

# Programmieren 1 – Informatik Arrays, Strings & Mathematik

Prof. Dr.-Ing. Maike Stern | 8.11.2023

- Kontrollstrukturen
  - If
  - Switch
  - While
  - For
- Funktionen
  - Funktionen mit Parametern und Rückgabewert
  - Geltungsbereich von Funktionen
  - Datentyp-Spezifizierer (global, static, const)
  - Rekursive Funktionen

Die Abbruchbedingung (oder eher Durchlaufbedingung) bei der While/DoWhile- sowie der For-Schleife wird dann aktiv, wenn die Bedingung *false* ist

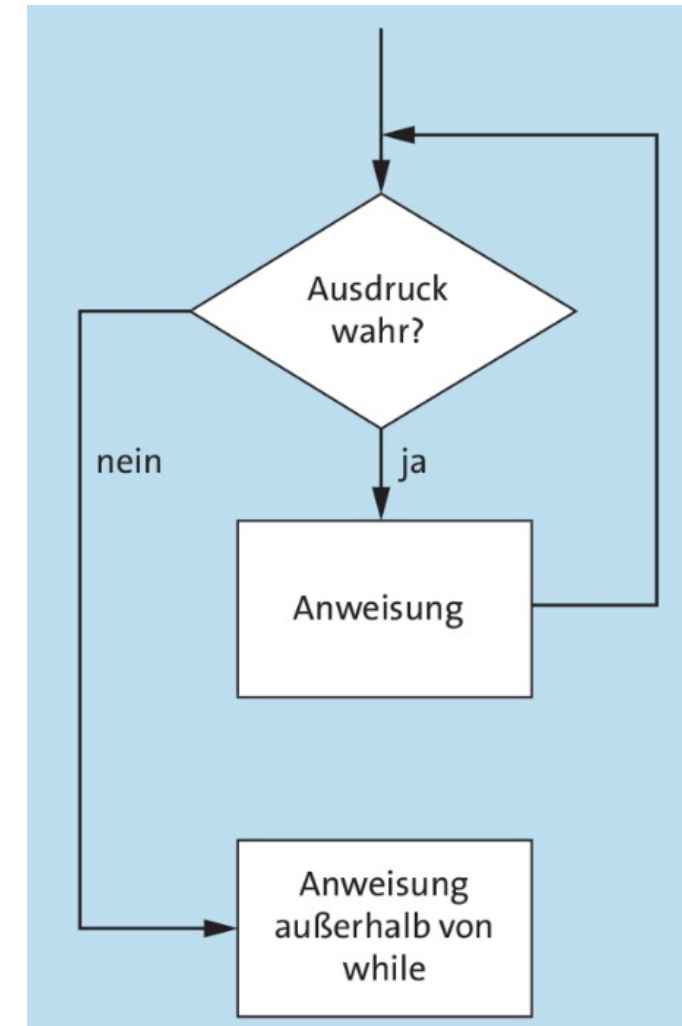
```
while(Bedingung == wahr){  
    // Anweisungen  
}
```

Solange die Bedingung  
wahr ist, wird die Schleife  
ausgeführt

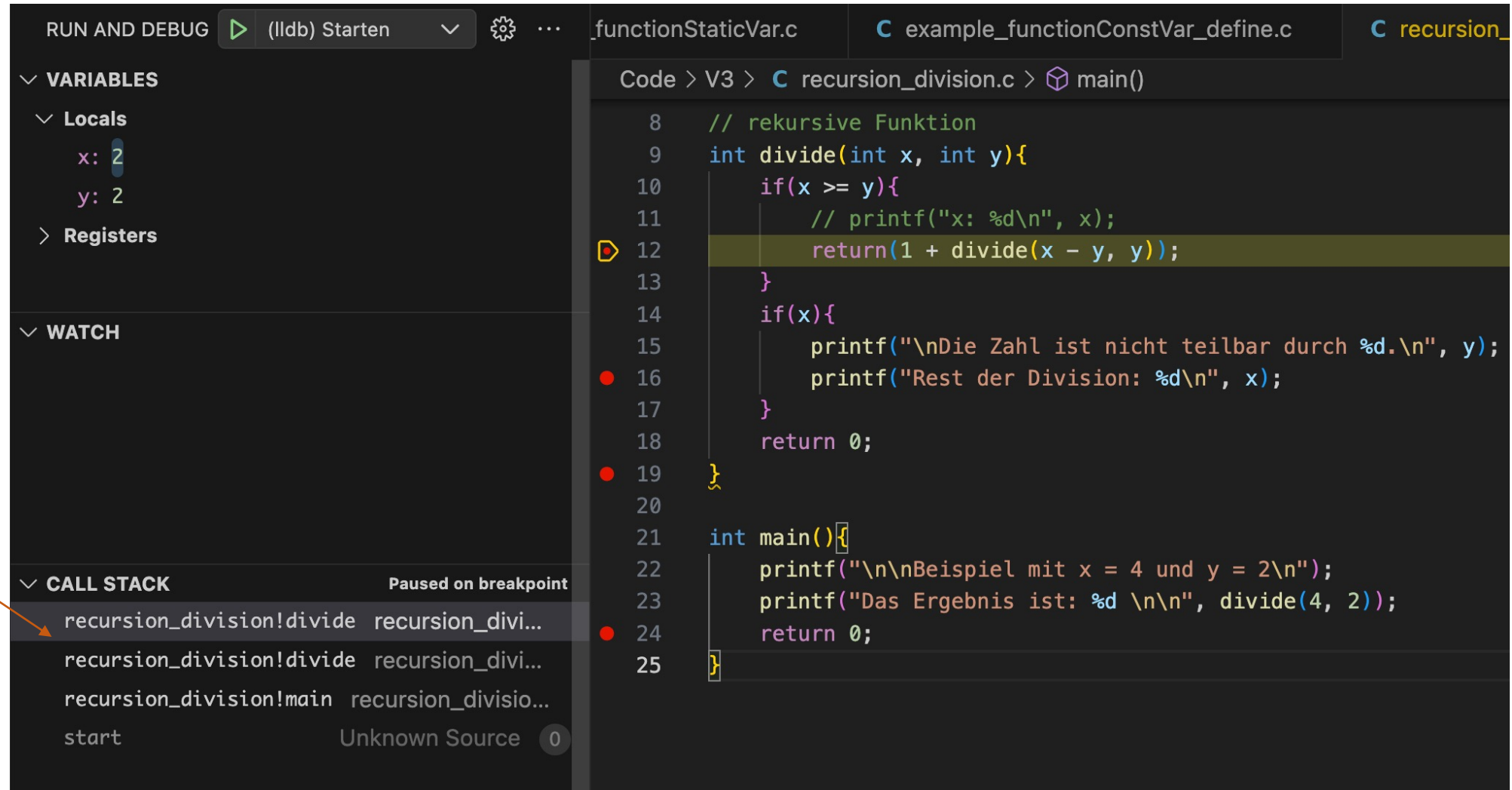
```
#include <stdio.h>
```

```
int main(){  
    for (char c = 'a', c <= 'z', c++){  
        printf("%c\n", c);  
    }  
    return 0;  
}
```

Solange c kleiner oder  
gleich 'z' ist, wird die  
Schleife ausgeführt



In der IDE wird beim Debuggen der Stack angezeigt, erst wie er sich aufbaut, dann wie er rückabgewickelt wird



RUN AND DEBUG (lldb) Starten

**VARIABLES**

- Locals**
  - x: 2
  - y: 2
- Registers**

**WATCH**

**CALL STACK** Paused on breakpoint

- recursion\_division!divide recursion\_divi...
- recursion\_division!divide recursion\_divi...
- recursion\_division!main recursion\_divisio...
- start Unknown Source 0

```
Code > V3 > C recursion_division.c > main()
8 // rekursive Funktion
9 int divide(int x, int y){
10     if(x >= y){
11         // printf("x: %d\n", x);
12         return(1 + divide(x - y, y));
13     }
14     if(x){
15         printf("\nDie Zahl ist nicht teilbar durch %d.\n", y);
16         printf("Rest der Division: %d\n", x);
17     }
18     return 0;
19 }
20
21 int main(){
22     printf("\n\nBeispiel mit x = 4 und y = 2\n");
23     printf("Das Ergebnis ist: %d \n\n", divide(4, 2));
24     return 0;
25 }
```

Die assert-Funktion ist eine Möglichkeit, um die Ausgaben seines Codes zu testen

```
#include <assert.h>
```

```
assert(function() == expectedValue);
```

Beispiel:

```
assert(addNumbers(2,3) == 5);
```

Ist das Ergebnis des Vergleichs nicht true, so wird eine Warnung ausgegeben

- Arrays
- Strings
- Mathematische Funktionen
- Zeiten

# Arrays

- Bisher haben wir Datentypen kennengelernt, die einen Wert speichern (Integer, Float, Double, Char)  
`float f = 2.178;`
- In vielen Anwendungen werden aber mehrdimensionale Variablen benötigt, also Matrizen
- Matrizen sind insbesondere in Simulationen aber auch Algorithmen der künstlichen Intelligenz von Bedeutung
- Die Programmiersprache Matlab, die für Modellierungen und Simulationen eingesetzt wird, ist Matrix-basiert  
Daher der Name: Matrix Laboratory
- Die meisten KI-Algorithmen sind im Kern Matrix-Multiplikationen, z.B. Regression, Neuronale Netze
- Aber auch in Standardanwendungen gibt es häufig Variablen, die mehrere Werte umfassen und logisch zusammengehören
- Der Datentyp, der Matrizen nachbildet, heißt Array



- Bisher haben wir Datentypen kennengelernt, die einen Wert speichern (Integer, Float, Double, Char)  
`float f = 2.178;`
- In vielen Anwendungen werden aber mehrdimensionale Variablen benötigt, also Matrizen
- Matrizen sind insbesondere in Simulationen aber auch Algorithmen der künstlichen Intelligenz von Bedeutung
- Die Programmiersprache Matlab, die für Modellierungen und Simulationen eingesetzt wird, ist Matrix-basiert  
Daher der Name: Matrix Laboratory
- Die meisten KI-Algorithmen sind im Kern Matrix-Multiplikationen, z.B. Regression, Neuronale Netze
- Aber auch in Standardanwendungen gibt es häufig Variablen, die mehrere Werte umfassen und logisch zusammengehören
- Der Datentyp, der Matrizen nachbildet, heißt Array

## Syntax

Datentyp Arrayname [Anzahl\_der\_Elemente];

Also:

`int i [5]; // eindimensional`

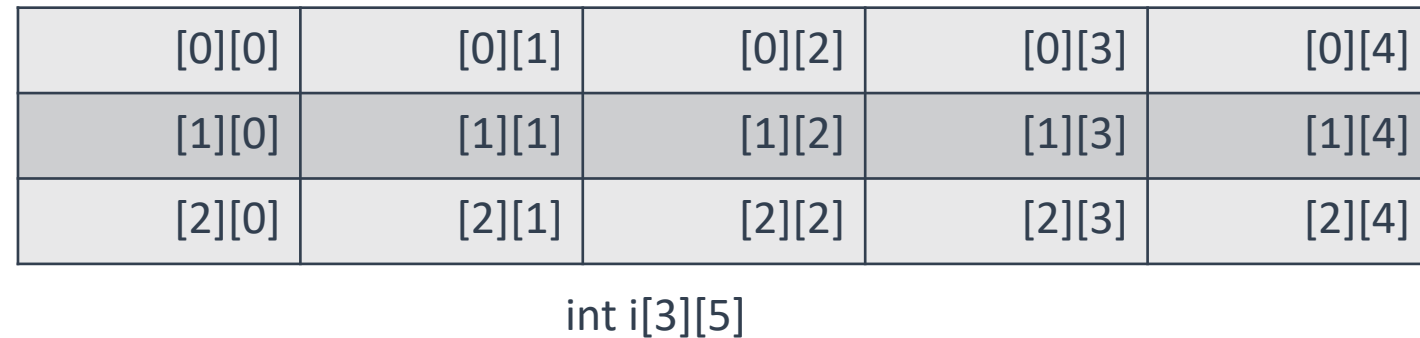
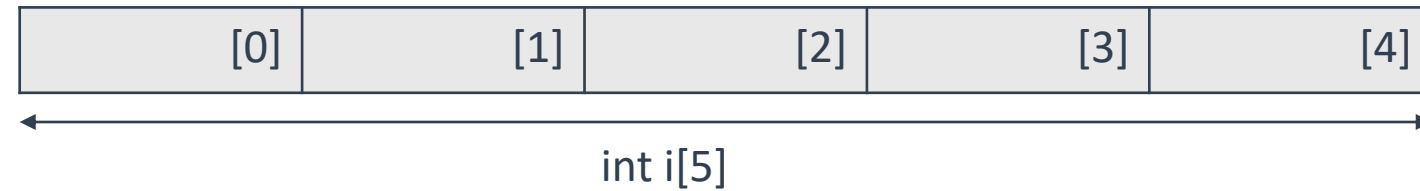
Spalten

`int i [3][5]; // zweidimensional`

Reihe

Spalten

Speicherplatz für eine  
Integervariable mit 4 Byte



## Syntax

Datentyp Arrayname [Anzahl\_der\_Elemente];

Also:

```
int i [5]; // eindimensional
```

Spalten

```
int i [3][5]; // zweidimensional
```

Reihe

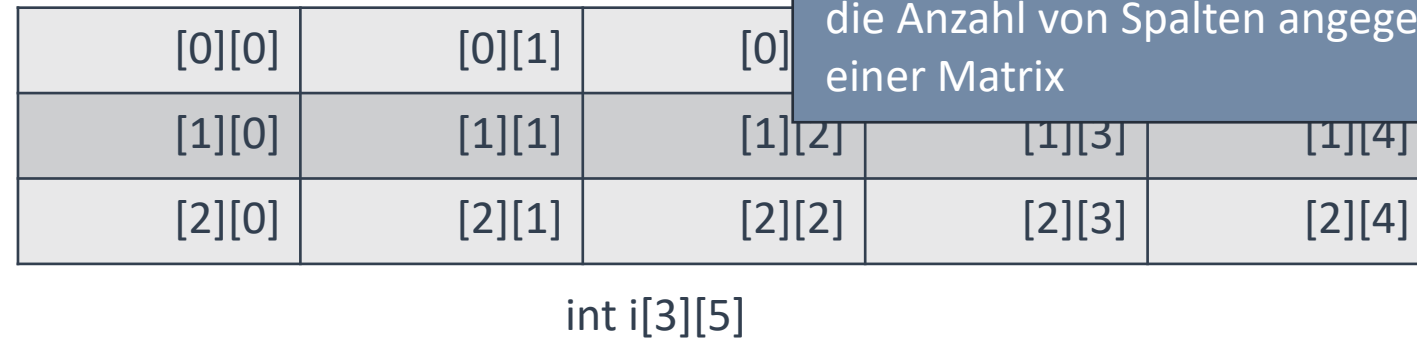
Spalten

Außer in der Programmiersprache Matlab fangen alle Sprachen das Zählen bei 0 an. Entsprechend beginnt der Index eines Arrays mit 5 Einträgen bei 0 und geht bis 4

Eindimensionale Arrays haben mehrere Spalten, wie eine Zahlenreihe



Bei mehrdimensionalen Arrays wird zunächst die Anzahl von Reihen und dann die Anzahl von Spalten angegeben, wie bei einer Matrix



## Syntax

Datentyp Arrayname  
[Anzahl\_der\_Elemente];

Also:

```
int i [5]; // eindimensional
```



Arrays können direkt bei der  
Deklaration initialisiert werden:

```
int i[5] = {234, 423, 857, 9, 0}
```



Wird das Array bei der Deklaration auch initialisiert, berechnet C die Arraygröße selbstständig, also `int i[] = {234, 9} → i[2]`.  
Besserer Stil ist es aber, die Größe dennoch anzugeben

Arrays können immer nur Werte eines Datentyps aufnehmen.

### Syntax

Datentyp Arrayname  
[Anzahl\_der\_Elemente];

Also:

```
int i [5]; // eindimensional
```



Oder einzeln:

```
i[0] = 234;
```

```
i[1] = 423;
```

...

↖  
Dabei werden die einzelnen  
Elemente des Arrays über den  
Index adressiert



### Syntax

Datentyp Arrayname  
[Anzahl\_der\_Elemente];

Also:

```
int i [5]; // eindimensional
```

Oder einzeln:

```
i[0] = 234;
```

```
i[1] = 423;
```

...

↖  
Dabei werden die einzelnen  
Elemente des Arrays über den  
Index adressiert

Aufgabe:

Überlegen Sie sich, wie Sie ein Array mittels  
For-Schleife über die Tastatur einlesen  
können

```
#include <stdio.h>
```

```
int main(){  
    int i[3];
```

```
    for(int j = 0; j <= 3; j++){  
        printf("\nEnter a number: ");  
        scanf("%d", &i[j]);  
    }
```

```
    for(int j = 0; j <= 3; j++){  
        printf("\nThe %d. number is: %d", j+1, i[j]);  
    }  
    printf("\n");  
}
```

Aufgabe:  
Wo ist der Bug?

In C ist ein häufiger und schwierig auffindbarer Fehler das Überschreiten von Array-Grenzen

- Während andere Programmiersprachen einen Fehler ausgeben, wenn über die Array-Grenzen hinaus Werte belegt werden, ist das bei C nicht der Fall

Das Array i hat 3 Elemente

In der For-Schleife werden 4  
Elemente belegt (0, 1, 2, 3)  
Richtig wäre also  $j < 3$

```
...  
int main(){  
    int i[3];  
  
    for(int j = 0; j <= 3; j++){  
        printf("\nEnter a number: ");  
        scanf("%d", &i[j]);  
    }  
}
```



In C ist ein häufiger und schwierig auffindbarer Fehler das Überschreiten von Array-Grenzen

- Während andere Programmiersprachen einen Fehler ausgeben, wenn über die Array-Grenzen hinaus Werte belegt werden, ist das bei C nicht der Fall
- Als Folge wird das vierte Arrayelement (das außerhalb der Grenze liegt) sehr wahrscheinlich mit dem eingegebenen Wert belegt
- ABER: Im Verlauf des Programms ist es ebenso wahrscheinlich, dass der Speicherplatz des vierten Elements mit einem anderen Wert überschrieben wird
- Was genau im Programm passiert, wenn auf ein ungültiges Element zugegriffen wird, hängt vom Compiler ab

```
...  
int main(){  
    int i[3];  
  
    for(int j = 0; j <= 3; j++){  
        printf("\nEnter a number: ");  
        scanf("%d", &i[j]);  
    }  
}
```

Die Deklaration der **Arraygröße** ist einer der Fälle, in denen eine Konstante eine gute Idee sein kann

Per `#define` wird eine Konstante angelegt (siehe 2. Vorlesung)

```
#include <stdio.h>
#define SIZE 3

int main(){
    int i[SIZE];

    for(int j = 0; j < SIZE; j++){
        printf("\nEnter a number: ");
        scanf("%d", &i[j]);
    }
}
```

Die tatsächliche Größe von hier 3 wird dann zur Laufzeit des Programms eingesetzt

Bisher haben wir Möglichkeiten kennengelernt, die Arrayelemente einzeln zu belegen

- Es gibt aber auch die Möglichkeit Arrays automatisch zu belegen, zumindest mit 0

```
int array[100] = {0}; // belegt bei der Initialisierung alle 100 Elemente mit 0
```

Bisher haben wir Möglichkeiten kennengelernt, die Arrayelemente einzeln zu belegen

- Es gibt aber auch die Möglichkeit Arrays automatisch zu belegen, zumindest mit 0

```
int array[100] = {0}; // belegt bei der Initialisierung alle 100 Elemente mit 0
```

- Das funktioniert, weil bei einer teilweisen Belegung von Arrays der Rest mit dem Wert 0 im jeweiligen Datentypen belegt wird, also:

```
float array[3] = {0.0}; // → array[0], array[1], array[2] haben alle den Wert 0.0
```

- Das Ganze funktioniert auch, wenn die ersten Werte ungleich Null sind:

```
float array[3] = {2.7}; // → array[0] = 2.7, array[1] und array[2] haben den Wert 0.0
```

Bisher haben wir Möglichkeiten kennengelernt, die Arrayelemente einzeln zu belegen

- Es gibt aber auch die Möglichkeit Arrays automatisch zu belegen, zumindest mit 0

```
int array[100] = {0}; // belegt bei der Initialisierung alle 100 Elemente mit 0
```

- Das funktioniert, weil bei einer teilweisen Belegung von Arrays der Rest mit dem Wert 0 im jeweiligen Datentypen belegt wird, also:

```
float array[3] = {0.0}; // → array[0], array[1], array[2] haben alle den Wert 0.0
```

- Das Ganze funktioniert auch, wenn die ersten Werte ungleich

```
float array[3] = {2.7}; // → array[0] = 2.7, array[1] und array[2]
```

Wie die automatische Initialisierung *nicht* funktioniert:

```
int array[100]; // Deklaration ohne Initialisierung  
array[2] = 2; // belegt nur das 3. Element mit 2, die  
restlichen Elemente sind undefiniert
```

### Aufgabe:

Schreiben Sie ein kurzes Programm, das ein Array mit 20 Elementen deklariert und mit den Werten 0, 1, 2, 3, 0, 0, ...0 belegt.

Eine zweite Möglichkeit Arrays automatisch zu initialisieren ist die memset-Funktion

```
void *memset(void *adres, int zeichen, size_t n); // Deklaration der memset-Funktion
```

Zugriff auf die Speicheradresse,  
an der das Array gespeichert ist

Zeichen, mit dem das  
Array gefüllt werden soll

Arraygröße

Wenn wir alle Elemente des Arrays mit 0 belegen wollen, geht das wie folgt:

```
int array[10]; // Array-Deklaration
```

```
memset(array, 0, sizeof(array)); // Belegen aller Arrayelemente mit 0
```

Achtung: Memset funktioniert  
nur mit 0 und -1

Ein paar Worte zur Memset-Funktion:

- Memset ist genauso eine Funktion wie printf oder scanf
- Im Gegensatz zu printf/scanf ist memset nicht in der *stdio.h*-Headerdatei sondern in der *string.h*-Headerdatei deklariert, das heißt:
  - Sie müssen memset nicht selbst deklarieren
  - sondern die Headerdatei mittels *#include <string.h>* einbinden
  - und können die memset-Funktion dann einfach verwenden
- Wenn Sie eine IDE zur Programmierung verwenden, zeigt diese Ihnen die Funktionsparameter an



```
...  
int main(){  
    int firstArray[5] = {1, 2, 3, 4, 5};  
    int secondArray[5] = {0, 2, 5, 4, 90};  
  
    ...  
}
```

Aufgabe:  
Überlegen Sie sich, wie Sie die Werte zweier Arrays miteinander vergleichen können.

# Eindimensionale Arrays

## Mit Arrays arbeiten | Werte vergleichen


Aufgabe:  
Wo ist der Bug?

```
...  
int main(){  
    int firstArray[5] = {1, 2, 3, 4, 5};  
    int secondArray[5] = {0, 2, 5, 4, 90};  
  
    for(int i = 0; i < 5; i++){  
        if(firstArray == secondArray){  
            printf("Arrays sind gleich an Position %d\n", i);  
        }  
        else{  
            printf("Die Arrays unterscheiden sich an Position %d\n", i);  
        }  
    }  
}
```

- Um zwei Arrays miteinander zu vergleichen, müssen die einzelnen Werte überprüft werden
- Das ist (insbesondere bei kleineren Arrays) mittels for-Schleife und if-Anweisung möglich

```
...  
int main(){  
    int firstArray[5] = {1, 2, 3, 4, 5};  
    int secondArray[5] = {0, 2, 5, 4, 90};  
  
    for(int i = 0; i < 5; i++){  
        if(firstArray[i] == secondArray[i]){  
            printf("Arrays sind gleich an Position %d\n", i);  
        }  
        else{  
            printf("Die Arrays unterscheiden sich an Position %d\n", i);  
        }  
    }  
}
```

Wichtig: es müssen die einzelnen Elemente verglichen werden!



- Neben einer Kombination von for-Schleife und if-Anweisung kann auch die Memcmp-Funktion verwendet werden, um den Inhalt zweier Arrays zu vergleichen
- Da Funktionen im Hintergrund oft so optimiert sind, dass sie hardware-nah operieren, sind sie in der Regel schneller als for+if → insbesondere bei großen Arrays hilfreich
- Syntax:

```
int memcmp(const void *adr1, const void *adr2, size_t n); // liefert 0 zurück, wenn die Arrays gleich sind
```

↑  
Speicheradresse  
des 1. Arrays

↖  
Speicheradresse  
des 2. Arrays

↖  
Größe des  
Speicherbereichs, der  
verglichen werden soll

```
...  
int main(){  
    int firstArray[5] = {1, 2, 3, 4, 5};  
    int secondArray[5] = {0, 2, 5, 4, 90};  
  
    // compare both arrays  
    if(memcmp(firstArray, secondArray, sizeofArray) == 0){  
        printf("\nBoth arrays are equal\n");  
    }  
}
```

- Neben einer Kombination von for-Schleife und if-Anweisung kann auch die Memcmp-Funktion verwendet werden, um den Inhalt zweier Arrays zu vergleichen
- Da Funktionen im Hintergrund oft so optimiert sind, dass sie hardwarenah operieren, sind sie in der Regel schneller als for+if → insbesondere bei großen Arrays hilfreich
- Syntax:

```
int memcmp(const void *adr1, const void *adr2, size_t n); // liefert 0 zurück, wenn die Arrays gleich sind
```

Die memcmp-Funktion vergleicht zwei Speicherbereiche miteinander, über eine Länge von n, nicht zwei Arrays.

```
...  
int main(){  
    int firstArray[SIZE];  
    int secondArray[SIZE];  
  
    for(int i = 0; i < SIZE; i++){  
        firstArray[i] = i;  
        secondArray[i] = i;  
    }  
  
    int sizeOfArray = sizeof(firstArray);  
  
    if(memcmp(firstArray, secondArray, sizeOfArray) == 0){  
        printf("\nBoth arrays are equal\n");  
    }  
    else{  
        printf("\nThe arrays are not equal\n");  
    }  
}
```

Aufgabe:  
Wie könnte man arrays\_memcmp.c  
verschönern?

- In den vorherigen Beispielen haben wir den sizeof-Operator verwendet, um die Arraygröße für die memcmp-Funktion zu berechnen
- Allerdings berechnet sizeof nicht die Anzahl der Elemente im Array sondern die **Größe einer Datenstruktur in Byte**



- In den vorherigen Beispielen haben wir den sizeof-Operator verwendet, um die Arraygröße für die memcmp-Funktion zu berechnen
- Allerdings berechnet sizeof nicht die Anzahl der Elemente im Array sondern die Größe einer Datenstruktur in Byte

Frage:  
Wie kann man sizeof nutzen, um die Anzahl der Arrayelemente zu berechnen?

- In den vorherigen Beispielen haben wir den sizeof-Operator verwendet, um die Arraygröße für die memcmp-Funktion zu berechnen
- Allerdings berechnet sizeof nicht die Anzahl der Elemente im Array sondern die Größe einer Datenstruktur in Byte
- Die Anzahl der Elemente in einem Array kann errechnet werden, indem man die gesamte Anzahl an Bytes des Arrays durch die Größe eines einzelnen Arrayelements teilt:

```
int numOfElements = sizeof(intArray) / sizeof(int);
```

- In den vorherigen Beispielen haben wir den sizeof-Operator verwendet, um die Arraygröße für die memcmp-Funktion zu berechnen
- Allerdings berechnet sizeof nicht die Anzahl der Elemente im Array sondern die Größe einer Datenstruktur in Byte

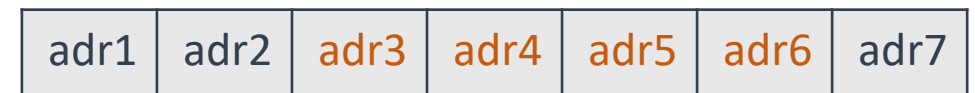
Warum konnten wir dann bei memset und memcmp einfach sizeof(array) verwenden?

- Weil memset und memcmp Funktionen sind, die den übergebenen Speicherbereich bearbeiten & sizeof(array) die Größe des Arrays übergibt

```
void *memset(void *adres, int zeichen, size_t n);
```

↑  
Speicheradresse

↖  
Größe des  
Speicherbereichs, der  
verändert werden soll



↑  
Speicheradresse

Ändere ab  
Speicheradresse  
adr3 vier Byte

Natürlich können auch Arrays als Argument an Funktionen übergeben werden, allerdings funktioniert das bei Arrays etwas anders:

- Rückblick: Funktionsparameter mit primitiven Variablen (Integer, Float, Double, Char, Boolean, ...)

**Call-by-Value:** Die Funktionsparameter werden als Kopie übergeben

- Im Beispiel wird *addNumbers* mit zwei Integervariablen **deklariert**, *a* und *b* (Funktionsparameter)
- Im Programm werden dann die Variablen *x* und *y* an die Funktion **übergeben** (Argumente)
- Da die Übergabe mittels Call-by-Value stattfindet, sind *a* und *b* nur **Kopien** von *x* und *y*. Werden *a* bzw. *b* in der Funktion verändert, so hat das keine Auswirkungen auf *x* bzw. *y*
- Arrays werden jedoch nicht mittels Call-by-Value sondern mittels Call-by-Reference übergeben

```
#include <stdio.h>

int addNumbers(int a, int b);

int main(){

    int x = 3, y = 4;
    int result = 0;

    result = addNumbers(x, y);

    return 0;
}

int addNumbers(int a, int b){
    return a + b;
}
```

Natürlich können auch Arrays als Argument an Funktionen übergeben werden, allerdings funktioniert das bei Arrays etwas anders: **Call-by-Reference** (Referenz = Verweis auf die Speicheradresse)

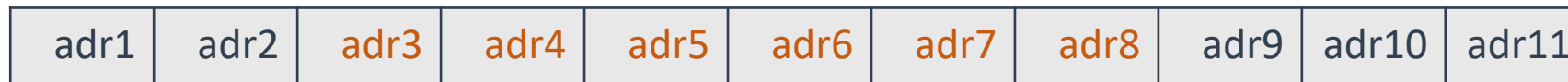
**Funktionsdeklaration** mit Array als Parameter

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



Übergabe des Arrays: das Array zerfällt  
in einen Zeiger → Die Funktion erhält  
die Anfangsadresse des ersten  
Elements im Array

Optional, aber sinnvoll: Anzahl  
der Elemente im Array



Speicheradresse des  
ersten Arrayelements

Arrayelemente: 6


Natürlich können auch Arrays als Argument an Funktionen übergeben werden, allerdings funktioniert das bei Arrays etwas anders: **Call-by-Reference** (Referenz = Verweis auf die Speicheradresse)

**Funktionsdeklaration** mit Array als Parameter

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

**Funktionsaufruf**

```
function(array, SIZE);
```



Array      Arraygröße

```
#include <stdio.h>
#define SIZE 2

void goToDirection(int coordinatesArray[]){

    coordinatesArray[0] += 1;
    coordinatesArray[1] -= 1;
}

int main(){
    int coordinates[SIZE] = {42, 7};

    goToDirection(coordinates);

    printf("\nNow at: %d and %d\n\n", coordinates[0], coordinates[1]);

    return 0;
}
```

Frage:

Welcher Wert wird für die Koordinaten auf der Konsole ausgegeben?

Der Funktionsaufruf mit einem Array als Argument könnte auch folgendermaßen aussehen:

`function(&array[0], SIZE);`

↑  
&array[0] entspricht ebenfalls  
der Anfangsadresse des Arrays

Adressoperator &

- Werden Arrays als Funktionsparameter übergeben, wird immer nur die Speicheradresse übergeben. Daher sind beide Varianten (*function(array, SIZE)* und *function(&array[0], SIZE)*) gleichwertig
- Da Arrays eben immer als Referenz übergeben werden, können sie auch nicht von einer Funktion zurückgegeben werden
- Möchte man aber trotzdem, unbedingt Arrays als Kopie bzw. Rückgabewert übergeben, so kann man sie in eine Struktur verpacken (den Struct-Datentyp werden wir später kennenlernen)

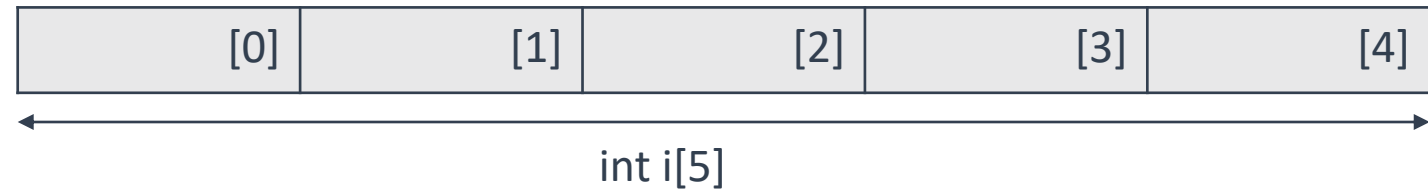


- Wie anfangs erwähnt, können Arrays mehrere Dimensionen annehmen
- Während eindimensionale Arrays in etwa Zahlenreihen entsprechen, entsprechen zweidimensionale Arrays Matrizen

Also:

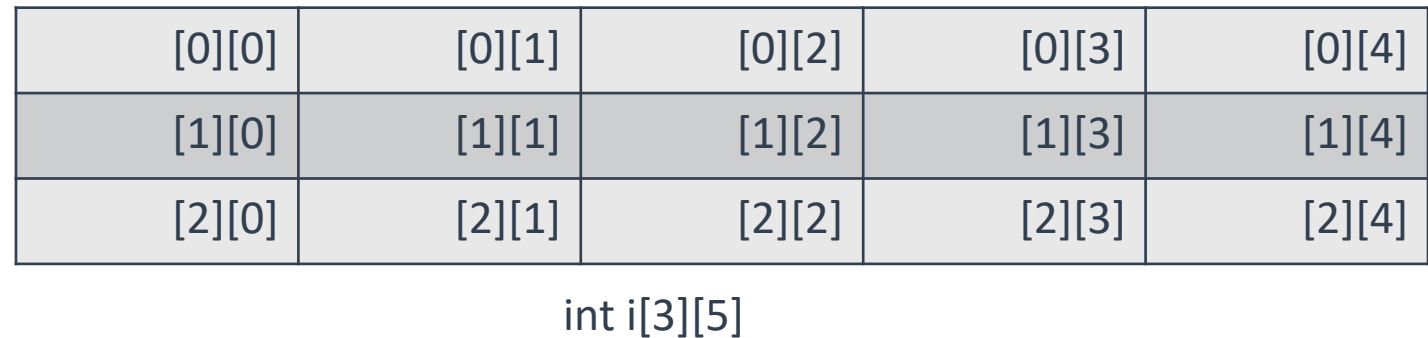
`int i [5]; // eindimensional`

↖  
Spalten



`int i [3][5]; // zweidimensional`

↖ ↖  
Reihe Spalten



Zweidimensionale Arrays können ähnlich wie eindimensionale initialisiert werden

```
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] = {{ [Bar],
                                     { [Bar],
                                     { [Bar] }};
```

für jede Reihe ein Paar  
geschweifte Klammern

Auch die automatische Auffüllung mit 0en funktioniert bei zweidimensionalen Arrays

```
int matrix[3][5] = {{0},  
                    {1},  
                    {0, 1}};
```

0	0	0	0	0
1	0	0	0	0
0	1	0	0	0

Übung:  
Wie lässt sich ein zweidimensionales Array  
mit Nutzereingaben füllen?

Aufgabe:  
Wo ist der Bug?

```
int matrix[ROWS][COLUMNS];

for(int i = 0, i <= ROWS, i++) {
    for(int j = 0, j <= COLUMNS, j++) {
        printf("Wert eingeben für Matrix[%d][%d]: \n", i, j);
        scanf("%d", matrix[i][j]);
    }
}

for(int i = 0, i <= ROWS, i++) {
    for(int j = 0, j <= COLUMNS, j++) {
        printf("Wert Matrix[%d][%d]: %d\n", i, j, matrix[i][j])
    }
}
```

```
int matrix[ROWS][COLUMNS];

for(int i = 0; i < ROWS; i++) {
    for(int j = 0; j < COLUMNS; j++) {
        printf("Wert eingeben für Matrix[%d][%d]: \n", i, j);
        scanf("%d", &matrix[i][j]);
    }
}

for(int i = 0; i < ROWS; i++) {
    for(int j = 0; j < COLUMNS; j++) {
        printf("Wert Matrix[%d][%d]: %d\n", i, j, matrix[i][j]);
    }
}
```

Natürlich können Arrays auch mehr als zwei Dimensionen einnehmen, drei zum Beispiel:

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

Matrizen      Reihen      Spalten

## Begriffsklärung:

Vektor: 1-dimensional

Matrix: 2-dimensional

Tensor: n-dimensional

Tensoren werden z.B. in KI-  
Algorithmen und der Bildverarbeitung  
sehr häufig eingesetzt.

1	2	3	4
[0][0][0]	[0][0][1]	[0][0][2]	[0][0][3]
6	7	8	9
[0][1][0]	[0][1][1]	[0][1][2]	[0][1][3]
0	0	0	0
[0][2][0]	[0][2][1]	[0][2][2]	[0][2][3]

Werden zwei- oder mehrdimensionale Arrays an Funktionen übergeben werden, so muss zwar die Größe der ersten Dimension nicht mit übergeben werden, die restlichen Größen aber schon:

Das heißt:

- Funktionsdeklaration:  

```
void function(int array[][COLUMNS], int dim_rows){...}
```

die zweite Dimension (Spalten)  
wird angegeben

die erste Dimension (Reihen)  
muss nicht angegeben werden

dafür kann die erste Dimension  
(Reihen) optional als zweiter Parameter  
übergeben werden
- Funktionsaufruf:  

```
function(array, ROWS);
```

- Im Gegensatz zu anderen Programmiersprachen ist die Arraygröße in C nicht so einfach änderbar
- Beispiel: In einem Programm werden die Matrikelnummern aller Informatik-Studierenden gesammelt. Als der Studiengang vor 50 Jahren aufgesetzt wurde, rechnete man mit 50 Studierenden pro Semester. Aktuell sind es jedoch 100 (Zahlen fiktiv).
- Ein Array mit 50 Elementen kann also **während der Laufzeit des Programms** nicht vergrößert werden

### Lösungen:

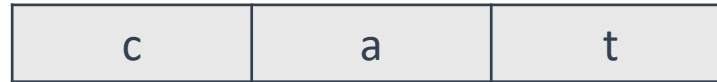
- Arrays von Anfang an überdimensionieren (kostet allerdings viel unnötigen Arbeitsspeicher)
- "Dynamische Arrays" verwenden (nächste Vorlesung)
- Verkettete Listen (später in der Vorlesung)



# Character Arrays & Strings

Neben Zahlen können Arrays auch mit einzelnen Zeichen (character) belegt werden

- `char c[3] = {'c', 'a', 't'};`

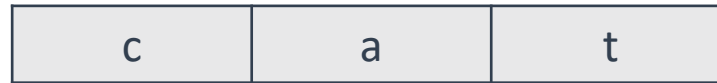


1 Byte

Gespeichert werden hier die einzelnen Zeichen (character), nicht das Wort "cat" (string).

Neben Zahlen können Arrays auch mit einzelnen Zeichen (character) belegt werden

- `char c[3] = {'c', 'a', 't'};`



1 Byte

Gespeichert werden hier die einzelnen Zeichen (character), nicht das Wort "cat" (string). Eine Zeichenfolge (string) kann ebenfalls als Array gespeichert werden:

- `char d[] = "dog";` // das Array d hat vier Elemente, drei sichtbare sowie das unsichtbare String-Ende-Zeichen: `\0`



wird für Strings automatisch  
erzeugt und im Speicher abgelegt

Der Unterschied zwischen String und Character Array ist, dass Strings immer mit `'\0'` beendet werden. Daher:  
`char c[] = {'c', 'a', 't', '\0'};` ist ebenfalls ein String

Byte mit dem Wert 0,  
nicht das Zeichen '0'

Erinnerung an die 2. Vorlesung: Zeichen (character) sind anhand der ASCII-Tabelle kodiert

- `char dog[] = {"dog"};` // ist gleichwertig zu
- `char dog[] = {'d', 'o', 'g', '\0'};` // ist gleichwertig zu
- `char dog[] = {100, 111, 103, 0};`

- Bisher haben wir zum Ausgeben und Eingeben von Strings die Funktionen printf und scanf kennengelernt
- Scanf haben wir allerdings bisher nur für einzelne Zeichen verwendet
- Denn: scanf liest nur bis zum ersten Leerzeichen ein

```
int main(){  
  
    char satz[50];  
  
    printf("\nSatzeingabe: \n");  
    scanf("%s", satz);  
  
    printf("%s\n", satz);  
  
    return 0;  
}
```

Frage:  
Warum muss kein Adressoperator verwendet werden, wenn scanf in ein Array speichert?

Eine andere Möglichkeit Strings (inklusive Leerzeichen) einzulesen, ist die `fgets`-Funktion

Syntax:

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

↑  
Pointer (Zeiger), auf das  
erste Zeichen in einem  
char-Array

↖  
Anzahl der Zeichen, die  
maximal eingelesen  
werden können

↖  
Von wo die Daten  
eingelesen werden

Eine andere Möglichkeit Strings (inklusive Leerzeichen) einzulesen, ist die `fgets`-Funktion

Syntax:

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

Aufruf:

```
char inputString[100]; // char-Array, das Platz für 100 Elemente (Zeichen) umfasst
```

```
fgets(inputString, 100, stdin);
```

↑  
Array, in dem der String  
gespeichert wird

↖  
Einlesen vom  
Standardinput = Tastatur

↖  
98 ist die maximale Anzahl der Zeichen die eingelesen  
werden, da `fgets` immer ein newline-Zeichen (`\n`) sowie  
ein String-Ende-Zeichen (`\0`) anhängt

Eine andere Möglichkeit Strings (inklusive Leerzeichen) einzulesen, ist  
Syntax:

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

Da Funktionen, die mit Strings arbeiten, Zeiger (Pointer) verwenden, werden wir uns erst mit Pointern beschäftigen und dann auf Strings zurückkommen

Aufruf:

```
char inputString[100]; // char-Array, das Platz für 100 Elemente (Zeichen) umfasst
```

```
fgets(inputString, 100, stdin);
```

Array, in dem der String  
gespeichert wird

Einlesen vom  
Standardinput = Tastatur

98 ist die maximale Anzahl der Zeichen die eingelesen  
werden, da fgets immer ein newline-Zeichen (\n) sowie  
ein String-Ende-Zeichen (\0) anhängt

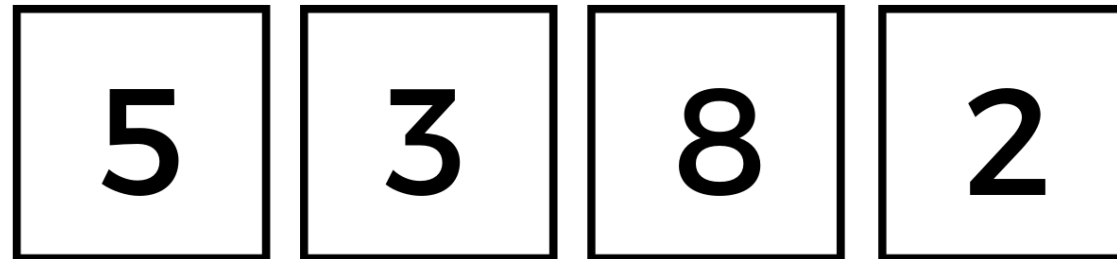


- Arrays sind Datenstrukturen, die mehrere Werte speichern können
- Arrays können eine bis mehrere Dimensionen haben: Vektor → Matrix → Tensor
- Es gibt verschiedene Arten Arrays zu deklarieren und initialisieren
  - `int array[5];`
  - `int array[] = {1, 2, 3, 4, 5};`
  - `for(int i = 0; i < 5; i++){  
    array[i] = i+1; }`
  - `int array[5] = {0, 1};`
  - ...
- Arrays werden Call-by-Reference an Funktionen übergeben, das heißt im Gegensatz zu primitiven Variablen werden sie nicht kopiert sondern es wird ein Zeiger auf die Anfangs-Speicheradresse übergeben
- Bei mehrdimensionalen Arrays müssen alle Dimensionen ab der 2. bei der Funktionsdeklaration angegeben werden
- Character-Arrays können sowohl Zeichen als auch Strings (also Zeichenketten) speichern

# Ein kurzer Einblick in Sortieralgorithmen

- Hat man Datenstrukturen mit mehreren (vielen) Werten, kommt man irgendwann an den Punkt an dem man den Inhalt sortieren möchte / muss
- Daher gibt es eine Vielzahl von Sortieralgorithmen, die zum einen historisch bedingt unterschiedlich schnell sind, sich aber auch für unterschiedliche Anwendungen unterschiedlich gut eignen
- In Algorithmen und Datenstrukturen (3. Semester) werden Sie sich Suchalgorithmen genauer anschauen
- Wie für vieles in der Vorlesung gilt: Im echten Leben würde man Sortieralgorithmen eher nicht selbst implementieren, sondern einen Algorithmus aus einer entsprechenden Bibliothek verwenden → umfangreich geprüft und getestet. Aber natürlich sollte man die Funktionsweise der Algorithmen vorher verstanden haben

- Bubble Sort ist ein recht einfaches aber auch langsames Sortiervorgehen, das wiederholt zwei benachbarte Zahlen vergleicht
- Bubble Sort am Beispiel der Zahlenreihe 5, 3, 8, 2:



Übung:

Arrayelemente: 5, 3, 8, 2.

- Schreiben Sie auf, wie die Elemente mittels Bubblesort getauscht werden.
- Wie muss der Code aussehen, um zwei Arrayelemente zu tauschen.

## Bubble-Sort-Algorithmus:

- Bei Bubble Sort wird bei jedem *Durchlauf* das gesamte Array durchlaufen
- Ist der linke Nachbar größer als der rechte (5, 3), so werden die Werte vertauscht (3, 5), mittels Ringtausch:

```
tmp = array[i]; // Zwischenspeichern des linken Werts in der Variable tmp  
array[i] = array[i+1]; // Die linke Stelle wird mit dem rechten Wert überschrieben  
array[i+1] = tmp; // Die rechte Stelle wird mit dem linken Wert überschrieben
```

- Nach jedem Durchlauf bekommt das letzte (noch nicht feste) Element einen festen Platz

## Bubble-Sort-Algorithmus:

- Bei Bubble Sort wird bei jedem *Durchlauf* das gesamte Array durchlaufen
- Ist der linke Nachbar größer als der rechte (5, 3), so werden diese vertauscht (3, 5), mittels Ringtausch:

```
tmp = array[i]; // Zwischenspeichern des linken Werts in der Variable tmp  
array[i] = array[i+1]; // Die linke Stelle wird mit dem rechten Wert überschrieben  
array[i+1] = tmp; // Die rechte Stelle wird mit dem linken Wert überschrieben
```

- Nach jedem Durchlauf bekommt das letzte (noch nicht feste) Element einen festen Platz

### Übung:

Wie die Arrayelemente getauscht werden ist jetzt klar. In welche Schleife/n würden Sie den Tausch-Codeblock einbetten und wie würden Sie den Schleifenkopf formulieren, um so oft durch die Elemente zu gehen wie notwendig, aber auch nicht öfter?

Die while-Schleife zählt die Anzahl der Arrayelemente runter

```
void bubbleSort(int array[], int size){  
    while(size--){  
        for (int i = 0; i < size; i++){  
            if(array[i] > array[i+1]){  
                int tmp = array[i];  
                array[i] = array[i+1];  
                array[i+1] = tmp;  
            }  
        }  
    }  
}
```

Wenn  $a > b$ , dann werden die Werte von a und b getauscht

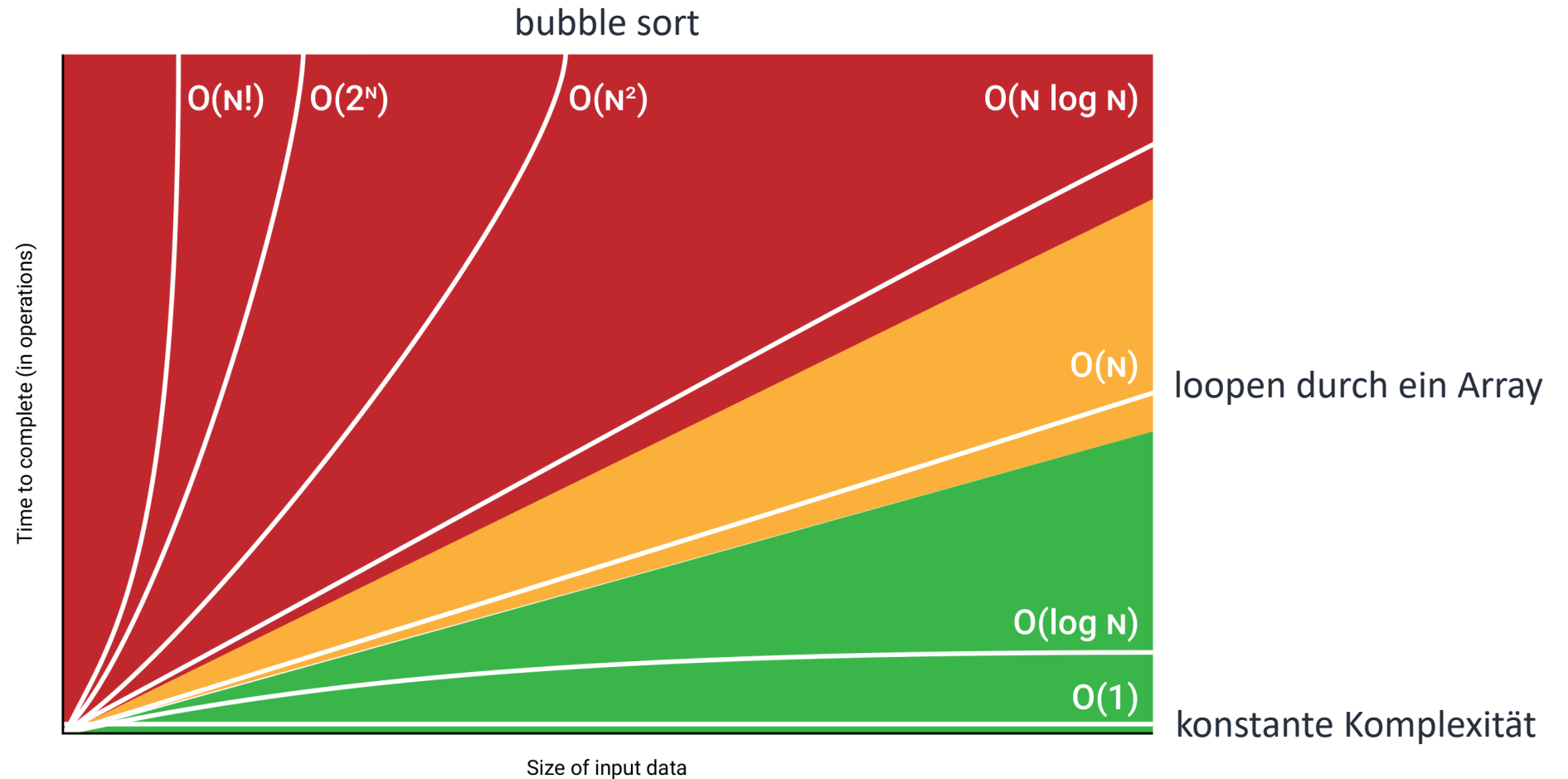
Frage:  
Warum wird tmp innerhalb der if-Anweisung deklariert?



Übersicht über verschiedene Sortieralgorithmen

<https://www.youtube.com/watch?v=kPRA0W1kECg>

Die Effizienz eines Algorithmus wird in der sogenannten Big-O-Notation bewertet



- Die Big-O-Notation ist eine Methode, mit der die Komplexität eines Algorithmus **abgeschätzt** wird.  
→ Wie verändert sich die Performance eines Algorithmus mit zunehmender Anzahl an Elementen (also: wie viel langsamer ist mein Algorithmus, wenn ich 1 Million Arrayelemente habe im Vergleich zu 10)
- Die Big-O-Notation nimmt keine genaue Berechnung vor, vielmehr wird eine Abschätzung des **Worst-Case-Falls** angegeben. Entsprechend muss die Laufzeit eines Algorithmus nicht so lange sein, wie die O-Notation suggeriert
- Es gibt Fälle, in denen eine Optimierung nicht möglich ist, z.B. das Travelling-Salesman-Problem

## Bedeutung der O-Notation

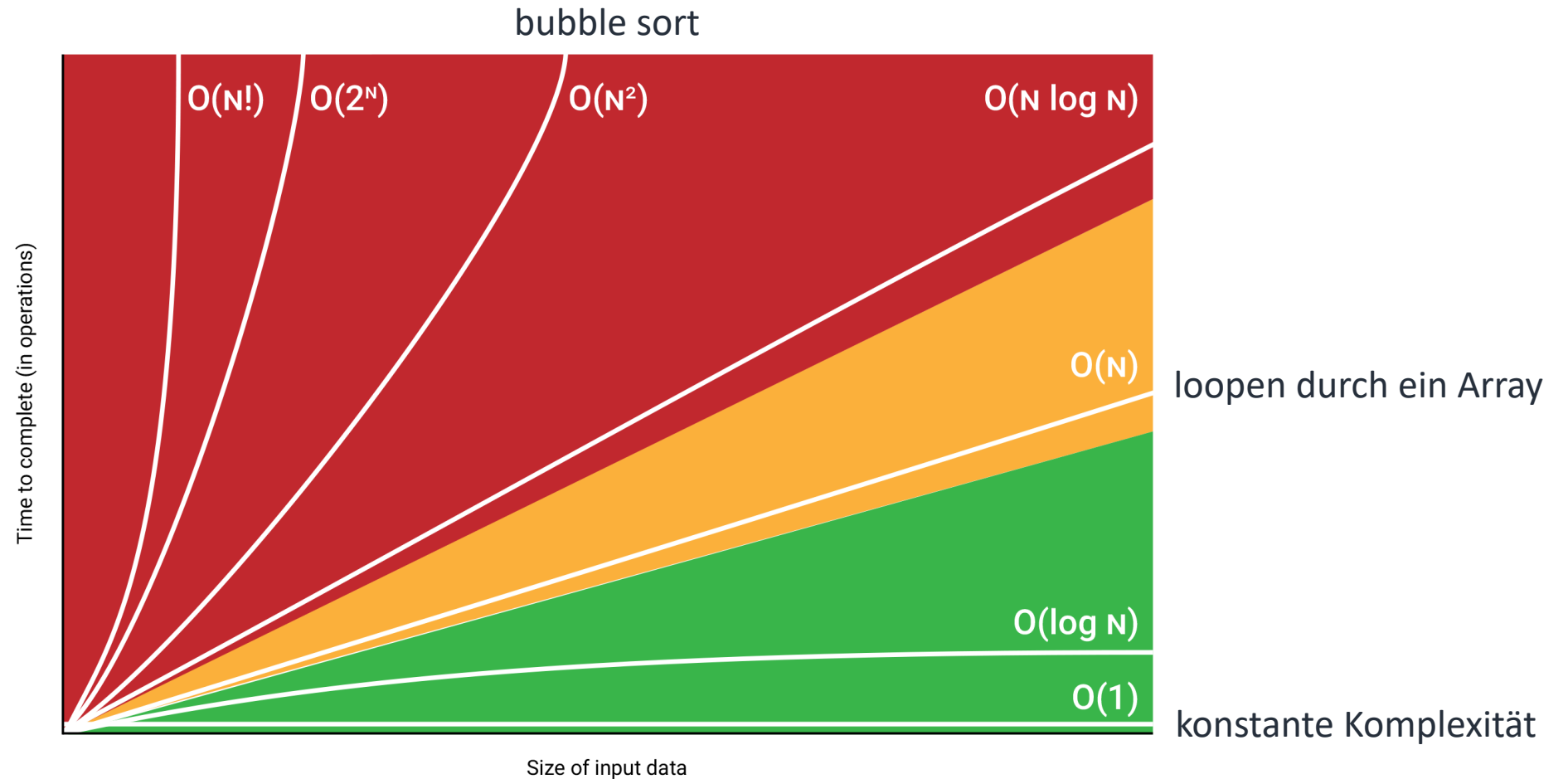
- $O(1)$  steht für eine konstante Komplexität, beispielsweise das Suchen eines Arrayelements anhand der Indexnummer
- $O(n)$  steht für eine lineare Komplexität, wobei  $n$  für die Anzahl von Elementen steht, beispielsweise die Anzahl von Elementen im Array. Die Laufzeit eines Algorithmus mit  $O(n)$ -Komplexität nimmt mit der Anzahl der Elemente linear zu

```
int sum(int n){  
    int sum = 0;  
    for(int i = 0; i <= n; i++){  
        sum += i;  
    }  
    return sum;  
}
```

```
int sum(int n){  
    return n * (n + 1) / 2;  
}
```

Frage:  
Wieviele Schritte brauchen die  
beiden Algorithmen in etwa, wenn  $n$   
= 100?

Die Effizienz eines Algorithmus wird in der sogenannten Big-O-Notation bewertet



# Mathematische Funktionen in C

### Zufallszahlengenerator (random number generator)

- Zufallszahlen kommen beim Programmieren immer wieder vor, daher gibt es in jeder Programmiersprache Pseudo-Zufallszahlengeneratoren
- Im BubbleSort-Code werden Zufallszahlen verwendet, um das Array zu füllen

```
#include <stdlib.h> // srand() / rand()
#include <time.h>

void generateRandomArray(int array[], int size, int min, int max){
    for (int i = 0; i < size; i++){
        array[i] = rand();
    }
}
```

Zufallszahlen generieren mit rand():

- stdlib.h bindet die rand-Funktion ein
- rand gibt pseudo-zufällige Integerzahlen aus (zwischen Null und einem Maximalwert, der implementationsabhängig ist)
- Zufallszahlengeneratoren generieren immer dieselbe Zahlenfolge, wenn man sie nicht mit einem unterschiedlichen Startwert (seed) startet: `srand(time(NULL));`  
rand() ist der eigentliche Zufallszahlengenerator, srand() wird verwendet um den Startwert zu ändern
- Indem man die vergangenen Sekunden seit dem 1.1.1970 um 00:00:00 Uhr verwendet, erhält man bei jedem Aufruf einen neuen seed

```
#include <stdlib.h> // srand() / rand()
#include <time.h>

int main(){
    srand(time(NULL));
}
```



Um Zufallszahlen in einem festgelegten Bereich zu erhalten, können wir die Modulo-Funktion verwenden:

- `rand() % 10` // Werte zwischen 0 und 9
- `rand() % (10 + 1)` // Werte zwischen 0 und 10
- Modulo gibt den Rest einer Division aus, also:  $1337 \% 10 = 133 \text{ Rest } 7$

Übung:  
Was ist das Ergebnis von  $52 \% 7$ ?

Um Zufallszahlen in einem festgelegten Bereich zu erhalten, können wir die Modulo-Funktion verwenden:

- `rand() % (10 + 1) // Werte zwischen 0 und 10`
- Modulo gibt den Rest einer Division aus, also:  $1337 \% 10 = 7$

```
#include <stdlib.h> // srand() / rand()
#include <time.h>

void generateRandomArray(int array[], int size, int min, int max){
    for (int i = 0; i < size; i++){
        array[i] = rand() % (max - min + 1) + min;
    }
}
```

Frage:  
Was wird in dieser Zeile berechnet?

Um Zufallszahlen in einem festgelegten Bereich zu erhalten, können wir die Modulo-Funktion verwenden:

- `rand() % (10 + 1)` // Werte zwischen 0 und 10
- Modulo gibt den Rest einer Division aus, also:  $1337 \% 10 = 7$
- Wollen wir einen Wertebereich von 20 bis 80, dann ergibt die Rechnung im Code z.B.:  
$$0 \% (80 - 20 + 1) + 20 = 0 \% 61 + 20 = 0 + 20 = 20$$
$$60 \% (80 - 20 + 1) + 20 = 60 \% 61 + 20 = 60 + 20 = 80$$
$$61 \% (80 - 20 + 1) + 20 = 61 \% 61 + 20 = 0 + 20 = 20$$
- Der Modulo-Operator hat Priorität vor  $+/-$

```
#include <stdlib.h> // srand() / rand()
#include <time.h>

void generateRandomArray(int array[], int size, int min, int max){
    for (int i = 0; i < size; i++){
        array[i] = rand() % (max - min + 1) + min;
    }
}
```

- Den Inkrement-Operator haben Sie schon in der for-Schleife kennengelernt:  
`for(int i = 0; i < x; i++){}`
- `i++` ist die Kurzschreibweise für: `i = i + 1;`
- `i--` ist die Kurzschreibweise für: `i = i - 1;`

Frage:  
In der Codezeile:  
`int a, b = 12;`  
`a = b++;`  
welchen Werte hat a und welchen Wert hat b?

- Werden Inkrement bzw. Dekrement-Operatoren in der Postfix-Schreibweise verwendet (also `i++` bzw. `i--`), so wird bei einer Zuweisung erst der "alte" Wert zurückgegeben und erst dann wird `i` erhöht bzw. verringert

`a = b++;` // `a` wird also erst der Wert von `b` zugewiesen und dann wird `b` erhöht

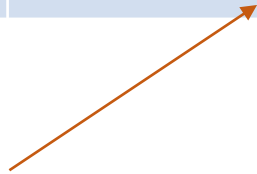
- Soll der erhöhte bzw. verringerte Werte sofort zugewiesen werden, kann die Präfix-Schreibweise verwendet werden

`a = ++b;` // `b` wird um eins erhöht und der erhöhte Wert wird `a` zugewiesen

Neben Inkrement und Dekrement gibt es auch zusammengesetzte Zuweisungen

Darstellung	Bedeutung
<code>+=</code>	<code>a+=b</code> ist gleichwertig zu <code>a = a + b</code>
<code>-=</code>	<code>a-=b</code> ist gleichwertig zu <code>a = a - b</code>
<code>*=</code>	<code>a*=b</code> ist gleichwertig zu <code>a = a * b</code>
<code>/=</code>	<code>a/=b</code> ist gleichwertig zu <code>a = a / b</code>
<code>%=</code>	<code>a%=b</code> ist gleichwertig zu <code>a = a % b</code>

Das '=' kommt als zweiter Operator,  
andernfalls wäre es eine Zuweisung zu a:  
`a = +a` ← falsch!



Wie in der Mathematik werden auch in der Programmierung verschiedene Operatoren mit unterschiedlicher Priorität ausgewertet

Operator	Bedeutung / Name	Priorität
++ , --	Erhöhung / Verringerung NACH Auswertung	1
++ , --	Erhöhung / Verringerung VOR Auswertung	2
!	Negation	2
(type)	Typumwandlung	2
*, /, %	Multiplikation, Division, Modulo	3
+, -	Addition, Subtraktion	4
+=, -=, *=, /=, %=	zusammengesetzte Zuweisungen	14

Frage:

- Wie kann die folgende Rechnung verändert werden, so dass zuerst addiert und subtrahiert wird und dann multipliziert und dividiert:  
 $23 - 5 * 199 + 13 \% 10$
- Welche Operation wird dann zuerst ausgeführt? Multiplikation oder Modulo?



- Natürlich gibt es in C auch für gängige mathematische Funktionen eine Standard-Bibliothek: math.h
- Wie die anderen Bibliotheken auch, wird math.h mittels include eingebunden. Bei Linux muss allerdings zusätzlich bei der Kompilierung die Compiler-Flag `-lm` verwendet werden:  
*gcc name.c -o name -lm*
- Allerdings enthält math.h für jeden Datentyp unterschiedliche Funktionen, z.B. `sqrtf()` für float, `sqrt()` für double, `sqrtl()` für long double...
- Der Einfachheit halber gibt es daher die Bibliothek tgmath.h, die diese Funktionen vereinheitlicht, das heißt die enthaltenen Funktionen nehmen float, double, long double sowie komplexe Zahlen an

- Welche mathematische Funktionen enthält math.h / tgmath.h?
  - typische Funktionen wie Wurzel (square root, `sqrt()`), Potenzen (`pow()`), aufrunden (`ceil()`), abrunden (`floor()`), ...  
`sqrt(value)`, `pow(value, 3)`
  - mathematische Konstanten wie Pi, e, Wurzel aus 2, ...
  - Makros zum Vergleichen von reellen Zahlen (`isgreater(x, y)`, `isfinite(x)`, `isinf(x)`, `isnan(x)`)
- Immer wenn man eine mathematische Funktion oder Konstante benötigt ist es sinnvoll vorher zu überprüfen ob diese Funktion / Konstante in einer Bibliothek vorkommt
  - Standard C library, z.B. [https://www.geeksforgeeks.org/c-library-math\\_h/](https://www.geeksforgeeks.org/c-library-math_h/)
  - im Headerfile nachschauen

Im Code haben wir gesehen, dass float und double nicht auf Gleichheit überprüft werden können, aufgrund der unterschiedlichen Genauigkeit der Zahlen

Fragen:

- Welches Ergebnis liefert der folgende Vergleich:  
`float f = 1.2;`  
`f == 1.2;`
- Wie könnte eine Funktion aussehen, die eine float- und eine double-Variable auf Gleichheit überprüft?

- Der Begriff NaN steht für not a number und kommt auf die eine oder andere Art in allen Programmiersprachen vor
- NaN kann als Ergebnis einer nicht erlaubten Rechnung ausgegeben werden (Division durch 0 z.B.), aber auch dann eingesetzt werden, wenn klar sein soll, dass die Variable keinen "echten" Wert hat
- Beispiel: Initialisieren von Variablen mit NaN → Wird bei einer späteren Abfrage festgestellt, dass der Wert der Variablen NaN ist, so ist klar, dass diese noch nicht initialisiert wurde (das ist bei einer Initialisierung mit 0 nicht so eindeutig)
- Syntax:  
`float f = NAN; // kann abgefragt werden mittels: isnan(f)`

- Arrays sind Datentypen, die mehrere Elemente gemeinsam abspeichern
- Die Größe von Arrays ist statisch, das heißt sie kann zur Laufzeit des Programms nicht mehr geändert werden
- Strings sind Character-Arrays mit zusätzlichem String-Ende-Zeichen ('\0')
- Arrays werden an Funktionen per Call-by-Reference übergeben, das heißt als Zeiger auf die Anfangsadresse des Arrays
- Um Arrays zu sortieren gibt es eine Vielzahl von Sortieralgorithmen, deren Schnelligkeit von der Art der Daten abhängt
- Die Big-O-Notation ist eine Methode, um die Performance eines Algorithmus zu bewerten
- Zufallszahlengeneratoren liefern pseudo-zufällige Zahlen und müssen mit einem variierenden Seed initialisiert werden, um jedes Mal eine neue Zahlenreihe zu liefern
- Mathematische Funktionen können in C über die `math.h`/`gtmath.h`-Datei eingebunden werden