# Vorlesungsskript zu „Vertiefung Programmieren“ Embedded C/C++


**Dozent:** Dipl.-Inf. (FH) Andreas Schmidt

DHBW
Duale Hochschule
Baden-Württemberg

# Embedded C/C++

## Pointer

# Pointer

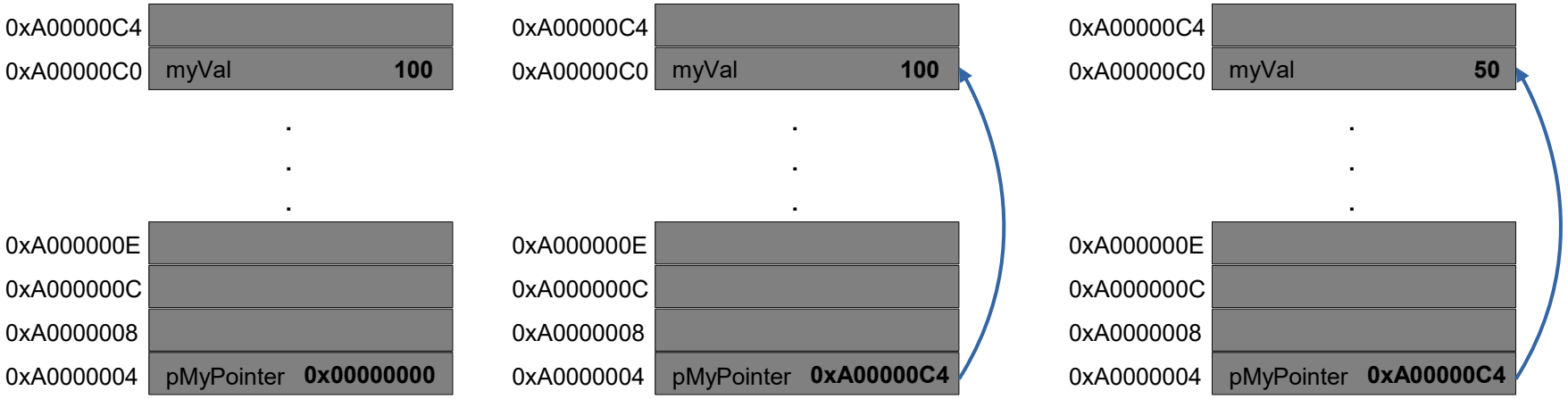
## Einfache Pointer

- Ein Zeiger (Pointer) ist eine Variable, dessen Inhalt kein Wert im eigentlichen Sinne ist, sondern eine Speicheradresse. Der Name Pointer kommt daher, dass diese Variable „auf die Speicheradresse zeigt“, die in ihr gespeichert ist

```
int* pMyPointer = 0;
int myVal = 100;
```

```
pMyPointer = &myVal;
```

```
*pMyPointer = 50;
```




Quelle: intern

# Pointer

## Einfache Pointer

- Auf einem 32-Bit System, ist eine Pointer-Variable i.d.R. 32-Bit (4 Byte) groß. Unabhängig vom Typ des Pointers

```
int*  pMyPointer;          sizeof(pMyPointer) = 4 Byte
char* pMyCharPointer;      sizeof(pMyCharPointer) = 4 Byte
```

Quelle: intern

# Pointer

## Einfache Pointer

```
int globalVar = 10;

void someFunc()
{
    char* pChar = 0;
    int* pInt = 0;
    short* pShort = 0;
    long* pLong = 0;
    long long* pLongLong = 0;
    float* pFloat = 0;
    double* pDouble = 0;

    globalVar = sizeof(pChar);
    globalVar = sizeof(pInt);
    globalVar = sizeof(pShort);
    globalVar = sizeof(pLong);
    globalVar = sizeof(pLongLong);
    globalVar = sizeof(pFloat);
    globalVar = sizeof(pDouble);
}
```

```
ldr     r3, .L4
mov     r2, #4
str     r2, [r3]

ldr     r3, .L4
mov     r2, #4
str     r2, [r3]

ldr     r3, .L4
mov     r2, #4
str     r2, [r3]

ldr     r3, .L4
mov     r2, #4
str     r2, [r3]

ldr     r3, .L4
mov     r2, #4
str     r2, [r3]

ldr     r3, .L4
mov     r2, #4
str     r2, [r3]

ldr     r3, .L4
mov     r2, #4
str     r2, [r3]
```

Quelle: intern

# Pointer

## Einfache Pointer

- Wichtig ist allerdings der Typ des Pointers, sobald es um die sog. Pointer-Arithmethik geht
  - Hierbei werden Rechenoperationen mit Pointern durchgeführt, welche auf die Speicheradresse, die im Pointer gespeichert ist, wirken.
  - So verändern z.B. die Post-Inkrement/Dekrement Operationen bei einem Pointer die Adresse um +/- sizeof(Typ des Pointers)

```
int*  pMyPointer = (int*)0xA00000C4;
pMyPointer++;
```

Pointer inkrementiert um 4 Byte
- → sizeof(int) = 4 Byte
- → Neue Adresse 0xA00000C8

```
char* pMyPointer = (int*)0xA00000C4;
pMyPointer++;
```

Pointer inkrementiert um 1 Byte
- → sizeof(char) = 1 Byte
- → Neue Adresse 0xA00000C5

Quelle: intern

DHBW
Duale Hochschule
Baden-Württemberg

# Embedded C/C++

## Alignment und Padding

# Alignment und Padding

## Byte Alignment

- Heutige Microcontroller und Prozessoren haben in der Regel eine Wortbreite von 32 Bit, 64 Bit oder mehr.
- Datenverarbeitungsbefehle sind i.d.R. optimiert auf die Verarbeitung von diesen Wortbreiten
- Damit der Zugriff und die Verarbeitung dieser Wortbreiten effizient erfolgt, müssen die Daten (und auch der Code) üblicherweise auf bestimmte Adressen ausgerichtet werden → Alignment
- In der ABI steht u.a. welches Alignment für welche Daten und Befehle notwendig ist
- Üblich sind hier 4-Byte oder 8-Byte Alignment ➔ d.h. Adressen müssen durch 4 oder 8 teilbar sein

- ➔ Der Compiler und Linker achten i.d.R. auf korrektes Alignment
- ➔ Bei der Verwendung von Pointern kann hier u.U. ein Problem auftreten

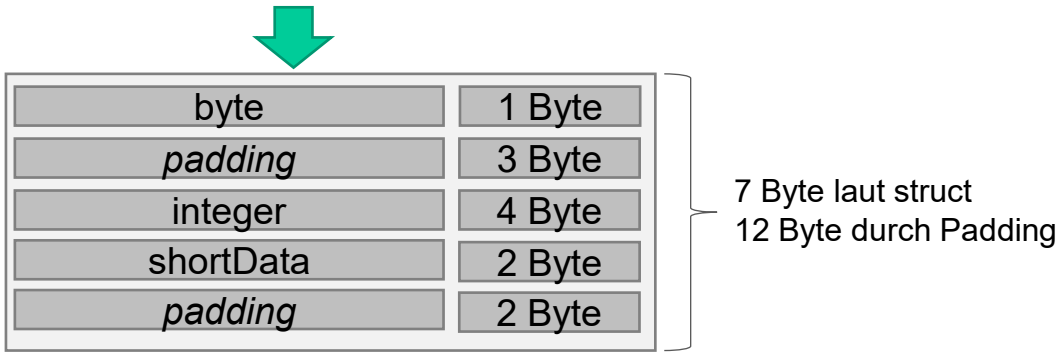
Quelle: intern

# Alignment und Padding

## Padding

- Aufgrund der Alignment Forderung kann es sein, dass der Compiler bei Datenstrukturen (struct) sog. Padding-Bytes einfügt, damit der Zugriff auf die Struktur-Elemente mit der entsprechenden Wortbreite des Prozessors stattfinden kann (performanter)

```
typedef struct _dataStruct {
    char byte;
    int integer;
    short shortData;
} dataStruct;
```




Quelle: intern

# Alignment und Padding

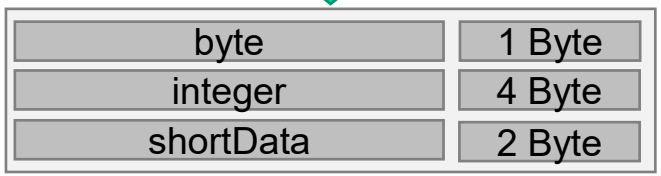
## Padding

- Aufgrund der Alignment Forderung kann es sein, dass der Compiler bei Datenstrukturen (struct) sog. Padding-Bytes einfügt, damit der Zugriff auf die Struktur-Elemente mit der entsprechenden Wortbreite des Prozessors stattfinden kann (performanter)

```
#pragma pack(push, 1)
typedef struct _dataStruct {
    char byte;
    int integer;
    short shortData;
} dataStruct;
#pragma pack(pop)
```

Aktuelle Alignment-Einstellung wird gespeichert und neues Alignment wird auf 1 Byte gesetzt

Aktuelle Alignment-Einstellung auf ursprünglichen Wert zurückgesetzt ➔ Alignment Änderung nur für diese Struktur




7 Byte laut struct
7 Byte ohne Padding

Zugriff auf Strukturelemente langsamer, da Compiler Code für einzelnen Byte-Zugriff erzeugen muss

Quelle: intern

DHBW
Duale Hochschule
Baden-Württemberg

# Embedded C/C++

## Sections und Volatile

# Sections

## Variablen in Section

- Der Compiler legt Variablen automatisch in die Sections .bss oder .data
- Es können aber auch durch Angabe spezieller Attribute Variablen in andere Sections platziert werden

```
uint32_t myGlocalVarInSection __attribute__ ((section (".heap")));
```

Quelle: intern

# Sections

## Funktionen in Section

- Der Compiler legt Funktionen automatisch in die Section .text bzw. in eine Unter-Section der .text Section (falls z.B. –ffunction-section als Parameter angegeben wird)

- Es können aber auch durch Angabe spezieller Attribute Funktion in andere Sections platziert werden

```
int functionInSection(int key) __attribute__ ((section (".auth")));
```

Quelle: intern

# Sections

## Funktionen in Section - Beispiel

### Funktion

```
int functionInSection(int key) __attribute__ ((section (".auth")));


int functionInSection(int key)
{
    return key ^ 0xABCDEF;
}
```

### Linker Section

```
/* Initialized auth section */
.auth :
{
  . = ALIGN(4);
  _sauth = .;

  *(.auth)
  *(.auth*)

  . = ALIGN(4);
  _eauth = .;
} >RAM AT> FLASH
```

Quelle: intern

# Sections

## Funktionen in Section - Beispiel

```
There are 23 section headers, starting at offset 0x47fd4:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .isr_vector       PROGBITS        08000000 001000 0001d8 00   A  0   0  1
  [ 2] .text             PROGBITS        080001d8 0011d8 008680 00  AX  0   0  8
  [ 3] .rodata           PROGBITS        08008858 009858 000108 00   A  0   0  8
  [ 4] .ARM.exidx.t[...] ARM_EXIDX       08008960 009960 000008 00  AL  2   0  4
  [ 5] .data             PROGBITS        20000000 00a000 00000c 00  WA  0   0  4
  [ 6] .auth             PROGBITS        2000000c 00a00c 00001c 00  AX  0   0  4
  [ 7] .bss              NOBITS          20000028 00a028 000268 00  WA  0   0  4
  [ 8] .heap             PROGBITS        20000290 00a028 000000 00   W  0   0  1
  [ 9] .stack            NOBITS          20000290 00a290 01fd70 00  WA  0   0  1
```

```
.auth           0x2000000c      0x1c load address 0x08008974
                0x2000000c                      . = ALIGN (0x4)
                0x20000028                      _sauth = .
 *(.auth)
 .auth          0x2000000c      0x1c obj/main.o
                0x2000000c                functionInSection
 *(.auth*)
                0x20000028                      . = ALIGN (0x4)
                0x20000028                      _sauth = .
```

Quelle: intern

# Volatile

## Volatile Variablen

- Bei der Verwendung von z.B. globalen Variablen innerhalb einer Funktion, kann der Compiler das Auslesen der Speicherstelle der Variable optimieren bzw. den gelesenen Wert in einem Register puffern (Zugriff deutlich schneller)
- Das hat zur Folge, dass der Wert der globalen Variable sich ändern kann (z.B. durch eine ISR), diese Änderung aber nicht in der Funktion erkannt wird, da der Wert nicht mehr aus dem Speicher sondern aus dem Register verwendet wird
- In einem solchen Fall muss die Variable mit dem „Modifier“ volatile gekennzeichnet werden.

- Einsatzgebiete von volatile Variablen
  - Variablen/Speicherstellen werden von der Hardware geändert
  - Multi-Threading Applikationen

```
volatile int32_t gInterruptFlag = 0;
```

Quelle: intern

# Volatile

## Volatile Variablen

```
int globalVar = 0;

int someFunction(int a)
{
    if (globalVar > 10)
    {
        return a * 10;
    }
    else
    {
        int localVar = globalVar;
        localVar++;

        return a * localVar;
    }
}
```

```
volatile int globalVar = 0;

int main()
{
    globalVar = 5;

    int res = someFunction(10);
}
```

Quelle: intern