

Programmieren 1 – Informatik

Pointer

Prof. Dr.-Ing. Maike Stern | 15.11.2023

- Arrays
- Strings
- Mathematische Funktionen

- Pointer
- Pointer & Arrays
- Strings
- Speicherkonzepte in C
- Stack- und Heapspeicher
- Heapspeicher allokieren und freigeben
- Pointer auf Pointer, Funktionspointer

Pointer (Zeiger) sind Variablen, die statt eines Werts (z.B. 1, 3.24, 'c') eine Adresse (z.B. 0x16ee1ae80) enthalten

- Ein Pointer hat also zwei Adressen:
 - Die eigene Speicheradresse (also die Adresse, an der der Pointer im Speicher abgelegt ist)
 - Und die Adresse, die im Speicherplatz des Pointers abgelegt ist (bei einer Integervariable liegt hier der Zahlenwert)
- Pointer können verwendet werden, um auf andere Variablen bzw. deren Speicheradresse zu zeigen. Warum das sinnvoll ist, zeigen die nächsten Folien

| Adresse | Typ | Name | Wert |
|-------------|-------|------|-------------|
| 0x16ee1ae80 | int | i | 1337 |
| 0x16ee1ae7c | float | f | 2.178 |
| 0x16ee1ae78 | int * | p | 0x16ee1ae80 |

Pointer (Zeiger) sind Variablen, die statt eines Werts (z.B. 1, 3.24, 'c') eine Adresse (z.B. 0x16ee1ae80) enthalten

- **Deklaration:** Datentyp_auf_den_der_Pointer_zeigt *Name // int *p

Der Stern vor dem Variablennamen
(der Indirektionsoperator)
kennzeichnet den Datentyp als Pointer

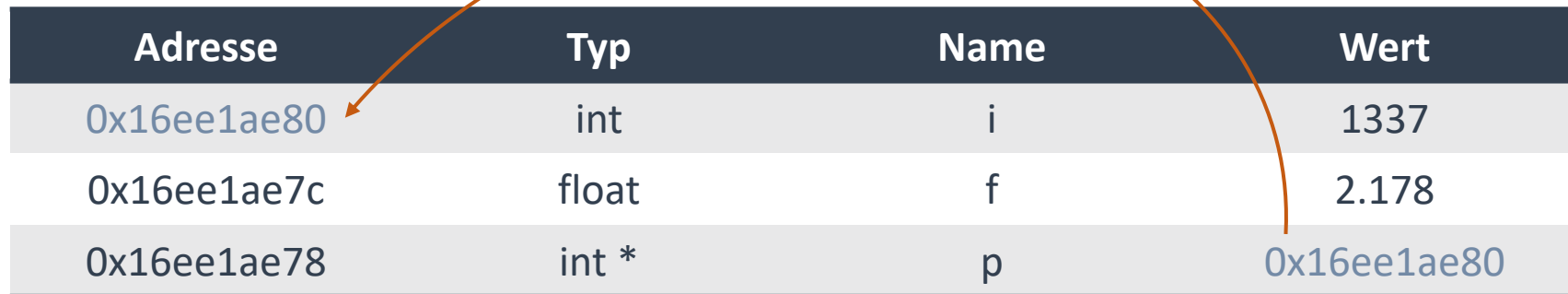
| Adresse | Typ | Name | Wert / Adresse |
|-------------|-------|------|----------------|
| 0x16ee1ae80 | int | i | 1337 |
| 0x16ee1ae7c | float | f | 2.178 |
| 0x16ee1ae78 | int * | p | --- |

Der Pointer *p* soll auf die Adresse der Integervariablen *i* zeigen (referenzieren),
daher wird *p* bei der Deklaration *int *p* als Integer-Pointer deklariert

Pointer (Zeiger) sind Variablen, die statt eines Werts (z.B. 1, 3.24, 'c') eine Adresse (z.B. 0x16ee1ae80) enthalten

- **Initialisierung:** Pointername = &Variablenname // $p = \&i$

Der Adressoperator & übergibt die
Adresse der Variablen an den Pointer



| Adresse | Typ | Name | Wert |
|-------------|-------|------|-------------|
| 0x16ee1ae80 | int | i | 1337 |
| 0x16ee1ae7c | float | f | 2.178 |
| 0x16ee1ae78 | int * | p | 0x16ee1ae80 |

Der Pointer p zeigt (referenziert) nach der
Initialisierung mit $p = \&i$ auf die Adresse
der Variablen i

Pointer (Zeiger) sind Variablen, die statt eines Werts (z.B. 1, 3.24, 'c') eine Adresse (z.B. 0x16ee1ae80) enthalten

- **Verwendung:** Da der Pointer p auf die Adresse der Variablen i zeigt (referenziert), können wir über den Pointer p auch den Wert der Variablen i abfragen. Hierzu wird ebenfalls der Stern (Indirektionsoperator) verwendet, der den Wert der Adresse dereferenziert:

```
int wertVonI = *p;
```

Beim Dereferenzieren wird der Wert der Variablen abgefragt, auf die der Pointer zeigt

| Adresse | Typ | Name | Wert |
|-------------|-------|------|-------------|
| 0x16ee1ae80 | int | i | 1337 |
| 0x16ee1ae7c | float | f | 2.178 |
| 0x16ee1ae78 | int * | p | 0x16ee1ae80 |

Pointer (Zeiger) sind Variablen, die statt eines Werts (z.B. 1, 3.24, 'c') eine Adresse (z.B. 0x16ee1ae80) enthalten

- **Verwendung:** Genauso ist es möglich, über den Pointer p der Variablen i einen neuen Wert zuzuweisen.

`*p = 10; // Die Variable i hat jetzt den Wert 10`

Über den Pointer kann auch der Wert der referenzierten Variable geändert werden

| Adresse | Typ | Name | Wert |
|-------------|-------|------|-------------|
| 0x16ee1ae80 | int | i | 10 |
| 0x16ee1ae7c | float | f | 2.178 |
| 0x16ee1ae78 | int * | p | 0x16ee1ae80 |

Was sind Pointer?

Die Syntax zusammengefasst

| Operation | Bedeutung | Beispiel | Bedeutung |
|--|------------------|---|--|
| Datentyp *Pointername | Deklaration | <code>float *floatPointer;</code> | Deklaration eines Pointers, der auf eine float-Variable zeigt. <i>*Pointername</i> steht hier dafür, dass ein Pointer vom angegebenen Datentyp angelegt wird. |
| <code>Pointername = &Variable</code> | Initialisierung | <code>floatPointer = &var;</code> | Weist dem float-Pointer die Adresse der Variablen <i>var</i> zu |
| <code>Variable = *Pointername</code> | Dereferenzierung | <code>newVar = *floatPointer;</code> | Der Wert der Variablen <i>var</i> , auf die der Pointer weist, wird der Variablen <i>newVar</i> zugewiesen. <i>*Pointername</i> steht hier und in der unteren Spalte quasi stellvertretend für die referenzierte Variable. |
| <code>*Pointername = Wert</code> | Dereferenzierung | <code>*floatPointer = 10.8;</code> | Der Variablen <i>var</i> , auf die der Pointer weist, wird der Wert 10.8 zugewiesen |
| <code>Pointername</code> | | <code>printf("%p", floatPointer);</code> | Gibt mit dem Formelzeichen %p die Adresse aus, auf die der Pointer zeigt |
| <code>&Pointername</code> | | <code>printf("%p", &floatPointer);</code> | Gibt mit dem Formelzeichen %p die Adresse aus, an der der Pointer selbst liegt |

Übung:

Skizzieren Sie ein kurzes Programm, mit einer Variablen und einem Pointer.

- Weisen Sie dem Pointer die Adresse der Variablen zu
- Wie können Sie die Adresse der Variablen ausgeben?
- Wie können Sie die Adresse des Pointers selbst sowie die Adresse auf die der Pointer zeigt ausgeben?
- Wie können Sie den Wert der Variablen einmal über die Variable selbst sowie einmal über den Pointer ausgeben?
- Weisen Sie der Variable über den Pointer einen neuen Wert zu
- Speichern Sie den Wert der Variablen in einer neuen Variable, indem Sie den Pointer dereferenzieren

| | |
|--------------------------|---|
| <code>int i;</code> | Deklaration Integer |
| <code>int *p;</code> | Deklaration Integer-Pointer |
| <code>p = &i;</code> | Initialisierung, Pointer p zeigt auf die Adresse der Integervariablen i |
| <code>*p;</code> | Dereferenzierung (Wert von i) |
| <code>p;</code> | Adresse, auf die p zeigt |
| <code>&p;</code> | Adresse von p |

Bereits bekannte Verwendung von Speicheradressen

Wo haben wir die Verwendung von Adressen schon kennengelernt? scanf-Funktion

Ein Parameter der scanf-Funktion ist die Variable, in die der eingelesene Wert gespeichert werden soll. Allerdings wird nicht der Wert der Variablen übergeben, wie bei allen anderen Funktionen die wir bisher kennengelernt haben, sondern die Adresse:

```
scanf("%d", &i);
```

Frage:
Warum arbeitet die scanf-Funktion mit der Adresse der Variablen und nicht einfach direkt mit *i*?

Was für Möglichkeiten gibt es für eine Funktion Daten in der main-Funktion nutzbar zu machen?

Geltungsbereich (scope) von Variablen:

In der 3. Vorlesung haben wir gesehen, dass Variablen nur innerhalb ihres Geltungsbereichs sichtbar sind:

- Variablen werden an eine Funktion mittels Call-by-Value übergeben, das heißt der Variablenwert wird in den Geltungsbereich der Funktion kopiert
 - Daher haben Änderungen innerhalb der Funktion keine Auswirkungen auf die Variablenwerte außerhalb der Funktion
 - Und: Sobald die Funktion verlassen wird, werden die Variablen im Geltungsbereich der Funktion gelöscht (bzw. der Stackframe (Speicherbereich) auf dem sie gespeichert wurden)
- Würde innerhalb der scanf-Funktion ein Wert von der Konsole eingelesen und in eine mit Call-by-Value übergebene Variable gespeichert, so wäre dieser Wert nach Beendigung der scanf-Funktion nicht mehr verfügbar

Wo haben wir die Verwendung von Adressen schon kennengelernt? scanf-Funktion

Eine Möglichkeit veränderte Parameterwerte an die main-Funktion weiterzureichen ist als Funktionsrückgabe (return)

- Die scanf-Funktion gibt jedoch die Anzahl der eingelesenen Werte als Rückgabewert zurück, nicht die eingelesenen Werte selbst
- Das würde auch bei mehr als einem eingelesenen Wert nicht funktionieren, da in C jeweils nur ein Wert zurückgegeben werden kann

```
#include <stdio.h>

int scanf(par1, int *a){
    *a = eingelesener_Wert;
    return Anzahl_eingelesener_Werte;
}

int main(){
    int i = 0, var = 0;
    var = scanf("%d", &i);

    return 0;
}
```

Wo haben wir die Verwendung von Adressen schon kennengelernt? scanf-Funktion

Deshalb nutzt die scanf-Funktion die Speicheradresse einer bereits existierenden Variable:

```
#include <stdio.h>

int scanf(parameter1, int *a){
    *a = eingelesener Wert;
    return Anzahl eingelesener Werte;
}

int main(){
    int i = 0, var = 0;
    var = scanf("%d", &i);

    return 0;
}
```

"Stackframe"

| main() | | | scanf() | | |
|--------|---------|------|----------|---------|--------|
| Name | Adresse | Wert | Name | Adresse | Wert |
| int i | 0x1020 | 0 | int *a | 0x256e | 0x1020 |
| | | | *a = 10; | | |
| int i | 0x1020 | 10 | int *a | 0x256e | 0x1020 |

Ändert den Wert von *i* im Stackframe
der main-Funktion mittels
Dereferenzierung des Pointers *a*

Wo haben wir die Verwendung von Adressen schon kennengelernt? scanf-Funktion

Deshalb nutzt die scanf-Funktion die Speicheradresse einer bereits existierenden Variable:

```
#include <stdio.h>

int scanf(parameter1, int *a){
    *a = eingesener Wert;
    return Anzahl eingelesener Werte;
}

int main(){
    int i = 0, var = 0;
    var = scanf("%d", &i);

    return 0;
}
```

"Stackframe"

| main() | | |
|--------|---------|------|
| Name | Adresse | Wert |
| int i | 0x1020 | 0 |
| | | |
| int i | 0x1020 | 10 |

Wird die scanf-Funktion verlassen, so wird der dazugehörige Stackframe gelöscht. Die Änderung der Variablen *i* fand jedoch im Stackframe der main-Funktion statt, weswegen die Änderung dauerhaft ist.

Wo haben wir die Verwendung von Adressen schon kennengelernt?

- Wird einer Funktion die Speicheradresse einer Variablen übergeben, so nennt man das **Call-by-Reference**
- Wird der Wert einer Variablen bei der Übergabe an die Funktion kopiert, so nennt man das **Call-by-Value**

Frage:
Wo haben wir Call-by-Reference schon
kennengelernt?

Pointer & Arrays

Wo haben wir die Verwendung von Adressen schon kennengelernt?

Arrays

Call-by-Reference haben wir schon im Zusammenhang mit Arrays kennengelernt: Wird ein Array an eine Funktion übergeben, so "zerfällt" das Array in einen Zeiger auf die Anfangsadresse des Arrays

```
int array[5] = {0, 1, 2, 3, 4};
```



Da Arrays ganz unterschiedliche Größen annehmen können, ist der Arrayname ein Zeiger auf die Anfangsadresse des Arrays

| [0] | [1] | [2] | [3] | [4] |
|--------|--------|--------|--------|--------|
| 0x1064 | 0x1068 | 0x106C | 0x1070 | 0x1074 |

Funktionsdeklaration mit Array als Parameter

```
void function(int array[], int n_Anzahl)
```

↑
Übergabe des Arrays als Zeiger
→ Die Funktion erhält die
Anfangsadresse des ersten
Elements im Array

← Optional, aber sinnvoll: Anzahl
der Elemente im Array

- Der Arrayname ist ein Zeiger auf die Anfangsadresse des Arrays (also das erste Element)

```
int array[5] = {0, 1, 2, 3, 4};
```

| [0] | [1] | [2] | [3] | [4] |
|--------|--------|--------|--------|--------|
| 0x1064 | 0x1068 | 0x106C | 0x1070 | 0x1074 |

- Arrays sind aber keine Zeiger!

| Adresse | Typ | Name | Wert |
|----------------|------------|---------|-----------------|
| 0x1064..0x1074 | int name[] | array | {0, 1, 2, 3, 4} |
| 0x16ee | int * | pointer | 0x16e8 |

Die Werte eines Arrays
sind Werte

Der Wert eines Pointers
ist eine Adresse

- Der Arrayname ist ein Zeiger auf die Anfangsadresse des Arrays (also das erste Element)

```
int array[5] = {0, 1, 2, 3, 4};
```

| [0] | [1] | [2] | [3] | [4] |
|--------|--------|--------|--------|--------|
| 0x1064 | 0x1068 | 0x106C | 0x1070 | 0x1074 |

- Adresse eines Arrays einem Pointer zuweisen

- `int *i_0 = array`
- `int *i_1 = &array[1]`

Der Arrayname zeigt auf die Anfangsadresse des Arrays, daher kann man den Adressoperator weglassen, wenn man ein Array einem Zeiger zuweist. Mit Adressoperator geht aber natürlich auch: `int *i = &array` oder `int *i = &array[0]`

Möchte man die Speicheradresse eines anderen als des ersten Arrayelements (das zweite Arrayelement z.B.), dann benötigt man den Adressoperator (weil Arrays eben keine Zeiger sind)

- Der Arrayname ist ein Zeiger auf die Anfangsadresse des Arrays (also das erste Element)

```
int array[5] = {0, 1, 2, 3, 4};
```

| [0] | [1] | [2] | [3] | [4] |
|--------|--------|--------|--------|--------|
| 0x1064 | 0x1068 | 0x106C | 0x1070 | 0x1074 |

- Adresse eines Arrays einem Pointer zuweisen

- int *i_0 = array
- int *i_1 = &array[1]

Der Arrayname zeigt auf die Anfangsadresse des Arrays, daher kann man den Adressoperator weglassen, wenn man ein Array einem Zeiger zuweist. Mit Adressoperator geht aber natürlich auch: int *i = &array oder int *i = &array[0]

Möchte man die Speicheradresse eines anderen als des ersten Arrayelements (das zweite Arrayelement z.B.), dann benötigt man den Adressoperator (weil Arrays eben keine Zeiger sind)

Übung:
Wie können Sie einem Pointer die Adresse des zweiten Arrayelements zuweisen?

Pointer werden immer mit einem Datentyp deklariert:

```
int *p; // der Pointer kann eingesetzt werden, um auf Integerwerte zu zeigen
```

- Der Datentyp des Pointers gibt an, auf welchen Datentyp der Pointer zeigen kann. Ein Integer-Pointer kann also nicht auf eine Float-Variable zeigen
- Pointer selbst haben immer dieselbe Größe (unabhängig davon, auf was für einen Datentyp der Pointer zeigt)
- Die Pointergröße ist systemabhängig, ein typischer Wert sind 8 Byte == 64 Bit auf modernen 64 Bit Betriebssystemen) oder 4 Byte auf Mikrocontrollern oder älteren Betriebssystemen.

```
int *i;  
sizeof(i);
```

Pointern wird ein Datentyp zugewiesen, um Pointer-Arithmetik zu ermöglichen:

- Dadurch, dass dem Pointer der Datentyp auf den er zeigt bekannt ist, kann die Adresse des Vorgänger- oder Nachfolgeelements berechnet werden



```
int main(){
    int intArray[SIZE] = {100, 111, 103, 10, 0};
    int *pointer;

    for (int i = 0; i < SIZE; i++){
        printf("Der Wert, auf den der Pointer zeigt:\n
        %d\n", *(pointer+i));
    }
}
```


Pointern wird ein Datentyp zugewiesen, um Pointer-Arithmetik zu ermöglichen.

- Dadurch, dass dem Pointer der Datentyp auf den er zeigt bekannt ist, kann die Adresse des Nachfolgeelements berechnet werden



Damit der Zeiger auf die nächste Adresse zeigt, muss die Operation `pointer+i` in Klammern stehen. Dadurch springt zunächst die Adresse ein Element weiter, bevor der Wert dereferenziert wird. Andernfalls würde zuerst der Wert dereferenziert und dann `i` hinzugezählt.

Priorität der Operanden:
Klammern > Dereferenzierung >
Addition/Subtraktion

```
int main(){
    int intArray[SIZE];
    int *pointer;

    for (int i = 0; i < SIZE; i++){
        printf("Der Wert, auf den der Pointer zeigt:\n%d\n", *(pointer+i));
    }
}
```

Übung:

Bevor wir uns den gerade gezeigten Code in der IDE anschauen, versuchen Sie ihn nachzuskizzieren:

- Variablen: Integerarray & Integerpointer
- Funktion: Ausgeben der Arrayelemente mittels Pointerarithmetik, das heißt die Adresse des Pointers wird in der For-Schleife hochgezählt und der Wert mittels Dereferenzierung ausgegeben

Abschließend lässt sich die Ähnlichkeit von Pointern und Arrays so zusammenfassen:

- Pointerarithmetik und Array-Indexierung sind gleichwertig, Pointer und Arrays aber nicht

Auch wenn die Pointer- und Arrayoperationen gleich ausgeführt werden – die Berechnungen im Hintergrund unterscheiden sich

- Ein Array ist ein einzelner, vorbelegter Datenblock aneinanderhängender Elemente desselben Datentyps
- Ein Pointer ist eine Referenz auf jedweden Datentyp, egal wo. Pointer zeigen auf anderweitig belegten Speicherplatz und können zur Laufzeit des Programms einem anderen Speicherbereich zugeordnet werden (so, wie eine Variablen einen anderen Wert annehmen kann)

| Pointer | | |
|-----------------------------|-------------------------------|---|
| int *p; | Deklaration | |
| p = &i; | Initialisierung | |
| *p; | Dereferenzierung (Wert von i) | |
| p; | Adresse, auf die p zeigt | |
| &p; | Adresse von p | |
| Auf Arrayelemente zugreifen | | |
| Zeigervariante | Arrayvariante | |
| *pointer | *array | erstes Element des Arrays |
| pointer[0] | array[0] | |
| *(pointer+n) | *(array+n) | Zugriff auf <i>n</i> tes Element |
| pointer[n] | array[n] | |
| pointer | array | Zugriff auf die Adresse des ersten Elements |
| &pointer[0] | &array[0] | |
| pointer+n | array+n | Zugriff auf die Adresse des <i>n</i> ten Elements |
| &pointer[n] | &array[n] | |

Übung:

Sie haben ein Array mit fünf Zahlen (1, 2, 3, 4, 5) und einen Pointer.

- Deklarieren Sie den Pointer
- Deklarieren und Initialisieren Sie das Array
- Geben Sie die Adresse des Pointers und des Arrays aus
- Initialisieren Sie den Pointer, so dass er auf das erste Element des Arrays zeigt
- Geben Sie die Adresse aus, auf die der Pointer zeigt
- Geben Sie über den Pointer den ersten, dritten und fünften Wert des Arrays aus (immer mit einer anderen Methode)
- Ändern Sie den fünften Wert des Arrays über den Pointer in 10 um

| Pointer | | |
|-----------------------------|-------------------------------|---|
| int *p; | Deklaration | |
| p = &i; | Initialisierung | |
| *p; | Dereferenzierung (Wert von i) | |
| p; | Adresse, auf die p zeigt | |
| &p; | Adresse von p | |
| Auf Arrayelemente zugreifen | | |
| Zeigervariante | Arrayvariante | |
| *pointer | *array | erstes Element des Arrays |
| pointer[0] | array[0] | |
| *(pointer+n) | *(array+n) | Zugriff auf ntes Element |
| pointer[n] | array[n] | |
| pointer | array | Zugriff auf die Adresse des ersten Elements |
| &pointer[0] | &array[0] | |
| pointer+n | array+n | Zugriff auf die Adresse des nten Elements |
| &pointer[n] | &array[n] | |

- In der letzten Vorlesung haben wir gesehen, dass Arrays Call-by-Reference an Funktionen übergeben werden
- Jetzt da wir wissen wie Pointer auf Arrays zeigen, welche **drei** Möglichkeiten gibt es, Arrays an Funktionen zu übergeben?

Übung:

```
int array[3] = {1, 2, 3}; // deklariert & initialisiert ein Array mit 3 Elementen  
int *pointer; // deklariert einen Pointer
```

```
pointer = array; // weist dem Pointer die Adresse des ersten Arrayelements zu
```

```
function(_____);
```

Auf welche drei Arten können Arrays an Funktionen übergeben werden?

- Adresse eines Arrays einem Pointer zuweisen
 - `int *i_0 = array` // Adresse 1. Arrayelement
 - `int *i_1 = &array[1]` // Adresse 2. Arrayelement
- Arrayelement mittels Pointer auslesen
 - `var = *pointer` // erstes Arrayelement
 - `var = *(pointer+2)` // drittes Arrayelement
 - `var = pointer[2]` // drittes Arrayelement
- Array an eine Funktion übergeben
 - `function(array, numElements);`
 - `function(&array[0], numElements);`
 - `function(pointer, numElements);`

```
void function(int *array, int numElements){
...
}

int main(){
    int numElements = SIZE;
    int array[SIZE] = {6, 2, 3, 4, 5};
    int *pointer;
    pointer = array;
    ...
}
```

- Adresse eines Arrays einem Pointer zuweisen
 - `int *i_0 = array` // Adresse 1. Arrayelement
 - `int *i_1 = &array[1]` // Adresse 2. Arrayelement
- Arrayelement mittels Pointer auslesen
 - `var = *pointer` // erstes Arrayelement
 - `var = *(pointer+2)` // drittes Arrayelement
 - `var = pointer[2]` // drittes Arrayelement
- Array an eine Funktion übergeben
 - `function(array, numElements);`
 - `function(&array[0], numElements);`
 - `function(pointer, numElements);`

```
void function(int *array, int numElements){
...
}

int main(){
    int numElements = SIZE;
    int array[SIZE] = {6, 2, 3, 4, 5};
    int *pointer;
    pointer = array;
    ...
}
```

Übung:
Wie können Sie einer Funktion das Array ab
der zweiten Stelle übergeben?

```
void function(int array, int numElements){  
...  
}  
  
int main(){  
    int numElements = SIZE;  
    int array[SIZE] = {6, 2, 3, 4, 5};  
    int *pointer;  
    pointer = array;  
    ...  
    function(array[1], numElements);  
    function((pointer+1), numElements);  
  
}
```


- Adresse eines Arrays einem Pointer zuweisen
 - `int *i_0 = array`
 - `int *i_1 = &array[1]`
- Arrayelement mittels Pointer auslesen
 - `var = *pointer` // erstes Arrayelement
 - `var = *(pointer+2)` // drittes Arrayelement
 - `var = pointer[2]` // drittes Arrayelement
- Array an eine Funktion übergeben
 - `function(array, numElements);`
 - `function(&array[0], numElements);`
 - `function(pointer, numElements);`

```
void function(int *array, int numElements){  
...  
}  
  
int main(){  
    int numElements = SIZE;  
    int array[SIZE] = {6, 2, 3, 4, 5};  
    int *pointer;  
    pointer = array;  
    ...  
    function(&array[1], numElements-1);  
    function((pointer+1), numElements-1);  
}
```

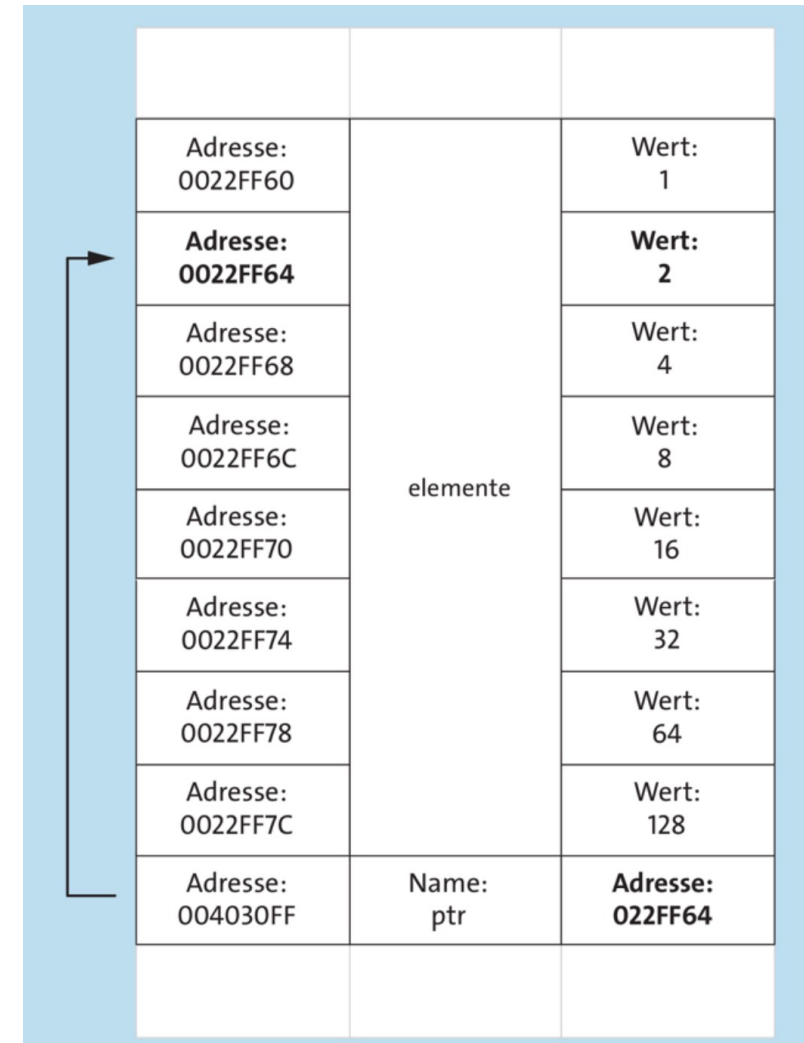
Achtung:

Die Variable `numElements`, die die Anzahl der Elemente im Array begonnen mit dem ersten Element angibt, muss ebenfalls angepasst werden um Bereichsüberschreitungen zu verhindern.

Array & Pointer

Was Sie sich merken sollten

- Arrays sind Datentypen, die mehrere Elemente umfassen
- Der Arrayname zeigt auf das erste Element im Array
- Dadurch ähneln sich Arrays und Pointer in verschiedenen Aspekten, insbesondere sind Pointerarithmetik und Arrayindexierung gleichwertig. Arrays sind aber dennoch keine Pointer
- Denn: Pointer speichern keine Werte ab, sondern Adressen



Pointer & Strings

- In der letzten Vorlesung haben wir Character-Arrays kennengelernt, also Strings
`char string[6] = "Hallo";`
- Strings sind Zeichenketten, die mit einem String-Ende-Zeichen ('\0') abgeschlossen werden
 - `char dog[] = "dog";` // Hier wird \0 automatisch angefügt. Ist gleichwertig zu
 - `char dog[] = 'd', 'o', 'g', '\0';` // ist gleichwertig zu
 - `char dog[] = 100, 111, 103, 0;`
- Wie bei anderen Arrays ist der Character-Arrayname ein Pointer auf das erste Element im Array

Ein **String** kann anders als ein Array auch als Pointer auf das erste Stringelement implementiert werden

`char *dog = "dog";` // ist ein Pointer auf das erste Element des String-Literals, 'd'

- Ein Pointer der auf einen String zeigt, zeigt auf die Adresse des ersten Buchstabens

- **Was nicht geht:**

`char *dog = {'d', 'o', 'g'}` bzw. `char *dog = {'d', 'o', 'g', '\0'}`;

`char array[] = "dog";`



`char *pointer = "dog";`



- In der Headerdatei <string.h> sind verschiedene Funktionen implementiert, mit denen man Strings bearbeiten kann
- Die Funktionen verwenden Character-Zeiger (char *), die auf die Anfangsadresse des eingelesenen Strings verweisen
- Dasselbe gilt für Input-Output-Funktionen, die wir bisher schon kennengelernt haben

Schaut man sich die Parameterlisten von Funktionen der Standard-Input-Output-Bibliothek an (stdio.h), so sieht man, dass printf, scanf, gets und andere Funktionen die Anfangsadresse eines String-Literals einlesen

```
char    *gets(char *);

void     perror(const char *) __cold;
int  printf(const char * __restrict, ...) __printflike(1, 2);
int  putc(int, FILE *);
int  putchar(int);
int  puts(const char *);
int  remove(const char *);
int  rename (const char *__old, const char *__new);
void  rewind(FILE *);
int  scanf(const char * __restrict, ...) __scanflike(1, 2);
void  setbuf(FILE * __restrict, char * __restrict);
int  setvbuf(FILE * __restrict, char * __restrict, int, size_t);
```

Frage:

Warum ist die Stringkonstante in der printf-Parameterliste explizit als Konstante (const) deklariert?

Schaut man sich die Parameterlisten von Funktionen der Standard-Input-Output-Bibliothek an (stdio.h), so sieht man, dass printf, scanf, gets und andere Funktionen die Anfangsadresse eines String-Literals einlesen

```
char    *gets(char *);  
  
void     perror(const char *) __cold;  
int  printf(const char * __restrict, ...) __printflike(1, 2);  
int  putc(int, FILE *);  
int  putchar(int);  
int  puts(const char *);  
int  remove(const char *);  
int  rename (const char *__old, const char *__new);  
void  rewind(FILE *);  
int  scanf(const char * __restrict, ...) __scanflike(1, 2);  
void  setbuf(FILE * __restrict, char * __restrict);  
int  setvbuf(FILE * __restrict, char * __restrict, int, size_t);
```

Warum ist die Stringkonstante in der printf-Parameterliste explizit als Konstante (const) deklariert?


- Konstanten können nicht ohne Compiler-Fehler verändert werden (Stringkonstanten bzw. Stringlitterale schon)
- Die Deklaration als Konstante ist eine Sicherheitsmaßnahme, um böswillige Manipulationen des Strings zu verhindern
- Pointer, die mit dem Zusatz const deklariert werden, sind sogenannte Read-Only-Pointer


```
char *fgets(char *string, int anzahl_zeichen, FILE *stream);
```

```
int main(){  
  
    char string[STRINGSIZE];  
  
    printf("Bitte einen Satz eingeben: \n");  
    fgets(string, STRINGSIZE, stdin);  
    printf("Eingelesener Text: %s\n", string);  
  
    return 0;  
}
```

strlen(): Die Länge eines Strings ermitteln

`size_t strlen(const char *string1);` // ermittelt die Länge von string1, ohne String-Ende-Zeichen

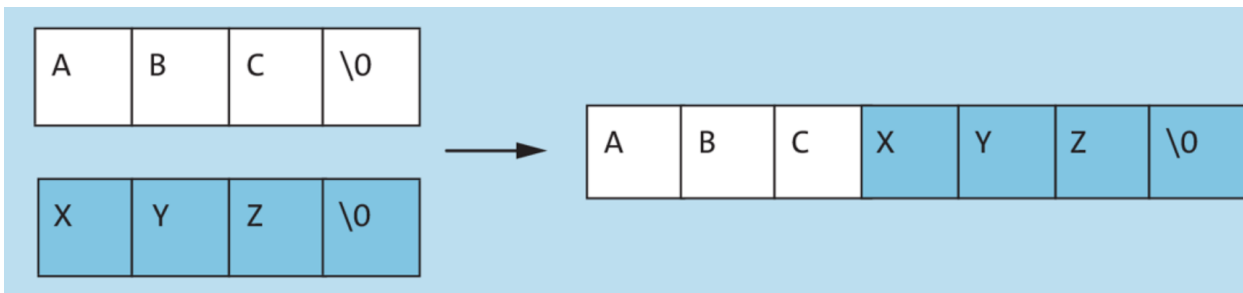


size_t ist ein primitiver, arithmetischer Datentyp für
Größenangaben, der Werte zwischen 0 und der
größtmöglichen, positiven Integer auf dem System
annehmen kann (entspricht meist unsigned long long int)
Formatzeichen: %zu (für printf)

- `strlen()`: Die Länge eines Strings ermitteln
`size_t strlen(const char *string1);` // ermittelt die Länge von string1, ohne String-Ende-Zeichen

- `strcat()`: Strings aneinanderhängen (string concatenate)

`char *strcat(char *string1, const char *string2);` // hängt zwei Strings hintereinander



Achtung:

`strcat()` fügt den zweiten String an den ersten an, überprüft aber nicht, ob der Speicherplatz des ersten Strings ausreichend groß ist. Ist er das nicht, wird anderweitig belegter Speicherplatz überschrieben!

- `strlen()`: Die Länge eines Strings ermitteln
`size_t strlen(const char *string1);` // ermittelt die Länge von string1, ohne String-Ende-Zeichen
- `strcat()`: Strings aneinanderhängen (string concatenate)
`char *strcat(char *string1, const char *string2);` // hängt zwei Strings hintereinander
- `strcmp()`: Vergleicht zwei Strings (string compare)
`char *strcmp(const char *string1, const char *string2);` // gibt 0 zurück, wenn die Strings identisch sind

- `strlen()`: Die Länge eines Strings ermitteln
`size_t strlen(const char *string1);` // ermittelt die Länge von string1, ohne String-Ende-Zeichen
- `strcat()`: Strings aneinanderhängen (string concatenate)
`char *strcat(char *string1, const char *string2);` // hängt zwei Strings hintereinander
- `strcmp()`: Vergleicht zwei Strings (string compare)
`char *strcmp(const char *string1, const char *string2);` // gibt 0 zurück, wenn die Strings identisch sind
- `strcpy()`: Kopiert einen String (string copy)
`char *strcpy(char *string1, const char *string2);` // kopiert einen String in einen anderen

String in den
kopiert wird

kann auch ein String-Literal sein,
also z.B. "Hallo"

Achtung:
Auch `strcpy()` überprüft nicht, ob der Speicherplatz des ersten Strings ausreichend groß ist. Ist er das nicht, wird anderweitig belegter Speicherplatz überschrieben!

- `strlen()`: Die Länge eines Strings ermitteln
`size_t strlen(const char *string1);` // ermittelt die Länge von string1, ohne String-Ende-Zeichen
- `strcat()`: Strings aneinanderhängen (string concatenate)
`char *strcat(char *string1, const char *string2);` // hängt zwei Strings hintereinander
- `strcmp()`: Vergleicht zwei Strings (string compare)
`char *strcmp(const char *string1, const char *string2);` // gibt 0 zurück, wenn die Strings identisch sind
- `strcpy()`: Kopiert einen String (string copy)
`char *strcpy(char *string1, const char *string2);` // kopiert einen String in einen anderen
- `strtok()`: Zerlegt einen String mit vorgegebenen Zeichen (string token)
`char *strtok(char *string1, const char *string2);` // trennt String 1 in zwei Hälften, die durch ein bestimmtes Token (=spezielles Zeichen / Wort) verbunden sind

- Strings können entweder als Character-Array oder als Character-Pointer implementiert werden
- Funktionen, die Strings einlesen oder ausgeben, tun das für gewöhnlich mittels Character-Pointern
- Syntax
`char *name = "string";` // ist ein Pointer auf das erste Element des String-Literals
- Werden Character-Pointer mit printf und dem Formatierungszeichen %s ausgegeben, so müssen sie nicht dereferenziert werden
`printf("Inhalt des Strings: %s", charPointer);`
- In der Headerdatei <string.h> sind verschiedene Funktionen implementiert, mit denen man Strings bearbeiten kann
- Nicht alle Funktionen in string.h sind sicher, das heißt man muss immer darauf achten, dass die Strings mit denen man arbeitet ausreichend groß sind

Speicherkonzepte

Stack- & Heapspeicher

- In den vorherigen Folien haben wir gesehen, warum die scanf-Funktion die Speicheradresse der Variablen einliest, in die gespeichert werden soll und dass Pointerarithmetik genutzt werden kann, um auf Arrays zuzugreifen
- Das sind aber Spezialfälle der Pointernutzung
- Wichtiger Use Case von Pointern ist die dynamische Speicherverwaltung sowie die Möglichkeit, die Lebensdauer von Variablen selbst zu verwalten
- Um diese Methoden zu verstehen, ist ein Einblick in die Speicherkonzepte von C-Programmen hilfreich

Das Speicherkonzept von C-Programmen

- Bei der Betrachtung des Geltungsbereichs von Variablen sowie der Rekursion haben wir uns schon mit dem Stack-Speicher beschäftigt
- Insgesamt gibt es vier Speicherbereiche, die im Verlauf eines C-Programms zur Verwendung kommen

| Codespeicher | Datenspeicher | Stackspeicher | Heapspeicher |
|---|---|---|---|
| Enthält den auszuführenden Programmcode in Maschinensprache | Speichert alle statischen Daten, also globale und statische Variablen | Speichert und löscht automatisch Funktionsdaten | Manuell belegbarer Speicher für die dynamische Speicherverwaltung |

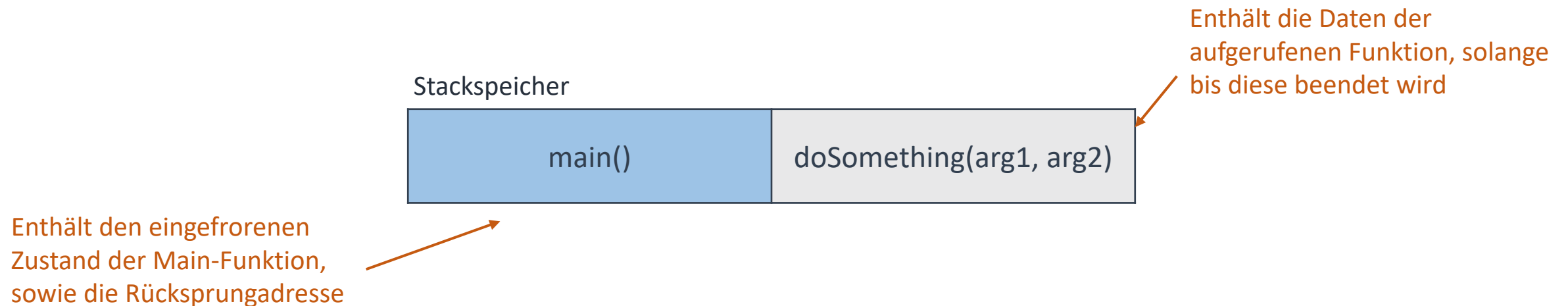
Das Speicherkonzept von C-Programmen: Stack & Heap

| | Stackspeicher | Heapspeicher |
|-------------------|---|---|
| Umfang | Abgeschlossenes Stacksegment für jede Funktion, die kopierten Funktionsargumente, lokale Funktionsvariablen sowie die Rücksprungadresse zur aufrufenden Funktion. | Steht allen Funktionen gleichermaßen zur Verfügung |
| Speicherzuweisung | Stackspeicher wird automatisch bereitgestellt und wieder freigegeben | Heapspeicher muss manuell reserviert und wieder freigegeben werden. Andernfalls entsteht ein Speicherleck (memory leak) |
| Lebensdauer | Daten im Stackspeicher werden automatisch freigegeben, sobald die Funktion endet | Daten im Heapspeicher bleiben bestehen, bis sie freigegeben werden |
| Schnelligkeit | schnell | langsamer |
| Größe | Fixierte Größe | Neuer Speicherbereich kann vom Betriebssystem zur Verfügung gestellt werden. |
| Schwachstelle | Reicht der Platz im Stack nicht aus, spricht man von Stack Overflow. Tritt häufiger bei tief verschachtelten oder rekursiven Funktionen auf. | Werden Daten auf dem Heap als unzusammenhängende Blöcke gespeichert, kann es zu Fragmentierungen kommen. Bei starken Fragmentierungen kann nicht mehr genug zusammenhängender Speicher bereitgestellt werden. |
| Anwendungsbereich | <ul style="list-style-type: none"> • Default-Speicher • Daten mit geringem Speichervolumen | <ul style="list-style-type: none"> • Daten mit hohem Speichervolumen • Anwendungen, bei denen der Speicherbedarf nicht von vornherein bekannt ist (dynamische Arrays) • Zur manuellen Festlegung der Daten-Lebensdauer |

Stackpeicher

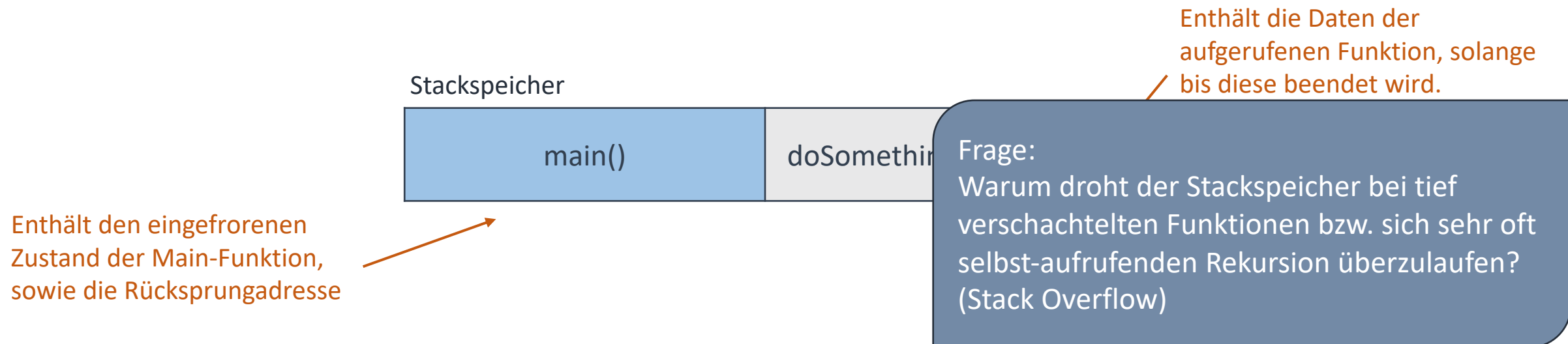
Den Stackspeicher haben wir uns in der 3. Vorlesung im Rahmen rekursiver Funktionen genauer angeschaut. Der Stackspeicher wird automatisch vom Prozessor verwaltet:

- Auf dem Stack liegt immer der Startup-Code sowie die Funktionsdaten der main-Funktion (Argumente, lokale Variablen)
- Wird eine neue Funktion aufgerufen, wird die aufrufende (main-)Funktion eingefroren und ein neuer Speicherbereich für die Daten der aufgerufenen Funktion bereitgestellt



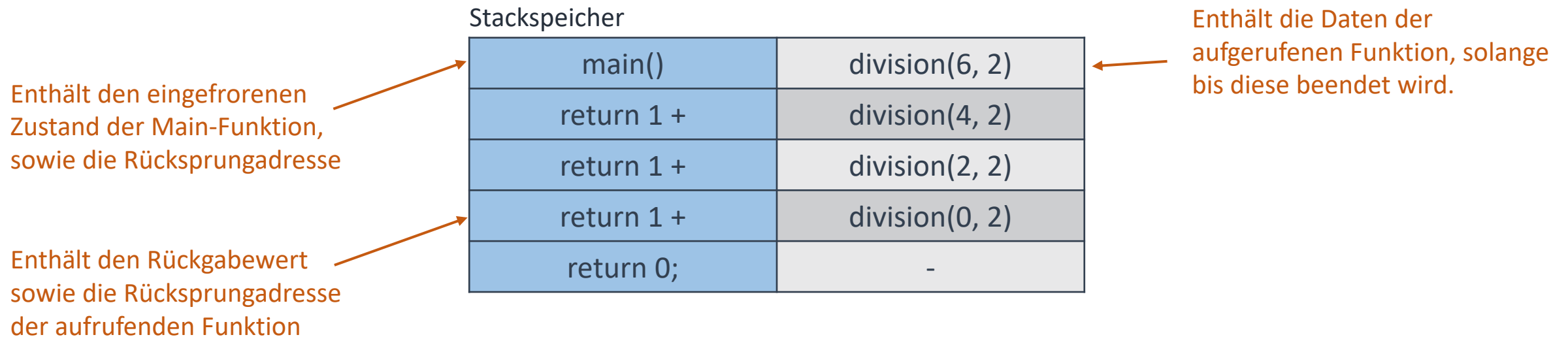
Den Stackspeicher haben wir uns in der 3. Vorlesung im Rahmen rekursiver Funktionen genauer angeschaut. Der Stackspeicher wird automatisch vom Prozessor verwaltet:

- Auf dem Stack liegt immer der Startup-Code sowie die Funktionsdaten der main-Funktion (Argumente, lokale Variablen)
- Wird eine neue Funktion aufgerufen, wird die aufrufende (main-)Funktion eingefroren und ein neuer Speicherbereich für die Daten der aufgerufenen Funktion bereitgestellt



Tief verschachtelte Funktionen (also Funktionen, die immer wieder neue Funktionen aufrufen) bzw. oft sich selbst aufrufende Rekursionen können einen Stack Overflow auslösen:

- Mit jedem neuen Funktionsaufruf wird dem Stack ein neuer Datenstapel (engl. Stack) hinzugefügt
- Erst wenn die Funktion mittels return beendet wird, wird der Speicherbereich wieder freigegeben
- Da der Stackspeicher eine fixierte Größe hat, kann der Speicherplatz zu Ende gehen → Stack Overflow



Stackspeicher: Begrifflichkeiten, die Sie sich merken sollten

- **Lokale Variablen**

Bezeichnet Variablen, die nur innerhalb eines bestimmten Geltungsbereichs (scope) gültig sind. Beispiel:

- Variablen, die innerhalb einer Schleife deklariert werden, sind nur innerhalb der Schleife gültig und sichtbar
- Variablen, die innerhalb einer Funktion / in der Funktionsparameterliste deklariert werden, sind nur innerhalb der Funktion gültig und sichtbar

- **Automatische Variable**

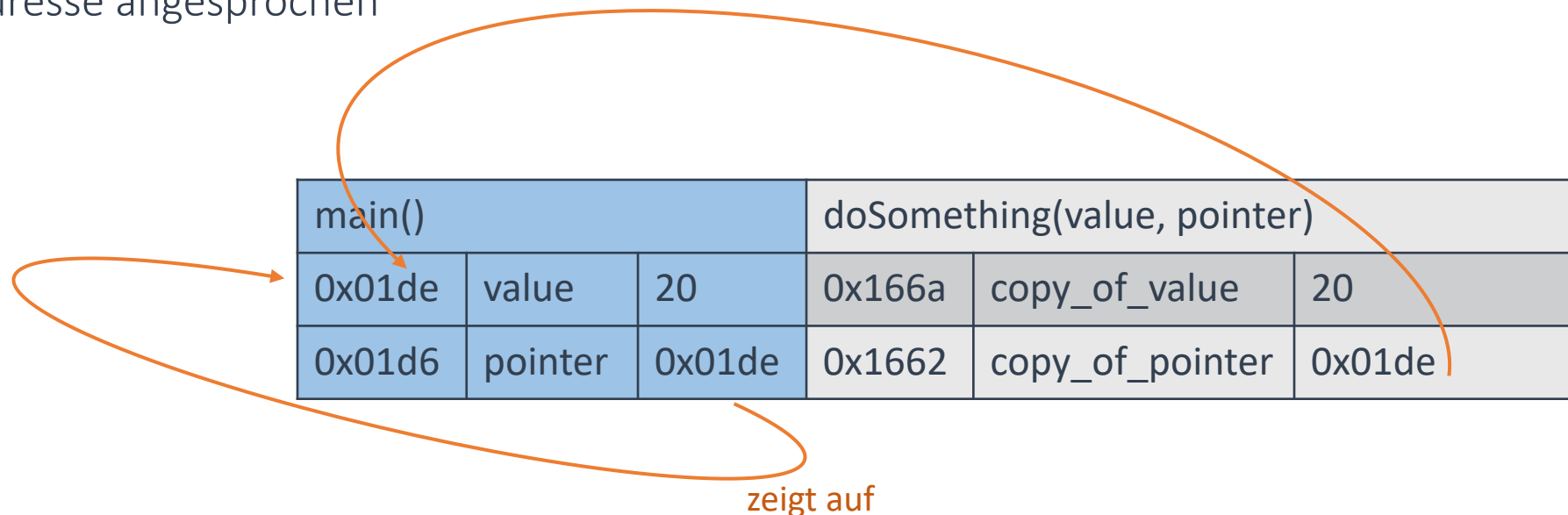
Bezieht sich darauf, dass lokale Variablen vom Compiler automatisch angelegt und wieder gelöscht werden, z.B. wenn die Funktion beendet ist. Lokale Variablen sind in der Regel automatische Variablen. Ausnahme: lokale statische Variablen: Diese sind ebenfalls nur innerhalb ihrer Funktion sichtbar, werden aber nach Beendigung der Funktion nicht gelöscht.

- **Stackframe**

Wird eine Funktion aufgerufen, werden die lokalen Funktionsvariablen sowie die Rücksprungadresse zur aufrufenden Funktion als Datenblock auf dem Stack abgelegt. So hat jede Funktion einen eigenen Datenblock, auf Englisch Stackframe. Endet eine Funktion, wird ihr Stackframe gelöscht.

Stackpeicher & Pointer in Funktionen

- Nimmt eine Funktion Parameter auf, so werden diese Call-by-Value übergeben, das heißt, die Parameterwerte werden in die Funktion kopiert
- Übergibt man jedoch Pointer an die Funktion, so werden die Werte Call-by-Reference (per Referenz) übergeben. Das heißt:
 - Die Adresse, die im Pointer hinterlegt wird, wird in die Funktion kopiert
 - Der Wert, der an der Adresse hinterlegt ist, wird nicht in die Funktion kopiert sondern wird über die Adresse angesprochen



Pointer auf Stackspeicher als Funktionsparameter

Die Variablen a, b und c werden in der main-Funktion angelegt. Das heißt es wird im Stackframe der main-Funktion Speicherplatz für diese drei Integervariablen reserviert und mit 2, 3 und 0 initialisiert.

| main() | | |
|--------|---|---|
| 0x012a | a | 2 |
| 0x0126 | b | 3 |
| 0x0122 | c | 0 |

```
void addNumbers(int *x, int *y, int *result){  
    *result = *x + *y;  
}  
  
int main(){  
    int a = 2, b = 3, c = 0;  
  
    addNumbers(&a, &b, &c);  
  
    return 0;  
}
```

Die Funktion liest die Adressen der Variablen a, b und c ein.

Pointer auf Stackspeicher als Funktionsparameter

Kein Rückgabewert, da die Wertänderung direkt im Stackframe (Speicherbereich) der main-Funktion vorgenommen wird

Die Funktionsparameter sind Pointer. Das heißt in der Funktion werden drei Pointer im Stackframe der addNumbers-Funktion angelegt und mit den Adressen von a, b und c belegt.

| addNumbers() | | |
|--------------|----------------|--------|
| 0x1fb1 | pointer x | 0x012a |
| 0x1fa8 | pointer y | 0x0126 |
| 0x1fa0 | pointer result | 0x0122 |

```
void addNumbers(int *x, int *y, int *result){
    *result = *x + *y;
}

int main(){
    int a = 2, b = 3, c = 0;

    addNumbers(&a, &b, &c);

    return 0;
}
```

Die Werte von a und b, auf die die Pointer x und y zeigen, werden vom Stackframe der main-Funktion dereferenziert und das Ergebnis der Addition wird in die Variable c geschrieben, auf die der Pointer result zeigt.

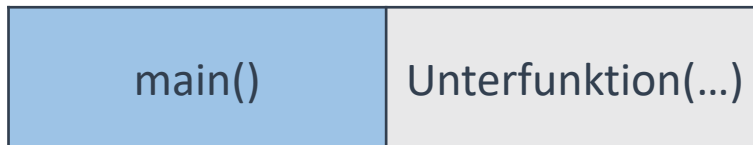
| main() | | |
|--------|---|---|
| 0x012a | a | 2 |
| 0x0126 | b | 3 |
| 0x0122 | c | 0 |

Der Stackspeicher ist der Default-Speicher für Funktionsdaten

- Wird eine neue Funktion aufgerufen, so wird ein neuer Stackframe reserviert und die Funktionsvariablen werden in diesem Bereich gespeichert. Jede Funktion hat also ihren eigenen Stackframe
- Wird die Funktion beendet, so wird der entsprechende Stackframe freigegeben
- Sollen lokale Daten aus anderen Funktionen (z.B. der main-Funktion) verwendet werden, so müssen diese kopiert werden
 - Bei Variablen werden die Werte kopiert
 - Bei Pointern wird die Adresse kopiert
- **Call-by-Value:** Die Übergabe von Variablenwerten an eine Funktion mittels Kopie. Wird bei primitiven Datentypen (int, float, double, char, bool, etc.) verwendet
- **Call-by-Reference:** Indirekte Übergabe der Werte über die Stack-Speicheradresse mittels Pointern. Wird bei komplexen Datentypen (array, string, struct) verwendet

- Übung:
- Überlegen Sie sich zwei Szenarien:
- Sie rufen über die main-Funktion eine Unterfunktion auf und übergeben eine Integervariable mittels call-by-value. Was passiert hierbei auf dem Stack?
 - Sie rufen über die main-Funktion eine Unterfunktion auf und übergeben einen Pointer. Dieser Pointer zeigt auf die Adresse einer Integervariablen. Was passiert hierbei auf dem Stack?

Stackspeicher



```
void Unterfunktion(parameter){  
...}
```

```
int main(){  
void Unterfunktion(argument);  
}
```

Heapspeicher

- Der Heapspeicher wird meistens vom Betriebssystem verwaltet
- Wie auch der Stackspeicher wächst der Heapspeicher bei Bedarf an bzw. gibt Speicher frei
- Wird Heapspeicher angefordert, so sieht das Betriebssystem im Arbeitsspeicher nach, ob noch ausreichend freier bzw. zusammenhängender Speicher vorhanden ist
- Während der Stackspeicherplatz automatisch vom Prozessor verwaltet wird, muss Heapspeicher durch den/die Programmierer:in manuell angefordert werden
- Das ist sinnvoll,
 - wenn die Lebenszeit von Daten nicht vom Programm verwaltet werden soll
 - zur dynamischen Speicherverwaltung
 - wenn die Daten ein hohes Speichervolumen haben

| Codespeicher | Datenspeicher | Stackspeicher | Heapspeicher |
|---|---|---|---|
| Enthält den auszuführenden Programmcode in Maschinensprache | Speichert alle statischen Daten, also globale und statische Variablen | Speichert und löscht automatisch Funktionsdaten | Manuell belegbarer Speicher für die dynamische Speicherverwaltung |

Heapspeicher manuell reservieren

Heapspeicher

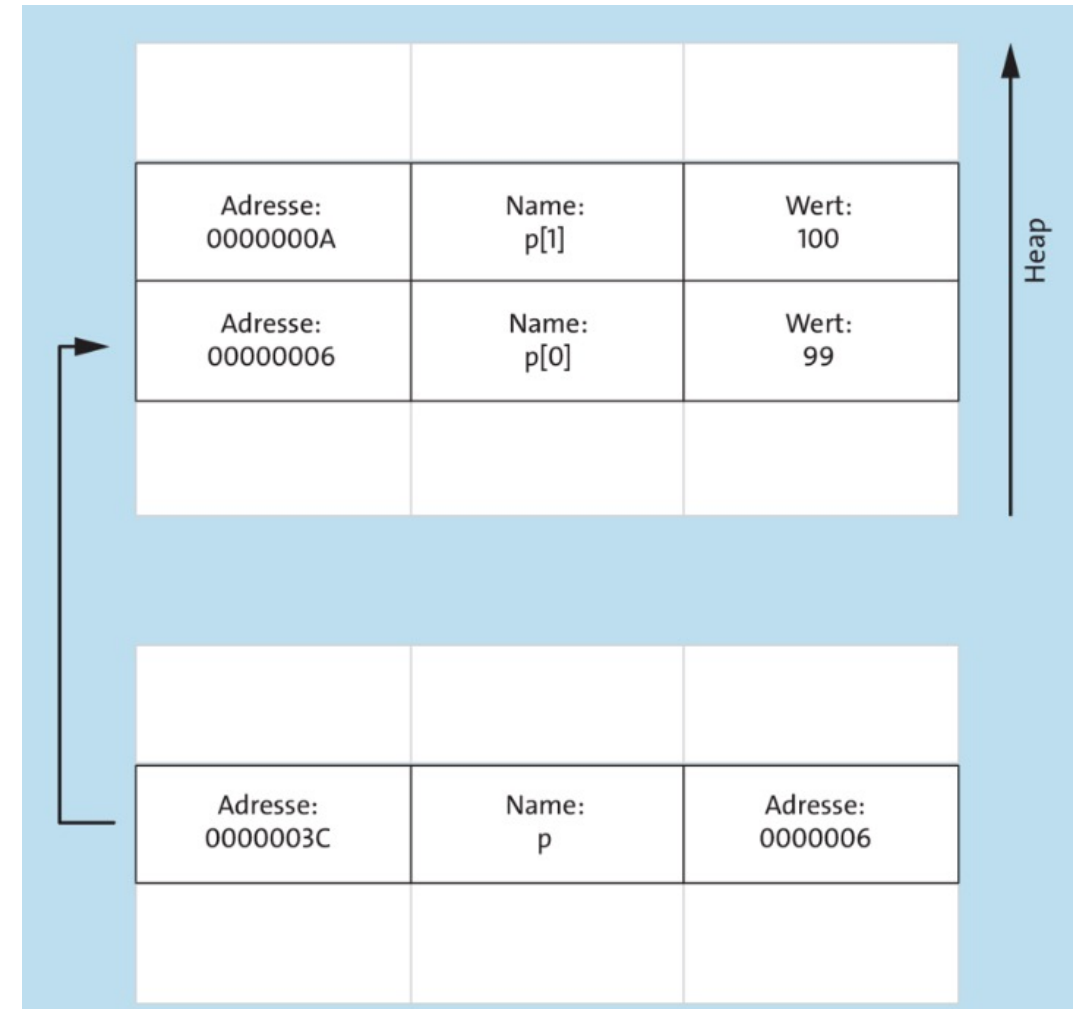
Speicher allokieren mit malloc()

- Um Speicher auf dem Heap zu allokieren (also zu reservieren) kann die Funktion malloc() verwendet werden (malloc = memory allocation)
- malloc() ist in der Headerdatei stdlib.h implementiert
- Syntax:

`void *malloc(size_t size);` // Deklaration

- Verwendung:

```
p = (int *) malloc(2*sizeof(int)); // Speicherallokation
*p = 99; // Wertzuweisung 1. Speicherplatz
*(p + 1) = 100; // Wertzuweisung 2. Speicherplatz
```



Explizite Typumwandlung zu int, da
malloc() einen void-Pointer ausgibt
(gleich dazu mehr).

Pointer, der auf die Anfangsadresse
des reservierten Speichers zeigt

int *p;

p = (int *) malloc(2*sizeof(int)); // Speicherallokation

*p = 99; // Wertzuweisung

*(p + 1) = 100; // Wertzuweisung

Größe des zu reservierenden Speichers angeben:

- (sizeof(Datentyp))
- (n*sizeof(Datentyp))
- sizeof(*p) // Größe des dereferenzierten Werts
- **falsch:** sizeof(p) // Größe des Pointers (z.B. 8 Byte)
- schlechter Stil: malloc(32) // magic number

Solange der reservierte Speicher nicht mit Werten
initialisiert wurde, sind die Werte undefiniert!

Grundsätzlich: Die Größe von Datentypen variiert von
System zu System, was bei Portierung zu Fehlern führen
kann

- Um Speicher auf dem Heap zu allokalieren (also zu reservieren) kann die Funktion malloc() verwendet werden (malloc = memory allocation)
- malloc() ist in der Headerdatei stdlib.h implementiert
- Syntax:

void *malloc(size_t size);

↑
malloc() gibt einen void pointer zurück

Falls nicht ausreichend zusammenhängender Heapspeicher gefunden wird, zeigt der Pointer auf NULL

```
int main(){  
    int *pointer;  
    pointer = (int *)malloc(2*sizeof(int));  
  
    if(pointer != NULL){  
        *pointer = 1337;  
        *(pointer + 1) = 42;  
        printf("Allokation \\  
erfolgreich.\n");  
    }  
    else{  
        printf("Kein virtueller RAM \\  
mehr verfügbar.\n");  
        return;  
    }  
}
```

- Verwendet man malloc() um Heapspeicher zu reservieren, wird ein sogenannter Void-Pointer zurückgegeben
`void *malloc(size_t size);`
- Void-Pointer sind typenlos, das heißt, ihnen ist kein Datentyp zugeordnet

Frage:
Was wäre die Folge, wenn die malloc-Funktion keinen void-Pointer zurückgeben würde?

```
int main(){  
    int *pointer;  
    pointer = (int *)malloc(2*sizeof(int));  
  
    if(pointer != NULL){  
        *pointer = 1337;  
        *(pointer + 1) = 42;  
  
        printf("Allokation \\  
erfolgreich.\n");  
    }  
    else{  
        printf("Kein virtueller RAM \\  
mehr verfügbar.\n");  
        return;  
    }  
}
```

- Verwendet man malloc() um Heapspeicher zu reservieren, wird ein sogenannter Void-Pointer zurückgegeben
- Void-Pointer sind typenlos, das heißt, ihnen ist kein Datentyp zugeordnet
- Dadurch dass ein void-Pointer typenlos ist, kann ihm eine beliebige Adresse zugewiesen werden. Beliebig bedeutet hier: integer, float, double, character, boolean, etc.
- Durch explizite Typumwandlung, kann ein void-Pointer in jeden anderen Datentypen umgewandelt werden

```
int a = 23;  
void *pointer;
```

```
pointer = (int *) &a; // void → int  
printf("%d", *(int *)pointer); // void → int
```

```
int main(){  
  
    void *pointer;  
    int i = 20;  
  
    pointer = (int *) &i;  
  
    printf("\n%d\n", *(int *)pointer);  
  
}
```

- Void-Pointer sind Pointer, denen kein Datentyp zugeordnet ist
- Daher werden void-Pointer häufig als Rückgabewert von Funktionen eingesetzt, deren Rückgabebetyp nicht auf einen Datentyp beschränkt ist
- Um den void-Pointer mit anderen Datentypen zu verwenden, wird der Typ umgewandelt (typecasting)
`pointer = (datentyp *) &variable; // dem Pointer wird die Adresse einer Datentyp-Variable zugewiesen`
- Soll der void-Pointer dereferenziert werden, so müssen zwei Zeiger verwendet werden
`variable = *(datentyp *)pointer; // der Variablen wird der Wert zugewiesen, auf den der Pointer zeigt`

| void-Pointer Typumwandlung | | Beispiel |
|----------------------------------|---------------------------------|--|
| <code>void *p = NULL;</code> | Deklaration | <code>void *p = NULL; int i = 10;</code> |
| <code>p = (typ *) &i;</code> | Typumwandlung & Initialisierung | <code>p = (int *) &i;</code> |
| <code>*(typ *)p = wert;</code> | Dereferenzierung | <code>*(int *)p = 22;</code> |
| <code>var = *(typ *)p;</code> | Dereferenzierung | <code>int x = *(int *)p;</code> |

Übung:

Skizzieren Sie ein Programm, mit:

- void-Pointer, Integer- und Character-Variable
- Weisen Sie dem void-Pointer die Integer zu und geben Sie den Wert der Integer über den Pointer aus
- Weisen Sie den void-Pointer dann dem Character zu und geben Sie das Zeichen aus

| void-Pointer Typumwandlung | | Beispiel |
|----------------------------------|---------------------------------|--|
| <code>void *p = NULL;</code> | Deklaration | <code>void *p = NULL; int i = 10;</code> |
| <code>p = (typ *) &i;</code> | Typumwandlung & Initialisierung | <code>p = (int *) &i;</code> |
| <code>*(typ *)p = wert;</code> | Dereferenzierung | <code>*(int *)p = 22;</code> |
| <code>var = *(typ *)p;</code> | Dereferenzierung | <code>int x = *(int *)p;</code> |

- Um Speicher auf dem Heap zu allokalieren (also zu reservieren) kann die Funktion malloc() verwendet werden (malloc = memory allocation)
- malloc() ist in der Headerdatei stdlib.h implementiert
- Syntax:

`void *malloc(size_t size);`

malloc() gibt einen void pointer zurück ✓

Falls nicht ausreichend zusammenhängender Heapspeicher gefunden wird, zeigt der Pointer auf NULL

```
int main(){  
    int *pointer;  
    pointer = (int *)malloc(2*sizeof(int));  
  
    if(pointer != NULL){  
        *pointer = 1337;  
        *(pointer + 1) = 42;  
        printf("Allokation \\  
erfolgreich.\n");  
    }  
    else{  
        printf("Kein virtueller RAM \\  
mehr verfügbar.\n");  
        return;  
    }  
}
```

Was ist ein NULL-Pointer und warum zeigt der Pointer auf NULL, wenn die Allokation von Heapspeicher **nicht** erfolgreich war?

- NULL ist per Definition ein Wert, der von allen anderen Zeigerwerten unterschiedlich ist
 - Der Adressoperator **&** liefert niemals NULL
 - ein erfolgreicher Aufruf von malloc() liefert niemals NULL
- NULL bedeutet also:
 - der Pointer zeigt noch auf nichts
 - bzw. es ist kein Speicher reserviert worden
- Ein NULL-Pointer ist also **kein** nicht-initialisierter Pointer, welcher undefiniert irgendwohin zeigt
- Sondern, NULL ist eine Möglichkeit, den Pointer **definiert nirgendwohin zeigen zu lassen** → NULL-Pointer

Was ist ein NULL-Pointer und warum zeigt der Pointer auf NULL, wenn die Allokation von Heapspeicher **nicht** erfolgreich war?

- Ein **nicht-erfolgreicher** Aufruf von malloc() liefert einen NULL-Pointer zurück
- Ob der Pointer auf NULL zeigt, kann getestet werden
 - `pointer == NULL`
 - `pointer == 0`
 - `pointer`
- Zeigt der Pointer auf NULL, dann ist klar, dass (je nach Anwendung)
 - entweder kein Heapspeicher frei war
 - bzw. der Zeiger auf noch keine Adresse zeigt

```
int main(){  
    int *pointer;  
  
    pointer = (int *)malloc(2*sizeof(int));  
  
    if(pointer != NULL){  
        *pointer = 1337;  
        *(pointer + 1) = 42;  
  
        printf("Allokation \\  
erfolgreich.\n");  
    }  
    else{  
        printf("Kein virtueller RAM \\  
mehr verfügbar.\n");  
        return;  
    }  
}
```

Ist NULL gleich 0?

- Die kurze Antwort: jein
Denn: NULL kann immer durch 0 ersetzt werden, aber 0 kann nicht immer durch NULL ersetzt werden
- NULL ist ein Präprozessor-Makro, das in der stddef-Headerdatei definiert ist, und das einer der drei Varianten entspricht:
#define NULL 0
#define NULL 0L
#define NULL (void *) 0
- Ob man NULL oder 0 einsetzt ist eine Stilfrage, wobei NULL explizit ist, also klar ausdrückt was gemeint ist
- Verwendet man 0, so wandelt der Compiler die 0 selbstständig in einen Null-Pointer um ((void *) 0)
- (Eine ausdrückliche Typumwandlung (also (void *) 0) ist bei bestimmten Funktionsaufrufen notwendig, wenn z.B. ein Null-Pointer einer Funktion als Argument übergeben wird)

- NULL ist ein Präprozessor-Makro das für Null-Pointer verwendet wird
- Lässt man einen Pointer auf NULL oder 0 zeigen, so zeigt dieser definiert nirgendwo hin
- Die malloc-Funktion gibt einen Null-Pointer zurück, wenn kein Speicherplatz reserviert werden konnte
- Wird der Null-Pointer im Quelltext verwendet, kann man entweder NULL oder 0 einsetzen

```
char *p = NULL;  
p = 0;  
if(p == NULL);  
if(p);
```
- Ist die NULL / 0 ein Funktionsargument, so sollte man eine explizite Typumwandlung verwenden `((typ *) 0)`

```
function(arg, (char *) 0);
```

- Um Speicher auf dem Heap zu allokalieren (also zu reservieren) kann die Funktion malloc() verwendet werden (malloc = memory allocation)
- malloc() ist in der Headerdatei stdlib.h implementiert
- Syntax:

`void *malloc(size_t size);`

malloc() gibt einen void pointer zurück ✓

Falls nicht ausreichend zusammenhängender Heapspeicher gefunden wird, zeigt der Pointer auf NULL ✓

```
int main(){  
    int *pointer;  
    pointer = (int *)malloc(2*sizeof(int));  
  
    if(pointer != NULL){  
        *pointer = 1337;  
        *(pointer + 1) = 42;  
        printf("Allokation \\  
erfolgreich.\n");  
    }  
    else{  
        printf("Kein virtueller RAM \\  
mehr verfügbar.\n");  
        return;  
    }  
}
```

Heapspeicher & Pointer in Funktionen

Pointer auf Heapspeicher als Funktionsparameter

Im Stackframe der main-Funktion werden die Variablen a und b sowie der Pointer p deklariert und initialisiert.

| main() | | |
|--------|---|--------|
| 0x012a | a | 2 |
| 0x0126 | b | 3 |
| 0x011d | p | 0xae10 |

Außerdem wird im Heapspeicher Speicherplatz für eine Integer reserviert. Der Pointer p selbst liegt im Stack von main(), zeigt aber auf den Speicherplatz im Heap.

| Heapspeicher | | | | |
|--------------|--|--|--|--|
| 0xae10 | | | | |

```
void addNumbers(int *x, int *y, int *result){  
    *result = *x + *y;  
}
```

```
int main(){  
    int a = 2, b = 3;  
    int *p = (int*) malloc(sizeof(int));  
  
    addNumbers(&a, &b, p);  
    free(p);  
    return 0;  
}
```

Die Funktion liest die Adressen der Variablen a, b und des Pointers p ein.

Pointer auf Heapspeicher als Funktionsparameter

Die Funktionsparameter sind Pointer. Das heißt in der Funktion werden drei Pointer im Stackframe der addNumbers-Funktion angelegt und mit den Adressen von a, b und p belegt. x und y zeigen auf Stackspeicher, *result* zeigt auf Heapspeicher

| main() | | |
|--------|---|--------|
| 0x012a | a | 2 |
| 0x0126 | b | 3 |
| 0x011d | p | 0xae10 |

| Heapspeicher | | | | |
|--------------|--|--|--|--|
| 0xae10 | | | | |

| addNumbers(value, pointer) | | |
|----------------------------|----------------|--------|
| 0x1fb1 | pointer x | 0x012a |
| 0x1fa8 | pointer y | 0x0126 |
| 0x1fa0 | pointer result | 0xae10 |

```
void addNumbers(int *x, int *y, int *result){
    *result = *x + *y;
}

int main(){
    int a = 2, b = 3;
    int *p = (int*) malloc(sizeof(int));

    addNumbers(&a, &b, p);
    return 0;
}
```

Die Werte von a und b, auf die die Pointer x und y zeigen, werden vom Stack der main-Funktion dereferenziert. Das Ergebnis der Addition wird in den Heapspeicher geschrieben, auf den der Pointer *result* zeigt

Pointer können von Funktionen auch zurückgegeben werden

- Das ist dann sinnvoll, wenn innerhalb der Funktion Speicherplatz auf dem Heap reserviert wird und die Heap-Adresse dann an die main-Funktion (oder eine andere Funktion) zurückgegeben wird
- Die Adresse des Heapspeichers wird in der aufrufenden Funktion, z.B. der main-Funktion, in einen Pointer geschrieben

Pointer auf Heapspeicher als Rückgabewert

Im Stackframe der main-Funktion werden die Variablen a und b deklariert und initialisiert. Außerdem wird ein Null-Pointer angelegt, für den Rückgabewert der Funktion addNumbers.

| main() | | |
|--------|----|------|
| 0x012a | a | 2 |
| 0x0126 | b | 3 |
| 0x011d | ip | NULL |

Die Funktion liest die Adressen der Variablen a und b ein und gibt die Adresse des Heapspeichers an den Pointer ip zurück

```
void addNumbers(int *x, int *y){
    int *result = (int*)
    malloc(sizeof(int));
    *result = *x + *y;
    return result;
}

int main(){
    int a = 2, b = 3;
    int *ip = NULL;

    ip = addNumbers2(&a, &b);
    return 0;
}
```

Pointer auf Heapspeicher als Rückgabewert

Innerhalb der Funktion wird Speicher auf dem Heap für eine Integervariable reserviert, auf den der Pointer *result* zeigt.

| Heapspeicher | | | | |
|--------------|--|--|--|--|
| 0xae10 | | | | |

| addNumbers(value, pointer) | | |
|----------------------------|----------------|--------|
| 0x1fb1 | pointer x | 0x012a |
| 0x1fa8 | pointer y | 0x0126 |
| 0x1fa0 | pointer result | 0xae10 |

| main() | | |
|--------|----|------|
| 0x012a | a | 2 |
| 0x0126 | b | 3 |
| 0x011d | ip | NULL |

```
void addNumbers(int *x, int *y){
    int *result = (int*) malloc(sizeof(int));
    *result = *x + *y;
    return result;
}

int main(){
    int a = 2, b = 3;
    int *ip = NULL;

    ip = addNumbers2(&a, &b);
    return 0;
}
```

Die Werte von a und b, auf die die Pointer x und y zeigen, werden vom Stack der main-Funktion dereferenziert. Das Ergebnis der Addition wird in den Heapspeicher geschrieben, auf den der Pointer *result* zeigt. Die Adresse des Heapspeichers wird dann über den Pointer *result* an den Pointer ip in main() zurückgegeben.

Heapspeicher freigeben

- Da Heapspeicher manuell reserviert wird, muss er auch manuell wieder freigegeben werden
- Freigeben bedeutet hier, dass der Speicher nicht mehr reserviert ist. Die Werte werden aber nicht notwendigerweise direkt gelöscht oder überschrieben
- Tatsächlich wird der Heapspeicher auch ohne Freigabe durch den/die Programmierer:in meistens vom Betriebssystem nach Beendigung des Programms freigegeben. Das ist aber nicht garantiert
- Syntax:

```
void free (void *p); // Deklaration
```

- Verwendung:

```
p = (int *)malloc(sizeof(int)); // Speicherallokation  
*p = 99; // Wertzuweisung  
free(p); // Speicherfreigabe
```

Heapspeicher allokieren und freigeben

Was Sie sich merken sollten

- Heapspeicher kann mittels malloc() reserviert werden
`pointer = (datentyp *) malloc(sizeof(datentyp));`
`pointer = (datentyp *) malloc(sizeof(*pointer));`
- Ist die Speicherreservierung **nicht** erfolgreich, so gibt malloc() **NULL** zurück
- Der reservierte Speicher muss nach der Verwendung freigegeben werden, da es sonst zu Speicherlecks (memory leaks) kommen kann
`free(pointer);`
- Wird aus Versehen Speicher freigegeben, der nicht zuvor mit malloc() reserviert wurde, so kann das unabsehbare Folgen haben

Übung:

Skizzieren Sie ein kurzes Programm, in dem Sie:

- mittels malloc() Platz für vier float-Variablen reservieren
- Abfragen, ob die Speicherreservierung erfolgreich war
- Im Falle der erfolgreichen Speicherreservierung die reservierten Plätze mit Werten belegen
- Die Werte im Speicher mittels printf ausgeben
- Und den reservierten Speicher wieder freigeben

Dynamische Arrays

Heapspeicher allokieren & reallokieren

- Bisher haben wir Arrays immer statisch angelegt, also ein Array mit einer bestimmten Größe deklariert, die dann zur Laufzeit nicht mehr geändert werden konnte
- Dadurch, dass wir jetzt manuell Speicher auf dem Heap reservieren können, können wir "dynamische Arrays" anlegen, also Arrays, deren Größe zur Laufzeit angepasst werden kann
- Dynamische Arrays sind also kein neuer Datentyp, der eine dynamische Speichernutzung unterstützt, sondern Arrays, deren Speicher manuell verwaltet wird
- Insbesondere vermischt das Konzept von dynamischen Arrays in C Arrays mit Pointern, da Arrayindexierung und Pointerarithmetik gleichwertig sind

```
int main(){

    int *values = NULL;
    int numElements = 0, i = 0;

    printf("\nAnzahl der Werte, die \
eingegeben werden sollen: \n");
    scanf("%d", &numElements);

    values = malloc(numElements * sizeof(int));

    if(!values){
        printf("\nFehler bei der \
Speicherreservierung.\n");
        return 1;
    }

    while(i < numElements){
        printf("Bitte %d. Wert eingeben: \n" \
i+1);
        scanf("%d", &values[i]);
        i++;
    }
}
```

Was, wenn für das dynamische Array zur Laufzeit Speicherplatz reserviert wurde, dieser aber nicht ausreicht?

- Dann kann mit der realloc-Funktion der Speicher vergrößert (oder verkleinert) werden

`void *realloc(void *pointer, size_t neueGroesse);` // pointer zeigt auf die Anfangsadresse des bereits allokierten Speichers

- Beispiel:

```
...  
pointer = (int *) realloc(pointer, 10*sizeof(int));
```

- Wird der Speicher vergrößert, so wird der neue Teil an den alten angehängt
- Wird der Speicher verkleinert, so wird der hintere Teil des Speicherblocks freigegeben

Eine weitere Möglichkeit Speicher zu allokieren ist mit der Funktion calloc()

- Diese funktioniert ähnlich wie malloc(), mit zwei Unterschieden:
 - Bei calloc() kann die Anzahl an Speicherobjekten als Parameter übergeben werden
 - Alle Werte des allokierten Speichers werden automatisch mit 0 initialisiert
- Syntax:

```
void *calloc(size_t anzahl, size_t groesse); // reserviert Heapspeicher und initialisiert mit 0
```

- Beispiel:

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

Mit malloc() würde das folgendermaßen aussehen:
pointer = (int *) malloc(100 * sizeof(int));
memset(pointer, 0, 100 * sizeof(int));

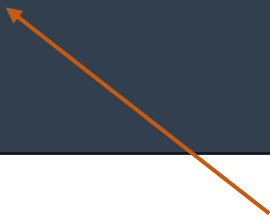
- Pointer sind Variablen, die keine Werte sondern Adressen speichern
 - Mittels Pointern können Werte Call-by-Reference an Funktionen übergeben werden
 - Der Arrayname zeigt auf die Anfangsadresse des Arrays, daher sind Pointerarithmetik und Array-Indexierung gleichwertig, Pointer und Arrays aber nicht
 - Strings sind spezielle Arten von Character-Arrays und zeichnen sich durch das String-Ende-Zeichen am Ende der Zeichenkette aus
- Es gibt verschiedene Speicherkonzepte in C
 - Defaultmäßig werden Funktionsdaten auf dem Stack gespeichert
 - Nachdem eine Funktion beendet wurde wird der dazugehörige Stackframe freigegeben
 - Der Heapspeicher ist dann das Mittel der Wahl, wenn die Lebensdauer der Daten selbstverwaltet werden soll sowie bei dynamischen Arrays und Daten mit hohem Speichervolumen
 - Heapspeicher kann mittels malloc() bzw. calloc() allokiert und mittels free() freigegeben werden
 - Der allokierte Heapspeicher wird über einen Pointer adressiert, der auf die Anfangsadresse des Speichers zeigt

Fortgeschrittene Pointerkonzepte

Pointer auf Pointer & Funktionspointer

- Für Variablen haben wir bereits verschiedene Spezifizierer kennengelernt
- Das Schlüsselwort *restrict* ist eine Möglichkeit eine Pointer-Variable zu qualifizieren
`int* restrict p = (int *) malloc(sizeof(int));`
- **Funktion:**
Das Schlüsselwort *restrict* zeigt an, dass auf einen Speicherbereich nur durch einen einzigen Pointer zugegriffen werden darf. Andere Pointer dürfen den Inhalt weder lesen noch verändern
- Das *restrict* Schlüsselwort ermöglicht es dem Compiler Optimierungen des Maschinencodes vorzunehmen, die Einhaltung von *restrict* kann jedoch nicht überprüft werden

```
int main(){  
    int* restrict p1 = (int*)malloc(sizeof(int));  
    int* p2 = NULL;  
  
    p2 = p1;  
    *p2 = 23;  
  
}
```



Das *restrict*-Schlüsselwort zeigt an, dass das unerwünschte Verhalten ist, da *p2* auf den Speicherplatz zugreift, auf den *p1* exklusiv verweisen sollte. Der Compiler kann die Einhaltung von *restrict* jedoch nicht überprüfen, daher wird keine Warnung / Fehler ausgegeben. Hier ist *restrict* also als eine Art Hinweis an den Programmierer zu verstehen

Restrict-Pointer werden häufig in Funktionen der Standardbibliothek, z.B. string.h, verwendet

Beispiel strncpy():

- Die Funktion strncpy kopiert einen String oder Teil eines Strings (src: source) in einen anderen String (dest: destination)
- Beide Strings sind als restrict-Pointer in der Parameterliste deklariert
- Das verhindert, dass zweimal derselbe String an die Funktion übergeben wird (was zu undefiniertem Verhalten führen würde)
- Der src-String ist außerdem als Konstante deklariert, so dass er durch die Funktion strncpy nicht verändert werden kann
- Wird zweimal derselbe String übergeben, wirft der Compiler einen Fehler

cppreference.com [Log in](#)

Page Discussion View Edit

C Strings library Null-terminated byte strings

strncpy, strncpy_s

Defined in header <string.h>

```
char *strncpy( char *dest, const char *src, size_t count );  
char *strncpy( char *restrict dest, const char *restrict src, size_t count );  
errno_t strncpy_s( char *restrict dest, rsize_t destsz,  
                  const char *restrict src, rsize_t count );
```

(1) (until C99) (since C99)
(2) (since C11)

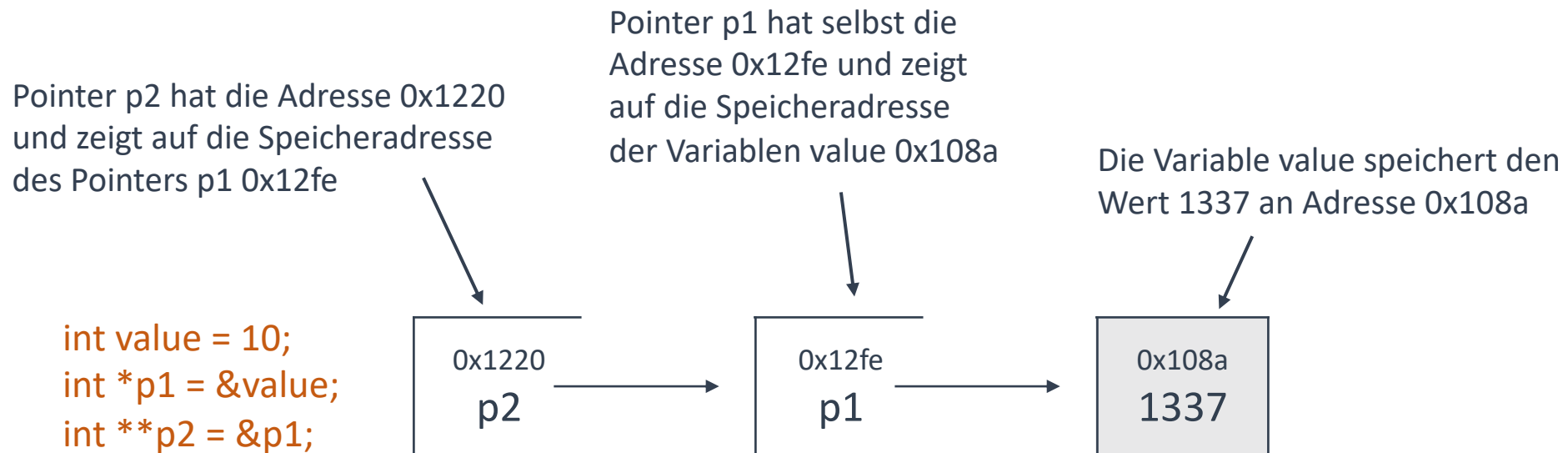
Pointer auf Pointer

POINTERS IN C PROGRAMMING



Was bedeutet Pointer auf Pointer?

Ein Pointer zeigt auf die Adresse eines Pointers, der auf die Adresse eines Werts zeigt



| Pointer (inklusive Pointer auf Pointer) | |
|---|---|
| <code>int *p = NULL;</code> | Deklaration mit Initialisierung auf NULL |
| <code>p = &i;</code> | Initialisierung |
| <code>*p;</code> | Dereferenzierung (Wert von i) |
| <code>p;</code> | Adresse, auf die p zeigt |
| <code>&p;</code> | Adresse von p |
| <code>int **p2 = NULL;</code> | Deklaration Pointer auf Pointer |
| <code>p2 = &p;</code> | Initialisierung |
| <code>**p2 = value;</code> | Dereferenzierung des Werts auf den p2 über p1 zeigt |

Übung:

Skizzieren Sie ein kurzes Programm, in dem Sie:

- eine Variable, einen Pointer und einen PointerPointer deklarieren
- den Pointer auf den Wert weisen lassen und den PointerPointer auf den Pointer
- Ändern Sie dann den Wert der Variable über den PointerPointer

- Wozu braucht man Pointer auf Pointer?
- Das Anwendungsgebiet von Pointern auf Pointer ist z.B. das dynamische Erstellen von Matrizen (also dynamische Arrays mit zwei Dimensionen)
- Um das dynamische Erstellen zweidimensionaler Arrays zu verstehen, sind zwei schon bekannte Konzepte notwendig:
 - Pointerarithmetik und Arrayindexierung sind gleichwertig
 - Speicherplatz wird mittels malloc() dynamisch reserviert

```
int matrix[3][5] = {{1, 2, 3, 4, 5},
                    {6, 7, 8, 9, 1},
                    {0, 0, 0, 0, 0}};
```

| | | | | |
|----------|----------|----------|----------|----------|
| 1 [0][0] | 2 [0][1] | 3 [0][2] | 4 [0][3] | 5 [0][4] |
| 6 [1][0] | 7 [1][1] | 8 [1][2] | 9 [1][3] | 1 [1][4] |
| 0 [2][0] | 0 [2][1] | 0 [2][2] | 0 [2][3] | 0 [2][4] |

Datentyp Name[Reihen][Spalten] = {{ ,
,
 };

für jede Reihe ein Paar
geschweifte Klammern

Beim Thema Arrays hatten wir gelernt, dass der Name eines Arrays ein Pointer auf die Speicheradresse des ersten Elements ist

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

↓
Pointer auf die Anfangsadresse

| | | | | |
|-------|-------|-------|-------|-------|
| 1 [0] | 2 [1] | 3 [2] | 4 [3] | 5 [4] |
|-------|-------|-------|-------|-------|

Wird ein eindimensionales, dynamisches Array mit `malloc()` angelegt, so wird ein Speicherbereich reserviert, auf dessen erstes Element ein Pointer zeigt

```
int *values;
```

```
values = malloc(5 * sizeof(int));
```

Pointer auf die Anfangsadresse



Erinnerung:

Pointerarithmetik und Arrayindexierung sind gleichwertig, Pointer & Arrays jedoch nicht: Auch wenn vieles gleich aussieht, im Hintergrund werden beide unterschiedlich vom Compiler verarbeitet

```
int main(){

    int *values = NULL;
    int numElements = 0, i = 0;

    printf("\nAnzahl der Werte, die \
eingegeben werden sollen: \n");
    scanf("%d", &numElements);

    values = malloc(numElements * sizeof(int));

    if(!values){
        printf("\nFehler bei der \
Speicherreservierung.\n");
        return 1;
    }

    while(i < numElements){
        printf("Bitte %d. Wert eingeben: \n" \
i+1);
        scanf("%d", &values[i]);
        i++;
    }
}
```

Pointer auf Pointer

Zweidimensionale dynamische Arrays

Ein dynamisches, zweidimensionales Array besteht aus:

- mehreren eindimensionalen Arrays
- die über Pointer adressiert werden
- und einem PointerPointer, der quasi die Hauptadresse ist

| | | | | |
|----------|----------|----------|----------|----------|
| 1 [0][0] | 2 [0][1] | 3 [0][2] | 4 [0][3] | 5 [0][4] |
| 6 [1][0] | 7 [1][1] | 8 [1][2] | 9 [1][3] | 1 [1][4] |
| 0 [2][0] | 0 [2][1] | 0 [2][2] | 0 [2][3] | 0 [2][4] |

Achtung:

Im Sinne der Implementierung ist die Beschreibung falsch herum: Es wird zuerst der PointerPointer, dann das Pointer-Array und dann die Integer-Arrays angelegt!

Pointer auf Pointer Zweidimensionale dynamische Arrays

| | | | | |
|----------|----------|----------|----------|----------|
| 1 [0][0] | 2 [0][1] | 3 [0][2] | 4 [0][3] | 5 [0][4] |
| 6 [1][0] | 7 [1][1] | 8 [1][2] | 9 [1][3] | 1 [1][4] |
| 0 [2][0] | 0 [2][1] | 0 [2][2] | 0 [2][3] | 0 [2][4] |

Ein dynamisches, zweidimensionales Array besteht aus:

- mehreren eindimensionalen Arrays
- die über Pointer adressiert werden
- und einem PointerPointer, der quasi die Hauptadresse ist

`column1 = malloc(numColumns * sizeof(int));`

| | | | | |
|--------|--------|--------|--------|--------|
| [0][0] | [0][1] | [0][2] | [0][3] | [0][4] |
| [1][0] | [1][1] | [1][2] | [1][3] | [1][4] |
| [2][0] | [2][1] | [2][2] | [2][3] | [2][4] |

Für jede Reihe im zweidimensionalen Array wird mittels `malloc()` Speicherplatz für die Anzahl an Spalten reserviert. Im Beispiel gibt es also drei eindimensionale Integer-Arrays, mit jeweils fünf Spalten (Elementen)

Achtung:

Im Sinne der Implementierung ist die Beschreibung falsch herum: Es wird zuerst der PointerPointer, dann das Pointer-Array und dann die Integer-Arrays angelegt!

Pointer auf Pointer Zweidimensionale dynamische Arrays

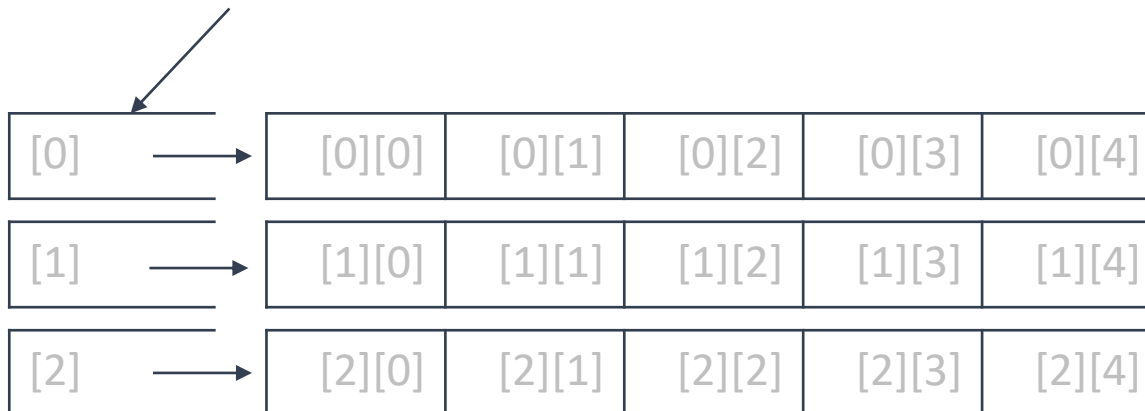
| | | | | |
|----------|----------|----------|----------|----------|
| 1 [0][0] | 2 [0][1] | 3 [0][2] | 4 [0][3] | 5 [0][4] |
| 6 [1][0] | 7 [1][1] | 8 [1][2] | 9 [1][3] | 1 [1][4] |
| 0 [2][0] | 0 [2][1] | 0 [2][2] | 0 [2][3] | 0 [2][4] |

Ein dynamisches, zweidimensionales Array besteht aus:

- mehreren eindimensionalen Arrays
- die über Pointer adressiert werden
- und einem PointerPointer, der quasi die Hauptadresse ist

Größe eines Integerpointers

`matrix = malloc(numRows * sizeof(int *));`



Die einzelnen Integer-Arrays werden über Pointer adressiert. Deshalb gibt es ein weiteres Array:

- die einzelnen Arrayelemente haben die Größe von Integerpointern (z.B. 8 Byte)
- es gibt so viele Elemente, wie es Reihen (Integerarrays) gibt

Achtung:

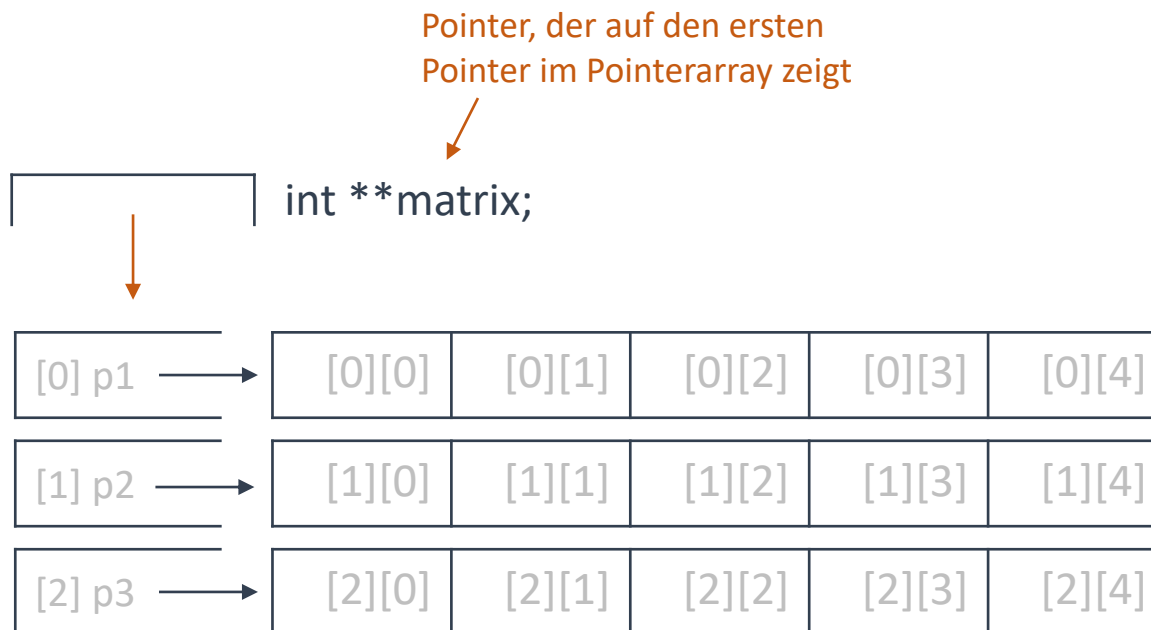
Im Sinne der Implementierung ist die Beschreibung falsch herum: Es wird zuerst der PointerPointer, dann das Pointer-Array und dann die Integer-Arrays angelegt!

Pointer auf Pointer Zweidimensionale dynamische Arrays

| | | | | |
|----------|----------|----------|----------|----------|
| 1 [0][0] | 2 [0][1] | 3 [0][2] | 4 [0][3] | 5 [0][4] |
| 6 [1][0] | 7 [1][1] | 8 [1][2] | 9 [1][3] | 1 [1][4] |
| 0 [2][0] | 0 [2][1] | 0 [2][2] | 0 [2][3] | 0 [2][4] |

Ein dynamisches, zweidimensionales Array besteht aus:

- mehreren eindimensionalen Arrays
- die über Pointer adressiert werden
- und einem PointerPointer, der quasi die Hauptadresse ist



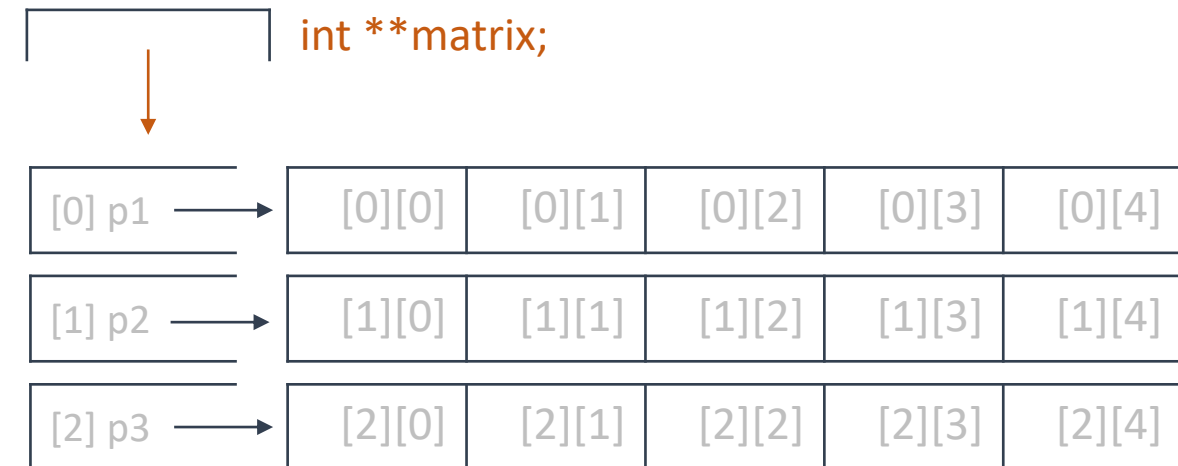
Jetzt gibt es mehrere Integer-Pointer, die jeweils auf die Anfangsadresse eines eindimensionalen Integer-Arrays zeigen. Um dieses Konstrukt als ein einziges zweidimensionales Array ansprechen zu können, wird ein PointerPointer verwendet, der auf das erste Element im Pointer-Array zeigt

Achtung:

Im Sinne der Implementierung ist die Beschreibung falsch herum: Es wird zuerst der PointerPointer, dann das Pointer-Array und dann die Integer-Arrays angelegt!

Die Implementierung eines zweidimensionalen, dynamischen Arrays ist wie folgt:

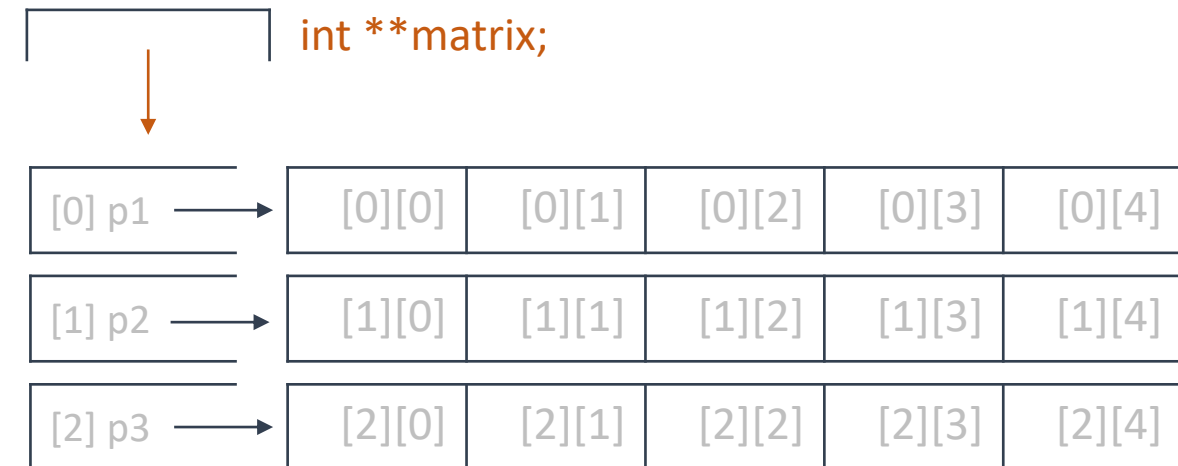
- Pointer auf Pointer
- Pointerarray
- Integerarrays



Die Implementierung eines zweidimensionalen, dynamischen Arrays ist wie folgt:

- Pointer auf Pointer
- Pointerarray
- Integerarrays

```
int **matrix; // Pointer auf pointer
```

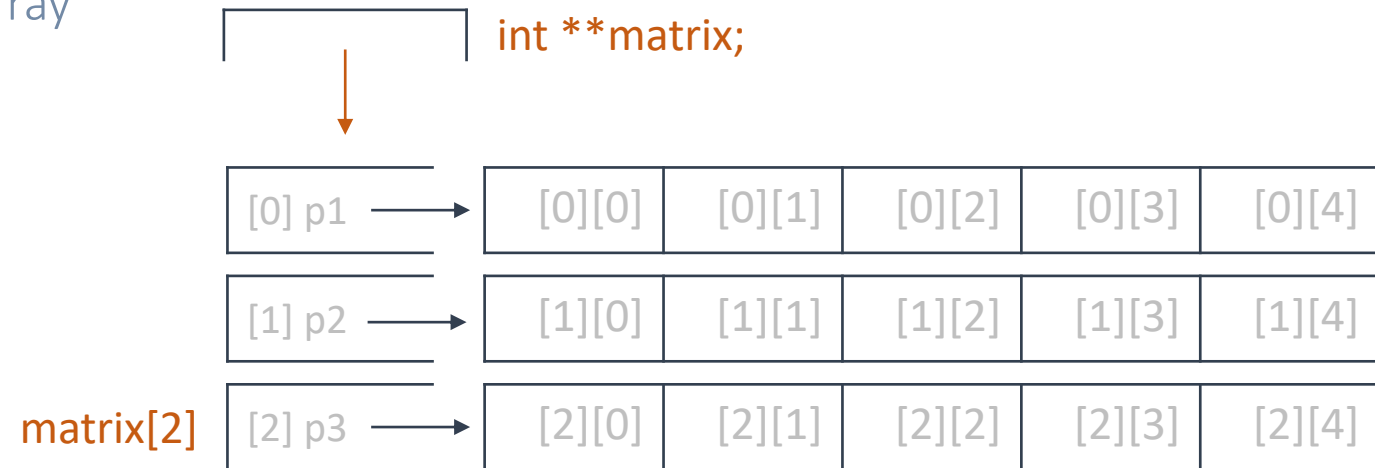


Die Implementierung eines zweidimensionalen, dynamischen Arrays ist wie folgt:

- Pointer auf Pointer
- Pointerarray
- Integerarrays

```
int **matrix; // Pointer auf pointer
```

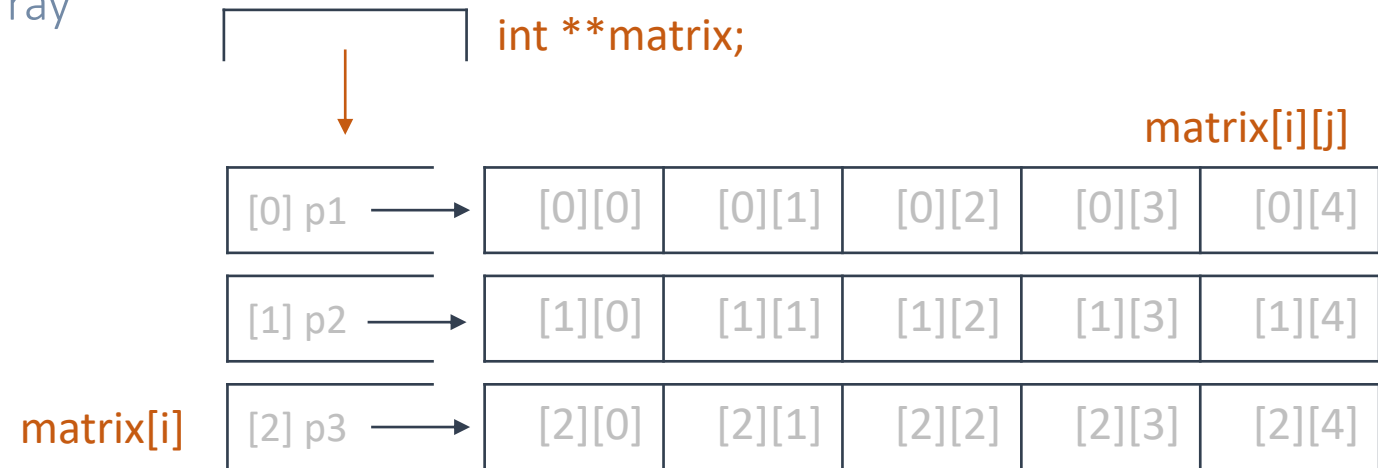
```
matrix = malloc(ROWS * sizeof(int *)); // Pointerarray
```



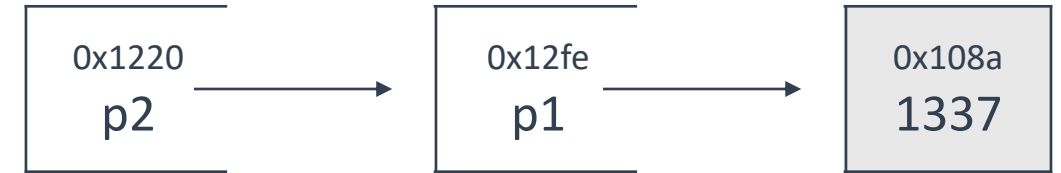
Die Implementierung eines zweidimensionalen, dynamischen Arrays ist wie folgt:

- Pointer auf Pointer
- Pointerarray
- Integerarrays

```
int **matrix; // Pointer auf pointer  
matrix = malloc(ROWS * sizeof(int *)); // Pointerarray  
// Integerarrays  
for (int i = 0; i < COLUMNS; i++) {  
    matrix[i] = malloc(COLUMNS * sizeof(int));  
}
```



- Pointer auf Pointer:
Ein Pointer zeigt auf die Adresse eines anderen Pointers
der wiederum auf die Adresse einer Variablen zeigt
- Zweidimensionale Arrays, deren Speicherplatz zur Laufzeit
änderbar sein soll (= dynamische Arrays), können mittels Pointern auf Pointer implementiert werden. Schön ist
das aber nicht.
Bessere Lösung: Structs, verkettete Listen
- Pointer auf Pointer fallen unter das Konzept: gut, schon mal davon gehört zu haben aber wenig sinnvolle
Einsatzmöglichkeiten



| Pointer (inklusive Pointer auf Pointer) | | | |
|--|---|-------------------|--------------|
| int *p = NULL; | Deklaration mit Initialisierung auf NULL | | |
| p = &i; | Initialisierung | | |
| *p; | Dereferenzierung (Wert von i) | | |
| p; | Adresse, auf die p zeigt | | |
| &p; | Adresse von p | | |
| int **p2 = NULL; | Deklaration Pointer auf Pointer | | |
| p2 = &p; | Initialisierung | | |
| **p2 = value; | Dereferenzierung des Werts auf den p2 über p1 zeigt | | |
| Zugriff auf zweidimensionale Arrays (Pointerarithmetik vs. Arrayindexierung) | | | |
| Zugriff auf | Option 1 | Option 2 | Option 3 |
| 1. Zeile, 1. Spalte | **matrix | *matrix[0] | matrix[0][0] |
| i _{te} Zeile, 1. Spalte | ** (matrix + i) | *matrix[i] | matrix[i][0] |
| 1. Zeile, j _{te} Spalte | * (*matrix + j) | * (matrix[0] + j) | matrix[0][j] |
| i _{te} Zeile, j _{te} Spalte | * (* (matrix + i) + j) | * (matrix[i] + j) | matrix[i][j] |

| | | | | |
|--------|--------|--------|--------|--------|
| [i][j] | [0][1] | [0][2] | [0][3] | [0][4] |
| [1][0] | [1][1] | [1][2] | [1][3] | [1][4] |
| [2][0] | [2][1] | [2][2] | [2][3] | [2][4] |

- Pointer können nicht nur auf Werte und andere Pointer (Adressen) zeigen, sondern auch auf die Anfangsadresse von Funktionscode
- Das heißt, Funktionspointer sind Pointer, die auf Funktionen zeigen
- Das ist deshalb praktisch, weil man sich z.b. eine Liste an Funktionen anlegen und diese recht einfach adressieren kann

Rückgabety
der Funktion

Parameterliste
der Funktion

```
int main(){  
    int (* ptr) (const char *, ...);  
  
    ptr = printf;  
    (* ptr) ("Informatik ist super.\n");  
  
    return 0;  
}
```

(* ptr)

Deklariert einen Pointer auf eine Funktion. Die Funktion kann an sich beliebig sein, muss aber die festgelegten Rahmenbedingungen einhalten, also:

- Rückgabety der Funktion: Integer
- Parameterliste 1. Platz: String

(const char *, ...)

Die drei Punkte zeigen optionale Parameter an.
Bsp: printf("blub %d, %c", var, c);

```
166 void    perror(const char *) __cold;  
167 int  printf(const char * __restrict, ...) __printflike(1, 2);  
168 int  putchar(int, FILE *);  
169 int  putchar(int);  
170 int  puts(const char *);  
171 int  remove(const char *);  
172 int  rename (const char *__old, const char *__new);  
173 void  rewind(FILE *);  
174 int  scanf(const char * __restrict, ...) __scanflike(1, 2);
```

Der Pointer ptr zeigt jetzt
auf die Anfangsadresse
der Funktion printf

```
int main(){  
    int (* ptr) (const char *, ...);  
    ptr = printf;  
    (* ptr) ("Informatik ist super.\n");  
    return 0;  
}
```

(* ptr) ersetzt den
Funktionsnamen

Ein sinnvolles Beispiel zu Funktionspointern schauen wir
uns an, wenn es um komplexe Datentypen (structs) geht

- Pointer sind Datentypen, die statt eines Werts eine Adresse beinhalten
- In den meisten Programmiersprachen wurden Pointer wegabstrahiert
- Verwendung finden Pointer insbesondere
 - in der dynamischen Speicherverwaltung
 - zur Festlegung der Variablen-Lebensdauer durch den/die Programmierer:in
 - zum Zugriff auf den Heap-Speicher → Daten mit großem Volumen
- Arrayindexierung und Pointerarithmetik sind gleichwertig, Arrays sind jedoch keine Pointer (auch wenn der Arrayname bei der Übergabe an eine Funktion zu einem Pointer auf die Adresse des ersten Elements zerfällt)

| Adresse | Typ | Name | Wert |
|-------------|-------|------|-------------|
| 0x16ee1ae80 | int | i | 10 |
| 0x16ee1ae7c | float | f | 2.178 |
| 0x16ee1ae78 | int * | p | 0x16ee1ae80 |

| Pointer | | |
|-----------------------------|--|---|
| int *p = NULL; | Deklaration mit Initialisierung auf NULL | |
| p = &i; | Initialisierung | |
| *p; | Dereferenzierung (Wert von i) | |
| p; | Adresse, auf die p zeigt | |
| &p; | Adresse von p | |
| Auf Arrayelemente zugreifen | | |
| *pointer | *array | erstes Element des Arrays |
| pointer[0] | array[0] | |
| *(pointer+n) | *(array+n) | Zugriff auf <i>n</i> tes Element |
| pointer[n] | array[n] | |
| pointer | array | Zugriff auf die Adresse des ersten Elements |
| &pointer[0] | &array[0] | |
| pointer+n | array+n | Zugriff auf die Adresse des <i>n</i> ten Elements |
| &pointer[n] | &array[n] | |

| Heapspeicher allokieren und freigeben | |
|---------------------------------------|--|
| int *p = NULL | Pointerdeklaration |
| p = (int *) malloc(sizeof(int)); | Reserviert Heapspeicher für eine Integer |
| p = (int *) calloc(10, sizeof(int)); | Reserviert Heapspeicher für einen Block von zehn Integer & initialisiert mit 0 |
| p = (int *) realloc(p, newSize); | Vergrößert oder verkleinert den reservierten Heapspeicher |
| free(p) | Gibt den reservierten Heapspeicher frei |

| void-Pointer Typumwandlung | | Beispiel |
|----------------------------|---------------------------------|--------------------------------|
| void *p = NULL; | Deklaration | void *p = NULL; int i = 10; |
| p = (typ *) &i; | Typumwandlung & Initialisierung | p = (int *) &i; |
| *(typ *)p = wert; | Dereferenzierung | *(int *)p = 22; |
| var = *(typ *)p; | Dereferenzierung | int x = *(int *)p; |