# Praktikum C-Programmierung 2026

## Heap

Jannek Squar

2026-05-27

Scientific Computing
Universität Hamburg

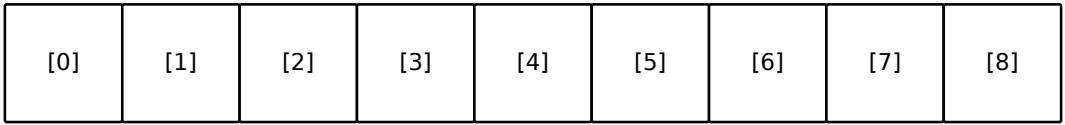
Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

# Outline

# Table of Contents

# Speicherlayout 1D

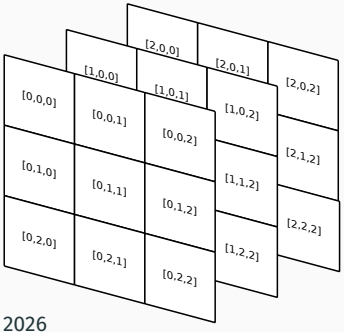
- Speicher ist eindimensional
- `int *foo = malloc(sizeof(int) * 10);`



# Speicherlayout 1D

- Use-Cases oft mehrdimensional
- Beispiele
  - Skalare Konstanten
  - 2D Oberflächen
  - 3D Atmosphärenschichten
  - 4D Zeitreihen von 3D-Werten

| | | |
|---|---|---|
| [0,0] | [0,1] | [0,2] |
| [1,0] | [1,1] | [1,2] |
| [2,0] | [2,1] | [2,2] |



# Erster Versuch

## Trivialer Ansatz geht leider nicht

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

int main(void) {
  int *arr = malloc(sizeof(int) * 9);
  int counter = 0;

  for (int zeile = 0; zeile < 3; zeile++) {
    for (int spalte = 0; spalte < 3; spalte++) {
      arr[zeile][spalte] = counter;
      counter++;
    }
  }

  for (int index = 0; index < counter; index++) {
    printf("%d ", arr[index]);
  }
  free(arr);
  return 0;
}
```

```
$ gcc -o wrong_index_01.x wrong_index_01.c
wrong_index_01.c: In function ‘main’:
wrong_index_01.c:10:23: error: subscripted value is
↪  neither array nor pointer nor vector
  10 |             arr[zeile][spalte] = counter;
```

# Erster Versuch

## Trivialer Ansatz geht leider nicht

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

int main(void) {
  int *arr = malloc(sizeof(int) * 9);
  int counter = 0;

  for (int zeile = 0; zeile < 3; zeile++) {
    for (int spalte = 0; spalte < 3; spalte++) {
      arr[zeile][spalte] = counter;
      counter++;
    }
  }

  for (int index = 0; index < counter; index++) {
    printf("%d ", arr[index]);
  }
  free(arr);
  return 0;
}
```

```
$ gcc -o wrong_index_01.x wrong_index_01.c
wrong_index_01.c: In function ‘main’:
wrong_index_01.c:10:23: error: subscripted value is
↪  neither array nor pointer nor vector
  10 |             arr[zeile][spalte] = counter;
```

# Zweiter Versuch

## Trivialer Ansatz geht leider noch immer nicht

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

int main(void) {
  int *arr = malloc(sizeof(int) * 9);
  int counter = 0;

  for (int zeile = 0; zeile < 3; zeile++) {
    for (int spalte = 0; spalte < 3; spalte++) {
      (&arr[zeile])[spalte] = counter;
      counter++;
    }
  }

  for (int index = 0; index < counter; index++) {
    printf("%d ", arr[index]);
  }
  free(arr);
  return 0;
}
```

```
$ gcc -o wrong_index_02.x wrong_index_02.c
$ ./wrong_index_02.x
0 3 6 7 8 0 0 0 0
```

Was intern passiert:

```
zeile==0: 0 1 2 0 0 0 0 0 0
zeile==1: 0 3 4 5 0 0 0 0 0
zeile==2: 0 3 6 7 8 0 0 0 0
```

## Trivialer Ansatz geht leider noch immer nicht

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

int main(void) {
  int *arr = malloc(sizeof(int) * 9);
  int counter = 0;

  for (int zeile = 0; zeile < 3; zeile++) {
    for (int spalte = 0; spalte < 3; spalte++) {
      (&arr[zeile])[spalte] = counter;
      counter++;
    }
  }

  for (int index = 0; index < counter; index++) {
    printf("%d ", arr[index]);
  }
  free(arr);
  return 0;
}
```

```
$ gcc -o wrong_index_02.x wrong_index_02.c
$ ./wrong_index_02.x
0 3 6 7 8 0 0 0 0
```

Was intern passiert:

```
zeile==0: 0 1 2 0 0 0 0 0 0
zeile==1: 0 3 4 5 0 0 0 0 0
zeile==2: 0 3 6 7 8 0 0 0 0
```

## Trivialer Ansatz geht leider noch immer nicht

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

int main(void) {
  int *arr = malloc(sizeof(int) * 9);
  int counter = 0;

  for (int zeile = 0; zeile < 3; zeile++) {
    for (int spalte = 0; spalte < 3; spalte++) {
      (&arr[zeile])[spalte] = counter;
      counter++;
    }
  }

  for (int index = 0; index < counter; index++) {
    printf("%d ", arr[index]);
  }
  free(arr);
  return 0;
}
```

```
$ gcc -o wrong_index_02.x wrong_index_02.c
$ ./wrong_index_02.x
0 3 6 7 8 0 0 0 0
```

## Was intern passiert:

```
zeile==0: 0 1 2 0 0 0 0 0 0
zeile==1: 0 3 4 5 0 0 0 0 0
zeile==2: 0 3 6 7 8 0 0 0 0
```

# Wie macht man es jetzt richtig?

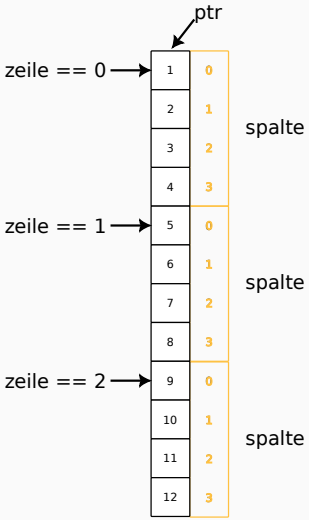
Ausgewählte Ansätze [1]:

- Manuelle Pointer-Arithmetik
- Zeilenpointer auf Stack-Array
- Zeilenpointer auf Heap-Array
- Gemischte Datenstruktur
- Pointer auf 2D VLA
- Pointer auf 1D VLA

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |

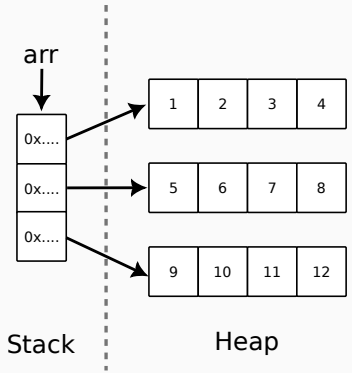
# Ansatz 1: Manuelle Pointer-Arithmetik

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

int main(void) {
  int r = 3, c = 4;

  int *ptr = malloc((r * c) * sizeof(int));

  for (int i = 0; i < r * c; i++)
    ptr[i] = i + 1;

  for (int zeile = 0; zeile < r; zeile++) {
    for (int spalte = 0; spalte < c; spalte++)
      printf("%d ", ptr[zeile * c + spalte]);
    printf("\n");
  }

  free(ptr);
  return 0;
}
```



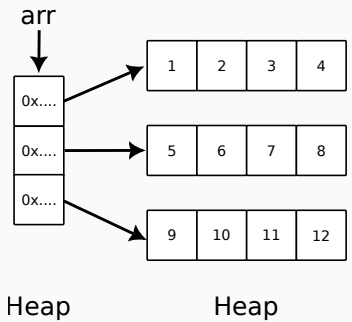
# Ansatz 2: Zeilenpointer auf Stack-Array

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

int main(void) {
  int r = 3, c = 4, zeile, spalte, count;

  int *arr[r];
  for (zeile = 0; zeile < r; zeile++)
    arr[zeile] = malloc(c * sizeof(int));

  count = 0;
  for (zeile = 0; zeile < r; zeile++)
    for (spalte = 0; spalte < c; spalte++)
      arr[zeile][spalte] = ++count;

  for (zeile = 0; zeile < r; zeile++)
    for (spalte = 0; spalte < c; spalte++)
      printf("%d ", arr[zeile][spalte]);

  for (zeile = 0; zeile < r; zeile++)
    free(arr[zeile]);

  return 0;
}
```



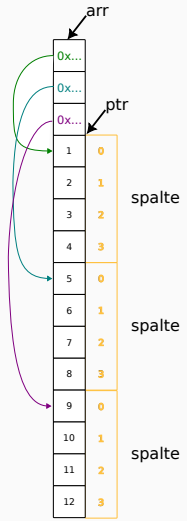
# Ansatz 3: Zeilenpointer auf Heap-Array

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

int main(void) {
  int r = 3, c = 4, zeile, spalte, count;

  int **arr = malloc(r * sizeof(int *));
  for (zeile = 0; zeile < r; zeile++)
    arr[zeile] = malloc(c * sizeof(int));

  count = 0;
  for (zeile = 0; zeile < r; zeile++)
    for (spalte = 0; spalte < c; spalte++)
      arr[zeile][spalte] = ++count;

  for (zeile = 0; zeile < r; zeile++)
    for (spalte = 0; spalte < c; spalte++)
      printf("%d ", arr[zeile][spalte]);

  for (zeile = 0; zeile < r; zeile++)
    free(arr[zeile]);

  free(arr);
  return 0;
}
```



# Ansatz 4: Gemischte Datenstruktur

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

int main(void) {
  int r = 3, c = 4, len = 0;
  int *ptr, **arr;
  int count = 0, zeile, spalte;

  len = sizeof(int *) * r + sizeof(int) * c * r;
  arr = malloc(len);

  ptr = (int *)(arr + r);

  for (zeile = 0; zeile < r; zeile++)
    arr[zeile] = (ptr + c * zeile);

  for (zeile = 0; zeile < r; zeile++)
    for (spalte = 0; spalte < c; spalte++)
      arr[zeile][spalte] = ++count;

  for (zeile = 0; zeile < r; zeile++)
    for (spalte = 0; spalte < c; spalte++)
      printf("%d ", arr[zeile][spalte]);
  free(arr);
  return 0;
}
```



# Ansatz 5: Pointer auf 2D VLA

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

int main(void) {
  int row = 3, col = 4, zeile, spalte, count;

  int (*arr)[row][col] = malloc(sizeof(*arr));

  count = 0;
  for (zeile = 0; zeile < row; zeile++)
    for (spalte = 0; spalte < col; spalte++)
      (*arr)[zeile][spalte] = ++count;

  for (zeile = 0; zeile < row; zeile++)
    for (spalte = 0; spalte < col; spalte++) {
      printf("%d ", (*arr)[zeile][spalte]);
    }

  free(arr);
  return 0;
}
```

Einschränkungen:

- Erst seit C99
- Seit C11+ nur noch optional
  - Stackoverflow leichter
  - Geringe Nutzung
- Sehr eingeschränkter Support durch MSVC

# Ansatz 6: Pointer auf 1D VLA

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

int main(void) {
  int row = 3, col = 4, zeile, spalte, count;

  int (*arr)[col] = malloc(row * sizeof(*arr));

  count = 0;
  for (zeile = 0; zeile < row; zeile++)
    for (spalte = 0; spalte < col; spalte++)
      arr[zeile][spalte] = ++count;

  for (zeile = 0; zeile < row; zeile++)
    for (spalte = 0; spalte < col; spalte++)
      printf("%d ", arr[zeile][spalte]);

  free(arr);
  return 0;
}
```

# Table of Contents

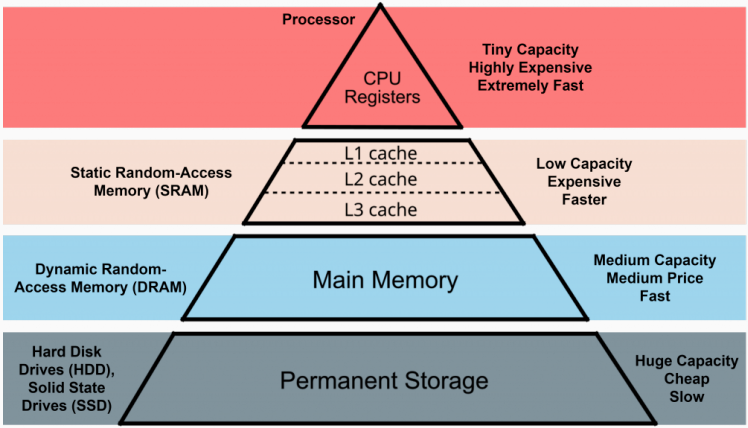
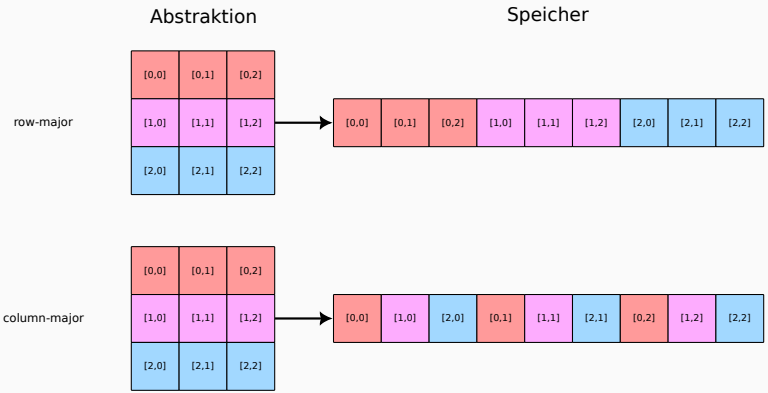



**Abbildung 1:** Generelle Speicher-Hierarchie [2]

# Speicher-Format




- C nutzt `row-major` Speicherlayout
- Fortran nutzt `column-major` Speicherlayout

# Motivation

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

int main(void) {
  int r = 30000;
  int c = 30000;

  int *arr = malloc((size_t)r * c * sizeof(int));

  for (int j = 0; j < r; j++) {
    for (int i = 0; i < c; i++) {
      arr[i * c + j] = i+j;
    }
  }

  free(arr);
  return 0;
}
```

```
$ gcc -o caches_01B.x caches_01B.c
$ time ./caches_01B.x
./caches_01B.x  13.35s user 1.64s system 99% cpu
↪  15.092 total
```

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

int main(void) {
  int r = 30000;
  int c = 30000;

  int *arr = malloc((size_t)r * c * sizeof(int));

  for (int i = 0; i < r; i++) {
    for (int j = 0; j < c; j++) {
      arr[i * c + j] = i+j;
    }
  }

  free(arr);
  return 0;
}
```

```
$ gcc -o caches_02B.x caches_02B.c
$  time ./caches_02B.x
./caches_02B.x  2.60s user 1.29s system 99% cpu 3.907
↪  total
```

# Motivation

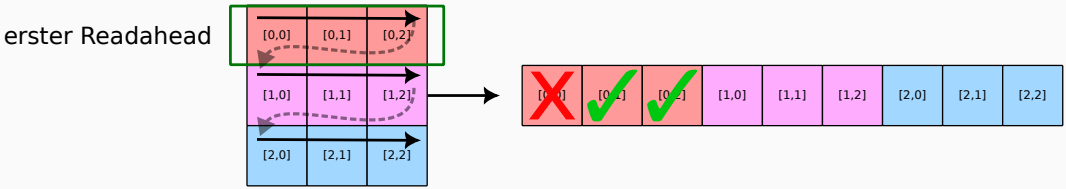
```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int r = 30000;
  int c = 30000;

  int *arr = malloc((size_t)r * c * sizeof(int));

  for (int j = 0; j < r; j++) {
    for (int i = 0; i < c; i++) {
      arr[i * c + j] = i+j;
    }
  }

  free(arr);
  return 0;
}
```

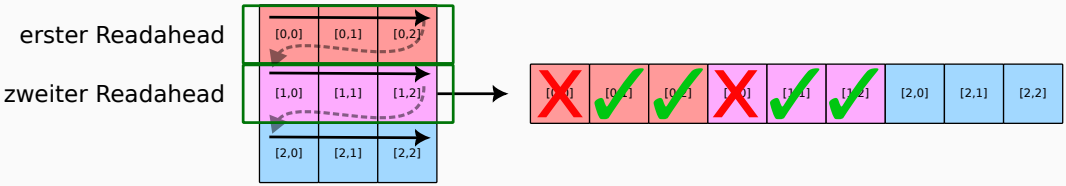
```
$ gcc -o caches_01B.x caches_01B.c
$ time ./caches_01B.x
./caches_01B.x  13.35s user 1.64s system 99% cpu
↪  15.092 total
```

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

int main(void) {
  int r = 30000;
  int c = 30000;

  int *arr = malloc((size_t)r * c * sizeof(int));

  for (int i = 0; i < r; i++) {
    for (int j = 0; j < c; j++) {
      arr[i * c + j] = i+j;
    }
  }

  free(arr);
  return 0;
}
```

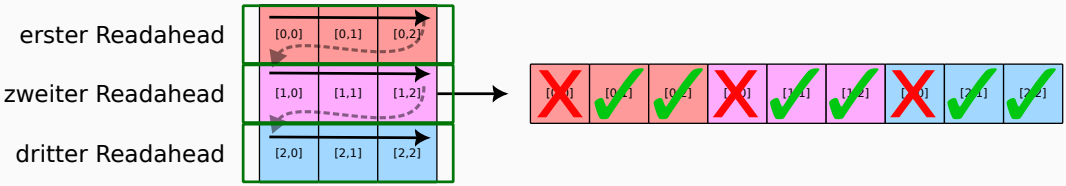
```
$ gcc -o caches_02B.x caches_02B.c
$  time ./caches_02B.x
./caches_02B.x  2.60s user 1.29s system 99% cpu 3.907
↪  total
```

# Gutes Zugriffsmuster

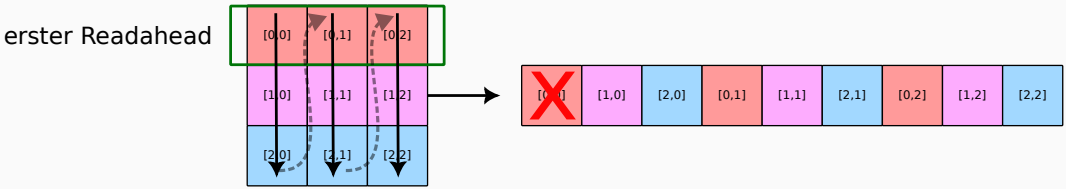



erster Readahead
zweiter Readahead
[0,0]
[0,1]
[0,2]
[1,0]
[1,1]
[1,2]
[2,0]
[2,1]
[2,2]
[2,0]
[2,1]
[2,2]

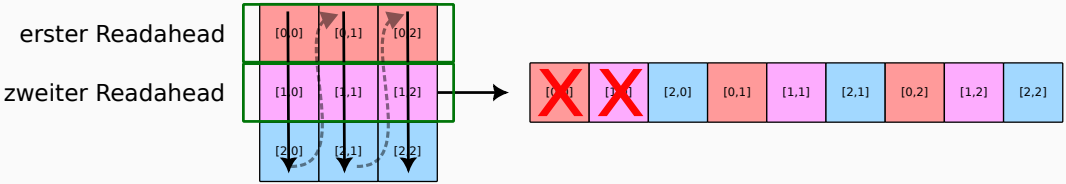
# Gutes Zugriffsmuster



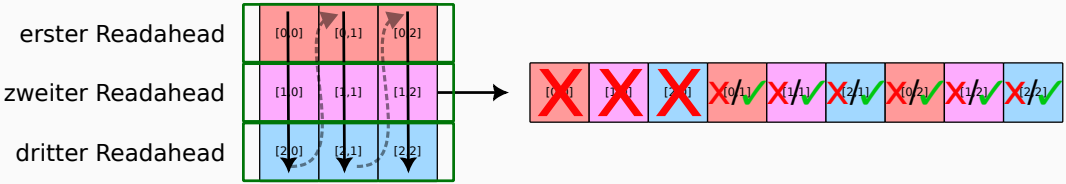
# Schlechtes Zugriffsmuster



# Schlechtes Zugriffsmuster



# Schlechtes Zugriffsmuster



# Perf-Statistiken

```
$ perf stat -e cache-misses ./caches_01B.x

 Performance counter stats for './caches_01B.x':

     2,881,919,351      cache-misses:u

      14.863819767 seconds time elapsed

      13.212312000 seconds user
       1.535485000 seconds sys
```

```
$ perf stat -e cache-misses ./caches_02B.x

 Performance counter stats for './caches_02B.x':

            51,757      cache-misses:u

       3.901318744 seconds time elapsed

       2.619147000 seconds user
       1.255825000 seconds sys
```

# Best Practice

- Daten möglichst nah an CPU(-Kern) halten
- Nur bedingt Einfluss auf Speicherort
- Ausnutzung von Heuristiken
  - Cache Readahead: „Der letzte Index ist der schnellste, der erste der langsamste“
  - Cache Eviction: Wiederverwendung temporal/räumlich lokaler Daten

# References


[1] How to dynamically allocate a 2D array in C? https://www.geeksforgeeks.org/c/dynamically-allocate-2d-array-c/

[2] Memory Hierarchy – How does computer memory work? https://spear-itn.eu/memory-hierarchy-how-does-computer-memory-work/