

Betriebssysteme

Grundlagen C & Pointer

Fachbereich
Angewandte Informatik

AI



Hochschule Fulda
University of Applied Sciences



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

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int min (int, int);
5
6 int minimum;
7
8 int main (int argc, char **argv) {
9     minimum = min (5, 10);
10    printf ("Minimum: %d\n", minimum);
11
12    exit (EXIT_SUCCESS);
13}
14
15 int min (int a, int b) {
16     if (a < b)
17         return a;
18     else
19         return b;
20}
```

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

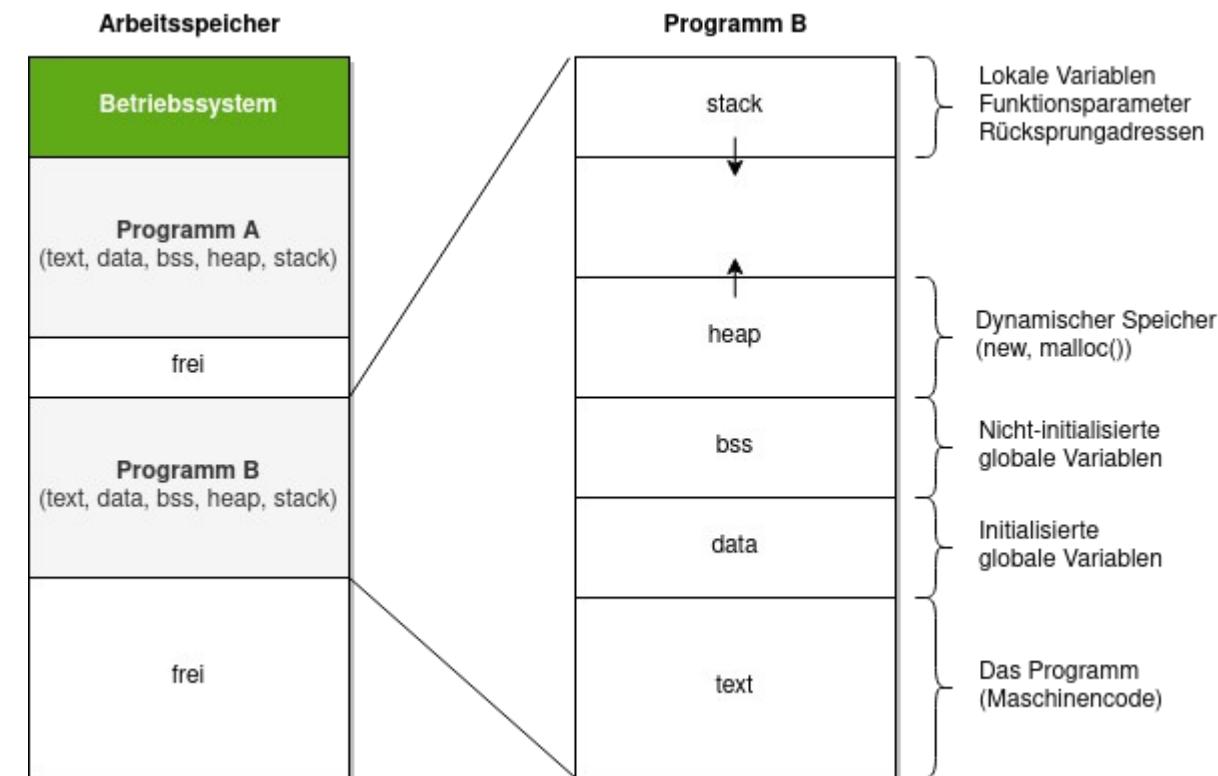
Definition der Funktion min

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

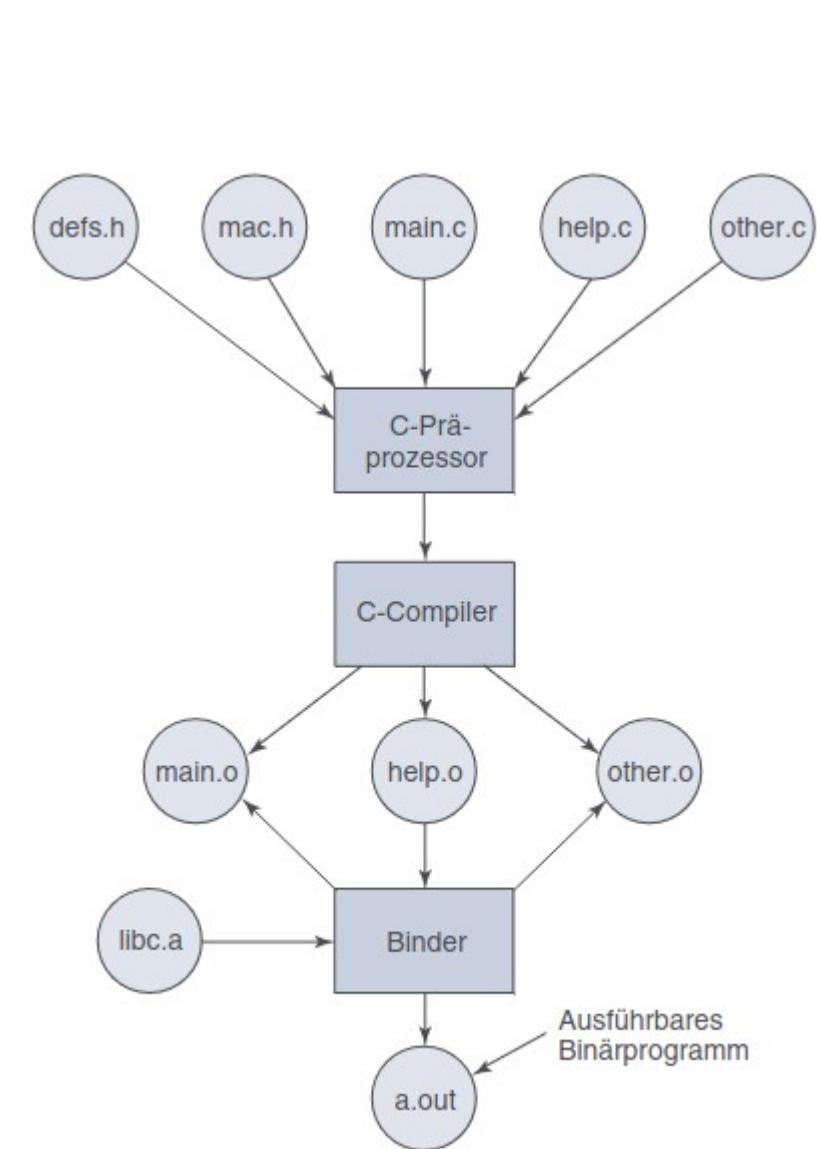
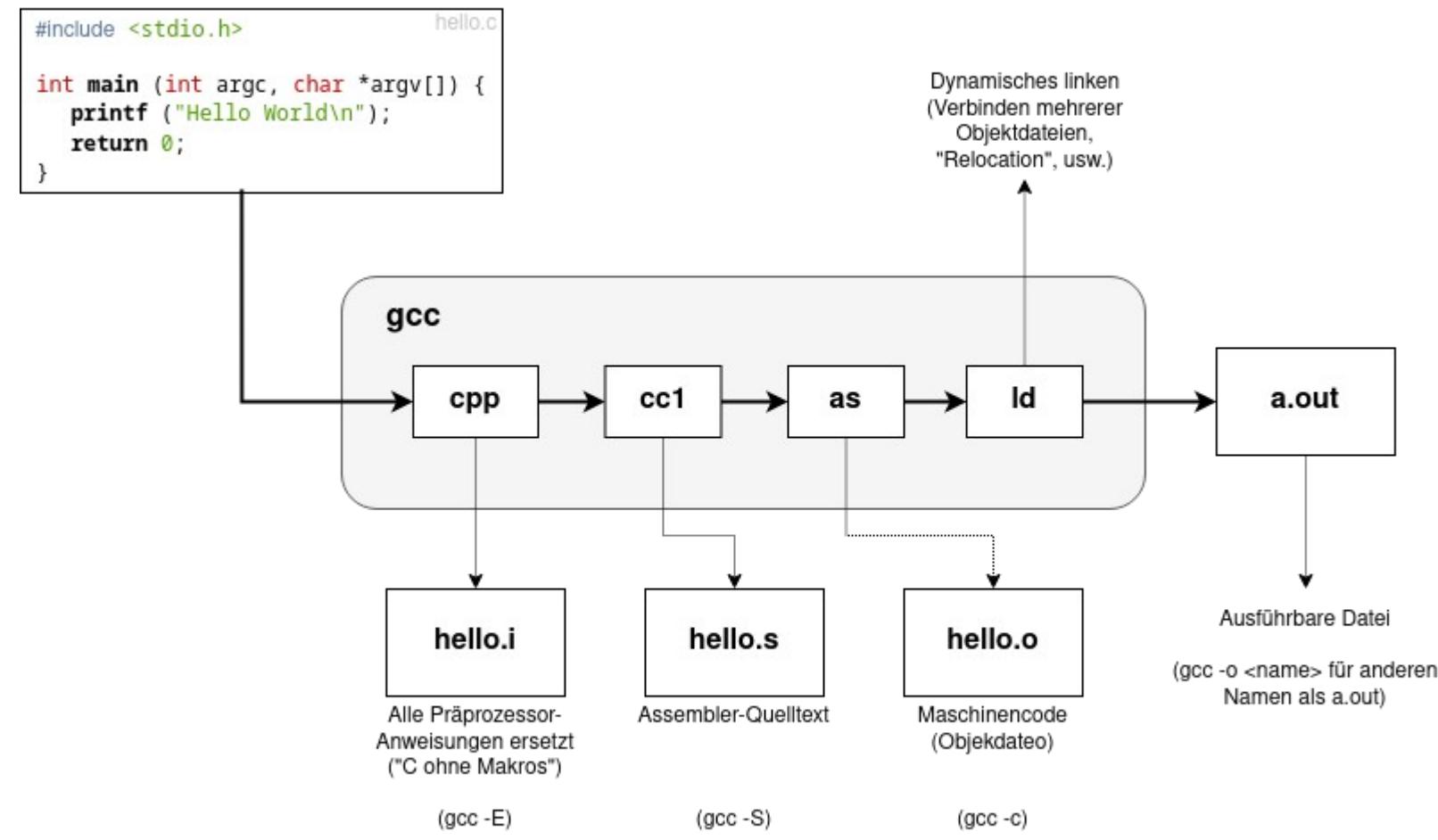


Programmlayout im Speicher

- **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



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]  

```

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
```

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

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

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 ($\rightarrow \text{INT_MAX} = +32767$), kann aber je nach Implementierung/Plattform auch länger sein.
Durch POSIX definierte Typen, wie z.B. **unit32_t** (= **unsigned integer 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

```
// 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);
```

```
// 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

```
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 }
```

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

```
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 }
```

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

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main (int argc, char **argv) {
5     int a = 5;
6     int *ptr;
7
8     ptr = &a;
9
10    printf ("%d\n", a);
11    printf ("%p\n", ptr);
12
13    printf ("%p\n", &a);
14    printf ("%d\n", *ptr);
15
16    exit(EXIT_SUCCESS);
17 }
```

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

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main (int argc, char **argv) {
5     int a = 5;
6     int *ptr;
7
8     ptr = &a;
9
10    printf ("%d\n", a);
11    printf ("%p\n", ptr);
12
13    printf ("%p\n", &a);
14    printf ("%d\n", *ptr);
15
16    exit(EXIT_SUCCESS);
17 }
```

Speicherlayout

(Zur Vereinfachung keine Unterscheidung zw. Stack und Heap)

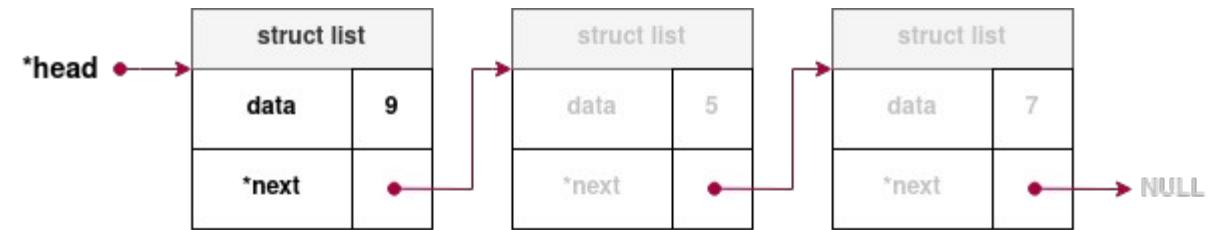
Address	Name	Typ	Value
0000			
0001	a	int	5
0010			
0011			
0100	ptr	int-Pointer	<0001>
0101			
0110			
0111			

Diagramm des Speicherlayouts:

- Zeile 0001: Variablenname "a", Typ "int", Wert "5".
- Zeile 0100: Variablenname "ptr", Typ "int-Pointer", Wert "<0001>". Der Wert zeigt auf die Adresse von "a".
- Zeile 0001: Zeigervariable "ptr" auf "a".
- Zeile 0001: Direkter Zugriff auf den Speicherort von "a".

Beispiel: Verkettete Liste

```
1 struct list {  
2     int date;  
3     struct list* next;  
4 }  
5  
6 int main (void) {  
7     struct list* head = NULL;  
8  
9     // Create new element  
10    struct list* new_el = malloc (...);  
11    new_el->data = 5;  
12    new_el->next = NULL;  
13  
14    // append element to list  
15    if (head == NULL) {  
16        head = new_el;  
17    } else {  
18        struct list* tmp = head;  
19        while (tmp->next != NULL) {  
20            tmp = tmp->next;  
21        }  
22        tmp->next = new_el;  
23    }  
24  
25    [ ... ]  
26 }
```



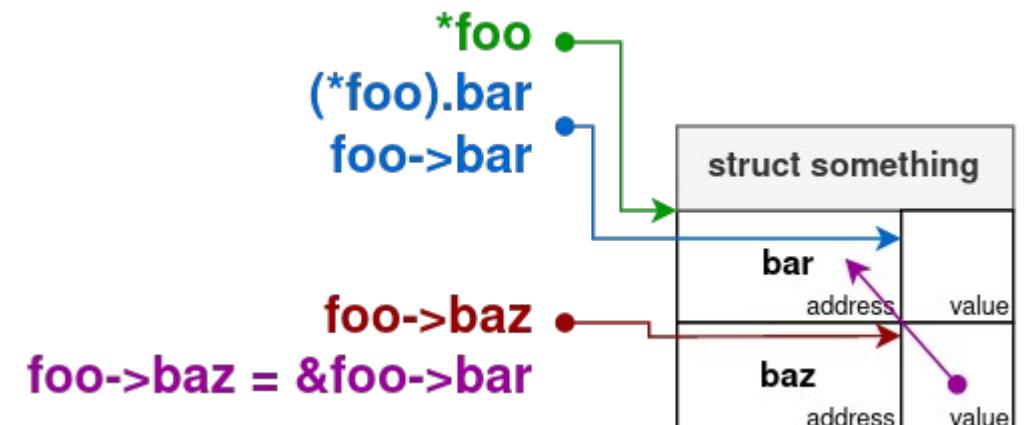
Address	Name	Typ	Value
0000	head	struct list Pointer	1001
0001	el	struct list Pointer	1001
0110	<td>int</td> <td>7</td>	int	7
0111	next	struct list Pointer	(NULL)
1000			
1001	<td>int</td> <td>9</td>	int	9
1010	next	struct list Pointer	1011
1011	<td>int</td> <td>5</td>	int	5
1100	next	struct list Pointer	0110

The table illustrates the memory layout of the heap and stack. The stack grows downwards (indicated by a downward arrow) and the heap grows upwards (indicated by an upward arrow). The 'head' pointer at address 0000 points to the first list node at address 0001. The 'el' pointer at address 0001 points to the new element at address 1001. The 'data' field of the new element contains the value 9. The 'next' field of the new element contains the address of the previous list node, which is 1011. The 'data' field of the previous list node contains the value 5. The 'next' field of the previous list node contains the address of the node after it, which is 0110.

Elementzugriff bei Zeigern

`foo->bar` == `(*foo).bar` != `*foo.bar`

```
1 struct data {
2     int bar;
3     int *baz;
4 }
5
6 int main (void) {
7     struct data* foo = malloc ( ... );
8
9     foo->bar = 5;
10    (*foo).bar = 5;
11
12    foo->baz = &foo->bar
13
14    [ ... ]
15 }
```

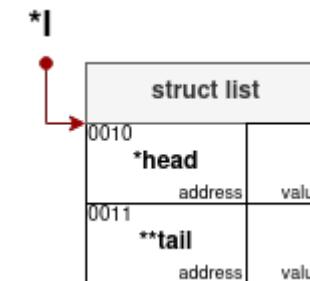


Beispiel: FIFO-Liste

```
1 struct node {
2     int data;
3     struct node *next;
4 };
5
6 struct list {
7     struct node *head;
8     struct node **tail;
9 };
10
11 // Create list
12 struct list *l = (struct list *) malloc(sizeof(struct list));
13 l->head = NULL;
14 l->tail = &l->head;
15
16 // Create element
17 struct node *el = (struct node *) malloc(sizeof(struct node));
18 el->data = 5;
19 el->next = NULL;
20
21 // Append
22 *l->tail = new_el;
23 l->tail = &new_el->next;
24
25 // Delete
26 struct list **p = &l->head;
27 while ((*p) != target) {
28     p = &(*p)->next;
29 }
30 *p = target->next;
31 free(target);
```

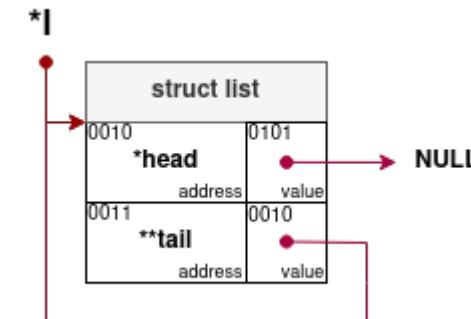
Beispiel: FIFO-Liste

```
1 struct node {  
2     int data;  
3     struct node *next;  
4 };  
5  
6 struct list {  
7     struct node *head;  
8     struct node **tail;  
9 };  
10  
11 // Create list  
12 struct list *l = (struct list *) malloc(sizeof(struct list));  
13 l->head = NULL;  
14 l->tail = &l->head;  
15  
16 // Create element  
17 struct node *el = (struct node *) malloc(sizeof(struct node));  
18 el->data = 5;  
19 el->next = NULL;  
20  
21 // Append  
22 *l->tail = new_el;  
23 l->tail = &new_el->next;  
24  
25 // Delete  
26 struct list **p = &l->head;  
27 while ((*p) != target) {  
28     p = &(*p)->next;  
29 }  
30 *p = target->next;  
31 free(target);
```



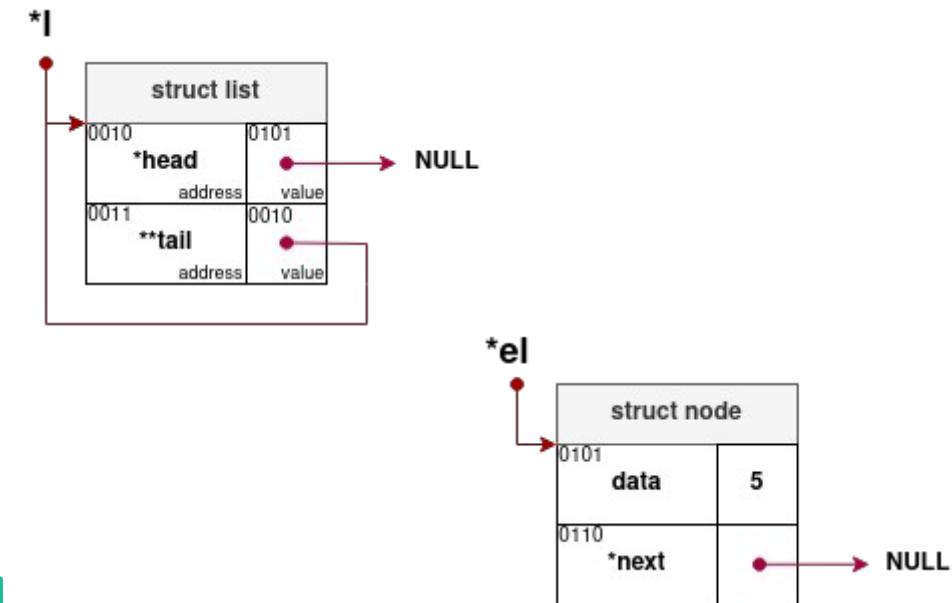
Beispiel: FIFO-Liste

```
1 struct node {
2     int data;
3     struct node *next;
4 };
5
6 struct list {
7     struct node *head;
8     struct node **tail;
9 };
10
11 // Create list
12 struct list *l = (struct list *) malloc(sizeof(struct list));
13 l->head = NULL;
14 l->tail = &l->head;
15
16 // Create element
17 struct node *el = (struct node *) malloc(sizeof(struct node));
18 el->data = 5;
19 el->next = NULL;
20
21 // Append
22 *l->tail = new_el;
23 l->tail = &new_el->next;
24
25 // Delete
26 struct list **p = &l->head;
27 while ((*p) != target) {
28     p = &(*p)->next;
29 }
30 *p = target->next;
31 free(target);
```



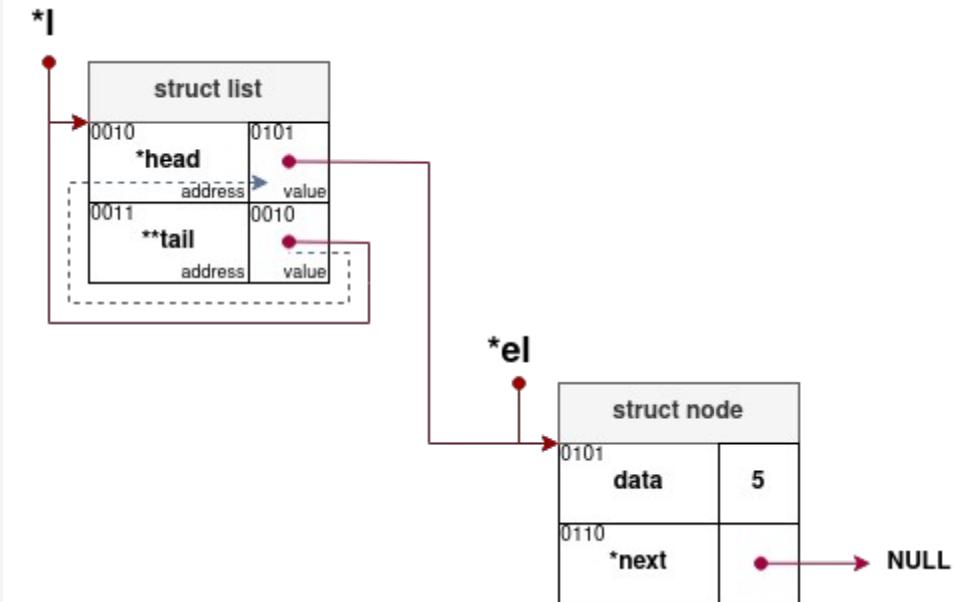
Beispiel: FIFO-Liste

```
1 struct node {
2     int data;
3     struct node *next;
4 };
5
6 struct list {
7     struct node *head;
8     struct node **tail;
9 };
10
11 // Create list
12 struct list *l = (struct list *) malloc(sizeof(struct list));
13 l->head = NULL;
14 l->tail = &l->head;
15
16 // Create element
17 struct node *el = (struct node *) malloc(sizeof(struct node));
18 el->data = 5;
19 el->next = NULL;
20
21 // Append
22 *l->tail = new_el;
23 l->tail = &new_el->next;
24
25 // Delete
26 struct list **p = &l->head;
27 while ((*p) != target) {
28     p = &(*p)->next;
29 }
30 *p = target->next;
31 free(target);
```



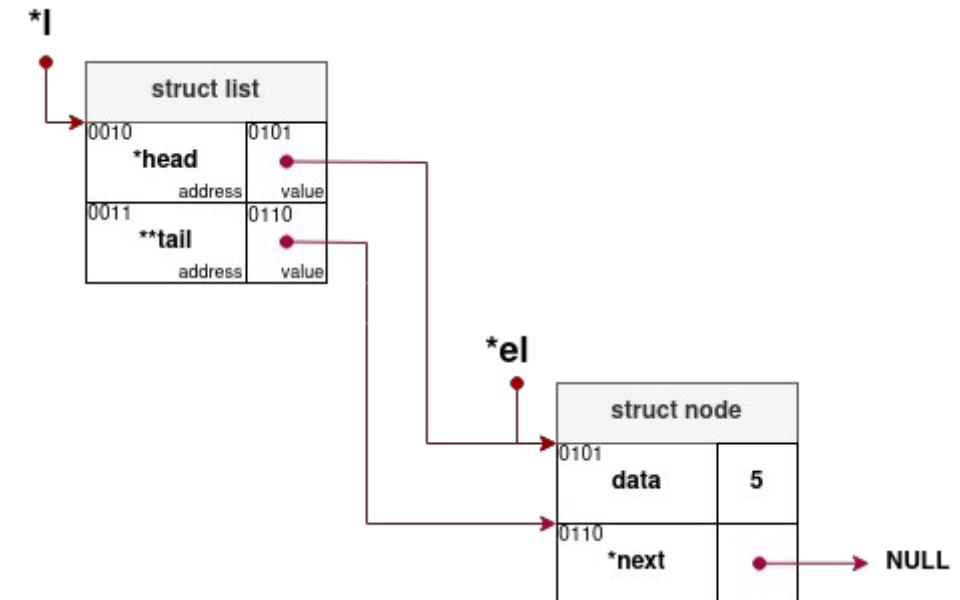
Beispiel: FIFO-Liste

```
1 struct node {
2     int data;
3     struct node *next;
4 };
5
6 struct list {
7     struct node *head;
8     struct node **tail;
9 };
10
11 // Create list
12 struct list *l = (struct list *) malloc(sizeof(struct list));
13 l->head = NULL;
14 l->tail = &l->head;
15
16 // Create element
17 struct node *el = (struct node *) malloc(sizeof(struct node));
18 el->data = 5;
19 el->next = NULL;
20
21 // Append
22 *l->tail = new_el;
23 l->tail = &new_el->next;
24
25 // Delete
26 struct list **p = &l->head;
27 while ((*p) != target) {
28     p = &(*p)->next;
29 }
30 *p = target->next;
31 free(target);
```



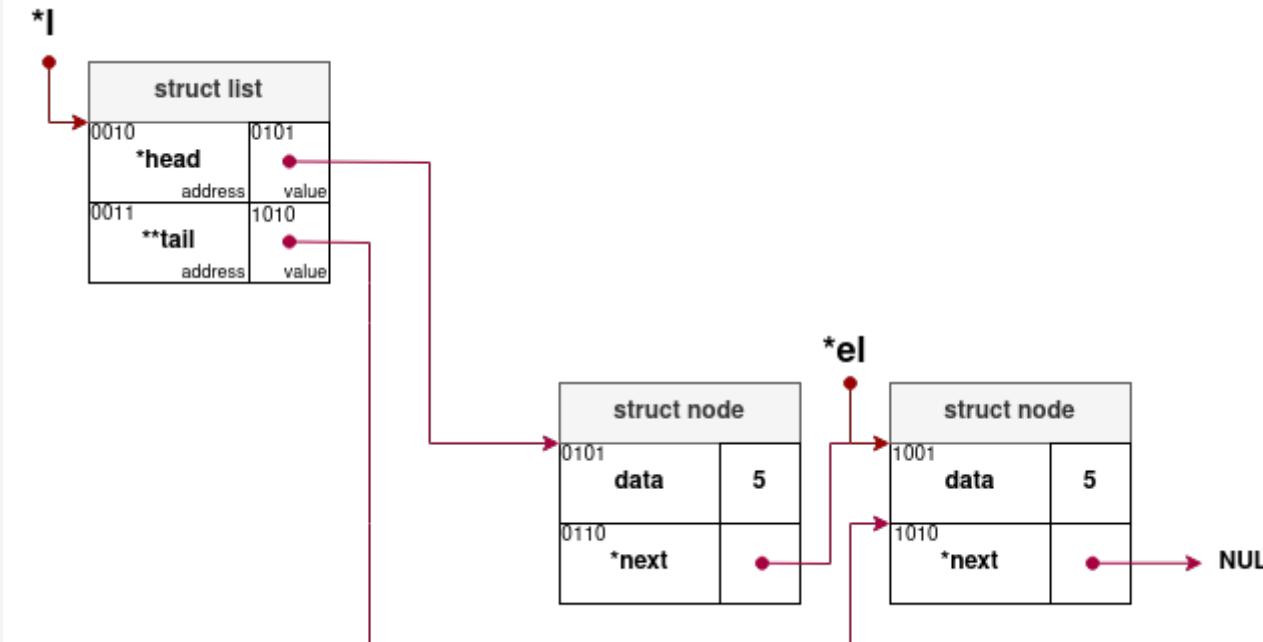
Beispiel: FIFO-Liste

```
1 struct node {  
2     int data;  
3     struct node *next;  
4 };  
5  
6 struct list {  
7     struct node *head;  
8     struct node **tail;  
9 };  
10  
11 // Create list  
12 struct list *l = (struct list *) malloc(sizeof(struct list));  
13 l->head = NULL;  
14 l->tail = &l->head;  
15  
16 // Create element  
17 struct node *el = (struct node *) malloc(sizeof(struct node));  
18 el->data = 5;  
19 el->next = NULL;  
20  
21 // Append  
22 *l->tail = new_el;  
23 l->tail = &new_el->next;  
24  
25 // Delete  
26 struct list **p = &l->head;  
27 while ((*p) != target) {  
28     p = &(*p)->next;  
29 }  
30 *p = target->next;  
31 free(target);
```



Beispiel: FIFO-Liste

```
1 struct node {
2     int data;
3     struct node *next;
4 };
5
6 struct list {
7     struct node *head;
8     struct node **tail;
9 };
10
11 // Create list
12 struct list *l = (struct list *) malloc(sizeof(struct list));
13 l->head = NULL;
14 l->tail = &l->head;
15
16 // Create element
17 struct node *el = (struct node *) malloc(sizeof(struct node));
18 el->data = 5;
19 el->next = NULL;
20
21 // Append
22 *l->tail = new_el;
23 l->tail = &new_el->next;
24
25 // Delete
26 struct list **p = &l->head;
27 while ((*p) != target) {
28     p = &(*p)->next;
29 }
30 *p = target->next;
31 free(target);
```



Funktionen und deren Signaturen

void foo (int);

The diagram shows the function signature **void foo (int);**. Three arrows point from the text labels to the corresponding parts of the signature:

- A blue arrow points to the word **void**, which is labeled "Datentyp des Rückgabewertes (void, also keiner)".
- A green arrow points to the word **foo**, which is labeled "Name der Funktion".
- A red arrow points to the word **int**, which is labeled "Parameter der Funktion foo (Integer Variable)".

Funktionen und deren Signaturen

```
void (*foo) (int);
```

The diagram shows the C-style function signature `void (*foo) (int);`. Three arrows point from the text labels to specific parts of the code:

- A blue arrow points to the return type `void`.
- A green arrow points to the parameter `(*foo)`, which is a pointer to a function.
- A red arrow points to the argument `(int)`, which is the parameter type.

Annotations explain the meaning of each part:

- Datentyp des Rückgabewertes
(void, also keiner)
- Name des Funktionszeigers
(Zeigervariable, die auf eine Funktion im Speicher zeigt)
- Parameter der Funktion auf die foo zeigt
(Integer Variable)

Funktionen und deren Signaturen

```
void * (*foo) (void *);
```

The diagram illustrates the structure of the function pointer type `void * (*foo) (void *);`. It consists of three main parts: a blue bracket on the left indicating the return type, a green bracket in the middle indicating the name of the function pointer variable, and a red bracket on the right indicating the parameter type. Arrows point from each bracket to their respective explanatory text.

- Blue arrow: Datentyp des Rückgabewertes
(Zeiger auf "etwas im Speicher", muss/kann per "cast" Typisiert werden)
- Green arrow: Name des Funktionszeigers
(Zeigervariable, die auf eine Funktion im Speicher zeigt)
- Red arrow: Parameter der Funktion auf die foo zeigt
(Zeiger auf "etwas" im Speicher)

Beispielcode: Funktionspointer

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 typedef struct person {
6     char name[10];
7     int age;
8 } person_t;
9
10 void print_person (person_t *p) {
11     printf ("%s: %d\n", p->name, p->age);
12     return;
13 }
14
15 int main (void) {
16     person_t *p = (person_t *) malloc (sizeof (person_t));
17     strncpy (p->name, "Sven", sizeof("Sven"));
18     p->age = 42;
19
20     void (*myprint)(person_t *) = &print_person;
21     (*myprint)(p);
22
23     exit (EXIT_SUCCESS);
24 }
```