INF1
Speicherverwaltung
und zweidimensionale Arrays

- Speicherorganisation
- Dynamischer Speicher in C
- Zweidimensionale Arrays

Speicherorganisation in C

Speicherorganisation

- Anlegen eines Arrays:

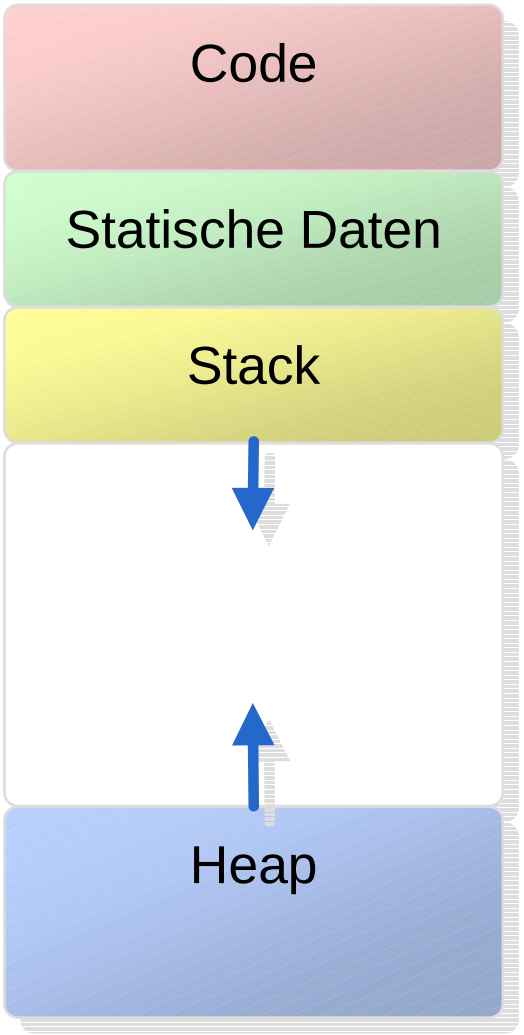
```
double messwerte[100];
```

- Grösse der Arrays ist fest im Programm codiert
- Ein Ändern der Grösse erfordert neues Kompilieren des Programms

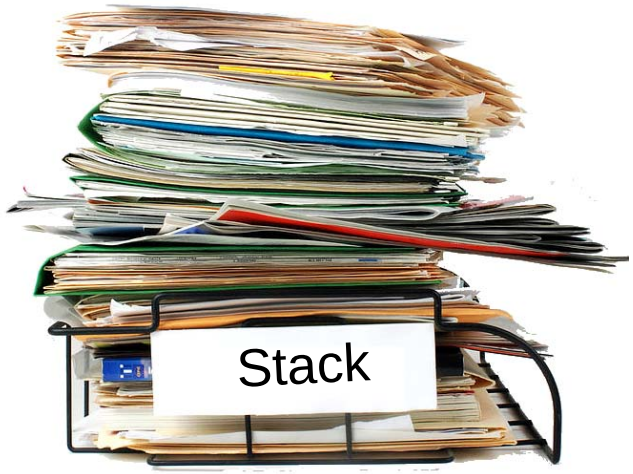
Speicherorganisation

- Oft ist aber nicht im voraus bekannt, wie viel Daten von einem Programm verarbeitet werden
- Es braucht daher eine Möglichkeit, zur Laufzeit des Programmes Speicher vom Betriebssystem anzufordern
- Wenn der Speicher nicht mehr benötigt wird, kann er wieder freigegeben werden

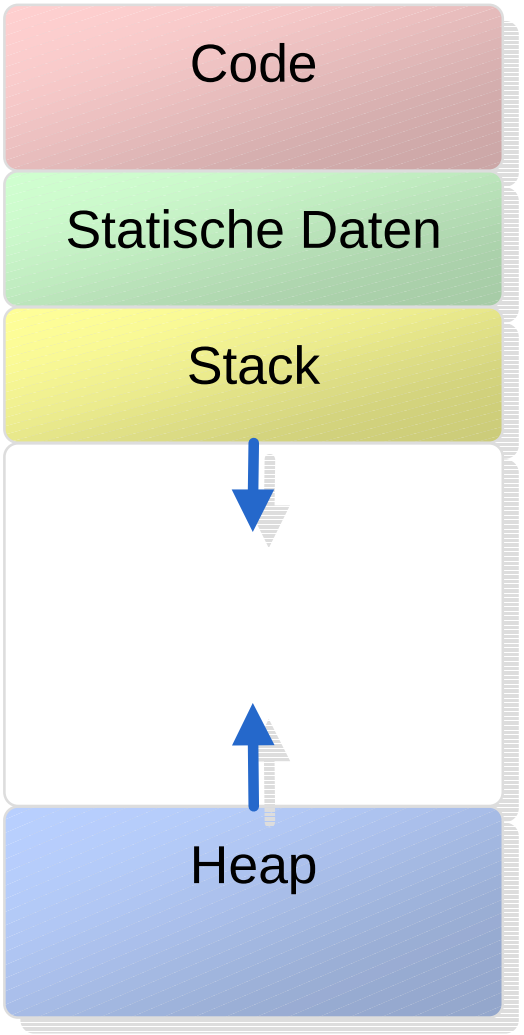
Speicherbereiche



- Speicher eines laufenden Programms (Prozess)
- Speicher für globale und statische Variablen wird im **statischen *Datenbereich*** alloziert
- Speicher für lokale Variablen wird auf dem ***Stack*** **automatisch** alloziert
- Auf dem ***Heap*** kann Speicher **dynamisch** alloziert werden



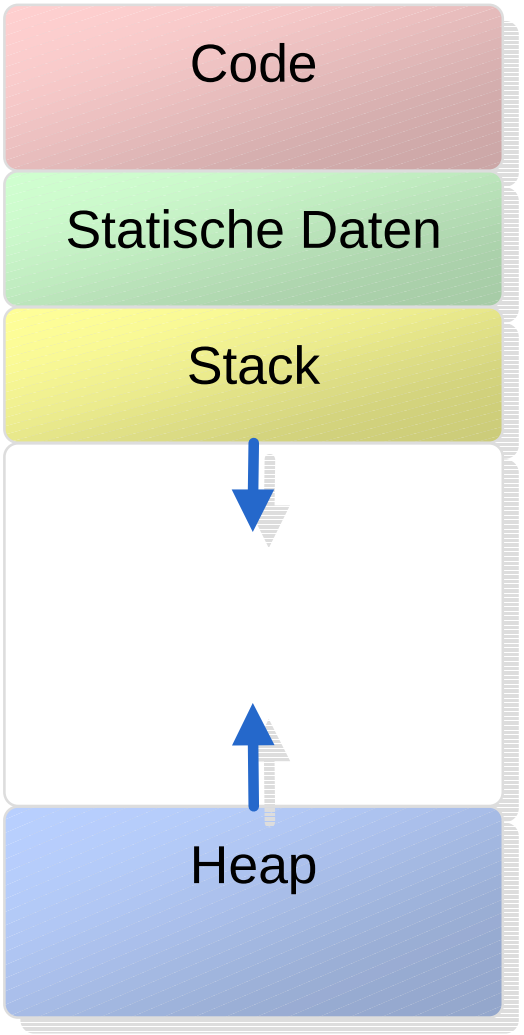
Statische Daten



- Globale Variablen
- und `static` innerhalb von Funktionen
- Statische Speicherallokation: Speicher wird beim Programmstart reserviert

```
int i;
void foo(void) {
   static int j;
   ... ...
}
```

Stack Daten



- Lokale Variablen und Übergabeparameter
- **Automatische Speicherallokation:**
  Beim Funktionsaufruf wird Speicher automatisch für Variablen angelegt und am Ende der Funktion wieder **gelöscht**

```
void foo(int i) {
   int j;
   ... ...
}
```

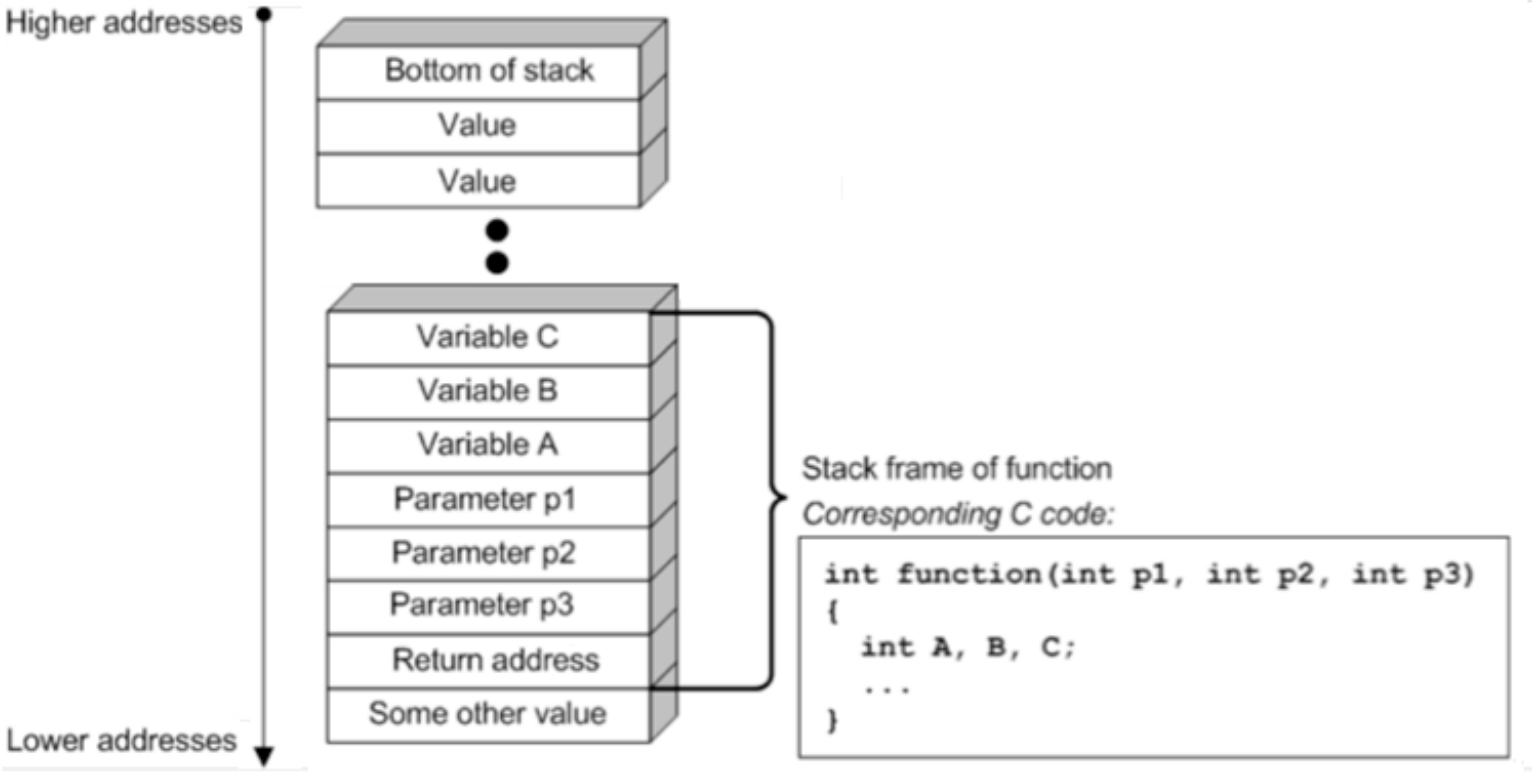
Falls man z.B. via Pointer noch einen Zugriff ermöglicht -> Dangling Pointer

Nach Verlassen der Funktion gehen Stack Daten verloren

Aufbau des Stacks

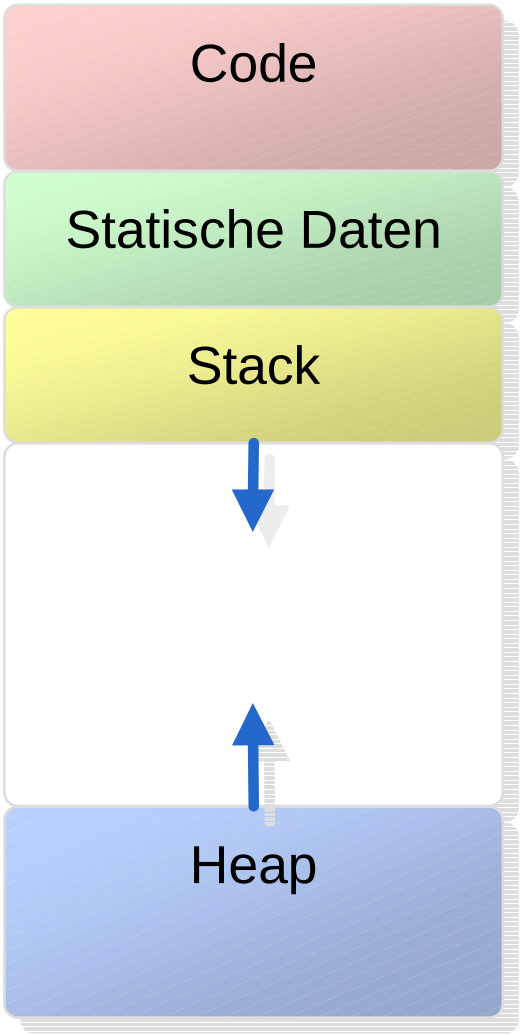
outlook

- Stack Variablen werden in sog "Stack Frames" organisiert
- Ein Stack Frame enthält: Variablen, Parameter und Rücksprungadresse



© www.drdobbs.com

Heap Daten



- Während der Laufzeit nach Bedarf dynamisch (d.h. zur Laufzeit) alloziert (kreiert)
- Genau so viel Speicherplatz alloziert, wie benötigt wird
- Wenn man zur Entwicklungszeit nicht weiss, wie viel Speicher benötigt wird

```
void foo(void) {
   int* pi = (int*) malloc(sizeof(int));
   ... ...
}
```

Werden explizit angelegt und sind so lange gültig, bis sie (explizit) wieder freigegeben werden

Dynamischer Speicher in C

Speicher dynamisch allozieren

```
void * malloc(int t_size);
```

- Alloziert Speicher der Grösse *size* Bytes auf dem Heap
- Gibt die Adresse (also einen Pointer) dieses Speicherplatz zurück (void *)
- Der Pointer muss noch in den gewünschten Datentyp konvertiert werden
- Benötigt Bibliothek *stdlib.h*

```
void * calloc(int n, int t_size);
```

- Alloziert Speicher der Grösse n * *size* Bytes auf dem Heap und **setzt Speicher zu 0** -> d.h. Speicher wird automatisch initialisiert -> good
- Ansonsten gleich wie malloc

Speicher dynamisch allozieren

- size gibt den Speicherplatzbedarf (in Bytes) an
- da der Platformabhängig ist, verwendet man sizeof(<Daten Typ>)

```
malloc(sizeof(int));

malloc(sizeof(double));
```

- Zurückgegeben wird ein void* den man zum gewünschten Typ umwandeln (casten) muss

```
int* pi = (int*)malloc(sizeof(int));
```

macht aber niemand

- Nach *malloc()* sollte man testen, ob der Rückgabewert NULL ist.
- Mit realloc kann man die Grösse des Speicher verändern

outlook

kann aber leicht zu Speicherproblemen verursachen -> Vorsicht

https://openbook.rheinwerk-verlag.de/c_von_a_bis_z/014_c_dyn_speicherverwaltung_008.htm

Speicher freigeben

```
void free(void * ptr);
```

- Gibt den dynamisch zugewiesenen Speicher wieder frei
- Der Zeiger *ptr* zeigt danach auf einen nicht mehr gültigen Speicherbereich und **darf nicht mehr verwendet werden**

Beispiel 1 (einfache Variable)

```
int *pi;
pi = (int *) malloc(sizeof(int));
*pi = 5;
...
free(pi); /* Speicher wird freigegeben */
```

```
double *pd;
pd = (double *) malloc(sizeof(double));
*pd = 5.0;
...
free(pd); /* Speicher wird freigegeben */
```

Beispiel 2 (Array von int)

```
int *pwerte;
n = 250;
pwerte = (int *) malloc(n * sizeof(int));

/* und so erfolgt der Zugriff: */
pwerte[95] = 12;

/* oder mit Zeigernotation: */
printf("%i\n", *(pwerte+95));

...
free(pwerte); /* Speicher wird freigegeben */
```

Array vergrössern

- Nun ist es auch kein Problem mehr, wenn sich zur Laufzeit des Programms herausstellt, dass ein Array zu klein ist
- Es wird dynamisch einfach ein grösseres Array angelegt
- Die bestehenden Werte müssen aber in das grössere Array umkopiert werden, z.B. mittels folgender Schleife

```
newArray =  (int*)malloc(NEWSIZE*sizeof(int));
for (i = 0; i < OLDSIZE; i++) {
  newArray[i] = oldArray[i];
}
```

Danach kann der Speicher des alten Arrays freigegeben werden

```
free(oldArray)
```

Beispiel: statischer Array

```
#include <stdio.h>

int main (void) {
    int n, i, numbers[100];
    printf("Anzahl Zahlen: ");
    scanf("%d", &n);

    for (i = 0; i < n; i++) {
        printf("%d. Zahl: ", (i + 1));
        scanf("%d", &numbers[i]);
    }

    printf("Die Zahlen sind:");
    for (i = 0; i < n; i++) {
        printf(" %d", numbers[i]);
    }
    printf("\n");
}
```

Beispiel: dynamischer Array

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

int main (void) {
    int n, i, *numbers;
    printf("Anzahl Zahlen: ");
    scanf("%d", &n);
    numbers = (int *) malloc(n*sizeof(int));
    for (i = 0; i < n; i++) {
        printf("%d. Zahl: ", (i + 1));
        scanf("%d", &numbers[i]);
    }
    printf("Die Zahlen sind:");
    for (i = 0; i < n; i++) {
        printf(" %d", numbers[i]);
    }
    printf("\n");
    // eigentlich nicht mehr nötig, da Ende Programm
    free(numbers);
}
```

Struct

- Auch Structs können zur Laufzeit alloziert werden

```
typedef stuct {
   int h, min, sec;
} Zeit

Zeit *pZeit;
pZeit = (Zeit*) malloc(sizeof(Zeit));
pZeit->h = 4;
...
free(pZeit); /* Speicher wird freigegeben */
```

Zu früh freigegeben

- Der Speicher wird dann wieder verwenden (bei einem der nächsten Speicheranforderungen).
- Lesen (oder gar Schreiben) über einen diesen Pointer führt zu "Dangling Pointer" Fehler
- Trick: Nach dem free den Pointer zu NULL zu setzen (kontrollierter Absturz bei Fehlern)

```
free(ptr);
*ptr = 6;
```

Falsche Grösse angegeben

- Bei einem Array wird die Bereichsüberschreitung nicht überprüft

```
int *pwerte;
n = 90;
pwerte = (int *) malloc(n * sizeof(int));
/* und so erfolgt der Zugriff: */
pwerte[95] = 12;
```

- Statt Grösse des Structs wird Grösse des Pointers auf das Struct angegeben

```
Datum* pDatum = (Datum*)malloc(sizeof(Datum*));
```

outlook

Zwei und mehrdimensionale Arrays

Einführung

outlook

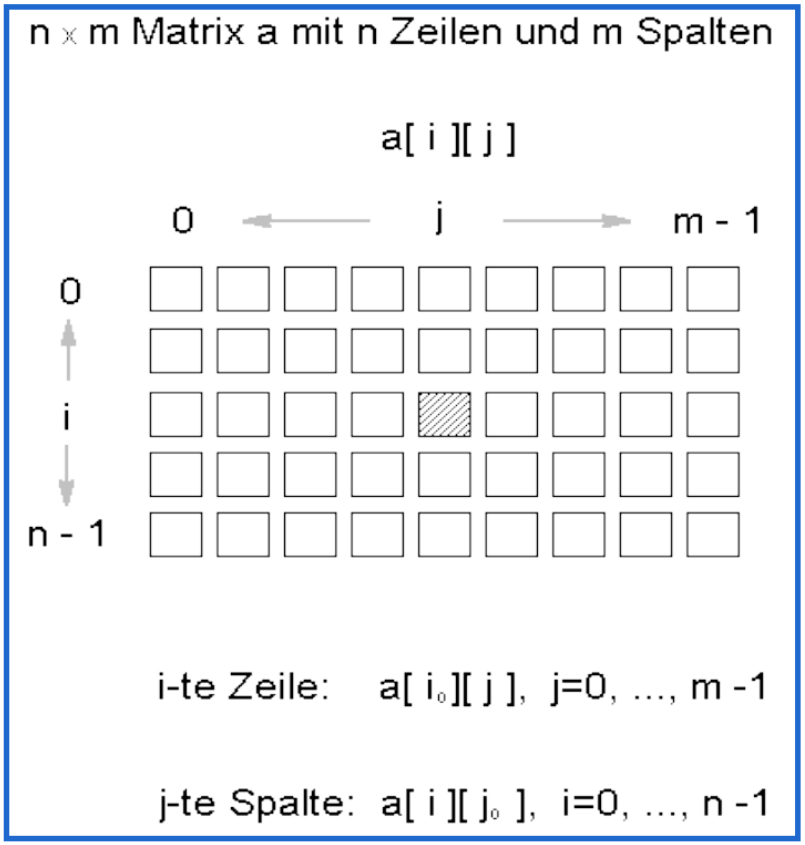
Tag / Filiale	Montag	Diens-tag	Mit-woch	Donner-stag	Freitag	Samstag	Sonntag
Zürich	76	68	85	65	43	100	5
Bern	41	33	36	35	28	49	0
Basel	10	21	13	14	17	15	0

- **Wöchentliche Umsatzzahlen eines Geschäftes** der Filialen Zürich, Bern und Basel
  → Als Tabelle darstellen:
  - Jede Zeile entspricht den Umsatzzahlen einer Filiale pro Woche
  - Jede Spalte entspricht den Umsatzzahlen für einen Tag in den verschiedenen Filialen
  - Gemeinsames Merkmal der Zahlen dieser Tabelle: Alle Einträge sind vom gleichen Datentyp.

- 2-Dimensionale Arrays werden häufig gebraucht für: Tabellen allgemein, Matrizenrechnung, Darstellung von Bildern, Schachbrett, Kinoreservation, etc.

2D Array in C

outlook



- 2D Array für die Umsatzzahlen
  Zeilen: n = 3
  Spalten: m = 7

```
int umsatz[3][7];
```

outlook

Deklaration und Erzeugung, Aufbau

Deklaration und Erzeugung von 2D Arrays

outlook

Deklaration:

- allgemein:

```
Datentyp Arrayname [3][7];
double umsatz[3][7];
```

  - Datentyp: Datentyp der einzelnen Elemente

Statische Erzeugung des Arrays (Grösse bestimmt):

```
Arrayname = Datentyp [AnzahlZeilen][AnzahlSpalten];
```

Dynamisch Erzeugung des Arrays

```
Datentyp Arrayname feld[][7] = (datentyp*) malloc(<size>);
double umsatz[][7] = (double*)malloc(3 * 7 * sizeof(double));
```

2-d Array: Initialisierung

outlook

```
int buffer[2][10] = { {1,2,3,4,5,6,7,8,9,10},
                      {11,12,13,14,15,16,17,18,19,20 }
                    };
```

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20

- Das erste Element buffer[0] ist ein Array von 10 int
- Im Speicher sind die Zahlen aber einfach sequentiell angeordnet

2-d Array: Initialisierung

outlook

- Beispiel
  - drei int-Arrays (Zeilen) mit 5 Elemente (Spalten)
  - erstes Element (**`matrix[0]`**) → Array **`{0, 1, 2, 3, 4}`**

```
int main(void) {
   int matrix[3][5] = { {  0,  1,  2,  3,  4},
                        { 10, 11, 12, 13, 14},
                        { 20, 21, 22, 23, 24} };
   printf("%d\n", matrix[1][2]);
   return 0;
}
```

  - Intialisierung so → erste Dimension kann weggelassen werden

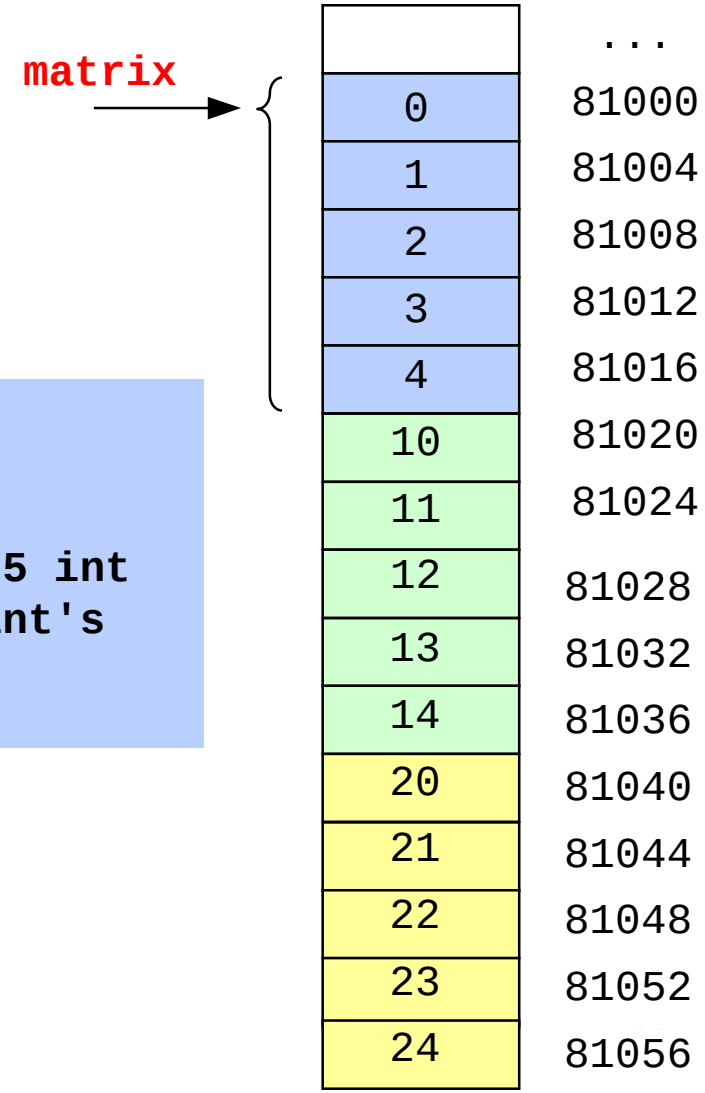
```
int matrix[][5] = { {  0,  1,  2,  3,  4},
                    { 10, 11, 12, 13, 14},
                    { 20, 21, 22, 23, 24} };
```

2-d Array: Speicheranordnung

outlook

- Elemente sequentiell in Speicher angeordnet
  - zeilenweise
  - **matrix** → Zeiger auf Array mit 5 **int**
    - *Typ:* ***int (*)[5]***
  - Anmerkung
    - *bei 1-d Array wäre das* ***int ****

```
int matrix[3][5] = {{  0,  1,  2,  3,  4},
                    { 10, 11, 12, 13, 14},
                    { 20, 21, 22, 23, 24}};
int (*piPtr)[5];  // pointer auf Array mit 5 int
int (*pPtr)[];   // pointer auf Array mit int's
piPtr = pPtr = matrix;
```



2-d Array: Verarbeitung

outlook

- Beispiel: Ausgabe Elemente
  - i.d.R. Zeile dann Spalten

```
int main(void) {
    const int ROWS = 3, COLS = 5;
    int matrix[ROWS][COLS] = { {  0,  1,  2,  3,  4},
                               { 10, 11, 12, 13, 14},
                               { 20, 21, 22, 23, 24} };

    int row, col;
    for (row = 0; row < ROWS; row++) {
        for (col = 0; col < COLS; col++) {
            printf("%6d", matrix[row][col]);
        }
        printf("\n");
    }
    return 0;
}
```

Mehrdimensionale Arrays als Parameter

2-d Array Funktionsparameter

outlook

- Funktionsdeklaration (Prototyp)
  - Array-Schreibweise mit und ohne 1. Dimension

```
double computeMean(int a2d[3][5], int rows, int cols);
double computeMean(int a2d[][5],  int rows, int cols);
```

  - Zeigerschreibweise

```
double computeMean(int (*a2d)[5],  int rows, int cols);
```

- Aufruf

```
mittelwert = computeMean(matrix, ROWS, COLS);
```

- Zugriff in Funktion: `a2d[r][c]`

Zweidimensionale Arrays - Array Schreibweise

outlook

```
mittelwert = computeMean (buffer, ROWS, COLS);

...

double computeMean (int feld[][10], int rows, int cols) {
  int r, c, summe=0;
  for(r=0; r<rows; r++) {
    for(c=0; c<cols; c++) {
      summe = summe + feld[r][c];
    }
  }
  return (double)summe/(rows * cols);
}
```

- Erste Dimension wird weggelassen

Zweidimensionale Arrays - Pointer Schreibweise

outlook

```
mittelwert = computeMean (buffer, ROWS, COLS);

...

double computeMean (int *feld[10], int rows, int cols) {
  int r, c, summe=0;
  for(r=0; r<rows; r++) {
    for(c=0; c<cols; c++) {
      summe = summe + feld[r][c];
    }
  }
  return (double)summe/(rows * cols);
}
```

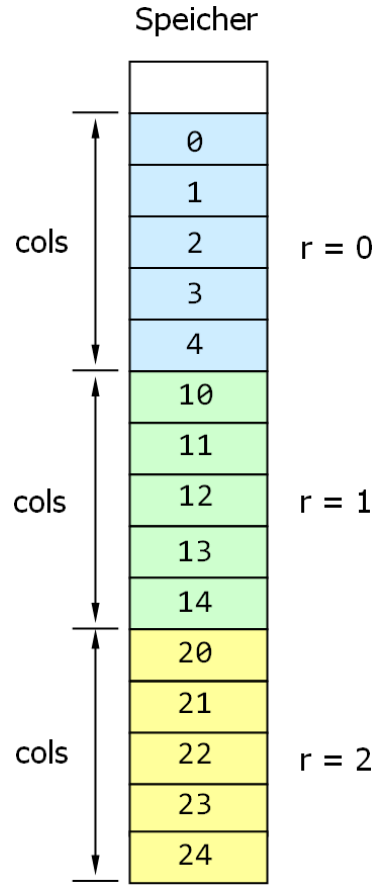
- Parameter in Zeiger-Schreibweise: Zeiger auf ein Array mit 10 int

Zweidimensionale Arrays als 1-d Pointer

outlook

- 1-d Array mit row/colunm Zugriff

```
double computeMean(int *a1d,int rows,int cols)
{
    int r, c, sum = 0;
    for (r = 0; r < rows; r++) {
        for (c = 0; c < cols; c++) {
            sum = sum + a1d[r*cols + c];
        }
    }
    return (double)sum/((double)(rows*cols));
}
```



Zweidimensionale Arrays als 1-d Pointer

outlook

- 2-d Array für Verarbeitung als 1D Pointer

```
double computeMean(int *a1d, int len) {
    int i, sum = 0;
    for (i = 0; i < len; i++) {
            sum += a1d[i];
    }
    return (double)sum/((double)len);
}

int main(void) {
    const int ROWS = 3;
    const int COLS = 5;
    int matrix[3][5] = { {  0,  1,  2,  3,  4},
                         { 10, 11, 12, 13, 14},
                         { 20, 21, 22, 23, 24} };
    double mean;
    mean = computeMean((int *)matrix, ROWS*COLS);
    printf("Mittelwert: %f\n", mean);
    return 0;
}
```

outlook

Array von Strings

Array von Strings

outlook

- Beispiel: 2 Varianten

```
int main(void) {
    char *month[12] = {"January", "February", "March",
                       "April", "May", "June", "July",
                       "August", "September", "October",
                       "November", "December"};
    int i;
    for (i = 0; i < 12; i++) {
        printf("%s\n", month[i]);
    }
    return 0;
}
```

```
int main(void) {
    char month[12][10] = {"January", "February", "March",
                          "April", "May", "June", "July",
                          "August", "September", "October",
                          "November", "December"};
    int i;
    for (i = 0; i < 12; i++) {
        printf("%s\n", month[i]);
    }
    return 0;
}
```

Unterschied ?

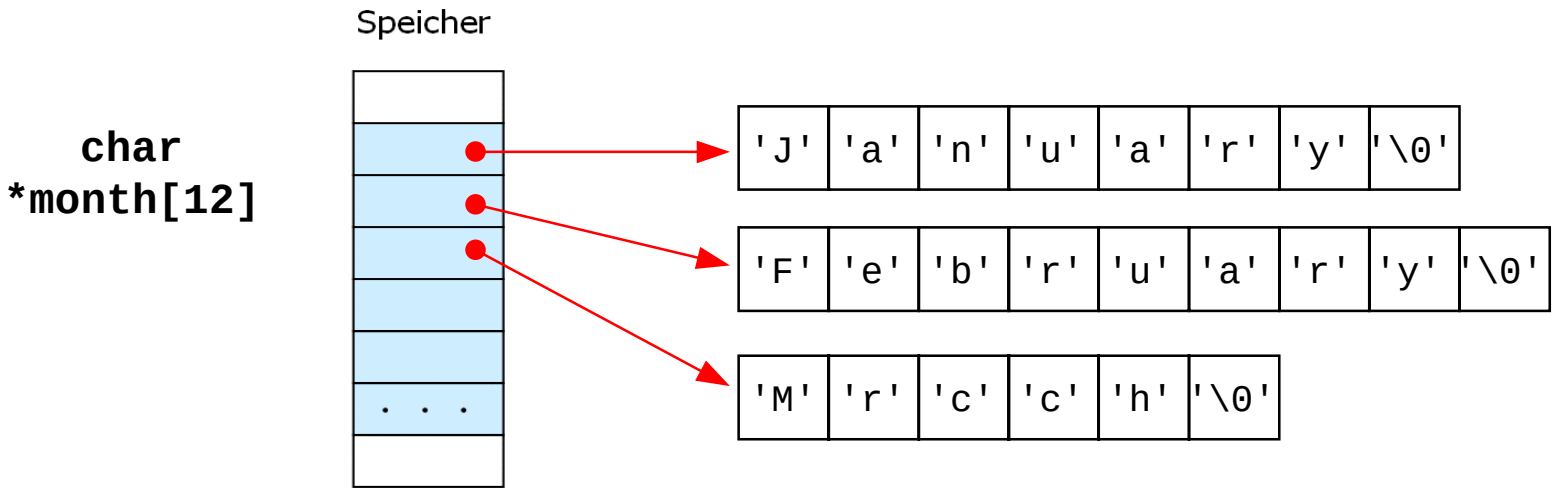
Speicherbedarf ?

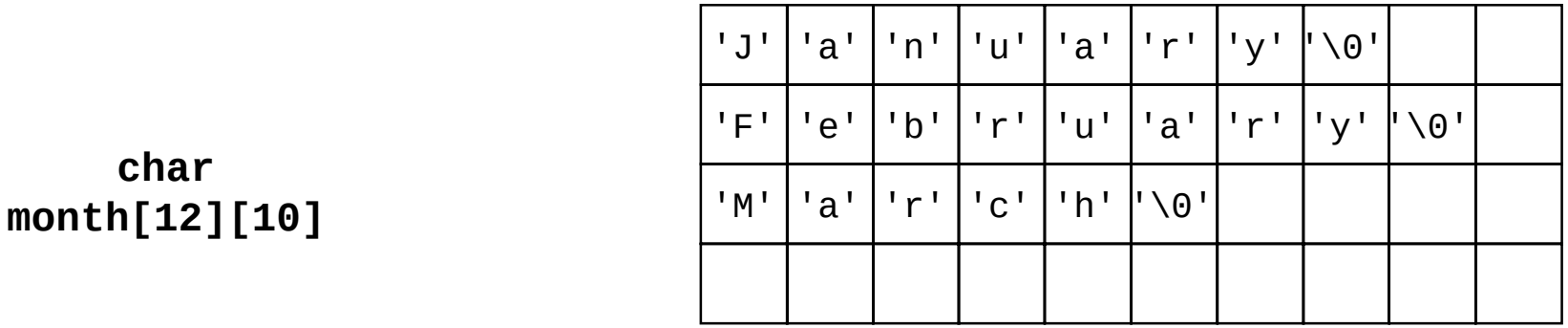
Wann welche Variante ?

... Array von Strings

outlook

- ... die 2 Varianten





... Array von Strings in Main Funktion

outlook

- Die Argumente, die dem Programm auf der Kommandezeile übergeben werden, werden als Array von Strings übergeben
- Beispiel

```
int main(int argc, char *argv[]) {
  if (argc == 2) {
   printf("%s %s",argv[0], argv[1]);
  }
  return 0;
}
```

arg[0] Name des Programms, z.B. Hello.exe

arg[1] erstes Argument

```
> hello World
hello.exe World
```

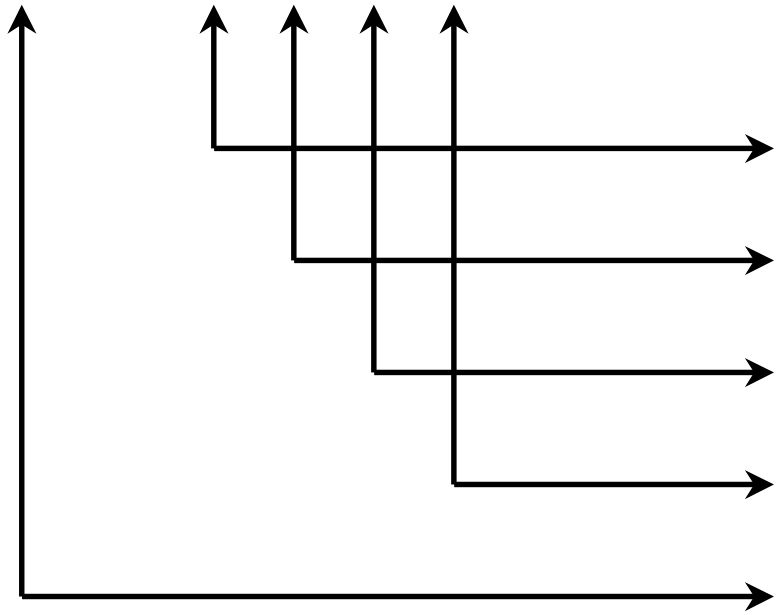
outlook

Dreidimensionale Arrays

Dreidimensionale Arrays

outlook

```
Double  c[4][5][6];
```



c ist ein

Array mit 4 Elementen vom Typ

Array mit 5 Elementen vom Typ

Array mit 6 Elementen vom Typ

double

Zusammenfassung

- 2-d Arrays
  - Elemente zeilenweise im Speicher abgelegt
  - Typendeklaration: verschiedene Möglichkeiten
  - kann für Verarbeitung in 1-d Array umgewandelt werden
  - Funktionsparameter : eine Dimension kann weggelassen werden

- Array mit Strings
  - 2 Möglichkeiten
    - *1-d Array mit Zeigern auf Strings*
    - *2-d char-Array*

Noch Fragen?