

# Programmiergrundlage C99

Vor C gab es B, eine Sprache entwickelt von Ken Thompson und basiert auf BCPL, eine Systemprogrammiersprache welche auf Algol 60 basiert, welche eine der frühesten & einflussreichsten Programmiersprachen ist.

Viele moderne Sprachen basieren auf C: C++, Java, Perl, etc.

C ist...

- Hohe Sprache auf niedrigem Niveau → Zugriff auf Konzepte der Maschinenebene (Bytes & Adressen)
- Kleine & einfache Sprache → Sehr viel musste selbst entwickelt werden
- Freizügige Sprache → C geht davon aus, dass man weiß, was man tut
- + Schnelle Sprache
- + Programmportabilität
- + Verschiedenste Nutzungsbereiche (von OS bis zu Datenvorarbeitung)
- + Starke Standardbibliothek
- Fehlerranfälligkeit durch Flexibilität von C → Programmfehler werden nicht durch den Compiler erkannt
- Schwierig zu verstehen
- Schwierig zum Verändern von Programmen

## Sequenz

Kann mehrere Ausdrücke ausführen. Je nach Befehl hat die Sequenz andere Effekte.

```
int a = 2, i = 3, b = 3;
```

↳ Variablen-deklaration/-initialisierung mit einer Sequenz deklariert/initialisiert die Variablen vom selben Typ

```
if (statement_1, statement_2, statement_3);
while (statement_1, statement_2, statement_3);
for (init_1, init_2; check_1, check_2; calc_1, calc_2);
```

letztes bestimmt Schleifenverhalten

## Deklarierung & Initialisierung

Deklarierung: int a;



Initialisierung: a = 2; oder int b = 3;  
↳ Erste Zuweisung

## break, return, continue, goto

- break bricht eine Schleife & Switch ab.
- return springt aus der aktuellen Funktion heraus zurück an die Aufrufstelle.
  - ↳ bei void return;
  - ↳ bei anderen Typen return (a);
- continue springt direkt ans Ende der Schleife (bezieht sich nur auf Schleifen wie while, for, do-while).
- goto springt direkt zu angegebenem Label (im Beispiel 'downtown').  
downtown:  
/\* continue stuff \*/

## Definition vs. Deklarierung

Deklarierung macht Namen und Typ bekannt.

Definition reserviert (zu dem) Speicher.

## "Lokale" Funktionsvariablen & Return

Die Werte von Funktionsvariablen können via return normal übergeben werden.

! Variablen & Pointer welche im Functionscope erstellt wurden sollten nicht returned werden, da diese außerhalb des Scopes nicht mehr existieren.

## Modulo CGJ

$$a \% b \rightarrow a \% |b|$$

Wenn a negativ ist, ist das Resultat ebenfalls negativ.

z.B.  $3 \% -5 = 3 ; -3 \% -5 = -3$

## scanf vs gets

- scanf schneidet nach dem ersten Leerzeichen ab: "Hello World"  $\Rightarrow$  "Hello"  
"%s" nimmt 'unendlich' viele Elemente auf (solange ohne Leerzeichen); "%ns" liest Anzahl n Zeichen bis
  - gets stoppt nach einem Zeilenumbruch, aber kann überfüllt werden, da die maximale Anzahl Zeichen nicht angegeben werden kann.
  - fgets wie gets einfach für Streams. Kann zusätzlich Anzahl einzulesende Zeichen angeben werden. Endet den String mit "\r\n" (von Entertaste).
-

## Kompilieren & Verknüpfen

### Vorverarbeitung

Ersetz von Direktive  
(#define, #include, ...)

### Kompilieren

Programm in Maschinenanweisungen übersetzen

### Verknüpfen / Linken

Übersetztes Programm & Standardfunktionen werden zusammengeknüpft

a > A

```
gross = klein + 32; oder = klein - 'a' + 'A';
klein = gross - 32; oder = gross - 'A' + 'a';
```

## Programmstruktur

- 1) `#include` Anweisungen
- 2) `#define` Anweisungen
- 3) Typ-Definition (`typedef`)
- 4) Deklaration von externen Variablen
- 5) Prototypen für andere Funktionen als `main`
- 6) Definition von `main`
- 7) Definitionen für andere Funktionen.

## Define (keine) Rekursion

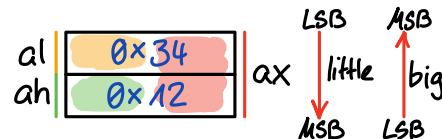
Define Präprozessoren können nicht rekursiv miteinander verwendet werden.

```
#define N(X) M((X) * (X))
#define M(Y) N((Y) + (Y))

int x = N(8);
    ↓
int x = N(((8)*(8)) + ((8)*(8)));
```

## big-Endian, little-Endian

Bezieht sich auf die Datengrößen 16-, 32-, 64-Bits.  
Es werden ganze Bytes betrachtet, nicht die Bits.



- ▶ little-endian wird das erste Byte als LSB betrachtet, das letzte als MSB.  
↳ z.B. 16-Bit = 0x1234 = 4'660
- ▶ big-endian wird das erste Byte als MSB betrachtet, das letzte als LSB.  
↳ z.B. 16-Bit = 0x3412 = 13'330

## Initialisieren

! entspricht nicht dem Zuweisungsoperator `=`

- ↳ C versucht den Initialwert auf den Datentyp der Variable zu konvertieren, mit den Regeln des Zuweisungsoperators.

```
struct { uint8_t test; } var = {.test = 44};
int8_t test[5] = {[0] = 1, [1] = 2, [2] = test[1]};
```

```
(const) int8_t variable = 5;
static int test = variable;
```

X statische Werte müssen konstant sein.

## Uninitialisierte Variablen

- ▶ static Variablen haben standardmäßig den Wert Null. → Lieber initialisieren
- ▶ Variablen mit automatischer Spezifizierung haben einen zufälligen Wert, da der automatische Speicher vor zu belegt wird.

## Speicherklasse

```

int a;
extern int b;
static int c;

void f(int d, register int e) {
    auto int g;
    int h;
    static int i;
    extern int j;
    register int k;
}

```

Name	Speicherklasse	Geltungsbereich	Verknüpfung
a	statisch	Datei	extern
b	statisch	Datei	⊗
c	statisch	Datei	intern
d	automatisch	Block	keine
e	automatisch	Block	keine
f	automatisch	Block	keine
h	automatisch	Block	keine
i	statisch	Block	keine
j	statisch	Block	⊗
k	automatisch	Block	keine

## Variablen

<b>auto</b>	⇒ Standardmäßig in <b>Blöcken</b> ⇒ Befreit den Speicher der Variable beim Verlassen des Blocks ⇒ In <b>Top-Level</b> (Dateiebene) <b>nicht möglich</b> ⇒ Wird immer wieder initialisiert
<b>static</b>	⇒ Standardmäßig im <b>Top-Level</b> ⇒ <b>Top:</b> Interne Verknüpfung (schützt Variable von aussen via <b>extern</b> ) ⇒ <b>Block:</b> Keine Verknüpfung ⇒ Wird einmalig initialisiert (fixer Speicher)
<b>extern</b>	⇒ Besitzt keinen realen Speicher → Referenz von bereits existierenden Variablen (Main & Modul) ⇒ Kann nur für Variablen auf <b>Top-Level</b> von anderen Dateien verwendet werden.

## Funktion

<b>extern</b>	⇒ Standardmäßig bei Funktionsprototypen (in Header)
<b>static</b>	⇒ Interne Verknüpfung & Namen wiederverwendbar in anderen Dateien

## Typenqualifizieren

<b>volatile</b>	⇒ Flüchtige Speicher (ändert ohne Programmeinfluss → z.B. <b>msTicks</b> ) in gewissen Fällen wegoptimiert (direkte, anstatt indirekte Referenz) ⇒ Teilt dem Kompiler mit, dass die Variable nicht wegoptimiert werden darf.
<b>const</b>	⇒ Nicht gleich wie <b>#define: const</b> deklariert eine schreibgeschützte Variable (Wert & Adresse) ⇒ Gilt nicht als Konstante (Grösse von statischen Feldern nicht setzbar damit)

## Deklaratoren

<b>int *(*x[10])(void)</b>	1. <b>Feld von...</b> 2. <b>Zeiger auf...</b> 3. <b>Funktionen...</b> 4. <b>ohne Argumente...</b> 5. <b>mit Rückgabewerte Zeiger auf int</b>	<b>int *ap[10]</b>	1. <b>Feld von...</b> 2. <b>Zeiger auf...</b> 3. <b>Ganzzahlen</b>
<b>int f(int)[];</b> <b>int g(int) (int);</b> <b>int a[10](int);</b>	✗ Funktionen können keine Felder zurückgeben ✗ Funktionen können keine Funktionen zurückgeben ✗ Funktionsfelder nicht möglich ✓ Funktionen können <b>Zeiger</b> (auf Felder, Funktionen,etc.) zurückgeben		



## Normal

```
if (expression_1)
{
    statement_1;
    statement_2;
}
else if (expression_2)
{
    statement_a;
    statement_b;
}
else
{
    statement_else;
    statement_elsf;
}
```

► Wenn Expression wahr, dann if-Statements, sonst else.

## Kurz

```
if (expression_1)
    single_statement;
else if (expression_2)
    single_statement;
else
    single_statement;
```

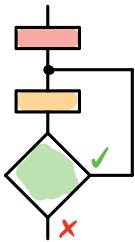
## Shorthand

```
if (expression) ? true_statement : false_statement;
```

True

False

## For-Schleife



`for (expr1; expr2; expr3)`

Wiederholt die Schleife bis das Prüfstatement false zurückgibt.

## Typendefinition

```
typedef type type_name;
```

## Struct / Union / Enum

## Funktionen

```
typedef int (function)(int, int);
function add;

int add(int a, int b){
    return a + b;
}
```

```
#include <int>
int a = 5;
int_ptr test = &a;
```

## Normal

```
typedef struct node_s { ... } node_t;
```

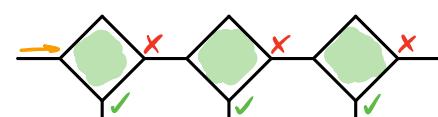
## S g

```
typedef char *string_ptr;
string_ptr strg = "Hello World";
printf("%s\n", strg);
```

\*1 Adresse verifizieren bevor auf den Hauptpointer geladen wird.

```
uint8_t *tmp = realloc(ptr, NEW_SIZE_BYTES);
if (tmp != NULL) {
    ptr = tmp;
}
```

{Wichtig!}

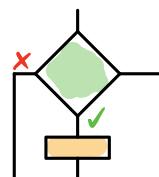


## Switch - Case

## Ganzzahl (signed & unsigned)

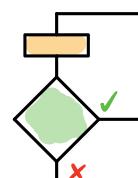
```
switch (expression)
{
    case 3:
    case 1:
        break;
    case default:
    case 5:
        break;
    default:
        break;
}
```

break verwenden zum Ausbrechen ohne weitere cases auszuführen.



## While - Schleife Ausführung 0+ mal

`while (expression) { statement }`



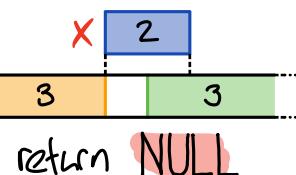
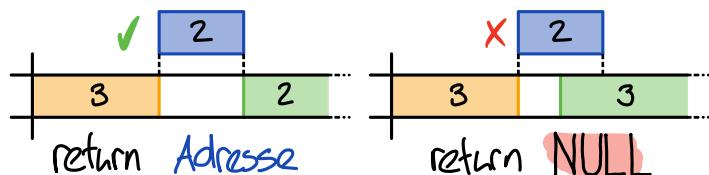
## Do - While - Schleife 1+ mal

`do { statement } while (expression);`

## Dynamischer Speicher

### malloc

```
malloc(AMOUNT * BYTE_SIZE);
```



### calloc

```
calloc(AMOUNT, BYTE_SIZE);
```

Gleich wie malloc, außer frischer Speicher wird mit 0 initialisiert.

### realloc \*

```
realloc(pointer, NEW_BYTE_SIZE);
```

Erweitert bestehender Speicher und gibt bei Erfolg die neue Adresse zurück, ansonsten NULL.

### Free

```
free(pointer);
```

Befreit den Speicher an der angegebenen Adresse.  
► Muss Anfang von Speicher

### Read-Only Pointers

```
uint8_t *const ptr = ...
```

ptr kann nicht verändert werden.

# Limits.h

Name	Wert	Formel	Beschreibung
SHRT_MIN	$\leq -32767$	$-(2^{15} - 1)$	Kleinster <code>short int</code> Wert
SHRT_MAX	$\geq +32767$	$2^{15} - 1$	Grösster <code>short int</code> Wert
USHRT_MAX	$\geq 65535$	$2^{16} - 1$	Grösster <code>unsigned short int</code> Wert
INT_MIN	$\leq -32767$	$-(2^{15} - 1)$	Kleinster <code>int</code> Wert
INT_MAX	$\geq +32767$	$2^{15} - 1$	Grösster <code>int</code> Wert
UINT_MAX	$\geq 65535$	$2^{16} - 1$	Grösster <code>unsigned int</code> Wert
LONG_MIN	$\leq -2147483647$	$-(2^{31} - 1)$	Kleinster <code>long int</code> Wert
LONG_MAX	$\geq +2147483647$	$2^{31} - 1$	Grösster <code>long int</code> Wert
ULONG_MAX	$\geq 4294967295$	$2^{32} - 1$	Grösster <code>unsigned long int</code> Wert
LLONG_MIN*	$\leq -9223372036854775807$	$-(2^{63} - 1)$	Kleinster <code>long long int</code> Wert
LLONG_MAX*	$\geq +9223372036854775807$	$2^{63} - 1$	Grösster <code>long long int</code> Wert
ULLONG_MAX*	$\geq 18446744073709551615$	$2^{64} - 1$	Grösster <code>unsigned long long int</code> Wert

\* ab C99

Name	Wert	Beschreibung
CHAR_BIT	$\geq 8$	Anzahl Bits pro Byte
SCHAR_MIN	$\leq 127$	Kleinster <code>signed char</code> Wert
SCHAR_MAX	$\geq +127$	Grösster <code>signed char</code> Wert
UCHAR_MAX	$\geq 255$	Grösster <code>unsigned char</code> Wert
CHAR_MIN	*	Kleinster <code>char</code> Wert
CHAR_MAX	**	Grösster <code>char</code> Wert

\* CHAR\_MIN ist gleich SCHAR\_MIN falls `char` vorzeichenbehaftet ist, andernfalls ist CHAR\_MIN gleich 0.

\*\* CHAR\_MAX ist entweder SCHAR\_MAX oder UCHAR\_MAX, abhängig davon ob `char` vorzeichenbehaftet oder vorzeichenlos ist.

Rang	Name	Symbol	Assoziativität
1	Array Indizierung	[]	links $\rightarrow$ rechts
	Funktionsaufruf	()	links
	Strukturelement	. ->	links
	Inkrement (postfix)	++	links
	Dekrement (postfix)	--	links
2	Inkrement (präfix)	++	rechts $\rightarrow$ links
	Dekrement (präfix)	--	rechts
	Referenz	&	rechts
	Dereferenz	*	rechts
	unäres Plus z.B. +6	+	rechts
	unäres Minus z.B. -5	-	rechts
	Komplement bitweise	~	rechts
	Negation logisch	!	rechts
	Grösse	sizeof	rechts
3	Cast	()	rechts
4	Multiplikativ	* / %	links
5	Additiv	+ -	links
6	Schieben bitweise	<< >>	links
7	Relationale Operatoren	< > <= >=	links
8	Gleichheit	== !=	links
9	Und bitweise	&	links
10	Oder exklusiv bitweise	^	links
11	Oder inklusiv bitweise		links
12	Und logisch	&&	links
13	Oder logisch		links
14	Bedingung	? :	rechts
15	Zuweisung $x =+y \rightarrow x =+y$	= *= /= %=	rechts
	$x =-y \rightarrow x =-y$	+= -= <<= >>=	
		&= ^=  =	
16	Sequenz	,	links

Arithmetische Operatoren

Zuweisungsoperator

Inkrement & Dekrement

Logische Ausdrücke

Bitweise Operatoren

links  $\rightarrow$  rechts

((a > b) > c) > d

rechts  $\rightarrow$  links

(a = (b = (c = d)))

Auswertung

1) Verwendete Operatoren aufisten

2) Liste sortieren

3) Höchste nach kleinste Priorität einklammern

Ausdrucksanweisung

Jedes Ausdruck kann auch als Anweisung verwendet werden.

$\hookrightarrow ++i; i$  wird zuerst inkrementiert und danach wird der Wert von  $i$  abgefragt & verworfen!

! Klammer werden zuerst aufgelöst

! Punkt & Komma vor Strich

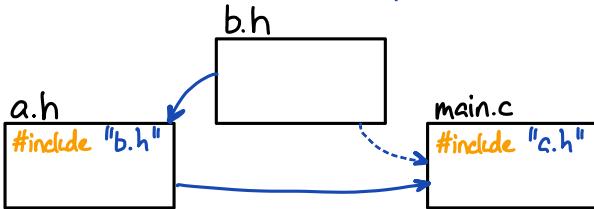
!  $i = i * j + k \neq i * = j + k$

# Präprozessor

## #include

```
#include <include.h>
#include "include.h"
```

<--> Durchsucht zuerst System-Kopfdateien (include)  
 ...> Durchsucht zuerst lokale Kopfdateien



## #undef

```
#undef identifier
```

Löst bestehende Definitionen & Makros mit gleichem Namen auf.

## #define

```
#define PI (3.14159265359) ✓ Wichtig!
#define MACRO(x,y,z,...) ((x) + (y) / (z))
```

## Identifier    Replacement-list    Parameters

## Probleme

```
#define MAX(x, y) ((x) > (y) ? (x) : (y))
MAX(t++, j) → ((t++) > (j) ? (t++) : (j))
```

! Argumente werden mehrmals ausgeführt.

## Andere Direktiven

```
#error "message" optional
#line 1("file")
#pragma data(heap_size = > 1000) ← tokens sind compiler-abhängig
```

# Structs

```
typedef struct tag_name_s{
    uint32_t a;
    uint8_t *b;
    uint8_t c;
    uint16_t d;
} type_name_t;
```

Order of Memory

→ ohne `typedef` können zwischen & Variablen deklariert werden.

gleich Struct → Kopieren

② & part. variable



Kopiert alles von b & a

# Vordefinierte Makros

Name	Bedeutung
_LINE	Zeilennummer der kompilierten Datei
_FILE	Dateiname der kompilierten Datei
_DATE	Datum der Kompilation
_TIME	Zeitpunkt der Kompilation
_STDC	1 falls C99 oder C89 Kompiler

## # & ## Operatoren

```
#define MACRO(n) (#n "Hello")
MACRO(2++) → ("2++" "Hello")
```

```
#define MAKE_ID(n) (id##n)
uint8_t MAKE_ID(1), MAKE_ID(2);
```

```
id1 = 4; id2 = 2;
```

(ident #x) → #x = "ident \\"foo\\"

(foo)

## Header-Guard

```
#ifndef _INCLUDE_H_
#define _INCLUDE_H_
/* ... declarations here ... */
#endif
```

Hält den Präprozessor davon auf, eine Header-Datei mehrmals zu inkludieren.

↳ Ansonsten kann es zu einem Error führen

## Union

Gleich wie bei struct:

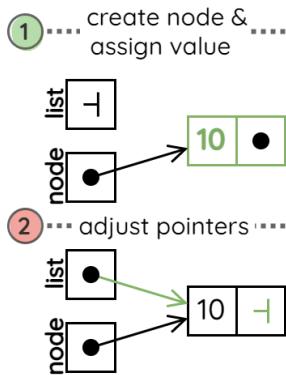
- = Operator zum Kopieren
- return kompatibel

```
typedef union tag_name_u{
    uint32_t a;
    struct{
        uint16_t b;
        uint16_t c;
    } byte;
} type_name_t;
```

Order of Memory

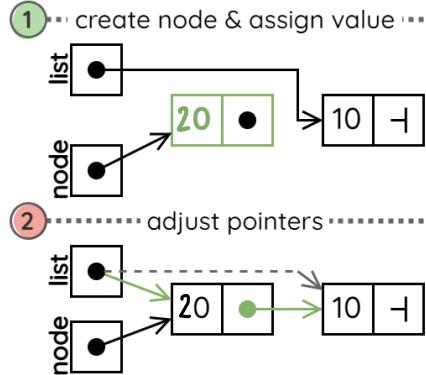


## Node in Liste einfügen (am Start)

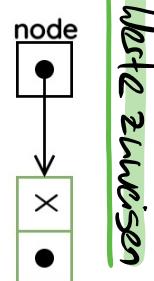


! Beides ist gleich.  
Sobald ein Node zwischen zwei Nodes eingefügt werden möchte, wird anders vorgegangen.

```
node_t *node = malloc(sizeof(node_t));
node->value = 10;
node->next = list;
list = node;
```

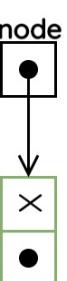
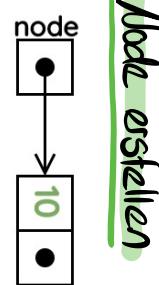


```
node->value = 10;
(*node).value = 10;
```



```
node_t *node = malloc(sizeof(node_t));
```

## Liste erstellen

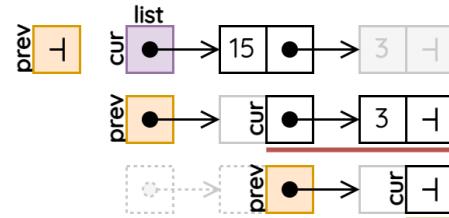


```
list = node;
```

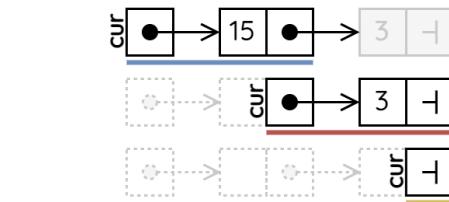
## Liste erstellen

## Durchsuchen einer Liste

... Search for previous & current node ...



... Search for current node ...



### Node suchen

```
for(cur = list;
    cur != NULL && cur->value != n;
    cur = cur->next);
```

### Vorherige Node suchen

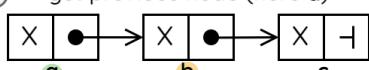
```
for(cur = list, prev = NULL;
    cur != NULL && cur->value != n;
    prev = cur, cur = cur->next);
```

### Such-Statement

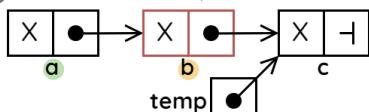
! Man muss zusätzlich prüfen, ob man sich am Ende oder am Anfang der Liste befindet.

## Node löschen

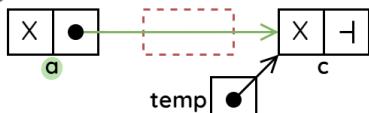
1 ... get previous node (here a) ...



2 ... save b->next ptr & free b ...



3 ... assign saved ptr to a-next ...



1 node\_t \*a = get\_previous\_node(b);

2 node\_t \*tmp = b->next;  
free(b);

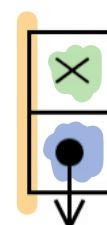
3 a->next = tmp;

```
typedef struct node_s {
    uint32_t value;
    struct node_s *next;
} node_t;
```

## Node

## Zeiger

- [type] \* ptr; deklariert einen Zeiger mit einem Typ.
- const[type] \* ptr; → kann Wert (von der Adresse) nicht geändert werden.
- [type] \*const ptr; → die Adresse kann nicht geändert werden.



Call by Value → void add(int x) kopiert Original (nicht änderbar)

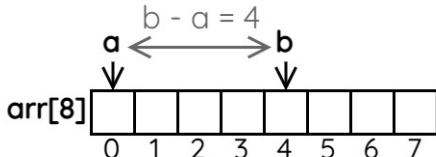
Call by Reference → void add(int \*x) referenziert Original (änderbar)

## Zeigerarithmetik

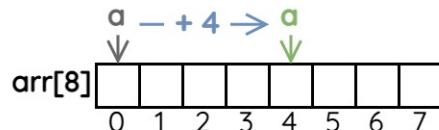
! Nur mit Felder-Pointer, ansonsten undefinedes Verhalten

int \*p = &a[0]; oder = a;

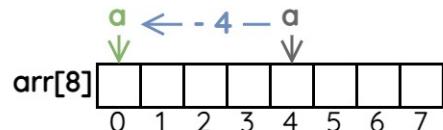
## Differenz zwischen zwei Pointer



## Ganzzahl-Addition



## Ganzzahl-Subtraktion



## Zeichenketten

```
char planets[][8] = {"Merkur", "Venus", "Erde", "Mars",  
                     "Jupiter", "Saturn", "Uranus", "Neptun"};
```

	0	1	2	3	4	5	6	7
0	'M'	'e'	'r'	'k'	'u'	'r'	'\0'	'\0'
1	'V'	'e'	'n'	'u'	's'	'\0'	'\0'	'\0'
2	'E'	'r'	'd'	'e'	'\0'	'\0'	'\0'	'\0'
3	'M'	'a'	'r'	's'	'\0'	'\0'	'\0'	'\0'
4	'J'	'u'	'p'	'i'	't'	'e'	'r'	'\0'
5	'S'	'a'	't'	'u'	'r'	'n'	'\0'	'\0'
6	'U'	'r'	'a'	'n'	'u'	's'	'\0'	'\0'
7	'N'	'e'	'p'	't'	'u'	'n'	'\0'	'\0'

## Zeichenketten

enden immer mit  
'\0' (außer kein Platz  
für '\0').

→ planets[0][0] = 'M'

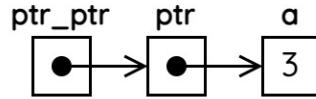
```
char * planets[] = {"Merkur", "Venus", "Erde", "Mars",  
                     "Jupiter", "Saturn", "Uranus", "Neptun"};
```

0	• →	'M'	'e'	'r'	'k'	'u'	'r'	'\0'
1	• →	'V'	'e'	'n'	'u'	's'	'\0'	
2	• →	'E'	'r'	'd'	'e'	'\0'		
3	• →	'M'	'a'	'r'	's'	'\0'		
4	• →	'J'	'u'	'p'	'i'	't'	'e'	'r'
5	• →	'S'	'a'	't'	'u'	'r'	'n'	'\0'
6	• →	'U'	'r'	'a'	'n'	'u'	's'	'\0'
7	• →	'N'	'e'	'p'	't'	'u'	'n'	'\0'

## Zeiger auf Zeiger

Wird verwendet um den Wert eines Zeigers in einer Funktion zu verändern.

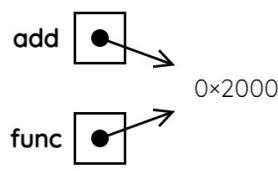
```
uint8_t a = 3;  
uint8_t *ptr = &a;  
uint8_t **ptr_ptr = &ptr;
```



↳ Funktionsparameter sind schlussendlich 'kopierte' Werte. Setzt man den Zeigerwert im Funktionsscope, wird dieser beim Scopeausstieg verworfen, daher Zeiger auf Zeiger.

## Zeiger auf Funktion

```
int add(int x, int y) { return (x + y); }  
int (*func)(int, int) = add;  
  
printf("%i\n", func(5,7));
```

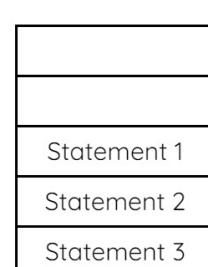


## Wichtig für Zeiger

char \*str = "hallo"; → sizeof(str) = 4  
\*str = 'd'; → undefinedes Verhalten!

char str[] = "hallo"; kann verändert werden

void f(int \*a / int a[])



→ Operator

ptr → x = \*ptr.x

## Typenkonvertierung

- implizite Konvertierung wird automatisch vom Compiler gemacht.
- explizite Konvertierung werden mit dem **cast**-Operator gemacht.

Ganzzahl → Gleitzahl

char  
int var = 'A';  
int var = (int) 'A';

## Explizite Konvertierung / ()-Operator

(type) expression → int i = (int) 4.0f;

## Usual Arithmetic Conversion

Operanden in kleinsten Typ konvertieren, der beide Werte aufnehmen kann.

### ► Promotion - Rangliste

1. long long int, unsigned long long int
2. long int, unsigned long int
3. int, unsigned int
4. short int, unsigned short int
5. char, signed char, unsigned char
6. -Bool (bool)

### ► Gleitzahl-Promotion

float → double → long double

↳ Mischungen von Ganzzahl- und Gleitkommazahl werden in den entsprechenden Gleitkomma-Typ konvertiert.

→ (int, double) → (double, double)

Schlägt dem Compiler vor, die angegebenen Casts & Operanden in den entsprechenden Typen zu konvertieren.

### ► Ganzzahl-Promotion ?? Wichtig

Operanden mit Typen geringer als Typ int oder unsigned int werden automatisch zu int oder unsigned int konvertiert (gecastet).

### ↳ signed gemischt mit unsigned

Vorzeichenbehaftete Operanden werden in den Typ des vorzeichenlosen Operanden konvertiert.

## Zeichenketten

## Zeichenliteral - Zeiger

char \*str = "Hello";

- Zeigt auf das Literal selbst → read-only

sizeof(str) = 4

↳ da es sich nur um einen Zeiger handelt.

strlen(str) = 5 → \0 wird ignoriert & strlen benötigt nur ein Zeiger

- Zeichenkette welche mit dem Literal "Hello" initialisiert wird.

sizeof(str) = 1 × 6 = 6

↳ char Feld (1 Byte pro Zeichen) mit 6 Zeichen

H e l l o \0 ↲ Standardmäßig bei Zeichenketten.

strlen(str) = 5 → \0 wird ignoriert.

↳ Feld kann auf Textgröße gesetzt werden ('\0 abgeschnitten), führt aber zu Problemen.

H e l l o \0 \0 \0 \0 \0 \0

↳ Am besten immer +1 bei fixer Länge für Platz von '\0'

↳ Ist das Feld grösser als der Text, werden die restlichen Felder mit '\0' (also 0) aufgefüllt (bei Kompilierung).

