

Kurseinheit 5: Komplexe Datentypen

1. Allgemein
2. Typvereinbarungen
3. Aufzählungstypen (Enumerationen)
4. Arrays
5. Strukturen
6. Unions
7. Bitfelder
8. Verschachtelte komplexe Datentypen

Übersicht KE 5

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

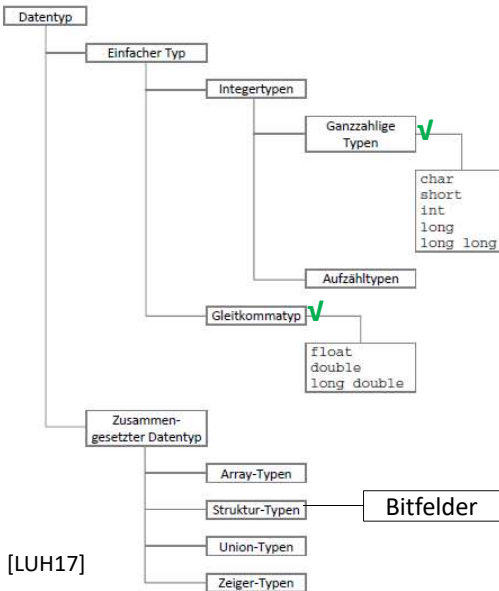
Korrespondierende Kapitel aus *C-Programmierung – Eine Einführung*: Kapitel 7

Zusatzthemen: Bubblesort mit Arrays, Kurzeinführung Speicher, Verschachtelte Datentypen

1. Allgemein

3

Klassifikation der Datentypen



Bei den einfachen Datentypen fehlt jetzt noch der Aufzählungstyp (Enumeration). Bei den zusammengesetzten Datentypen werden Arrays, Strukturen, Bitfelder (Sonderform der Strukturen) und Unions in dieser KE behandelt.

Zeiger-Typen werden in KE 6-8 behandelt.

Ein **komplexer Datentyp** ist in der Regel eine Zusammensetzung aus anderen vorhandenen Datentypen.

https://www.ibm.com/support/knowledgecenter/de/S5GU8G_11.70.0/com.ibm.ddi.doc/ids_ddi_313.htm

Auch Aufzählungstypen (Enumerations) werden als komplexe Datentypen angesehen.

Ingenieur-Informatik, Medizininformatik und Mensch

Ingenieur-Informatik / Programmierung 2 (C): KE 5: Komplexe Datentypen

Prof. Dr.-Ing. Daniel Fischer - Version 3.0.2

1. Allgemein

4

Speicher

Ein 32-Bit Rechner kann auf 2^{32} Speicherstellen zugreifen.

Dieser kann aber auch meist nur in Viererblöcken lesen (und schreiben).

Für die Startadresse des Viererblocks gilt:

Startadresse % 4 == 0.

Idealerweise sollte eine Variable in einem Viererblock liegen, Ansonsten muss der Rechner zweimal lesen (dto. beim Schreiben).

Viererblock	0x0200020F	
	0x0200020E	
	0x0200020D	
	0x0200020C	
Viererblock	0x0200020B	
	0x0200020A	
	0x02000209	0x00
	0x02000208	0x00
Viererblock	0x02000207	0x00
	0x02000206	0x02
	0x02000205	
	0x02000204	
Viererblock	0x02000203	0x00
	0x02000202	0x00
	0x02000201	0x00
	0x02000200	0x01
Adresse der Speicherstelle		Wert (Byte)

```
int iVal1 = 1;
int iVal2 = 2;
```

iVal2 (0x00000002) liegt ungünstig im Speicher. Zwei Lesezugriffe. iVal2 ist **misaligned**.

iVal1 (0x00000001) liegt günstig im Speicher. Nur ein Lesezugriff. iVal1 ist **aligned**.

Ingenieur-Informatik, Medizininformatik und Mensch

Ingenieur-Informatik / Programmierung 2 (C): KE 5: Komplexe Datentypen

Prof. Dr.-Ing. Daniel Fischer - Version 3.0.2

1. Allgemein

5

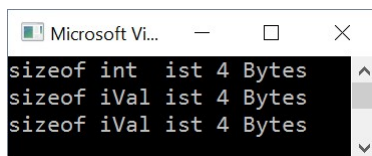
Wiederholung sizeof

Bei sizeof handelt es sich um einen **Operator**, nicht um eine Funktion. Dieser Operator kann **für Datentypen und Variablen** angewendet werden. Der Operator liefert als Rückgabewert die Größe in Bytes (unsigned).

Nur bei Datentypen muss nach sizeof geklammert () werden (nicht bei Variablen).

```
printf("sizeof int ist %Iu Bytes\n", sizeof(int));  
printf("sizeof iVal ist %Iu Bytes\n", sizeof(iVal));  
printf("sizeof iVal ist %Iu Bytes\n", sizeof iVal);
```

%Iu ist Formatierer für Datentyp size_t, welcher vom sizeof-Operator zurückgeliefert wird.



Der sizeof-Operator veranschaulicht bei komplexen Datentypen nochmals deren Größe und indirekt auch wie Variablen von komplexen Datentypen im Speicher abgelegt sind. Oft werden dort aus Performanzgründen (Alignment) noch zusätzliche unbenutzte Bytes eingefügt (Paddingbytes oder Füllbytes).

2. Typvereinbarungen

6

Prinzip typedef

Für jeden Datentyp kann ein anderer Name festgelegt werden. Die Größe und die Darstellung im Speicher ändert sich jedoch nicht.

Metapher: Maske

Erscheint als was „Neues“ – dahinter verbirgt sich etwas „Bekanntes“.



Typedefs erhöhen die Lesbarkeit des Codes und können auch zur Vereinheitlichung beitragen. Typedefs müssen programmglobally bekannt sein und finden sich häufig in Headerdateien. Bei einfachen C-Programmen (nur Main.c) können diese auch über main stehen.

```
typedef int Integer_t;
```

Name **bestehender** Datentyp **Neuer Name** für bestehenden Datentyp

Nach dem C-Coding Styleguide sollte ein „_t“ im neuen Name eingefügt werden. So ist sofort ersichtlich, dass es sich um ein Typedef handelt.

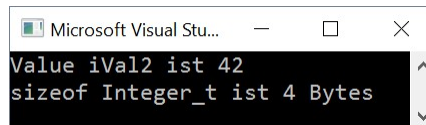
2. Typvereinbarungen

7

typedef - Beispiele

```
typedef int Integer_t;
```

```
Integer_t iVal2 = 42;
printf("Value iVal2 ist %d\n", iVal2);
printf("sizeof Integer_t ist %Iu Bytes\n", sizeof(Integer_t));
```



Im Umfeld von Embedded Systems werden Datentypen basierend auf der Headerdatei `<stdint.h>` verwendet. Hierbei wird die Größe in Bit sowie die „Signedness“ im neuen Namen hinterlegt.

<code>typedef signed char</code>	<code>int8_t;</code>	<code>typedef unsigned char</code>	<code>uint8_t;</code>
<code>typedef short</code>	<code>int16_t;</code>	<code>typedef unsigned short</code>	<code>uint16_t;</code>
<code>typedef int</code>	<code>int32_t;</code>	<code>typedef unsigned int</code>	<code>uint32_t;</code>
<code>typedef long long</code>	<code>int64_t;</code>	<code>typedef unsigned long long</code>	<code>uint64_t;</code>

3. Aufzählungstypen (Enumerationen)

8

Prinzip Enumeration

Oft werden Integer für thematisch zusammenhängende Konstanten verwendet.

Beispiel: Wochentage Mo, Tu, We, Th, Fr, Sa, Su

Dies wird schnell im Code unübersichtlich:

```
if (iDay == 3)
```

Die Verwendung von Enumerationen macht den Code **lesbarer und wartbarer**. „Hinter“ einer Enumeration verbirgt sich jedoch meist der Datentyp `int`.

Deklaration einer Enumeration

```
enum DayOfWeek {Mo = 0, Tu, We, Th, Fr, Sa, Su};
```

Automatisch werden „intern“ die Bezeichner durchnummeriert (von 0 aufsteigend). Die zugeordneten Werte können aber auch beeinflusst werden.

```
enum DayOfWeek {Mo = 0, Tu = 1, We, Th, Fr, Sa = 10, Su};
```

↖ ↗ ↖ ↗ ↖ ↗ ↖ ↗
+1 +1 +1 +1

Definition und Initialisierung einer Enumerationsvariablen

```
enum DayOfWeek eDayOfWeek1 = Mo;
```

Enumerationsvariablen beginnen mit einem „e“. C-Coding Styleguide KD1

3. Aufzählungstypen (Enumerationen)

9

Enumerationen in Kontrollstrukturen

```
switch (eDayOfWeek1)
{
    case Mo:
        printf("Very hard day!\n");
        break;
    case Tu: // Fall through
    case We: // Fall through
    case Th: // Fall through
    case Fr:
        printf("Hard day!\n");
        break;
    case Sa: // Fall through
    case Su:
        printf("Party day!\n");
        break;
    default:
        printf("Irregular day!\n");
        break;
}
```

Kontrollstrukturen werden dadurch auch lesbarer.

```
if (eDayOfWeek1 == Mo)
{
    // StartWeekendTalk(void);
}
```

Es können aber auch die zugeordneten Werte abgefragt werden. Dies schadet allerdings der Lesbarkeit und ist fehleranfällig, falls später die Zuordnung geändert wird.

```
// Don't do it
if (eDayOfWeek1 == 0)
{
    // StartWeekendTalk(void);
}
```

4. Arrays

10

Prinzip Arrays

In Arrays können mehrere Elemente eines gleichen Datentyps abgelegt werden.

```
int aiArr1[10];
```

aiArr1[0] aiArr1[1] ... aiArr1[9] ~~aiArr1[10]~~

Häufige Fehlerquelle in C

C-Coding Styleguide: a für Array, i für Integer („Array of Integer“)

Zugriffe können über den Index erfolgen. Der Index läuft hier von 0 bis **9**!

```
for (iIndex = 0; iIndex < 10; iIndex++)
{
    aiArr1[iIndex] = 100 - iIndex;
}
```

Welche Werte stehen nun im Array aiArr1?

Welchen Wert hat iIndex **nach** der Schleife?

4. Arrays

11

Arrays: Definition, Initialisierung und Zugriff

Definition

```
int aiArr2[4];
```

aiArr2[0]	aiArr2[1]	aiArr2[2]	aiArr2[3]
-----------	-----------	-----------	-----------

Definition und Initialisierung

```
int aiArr2[4] = {0}; // All Elements are 0
```

```
int aiArr2[4] = {1, 73, 42, -99}; // Elements are initialized
```

```
int aiArr2[4] = {1, 73}; // First 2 Elements are initialized - next are 0
```

```
int aiArr2[] = {1, 73, 42, -99}; // No. of Elements generated by Compiler
```

```
int aiArr2[4] = {[0] = 1, [2] = 42}; // C99 - other elements are 0
```

Zugriff (R/W)

```
iVal = aiArr2[2];
```

```
aiArr2[2] = 99999;
```

4. Arrays

12

Arrays im Speicher

```
int aiArr2[4] = {1, 73, 42, -99}; // Elements are initialized
```

Negative Zahlen

werden im
Zweierkomplement
dargestellt.

$99_{10} =$
 $0x00000063$

Bits invertieren:
 $0xFFFFF9C$

Eins addieren:
 $-99_{10} =$
 $0xFFFFF9D$

Viererbblock	0x0200020F	0xFF	aiArr2[3]
	0x0200020E	0xFF	
	0x0200020D	0xFF	
	0x0200020C	0x9D	
Viererbblock	0x0200020B	0x00	aiArr2[2]
	0x0200020A	0x00	
	0x02000209	0x00	
	0x02000208	0x2A	
Viererbblock	0x02000207	0x00	aiArr2[1]
	0x02000206	0x00	
	0x02000205	0x00	
	0x02000204	0x49	
Viererbblock	0x02000203	0x00	aiArr2[0]
	0x02000202	0x00	
	0x02000201	0x00	
	0x02000200	0x01	

4. Arrays

13

Zweidimensionale Arrays

Definition

```
int aiArr2D[2][3];
```


y x

Definition und Initialisierung

```
int aiArr2D[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

y x

aiArr2D[0][0]	1	aiArr2D[0][1]	2	aiArr2D[0][2]	3
aiArr2D[1][0]	4	aiArr2D[1][1]	5	aiArr2D[1][2]	6



Zugriff (R/W)

```
iVal = aiArr2D[1][1];
```

```
aiArr2D[1][1] = 99999;
```

Beispiele für die Anwendung von zweidimensionalen Arrays: Schachbrett mit Figuren, Matrizen, Bilder, ...

4. Arrays

14

Dreidimensionale Arrays

Definition

```
int aiArr3D[4][2][3];
```

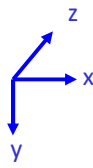
z y x

Definition und Initialisierung

```
int aiArr3D[4][2][3] = {{{ 1, 2, 3}, { 4, 5, 6}},  
                          {{ 7, 8, 9}, {10, 11, 12}},  
                          {{13, 14, 15}, {16, 17, 18}},  
                          {{19, 20, 21}, {22, 23, 24}}};
```

z y x

aiArr3D[3][0][0]	19	aiArr3D[3][0][1]	20	aiArr3D[3][0][2]	21
aiArr3D[2][0][0]	13	aiArr3D[2][0][1]	14	aiArr3D[2][0][2]	15
aiArr3D[1][0][0]	7	aiArr3D[1][0][1]	8	aiArr3D[1][0][2]	9
aiArr3D[0][0][0]	1	aiArr3D[0][0][1]	2	aiArr3D[0][0][2]	3
aiArr3D[0][1][0]	4	aiArr3D[0][1][1]	5	aiArr3D[0][1][2]	6



Zugriff (R/W)

```
iVal = aiArr3D[0][1][1];
```

```
aiArr3D[0][1][1] = iVal;
```

4. Arrays

15

Beispiel Sortieren mit BubbleSort

```
int aiArr3[10] = { 1, 7, -4, 5, -1, -99, 100, -73, 2, -1 };
```

```
for (iY = 0; iY < (10 - 1); iY++)
{
    for (iX = 0; iX < (10 - 1 - iY); iX++)
    {
        if (aiArr3[iX] < aiArr3[iX + 1])
        {
            iDum = aiArr3[iX];
            aiArr3[iX] = aiArr3[iX + 1];
            aiArr3[iX + 1] = iDum;
        }
    }
}
```

BubbleSort ist ein iteratives (schrittweise, wiederholendes) Sortierverfahren.

} Hier das typische „Vertausche-Idiom“ (swap)

Bubble Sort – Sortverfahren 6: <https://www.youtube.com/watch?v=qtXb0QnOceY>

Bubble-sort with Hungarian folk dance: <https://www.youtube.com/watch?v=lyZQPjUT5B4>

4. Arrays

16

Char-Arrays - Zeichenketten

Char-Arrays (Zeichenketten) nehmen in C eine **Sonderrolle** ein. Ein Char-Array muss mit einem speziellem Zeichen, dem **EOS-Zeichen (0x00)** abgeschlossen werden. Funktionen (z.B. aus string.h – siehe später) erwarten dieses Zeichen. So druckt auch printf solange aus, bis es auf ein EOS-Zeichen trifft.

Definition

```
char acStr[12];
```

Definition und Initialisierung

```
char acStr[12] = "Hello World"; // Size calculated by Programmer
```

```
char acStr[] = "Hello World"; // Size calculated by Compiler
```

Zugriff (R/W)

```
cCh = acStr[10];
```

```
acStr[10] = cCh;
```

```
acStr[9] = 'E';
```


4. Arrays


17

Char-Arrays – Zeichenketten - Speicher

```
char acStr[12] = "Hello World"; // Size calculated by Programmer
```

Viererblock	0x0200020F		}	Gehört nicht mehr zum Array acStr
	0x0200020E			
	0x0200020D			
	0x0200020C			
Viererblock	0x0200020B	0x00 EOS	acStr[11]	
	0x0200020A	0x64	acStr[10]	
	0x02000209	0x6C	acStr[9]	
	0x02000208	0x72	acStr[8]	
Viererblock	0x02000207	0x6F	acStr[7]	
	0x02000206	0x57	acStr[6]	
	0x02000205	0x20	acStr[5]	
	0x02000204	0x6F	acStr[4]	
Viererblock	0x02000203	0x6C	acStr[3]	
	0x02000202	0x6C	acStr[2]	
	0x02000201	0x65	acStr[1]	
	0x02000200	0x48	acStr[0]	

Dez	Hex	Zeichen	Dez	Hex	Zeichen
64	40	@	96	60	`
65	41	A	97	61	a
66	42	B	98	62	b
67	43	C	99	63	c
68	44	D	100	64	d
69	45	E	101	65	e
70	46	F	102	66	f
71	47	G	103	67	g
72	48	H	104	68	h
73	49	I	105	69	i
74	4A	J	106	6A	j
75	4B	K	107	6B	k
76	4C	L	108	6C	l
77	4D	M	109	6D	m
78	4E	N	110	6E	n
79	4F	O	111	6F	o
80	50	P	112	70	p
81	51	Q	113	71	q
82	52	R	114	72	r
83	53	S	115	73	s
84	54	T	116	74	t
85	55	U	117	75	u
86	56	V	118	76	v
87	57	W	119	77	w
88	58	X	120	78	x
89	59	Y	121	79	y
90	5A	Z	122	7A	z

 Hochschule Mittelhessen
University of Applied Sciences

Ingenieur-Informatik / Programmierung 2 (C); KE 5: Komplexe Datentypen

Prof. Dr.-Ing. Daniel Fischer - Version 3.0

4. Arrays

18

Char-Arrays – Zeichenketten - Kopieren

Mit dem Zuweisungsoperator = können nur einzelne Zeichen kopiert werden. Um ganze Zeichenketten (inklusive dem EOS) zu kopieren, stehen Funktionen in <string.h> zur Verfügung.

```
char acStrDestination[12];  
char acStrSource[12];
```

Hier schon erkennbar: Beide Zeichenketten sind nicht gleich lang! Dies ist zu vermeiden.

Funktionen aus <string.h>:

```
char* strcpy (char* pcDestination, const char* pcSource);
```

Kopiert alle Zeichen (inklusive EOS). Ist die Zeichenketten pcDestination zu klein, so kommt es zu einem Overflow. Daher gilt **strcpy** als **unsichere Funktion**.

```
errno_t strcpy_s (char* pcDestination, rsize_t uSize,  
const char* pcSource);
```

Besser ist strcpy_s: In uSize ist die maximale Anzahl zu kopierender Zeichen anzugeben. Gibt bei Erfolg ein Null (0) zurück. Die Funktion überprüft die Größe der Zeichenketten. Das zusätzliche EOS wird dabei berücksichtigt. Fehlercode (errno_t) wird zurückgegeben.

```
strcpy_s(acStrSource, 12U, (const char*)"Hello World");  
strcpy_s(acStrDestination, 12U, acStrSource);
```

5. Strukturen

19

Prinzip

Einzelne Variablen stehen in einer Beziehung zueinander.

```
char acLastName[30];  
int iA;  
char acFirstName[30];  
char acInputString[255];  
char acStreet[40];  
unsigned int uiZipCode;  
char acTown[40];
```

Diese 5 Variablen gehören logisch zusammen und repräsentieren eine Adresse. Diese Adresse ist als solche hier nicht oder nur schwer erkenn- und handhabbar!



Besser:
Zusammenpacken

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



5. Strukturen

20

Strukturen: Deklaration, Definition, Initialisierung und Zugriff

Deklaration

```
struct Range1  
{  
    int iMin;  
    char cCategory;  
    int iMax;  
    char acUnits[3];  
};
```

```
typedef struct Range1 sRange1_t;
```

Mit einem typedef ist später bei Definitionen und Deklarationen das Schlüsselwort struct nicht mehr notwendig.

Definition

```
struct Range1 sRange1A;  
sRange1_t sRange1B;
```

Mit und ohne typedef. Strukturvariablen beginnen mit einem „s“. C-Coding Styleguide KD1

Definition und Initialisierung

```
struct Range1 sRange1A = {0}; // All elements are zero
```

```
struct Range1 sRange1A = {0, 'H', 10, "m"};
```

Zugriff (R/W) mit Punktoperator

```
iA = sRange1A.iMax;
```

```
sRange1A.cCategory = 'D';  
sRange1A.acUnits[0] = 0x00;  
strcpy_s(sRange1A.acUnits, 3U, "mm");
```

5. Strukturen

21

Strukturen – Konsole I/O

```
printf("iMin: %i\n", sRange1A.iMin);
printf("iMax: %i\n", sRange1A.iMax);
printf("cCategory: %c\n", sRange1A.cCategory);
printf("acUnits: %s\n", sRange1A.acUnits);
```

```
iMin: 0
iMax: 10
cCategory: D
acUnits: mm
```

```
printf("Please enter iMin:");
scanf_s("%i", &sRange1A.iMin);
printf("Please enter iMax:");
scanf_s("%i", &sRange1A.iMax);
printf("Please enter cCategory:");
scanf_s(" %c", &sRange1A.cCategory, 1); //Trick with SPACE
printf("Please enter acUnits:");
scanf_s("%2s", sRange1A.acUnits, (unsigned int)_countof(sRange1A.acUnits));
// ->without & is also possible with %s
```

Zusätzlicher Wert bei %c und %s notwendig – Anzahl Zeichen.

_countof (MS spezifisch) stammt aus der <stdlib.h> und liefert die Anzahl der Elemente im Array.

5. Strukturen

22

Strukturen im Speicher – aligned (default)

Viererblock	0x0200020F	Paddingbyte	sRange1B. acUnits	{	struct Range1 { int iMin; char cCategory; int iMax; char acUnits[3]; };	
	0x0200020E	xx				
	0x0200020D	0x00 EOS				
	0x0200020C	0x6D				
Viererblock	0x0200020B	0x00	sRange1B. iMax	{	struct Range1 sRange1B = {1, 'H', 10, "m"};	
	0x0200020A	0x00				
	0x02000209	0x00				
	0x02000208	0x0A				
Viererblock	0x02000207	Paddingbyte	sRange1B. cCategory	{	Der Compiler fügt als default immer leere Füllbytes (Grau) ein, damit ein Element mit einem Lesevorgang gelesen werden kann. Diese Variante benötigt mehr Speicher , ist aber schneller .	
	0x02000206	oder				
	0x02000205	Füllbyte				
	0x02000204	0x48 ←				
Viererblock	0x02000203	0x00	sRange1B. iMin	{		
	0x02000202	0x00				
	0x02000201	0x00				
	0x02000200	0x01				

```
printf("Sizeof sRange1B ist: %Iu\n", sizeof(sRange1B));
```

Ergebnis ?

5. Strukturen
23

Strukturen im Speicher – packed

Viererbblock	0x0200020F		
	0x0200020E		
	0x0200020D		
	0x0200020C		
Viererbblock	0x0200020B	xx	sRange2B. acUnits
	0x0200020A	0x00 EOS	
	0x02000209	0x6D	
	0x02000208	0x00	
Viererbblock	0x02000207	0x00	sRange2B. iMax
	0x02000206	0x00	
	0x02000205	0x0A	
	0x02000204	0x48 ←	
Viererbblock	0x02000203	0x00	sRange2B. cCategory
	0x02000202	0x00	
	0x02000201	0x00	
	0x02000200	0x01	

```
#pragma pack(1)
struct Range2
{
    int iMin;
    char cCategory;
    int iMax;
    char acUnits[3];
};

struct Range2 sRange2B =
    {1, 'H', 10, "m"};
```

Der Compiler fügte keine Füllbytes ein. Ggf. ist beim Zugriff zweimal zu lesen. Diese Variante benötigt **weniger Speicher**, ist aber **langsamer**.

```
printf("Sizeof sRange2B ist: %Iu\n", sizeof(sRange2B));
```

Ergebnis ?

Ingenieur-Informatik / Programmierung 2 (C): KE 5: Komplexe Datentypen
Prof. Dr.-Ing. Daniel Fischer - Version 3.0.2

5. Strukturen
24

Strukturen im Speicher – Optimierte Struktur

Viererbblock	0x0200020F		
	0x0200020E		
	0x0200020D		
	0x0200020C		
Viererbblock	0x0200020B	xx	sRange3B. acUnits
	0x0200020A	0x00 EOS	
	0x02000209	0x6D	
	0x02000208	0x48 ←	
Viererbblock	0x02000207	0x00	sRange3B. cCategory
	0x02000206	0x00	
	0x02000205	0x00	
	0x02000204	0x0A	
Viererbblock	0x02000203	0x00	sRange3B. iMax
	0x02000202	0x00	
	0x02000201	0x00	
	0x02000200	0x01	

```
struct Range3
{
    int iMin;
    int iMax;
    char cCategory;
    char acUnits[3];
};

struct Range3 sRange3B =
    {1, 10, 'H', "m"};
```

Diese (vom Programmierer) optimierte Struktur benötigt **weniger Speicher** und ist auch **schneller**, da beim Lesen eines Elementes immer nur ein Lesevorgang nötig ist.

```
printf("Sizeof sRange3B ist: %Iu\n", sizeof(sRange3B));
```

Ergebnis ?

Ingenieur-Informatik / Programmierung 2 (C): KE 5: Komplexe Datentypen
Prof. Dr.-Ing. Daniel Fischer - Version 3.0.2

6. Unions

25

Prinzip und Union im Speicher

Ein Union ist wie eine Struktur aufgebaut. Einziger Unterschied ist, dass sich die Elemente den Speicher sozusagen teilen.

```
union Data
{
    char acName[12];      ①
    unsigned int uiNumber; ②
    short int siKey;      ③
};
```

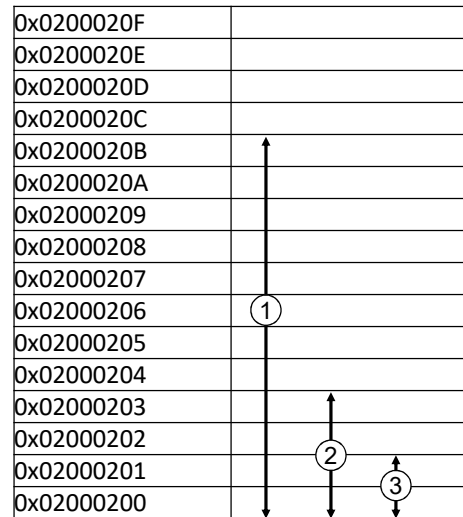
```
union Data uData;
```

```
printf("Sizeof uData ist: %Iu\n",
sizeof(uData));
```

Ergebnis ?

Anwendung von Unions:

- Speicherplatz sparen
- Konvertierung von Datentypen



6. Unions

26

Unions: Deklaration, Definition, Initialisierung und Zugriff

Deklaration

```
union Data
{
    char acName[12];
    unsigned int uiNumber;
    short int siKey;
};
```

```
typedef union Data uData_t;
```

Mit einem typedef ist später bei Definitionen und Deklarationen das Schlüsselwort union nicht mehr notwendig.

Definition

```
union Data uData;
uData_t uData2;
```

Mit und ohne typedef. Unionvariablen beginnen mit einem „u“. C-Coding Styleguide KD1

Definition und Initialisierung

```
union Data uData = {"Hello World"}; // Only one element can be initialized
```

```
union Data uData = { .uiNumber=6000 }; // Element uiNumber is initialized
```

Zugriff (R/W)

```
siVal = uData.siKey;
```

```
uData.siKey = 30000;
uData.uiNumber = 0xBADDCAFE;
strcpy_s(uData.acName, 12U, "Hello World");
```

7. Bitfelder

27

Prinzip

Einzelne Bits oder **Bitgruppen** lassen sich innerhalb eines structs zu **Bitfeldern** zusammenfassen. Auf diese Bitfelder kann dann zugegriffen werden.

```
struct Date
{
    unsigned int uiDay : 5;
    unsigned int uiMonth : 4;
    unsigned int uiYear : 12;
};
```

Laut C-Coding Styleguide muss in diesem Kurs als Datentyp für Elemente eines Bitfeldes ein **unsigned int** verwendet werden (KD6).

Der Compiler packt dann die einzelnen Bits oder Bitgruppen in möglichst wenige unsigned int Variablen zusammen. Dies spart Speicher, jedoch ist der Zugriff etwas langsamer, da intern durch weitere Maschinenbefehle die Bits an der richtigen Stellen gelesen oder geschrieben werden müssen. Bitfelder machen den Code im Gegensatz zu bitorientierten Operatoren lesbarer. Bitfelder werden verwendet um:

- Auf einzelne Bits zuzugreifen
- Speicher einzusparen
- Status eines Gerätes (meist einzelne Bits/-gruppen) in einem Bitfeld abzuspeichern

```
printf("Sizeof bfDateA ist: %Iu\n", sizeof(bfDateA));
```

Ergebnis ?

7. Bitfelder

28

Bitfelder: Deklaration, Definition, Initialisierung und Zugriff

Deklaration

```
struct Date
{
    unsigned int uiDay : 5;
    unsigned int uiMonth : 4;
    unsigned int uiYear : 12;
};
```

```
typedef struct Date bfDate_t;
```

Mit einem typedef ist später bei Definitionen und Deklarationen das Schlüsselwort struct nicht mehr notwendig.

Definition

```
struct Date bfDateA;
bfDate_t bfDateB;
```

Mit und ohne typedef. Bitfeldervariablen beginnen mit einem „bf“. C-Coding Styleguide KD1

Definition und Initialisierung

```
struct Date bfDateA = {0}; // All elements are zero
```

```
struct Date bfDateA = {9U, 8U, 2020U};
```

Zugriff (R/W)

```
uiVal = bfDateA.uiDay;
```

```
bfDateA.uiMonth = uiVal;
bfDateA.uiYear = 2021U;
```

7. Bitfelder

29

Bitfelder im Speicher

```
struct Date
{
    unsigned int uiDay : 5;
    unsigned int uiMonth : 4;
    unsigned int uiYear : 12;
};
```

```
struct Date bfDateA;
```

Wie die Bits im Speicher abgelegt werden, hängt vom Compiler ab. Daher **nie** noch mit bitorientierten Operatoren darauf zugreifen. Eine Möglichkeit (MS C Compiler) wie diese abgelegt wurden, ist rechts erkennbar.

■ uiDay
■ uiMonth
■ uiYear

0x0200020F	
0x0200020E	
0x0200020D	
0x0200020C	
0x0200020B	
0x0200020A	
0x02000209	
0x02000208	
0x02000207	
0x02000206	
0x02000205	
0x02000204	
0x02000203	unused
0x02000202	unused
0x02000201	
0x02000200	

8. Verschachtelte komplexe Datentypen

30

Prinzip: Struct im Struct und Bitfeld im Struct

Die komplexen Datentypen lassen sich beliebig miteinander kombinieren. Ein komplexer Datentypen enthält dabei einen weiteren komplexen Datentyp als Element. Der Zugriff auf das Kind-Element geschieht dabei über den **Punktoperator**.

```
struct Time
{
    unsigned int uiHours;
    unsigned int uiMinutes;
    unsigned int uiSecs;
};
```

```
struct TimeStamp
{
    struct Time sTime;
    struct Date bfDate;
};
typedef struct TimeStamp sTimeStamp_t;
```

```
struct Date
{
    unsigned int uiDay : 5;
    unsigned int uiMonth : 4;
    unsigned int uiYear : 12;
};
```

```
sTimeStamp_t sTimeStamp = {0};
sTimeStamp.bfDate.uiDay = 1U;
```

Punktoperator

8. Verschachtelte komplexe Datentypen

31

Prinzip: Arrays von Strukturen

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

```
struct Address
    asCustomers[100];
```

```
asCustomers[42].uiZipCode =
    77652U;
```

↑
Punktoperator

```
struct TimeStamp
{
    struct Time sTime;
    struct Date sDate;
};
typedef struct Timestamp sTimeStamp_t;
```

```
sTimeStamp_t asTimeStampAll[100];
```

```
asTimeStampAll[73].sTime.uiSecs = 0U;
```

↑ ↑
Punktoperator

8. Verschachtelte komplexe Datentypen

32

Praxisbeispiel: Struct und Variable in einem Union

```
typedef union
{
    struct
    {
        uint32_t ISR:9; /*!< bit: 0.. 8 Exception number*/
        uint32_t _reserved0:1; /*!< bit: 9 Reserved*/
        uint32_t ICI_IT_1:6; /*!< bit: 10..15 ICI/IT part 1*/
        uint32_t _reserved1:8; /*!< bit: 16..23 Reserved*/
        uint32_t T:1; /*!< bit: 24 Thumb bit*/
        uint32_t ICI_IT_2:2; /*!< bit: 25..26 ICI/IT part 2*/
        uint32_t Q:1; /*!< bit: 27 Saturation condition flag*/
        uint32_t V:1; /*!< bit: 28 Overflow condition code flag*/
        uint32_t C:1; /*!< bit: 29 Carry condition code flag*/
        uint32_t Z:1; /*!< bit: 30 Zero condition code flag*/
        uint32_t N:1; /*!< bit: 31 Negative condition code flag*/
    }b;
    uint32_t w; /*!< Type used for word access*/
}xPSR_Type;
```

Bibliothek CMSIS (Cortex-M3 Microcontroller) – Vorlesung Embedded Systems 1
Hier: typedef direkt mit Union-Deklaration kombiniert

Behandelte Schlüsselwörter in KE 5

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			