

# Übungen zu Betriebssysteme

## Einführung in die Programmiersprache C

---

Sommersemester 2023

Henriette Hofmeier, Manuel Vögele, Benedict Herzog, Timo Hönig

Bochum Operating Systems and System Software Group (BOSS)



RUHR  
UNIVERSITÄT  
BOCHUM

RUB

## 1 C Grundlagen

1.1 Hello World

1.2 Datentypen

1.3 Operatoren

1.4 Kontrollstrukturen

1.5 Einschub: Übersetzen

## 2 C Besonderheiten

2.1 Zeiger

2.2 Funktionen

2.3 Formatierte Ausgabe

2.4 Speicherverwaltung

## 3 Fortgeschrittenes C

3.1 Typumwandlungen

3.2 Zeigerarithmetik

3.3 Zeichenketten und Ein-/Ausgabe

## Was ist C?

- C ist alt (1970er)
- C ist stabil (Versionen: C89, C99, C11/C18)
- C ist eine hardwarenahe Hochsprache
- C ist imperativ und prozedural
- C ist typisiert aber nicht typsicher
- C schützt den Programmierer nicht
  - manuelle Speicherverwaltung
  - keine Checks auf Programmierfehler
  - keine Checks der Speicherzugriffe

## Was ist C?

- C ist alt (1970er)
- C ist stabil (Versionen: C89, C99, C11/C18)
- C ist eine hardwarenahe Hochsprache
- C ist imperativ und prozedural
- C ist typisiert aber nicht typsicher
- C schützt den Programmierer nicht
  - manuelle Speicherverwaltung
  - keine Checks auf Programmierfehler
  - keine Checks der Speicherzugriffe

## Warum C?

- Laufzeiteffizienz
- Platzeffizienz
- Systemnähe
- Portabilität

## Was ist C?

- C ist alt (1970er)
- C ist stabil (Versionen: C89, C99, C11/C18)
- C ist eine hardwarenahe Hochsprache
- C ist imperativ und prozedural
- C ist typisiert aber nicht typsicher
- C schützt den Programmierer nicht
  - manuelle Speicherverwaltung
  - keine Checks auf Programmierfehler
  - keine Checks der Speicherzugriffe

## Warum C?

- Laufzeiteffizienz
- Platzeffizienz
- Systemnähe
- Portabilität

## Warum kein C?

- Fehleranfälligkeit
- Umständlichkeit
- Alte Programmiersprachenkonzepte

# Hello World!

```
#include <stdio.h>

void greet_user(void) {
    printf("Hello World!\n");
}

int main(int argc, char *argv[]) {
    greet_user();
    return 0;
}
```

# Hello World!

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

```
void greet_user(void) {  
    printf("Hello World!\n");  
}
```

Anzahl Befehlszeilenparameter

Befehlszeilenparameter

```
int main(int argc, char *argv[]) {  
    greet_user();  
    return 0;  
}
```

# Hello World!

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

```
void greet_user(void) {  
    printf("Hello World!\n");  
}
```

← Funktions**definition**

```
int main(int argc, char *argv[]) {  
    greet_user();  
    return 0;  
}
```

← Funktions**aufruf**



# Hello World!

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

← **C Library** (libc) einbinden

```
void greet_user(void) {
```

```
    printf("Hello World!\n");
```

← libc Funktion zur Ausgabe aufrufen

```
}
```

```
int main(int argc, char *argv[]) {
```

```
    greet_user();
```

```
    return 0;
```

```
}
```

# Hello World!

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

```
void greet_user(void) {  
    printf("Hello World!\n");  
}
```

```
int main(int argc, char *argv[]) {  
    greet_user();  
    return 0;  
}
```

← Programm mit dem Exitcode 0 beenden

# Hello World!

```
#include <stdio.h>

void greet_user(void) {
    printf("Hello World!\n");
}

int main(int argc, char *argv[]) {
    greet_user();
    return 0;
}
```

```
user@host:~$ gcc hello.c -o hello
user@host:~$ ./hello
Hello World!
user@host:~$ ...
```

## Primitive Datentypen

- Leerer Datentyp

`void`

- Booleans (seit C99)

`_Bool, bool`

- Zeichen (8-Bit Zahlen in ASCII-Code)

`char`

# Datentypen in C

## Primitive Datentypen

- Leerer Datentyp

`void`

- Booleans (seit C99)

`_Bool, bool`

- Zeichen (8-Bit Zahlen in ASCII-Code)

`char`

- Ganzzahlen

`char, short, int, long, long long`

- Gleitkommazahlen

`float, double, long double`

```
#include <stdbool.h>

int main(void) {
    bool b = true;
    char c = 'A';
    int i = 0x2A;
    unsigned int u = 42;
    float f = -1.23f;
}
```

# Datentypen in C

## Primitive Datentypen

- Leerer Datentyp

`void`

- Booleans (seit C99)

`_Bool, bool`

- Zeichen (8-Bit Zahlen in ASCII-Code)

`char`

- Ganzzahlen

`char, short, int, long, long long`

- Gleitkommazahlen

`float, double, long double`

```
#include <stdbool.h>

int main(void) {
    bool b = true;
    char c = 'A';
    int i = 0x2A;
    unsigned int u = 42;
    float f = -1.23f;
}
```

Zusätzlich: Typ-Modifer `signed` (Default), `unsigned` und `const`

## Datentypen in C - Größe und Wertebereich

- Datentypgröße (und damit der Wertebereich) ist plattformabhängig
- Größe zur Compilezeit ermittelbar: `sizeof(int)` oder `sizeof(<var>)`  
Rückgabewert `sizeof`: `size_t`

	Java	C-Standard	gcc (x86-32)	gcc (x86-64)
char	16	$\geq 8$	8	8
short	16	$\geq 16$	16	16
int	32	$\geq 16$	32	32
long	64	$\geq 32$	32	64
long long	-	$\geq 64$	64	64

# Datentypen in C - Größe und Wertebereich

- Datentypgröße (und damit der Wertebereich) ist plattformabhängig
- Größe zur Compilezeit ermittelbar: `sizeof(int)` oder `sizeof(<var>)`  
Rückgabewert `sizeof`: `size_t`

	Java	C-Standard	gcc (x86-32)	gcc (x86-64)
char	16	$\geq 8$	8	8
short	16	$\geq 16$	16	16
int	32	$\geq 16$	32	32
long	64	$\geq 32$	32	64
long long	-	$\geq 64$	64	64

- Außerdem gilt:
  - $\text{char} \leq \text{short} \leq \text{int} \leq \text{long} \leq \text{long long}$
  - $\text{float} \leq \text{double} \leq \text{long double}$
- Wertebereich:

$$\begin{array}{llll} \text{unsigned:} & 0 & \rightarrow & +(2^{\text{Bits}} - 1) \\ \text{signed:} & -(2^{\text{Bits}-1}) & \rightarrow & +(2^{\text{Bits}-1} - 1) \end{array}$$



## Arrays

- Felder von Daten gleichen Typs
- 0 indiziert
- Falls Größe zur Compilezeit ermittelbar: `sizeof()`
- ! Keine Längenprüfung beim Zugriff!

```
#include <stdio.h>

int main(void) {
    int numbers[4] = {1,2,3,4};
    int numbers2[2]; // uninit.
    numbers2[0] = 23;
    numbers2[1] = 42;

    char string[] = "Hallo Welt";
    printf("%c\n", string[6]); // 'W'
    string[6] = 'B';
    printf("%c\n", string[6]); // 'B'
}
```

# Datentypen in C - Felder

## Arrays

- Felder von Daten gleichen Typs
- 0 indiziert
- Falls Größe zur Compilezeit ermittelbar: `sizeof()`
- ! Keine Längenprüfung beim Zugriff!

## Strings

- Array aus `chars`
- Terminiert mit `'\0'`

'H'	'a'	'l'	'l'	'o'	' '	'W'	'e'	'l'	't'	'\0'
0	1	2	3	4	5	6	7	8	9	10
72	97	108	108	111	32	87	101	108	116	0

ASCII

```
#include <stdio.h>

int main(void) {
    int numbers[4] = {1,2,3,4};
    int numbers2[2]; // uninit.
    numbers2[0] = 23;
    numbers2[1] = 42;

    char string[] = "Hallo Welt";
    printf("%c\n", string[6]); // 'W'
    string[6] = 'B';
    printf("%c\n", string[6]); // 'B'
}
```

## Strukturen

- fassen Daten verschiedenen Typs zusammen
- ähnlich zu Klassen – nur ohne Methoden

```
struct date {  
    int year;  
    int month;  
    int day;  
};  
  
typedef struct date date_t;  
  
int main(void) {  
    struct date today = {2022, 2, 2};  
  
    // day jew. implizit genullt  
    date_t tomorrow = {2022, 2};  
    date_t yesterday = {  
        .year = 2022,  
        .month = 2  
    };  
  
    tomorrow.day = today.day + 1;  
    yesterday.day = today.day - 1;  
}
```

## Strukturen

- fassen Daten verschiedenen Typs zusammen
- ähnlich zu Klassen – nur ohne Methoden

## Typ-Aliase

- alternativer Name für einen Typ
- z.B.: `stddef.h`:  
`typedef long unsigned int size_t`  
(nur gültig für x86-64!)
- praktisch z.B. für Strukturen

```
struct date {  
    int year;  
    int month;  
    int day;  
};  
  
typedef struct date date_t;  
  
int main(void) {  
    struct date today = {2022, 2, 2};  
  
    // day jew. implizit genullt  
    date_t tomorrow = {2022, 2};  
    date_t yesterday = {  
        .year = 2022,  
        .month = 2  
    };  
  
    tomorrow.day = today.day + 1;  
    yesterday.day = today.day - 1;  
}
```

# Arithmetische Operatoren in C

## alle Ganz-/Fließkommazahlen

+	Addition
-	Subtraktion
*	Multiplikation
/	Division
unäres -	negatives Vorzeichen (z.B. -a)
unäres +	positives Vorzeichen (z.B. +3)

```
int main(void) {  
    // Integerdivision  
    int a = 51 / 5; // a = 10  
    int c = (a + 5 - (-3) / 10) * 10;  
  
    int d = 1;  
  
    // Präinkrement/-dekrement  
    // (erst Änderung, dann Wert)  
    int e = ++d1; // e = 2, d1 = 2  
  
    d = 1;  
  
    // Postinkrement/-dekrement  
    // (erst Wert, dann Änderung)  
    int h = d1++; // h = 1, d1 = 2  
}
```

# Arithmetische Operatoren in C

## alle Ganz-/Fließkommazahlen

+	Addition
-	Subtraktion
*	Multiplikation
/	Division
unäres -	negatives Vorzeichen (z.B. -a)
unäres +	positives Vorzeichen (z.B. +3)

## nur Ganzzahlen

%	Modulo (Rest bei Division)
++	Inkrement um 1
--	Dekrement um 1

## abgekürzte Zuweisung

a <op>= b; entspricht a = a <op> b;

```
int main(void) {  
    // Integerdivision  
    int a = 51 / 5;    // a = 10  
    int c = (a + 5 - (-3) / 10) * 10;  
  
    int d = 1;  
  
    // Präinkrement/-dekrement  
    // (erst Änderung, dann Wert)  
    int e = ++d1;    // e = 2, d1 = 2  
  
    d = 1;  
  
    // Postinkrement/-dekrement  
    // (erst Wert, dann Änderung)  
    int h = d1++;    // h = 1, d1 = 2  
}
```

## Logische Operatoren

<code>&amp;&amp;</code>	Konjunktion (log. "und")
<code>  </code>	Disjunktion (log. "oder")
<code>!</code>	Negation (log. "nicht")

- Ergebnis und Operanden sind vom Typ `int`

**Operanden:**

0	→	falsch
≠ 0	→	wahr

**Ergebnis:**

falsch	→	0
wahr	→	1

- Auswertung endet sobald Ergebnis feststeht  
(*Lazy Evaluation*)

```
#include <stdbool.h>

int main(void) {
    bool a = true;
    int b = 0;           // false

    bool c = a && b;      // false

    // (a && b) wird nicht
    // ausgewertet
    int g = 42 || (a && b);
}
```

## Bitoperatoren

&	bitweises "und"
	bitweises "oder"
^	bitweises "exklusiv-oder"
~	bitweises "nicht"
<<	bitweises Schieben (links)
>>	bitweises Schieben (rechts)

```
int main(void) {  
    int a = 0x10;  
    int b = 0x01;  
  
    int c = a | b;    // c = 0x11  
    int d = a & b;    // d = 0  
  
    unsigned g = (1 << 2); // g = 4  
}
```



# Vergleichsoperatoren in C

## Vergleichsoperatoren

<	kleiner
<=	kleiner gleich
>	größer
>=	größer gleich
==	gleich
!=	ungleich

### ■ Ergebnis ist vom Typ `int`

wahr	→	1
falsch	→	0

```
int main(void) {  
    int a = 42;  
    int b = 23;  
  
    if(a < b) { /* ... */ } // false  
    if(b >= a) { /* ... */ } // false  
    if(a == b) { /* ... */ } // false  
    if(a != b) { /* ... */ } // true  
}
```

# Kontrollstrukturen - Bedingte Ausführung

```
if(<condition>) {  
    // <condition> wahr  
} else {  
    // <condition> unwahr  
}
```

```
if(<cond>) {  
    // <cond> wahr  
} else if(<cond2>) {  
    // <cond> unwahr und  
    // <cond2> wahr  
} else {  
    // <cond> und  
    // <cond2> unwahr  
}
```

```
switch(<value>) {  
    case 0: /* ... */ break;  
    case 1: /* ... */  
    case 2: /* ... */ break;  
    default: /* ... */ break;  
}
```

- Die Bearbeitung bei `switch`-Anweisungen muss explizit mit `break` beendet werden  
→ ansonsten *fall through* in den nächsten `case`

# Kontrollstrukturen - Schleifen

```
while(<condition>) {  
    /* ... */  
}
```

```
while(42) {  
    /* ... */  
}
```

```
do {  
    /* ... */  
} while(<condition>);
```

```
do {  
    /* ... */  
} while(a > 0);
```

```
for(<init>; <cond>; <inc>) {  
    /* ... */  
}
```

```
for(int i = 0; i < 10; ++i) {  
    /* ... */  
}
```

- `break` verlässt die (innerste) Schleife
- `continue` beginnt direkt nächsten Schleifendurchlauf

# Einschub: Portable Programme

- Entwicklung portabler Programme durch Verwendung definierter Schnittstellen

## ANSI C11

- Normierung des Sprachumfangs der Programmiersprache C
- Standard-Bibliotheksfunktionen, z. B. `printf`, `malloc`

# Einschub: Portable Programme

- Entwicklung portabler Programme durch Verwendung definierter Schnittstellen

## ANSI C11

- Normierung des Sprachumfangs der Programmiersprache C
- Standard-Bibliotheksfunktionen, z. B. `printf`, `malloc`

## Single UNIX Specification, Version 4 (SUSv4)

- Standardisierung der Betriebssystemschnittstelle
- Wird von verschiedenen Betriebssystemen implementiert:
  - Solaris, HP/UX, AIX (SUSv3)
  - Mac OS X (SUSv3)
  - ... und natürlich Linux (SUSv4, aber nicht offiziell zertifiziert)

# Einschub: Übersetzen von C Programmen

- Übersetzen einer Quelldatei mit gcc:

```
gcc <flags> -o <output> input [input...]
```

- Flags:

-o	setze Name der Ausgabedatei
-std=c11	benutze Version C11 des ANSI C-Standards
-pedantic	akzeptiere nur ANSI-C11-konformen Quellcode
-D_XOPEN_SOURCE=700	erlaube nur SUSv4-konforme Betriebssystemaufrufe
-Wall	aktiviere zusätzliche Warnungen
-Werror	behandle Warnungen wie Fehler
-g	erzeuge Debug-Symbole in der ausführbaren Datei (praktisch zum debuggen – ggf. sehr große Programme)

- Beispiel:

```
user@host:~$ gcc -std=c11 -pedantic -D_XOPEN_SOURCE=700 -Wall -Werror -o test test.c
user@host:~$ ./test
```

## Einschub: Anforderungen an abgegebene Lösungen

- C-Sprachumfang konform zu ANSI C11
- Betriebssystemschnittstelle konform zu SUSv4
- **warnings-** und **fehlerfrei** in der BS-VM mit folgenden gcc-Optionen übersetzen:  
`-std=c11 -pedantic -D_XOPEN_SOURCE=700 -Wall -Werror`

## Einschub: Anforderungen an abgegebene Lösungen

- C-Sprachumfang konform zu ANSI C11
- Betriebssystemschnittstelle konform zu SUSv4
- **warnings-** und **fehlerfrei** in der BS-VM mit folgenden gcc-Optionen übersetzen:  
-std=c11 -pedantic -D\_XOPEN\_SOURCE=700 -Wall -Werror
- Später: **Korrekte** Fehlerbehandlung
- Später: Soweit möglich keine (modul-)globalen Variablen
- Siehe Korrekturrichtlinien (siehe Moodle)



## 1 C Grundlagen

1.1 Hello World

1.2 Datentypen

1.3 Operatoren

1.4 Kontrollstrukturen

1.5 Einschub: Übersetzen

## 2 C Besonderheiten

2.1 Zeiger

2.2 Funktionen

2.3 Formatierte Ausgabe

2.4 Speicherverwaltung

## 3 Fortgeschrittenes C

3.1 Typumwandlungen

3.2 Zeigerarithmetik

3.3 Zeichenketten und Ein-/Ausgabe

## Zeiger

- enthält die **Adresse** einer anderen Variable
- \* im Typ zeigt Zeigertyp an
- Syntax: type \*name  
→ Zeiger mit Namen name der auf Variable vom Typ type zeigt

```
#include <stdio.h>

int main(void) {
    int a = 42;

    // b zeigt auf a
    int *b = &a;

    // lese a (über b)
    printf("%d\n", *b);

    // lese a direkt
    printf("%d\n", a);

    // schreibe a (über b)
    *b = 23;
    printf("%d\n", *b);
    printf("%d\n", a);

    // schreibe a (direkt)
    a = 42;
    printf("%d\n", *b);
    printf("%d\n", a);
}
```

# Zeigerkonzept in C

## Zeiger

- enthält die **Adresse** einer anderen Variable
- \* im Typ zeigt Zeigertyp an
- Syntax: type \*name  
→ Zeiger mit Namen name der auf Variable vom Typ type zeigt

## Adressoperator &

- liefert die Adresse einer Variablen

## Dereferenzierungsoperator \*

- kann nur auf Zeiger angewendet werden
- greift auf die Variable zu, auf die der Zeiger zeigt

```
#include <stdio.h>

int main(void) {
    int a = 42;

    // b zeigt auf a
    int *b = &a;

    // lese a (über b)
    printf("%d\n", *b);

    // lese a direkt
    printf("%d\n", a);

    // schreibe a (über b)
    *b = 23;
    printf("%d\n", *b);
    printf("%d\n", a);

    // schreibe a (direkt)
    a = 42;
    printf("%d\n", *b);
    printf("%d\n", a);
}
```

## Mehrfachbelegung

- ein \* kann bedeuten:
  - Multiplikation
  - Zeigertypdefinition
  - Zeigerdereferenzierung
- ein & kann bedeuten:
  - bitweises "und"
  - Adressoperator

```
int main(void) {  
    int a = 42;  
  
    // Zeigerdefinition  
    int *b;  
  
    // Adressoperator  
    b = &a;  
  
    // Multiplikation  
    int x = a * 3;  
  
    // bitweises und  
    int y = a & 3;  
  
    // Dereferenzierung  
    *b = 23;  
}
```

# Zeigerkonzept in C - Anwendungsbeispiel

```
typedef struct ListNode {  
    int data;  
    struct ListNode *next;  
} node_t;
```

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

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

```
    node_t *list_head = NULL;
```

```
    node_t node0 = { .data = 42, .next = NULL };
```

```
    node_t node1 = { .data = 1337, .next = &node0 };
```

```
    node_t node2 = { .data = -23, .next = &node1 };
```

```
    list_head = &node2;
```

```
// [...]
```

list\_head

NULL

# Zeigerkonzept in C - Anwendungsbeispiel

```
typedef struct ListNode {  
    int data;  
    struct ListNode *next;  
} node_t;
```

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

```
int main(void) {  
    node_t *list_head = NULL;  
  
    node_t node0 = { .data = 42, .next = NULL };  
    node_t node1 = { .data = 1337, .next = &node0 };  
    node_t node2 = { .data = -23, .next = &node1 };  
  
    list_head = &node2;  
    // [...]
```

list\_head



node0



# Zeigerkonzept in C - Anwendungsbeispiel

```
typedef struct ListNode {  
    int data;  
    struct ListNode *next;  
} node_t;
```

```
#include <stdio.h>  
  
int main(void) {  
    node_t *list_head = NULL;  
  
    node_t node0 = { .data = 42, .next = NULL };  
    node_t node1 = { .data = 1337, .next = &node0 };  
    node_t node2 = { .data = -23, .next = &node1 };  
  
    list_head = &node2;  
    // [...]
```

list\_head



# Zeigerkonzept in C - Anwendungsbeispiel

```
typedef struct ListNode {  
    int data;  
    struct ListNode *next;  
} node_t;
```

```
#include <stdio.h>  
  
int main(void) {  
    node_t *list_head = NULL;  
  
    node_t node0 = { .data = 42,    .next = NULL };  
    node_t node1 = { .data = 1337,  .next = &node0 };  
    node_t node2 = { .data = -23,   .next = &node1 };  
  
    list_head = &node2;  
    // [...]
```

list\_head



node2



node1



node0





# Zeigerkonzept in C - Anwendungsbeispiel

```
typedef struct ListNode {  
    int data;  
    struct ListNode *next;  
} node_t;
```

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

```
int main(void) {  
    node_t *list_head = NULL;  
  
    node_t node0 = { .data = 42,    .next = NULL };  
    node_t node1 = { .data = 1337, .next = &node0 };  
    node_t node2 = { .data = -23,  .next = &node1 };
```

```
    list_head = &node2;
```

```
    // [...]
```



# Zeigerkonzept in C - Anwendungsbeispiel

```
typedef struct ListNode {  
    int data;  
    struct ListNode *next;  
} node_t;
```

```
// [...]
```

```
// node2: -23
```

```
printf("%d\n", (*list_head).data);
```

```
printf("%d\n", list_head->data);
```

```
// node1: 1337
```

```
printf("%d\n", (*(list_head->next).data);
```

```
printf("%d\n", list_head->next->data);
```

```
// [...]
```



# Zeigerkonzept in C - Anwendungsbeispiel

```
typedef struct ListNode {  
    int data;  
    struct ListNode *next;  
} node_t;
```

```
// [...]
```

```
// node2: -23
```

```
printf("%d\n", (*list_head).data);
```

```
printf("%d\n", list_head->data);
```

```
// node1: 1337
```

```
printf("%d\n", ((*list_head).next).data);
```

```
printf("%d\n", list_head->next->data);
```

```
// [...]
```



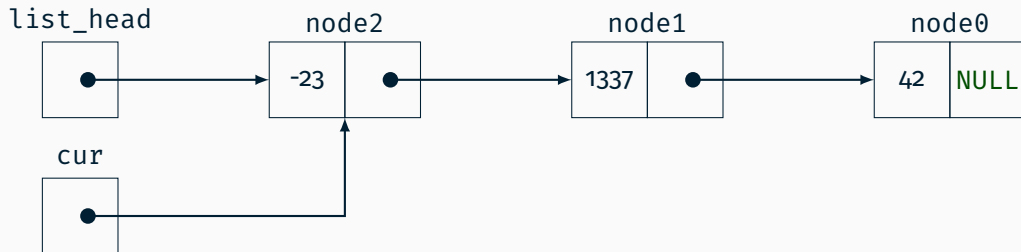
- `->` ist die lesbarere Form von `(*var)`.

# Zeigerkonzept in C - Anwendungsbeispiel

```
typedef struct ListNode {  
    int data;  
    struct ListNode *next;  
} node_t;
```

```
// [...]
```

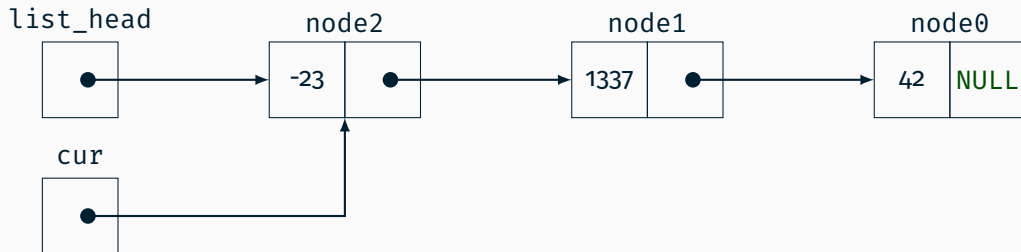
```
node_t *cur = list_head;  
while(cur != NULL) {  
    printf("%d\n", cur->data);  
    cur = cur->next;  
}  
printf("fertig\n");  
} // end main
```



# Zeigerkonzept in C - Anwendungsbeispiel

```
typedef struct ListNode {  
    int data;  
    struct ListNode *next;  
} node_t;
```

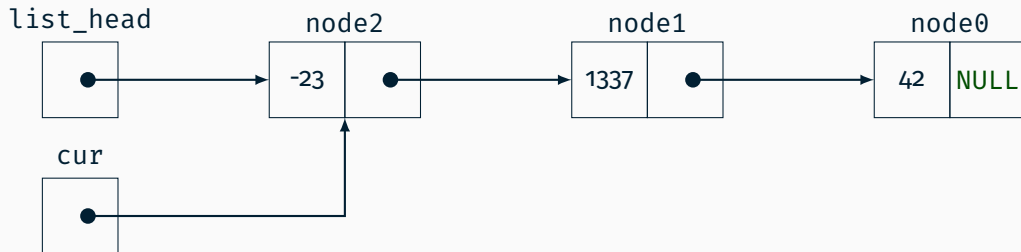
```
// [...]  
  
node_t *cur = list_head;  
while(cur != NULL) {  
    printf("%d\n", cur->data);  
    cur = cur->next;  
}  
printf("fertig\n");  
} // end main
```



# Zeigerkonzept in C - Anwendungsbeispiel

```
typedef struct ListNode {  
    int data;  
    struct ListNode *next;  
} node_t;
```

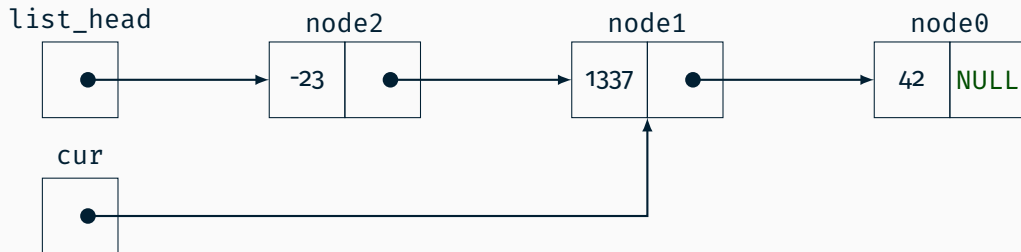
```
// [...]  
node_t *cur = list_head;  
while(cur != NULL) {  
    printf("%d\n", cur->data);  
    cur = cur->next;  
}  
printf("fertig\n");  
} // end main
```



# Zeigerkonzept in C - Anwendungsbeispiel

```
typedef struct ListNode {  
    int data;  
    struct ListNode *next;  
} node_t;
```

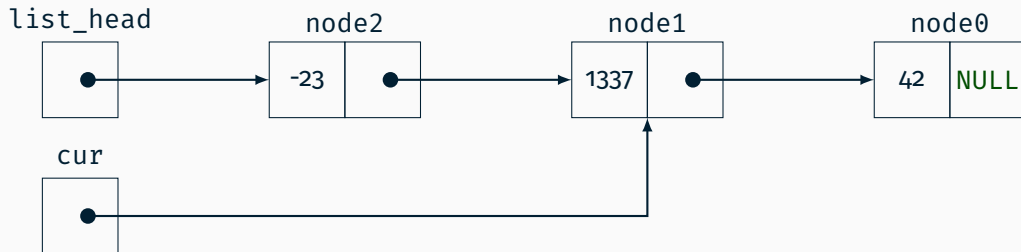
```
// [...]  
  
node_t *cur = list_head;  
while(cur != NULL) {  
    printf("%d\n", cur->data);  
    cur = cur->next;  
}  
printf("fertig\n");  
} // end main
```



# Zeigerkonzept in C - Anwendungsbeispiel

```
typedef struct ListNode {  
    int data;  
    struct ListNode *next;  
} node_t;
```

```
// [...]  
  
node_t *cur = list_head;  
while(cur != NULL) {  
    printf("%d\n", cur->data);  
    cur = cur->next;  
}  
printf("fertig\n");  
} // end main
```

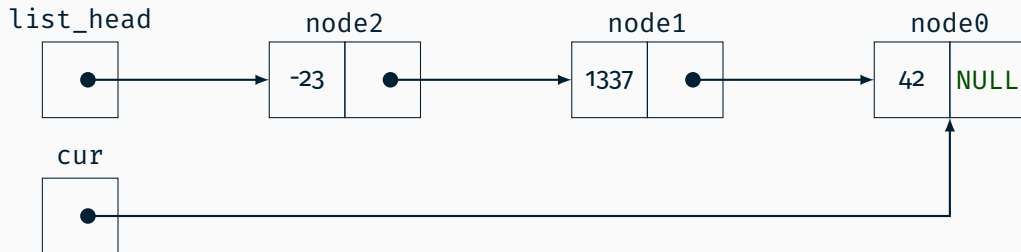




# Zeigerkonzept in C - Anwendungsbeispiel

```
typedef struct ListNode {  
    int data;  
    struct ListNode *next;  
} node_t;
```

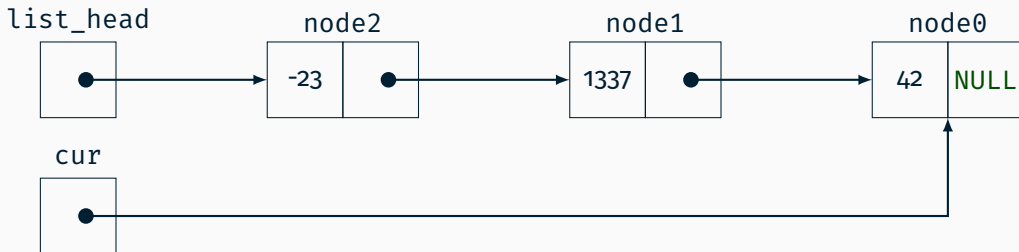
```
// [...]  
node_t *cur = list_head;  
while(cur != NULL) {  
    printf("%d\n", cur->data);  
    cur = cur->next;  
}  
printf("fertig\n");  
} // end main
```



# Zeigerkonzept in C - Anwendungsbeispiel

```
typedef struct ListNode {  
    int data;  
    struct ListNode *next;  
} node_t;
```

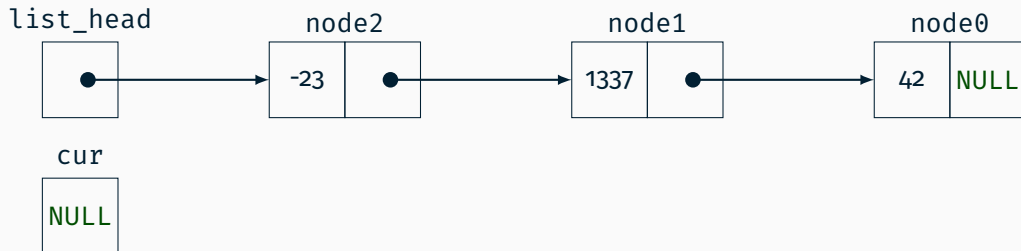
```
// [...]  
  
node_t *cur = list_head;  
while(cur != NULL) {  
    printf("%d\n", cur->data);  
    cur = cur->next;  
}  
printf("fertig\n");  
} // end main
```



# Zeigerkonzept in C - Anwendungsbeispiel

```
typedef struct ListNode {  
    int data;  
    struct ListNode *next;  
} node_t;
```

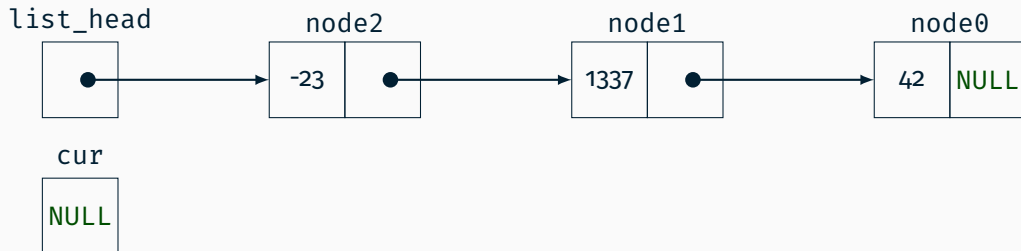
```
// [...]  
node_t *cur = list_head;  
while(cur != NULL) {  
    printf("%d\n", cur->data);  
    cur = cur->next;  
}  
printf("fertig\n");  
} // end main
```



# Zeigerkonzept in C - Anwendungsbeispiel

```
typedef struct ListNode {  
    int data;  
    struct ListNode *next;  
} node_t;
```

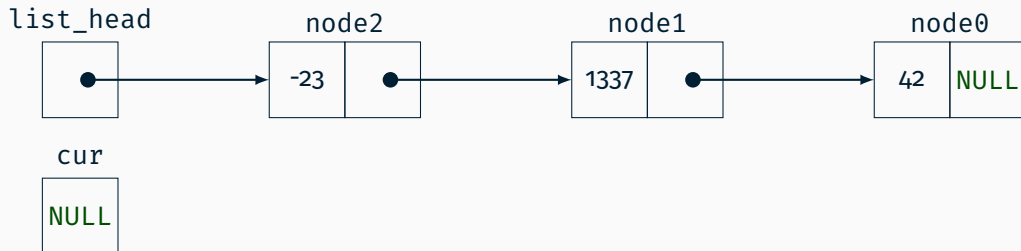
```
// [...]  
  
node_t *cur = list_head;  
while(cur != NULL) {  
    printf("%d\n", cur->data);  
    cur = cur->next;  
}  
printf("fertig\n");  
} // end main
```



# Zeigerkonzept in C - Anwendungsbeispiel

```
typedef struct ListNode {  
    int data;  
    struct ListNode *next;  
} node_t;
```

```
// [...]  
node_t *cur = list_head;  
while(cur != NULL) {  
    printf("%d\n", cur->data);  
    cur = cur->next;  
}  
printf("fertig\n");  
} // end main
```



- Elementare Bausteine für die Modularisierung
- Unterscheidung zwischen
  - Funktions**deklaration**  
→ Schnittstelle (Parameter, Name, Rückgabewert)
  - Funktions**definition**  
→ Implementierung
  - Funktionen müssen vor Verwendung deklariert sein
  - Def. und Dekl. können gemeinsam erfolgen

```
// Funktionsdeklaration  
// (Parameternamen optional)  
float func1(int);  
float func2(int a);  
  
// Funktionsdekl. und -definition  
// (keine Parameter)  
void func3(void) {  
    // [...]  
}  
  
int main(void) {  
    float f1 = func1(2);  
    float f2 = func2(2);  
    func3();  
}  
  
// Funktionsdefinition  
float func1(int a) {/* ... */}  
  
// Funktionsdefinition  
float func2(int a) {/* ... */}
```

- Elementare Bausteine für die Modularisierung
- Unterscheidung zwischen
  - Funktions**deklaration**  
→ Schnittstelle (Parameter, Name, Rückgabewert)
  - Funktions**definition**  
→ Implementierung
  - Funktionen müssen vor Verwendung deklariert sein
  - Def. und Dekl. können gemeinsam erfolgen
- *Offene* Definition (leere Parameterliste)
  - erlaubt Aufruf mit beliebigen Parametern  
→ i.d.R. schlechter Programmierstil!
  - `void` erzwingt parameterlose Funktion

```
// Funktionsdeklaration  
// (Parameternamen optional)  
float func1(int);  
float func2(int a);  
  
// Funktionsdekl. und -definition  
// (keine Parameter)  
void func3(void) {  
    // [...]  
}  
  
int main(void) {  
    float f1 = func1(2);  
    float f2 = func2(2);  
    func3();  
}  
  
// Funktionsdefinition  
float func1(int a) {/* ... */}  
  
// Funktionsdefinition  
float func2(int a) {/* ... */}
```

- Funktionsaufrufe sind *call by value*

## Call by Value

```
void swap(int x, int y) {  
    int tmp = y;  
    y = x;  
    x = tmp;  
}  
  
int main(void) {  
    int a = 42;  
    int b = 1337;  
  
    // a = 42; b = 1337  
    swap(a, b);  
    // a = 42; b = 1337  
}
```



- Funktionsaufrufe sind *call by value*
- *call by reference* kann mit Zeigern nachgebaut werden

## Call by Value

```
void swap(int x, int y) {  
    int tmp = y;  
    y = x;  
    x = tmp;  
}  
  
int main(void) {  
    int a = 42;  
    int b = 1337;  
  
    // a = 42; b = 1337  
    swap(a, b);  
    // a = 42; b = 1337  
}
```

## Call by Reference

```
void swap(int* x, int* y) {  
    int tmp = *y;  
    *y = *x;  
    *x = tmp;  
}  
  
int main(void) {  
    int a = 42;  
    int b = 1337;  
  
    // a = 42; b = 1337  
    swap(&a, &b);  
    // a = 1337; b = 42  
}
```

# Formatierte Ausgabe in C

- Ausgabe durch printf()
- printf() erwartet Formatstring und Variablen
  - Formatstring: String mit Platzhaltern
  - Platzhalter: Enthalten Formatierungsanweisungen
  - Variablen: Werden gemäß der Platzhalter eingefügt

```
#include <stdio.h>

int main(void) {
    printf("char: %c", 'A');
    // --> "char: A"

    printf("%s %s!", "Hello", "World");
    // --> "Hello World!"

    printf("pi ist genau %i", 3);
    // --> "pi ist genau 3"

    printf("pi ist %.2f", 3.1415);
    // --> "pi ist 3.14"

    int tmp;
    printf("address of tmp: %p", &tmp);
    // --> "adress of tmp: 0x7ff..."

    int a[256];
    size_t len = sizeof(a);
    printf("a ist %zu bytes", len);
    // --> "a ist 1024 bytes"
}
```

# Formatierte Ausgabe in C

- Ausgabe durch `printf()`
- `printf()` erwartet Formatstring und Variablen
  - Formatstring: String mit Platzhaltern
  - Platzhalter: Enthalten Formatierungsanweisungen
  - Variablen: Werden gemäß der Platzhalter eingefügt
- Platzhalter
  - `%c` ein Zeichen
  - `%s` eine `'\0'`-terminierte Zeichenkette
  - `%i` einen `int`-Wert; mit führenden 0en: `%05i`
  - `%u` einen `unsigned int`-Wert
  - `%x` einen `unsigned int`-Wert als Hex
  - `%p` Zeiger
  - `%f` Gleitkomma-Wert; mit 2 Nachkommastellen: `%.2f`
- Längenmodifikatoren
  - `hh/h` für `char/short` (`%hu` für `unsigned short`)
  - `l/ll` für `long/long long` (`%li` für `long`)
  - `z` für `size_t`

```
#include <stdio.h>

int main(void) {
    printf("char: %c", 'A');
    // --> "char: A"

    printf("%s %s!", "Hello", "World");
    // --> "Hello World!"

    printf("pi ist genau %i", 3);
    // --> "pi ist genau 3"

    printf("pi ist %.2f", 3.1415);
    // --> "pi ist 3.14"

    int tmp;
    printf("address of tmp: %p", &tmp);
    // --> "adress of tmp: 0x7ff..."

    int a[256];
    size_t len = sizeof(a);
    printf("a ist %zu bytes", len);
    // --> "a ist 1024 bytes"
}
```

## Statische Allokation

- globale Variablen
- lokale `static` Variablen

→ Bedarf bekannt zur Compilezeit

→ Lebensdauer: Programmausführung

```
#include <stdio.h>

int g = 23; // globale Variable

int f(void) {
    // lokale auto Variable (Stack)
    // ('auto' ist optional)
    /* auto */ int a = 42;
    return a++;
}

int f2(void) {
    // lokale static Variable
    static int a = 42;
    return a++;
}

int main(void) {
    printf("%d, %d, %d", g, f(), f2());
    // --> 23, 42, 42

    printf("%d, %d, %d", g, f(), f2());
    // --> 23, 42, 43 (!!)
```

## Statische Allokation

- globale Variablen
- lokale static Variablen

- Bedarf bekannt zur Compilezeit
- Lebensdauer: Programmausführung

## Dynamische Allokation → Laufzeit

- lokale auto-Variablen (Stack)
  - Bedarf bekannt zur Laufzeit
  - Lebensdauer: Funktion
- explizit angeforderter Speicher (Heap)
  - Bedarf bekannt zur Laufzeit
  - Lebensdauer: explizit
  - malloc(), free()

```
#include <stdio.h>

int g = 23; // globale Variable

int f(void) {
    // lokale auto Variable (Stack)
    // ('auto' ist optional)
    /* auto */ int a = 42;
    return a++;
}

int f2(void) {
    // lokale static Variable
    static int a = 42;
    return a++;
}

int main(void) {
    printf("%d, %d, %d", g, f(), f2());
    // --> 23, 42, 42

    printf("%d, %d, %d", g, f(), f2());
    // --> 23, 42, 43 (!! )
}
```

# Dynamische Speicherverwaltung

`void *malloc(size_t size)`

- Anforderung von size Bytes
- Rückgabe: Zeiger auf Speicher
- Fehlerfall: Rückgabe von `NULL`
- Speicher ist nicht initialisiert!

`void free(void *mem)`

- Rückgabe des Speichers
- kein Rückgabewert

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

```
typedef struct date {  
    int day;  
    int month;  
    int year;  
} date_t;
```

```
int main(void) {  
    date_t *t = malloc(sizeof(date_t));  
    if(t == NULL) {  
        // Fehlerbehandlung  
    }  
  
    // ...  
  
    free(t);  
}
```

# Dynamische Speicherverwaltung

`void *malloc(size_t size)`

- Anforderung von size Bytes
- Rückgabe: Zeiger auf Speicher
- Fehlerfall: Rückgabe von `NULL`
- Speicher ist nicht initialisiert!

`void free(void *mem)`

- Rückgabe des Speichers
- kein Rückgabewert

## Achtung

- genau ein mal `free()`  
→ *double free* oder Speicherleck
- kein Nutzen nach `free()`  
→ *use after free*

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

```
typedef struct date {  
    int day;  
    int month;  
    int year;  
} date_t;
```

```
int main(void) {  
    date_t *t = malloc(sizeof(date_t));  
    if(t == NULL) {  
        // Fehlerbehandlung  
    }  
  
    // ...  
  
    free(t);  
}
```

# Dynamische Speicherverwaltung

```
void *memset(void *mem, int val, size_t n)
```

- initialisiert ersten n bytes von mem mit dem Byte val

```
void *calloc(size_t nmemb, size_t size)
```

- allokiert Speicher für nmemb Elemente der Größe size
- initialisiert den Speicher mit 0
- Fehlerfall: Rückgabe von NULL

```
void *realloc(void *mem, size_t size)
```

- allokiert size Bytes
- kopiert Inhalte von mem (soweit möglich)
- gibt mem frei
- Fehlerfall: Rückgabe von NULL; mem nicht gefree()ed

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

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

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

```
    int *a = malloc(100 * sizeof(int));  
    int *b = calloc(100, sizeof(int));
```

```
    if(a == NULL || b == NULL) {
```

```
        // Fehlerbehandlung
```

```
    }
```

```
    memset(a, 0, 100 * sizeof(int));
```

```
    // a und b mit 0 initialisiert
```

```
    // ...
```

```
    // 100 Elemente sind zu wenig
```

```
    a = realloc(a, 150 * sizeof(int));
```

```
    if(a == NULL) {
```

```
        // Fehlerbehandlung
```

```
    }
```

```
    // ...
```

```
    free(a); free(b);
```

```
}
```



## 1 C Grundlagen

1.1 Hello World

1.2 Datentypen

1.3 Operatoren

1.4 Kontrollstrukturen

1.5 Einschub: Übersetzen

## 2 C Besonderheiten

2.1 Zeiger

2.2 Funktionen

2.3 Formatierte Ausgabe

2.4 Speicherverwaltung

## 3 Fortgeschrittenes C

3.1 Typumwandlungen

3.2 Zeigerarithmetik

3.3 Zeichenketten und Ein-/Ausgabe

- Operationen werden mit der größten beteiligten Größe, **mind.** jedoch `int`, berechnet
  - `short` und `char` werden implizit erweitert
  - Ergebnis wird auf den Zieldatentyp abgeschnitten/erweitert
- Fließkommaoperationen werden **mindestens** mit `double` berechnet
- Fließkommatypen gelten als *größer* als Ganzzahltypen
- `unsigned`-Typen gelten als größer als `signed`-Typen

```
char a = 100, b = 3; // 1 Byte [-128;127]
char res;           // 1 Byte [-128;127]
long c = 4;         // 8 Byte ...
```

$$\underbrace{\text{res}}_{\text{char: 75}} = \underbrace{\text{a}}_{\text{int: 100}} * \underbrace{\text{b}}_{\text{int: 3}} / \underbrace{\text{c}}_{\text{long: 4}} ;$$

$$\underbrace{\hspace{10em}}_{\text{int: 300}}$$

$$\underbrace{\hspace{10em}}_{\text{long: 300}}$$

$$\underbrace{\hspace{10em}}_{\text{long: 75}}$$

- Operationen werden mit der größten beteiligten Größe, **mind.** jedoch `int`, berechnet
  - `short` und `char` werden implizit erweitert
  - Ergebnis wird auf den Zieldatentyp abgeschnitten/erweitert
- Fließkommaoperationen werden **mindestens** mit `double` berechnet
- Fließkommatypen gelten als *größer* als Ganzzahltypen
- `unsigned`-Typen gelten als größer als `signed`-Typen

```
char a = 100, b = 3; // 1 Byte [-128;127]
char res;           // 1 Byte [-128;127]
float c = 4.0f;      // ...
```

$$\underbrace{\text{res}}_{\text{char: 75}} = \underbrace{\underbrace{\text{a}}_{\text{int: 100}} * \underbrace{\text{b}}_{\text{int: 3}}}_{\text{int: 300}} / \underbrace{\text{c}}_{\text{double: 4.0}};$$
$$\underbrace{\underbrace{\text{double: 300.0}}}_{\text{double: 75.0}}$$

- Operationen werden mit der größten beteiligten Größe, **mind.** jedoch `int`, berechnet
  - `short` und `char` werden implizit erweitert
  - Ergebnis wird auf den Zieldatentyp abgeschnitten/erweitert
- Fließkommaoperationen werden **mindestens** mit `double` berechnet
- Fließkommatypen gelten als *größer* als Ganzzahltypen
- `unsigned`-Typen gelten als größer als `signed`-Typen

```
int s = -1, res;           // 4 Byte [-2147483648;2147483647]
unsigned u = 1;            // 4 Byte [0;4294967295]
```

$$\underbrace{\text{res}}_{\text{int: } 0 \text{ (false)}} = \underbrace{\text{s}}_{\text{int: } -1} < \underbrace{\text{u}}_{\text{unsigned: } 1};$$

$$\underbrace{\text{unsigned: } 4294967295}_{\text{unsigned: } 0 \text{ (false)}}$$

- Operanden können gezielt in einen anderen Typen umgewandelt werden (<type>)
- Keine Laufzeitüberprüfung – der Compiler macht einfach!

```
int s = -1, res;           // 4 Byte [-2147483648;2147483647]
unsigned u = 1;           // 4 Byte [0;4294967295]
```

```
    res      =  s  < (int) u;
int: 1 (true)  int: -1  int: 1
               └──────────┘
                   int: 1 (true)
```

## Arrays

- Arrays sind Zeiger auf erstes Element
  - \* und [ ] geht beides
  - können anderen Zeigern zugewiesen werden
  - können als Zeiger übergeben werden
  - Arrays werden *by-reference* übergeben
- Unterschied zu *klassischen* Zeigern
  - Zeiger nicht explizit gespeichert  
→ Compiler fügt Adresse ein
  - &-Operator ergibt wieder den Zeiger
  - `sizeof( )` gibt Arraygröße statt Zeigergröße zurück

```
#include <stdio.h>

int f(int *i) {
    return *i;
}

int main(void) {
    int a[4] = {1, 2, 3, 4};
    printf("%d", a[0]); // --> 1
    printf("%d", *a);   // --> 1

    int *b = a;
    printf("%d", *b);   // --> 1

    printf("%d", f(a)); // --> 1
    printf("%d", f(b)); // --> 1

    printf("%p", a); // 0x7f...9af0
    printf("%p", b); // 0x7f...9af0

    printf("%p", &a); // 0x7f...9af0
    printf("%p", &b); // 0x7f...9ae8
}
```

## Funktionszeiger

- Zeiger auf Funktionen
- Syntax:  
    <ret type> (\*<name>)(<param types>)

## Typische Anwendungsbeispiele

- als Übergabeparameter an Funktionen
  - als Feld in Strukturen
- z.B. Vergleichsfunktion für Sortierungsfunktion
- z.B. Callback-Fkt. für Ereignisse (onKeyPress( ))
- z.B. Startfunktion für Threads

```
#include <stdio.h>

void mylog(void (*f)(char *), char *msg) {
    print_timestamp();
    f(msg);
}

void error(char *msg) {
    set_color(RED);
    printf("ERROR: %s\n", msg);
    reset_color();
}

void info(char *msg) {
    printf("INFO: %s\n", msg);
}

int main(void) {
    mylog(info, "app started");
    // --> "<time>: INFO: app started"
    mylog(&error, "error code 42");
    // --> "<time>: ERROR: error code 42"
}
```

## ■ `const int*`

- ein Zeiger auf einen konstanten `int`-Wert  
→ Wert nicht veränderbar

## ■ `int* const`

- ein konstanter Zeiger auf einen `int`-Wert  
→ Zeiger nicht veränderbar

```
int main(void) {  
    int a = 42, b = 23;  
  
    const int* p1 = &a;  
    int* const p2 = &a;  
  
    // *p1 = 43; !! Compilerfehler !!  
    p1 = &b;  
  
    *p2 = 23; // a --> 23  
    // p2 = &b; !! Compilerfehler !!  
}
```



## Zeigerarithmetik

- Mit Zeigern kann man rechnen
- Verhalten hängt von der Größe des Ziels ab

```
#include <stdio.h>

int main(void) {
    // Zeiger auf 32bit (4 Byte) Ganzzahl
    int *ptr1 = /* ... */;
    printf("%p", ptr1);      // 0x1000
    printf("%p", ptr1 + 1);  // 0x1004 (1*4Byte)
    printf("%p", ptr1 + 5);  // 0x1014 (5*4Byte)

    // Zeiger auf 64bit (8 Byte) Ganzzahl
    long *ptr2 = /* ... */;
    printf("%p", ptr2);      // 0x1000
    printf("%p", ptr2 + 1);  // 0x1008 (1*8Byte)
    printf("%p", ptr2 + 5);  // 0x1028 (5*8Byte)
}
```

## Zeigerarithmetik

- Mit Zeigern kann man rechnen
- Verhalten hängt von der Größe des Ziels ab
- Array-Zugriff per [ ] äquivalent

```
#include <stdio.h>

int main(void) {
    // Zeiger auf 32bit (4 Byte) Ganzzahl
    int *ptr1 = /* ... */;
    printf("%p", ptr1);      // 0x1000
    printf("%p", ptr1 + 1);  // 0x1004 (1*4Byte)
    printf("%p", ptr1 + 5);  // 0x1014 (5*4Byte)

    // Zeiger auf 64bit (8 Byte) Ganzzahl
    long *ptr2 = /* ... */;
    printf("%p", ptr2);      // 0x1000
    printf("%p", ptr2 + 1);  // 0x1008 (1*8Byte)
    printf("%p", ptr2 + 5);  // 0x1028 (5*8Byte)

    int a[4] = {1,2,3,4};
    int *b = a;
    printf("%p|%i", &a[1], a[1]); // 0x1004|2
    printf("%p|%i", &b[1], b[1]);  // 0x1004|2
    printf("%p|%i", a+1, *(a+1)); // 0x1004|2
}
```

## Zeigerarithmetik

- funktioniert mit beliebigen Strukturen
- Zeiger können auch auf Zeiger zeigen

```
#include <stdio.h>

typedef struct date {
    int year;
    int month;
    int day;
} date_t;

int main(void) {
    date_t *ptr = /* ... */;
    date_t **ptrptr = &ptr;

    printf("%p", ptr);      // 0x1000
    printf("%p", ptr + 1);  // 0x100c

    // Annahme: 64-bit System
    printf("%p", ptrptr);    // 0x2000
    printf("%p", ptrptr + 1); // 0x2008
}
```

# Zeigerarithmetik – Beispiel

```
struct test {  
    long num; // 8 Byte  
    void *ptr; // 8 Byte (64-bit System)  
}
```

struct test \*arr .....

6	ptr = ●
	num = 9
5	ptr = ●
	num = 8
4	ptr = ●
	num = 7
3	ptr = ●
	num = 6
2	ptr = ●
	num = 5
1	ptr = ●
	num = 4
0	ptr = ●
	num = 3

# Zeigerarithmetik – Beispiel

```
struct test {  
    long num; // 8 Byte  
    void *ptr; // 8 Byte (64-bit System)  
}
```

```
struct test *arr  
    &arr[0]
```

6	ptr = ●
	num = 9
5	ptr = ●
	num = 8
4	ptr = ●
	num = 7
3	ptr = ●
	num = 6
2	ptr = ●
	num = 5
1	ptr = ●
	num = 4
0	ptr = ●
	num = 3

# Zeigerarithmetik – Beispiel

```
struct test {  
    long num; // 8 Byte  
    void *ptr; // 8 Byte (64-bit System)  
}
```

```
struct test *arr  
    &arr[0] - - - -  
    &arr[2]
```

6	ptr = ●	num = 9
5	ptr = ●	num = 8
4	ptr = ●	num = 7
3	ptr = ●	num = 6
2	ptr = ●	num = 5
1	ptr = ●	num = 4
0	ptr = ●	num = 3

# Zeigerarithmetik – Beispiel

```
struct test {  
    long num; // 8 Byte  
    void *ptr; // 8 Byte (64-bit System)  
}
```

```
struct test *arr  
    &arr[0]  
    &arr[2]  
    arr + 3
```

6	ptr = ●	num = 9
5	ptr = ●	num = 8
4	ptr = ●	num = 7
3	ptr = ●	num = 6
2	ptr = ●	num = 5
1	ptr = ●	num = 4
0	ptr = ●	num = 3

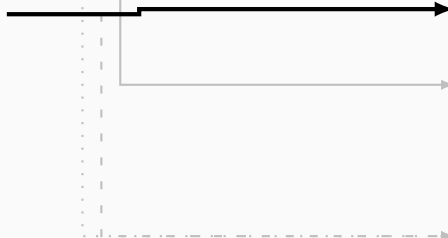


# Zeigerarithmetik – Beispiel

```
struct test {  
    long num; // 8 Byte  
    void *ptr; // 8 Byte (64-bit System)  
}
```

```
struct test *arr  
    &arr[0]  
    &arr[2]  
    arr + 3  
    ((long*) arr) + 7
```

6	ptr = ●	num = 9
5	ptr = ●	num = 8
4	ptr = ●	num = 7
3	ptr = ●	num = 6
2	ptr = ●	num = 5
1	ptr = ●	num = 4
0	ptr = ●	num = 3



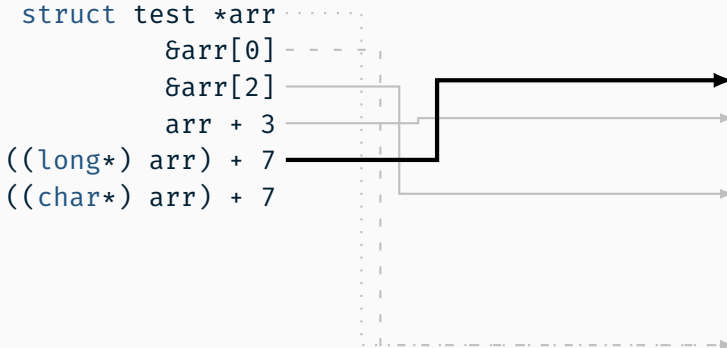


# Zeigerarithmetik – Beispiel

```
struct test {  
    long num; // 8 Byte  
    void *ptr; // 8 Byte (64-bit System)  
}
```

```
struct test *arr  
    &arr[0]  
    &arr[2]  
    arr + 3  
((long*) arr) + 7  
((char*) arr) + 7
```

6	ptr = ●	num = 9
5	ptr = ●	num = 8
4	ptr = ●	num = 7
3	ptr = ●	num = 6
2	ptr = ●	num = 5
1	ptr = ●	num = 4
0	ptr = ●	num = 3



# Zeigerarithmetik – Beispiel

```
struct test {  
    long num; // 8 Byte  
    void *ptr; // 8 Byte (64-bit System)  
}
```

```
struct test *arr  
    &arr[0]  
    &arr[2]  
    arr + 3  
((long*) arr) + 7  
((char*) arr) + 7  
    &(5[arr])
```

6	ptr = ●	num = 9
5	ptr = ●	num = 8
4	ptr = ●	num = 7
3	ptr = ●	num = 6
2	ptr = ●	num = 5
1	ptr = ●	num = 4
0	ptr = ●	num = 3

# Zeigerarithmetik – Beispiel

```
struct test {  
    long num; // 8 Byte  
    void *ptr; // 8 Byte (64-bit System)  
}
```

```
struct test *arr  
    &arr[0]  
    &arr[2]  
    arr + 3  
((long*) arr) + 7  
((char*) arr) + 7  
    &(5[arr])  
    &(5[5])
```

6	ptr = ●	num = 9
5	ptr = ●	num = 8
4	ptr = ●	num = 7
3	ptr = ●	num = 6
2	ptr = ●	num = 5
1	ptr = ●	num = 4
0	ptr = ●	num = 3

# Zeigerarithmetik – Beispiel

```
struct test {  
    long num; // 8 Byte  
    void *ptr; // 8 Byte (64-bit System)  
}
```

```
struct test *arr  
    &arr[0]  
    &arr[2]  
    arr + 3  
((long*) arr) + 7  
((char*) arr) + 7  
    &(5[arr])  
    &(5[5])  
(void*) arr
```

6	ptr =	•
	num = 9	
5	ptr =	•
	num = 8	
4	ptr =	•
	num = 7	
3	ptr =	•
	num = 6	
2	ptr =	•
	num = 5	
1	ptr =	•
	num = 4	
0	ptr =	•
	num = 3	

# Zeigerarithmetik – Beispiel

```
struct test {  
    long num; // 8 Byte  
    void *ptr; // 8 Byte (64-bit System)  
}
```

```
struct test *arr
```

```
&arr[0]
```

```
&arr[2]
```

```
arr + 3
```

```
((long*) arr) + 7
```

```
((char*) arr) + 7
```

```
&(5[arr])
```

```
&(5[5])
```

```
(void*) arr
```

```
((void*) arr) + 4
```

6	ptr =	•
	num = 9	
5	ptr =	•
	num = 8	
4	ptr =	•
	num = 7	
3	ptr =	•
	num = 6	
2	ptr =	•
	num = 5	
1	ptr =	•
	num = 4	
0	ptr =	•
	num = 3	



# Zeigerarithmetik – Beispiel

```
struct test {  
    long num; // 8 Byte  
    void *ptr; // 8 Byte (64-bit System)  
}
```

```
struct test *arr
```

```
&arr[0]
```

```
&arr[2]
```

```
arr + 3
```

```
((long*) arr) + 7
```

```
((char*) arr) + 7
```

```
&(5[arr])
```

```
&(5[5])
```

```
(void*) arr
```

```
((void*) arr) + 4
```

6	ptr =	•
	num = 9	
5	ptr =	•
	num = 8	
4	ptr =	•
	num = 7	
3	ptr =	•
	num = 6	
2	ptr =	•
	num = 5	
1	ptr =	•
	num = 4	
0	ptr =	•
	num = 3	

- **Recap:** String sind '`\0`'-terminierte `char`-Arrays
- Umgang mit Strings ist umständlich

```
#include <stdio.h>

int main(void) {
    char s1[] = "test";
    char s2[] = "test";

    /* !! Häufige Fehler !! */
    // vergl. Zeiger nicht String!
    if(s1 == s2) {
        printf("gleich\n");
    } else {
        printf("ungleich\n");
    }
    // --> ungleich

    // kopiert Zeiger nicht String!
    char *cpy = s2;

    // Zeigeraddition nicht def.!
    // Compilerfehler
    char *new = s1 + s2;
}
```

falsch - nicht nachmachen

- **Recap:** String sind `'\0'`-terminierte `char`-Arrays
- Umgang mit Strings ist umständlich
- Strings vergleichen:
  - in Schleife jedes Zeichen vergleichen
  - Schleife beenden sobald ein `'\0'` auftritt
- String kopieren:
  - Länge des Strings bestimmen
  - neuen Speicher anlegen (Stack o. `malloc()`)
  - jedes Zeichen kopieren
- String konkatenieren:
  - Länge beider Strings bestimmen
  - neuen Speicher anlegen (Stack o. `malloc()`)
  - jedes Zeichen des ersten Strings kopieren
  - jedes Zeichen des zweiten Strings kopieren

```
#include <stdio.h>

int main(void) {
    char s1[] = "test";
    char s2[] = "test";

    /* !! Häufige Fehler !! */
    // vergl. Zeiger nicht String!
    if(s1 == s2) {
        printf("gleich\n");
    } else {
        printf("ungleich\n");
    }
    // --> ungleich

    // kopiert Zeiger nicht String!
    char *cpy = s2;

    // Zeigeraddition nicht def.!
    // Compilerfehler
    char *new = s1 + s2;
}
```

falsch - nicht nachmachen



- Stringoperationen mittels libc-Funktionen
  - Länge: `size_t strlen(const char*)`
  - Vergleich: `int strcmp(const char*, const char*)`
  - Anhängen: `char* strcat(char*, const char*)`
  - Kopieren: `char* strcpy(char*, const char*)`
- Arbeiten zeichenweise bis `'\0'`
  - Sicherstellen das genügend Speicher vorhanden ist!

## ■ Stringoperationen mittels libc-Funktionen

- Länge: `size_t strlen(const char*)`
- Vergleich: `int strcmp(const char*, const char*)`
- Anhängen: `char* strcat(char*, const char*)`
- Kopieren: `char* strcpy(char*, const char*)`

## ■ Arbeiten zeichenweise bis `'\0'`

→ Sicherstellen das genügend Speicher vorhanden ist!

## ■ Varianten mit Längenbegrenzung:

- Vergleich: `int strncmp(const char*, const char*, size_t)`
- Kopieren: `char* strncpy(char*, const char*, size_t)`
- Anhängen: `char* strncat(char*, const char*, size_t)`

- Stringoperationen mittels libc-Funktionen
  - Länge: `size_t strlen(const char*)`
  - Vergleich: `int strcmp(const char*, const char*)`
  - Anhängen: `char* strcat(char*, const char*)`
  - Kopieren: `char* strcpy(char*, const char*)`
- Arbeiten zeichenweise bis `'\0'`
  - Sicherstellen das genügend Speicher vorhanden ist!
- Varianten mit Längenbegrenzung:
  - Vergleich: `int strncmp(const char*, const char*, size_t)`
  - Kopieren: `char* strncpy(char*, const char*, size_t)`
  - Anhängen: `char* strncat(char*, const char*, size_t)`

## **strcpy/strncpy & strcat/strncat vermeiden!**

`strcpy, strcat`: Schreiben ggf. über Zielarray raus

`strncpy, strncat`: Behandlung der Längenbeschränkung unlogisch

# Zeichenketten (besser) formatieren

```
int snprintf(char *str, size_t size, const char *format, ...);
```

- Sichere Benutzung durch size
- Sinnvolle Behandlung bei Längenbeschr.  
→ immer `'\0'` terminiert
- Längenbeschränkung erkennbar  
→ Rückgabewert  $\geq$  size

```
#include <stdio.h>

int main(void) {
    char s[64];
    char *fmt = "pi ist %.2f (als int: %d)";

    int ret = snprintf(s, 64, fmt, 3.1415, 3);
    if(ret >= 64) {
        printf("String truncated!\n");
        printf("%d chars omitted\n", ret-63);
    }

    printf("%s\n", s);
    // "pi ist 3.14 (als int: 3)"
}
```

# Zeichenketten (besser) formatieren

```
int snprintf(char *str, size_t size, const char *format, ...);
```

- Sichere Benutzung durch size
- Sinnvolle Behandlung bei Längenbeschr.  
→ immer `'\0'` terminiert
- Längenbeschränkung erkennbar  
→ Rückgabewert  $\geq$  size
- **Achtung:** Formatstring darf nicht „von außen“ kommen  
→ gilt auch für `printf()`!

```
#include <stdio.h>

int main(void) {
    char s[64];
    char *fmt = "pi ist %.2f (als int: %d)";

    int ret = snprintf(s, 64, fmt, 3.1415, 3);
    if (ret >= 64) {
        printf("String truncated!\n");
        printf("%d chars omitted\n", ret-63);
    }

    printf("%s\n", s);
    // "pi ist 3.14 (als int: 3)"
}
```