

Kurseinheit 6: Zeiger 1

1. Memory
2. Einführung in Zeiger
3. Call by Value, Call by Reference

Übersicht KE 6

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

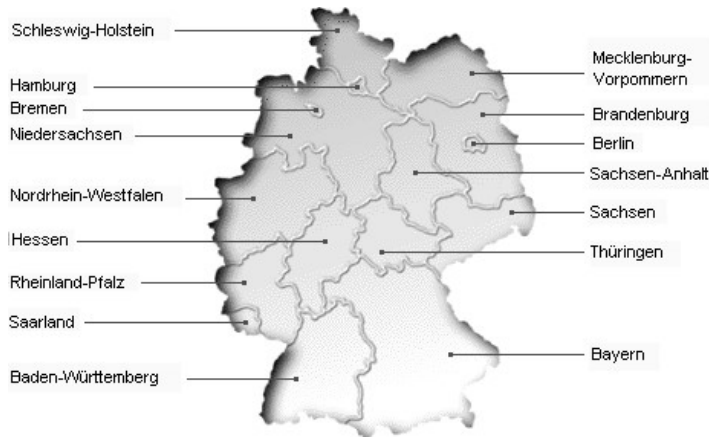
Korrespondierende Kapitel aus *C-Programmierung – Eine Einführung*: Kapitel 8

Zusatzthemen: Memory Map

1. Memory

3

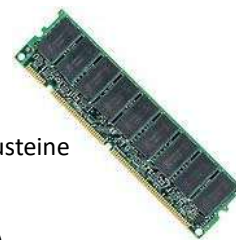
Memory Map



Wie auf einer Landkarte ist auch der Speicher in unterschiedliche Bereiche eingeteilt.

Wo befindet sich der Programmcode und die Daten (Variablen)?

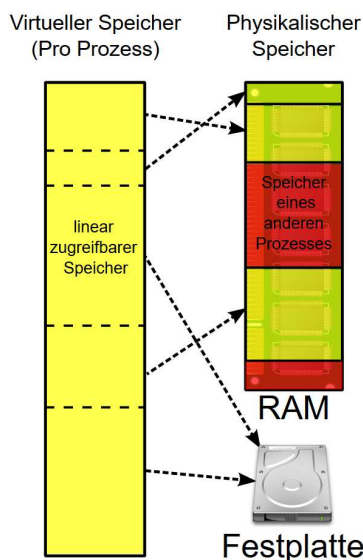
RAM-Bausteine
(Random
Access
Memory)



1. Memory

4

Virtuelle Speicherverwaltung



Moderne Betriebssysteme verwenden eine virtuelle Speicherverwaltung mit einem virtuellen Adressraum (Speicher) pro Prozess.

Ein Prozess ist ein Programm in Ausführung (vereinfacht).

Details z.B.

https://de.wikipedia.org/wiki/Speicherverwaltung#Virtueller_Adressraum

Wichtig für diese Lehrveranstaltung:

Alle Adressen, die im Rahmen vom Debugging gesehen/angezeigt werden, stammen aus dem virtuellen Speicher.

Die gesamte Verwaltung im Hintergrund wird durch das Betriebssystem erledigt und wird im Rahmen dieser Lehrveranstaltung **nicht** behandelt.

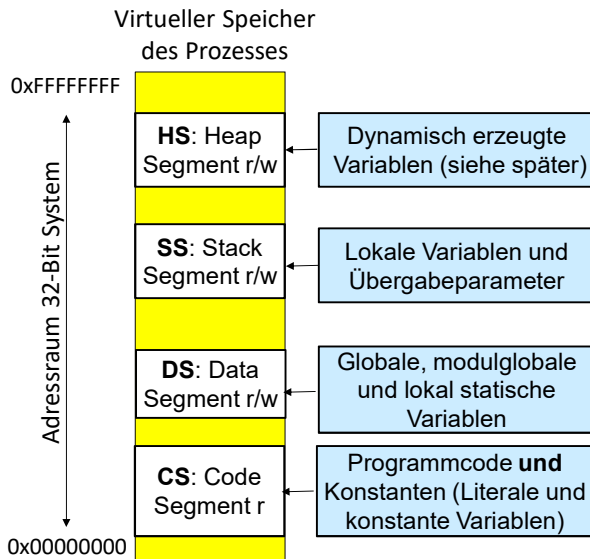
Separate Lehrveranstaltung: Betriebssysteme

Quelle: <https://upload.wikimedia.org/wikipedia/commons/thumb/2/25/Virtual-memory-german.svg/800px-Virtual-memory-german.svg.png>

1. Memory

5

Bereiche des virtuellen Speichers



Wird ein Programm gestartet, so wird ein virtueller Speicher für diesen Prozess angelegt. Es existieren dann vier Bereiche.

Wird mit einem Zeiger auf ungültigen Bereich lesend oder schreibend zugegriffen, oder wird schreibend auf das CS zugegriffen, so wird eine Exception vom Betriebssystem ausgeführt – der Prozess wird beendet.

Zeiger müssen folglich vor deren ersten Verwendung auf gültigen Speicherbereich zeigen!

1. Memory

6

Speicherausrichtung (Alignment)

Der Compiler/das Betriebssystem stellen dabei sicher, dass der Programmcode und die Variablen aligned sind. Mit dem C11 Schlüsselwort `_Alignof` (Operator) kann das Alignment der Datentypen abgefragt werden.

```
printf("\n----_Alignof----\n");
printf("_Alignof(char): %u\n", _Alignof(char));
printf("_Alignof(short int): %u\n", _Alignof(short int));
printf("_Alignof(int): %u\n", _Alignof(int));
printf("_Alignof(long int): %u\n", _Alignof(long int));
printf("_Alignof(long long int): %u\n", _Alignof(long long int));
```

The screenshot shows a terminal window with the following output:

```
----_Alignof----
_Alignof(char): 1
_Alignof(short int): 2
_Alignof(int): 4
_Alignof(long int): 4
_Alignof(long long int): 8
```

1. Memory

7

Little und Big Endian

Die meisten Datentypen haben eine Größe (sizeof) größer als 1. Es gibt verschiedene Möglichkeiten, wie die Bytes im Speicher abgelegt werden. Die zwei bekanntesten Varianten sind **Little** und **Big Endian**. Der Intelprozessor verwendet Little Endian.

```
int iVal = 0xBADDCAFE;
```

Little Endian

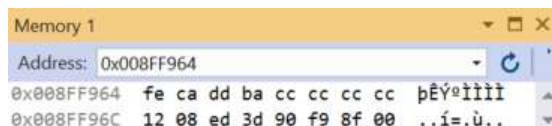
Das niederwertigste Byte befindet sich an der niederwertigsten Adresse

0x008FF967	0xBA
0x008FF966	0xDD
0x008FF965	0xCA
0x008FF964	0xFE

Big Endian

Das höchstwertige Byte befindet sich an der niederwertigsten Adresse

0x008FF967	0xFE
0x008FF966	0xCA
0x008FF965	0xDD
0x008FF964	0xBA



Beispiel für Big Endian mit gleicher Startadresse

2. Einführung in Zeiger

8

Vergleich – zwei Uhren

Worin unterscheiden sich die beiden Uhren?



Daten sind direkt verfügbar.

Variable

```
int iVal;
```



Zeiger zeigen auf die Daten.

Zeiger

```
int* piVal;
```

2. Einführung in Zeiger

9

Bewertung Zeiger

Zeiger sind ein sehr
mächtiges Sprachmittel!



Zeiger sind aber auch
sehr fehleranfällig!



Für was werden Zeiger verwendet?

- Funktionen können mehrere Rückgabewerte haben
- Auf Hardware kann direkt zugegriffen werden
- Es kann dynamisch Speicher allokiert werden – Zeiger zeigen dann auf diesen Speicher
- ...

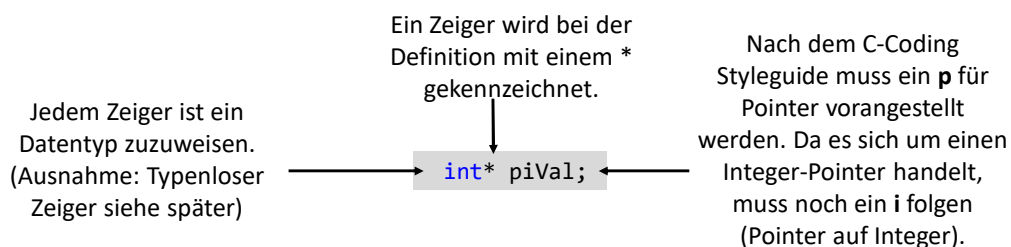
Viele Programmiersprachen (Java, C#, Python) verfügen nicht über Zeiger.

2. Einführung in Zeiger

10

Definition

Mit **Zeiger** (englisch pointer) wird in der Informatik ein Objekt einer Programmiersprache bezeichnet, das eine Speicheradresse zwischenspeichert. Der **Zeiger** referenziert (verweist, zeigt auf) einen Ort im Hauptspeicher des Computers. Hier können Variablen, Objekte oder Programmanweisungen gespeichert sein. Quelle: wikipedia.de



Wichtig: Ein Zeiger muss immer auf eine gültige Adresse zeigen. Steht diese am Anfang noch nicht fest, so muss nach dem C-Coding Styleguide ein Zeiger auf NULL (0) zeigen (DV14).

```
int* piVal = NULL;
```

2. Einführung in Zeiger

11

Variable und Zeiger im Speicher

```
int iVal = 0xBADDCAFE;
// Definition and Initialization piVal
int* piVal = &iVal;
```

```
printf("Wert von iVal:      %X\n", iVal);
printf("Adresse von iVal:  %p\n", &iVal);
printf("Wert von piVal:    %08X\n", (unsigned int)piVal);
printf("Adresse von piVal: %p\n", &piVal);
```

```
C:\Users\Danie...
Wert von iVal:      BADDCAFE
Adresse von iVal:  012FFAD4
Wert von piVal:    012FFAD4
Adresse von piVal: 012FFAC8
```

Compiler legt
die beiden
Variablen iVal
und piVal
aligned im
Speicher ab.

Viererbblock	0x012FFAD7	0xBA
	0x012FFAD6	0xDD
	0x012FFAD5	0xCA
	0x012FFAD4	0xFE

Viererbblock	0x012FFACB	0x01
	0x012FFACA	0x2F
	0x012FFAC9	0xFA
	0x012FFAC8	0xD4

iVal
↑
piVal zeigt auf
iVal (enthält als Wert
die Adresse von iVal)

2. Einführung in Zeiger

12

Variable und Zeiger im Speicher - Memoryfenster

```
int iVal = 0xBADDCAFE;
// Definition and Initialization piVal
int* piVal = &iVal;
```

```
printf("Wert von iVal:      %X\n", iVal);
printf("Adresse von iVal:  %p\n", &iVal);
printf("Wert von piVal:    %08X\n", (unsigned int)piVal);
printf("Adresse von piVal: %p\n", &piVal);
```

```
C:\Users\Danie...
Wert von iVal:      BADDCAFE
Adresse von iVal:  012FFAD4
Wert von piVal:    012FFAD4
Adresse von piVal: 012FFAC8
```

Während dem Debugging kann auch in einem Memoryfenster der Speicherinhalt angezeigt werden. (Debug -> Windows -> Memory 1, Adresse eingeben).

Memory 1		
Address:	0x012FFAC8	
0x012FFAC8	d4 fa 2f 01 cc cc cc cc cc cc cc cc	Ôú/.iiiiiii
0x012FFAD4	fe ca dd ba cc cc cc cc 14 75 68 ac	pÊÿ°iiii.uh~

Achtung: Memoryfenster: Niedrige Adressen stehen hier oben

2. Einführung in Zeiger

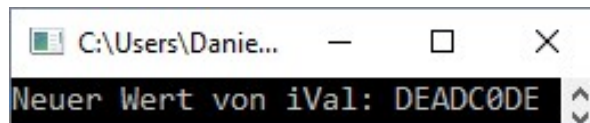
13

Was bringt dieser Aufwand?

```
int iVal = 0xBADDCAFE;  
// Definition and Initialization piVal  
int* piVal = &iVal;
```

Mit einem Zeiger kann die Variable, auf die der Zeiger zeigt, verändert werden. In diesem trivialen Beispiel ist der Vorteil noch nicht ersichtlich!

```
*piVal = 0xDEADC0DE;  
printf("Neuer Wert von iVal: %X\n", iVal);
```



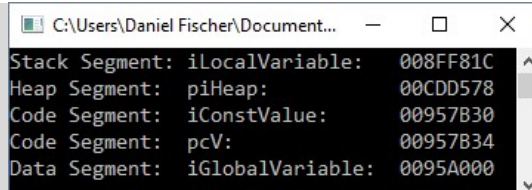
Um den Inhalt der Variablen zu verändern, auf die ein Zeiger zeigt, ist die Nutzung des **Dereferenzierungsoperator** * notwendig. Die Adresse einer Variablen liefert der **Adressoperator** &.

2. Einführung in Zeiger

14

Bereiche des virtuellen Speichers - Beispiel

```
const int iConstValue = 4711;  
int iGlobalVariable = 73;  
  
int main(void)  
{  
    int iLocalVariable = 42;  
    // malloc is explained later  
    int* piHeap = (int*) malloc (100 * sizeof(int));  
    char* pcV = "Hello World";  
  
    printf("Stack Segment: iLocalVariable:  %p\n", &iLocalVariable);  
    printf("Heap Segment:  piHeap:          %08X\n", (unsigned int)piHeap);  
    printf("Code Segment:  iConstValue:      %p\n", &iConstValue);  
    printf("Code Segment:  pcV:                %08X\n", (unsigned int)pcV);  
    printf("Data Segment:  iGlobalVariable:    %p\n", &iGlobalVariable);  
  
    return 0;  
}
```



2. Einführung in Zeiger

15

Wichtiger Merksatz (1) – Einfacher Datentyp

```
int iVal = 0xBADDCAFE;  
// Definition and Initialization piVal  
int* piVal = &iVal;
```

	Wert	Adresse
Variable: int iVal	iVal	&iVal
Zeiger: int* piVal	*piVal	piVal

Niemals Adressen und Werte in einer Zuweisungsoperation mischen („Nie Blau mit Rot mischen!“). Korrekt sind die folgenden Möglichkeiten.

```
iVal = *piVal;
```

```
piVal = &iVal;
```

```
*piVal = iVal;
```

Dereferenzierungsoperator *

Adressoperator &

2. Einführung in Zeiger

16

Wichtiger Merksatz (2) – Structs, Unions und Bitfelder

```
struct Range1 sRange = {0};  
// Definition and Initialization psRange  
struct Range1* psRange = &sRange;
```

	Wert	Adresse
Variable: struct Range1 sRange	sRange	&sRange
Zeiger: struct Range1* psRange	*psRange	psRange

Niemals Adressen und Werte in einer Zuweisungsoperation mischen („Nie Blau mit Rot mischen!“).

Für Structs, Unions und Bitfelder existiert der Zuweisungsoperator! Nicht für Arrays!

```
sRange = *psRange;
```

```
psRange = &iRange
```

```
*psRange = sRange;
```

Dereferenzierungsoperator *

Adressoperator &

2. Einführung in Zeiger

17

Zeiger auf komplexe Datentypen

Zeiger können ebenso auch für komplexe Datentypen wie z.B. Strukturen eingesetzt werden.

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

Grundsätzlich sollte immer ein typedef für die Struktur und ein Pointer auf die Struktur angelegt werden.

Siehe C-Coding Styleguide KD3

```
int main(void)
{
    sAddress_t sAddress = {0};
    // Definition and Initialization
    // of psAddress
    psAddress_t psAddress = &sAddress;

    psAddress->uiZipCode = 77652U;
    strcpy_s(psAddress->acLastName,
        30U, (const char*)"Mustermann");

    return 0;
}
```

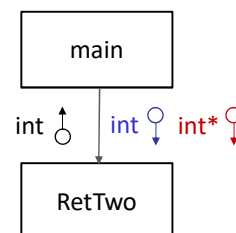
psAddress: C-Coding Styleguide KD2
p: pointer
ps: pointer to struct

3. Call by Value, Call by Reference

18

Problem: Parameterübergabe

Bisher konnte immer nur maximal ein Wert von einer Funktion zurückgegeben werden. Sollen weitere Werte „zurückgegeben“ werden, so müssen diese mittels „**Call by Reference**“ übergeben werden (Zeiger). Bisher wurden immer Variablen „**Call by Value**“ (keine Zeiger) übergeben.



```
int main(void)
{
    int iC;
    int iD;

    iD = RetTwo(42, &iC);
    printf("iC: %d - iD: %d", iC, iD);

    return 0;
}
```

iC: 21 - iD: 43

```
int RetTwo(int iA, int* piB)
{
    int iRet;
    iRet = iA + 1;
    *piB = iRet / 2;

    return iRet;
}
```

piB zeigt auf iC aus main

3. Call by Value, Call by Reference

19

Einfache Datentypen

Call by Value

Kopie wird
übergeben

```
void DoCallByValue(int iA);
void DoCallByReference(int* piA);

int main(void)
{
    int iTest = 42;
    DoCallByValue(iTest);
    printf("iTest: %d\n", iTest);
    DoCallByReference(&iTest);
    printf("iTest: %d\n", iTest);

    return 0;
}
```

Variable

```
void DoCallByValue(int iA)
{
    iA = 73;
}
```

Call by Reference

Verweis auf
Original wird
übergeben.

```
iTest: 42
iTest: 73
```

Zeiger

```
void DoCallByReference(int* piA)
{
    *piA = 73;
}
```

3. Call by Value, Call by Reference

20

Elemente von Strukturen

Call by Value

Kopie wird
übergeben

```
void DoCallByValueElement(float fA);
void DoCallByReferenceElement(float* fpA);

int main(void)
{
    sRect_t sRect = {1.f, 1.f};
    DoCallByValueElement(sRect.fHeight);
    printf("sRect.fHeight: %f\n", sRect.fHeight);
    DoCallByReferenceElement(&(sRect.fHeight));
    printf("sRect.fHeight: %f\n", sRect.fHeight);

    return 0;
}
```

Variable

```
void DoCallByValueElement(float fA)
{
    fA = 2.178f;
}
```

```
struct Rectangle
{
    float fHeight;
    float fWidth;
};
typedef struct Rectangle
sRect_t;
typedef sRect_t* psRect_t;
```

Call by Reference

Verweis auf
Original wird
übergeben.

```
sRect.fHeight: 1.000000
sRect.fHeight: 2.178000
```

Zeiger

```
void DoCallByReferenceElement(float* fpA)
{
    *fpA = 2.178f;
}
```

3. Call by Value, Call by Reference

21

Strukturen

Call by Value

Kopie wird übergeben

```
void DoCallByValueStruct(sRect_t sR);
void DoCallByReferenceStruct(psRect_t psR);

int main(void)
{
    sRect_t sRect = {1.f, 1.f};

    DoCallByValueStruct(sRect);
    printf("sRect.fHeight: %f\n", sRect.fHeight);
    DoCallByReferenceStruct(&sRect);
    printf("sRect.fHeight: %f\n", sRect.fHeight);

    return 0;
}
```

Variable

```
void DoCallByValueStruct(sRect_t sR)
{
    sR.fHeight = 2.178f;
}
```

. Punktoperator

Call by Reference

Verweis auf Original wird übergeben.

```
struct Rectangle
{
    float fHeight;
    float fWidth;
};
typedef struct Rectangle
    sRect_t;
typedef sRect_t* psRect_t;
```

Zeiger

```
void DoCallByReferenceStruct(psRect_t psR)
{
    psR->fHeight = 2.178f;
}
```

-> Zeigeroperator

3. Call by Value, Call by Reference

22

Arrays (eindimensional)

Arrays werden in C immer **Call by Reference** übergeben.

```
void DoCallbyReferenceArray1(double* pdA);
void DoCallbyReferenceArray2(double pdA[]);
void DoCallbyReferenceArray3(double pdA[10]);

int main(void)
{
    double adVal[10] = {0};

    DoCallbyReferenceArray1(adVal);
    DoCallbyReferenceArray2(adVal);
    DoCallbyReferenceArray3(adVal);

    printf("adVal[0]:%lf\nadVal[1]:%lf\nadVal[2]:%lf\n",
        adVal[0], adVal[1], adVal[2]);

    return 0;
}
```

Drei
Möglichkeiten

3. Call by Value, Call by Reference

23

Arrays (eindimensional)

```
void DoCallbyReferenceArray1(double* pdA)
{
    *pdA = 3.1415; // First element
}

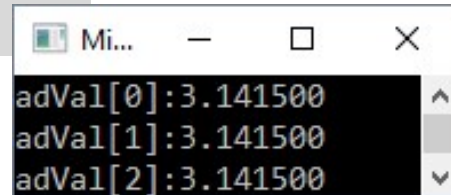
void DoCallbyReferenceArray2(double pdA[])
{
    pdA[1] = 3.1415; // Second element
}

void DoCallbyReferenceArray3(double pdA[10])
{
    pdA[2] = 3.1415; // Third element
}
```

In der Praxis wird noch ein zusätzlicher Parameter mitgegeben, welcher die Anzahl der Elemente im Array enthält, z.B. `size_t uiSize`

Bei char-Arrays wird oft darauf verzichtet, das EOS das Ende des Arrays kennzeichnet.

Nach dem Aufruf der Funktionen sind die ersten drei Elemente verändert. Siehe main vorherige Folie.



adVal[0]:3.141500
adVal[1]:3.141500
adVal[2]:3.141500

3. Call by Value, Call by Reference

24

Arrays (zweidimensional)

Ebenso können zweidimensionale Arrays per Reference übergeben werden.

Definition und Initialisierung

```
unsigned char aucChessboard[8][8] = {0};
```

Funktions-deklarationen

```
void InitChessBoard1(unsigned char acChessBoard[][8]);
void InitChessBoard2(unsigned char acChessBoard[8][8]);
```

Bei der Übergabe eines n-dimensionalen Arrays muss die höchste Dimension nicht unbedingt angegeben werden. Die anderen n-1 Dimensionen müssen beim Aufruf der Funktion bekannt sein.

Funktionsaufrufe

```
InitChessBoard1(aucChessBoard);
InitChessBoard2(aucChessBoard);
```

„Variable length arrays (C99)“ werden derzeit vom Microsoft C-Compiler nicht unterstützt.

```
//void InitChessBoard3(unsigned int uiHeight, unsigned int uiWidth,
//                      unsigned char acChessBoard[uiHeight][uiWidth]);
```

3. Call by Value, Call by Reference

25

Sanity Checks

Bei der Übergabe von Zeigern (Call by Reference) ist zukünftig wie folgt zu verfahren:
Am Anfang der Funktion ist der übergebene Zeiger auf NULL zu überprüfen (C-Coding Styleguide F13). Dies wird Sanity Check genannt. Bei Einhaltung des C-Coding Styleguides kann dann davon ausgegangen werden, dass ein Wert ungleich NULL einen gültigen Zeiger kennzeichnet und dann kann mittels Dereferenzierungsoperator * ohne Exception darauf zugegriffen werden.

Sanity Check engl. bedeutet Plausibilitätsprüfung.

Ohne Sanity Check

```
void DoCallByReference(int* piA)
{
    *piA = 73;
}
```

Exception bei `piA == NULL`

Mit Sanity Check

```
void DoCallByReference(int* piA)
{
    if (piA != NULL)
    {
        *piA = 73;
    }
}
```

Keine Exception bei `piA == NULL`

3. Call by Value, Call by Reference

26

Rückblick: printf und scanf_s

printf darf die übergebenen Parameter nicht verändern. Daher werden diese dort **Call by Value** übergeben. Die Funktion **scanf_s** muss die übergebenen Parameter ändern. Daher werden diese dort mit **Call by Reference** übergeben.

```
scanf_s("%d", &iVal);
scanf_s("%f", &fVal);
scanf_s("%lf", &dVal);
scanf_s(" %c", &cVal, 1); // SPACE TRICK
scanf_s("%9s", &acString, (unsigned int)_countof(acString));
```

Aus KE 3

```
int iVal;
float fVal;
double dVal;
char cVal;
char acString[10];
```

Da **Arrays immer mit Call by Reference übergeben werden**, kann der Adressoperator bei `acString` auch weggelassen werden.

Weitere Besonderheit von `printf` und `scanf_s` (sowie von wenigen weiteren Funktionen):
Die Funktionen haben eine variable Anzahl an Übergabeparametern. Deklaration lautet

```
int printf (const char *format, ...);
int scanf_s (const char *format, ...);
```

Wird hier in dieser Lehrveranstaltung nicht weiter behandelt.

Behandelte Schlüsselwörter in KE 6

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			