof a); /* 6 */
printf("sizeof s = %lu\n", (unsigned long) sizeof s); /* 4 bzw. 8 */
printf("sizeof t = %lu\n", (unsigned long) sizeof t); /* 4 bzw. 8 */

printf("strlen(a) = %lu\n", (unsigned long) strlen(a)); /* 5 */
printf("strlen(s) = %lu\n", (unsigned long) strlen(s)); /* 5 */
printf("strlen(t) = %lu\n", (unsigned long) strlen(t)); /* 10 */

s = a; /* copies the address, not the string */
/* a = s; syntax error */

free(t);
return 0;
}
```

# ANSI-C Abgeleitete Typen: Felder von Feldern

Mehrdimensionale Felder am Beispiel einer 2x3-Matrix

- **Variablen-Definition:** `int matrix[2][3] = {{10, 11, 12}, {20, 21, 22}};`

- **Wert:** zeilenweise Folge der Matrix-Elemente  
(Zugriff nur elementweise mit Indizierungs-Operatoren)

- **Platzbedarf:** `sizeof matrix`  $\equiv$  `2 * 3 * sizeof (int)`

- Indizierung: `matrix[i][j]`  $\equiv$  `*(*(matrix + i) + j)`

- Grafische Darstellung:

	[0]	[1]	[2]
[0]	10	11	12
[1]	20	21	22

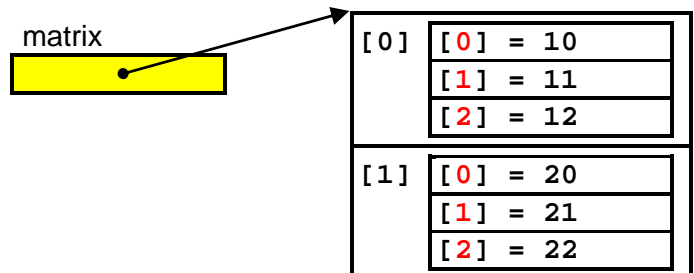
## Beispiel-Programm Matrix-Zeiger (1)

- Quellcode:

```
#include <stdio.h>
#include <stdlib.h>
#define M 3 /* number of columns */
int main(void)
{
    /* allocate and initialize memory for 2x3 matrix */
    const int n = 2; /* number of lines */
    int i, j;
    int (*matrix)[M] = (int(*)[M]) malloc(n * M * sizeof (int));
    if (matrix == NULL) ... /* error handling */

    matrix[0][0] = 10;
    matrix[0][1] = 11;
    matrix[0][2] = 12;
    matrix[1][0] = 20;
    matrix[1][1] = 21;
    matrix[1][2] = 22;
    ...
}
```

Spaltenanzahl muss bereits zur Übersetzungszeit feststehen



## Beispiel-Programm Matrix-Zeiger (2)

- Fortsetzung Quellcode:

```
...
/* print matrix addresses and values */
printf("&matrix = %p\n", (void*)&matrix);
printf("matrix = %p\n", (void*)matrix);
for (i = 0; i < n; ++i)
{
    printf("[%d] %p: %p\n", i, (void*)&matrix[i], (void*)matrix[i]);
    for (j = 0; j < M; ++j)
    {
        printf("    [%d] %p: %d\n", j, (void*)&matrix[i][j], matrix[i][j]);
    }
}

/* print matrix size */
printf("sizeof matrix = %lu\n", (unsigned long)sizeof matrix);
printf("%d * sizeof *matrix = %lu\n", n, (unsigned long)n * sizeof *matrix);
free(matrix);
return 0;
}
```

## ANSI-C Abgeleitete Typen: Felder von Zeigern

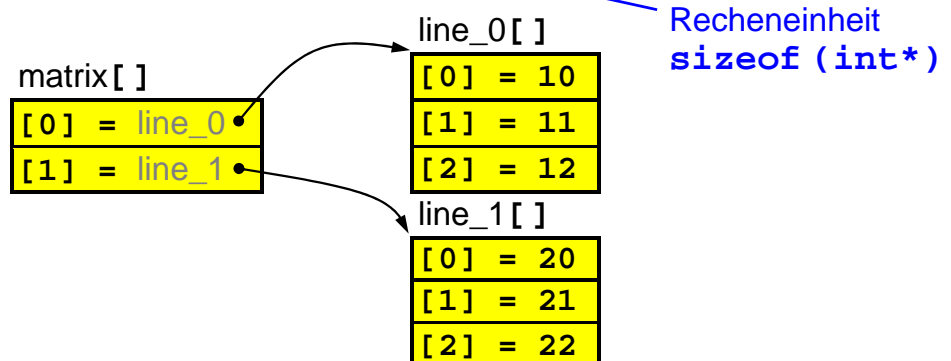
Felder von Zeigern am Beispiel einer 2x3-Matrix

- Variablen-Definition:** `int line_0[3] = {10, 11, 12};`  
`int line_1[3] = {20, 21, 22};`  
`int *matrix[2] = {line_0, line_1};`

- Wert:** Folge von Zeilen-Adressen

- Platzbedarf:** `sizeof matrix`  $\equiv 2 * \text{sizeof}(\text{int}^*)$
- Indizierung:** `matrix[i][j]`  $\equiv *((*(matrix + i) + j))$

- Grafische Darstellung:



## Beispiel-Programm Matrix-Doppelzeiger (1)

- Quellcode:

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

int main(void)
{
    /* allocate and initialize memory for 2x3-matrix */
    const int n = 2; /* number of lines */
    const int m = 3; /* number of columns */
    int i, j;

    int **matrix = (int**) malloc(n * sizeof (int*));
    if (matrix == NULL) ... /* error handling */
    for (i = 0; i < n; ++i)
    {
        matrix[i] = (int*) malloc(m * sizeof (int));
        if (matrix[i] == NULL) ... /* error handling */
    }
    ...
}
```

sowohl Zeilen- als auch Spaltenanzahl  
brauchen erst zur Laufzeit festzustehen

## Beispiel-Programm Matrix-Doppelzeiger (2)

- Fortsetzung Quellcode:

```
...
matrix[0][0] = 10;
...
matrix[1][2] = 22;
} /* wie Feld von Feldern (Folie 2-39)

/* print matrix addresses and values */
...
} /* wie Feld von Feldern (Folie 2-40),
   aber m statt M

/* free matrix memory */
for (i = 0; i < n; ++i)
{
    free(matrix[i]);
}
free(matrix);
return 0;
}
```

## ANSI-C Abgeleitete Typen: Vergleich mit Java

---

Bei abgeleiteten Typen kaum Gemeinsamkeiten zwischen ANSI-C und Java:

- ANSI-C Zeiger bieten sehr viel mehr Möglichkeiten als Java Referenzen  
*in Java nur Referenzen auf Objekte im Heap*  
*in ANSI-C Zeiger auf jeden beliebigen Speicherbereich, auch auf dem Stack*
- ANSI-C kennt keinen echten Feld-Typ  
*der Indexoperator ist nur eine Kurzschreibweise für Adressarithmetik*  
*und kann auf jede beliebige Adresse angewendet werden*  
*die Feldlänge wird nicht im Feld hinterlegt,*  
*deshalb beim Feldzugriff keine automatische Überwachung der Indexgrenzen*  
*in Java Felder nur im Heap, in ANSI-C auch auf dem Stack*  
*Felder von Feldern gibt es in Java nicht*
- ANSI-C kennt keinen echten String-Typ  
*nur Felder von Zeichen mit ungültigem Zeichen ' \0 ' als Endemarkierung*

## ANSI-C Abgeleitete Typen: Empfehlungen

---

- **Zeiger-Typen** sind ein zentrales Konzept von ANSI-C
- **Feld-Typen** sind verkappte Verwandte der Zeiger  
*der Name einer Feld-Variablen ist kein Name für einen Speicherbereich,*  
*sondern ein Name für die Adresse eines Speicherbereichs*  
*an Stelle von Feld-Variablen besser Zeiger auf mit **calloc** bzw. **malloc***  
*dynamisch reservierten Speicher verwenden (**free** nicht vergessen!)*  
*an Stelle der Felder von Feldern besser Felder von Zeigern verwenden*
- **Strings** sind Felder von Einzelzeichen  
*Speicherreservierung per Feld-Variable (vermeiden) oder dynamisch per **malloc** (besser)*  
*beim Speicherplatzbedarf das abschließende ' \0 ' -Zeichen nicht vergessen!*

# ANSI-C Benutzerdefinierte Typen: enum

Eine Aufzählung (*Enumeration*) definiert Namen für `int`-Literele.

- **Typ-Deklaration:**

*Vorsicht:  
die Namen der  
Enumeratoren  
sind nicht lokal  
zur Typdeklaration!*

```
enum Enumname
{
    Enumerator_1 = Wert_1,
    Enumerator_2 = Wert_2,
    ...
    Enumerator_N = Wert_N
};
```

*Die Angabe der  
Enumerator-Werte  
ist optional.*

*Default-Wert für den  
ersten Enumerator ist 0,  
für die anderen der  
Vorgängerwert plus 1.*

- **Variablen-Definition:**

```
enum Enumname Name = Enumerator;
```

- **Wert:**

einer der Enumerator-Werte

*Enumerator-Werte können überall verwendet werden,  
wo `int`-Werte verwendet werden können.*

- **Platzbedarf:**

```
sizeof (enum Enumname) == sizeof (int)
```

## Beispiel-Programm enum-Variable

- Quellcode:

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

```
enum month {jan = 1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec};
```

```
int main(void)
{
```

```
    /* enum month aMonth = 3; funktioniert bei C, aber nicht bei C++ */
```

```
    enum month aMonth = mar;
```

```
    /* print variable value */
```

```
    printf("aMonth = %d\n", aMonth);
```

```
    /* print variable address */
```

```
    printf("&aMonth = %p\n", (void*) &aMonth);
```

```
    /* print variable size */
```

```
    printf("sizeof aMonth = %lu\n", (unsigned long) sizeof aMonth);
```

```
    return 0;
```

```
}
```

Konsolenausgabe  
des Programms:

```
aMonth = 3
&aMonth = 0x22efc4
sizeof aMonth = 4
```



## ANSI-C Benutzerdefinierte Typen: struct (1)

Eine Struktur fasst Werte beliebiger Typen zusammen.

- Typ-Deklaration:

```
struct Strukturname
{
    Typ_1 Komponente_1;
    ...
    Typ_N Komponente_N;
};
```

- Variablen-Definition:

```
struct Strukturname Name = { Wert_1, ..., Wert_N };
```

- Wert: Folge der Komponenten-Werte.

- Platzbedarf:  $\sum_{i=1}^N \text{sizeof}(\text{Typ}_i) \leq \text{sizeof}(\text{struct Strukturname})$   
wegen Alignment der Komponenten

- Grafische Darstellung:

Name
Komponente_1 = Wert_1
:
Komponente_N = Wert_N

## ANSI-C Benutzerdefinierte Typen: struct (2)

- Komponentenauswahl-Operatoren (Punkt und Pfeil):

`Name.Komponente_1`

`Zeigername->Komponente_1`

Pfeil ist Kurzschreibweise für  
`(*Zeigername).Komponente_2`

- Adresse einer Komponente:

`&Name.Komponente_1`

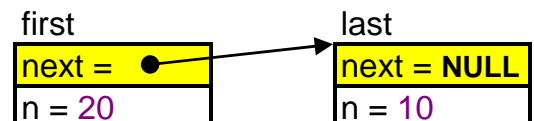
`&Zeigername->Komponente_1`

Adresse der ersten Komponente  
ist Adresse der Struktur insgesamt

- Verkettete Strukturen enthalten einen Zeiger auf den eigenen Strukturtyp:

```
struct int_list
{
    struct int_list *next; /* Verkettung */
    int n;
};

struct int_list last = { NULL, 10 };
struct int_list first = { &last, 20 };
```



# Beispiel-Programm struct-Variable

- Quellcode:

```
#include <stdio.h>

struct date
{
    int day;
    const char *month;
    int year;
};

...

int main(void)
{
    struct date d = {1, "September", 2000};
    /* print variable value */
    printf("%d. %s %d\n", d.day, d.month, d.year);
    /* print variable address */
    printf("&d = %p\n", (void*) &d);
    printf("&d.day = %p\n", (void*) &d.day);
    printf("&d.month = %p\n", (void*) &d.month);
    printf("&d.year = %p\n", (void*) &d.year);
    /* print variable size */
    printf("sizeof d = %lu\n",
        (unsigned long) sizeof d);
    return 0;
}
```

## ANSI-C Benutzerdefinierte Typen: union (1)

Eine Variante ist eine Struktur, bei der alle Komponenten dieselbe Adresse haben.

- Typ-Deklaration:

```
union Unionname
{
    Typ_1 Variante_1;
    ...
    Typ_N Variante_N;
};
```

zu einer Zeit kann  
nur eine der Varianten  
gespeichert sein

nur die erste Variante  
kann initialisiert werden

- Variablen-Definition:

```
union Unionname Name = {Wert_1};
```

- Wert: der Wert einer der Varianten

- Platzbedarf:  $\text{sizeof}(\text{union } \text{Unionname}) \equiv \text{MAX}_{i=1}^N \text{sizeof}(\text{Type}_i)$

- Grafische Darstellung:

Name  
Variante\_1 = Wert\_1

## ANSI-C Benutzerdefinierte Typen: union (2)

- Variantenauswahl-Operatoren (Punkt und Pfeil):

`Name.Variante_2`

`Zeigername->Variante_2`

- unbenannte Varianten:

```
enum int_or_string {type_int, type_string};
struct struct_with_union
{
    enum int_or_string u_type;
    union
    {
        int i;
        char *s;
    } u;
};
```

*hier kein Unionname*

```
struct struct_with_union x;
x.u_type = type_int;
x.u.i = 1;
x.u_type = type_string;
x.u.s = "Hallo";
```

## ANSI-C Benutzerdefinierte Typen: typedef

Eine typedef-Deklaration definiert lediglich einen Aliasnamen für einen Typ.

- Deklaration: `typedef Typname Aliasname;`
- Variablen-Definition:  
`Typname Name;`  
`Aliasname Name;` } *beide Definitionen sind gleichwertig*
- besonders nützlich bei enum-, struct und union-Typen:

```
struct date
{
    int day;
    const char *month;
    int year;
};
typedef struct date date;
date d = {1, "September", 2000}; /* statt struct date d... */
```

*date ist Aliasname für struct date  
(gleicher Bezeichner für struct und Alias  
ist erlaubt und übliche Konvention)*

## ANSI-C Benutzerdefinierte Typen: Vergleich mit Java

---

Bei den benutzerdefinierten Typen große Unterschiede zwischen ANSI-C und Java:

- **enum**-Typen sind sehr viel primitiver realisiert als in Java  
*in ANSI-C eigentlich nur eine nette Schreibweise für ganzzahlige Konstanten*
- **struct**-Typen sind eine primitive Vorstufe der Java-Klassen  
*nur öffentliche Instanzvariablen*  
*keine Methoden und Konstruktoren*  
*keine Vererbung*  
*auch Wert-Variablen möglich (in Java nur Speicherreservierung mit **new**)*
- **union**-Typen gibt es in Java nicht  
*in Java wegen Vererbung und Polymorphie überflüssig*

## ANSI-C Benutzerdefinierte Typen: Empfehlungen

---

- **enum**-Typen sind nützlich für die Codierung nicht-numerischer Information.  
*Verarbeitung oft mit **switch**-Anweisungen*
- **struct**-Typen sind das zentrale Konzept für benutzerdefinierte Typen  
*verkettete Strukturen sind oft ein guter Ersatz für Felder*
- **union**-Typen gefährden die Typsicherheit  
*vorzugsweise innerhalb eines **struct**-Typs als unbenannte Variante zusammen mit einer Typ-Komponente verwenden*
- **typedef**-Aliasnamen sind eine nützliche Schreibvereinfachung  
*können Programme änderungsfreundlicher machen*

# ANSI-C Daten: Lernzettel

---

#define Adresse Adressoperator Alignment Array calloc char const double  
enum Feld Feld von Feldern Feld von Zeigern float free Indexoperator  
Inhaltsoperator int Literal long malloc Pointer short signed sizeof size\_t  
strcat strcmp strcpy strlen struct symbolische Konstante typedef union  
unsigned Variable void wchar\_t Zeiger Zeigerarithmetik