Betriebssysteme

Grundlagen C & Pointer

Fachbereich
Angewandte Informatik
AI

Agenda

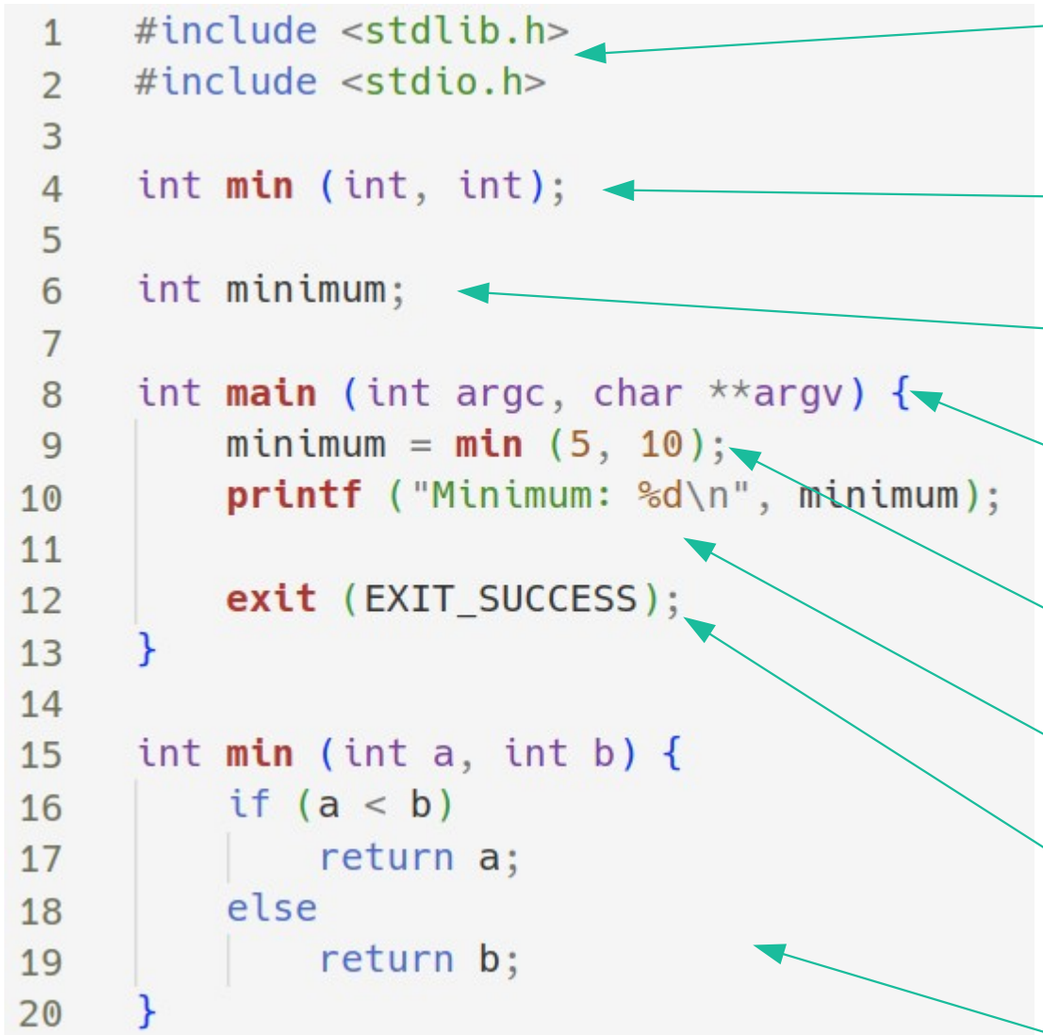
- Grundlagen
  - Anatomie eines C-Programms
  - Programm im Speicher
  - Vom C-Quelltext zur Binärdatei
  - Paar weitere Dinge (z.B. Datentypen)

- Variablen im Speicher
  - “Normale” Variablen vs. Pointer Variablen
  - Beispiel
    - Verkettete Liste
    - FIFO-Liste

- Funktionen und deren Signaturen

- Einführung in die Programmiersprache C
  - http://www.informatik.htw-dresden.de/~bruns/lehrheft.pdf

Grundlagen



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

int min (int, int);

int minimum;

int main (int argc, char **argv) {
    minimum = min (5, 10);
    printf ("Minimum: %d\n", minimum);

    exit (EXIT_SUCCESS);
}

int min (int a, int b) {
    if (a < b)
        return a;
    else
        return b;
}
```

Präprozessor-Statement (beginnt mit #)
Einbinden externen Programmcodes (z.B. aus der C-Standardbibliothek).
(hier: <stdio.h> enthält printf(); <stdlib.h> definiert exit() und EXIT_SUCCESS)
Hinweis: auch hier man-Pages (z.B.: `man stdio.h`)

Prototyp Deklaration (nötig da Funktion benutzt wird bevor sie definiert wird)
- Die Funktion heißt **min**, besitzt den Rückgabetyp **int** und erhält zwei Parameter vom Typ **int**

Globale Variable
- Mit dem Namen **minimum** und vom Typ **int**

Definition des Hauptprogramms (main)
- Das Hauptprogramm gibt immer einen Wert vom Typ **int** zurück
- Der erste Parameter gibt an wie viele Argumente übergeben wurden
- Der zweite Parameter zeigt auf diese Argumente im Hauptspeicher

Funktionsaufruf
- Aufruf der Funktion **min**, mit zwei **int**-Parametern (5 und 10)
- Speichern des Rückgabewertes in der Variable **minimum**

Ausgabe
- Ausgabe der kleineren Zahl (minimum) auf dem Bildschirm

Beenden des Hauptprogramms
- EXIT_SUCCESS bzw. EXIT_FAILURE sind Plattformabhängig auf den korrekten Wert (z.B. Linux: 0 = SUCCESS, 1 = FEHLER) definiert

Definition der Funktion min
- Die Funktion erhält zwei Nummern vom Typ int und gibt die kleinere der beiden zurück

Programmlayout im Speicher

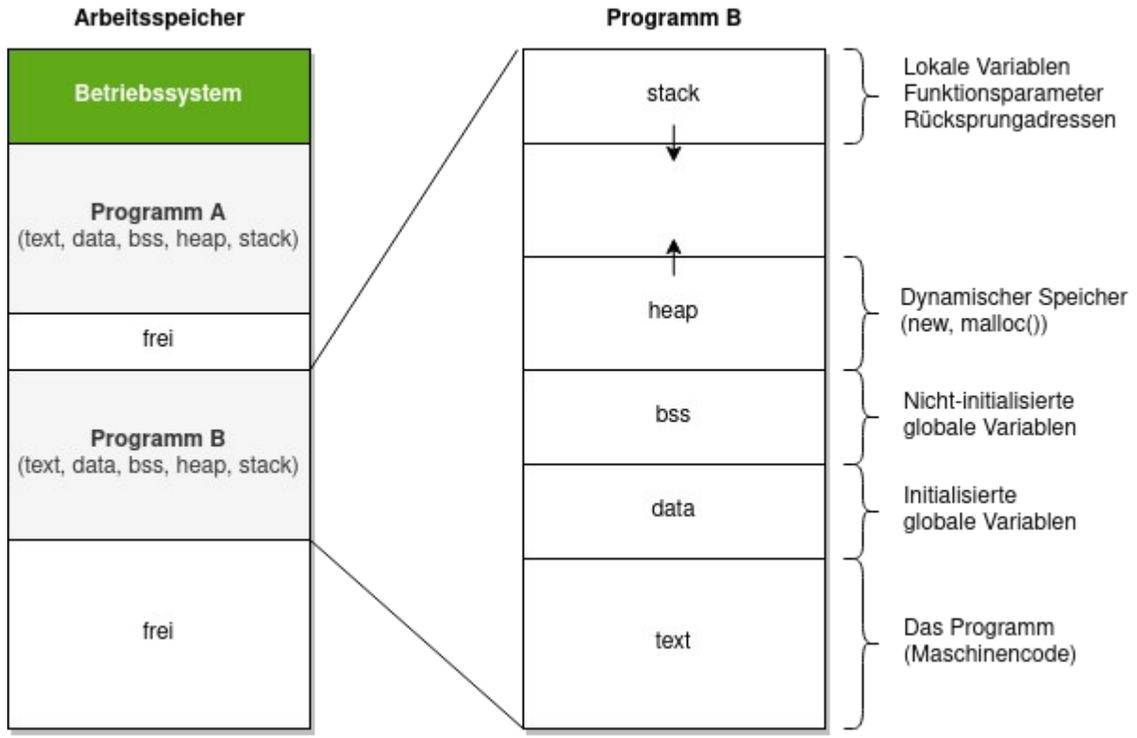
- Text-Segment
  (Enthält den eigentlichen Programmcode)

- Data-Segment
  (Enthält alle globalen Variablen, die mit einem Wert vorinitialisiert sind)
  - int i = 5; char name[] = “Thomas”;

- BSS-Segment
  (Enthält alle globalen Variablen, die **nicht** mit einem Wert vorinitialisiert sind → Einsparung von Speicher in der Binärdatei)
  - int i; char foo[100000];

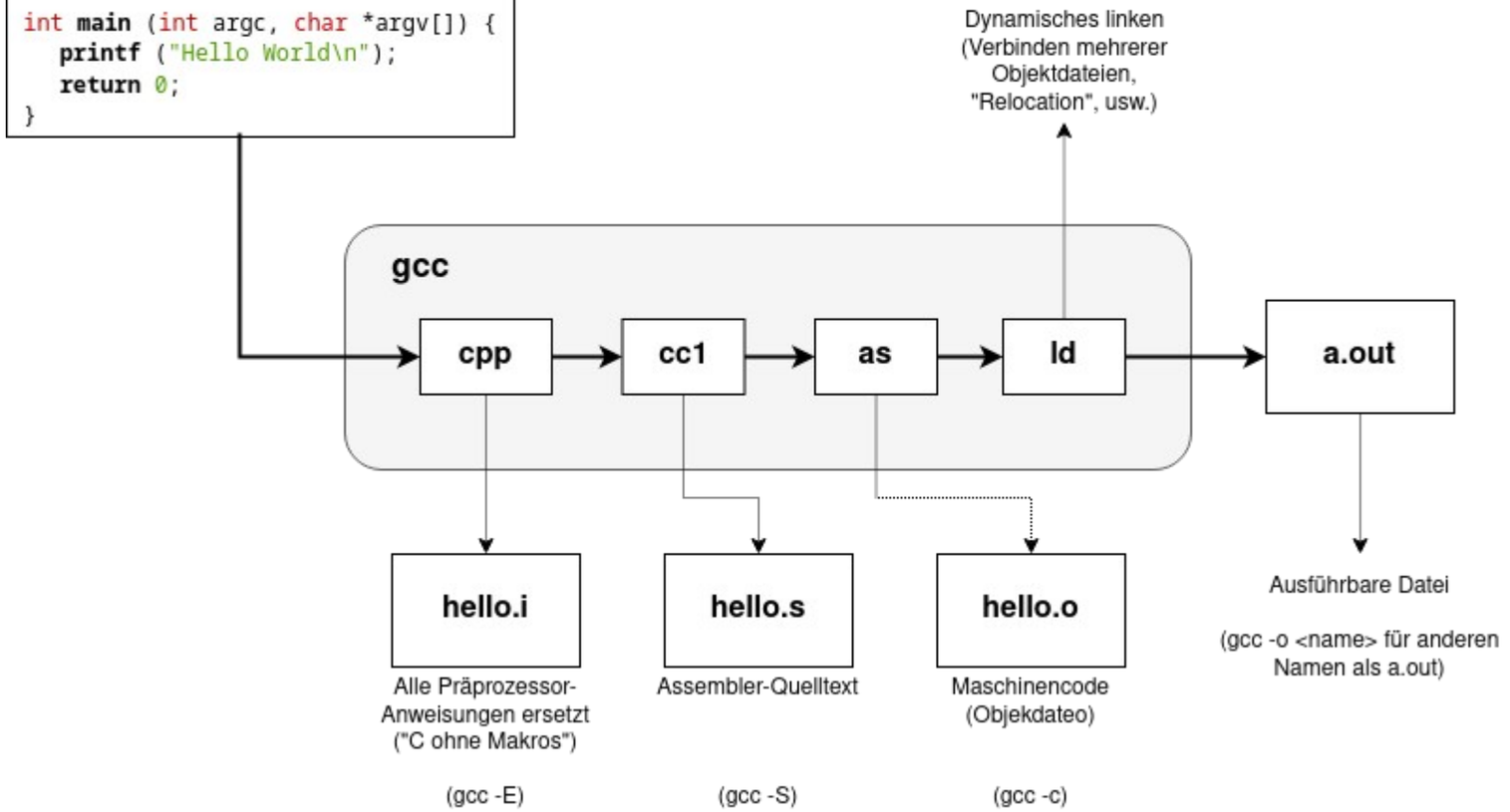
- Stack
  (Enthält alle lokalen Variablen, Funktionsparameter und Rücksprungadressen)

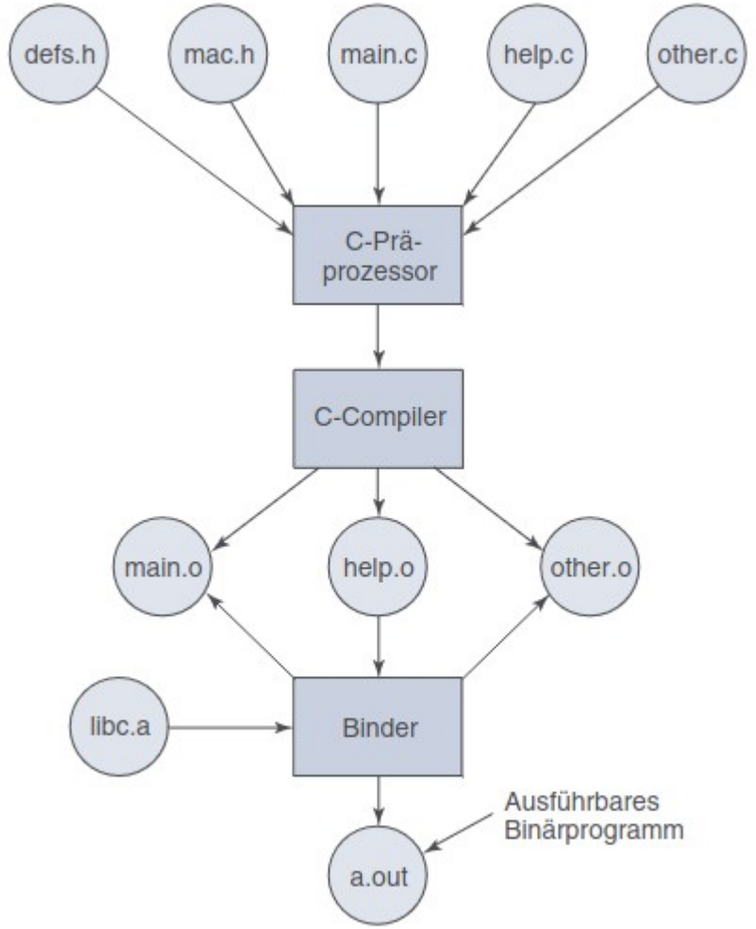
- Heap
  (Dynamischer Speicher. Enthält alle Daten, deren Speicher per malloc() vom Betriebssystem angefordert wurde)



Vom Quellcode zum ausführbaren Programm

```c
#include <stdio.h>

int main (int argc, char *argv[]) {
   printf ("Hello World\n");
   return 0;
}
```
hello.c





Tanenbaum, Fig. 1.30

Beispiel: Architekturabhängiger Assembler-Quellcode

- Programm `minimal_return.c` aus Übung 3 mit `gcc -S` übersetzt
  (Erkennbar: Unterschiedlicher Code für die gleiche Funktionalität (`return 42`))

AMD Ryzen 7

```
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq%rsp, %rbp
    .cfi_def_cfa_register 6
    movl$42, %eax
    popq%rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

Apple M1

```
_main:
        .cfi_startproc
        sub     sp, sp, #16
        .cfi_def_cfa_offset 16
        str     wzr, [sp, #12]
        mov     w0, #42
        add     sp, sp, #16
        ret
        .cfi_endproc
```

C-Präprozessor

- Einbinden von Dateien
  (Inhalt der include-Datei wird vor der Compilierung durch den Präprozessor an der Stelle des include-Statemeitn eingefügt)

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

- Makros (ohne Parameter)
  (Erlauben zum Beispiel die Definition von Konstanten. Überall wo im Programmcode der Name der Konstante (hier PI) geschrieben wird, findet vor der Compilierung eine Ersetzung durch den Wert statt)

```
#define PI 3.14159
```

- Bedingte Ersetzung
  (Falls eine Konstante auf einen bestimmten Wert definiert ist, kann unterschiedlicher (z.B. Architekturabhängiger) Programmcode ausgeführt werden.)

```
#if VERBOSE >= 2
   printf ("Something\n");
#endif
```

```
#ifdef WIN32
    #include <windows.h>
#else
    #include <unistd.h>
#endif
```

- u.v.m, aber nicht Gegenstand der Veranstaltung

Datentypen

- Typen in C
  - **`int`**, **`char`**, **`float`**, **`double`** und **Pointer** (zzgl. signed/unsigned, short/long)
  - Aber auch viele weitere, z.B. `int8_t`, `uint32_t`, `int64_t`, usw.

- Eigene Typen

```
typedef <Typ> <Name>;
```

```
typedef int myint_x;
myint_x a = 5;
```

```
typedef struct foo {
    int a;
    struct foo* next;
} foo_x;
foo_x myfoo = (foo_x *) malloc (sizeof (foo_x));
```

- In C gibt es nur vier native Typen, aber jede Menge über den C-Standard hinaus definierte Typen – das hat diverse Gründe. Hier zum Beispiel einer:

  Der Datentyp **`int`** ist laut C-Standard "mindestens" **16 bit** lang (→ INT_MAX = +32767), kann aber je nach Implementierung/Plattform auch länger sein.

  Durch POSIX definierte Typen, wie z.B. **`unit32_t`** (= **u**nsigned **int**eger **32** bit) erlauben verlässliche Annahmen über die genutzten Typen und deren Länge.

  Durch **POSIX** definierte Datentypen enden auf **`_t`** um die "Vermüllung" des Namensraums zu begrenzen. POSIX empfiehlt, eigene Datentypen nicht auf `_t` enden zu lassen, da der Standard ggf um gleichnamige Typen erweitert werden könnte.
- Ein anderes Beispiel: **`pid_t`** ist ein eigener Typ für die Darstellung von Prozess-IDs (obwohl eigentlich nur ein Integer)

Heap-Speicher

- Speicher vom Heap besorgen
  - Heap Speicher wird vom Programmierer selbst verwaltet und muss mittels malloc() angefordert und mittels free() wieder freigegeben werden!
  - Alles was nicht auf dem Heap, sondern auf dem Stack gespeichert wird, geht nach Beendigung des Code-Block, also z.B. nach Ende der Funktion in der es definiert wurde, verloren

```c
// Beispiel: int-Pointer statt normaler int Variable

int* a = NULL;
a = (int *) malloc (sizeof (int));

// a benutzen
*a = 5;
[…]

// Speicher wieder frei geben
free (a);
```

```c
// Beispiel: Pointer auf Datenstruktur (z.B. List-Node)

struct foo {
     int bar;
     char baz;
};

[…]

struct foo* myFoo = NULL;

myFoo = (struct foo *) malloc (sizeof (struct foo));

myFoo->bar = 5;
myFoo->baz = 'a';

free (myFoo);

[…]
```

Grundlagen: Pointer

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

int main (int argc, char **argv) {
    int a = 5;
    char b = 'x';
    int c;
    int *ptr;

    ptr = &a;

    int myArray[3];

    char hiVar[5] = "Hi!";

    exit(EXIT_SUCCESS);
}
```

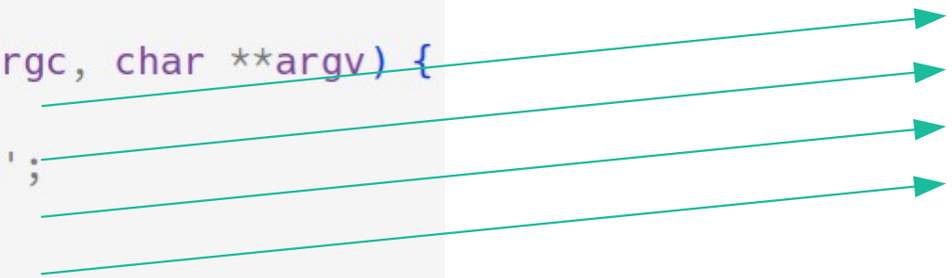
Speicherlayout

(Zur Vereinfachung keine Unterscheidung zw. Stack und Heap)

Address	Name	Typ	Value
0000			
0001			
0010			
0011			
0100			
0101			
0110			
0111			
1000			
1001			
1010			
1011			
1100			
1101			
1110			
1111			

Grundlagen: Pointer

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

int main (int argc, char **argv) {
    int a = 5;
    char b = 'x';
    int c;
    int *ptr;

    ptr = &a;

    int myArray[3];

    char hiVar[5] = "Hi!";

    exit(EXIT_SUCCESS);
}
```



Speicherlayout

(Zur Vereinfachung keine Unterscheidung zw. Stack und Heap)

Address	Name	Typ	Value
0000			
0001	a	int	5
0010	b	char	x
0011	c	int	(undefined)
0100	ptr	int-Pointer	(undefined)
0101			
0110			
0111			
1000			
1001			
1010			
1011			
1100			
1101			
1110			
1111			

Grundlagen: Pointer

```
1   #include <stdlib.h>
2   #include <stdio.h>
3
4   int main (int argc, char **argv) {
5       int a = 5;
6       char b = 'x';
7       int c;
8       int *ptr;
9
10      ptr = &a;
11
12      int myArray[3];
13
14      char hiVar[5] = "Hi!";
15
16      exit(EXIT_SUCCESS);
17  }
```

Speicher nach Ausführung von Zeile 10

Speicherlayout

(Zur Vereinfachung keine Unterscheidung zw. Stack und Heap)

Address	Name	Typ	Value
0000			
0001	a	int	5
0010	b	char	x
0011	c	int	(undefined)
0100	ptr	int-Pointer	**<0001>**
0101			
0110			
0111			
1000			
1001			
1010			
1011			
1100			
1101			
1110			
1111			

Grundlagen: Pointer

```
1   #include <stdlib.h>
2   #include <stdio.h>
3
4   int main (int argc, char **argv) {
5       int a = 5;
6       char b = 'x';
7       int c;
8       int *ptr;
9
10      ptr = &a;
11
12      int myArray[3];
13
14      char hiVar[5] = "Hi!";
15
16      exit(EXIT_SUCCESS);
17  }
```

Speicher nach Ausführung von Zeile 12

Speicherlayout

(Zur Vereinfachung keine Unterscheidung zw. Stack und Heap)

Address	Name	Typ	Value
0000			
0001	a	int	5
0010	b	char	x
0011	c	int	(undefined)
0100	ptr	int-Pointer	**<0001>**
0101			
0110	myArray	int-Pointer	**<0111>**
0111	*myArray[0]*	int	(undefined)
1000	*myArray[1]*	int	(undefined)
1001	*myArray[2]*	int	(undefined)
1010			
1011			
1100			
1101			
1110			
1111			

Grundlagen: Pointer

```
1   #include <stdlib.h>
2   #include <stdio.h>
3
4   int main (int argc, char **argv) {
5       int a = 5;
6       char b = 'x';
7       int c;
8       int *ptr;
9
10      ptr = &a;
11
12      int myArray[3];
13
14      char hiVar[5] = "Hi!";
15
16      exit(EXIT_SUCCESS);
17  }
```

Speicher nach Ausführung von Zeilen 14

Speicherlayout

(Zur Vereinfachung keine Unterscheidung zw. Stack und Heap)

Address	Name	Typ	Value
0000			
0001	a	int	5
0010	b	char	x
0011	c	int	(undefined)
0100	ptr	int-Pointer	**<0001>**
0101			
0110	myArray	int-Pointer	**<0111>**
0111	*myArray[0]*	int	(undefined)
1000	*myArray[1]*	int	(undefined)
1001	*myArray[2]*	int	(undefined)
1010	hiVar	char-Pointer	**<1011>**
1011	*hiVar[0]*	char	H
1100	*hiVar[1]*	char	i
1101	*hiVar[2]*	char	!
1110	*hiVar[3]*	char	\0
1111	*hiVar[4]*	char	(undefined)

Grundlagen: Pointer

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

int main (int argc, char **argv) {
    int a = 5;
    int *ptr;

    ptr = &a;

    printf ("%d\n", a);
    printf ("%p\n", ptr);

    printf ("%p\n", &a);
    printf ("%d\n", *ptr);

    exit(EXIT_SUCCESS);
}
```

Speicherlayout

(Zur Vereinfachung keine Unterscheidung zw. Stack und Heap)

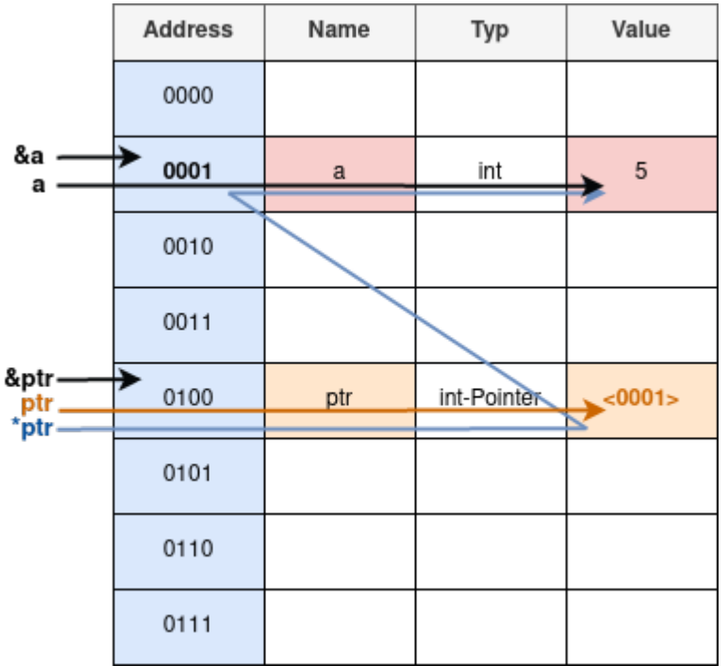
Address	Name	Typ	Value
0000			
0001	a	int	5
0010			
0011			
0100	ptr	int-Pointer	**<0001>**
0101			
0110			
0111			

Grundlagen: Pointer

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

int main (int argc, char **argv) {
    int a = 5;
    int *ptr;

    ptr = &a;

    printf ("%d\n", a);
    printf ("%p\n", ptr);

    printf ("%p\n", &a);
    printf ("%d\n", *ptr);

    exit(EXIT_SUCCESS);
}
```

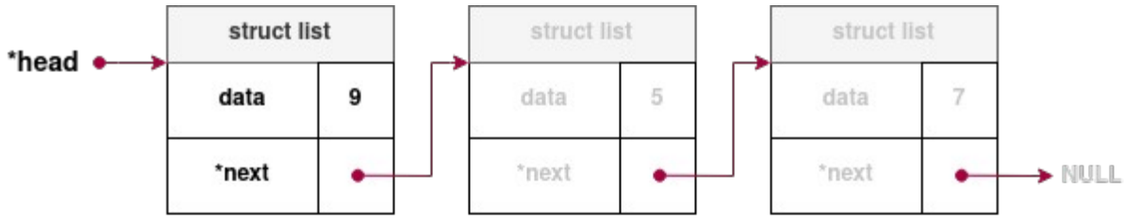
Speicherlayout

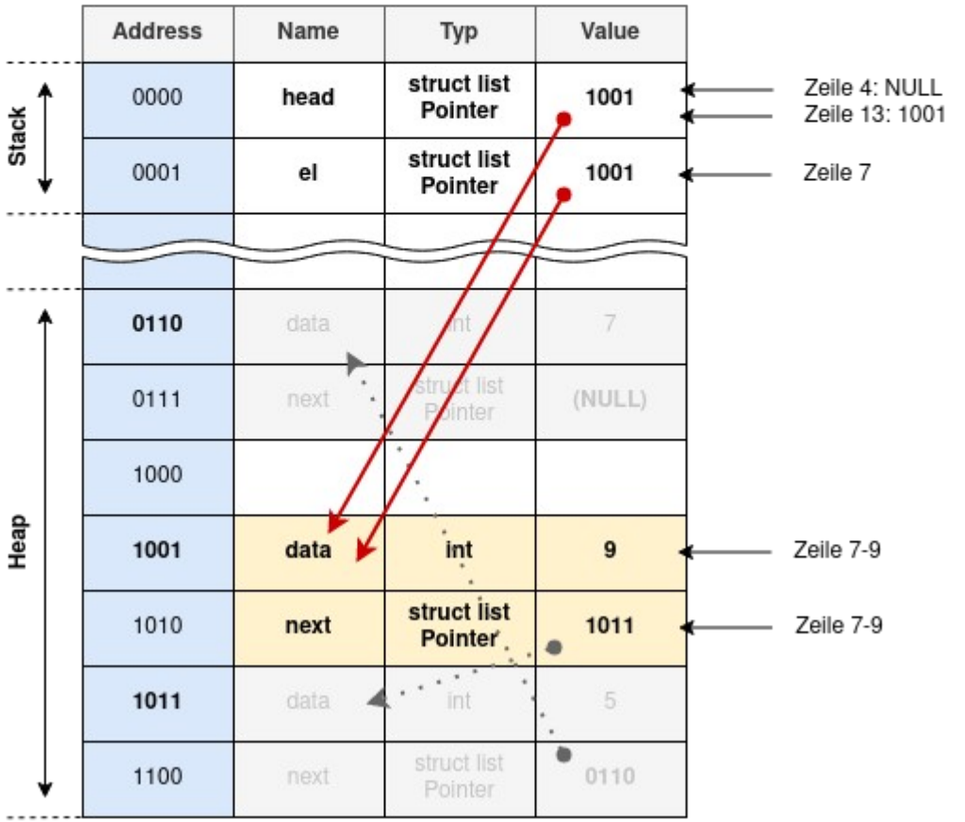
(Zur Vereinfachung keine Unterscheidung zw. Stack und Heap)



Beispiel: Verkettete Liste

```c
struct list {
    int date;
    struct list* next;
}

int main (void) {
    struct list* head = NULL;

    // Create new element
    struct list* new_el = malloc (...);
    new_el->data = 5;
    new_el->next = NULL;

    // append element to list
    if (head == NULL) {
        head = new_el;
    } else {
        struct list* tmp = head;
        while (tmp->next != NULL) {
            tmp = tmp->next;
        }
        tmp->next = new_el;
    }

    [ ... ]
}
```

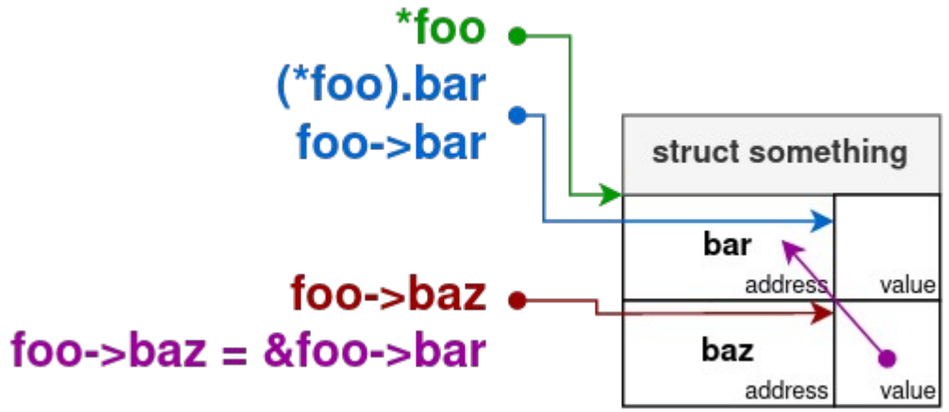




Elementzugriff bei Zeigern

foo→bar == (*foo).bar != *foo.bar

```c
struct data {
    int bar;
    int *baz;
}

int main (void) {
    struct data* foo = malloc (...);

    foo->bar = 5;
    (*foo).bar = 5;

    foo-baz = &foo->bar

    [ ... ]
}
```

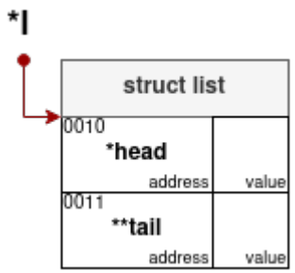


Beispiel: FIFO-Liste

```c
struct node {
    int data;
    struct node *next;
};

struct list {
    struct node *head;
    struct node **tail;
};

// Create list
struct list *l = (struct list *) malloc(sizeof(struct list));
l->head = NULL;
l->tail = &l->head;

// Create element
struct node *el = (struct node *) malloc(sizeof(struct node));
el->data = 5;
el->next = NULL;

// Append
*l->tail = new_el;
l->tail = &new_el->next;

// Delete
struct list **p = &l->head;
while ((*p) != target) {
    p = &(*p)->next;
}
*p = target->next;
free(target);
```

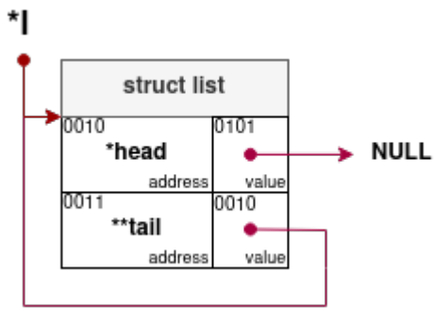
Beispiel: FIFO-Liste

```c
struct node {
    int data;
    struct node *next;
};

struct list {
    struct node *head;
    struct node **tail;
};

// Create list
struct list *l = (struct list *) malloc(sizeof(struct list));
l->head = NULL;
l->tail = &l->head;

// Create element
struct node *el = (struct node *) malloc(sizeof(struct node));
el->data = 5;
el->next = NULL;

// Append
*l->tail = new_el;
l->tail = &new_el->next;

// Delete
struct list **p = &l->head;
while ((*p) != target) {
    p = &(*p)->next;
}
*p = target->next;
free(target);
```



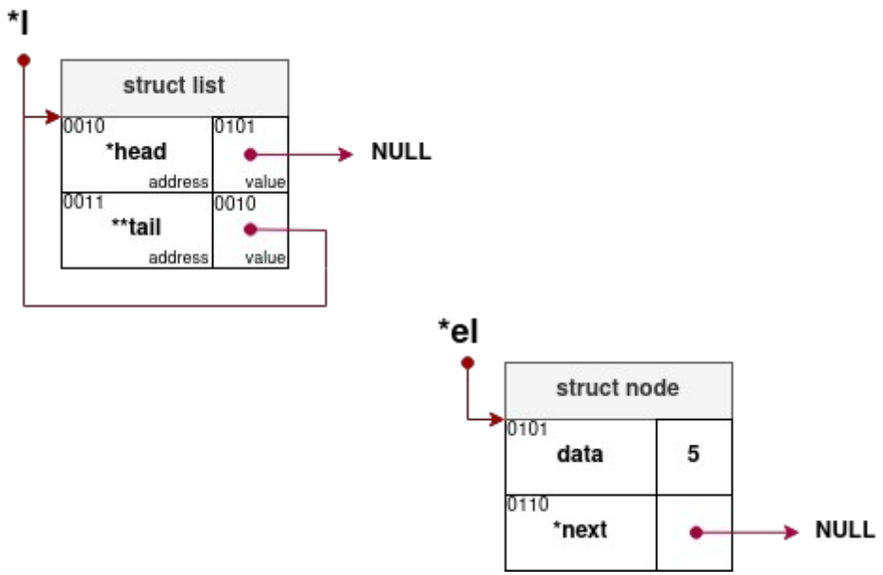
Beispiel: FIFO-Liste

```c
struct node {
    int data;
    struct node *next;
};

struct list {
    struct node *head;
    struct node **tail;
};

// Create list
struct list *l = (struct list *) malloc(sizeof(struct list));
l->head = NULL;
l->tail = &l->head;

// Create element
struct node *el = (struct node *) malloc(sizeof(struct node));
el->data = 5;
el->next = NULL;

// Append
*l->tail = new_el;
l->tail = &new_el->next;

// Delete
struct list **p = &l->head;
while ((*p) != target) {
    p = &(*p)->next;
}
*p = target->next;
free(target);
```



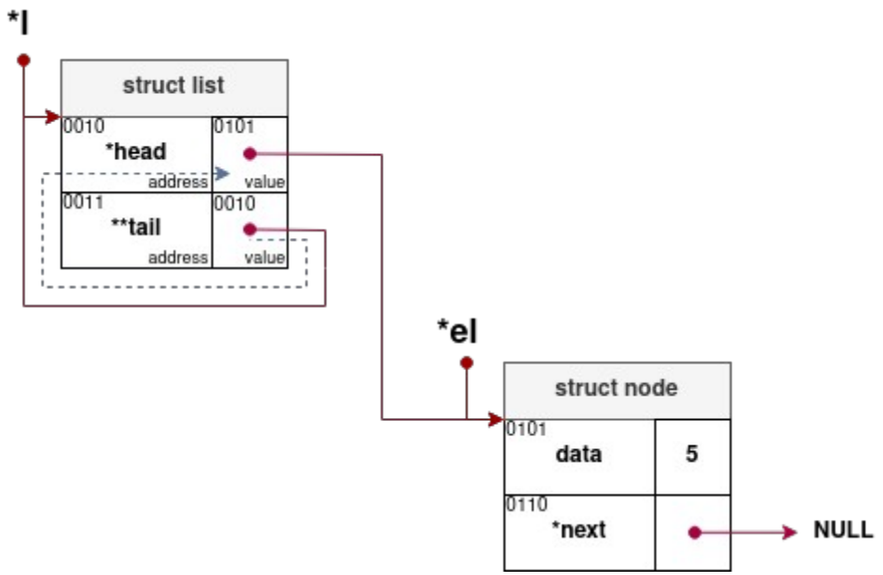
Beispiel: FIFO-Liste

```c
struct node {
    int data;
    struct node *next;
};

struct list {
    struct node *head;
    struct node **tail;
};

// Create list
struct list *l = (struct list *) malloc(sizeof(struct list));
l->head = NULL;
l->tail = &l->head;

// Create element
struct node *el = (struct node *) malloc(sizeof(struct node));
el->data = 5;
el->next = NULL;

// Append
*l->tail = new_el;
l->tail = &new_el->next;

// Delete
struct list **p = &l->head;
while ((*p) != target) {
    p = &(*p)->next;
}
*p = target->next;
free(target);
```



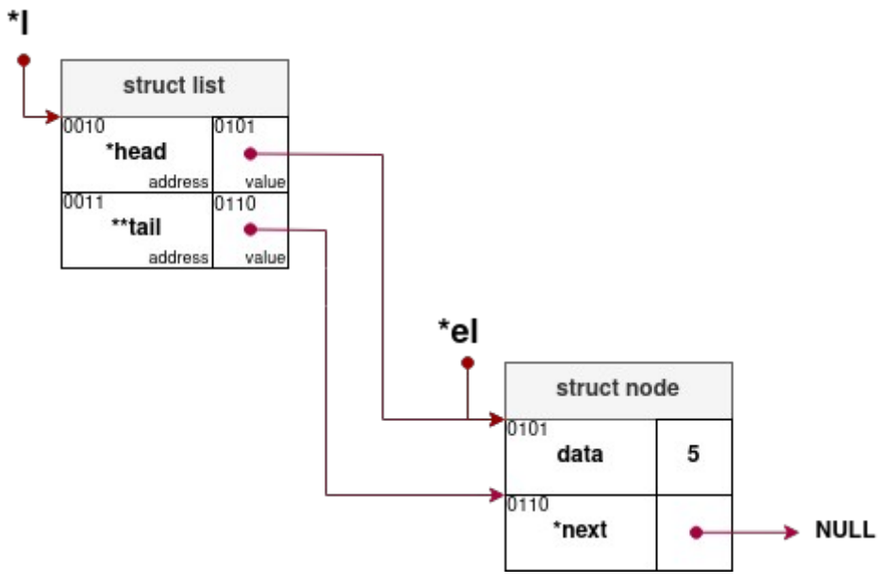
Beispiel: FIFO-Liste

```c
struct node {
    int data;
    struct node *next;
};

struct list {
    struct node *head;
    struct node **tail;
};

// Create list
struct list *l = (struct list *) malloc(sizeof(struct list));
l->head = NULL;
l->tail = &l->head;

// Create element
struct node *el = (struct node *) malloc(sizeof(struct node));
el->data = 5;
el->next = NULL;

// Append
*l->tail = new_el;
l->tail = &new_el->next;

// Delete
struct list **p = &l->head;
while ((*p) != target) {
    p = &(*p)->next;
}
*p = target->next;
free(target);
```



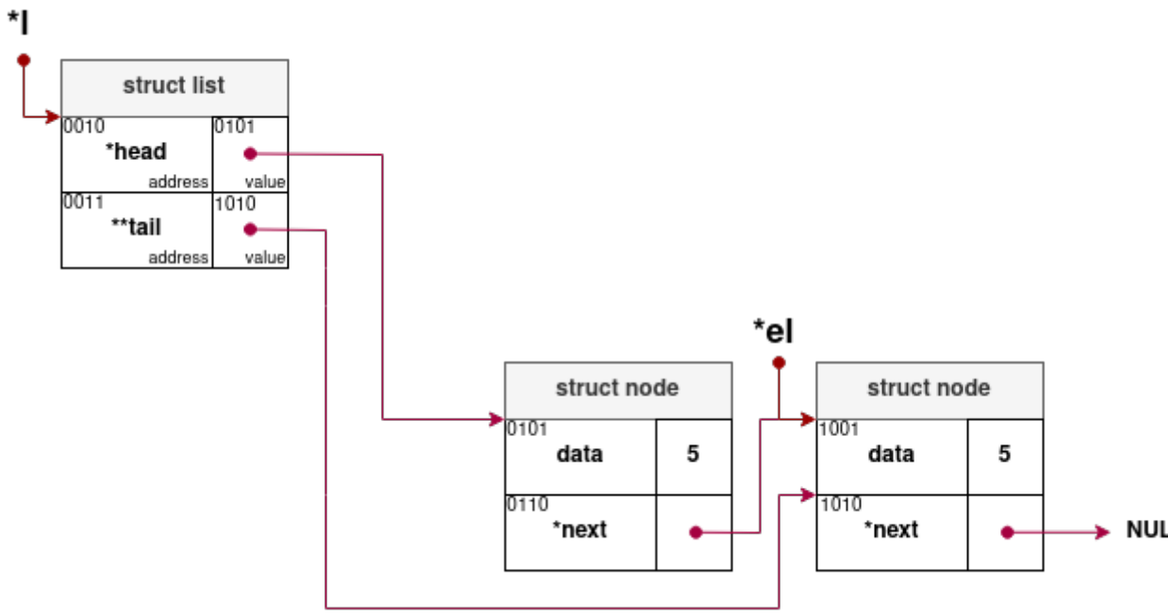
Beispiel: FIFO-Liste

```c
struct node {
    int data;
    struct node *next;
};

struct list {
    struct node *head;
    struct node **tail;
};

// Create list
struct list *l = (struct list *) malloc(sizeof(struct list));
l->head = NULL;
l->tail = &l->head;

// Create element
struct node *el = (struct node *) malloc(sizeof(struct node));
el->data = 5;
el->next = NULL;

// Append
*l->tail = new_el;
l->tail = &new_el->next;

// Delete
struct list **p = &l->head;
while ((*p) != target) {
    p = &(*p)->next;
}
*p = target->next;
free(target);
```

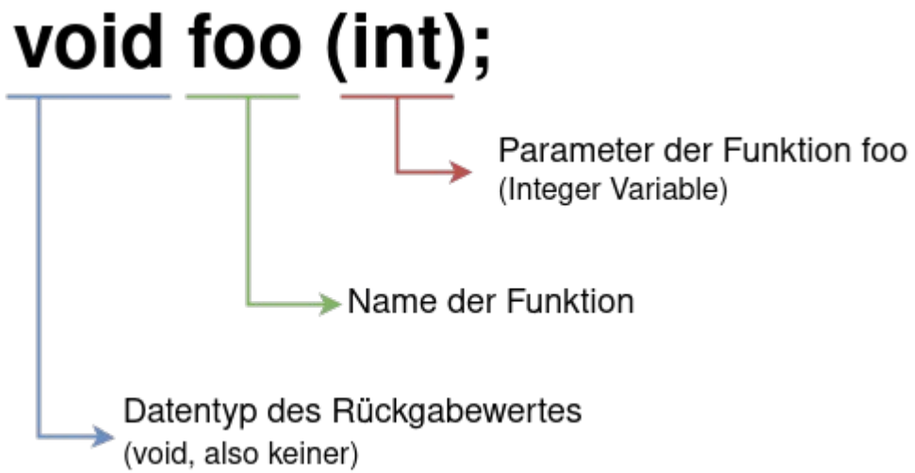


Beispiel: FIFO-Liste

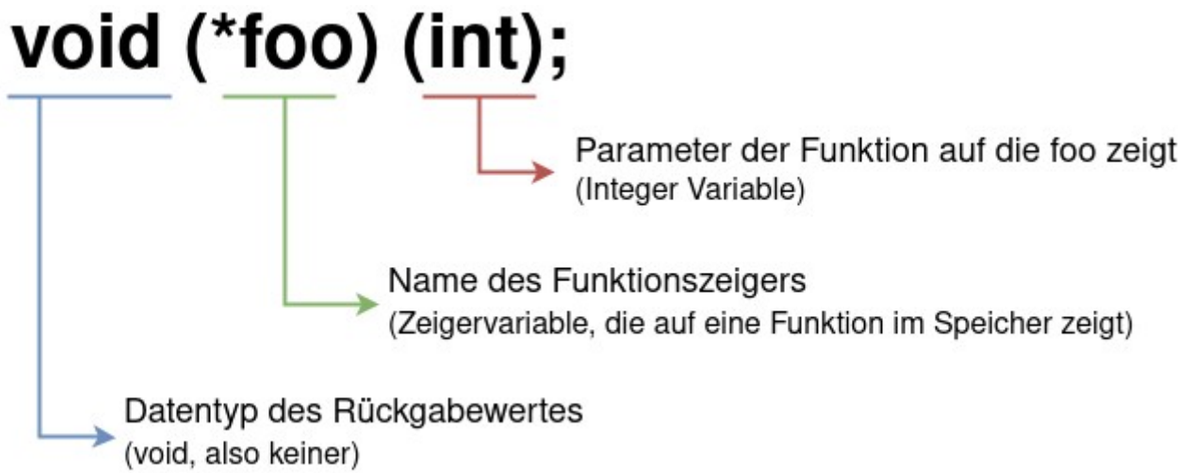
```c
struct node {
    int data;
    struct node *next;
};

struct list {
    struct node *head;
    struct node **tail;
};

// Create list
struct list *l = (struct list *) malloc(sizeof(struct list));
l->head = NULL;
l->tail = &l->head;

// Create element
struct node *el = (struct node *) malloc(sizeof(struct node));
el->data = 5;
el->next = NULL;

// Append
*l->tail = new_el;
l->tail = &new_el->next;

// Delete
struct list **p = &l->head;
while ((*p) != target) {
    p = &(*p)->next;
}
*p = target->next;
free(target);
```



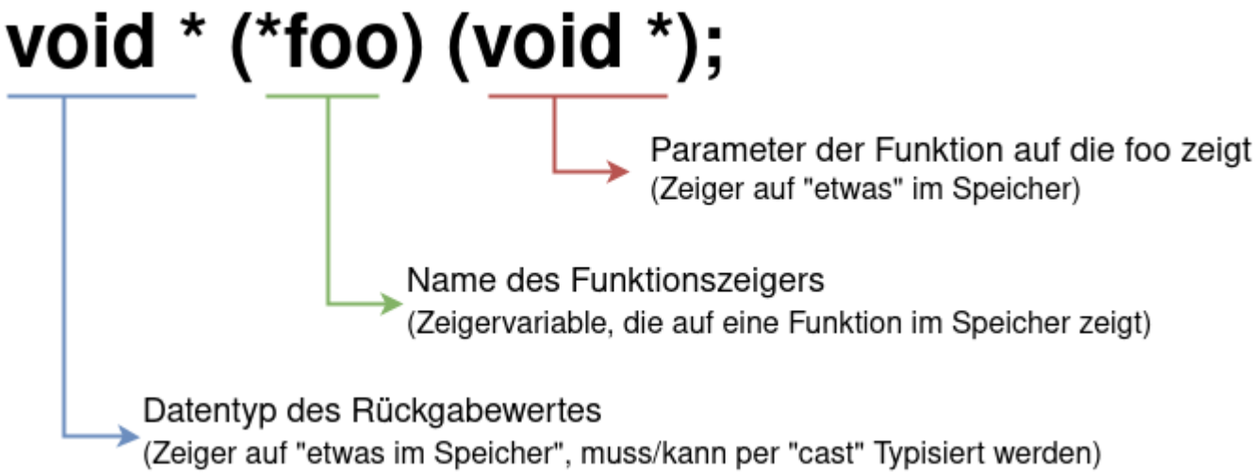
Funktionen und deren Signaturen



Funktionen und deren Signaturen



Funktionen und deren Signaturen



Beispielcode: Funktionspointer

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

typedef struct person {
    char name[10];
    int age;
} person_t;

void print_person (person_t *p) {
    printf ("%s: %d\n", p->name, p->age);
    return;
}

int main (void) {
    person_t *p = (person_t *) malloc (sizeof (person_t));
    strncpy (p->name, "Sven", sizeof("Sven"));
    p->age = 42;

    void (*myprint)(person_t *) = &print_person;
    (*myprint)(p);

    exit (EXIT_SUCCESS);
}
```