

## Kurseinheit 7: Zeiger 2

1. Zeigerarithmetik einfacher Datentypen
2. Zeigerarithmetik komplexer Datentypen
3. Generischer Zeiger, Casts und restrict
4. Dynamische Speicherverwaltung

## Übersicht KE 7

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: Bubblesort mit Zeigern

## 1. Zeigerarithmetik einfacher Datentypen

3

### Übersicht: Operationen mit Zeigern (1)

Folgende Operationen mit Zeigern sind **sinnvoll**:

**Zuweisung** Zuweisung mit NULL (entspricht dem Wert 0x00000000)

Wie kann anhand des Wertes eines Zeigers erkannt werden, ob dieser gültig ist (zeigt auf eine ihm zugewiesene Adresse einer existierenden Variablen)? Dies ist eigentlich nicht möglich! Hier wird ein Trick verwendet: An der Speicherstelle 0x00000000 kann sich keine Variable befinden. Alle noch nicht initialisierten Zeiger lässt man darauf zeigen. Somit kann immer abgefragt werden, ob der Zeiger gültig ist oder nicht. Siehe C-Coding Styleguide DV14!

```
int* pi1Val1 = NULL;
```

```
if (pi1Val1 == NULL)
{
    pi1Val1 = &iVal1;
}
```

Zuweisung einer Adresse

Zwei Möglichkeiten: Adresse einer Variablen oder anderer Zeiger.

```
pi1Val1 = &iVal1;
```

```
pi2Val1 = pi1Val1;
```

Der Zeigertyp (bzw. &Variable) muss dabei vom gleichen Typ sein. Ausnahme void\*-Zeiger – siehe später.

## 1. Zeigerarithmetik einfacher Datentypen

4

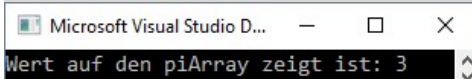
### Übersicht: Operationen mit Zeigern (2)

**Inkrementieren**  
**Dekrementieren**

Zeiger können inkrementiert und dekrementiert werden. Der Wert des Zeigers („Adresse auf die der Zeiger zeigt“) wird dabei automatisch um die Anzahl der Bytes vergrößert oder verringert, wie ein Element diesen Datentyps im Speicher benötigt.

```
int aiArray[5] = { 0, 1, 2, 3, 4 };
int* pi1Array = &aiArray[2];
```

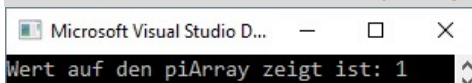
```
pi1Array++;
printf("Wert auf den pi1Array zeigt ist: %d\n", *pi1Array);
```



**Addieren**  
**Subtrahieren**

Auch bei der Subtraktion und der Addition gilt, dass der Wert des Zeigers automatisch um die Anzahl der Bytes vergrößert oder verringert wird.

```
pi1Array = pi1Array - 2;
printf("Wert auf den pi1Array zeigt ist: %d\n", *pi1Array);
```



## 1. Zeigerarithmetik einfacher Datentypen

5

### Übersicht: Operationen mit Zeigern (3)

#### Differenz zweier Zeiger

Das Ergebnis ist immer die Anzahl der Elemente (abhängig vom Datentyp)

```
pi1Array = &aiArray[1];
pi2Array = &aiArray[4];
aiArray[1] = 99;
aiArray[4] = 100;

iDifference = pi1Array - pi2Array;
printf("Differenz pi1Array - pi2Array ist: %d\n", iDifference);
```

#### Vergleichsoperatoren

Die Vergleichsoperationen == und != werden angewendet, um z.B. auf ungleich NULL zu überprüfen oder um zu überprüfen, ob zwei Zeiger auf die gleiche Variable zeigen.

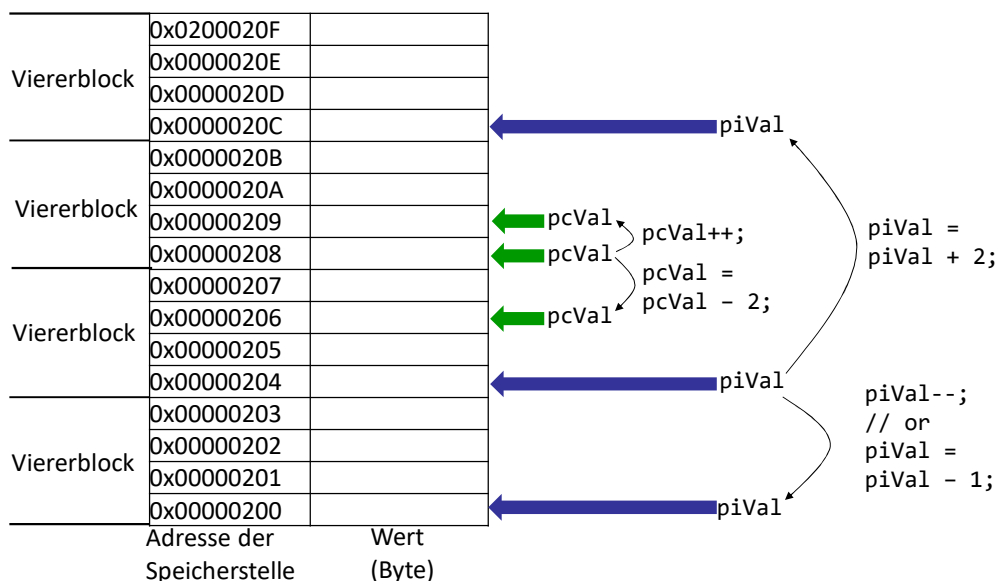
```
if (pi1Array != NULL)
{
    // Do something
}
```

```
if (pi1Array == pi2Array)
{
    // Do something
}
```

## 1. Zeigerarithmetik einfacher Datentypen

6

### Zeigerarithmetik - Speicher



## 2. Zeigerarithmetik komplexer Datentypen

7

### Zugriff auf Arrays mit Zeigern: Zwei Möglichkeiten mit Zeigern

#### Methode 1:

Zeiger wird verschoben (Inkrement).  
Wo steht der Zeiger danach?

```
pi1Array = &aiArray[0];

for (iA = 0; iA < 5; iA++)
{
    *pi1Array = 42;
    pi1Array++;
}
```

#### Schlechter Stil

```
for (iA = 0; iA < 5; iA++, pi1Array++)
```

Der Einsatz des Kommaoperators ist nicht zu empfehlen. Laut C-Coding Styleguide ist dieser nicht erlaubt (PA2).

#### Methode 2:

Zeiger wird nicht verschoben – es wird mit einem Offset gearbeitet.

```
pi1Array = &aiArray[0];

for (iA = 0; iA < 5; iA++)
{
    *(pi1Array + iA) = 42;
}
```

Diese Methode ist weniger fehleranfällig, da der Zeiger immer noch an den Anfang des Arrays zeigt.

Diese Methode ist vorzuziehen.

## 2. Zeigerarithmetik komplexer Datentypen

8

### Zugriff auf Arrays mit Zeigern: BubbleSort mit Methode 1

```
void DoBubbleSortWithPointerMethod1(int* paiA, unsigned int uiSize)
{
    unsigned uiY;
    unsigned uiX;
    int* paiDummy;
    int iDummy;

    for (uiY = 0U; uiY < (uiSize - 1U); uiY++)
    {
        paiDummy = paiA;
        for (uiX = 0U; uiX < (uiSize - 1U - uiY); uiX++)
        {
            if (*(paiDummy + 0) > *(paiDummy + 1))
            {
                iDummy = *(paiDummy + 0);
                *(paiDummy + 0) = *(paiDummy + 1);
                *(paiDummy + 1) = iDummy;
            }
            paiDummy++;
        }
    }
}
```

#### Methode 1:

(Hilfs-)Zeiger wird verschoben (Inkrement).

Es ist ein zweiter (Hilfs-) Zeiger notwendig. Ein Zeiger wird verschoben (paiDummy), ein zweiter Zeiger zeigt immer noch an den Anfang des Arrays (paiA). Dieser ist notwendig, da der (Hilfs-)Zeiger vor der inneren Schleife wieder an den Anfang des Arrays zeigen muss.

## 2. Zeigerarithmetik komplexer Datentypen

9

### Zugriff auf Arrays mit Zeigern: BubbleSort mit Methode 2

```
void DoBubbleSortWithPointerMethod2(int* paiA, unsigned int uiSize)
{
    unsigned uiY;
    unsigned uiX;
    int iDummy;

    for (uiY = 0U; uiY < (uiSize - 1U); uiY++)
    {
        for (uiX = 0U; uiX < (uiSize - 1U - uiY); uiX++)
        {
            if (*(paiA + uiX + 0) > *(paiA + uiX + 1))
            {
                iDummy = *(paiA + uiX + 0);
                *(paiA + uiX + 0) = *(paiA + uiX + 1);
                *(paiA + uiX + 1) = iDummy;
            }
        }
    }
}
```

Methode 2:  
Zeiger wird nicht verschoben.  
Es wird mit einem Offset  
gearbeitet.

## 2. Zeigerarithmetik komplexer Datentypen

10

### Äquivalenz Zeiger-Array

Zwischen Zeiger und Array gibt es eine gewisse Äquivalenz. Arrays werden in C intern als Adressen betrachtet.

```
char acTown[] = "Ofgenburg";
```

Zugriff direkt über Array

Zugriff über (Hilfs-)Zeiger

```
acTown[2] = 'f';
```

Alternative  
↕

```
*(acTown + 2) = 'f';
```

Äquivalenz

```
*(pacTown + 2) = 'f';
```

Alternative  
↕

```
pacTown[2] = 'f';
```

**Unterschiede:** Arrays belegen mit Programmstart einen Speicherbereich und dürfen nicht ähnlich wie Zeiger inkrementiert/dekrementiert werden (~~acTown++~~). Zeigern muss ein Speicherbereich zugewiesen werden, Zeigerarithmetik ist erlaubt.

## 2. Zeigerarithmetik komplexer Datentypen

11

### Nutzung Äquivalenz mit Zeigern: BubbleSort

```
void DoBubbleSortWithEquivalencePointerArray(int* paiA,
                                             unsigned int uiSize)
{
    unsigned uiY;
    unsigned uiX;
    int iDummy;

    for (uiY = 0U; uiY < (uiSize - 1U); uiY++)
    {
        for (uiX = 0U; uiX < (uiSize - 1U - uiY); uiX++)
        {
            if (paiA[uiX] > paiA[uiX + 1])
            {
                iDummy = paiA[uiX];
                paiA[uiX] = paiA[uiX + 1];
                paiA[uiX + 1] = iDummy;
            }
        }
    }
}
```

Dieses Notation (Array) ist deutlich übersichtlicher

## 2. Zeigerarithmetik komplexer Datentypen

12

### Struct im Speicher

```
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;
```

Oft wird die obige kompaktere Form eines typedefs angewendet. Bisher wurde diese Variante verwendet:

```
struct Address
{
    //..
};
typedef struct Address sAddress_t;
typedef sAddress_t* psAddress_t;
```

Hohe Adresse

Das erste Element im Struct liegt im Speicher bei der niedrigen Adresse.

Niedrige Adresse

### Speicher

acTown (40 Bytes)
uiZipCode (4 Bytes)
acStreet (40 Bytes)
acFirstName (30 Bytes)
acLastName (30 Bytes)

## 2. Zeigerarithmetik komplexer Datentypen

13

### Zeigerarithmetik mit Strukturen (1)

Zeigerarithmetik kann auch auf Strukturzeiger angewendet werden.

```
sAddress_t asAllAddress[4] =  
    {{ "Ell", "Peter", "Im Walde 4", 77704U, "Oberkirch"},  
      { "Maier", "Frida", "Urweg 3", 77652U, "Offenburg"},  
      { "Huber", "Siggi", "Irrweg 42", 77933U, "Offenburg"},  
      { "Auber", "Willi", "Am Rhein 1", 77694U, "Kehl"} };  
psAddress_t psAddress1 = &asAllAddress[2];  
psAddress_t psAddress2 = &asAllAddress[0];
```

```
psAddress1--;  
printf("acLastName of psAddress1 ist: %s\n", psAddress1->acLastName);  
psAddress2 = psAddress2 + 3;  
printf("uiZipCode of psAddress2 ist: %u\n", psAddress2->uiZipCode);  
psAddress2++;  
printf("uiZipCode of psAddress2 ist: %u\n", psAddress2->uiZipCode);
```

```
Microsoft Visual Studio Debug ...  
acLastName of psAddress1 ist: Maier  
uiZipCode of psAddress2 ist: 77694  
uiZipCode of psAddress2 ist: 3435973836
```

Was ist hier passiert?

## 2. Zeigerarithmetik komplexer Datentypen

14

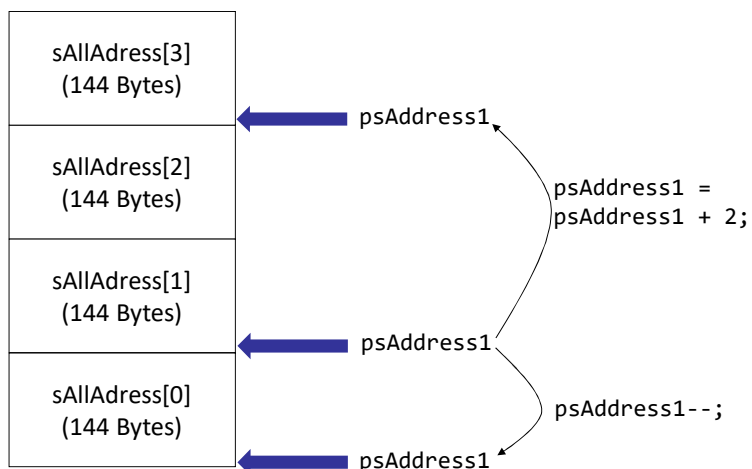
### Zeigerarithmetik mit Strukturen (2)

```
sAddress_t asAllAddress[4];  
psAddress_t psAddress1 = &asAllAddress[1];
```

Hohe Adresse

Das erste (Struct-) Element des Arrays liegt im Speicher bei der niedrigen Adresse.

Niedrige Adresse



## 2. Zeigerarithmetik komplexer Datentypen

15

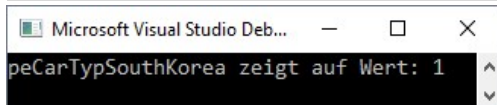
### Zeigerarithmetik mit Enumeration, Unions und Bitfeldern

Eine **Enumeration** (Aufzählungstyp) hat im Hintergrund einen Integer-Typ. Entsprechend kann hier auch die gleiche Zeigerarithmetik angewendet werden.

```
enum CarTypesSouthKorea {Hyundai = 0, Kia = 1};
```

```
enum CarTypesSouthKorea eCarTypSouthKorea = Hyundai;  
enum CarTypesSouthKorea* peCarTypSouthKorea = &eCarTypSouthKorea;
```

```
*peCarTypSouthKorea = Kia;  
printf("peCarTypSouthKorea zeigt auf Wert: %i\n", *peCarTypSouthKorea);
```



Entsprechend können dann auch Arrays von Enumerationen definiert und Zeigerarithmetik implementiert werden.

Zeigerarithmetik bei **Bitfeldern** und **Unions** ist vergleichbar wie bei Strukturen. Bitfelder und Unions werden in Embedded Systems nochmals vertiefend behandelt.

## 3. Generischer Zeiger, Casts und restrict

16

### Generischer Zeiger

Ein generischer Zeiger ist ein datentyploser Zeiger. Ihm ist kein spezieller Datentyp zugeordnet. Die Definition (und Initialisierung mit NULL) geschieht dabei wie folgt:

```
void* pvGeneric = NULL;
```

Nach dem C-Coding Styleguide ist mit „pv“ (pointer auf void) als Präfix beim Variablenamen ein generischer Zeiger zu definieren.

```
char cVal = 'H';  
char* pc = &cVal;  
unsigned int uiVal1 = 0xFADEC0DE;  
unsigned int* pui = &uiVal1;  
void* pvGeneric = NULL;  
  
pvGeneric = pc;  
printf("%c\n", *pc);  
//printf("%c\n", *pvGeneric); //Error  
  
pvGeneric = &uiVal1;  
//uiVal2 = 2U + *pvGeneric; //Error
```

Generische Zeiger können auf beliebige Variablen zeigen. Der Zugriff auf den Inhalt mit dem Deferenzierungsoperator `*` scheitert aber. Der Compiler muss wissen, welcher Datentyp sich dahinter verbirgt.

Abhilfe: Der generische Zeiger muss vorher gecastet werden.



### 3. Generischer Zeiger, Casts und restrict

17

#### Generischer Zeiger – Cast bei \*

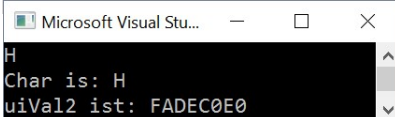
```
char cVal = 'H';
char* pc = &cVal;
unsigned int uiVal1 = 0xFADEC0DE;
unsigned int* pui = &uiVal1;
unsigned int uiVal2;
void* pvGeneric = NULL;
```

Statt eines Casts könnte ein Hilfszeiger vom gewünschten Datentyp definiert werden und diesem den generischen Zeiger zuweisen.

```
pvGeneric = pc;
printf("%c\n", *pc);
//printf("%c\n", *pvGeneric); //Error
printf("Char is: %c\n", *((char*)pvGeneric));

pvGeneric = &uiVal1;
//uiVal2 = 2U + *pvGeneric; //Error
uiVal2 = 2U + *((unsigned int*)pvGeneric);
printf("uiVal2 ist: %X\n", uiVal2);
```

Cast des generischen Zeigers vor der Dereferenzierung.



### 3. Generischer Zeiger, Casts und restrict

18

#### Unterschied NULL und 0

Zwischen NULL und 0 gibt es einen Unterschied in C. Bei 0 handelt es sich um den numerischen Wert 0, bei NULL handelt es sich um einen generischen Zeiger, der auf die Adresse 0x00000000 zeigt. NULL ist in <vcruntime.h> deklariert (diese Headerdatei wird immer automatisch eingebunden). Details zu #define später in dieser KE.

```
#define NULL ((void*)0)
```

In C++ ist NULL als numerischer Wert 0 deklariert. Dort gibt es aber dafür einen nullptr.

C++ baut auf C auf (prozedurale Erweiterungen + Objektorientierung + standardisierte Bibliotheken). Mit ++ ist der Postincrement-Operator gemeint: C wird inkrementiert!

C++ und C sind aber nicht immer zu 100% kompatibel. Werden C-Programme mit einem C++ Compiler übersetzt, so kann es an wenigen Stellen zu Fehlern oder unvorhersehbarem Verhalten führen.

EI, EI+, MKA, EI3nat und AI haben noch C++ in einer Lehrveranstaltung. Dort werden die Kenntnisse aus dieser Lehrveranstaltung vorausgesetzt.

### 3. Generischer Zeiger, Casts und restrict

19

#### Cast von Zeigern

##### C Compiler

```
char cVal = 'H';
char* pc = &cVal;
unsigned int uiVal1 = 0xFADEC0DE;
unsigned int* pui = &uiVal1;
sAddress_t sAddress;
psAddress_t psAddress = &sAddress;
void* pvGeneric = NULL;

// Implicit Casts
pc = psAddress; // Warning
pvGeneric = pui;
pc = pvGeneric;
pui = psAddress + 100; // Warning
```

##### C++ Compiler

```
pc = psAddress; // error
pvGeneric = pui; // no error
pc = pvGeneric; // error
pui = psAddress + 100; // error
```

In **C** können beliebige Zeiger unabhängig vom Datentyp einander zugewiesen werden. Es geschieht hier ein **impliziter** Cast!

Es gibt nur wenige praktische Fälle, bei denen eine Zuweisung zweier nicht generischer Zeiger sinnvoll sein könnte. Ein Zuweisung eines konkreten Zeigertyps mit einem generischen Zeiger ist häufiger anzutreffen.

In **C++** findet kein impliziter Cast statt und der Compiler generiert einen Fehler (außer Zeile 2 links unten). In C++ ist daher ein **expliziter** Cast notwendig. Um die **Kompatibilität** C zu C++ zu gewährleisten, muss auch in C ein expliziter Cast verwendet werden.

### 3. Generischer Zeiger, Casts und restrict

20

#### restrict Zeiger

In C99 gibt es das Schlüsselwort **restrict** für Zeiger. Wird bei der Definition eines Zeigers dieses Keyword verwendet, so teilt man dem Compiler mit, dass nur über diesen Zeiger (welcher auch „verschoben“ werden kann, z.B. pc++) auf die Variable zugegriffen wird. Dadurch kann der Compiler den Maschinencode besser optimieren.

```
char cDataObject = 'X';
char* restrict pc = &cDataObject;
```

```
char *strcpy(char* restrict pcDest, const char* restrict pcSource);
```

Neue C99 Deklaration von strcpy

Scheinbar wird das Schlüsselwort **restrict** im **MS C-Compiler** immer noch **nicht unterstützt**.  
<https://stackoverflow.com/questions/48615184/does-visual-studio-2017-fully-support-c99>

Microsoft scheint den C-Compiler kaum noch weiterzuentwickeln. C wird hauptsächlich im Bereich von Embedded Systems und Linux eingesetzt. Der C-Compiler von GCC (Gnu Compiler Collection) unterstützt das Keyword **restrict** – ebenso viele Embedded C-Compiler.

## 4. Dynamische Speicherverwaltung

21

### Wieso dynamische Speicherverwaltung

```
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;

sAddress_t asAllCustomerAddress[?];
```

Es soll eine Software für Firmen entwickelt werden, die ihre Kundenadressen speichern wollen. Wie viele Variablen benötigen Sie (Wie viele Elemente hat das Array)?

Bauträger: 10?

Autohändler: 100?

Media-Markt: 10.000?

In einem solchen Fall sollte sich das Programm variabler verhalten und zur Laufzeit den nötigen Speicher anfordern. Dies wird ist durch die **dynamische Speicherverwaltung** möglich, welche den Speicher aus dem **Heap (HS: Heap Segment)** anfordert.

## 4. Dynamische Speicherverwaltung

22

### Grundstruktur einfacher Datentyp

```
int* piMemory = NULL; // CSG: DV14

// CSG: DV15
piMemory = (int*)malloc(100 * sizeof(int));

if (piMemory != NULL)
{
    // Dynamic memory allocation successful

    // Do something with allocated memory

    // If allocated memory not needed anymore
    free(piMemory);
    piMemory = NULL; // CSG: DV16
}
```

Mit malloc (**m**emory **a**llocation) wird Speicher auf dem **Heap (HS)** angefordert. Scheitert dies, wird ein NULL-Zeiger zurückgegeben.

Nur wenn der Zeiger ungleich NULL ist, darf mit dieser dann verwendet werden (Dereferenzierungsoperator).

Nach free ist der Zeiger nicht mehr gültig, hat aber noch einen Wert ungleich NULL. Daher muss dieser besser noch auf NULL gesetzt werden.

**Memory Leak**



Wird der Heap-Speicher nicht mehr benötigt, ist dieser frei zu geben. Ansonsten entstehen sogenannte **Memory Leaks**.

## 4. Dynamische Speicherverwaltung

23

### Grundstruktur komplexer Datentyp

```
psAddress_t psAddress = NULL; // CSG: DV14
```

```
// CSG: DV15
```

```
psAddress = (psAddress_t)malloc(100 * sizeof(sAddress_t));
```

```
if (psAddress != NULL)
```

```
{
```

```
    // Dynamic memory allocation successful
```

```
    // Do something with allocated memory
```

```
    // If allocated memory not needed anymore
```

```
    free(psAddress);
```

```
    psAddress = NULL; // CSG: DV16
```

```
}
```

```
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;
```

Bei komplexen Datentyp sollte mit den typedefs gearbeitet werden.

## 4. Dynamische Speicherverwaltung

24

### Wie können Memory Leaks gefunden werden?



WH: Wird der Heap-Speicher nicht mehr benötigt, ist dieser frei zu geben. Ansonsten entstehen sogenannte **Memory Leaks**.

In umfangreicheren Anwendungen ist es recht schwer, dies konsequent umzusetzen: Allokierung und Freigabe (free) in unterschiedlichen Funktionen, richtiges Reagieren auf Fehlerfälle.

**Irgendwann** (Min, Std., Tage, Wochen) ist kein Heap mehr verfügbar!

### Lösungsmöglichkeit mit MSVS

- Target Debug in IDE anwählen
- In c-Dateien mit dynamischer Speicherverwaltung und in main oben hinzufügen:

```
#define _CRTDBG_MAP_ALLOC
```

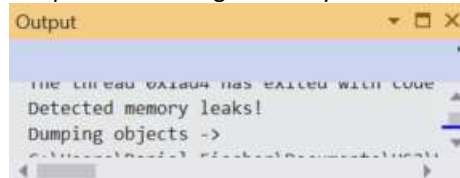
```
#include <stdlib.h>
```

```
#include <crtDBG.h>
```

- Am Ende von main hinzufügen

```
_CrtDumpMemoryLeaks();
```

- Output-Fenster zeigt Memory Leak an!



## 4. Dynamische Speicherverwaltung

25

### malloc im Detail – Dynamische Allokation

Die Funktion malloc reserviert (allokiert – Dynamische Allokation) zusammenhängenden Speicher auf dem Heap und gibt einen generischen Zeiger (typenloser Zeiger) an den Anfang des Speicherbereichs zurück.

```
piMemory = (int*)malloc(100 * sizeof(int));
```

Ein Cast wäre in C nicht notwendig, allerdings um die Kompatibilität C zu C++ zu gewährleisten, muss auch in C ein expliziter Cast verwendet werden.

malloc erwartet die Anzahl zu allozierender Bytes als Übergabeparameter. Um plattformunabhängig zu bleiben, sollte hier mit sizeof gearbeitet werden: „Allokiere 100 Integer“!

Über den gecasteten Zeiger kann dann zugegriffen werden.

```
*(piMemory + 42) = 73;  
piMemory[42] = 73;
```

Alternative  
Zugriffsmöglichkeiten

void\* →

Heap



## 4. Dynamische Speicherverwaltung

26

### Alle Funktionen aus stdlib.h

```
void* malloc(size_t uiSize);
```

```
piMemory = (int*)malloc(100 * sizeof(int));
```

Dynamische Allokation von Speicher auf dem Heap. Nur ein Übergabeparameter!

```
void* calloc(size_t uiNumber, size_t uiSizeOfElement);
```

```
piMemory = (int*)calloc(100, sizeof(int));
```

Dynamische Allokation von Speicher auf dem Heap und Initialisierung mit 0. „c“ vor „alloc“ könnte für „Clear“ stehen (Initialisierung mit 0). Der allokierte Heapspeicher enthält sonst Zufallswerte. Zwei Übergabeparameter!

```
void* realloc(void* pMemory, size_t uiNewSize);
```

```
piMemory = (int*)realloc(piMemory, 500 * sizeof(int));
```

Ein zuvor mit malloc, calloc oder realloc dynamisch allokiertes Speicherbereich kann vergrößert oder verkleinert werden

```
void free(void* pv);
```

```
free(piMemory);
```

Ein zuvor mit malloc, calloc oder realloc dynamisch allokiertes Speicherbereich wird freigegeben. Pointer hat danach **nicht** den Wert NULL.

## 4. Dynamische Speicherverwaltung

27

### Interner Aufbau

```
piMemory = (int*)calloc(10, sizeof(int));
```

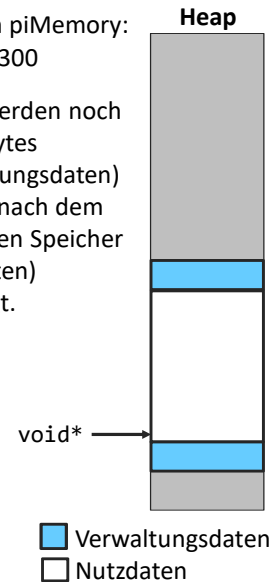
Wert von piMemory:  
0x00D31300

Memory 1

Address: 0x00D312E8

0x00D312E8	00 00 00 00 00 00 00 00	.....
0x00D312F0	01 00 00 00 28 00 00 00	....( ...
0x00D312F8	53 00 00 00 fd fd fd fd	S...ýýýý
0x00D31300	00 00 00 00 00 00 00 00	.....
0x00D31308	00 00 00 00 00 00 00 00	.....
0x00D31310	00 00 00 00 00 00 00 00	.....
0x00D31318	00 00 00 00 00 00 00 00	.....
0x00D31320	00 00 00 00 00 00 00 00	.....
0x00D31328	fd fd fd fd e0 5b d3 00	ýýýýà[ó.
0x00D31330	3c 8d 15 48 09 8c 00 00	<..H.Æ..
0x00D31338	a0 b9 d2 00 98 b1 d2 00	.ò.~±ò.
0x00D31340	58 52 d2 00 00 00 00 00	XRò....
0x00D31348	04 8d 14 71 01 8c 00 0c	...q.Æ..
0x00D31350	00 00 00 00 00 00 00 00	.....

Intern werden noch einige Bytes (Verwaltungsdaten) vor und nach dem allokierten Speicher (Nutzdaten) eingefügt.



## 4. Dynamische Speicherverwaltung

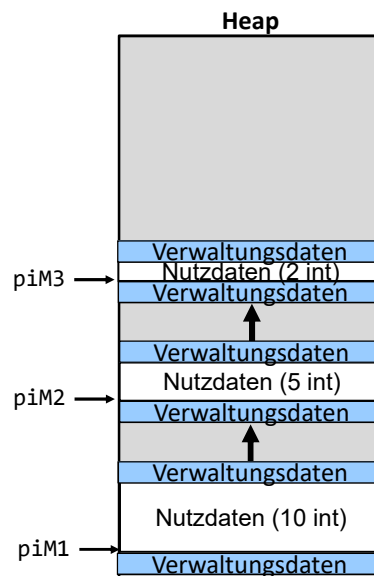
28

### Mehrmalige dynamische Speicherallokation

```
int* piM1 = NULL;  
int* piM2 = NULL;  
int* piM3 = NULL;
```

```
piM1 = (int*)malloc(10 * sizeof(int));  
piM2 = (int*)malloc(5 * sizeof(int));  
piM3 = (int*)malloc(2 * sizeof(int));
```

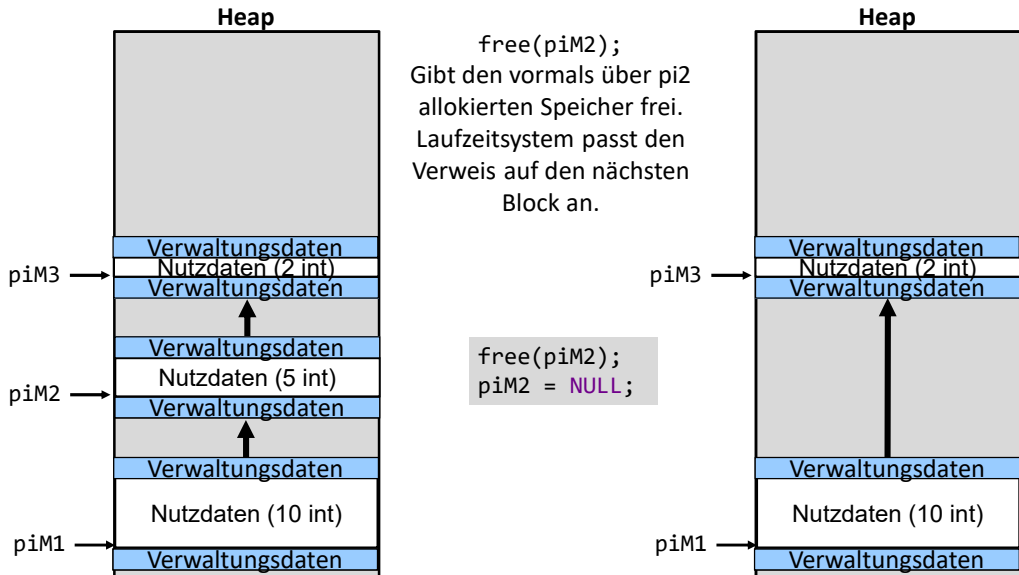
Intern werden über das Laufzeitsystem (stdlib) die allokierten Blöcke miteinander verkettet (Verkettete Liste, siehe später). Ebenso sind noch weitere Informationen wie z.B. die Größe der Nutzdaten oder ein Zeiger auf den nächsten allokierten Speicherblock in den Verwaltungsdaten enthalten.



## 4. Dynamische Speicherverwaltung

29

### Mehrmalige dynamische Speicherallokation- Speicherfreigabe



## 4. Dynamische Speicherverwaltung

30

### Fehlerhandling

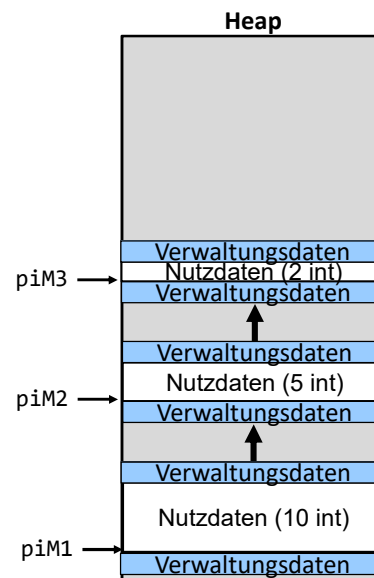
malloc, calloc und realloc versuchen **zusammenhängenden** Speicher auf dem Heap zu allokiieren. Gelingt dies nicht, wird eine NULL zurückgegeben. Ursachen:

- Es ist nicht genügend Speicher vorhanden
- Heap ist fragmentiert (Speicher da, aber nicht mehr zusammenhängend)

Daher ist immer nach der Ausführung der Funktionen der Rückgabewert zu überprüfen.

```
piM1 = (int*)malloc(10 * sizeof(int));
if (piM1 != NULL)
{
    // Do something with piM1
}
```

Default Heap Size im virtuellen Speicher (MSVS): 1 MB



## 4. Dynamische Speicherverwaltung

31

### calloc und realloc im Detail

Die Funktion **calloc** entspricht weitestgehend der Funktion **malloc**. Folgende Unterschiede gibt es:

- Die Größe der benötigten Nutzdaten ist nicht mehr zu berechnen (z.B.  $100 * \text{sizeof}(\text{int})$ ), stattdessen werden nur die Anzahl der Elemente (z.B. 100) und die Größe eines Elements (z.B.  $\text{sizeof}(\text{int})$ ) mitgegeben).
- Die Nutzdaten sind mit 0 initialisiert (c: Clear)

Die Funktion **realloc** versucht die Größe der Nutzdaten zu verändern und passt dabei auch die Verwaltungsdaten an.

- Verkleinern der Daten: trivial
- Vergrößern der Daten: trivial / nicht trivial

**Fall 1:** Es ist genügend Speicher zum nächsten Block vorhanden (trivial). Block kann vergrößert werden.

**Fall 2:** Es ist nicht genügend Speicher zum nächsten Block vorhanden. Es muss daher ein neuer Block allokiert werden und die bereits vorhandenen Nutzdaten werden kopiert (nicht trivial).

## 4. Dynamische Speicherverwaltung

32

### Vergleich: Dynamische und statische Speicherallokation

	Statische Speicherallokation	Dynamische Speicherallokation
Beispiele	<code>int iArray[10];</code> <code>struct C sC1;</code> <code>struct C asC[100];</code>	<code>int* pi = (int*)</code> <code>malloc(100 * sizeof(int));</code>
Zeitpunkt der Speicherallokation	Programmierung	Laufzeit
Funktionen	- Keine – nur Definition von Variablen und Arrays	malloc, calloc, realloc und free
Speicherbereich	Stack (SS) Globale Daten (DS) Code (CS)	Heap (HS)



## Behandelte Schlüsselwörter in KE 7

Schlüsselwörter C89:

<del>auto</del> ✓	do ✓	<del>goto</del> ✓	signed ✓	unsigned ✓
break ✓	double ✓	if ✓	sizeof ✓	void ✓✓
case ✓	else ✓	int ✓	static ✓	volatile ✓
char ✓	enum ✓	long ✓	struct ✓	while ✓
const ✓	extern ✓	register ✓	switch ✓	
<del>continue</del> ✓	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			