

Kurseinheit 8: Zeiger 3

1. Zeiger auf Zeiger
2. Datenstrukturen
3. Datenstruktur verkettete Liste
4. Funktionszeiger

Übersicht KE 8

Lehrveranstaltung Ingenieur-Informatik – 2 SWS/2 Credits: EI1, EI+1, MKA1, MK+1, EI3nat3
Lehrveranstaltung Programmierung 2 (Teil C) – 2 SWS/2 Credits: AI2

Unterrichtsdauer für diese Kurseinheit: 90 Minuten

Korrespondierende Kapitel aus *C-Programmierung – Eine Einführung*: Kapitel 8

Zusatzthemen: Sortieren mit QuickSort

1. Zeiger auf Zeiger

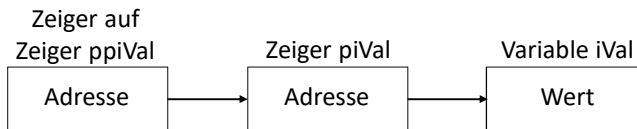
3

Prinzip

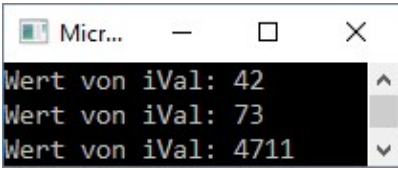
Zeiger auf Zeiger ist ein recht komplexes Thema. Hauptsächlich Anwendungsgebiete sind:

- Verschieben eines Zeigers in einer Funktion
- Dynamische Erzeugung von mehrdimensionalen Arrays

```
int iVal = 42;  
int* piVal = NULL;  
int** ppiVal = NULL;
```



```
piVal = &iVal;  
ppiVal = &piVal;  
  
printf("Wert von iVal: %i\n", iVal);  
*piVal = 73;  
printf("Wert von iVal: %i\n", iVal);  
**ppiVal = 4711;  
printf("Wert von iVal: %i\n", iVal);
```



```
Micr...  
Wert von iVal: 42  
Wert von iVal: 73  
Wert von iVal: 4711
```

1. Zeiger auf Zeiger

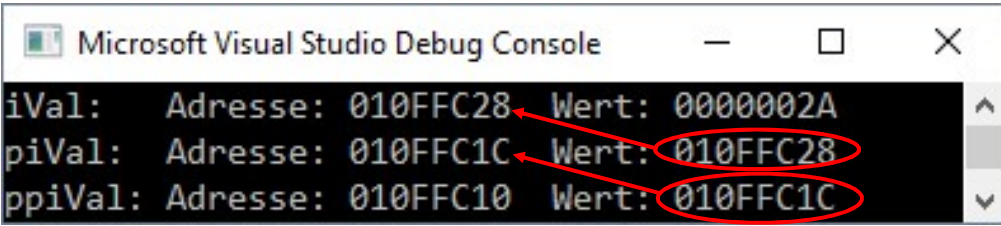
4

Speicherabbild

```
int iVal = 42;  
int* piVal = NULL;  
int** ppiVal = NULL;
```

C-Coding Styleguide:
pp für pointer to pointer

```
piVal = &iVal;  
ppiVal = &piVal;  
printf("iVal: Adresse: %08p Wert: %08X\n", &iVal, iVal);  
printf("piVal: Adresse: %08p Wert: %08X\n", &piVal, piVal);  
printf("ppiVal: Adresse: %08p Wert: %08X\n", &ppiVal, ppiVal);
```



```
Microsoft Visual Studio Debug Console  
iVal: Adresse: 010FFC28 Wert: 0000002A  
piVal: Adresse: 010FFC1C Wert: 010FFC28  
ppiVal: Adresse: 010FFC10 Wert: 010FFC1C
```


1. Zeiger auf Zeiger

7

Anwendungsgebiet: Dyn. Erzeugung von mehrdimensionalen Arrays

```
ppcA = (char**)malloc(4 * sizeof(char*)); 1

if (ppcA != NULL)
{
    ppcA[0] = (char*)malloc(3 * sizeof(char)); //equivalent code 2
    ppcA[1] = (char*)malloc(12 * sizeof(char)); //equivalent code 3
    *(ppcA + 2) = (char*)malloc(3 * sizeof(char)); 4
    *(ppcA + 3) = (char*)malloc(48 * sizeof(char)); 5

    if ((ppcA[0] != NULL) && (ppcA[1] != NULL) &&
        (ppcA[2] != NULL) && (ppcA[3] != NULL))
    {
        strncpy_s(ppcA[0], 3, "EI", 3);
        strncpy_s(ppcA[1], 12, "Hello World", 12);
        strncpy_s(ppcA[2], 3, "42", 3);
        strncpy_s(ppcA[3], 48, "Dies ist eine wirklich sehr lange Zeichenkette.", 48);

        // free ppcA[0], ... if != NULL
        free(ppcA);
        ppcA = NULL;
    }
}
```

Sicheres strncpy_s – seit C11

free: Umgekehrte Reihenfolge
wie beim malloc (erst 2,3,4,5 oder
5,4,3,2) dann erst 1

2. Datenstrukturen

8

Was ist eine Datenstruktur?

In der Informatik und Softwaretechnik ist eine **Datenstruktur** ein Objekt, welches zur Speicherung und Organisation von Daten dient. Es handelt sich um eine Struktur, weil die Daten in einer bestimmten Art und Weise angeordnet und verknüpft werden, um den Zugriff auf sie und ihre Verwaltung effizient zu ermöglichen.

Quelle: wikipedia.de

Datenstruktur = Komplexer Datentyp + Funktionen für diesen komplexen Datentyp

Meist Struktur oder/und Array

```
typedef struct Address
{
    char acLastName[30];
    char acFirstName[30];
    char acStreet[40];
    unsigned int uiZipCode;
    char acTown[40];
}sAddress_t;
typedef sAddress_t* psAddress_t;
```

```
void SortAddressListByLastName(
    psAddress_t psAllAddress,
    unsigned int uiSize);
void PrintAddressList(psAddress_t psAllAddress,
    unsigned int uiSize);
```

2. Datenstrukturen

9

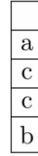
Beispiele Datenstrukturen

Array



Dynamische Arrays

Stack (LIFO)

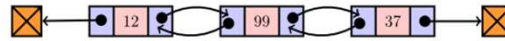


Warteschlange (FIFO)

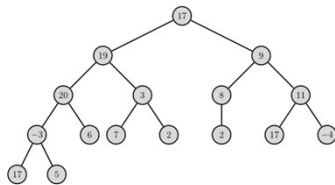
Einfach verkettete Liste



Doppelt verkettete Liste



Bäume



Graph

...

Studiengänge wie AI, AKI, WIN und WINplus haben dafür eine separate Lehrveranstaltung. In diesem Kurs soll nur die einfach verkettete Liste im Detail behandelt (und implementiert) werden.

Bild von <https://martin-thoma.com/ubersicht-uber-datenstrukturen/>

Ingenieur-Informatik und Informatik

Ingenieur-Informatik / Programmierung 2 (C): KE 8: Zeiger 3

Prof. Dr.-Ing. Daniel Fischer - Version 3.0.1

3. Einfach verkettete Liste

10

Liste = Array?

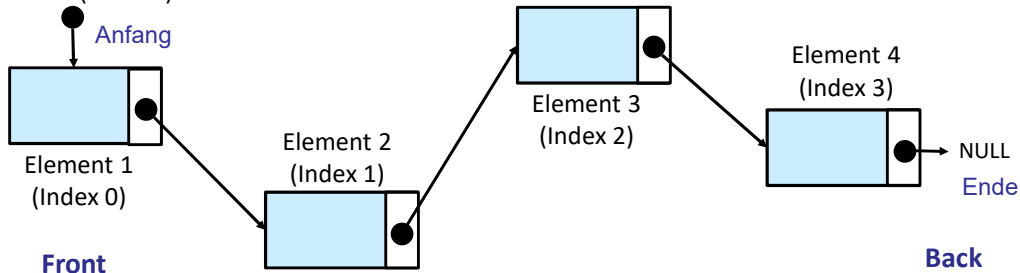
Eine Liste hat gegenüber einem Array zwei **Vorteile**:

- Die benachbarten Elemente müssen nicht nebeneinander im Speicher liegen
- Es ist ein Einfügen und ein Löschen möglich, ohne die restlichen Elemente im Speicher zu verschieben (was bei einem Array der Fall wäre).

Nachteile:

- Zugriff (I/O) dauert länger
- Aufwändiger dies selbst zu implementieren

Anker (Anchor) auf Element 1



Ingenieur-Informatik und Informatik

Ingenieur-Informatik / Programmierung 2 (C): KE 8: Zeiger 3

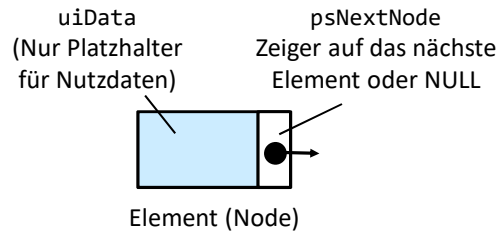
Prof. Dr.-Ing. Daniel Fischer - Version 3.0.1

3. Einfach verkettete Liste

11

Komplexer Datentyp für eine einfach verkettete Liste

```
struct Node
{
    unsigned int uiData;
    struct Node* psNextNode;
};
typedef struct Node sNode_t;
typedef sNode_t* psNode_t;
typedef psNode_t* ppsNode_t;
```



Definition und Initialisierung eines Anchor und vier noch nicht verketteter Elemente

```
psNode_t psAnchorNode = NULL; ← Zeigt der Anker auf NULL, ist die Liste leer
sNode_t asNodes[2] = {{1U, NULL},
                      {2U, NULL}};
sNode_t sNodeA = {3U, NULL};
psNode_t psNode = (psNode_t)malloc(sizeof(sNode_t));
psNode->psNextNode = NULL;
psNode->uiDummyData = 4U;
```

3. Einfach verkettete Liste

12

Funktionen der verketteten Liste

Die Liste soll jeweils eine Kopie der Elemente dynamisch in der Liste angelegen.

```
// API functions for linked list
int InsertElementBack(ppsNode_t ppsAnchor, psNode_t psNewNodePassed);
int InsertElementFront(ppsNode_t ppsAnchor, psNode_t psNewNodePassed);
int InsertElementAt (ppsNode_t ppsAnchor, psNode_t psNewNodePassed,
                    unsigned int uiPos);
int DeleteElementBack(ppsNode_t ppsAnchor);
int DeleteElementFront(ppsNode_t ppsAnchor);
int DeleteElementAt (ppsNode_t ppsAnchor, unsigned int uiPos);
int DeleteList(ppsNode_t ppsAnchor);
int DeleteElementRecursive(psNode_t psNodeDelete);

// Utilities functions for linked list
void PrintElements(psNode_t psNode);
void PrintList(psNode_t psNodeAnchor);
unsigned int GetNumberElements(psNode_t psNode);
```

Funktionen sind in SolutionV301.sln vorgegeben – nur DeleteElementFront und DeleteElementAt sind noch im Labor zu implementieren.

3. Einfach verkettete Liste

13

Zeiger auf Zeiger auf Node und Sanity Checks

Am Beispiel von InsertElementBack soll hier auf zwei Details nochmals eingegangen werden.

```
int InsertElementBack(ppsNode_t ppsAnchor, ppsNode_t psNewNodePassed);
```

Der Anker muss als **Zeiger auf Zeiger** übergeben werden, da der Wert des Zeigers in der aufrufenden Funktion geändert werden muss (siehe Zeiger auf Zeiger).

Sanity Checks:

In den Insert-Funktionen wird immer eine Kopie von psNewNodePassed mit malloc erstellt. In InsertElementBack muss überprüft werden, ob dies überhaupt erfolgreich war. Ebenso muss überprüft werden, ob die Zeiger ppsAnchor und psNewNodePassed ungleich NULL sind. In den Delete-Funktionen muss der Speicher des Elements mit free auch wieder freigegeben werden. Ansonsten kommt es zu Memory Leaks.

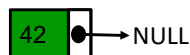
Am Beispiel der verketteten Liste ist erkennbar, dass sicherer Code auch eine höhere zyklomatische Komplexität (Cyclomatic Complexity CC) besitzt. Der Preis für die Sanity Checks ist somit umfangreicher und langsamerer Maschinencode.

3. Einfach verkettete Liste

14

Beispiel Funktion: InsertElementBack

Kopie dieser Node soll am Ende der Liste hinzugefügt werden.



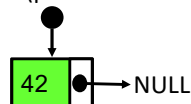
Fallunterscheidungen

Fall 1: Liste ist leer

Anchor (ppsAnchorNode)



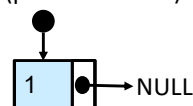
Anchor (ppsAnchorNode)



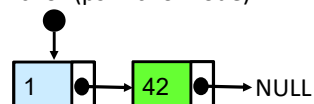
Kopie obiger Node

Fall 2: Liste ist nicht leer

Anchor (ppsAnchorNode)



Anchor (ppsAnchorNode)



Kopie obiger Node

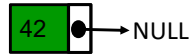
*vorher
nachher*

3. Einfach verkettete Liste

15

Beispiel Funktion: InsertElementFront

Kopie dieser Node soll am Anfang der Liste hinzugefügt werden.



Fallunterscheidungen

Fall 1: Liste ist leer

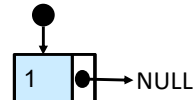
Anchor (psAnchorNode)



NULL

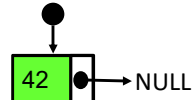
Fall 2: Liste ist nicht leer

Anchor (psAnchorNode)



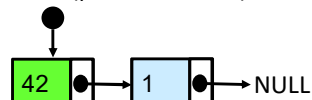
*vorher
nachher*

Anchor (psAnchorNode)



Kopie obiger Node

Anchor (psAnchorNode)



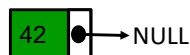
Kopie obiger Node

3. Einfach verkettete Liste

16

Beispiel Funktion: InsertElementAt (1)

Kopie dieser Node soll am Index uiPos eingefügt werden.

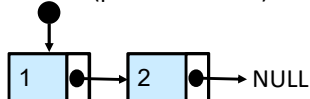


Erstes Element in der Liste hat Index 0, letztes Element hat den Index (Anzahl Elemente - 1)

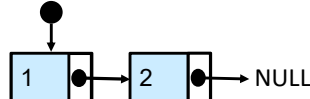
Fallunterscheidungen

Fall 1: uiPos > Anzahl Element

Anchor (psAnchorNode)



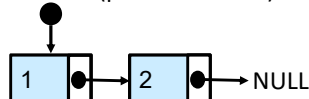
Anchor (psAnchorNode)



Insert nicht möglich

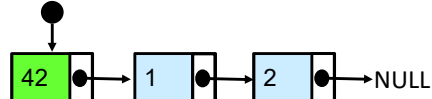
Fall 2: uiPos == 0

Anchor (psAnchorNode)



*vorher
nachher*

Anchor (psAnchorNode)



Kopie obiger Node

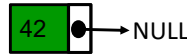
Delegation an InsertElementFront möglich

3. Einfach verkettete Liste

17

Beispiel Funktion: InsertElementAt (2)

Kopie dieser Node soll am Index uiPos eingefügt werden.

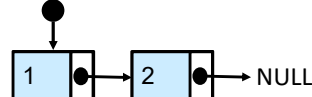


Erstes Element in der Liste hat Index 0, letztes Element hat den Index (Anzahl Elemente – 1)

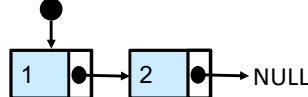
Fallunterscheidungen

Fall 3: uiPos == Anzahl Elemente und uiPos != 0 Fall 4: uiPos > 0 und uiPos < Anzahl Elemente

Anchor (psAnchorNode)

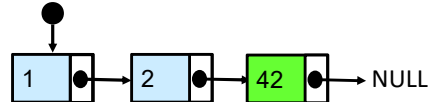


Anchor (psAnchorNode)

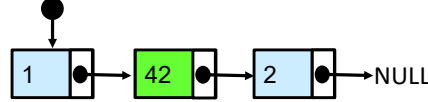


*vorher
nachher*

Anchor (psAnchorNode)



Anchor (psAnchorNode)



Kopie obiger Node

Delegation an InsertElementBack möglich

4. Funktionszeiger

18

Prinzip


Soll oder kann erst zur Laufzeit entschieden werden, welche Funktion aufzurufen ist, so können hierzu Funktionszeiger eingesetzt werden.

Der Maschinencode wie auch die Konstanten liegen im Codesegment CS. Variablen liegen hingegen im Daten-, Stack- oder Heapsegment.

Wie auch bei Zeigern, können Funktionszeiger auf Adressen im Speicher zeigen. Während „normale“ Zeiger auf Variablen zeigen, zeigen Funktionszeiger auf den ersten Maschinenbefehl einer Funktion (Codebeispiel unten: (0x)B510).

„Normale“ Zeiger werden zum Lesen und Beschreiben von Variablen und zum Lesen von konstanten Variablen verwendet.

Funktionszeiger werden zum Aufruf von Funktionen verwendet.

Funktionszeiger 
*Thumb-2 Befehlssatz
(Cortex Microcontroller)*

Adresse	Maschinencode	Assemblercode
0x00001244	B510	PUSH {R4, LR}
0x00001246	4816	LDR R0, [PC, #88]
0x00001248	6800	LDR R0, [R0, #0x00]
0x0000124A	F0400004	ORR R0, R0, #0x04

4. Funktionszeiger

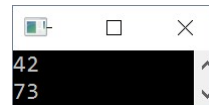
19

Definition Funktionszeiger, Zeigerarithmetik, typedef

```
void (*pf)(int);  
void PrintInteger(int iVal);  
  
int main(void)  
{  
    pf = &PrintInteger;  
    //pf = PrintInteger;  
  
    pf(42);  
    pf(73);  
  
    return 0;  
}  
  
void PrintInteger(int iVal)  
{  
    printf("%d\n", iVal);  
}  
  
typedef void(*pf_t)(int);
```

Definition eines globalen Funktionszeigers mit dem Namen pf. Klammerung beachten! Ohne Klammern wäre es eine Funktionsdeklaration: void* pf(int);
pf zeigt auf die Funktion PrintInteger. Übergabeparameter und Rückgabewert müssen kompatibel sein.

Aufrufe



Statt literaler Konstante kann auch eine konstante Variable oder eine Variable hier übergeben werden.

Werden häufig Funktionszeiger mit gleichen Übergabe- und Rückgabewert definiert, dann empfiehlt sich ein typedef. „Neuer“ Typ: pf_t

4. Funktionszeiger

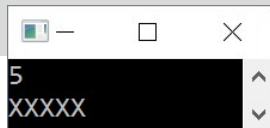
20

Funktionszeiger „umbiegen“ zur Laufzeit

```
void (*pf)(int);  
void PrintInteger(int iVal);  
void PrintXMultipleTimes(int iVal);
```

Funktionszeiger und Funktionen müssen typkompatibel sein

```
int main(void)  
{  
    pf = &PrintInteger;  
    pf(5);  
    pf = &PrintXMultipleTimes;  
    pf(5);  
  
    return 0;  
}
```



```
void PrintInteger(int iVal)  
{  
    printf("%d\n", iVal);  
}  
  
void PrintXMultipleTimes(int iVal)  
{  
    int iCounter;  
  
    for (iCounter = 0;  
         iCounter < iVal;  
         iCounter++)  
    {  
        putchar('X');  
    }  
    putchar('\n');  
}
```

4. Funktionszeiger

21

Übergabe von Funktionszeigern an Funktionen

```
void (*pf)      (int);  
void PrintInteger (int iVal);  
void PrintXMultipleTimes(int iVal);  
  
void CallWithFunctionPointer(int, void(*)(int));  
//void CallWithFunctionPointer(int iVal, void(*pf)(int));
```

Variablenamen
optional

```
pf = &PrintInteger;  
CallWithFunctionPointer(11, pf);
```

```
void CallWithFunctionPointer(int iVal, void(*pfP)(int))  
{  
    pfP(iVal);  
}
```

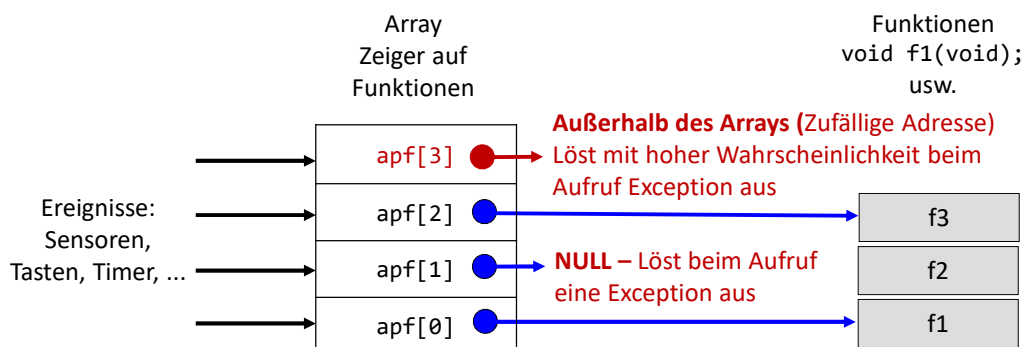


4. Funktionszeiger

22

Anwendungsbeispiel 1: Dynamisches Funktions-Routing

```
void (*apf[3])(void);
```



Funktionszeiger müssen bei der Verwendung immer auf eine vorhandene Funktion zeigen. Nie außerhalb des Arrays zugreifen! Das Arrayelement mit dem Index 3 ist hier im Beispiel schon ungültig (Zeigt auf zufällige Adresse).

4. Funktionszeiger

23

Anwendungsbeispiel 1: Dynamisches Funktions-Routing

```
void (*apf[3])(void);
```

```
void f1(void);
```

```
void f2(void);
```

```
void f3(void);
```

```
void f1(void)
```

```
{
    puts("f1 is called");
}
```

```
void f2(void)
```

```
{
    puts("f2 is called");
}
```

```
void f3(void)
```

```
{
    puts("f3 is called");
}
```

```
apf[0] = &f1;
```

```
apf[1] = &f2;
```

```
apf[2] = &f3;
```

```
apf[0]();
```

```
apf[1]();
```

```
apf[2]();
```

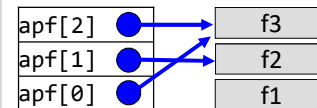
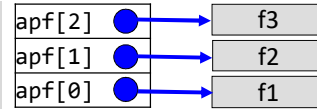
```
apf[0] = &f3;
```

```
apf[0]();
```

```
apf[1]();
```

```
apf[2]();
```

```
f1 ist called
f2 ist called
f3 ist called
f3 ist called
f2 ist called
f3 ist called
```



Zur Laufzeit wird apf[0] geändert. Der Funktionszeiger zeigt jetzt auf die Funktion f3.

Umgangssprachlich wird häufig gesagt: „Der Zeiger wird umgebogen!“

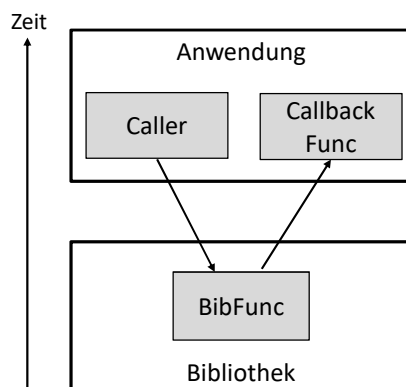
4. Funktionszeiger

24

Anwendungsbeispiel 2: Callback-Funktion

Eine Rückruffunktion (englisch **callback function**) bezeichnet in der Informatik eine **Funktion**, die einer anderen **Funktion**, meist einer vorgefertigten Bibliotheks- oder Betriebssystemfunktion, als Parameter übergeben und von dieser unter definierten Bedingungen mit definierten Argumenten aufgerufen wird.

Quelle: wikipedia.org



In wenigen Fällen muss eine Bibliotheksfunktion eine Funktion aus der Anwendung aufrufen.

Eine Bibliothek ist allerdings schon viel früher (Jahre) entwickelt worden und kennt den Namen der Callback-Funktion nicht.

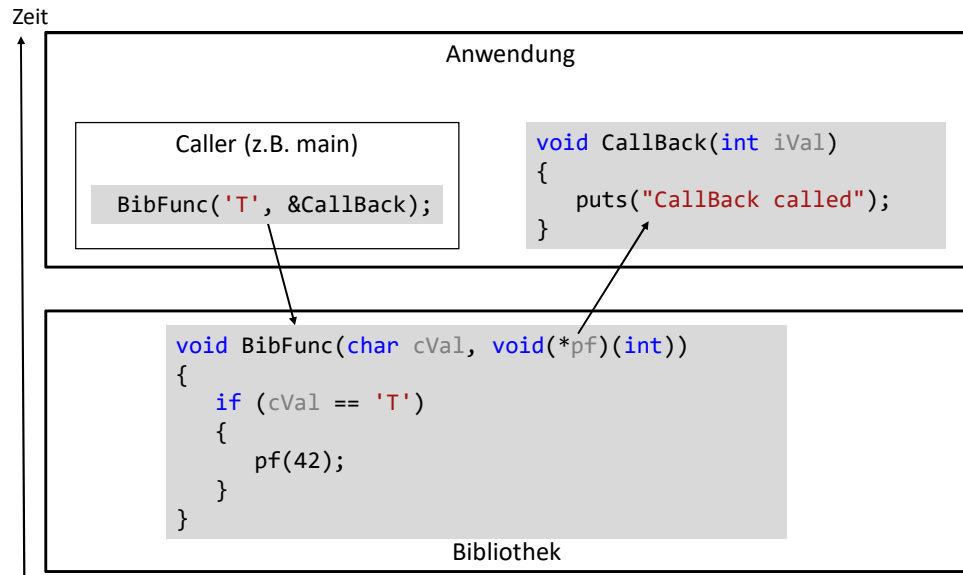
Die Bibliothek kann nicht recompiliert werden, da der Sourcecode nicht vorliegt.

Dieses Problem kann mit Funktionszeigern realisiert werden.

4. Funktionszeiger

25

Anwendungsbeispiel 2: Callback-Funktion



4. Funktionszeiger

26

Callback-Funktion am Beispiel von Quicksort

Quicksort ist eine generische (datentypunabhängige) schnelle rekursive Sortierfunktion aus der stdlib. Für den Vergleich (größer/kleiner) muss die Applikation eine Vergleichsfunktion (datentypabhängig) zur Verfügung stellen. Funktionszeiger `pfCompare` zeigt darauf.

```
void qsort (void* pvData, size_t uiNumberElements, size_t uiSizeElement,
            int (*pfCompare)(const void*, const void*));
```

Compare-Funktion erhält zwei generische Zeiger auf die zwei Variablen und muss folgende Werte zurückgeben:

- <0: Erste Variable ist kleiner
- 0: Variablen sind gleich
- >0: Erste Variable ist größer.

Rechts eine Implementierung für Integer. Für jeden Datentyp, der für `qsort` verwendet wird, ist eine eigene neue Funktion nötig.

```
int CompareInteger(const void* pcv1,
                  const void* pcv2)
{
    int iRet;
    int* piV1 = (int*)pcv1;
    int* piV2 = (int*)pcv2;
    iRet = *piV1 - *piV2;
    return iRet;
}
```

Cast von generischen zu konkreten Zeigern

4. Funktionszeiger

27

Callback-Funktion am Beispiel von Quicksort

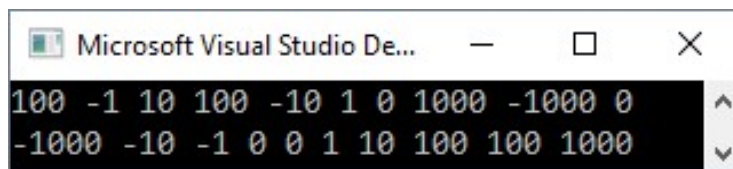
In der Anwendung ist qsort dann recht einfach.

```
int iaArray[10] = { 100, -1, 10, 100, -10, 1, 0, 1000, -1000, 0 };
int (*pfCompareInteger)(const void* pcv1, const void* pcv2);
```

```
// Print Array before Sorting
```

```
pfCompareInteger = &CompareInteger;
qsort(iaArray, 10, sizeof(int), pfCompareInteger);
```

```
// Print Array after Sorting
```



Zusammenfassung KE 8

28

Behandelte Schlüsselwörter in KE 8

Schlüsselwörter C89:

auto ✓	do ✓	goto ✓	signed ✓	unsigned ✓
break ✓	double ✓	if ✓	sizeof ✓	void ✓✓
case ✓	else ✓	int ✓	static ✓	volatile ✓
char ✓	enum ✓	long ✓	struct ✓	while ✓
const ✓	extern ✓	register ✓	switch ✓	
continue ✓	float ✓	return ✓	typedef ✓	
default ✓	for ✓	short ✓	union ✓	

Schlüsselwörter ab C99:

_Bool ✓	_Complex ✓	_Imaginary ✓	inline	restrict ✓
---------	------------	--------------	--------	------------

Schlüsselwörter ab C11:

_Alignas	_Alignof ✓	_Atomic	_Generic	_Noreturn
_Static_assert	_Thread_local			

Keine neuen Schlüsselwörter in KE 8 behandelt.