Pointer in C

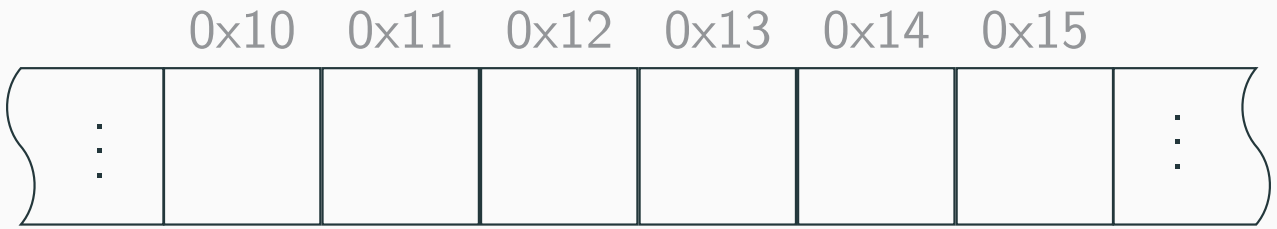
Florian Bünger

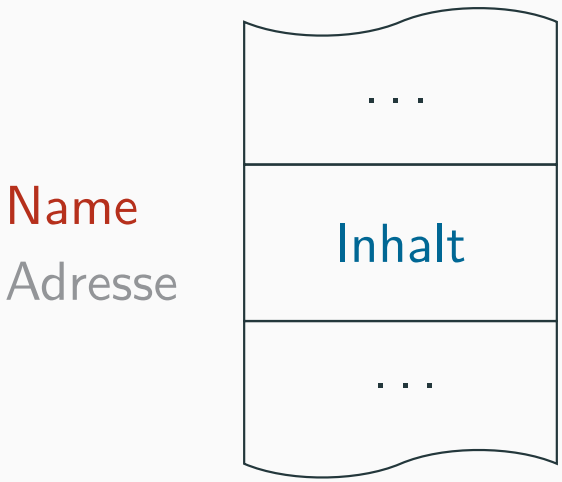
16.11.2021

Institut für Zuverlässiges Rechnen

Das Speichermodell

Das Speichermodell





Unterteilung in 1 Byte (=8 Bit) Blöcke.

Jeder Block hat:

- einen Inhalt
- eine Adresse (implizit durch Position bestimmt)
- einen Namen (leichter zu merken)

Statische Arrays

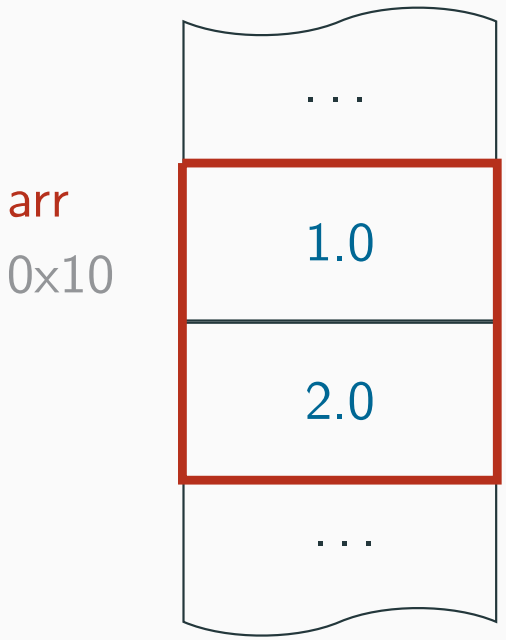
Deklaration und Initialisierung


```
1  // Methode 1
2  int arr[] = {10,20,30};
3
4  // Methode 2
5  int arr[3];
6  arr[0] = 10;
7  arr[1] = 20;
8  arr[2] = 30;
9
10 // Methode 3
11 int arr[3];
12 for (int i = 0; i < 3; i++) {
13   arr[i] = (i + 1) * 10;
14 }
```

Zugriff auf Inhalt und Adresse

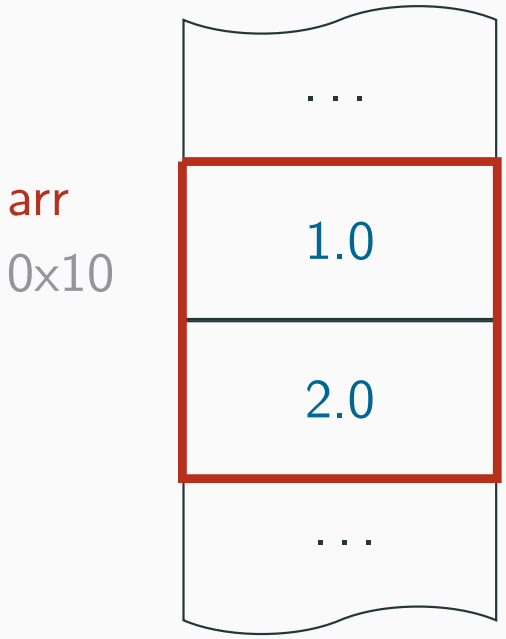

```c
int arr[] = {10,20,30};

// Inhalt
printf("%d", arr[0]); // 10
printf("%d", arr[2]); // 30

// Adresse
printf("%p", &arr[0]); // 0x10
printf("%p", &arr[2]); // 0x18
```

Statische Arrays - Quiz



- Deklarieren Sie ein Array, das die `double`-Werte 1.0 und 2.0 speichert.

Statische Arrays - Quiz

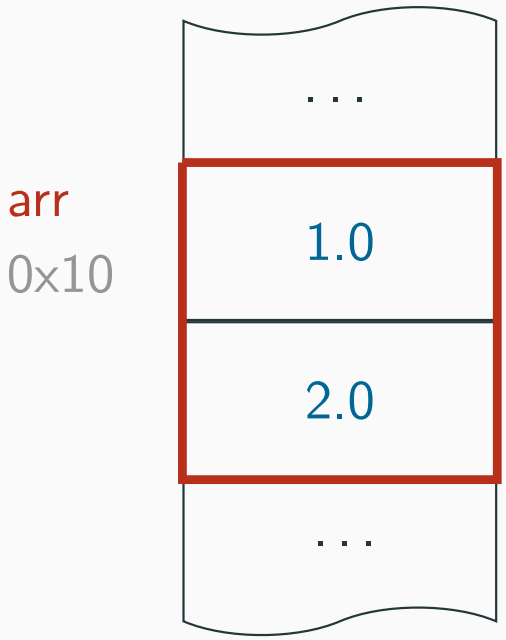


- Deklarieren Sie ein Array, das die `double`-Werte 1.0 und 2.0 speichert.

```
double arr[] = {1.0,2.0};
```

- Wie geben Sie den Inhalt des zweiten Feldes aus?

Statische Arrays - Quiz



- Deklarieren Sie ein Array, das die double-Werte 1.0 und 2.0 speichert.

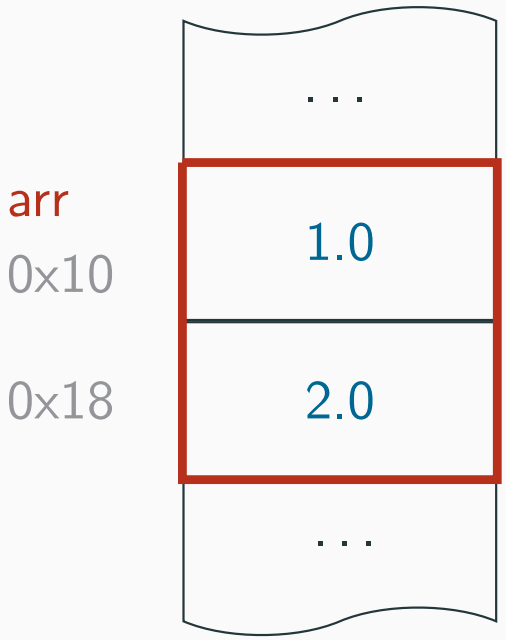
```c
double arr[] = {1.0,2.0};
```

- Wie geben Sie den Inhalt des zweiten Feldes aus?

```c
printf("%f", arr[1]); // 2.0
```

- Wie lautet die Adresse vom zweiten Feld?

Statische Arrays - Quiz



- Deklarieren Sie ein Array, das die double-Werte 1.0 und 2.0 speichert.

```c
double arr[] = {1.0,2.0};
```

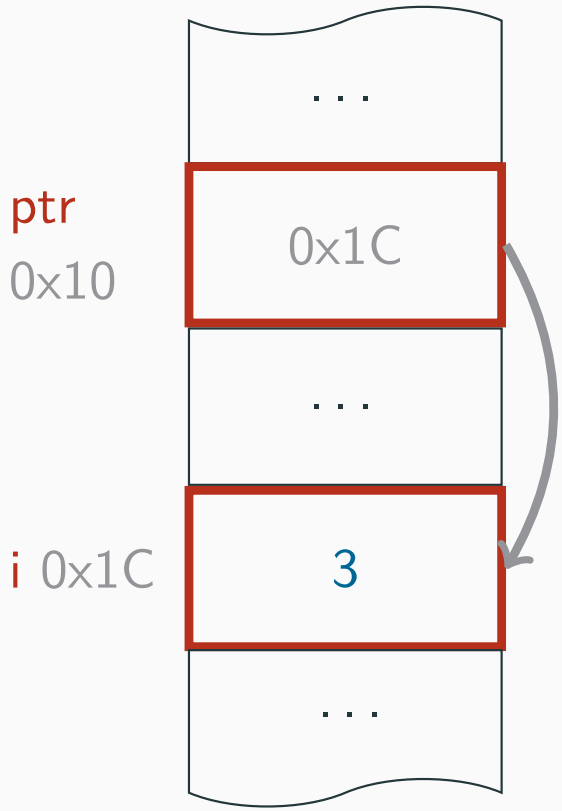
- Wie geben Sie den Inhalt des zweiten Feldes aus?

```c
printf("%f", arr[1]); // 2.0
```

- Wie lautet die Adresse vom zweiten Feld?

Pointer

Deklaration und Initialisierung




```
1  // Deklaration
2  int  i;
3  int* ptr;
4
5  // Initialisierung
6    i =  3;
7  ptr = &i;
8
9  // Inhalt
10 printf("%d",  i  ); // 3
11 printf("%d", *ptr); // 3
12
13 // Adresse
14 printf("%p", &i  ); // 0x1C
15 printf("%p",  ptr); // 0x1C
16 printf("%p", &ptr); // 0x10
```

Zwei wichtige Operatoren

Adressoperator

& „Adresse von“

```
printf("%p", &ptr); // 0x10
printf("%p", &i  ); // 0x1C
```

Indirektionsoperator

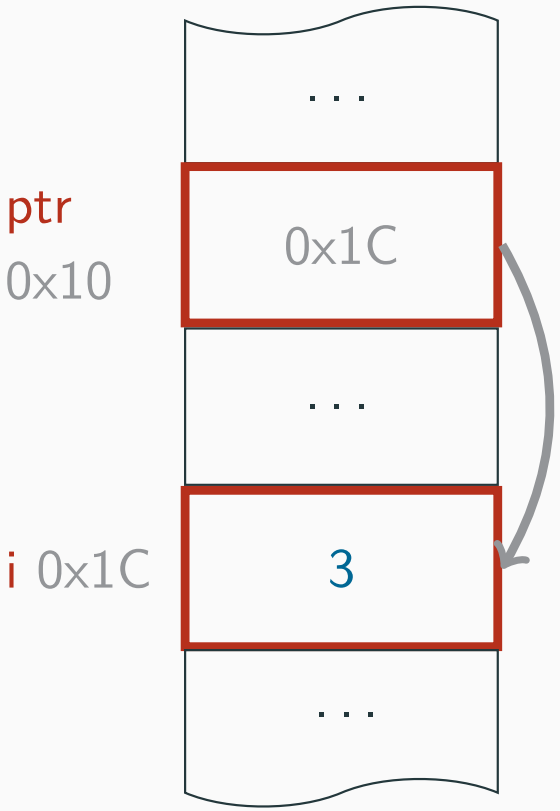
* „Inhalt von“

```
printf("%d", *ptr); // 3
printf("%d", *i  ); // NIEMALS MACHEN!
```

Pointerdeklaration

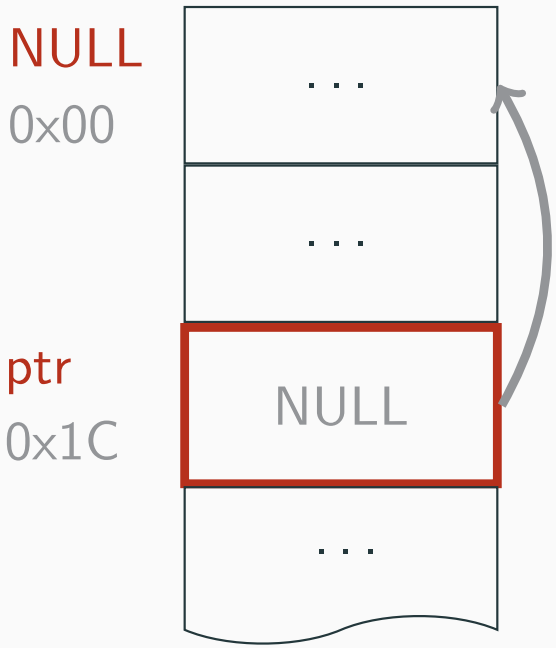
```
int* ptr;
```

Den Inhalt verändern




```c
int   i   = 3;
int* ptr = &i;
printf("%d",  i  ); // 3
printf("%d", *ptr); // 3

i = 4;
printf("%d",  i  ); // 4
printf("%d", *ptr); // 4

*ptr = 5;
printf("%d",  i  ); // 5
printf("%d", *ptr); // 5
```

Was ist NULL?



- NULL ist ein Makro für die Adresse „0x00“.

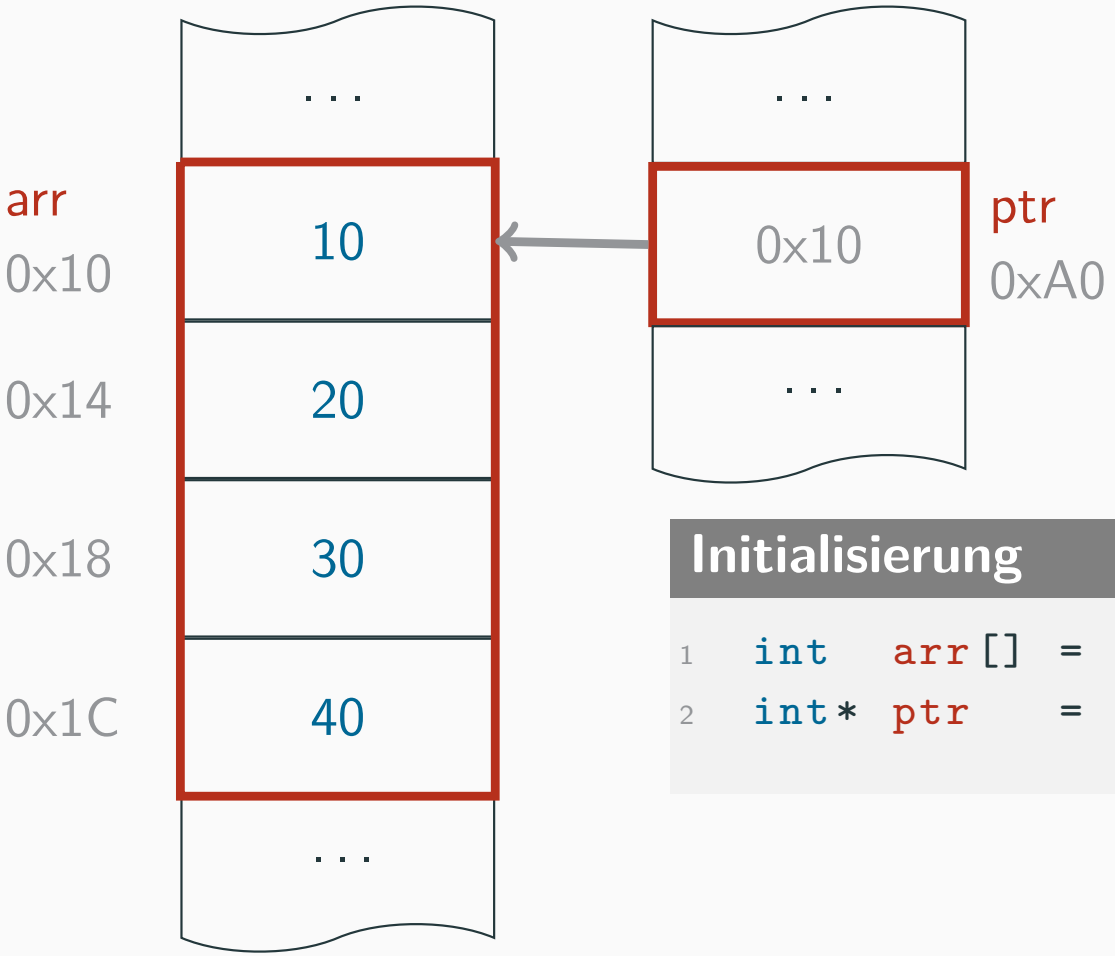
```
#define NULL ((void *)0)
```

- Definiert unbekannten Zustand eines Pointers.

```
int* ptr = NULL;

printf("%d", *ptr);     // Crash!

if (ptr != NULL) {      // Absichern!
  printf("%d", *ptr);
}
```

- Makro wird durch Einbinden von <stdio.h> oder <stdlib.h> bekannt.

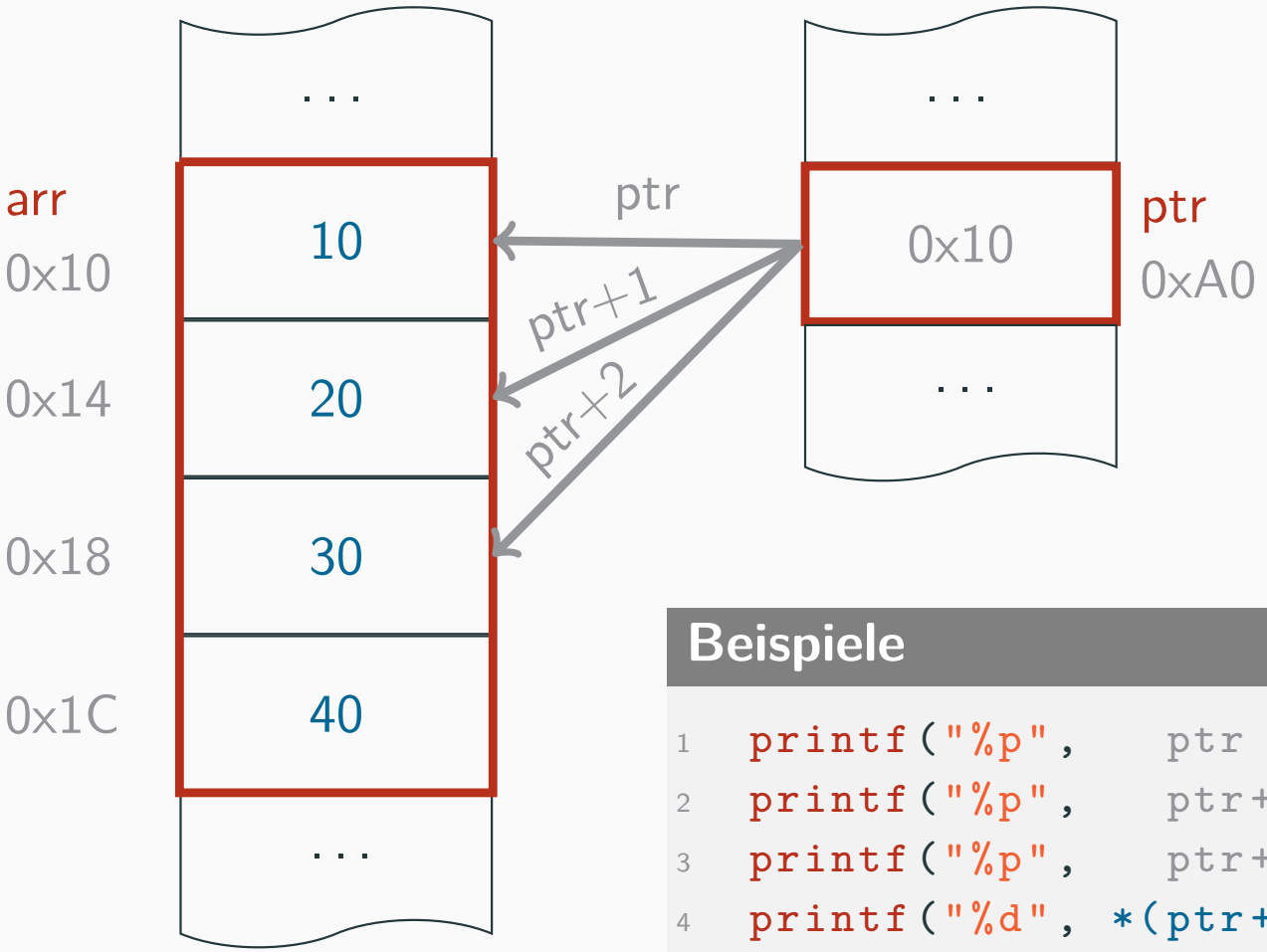
Pointer, die auf Arrays zeigen



Initialisierung

```
1  int  arr[] = {10,20,30,40};
2  int* ptr   = arr; // kein & !!
```

Pointerarithmetik - Mit Pointern rechnen



Beispiele

```
1  printf("%p",    ptr    ); // 0x10
2  printf("%p",    ptr+1  ); // 0x14
3  printf("%p",    ptr+2  ); // 0x18
4  printf("%d", *(ptr+2)); // 30
```

Pointerarithmetik - Mit Pointern rechnen

- `ptr+1` erhöht den Wert der Adresse um `sizeof(int)` = 4 Byte, da `ptr` ein `int`-Pointer ist!
- Adressen ausrechnen, ohne `ptr` zu verändern

```
ptr     ptr+1     ptr-1
```

- Adressen ausrechnen und `ptr` verändern

```
ptr = ptr+1;     ptr++;          ++ptr;
ptr = ptr-1;     ptr--;          --ptr;
ptr = ptr+42;    ptr += 42;
ptr = ptr-42;    ptr -= 42;
```

- Pointer subtrahieren (`ptr1-ptr2`) möglich, aber:
  - Das Ergebnis hängt vom Pointer-Datentyp ab!
  - Das Ergebnis kann negativ sein!

Äquivalenz zwischen Pointern und Arrays

```
int  arr[] = {10,20,30,40};
int* ptr   = arr;
```

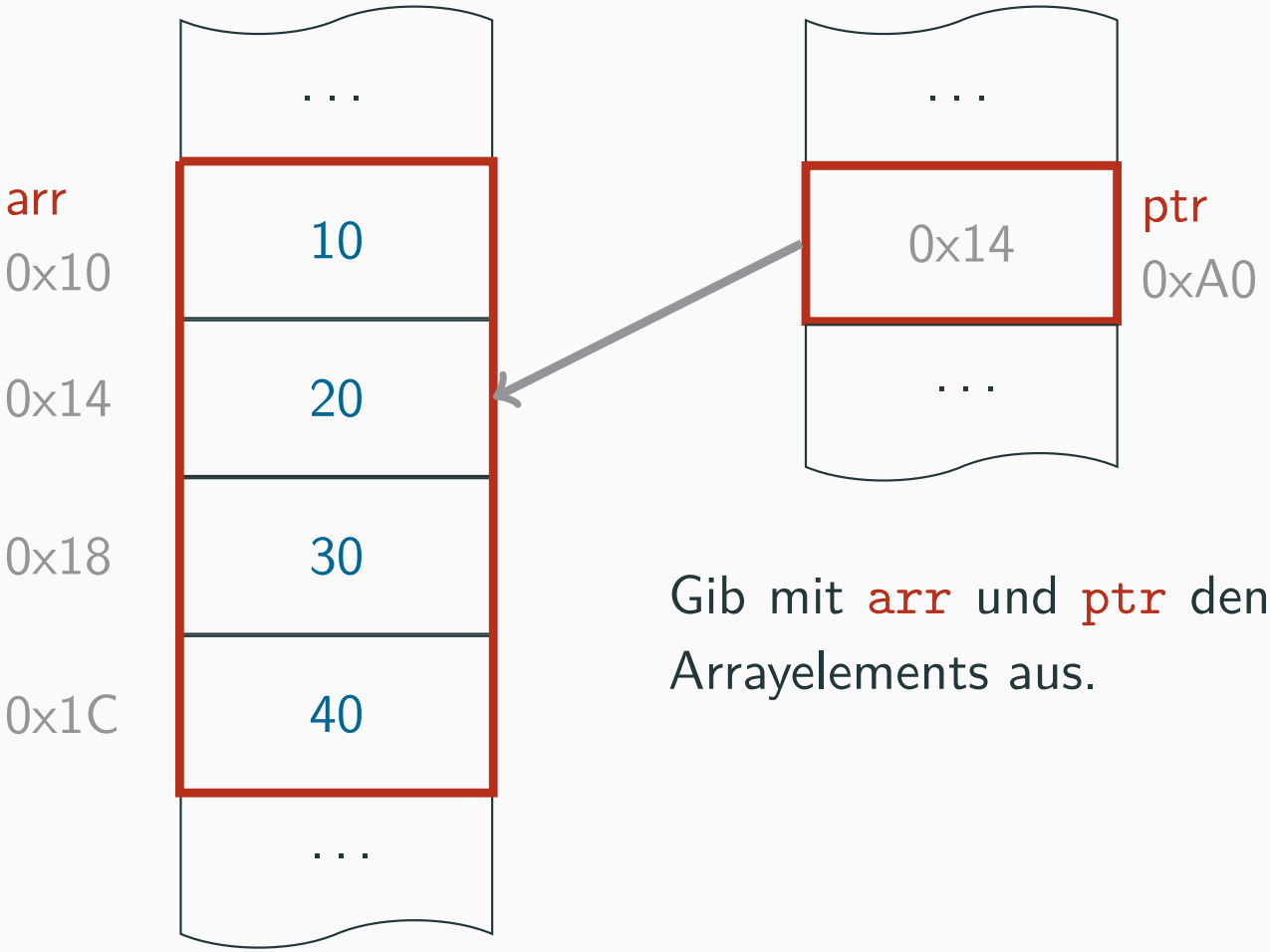
- Adresszugriff, bei `arr` ist die Adresse nicht änderbar!

	Pointer-Variante	Array-Variante
Erste Adresse	`arr`	`&arr[0]`
	`ptr`	`&ptr[0]`
Adresse mit Index n	`arr+n`	`&arr[n]`
	`ptr+n`	`&ptr[n]`

- Inhaltszugriff

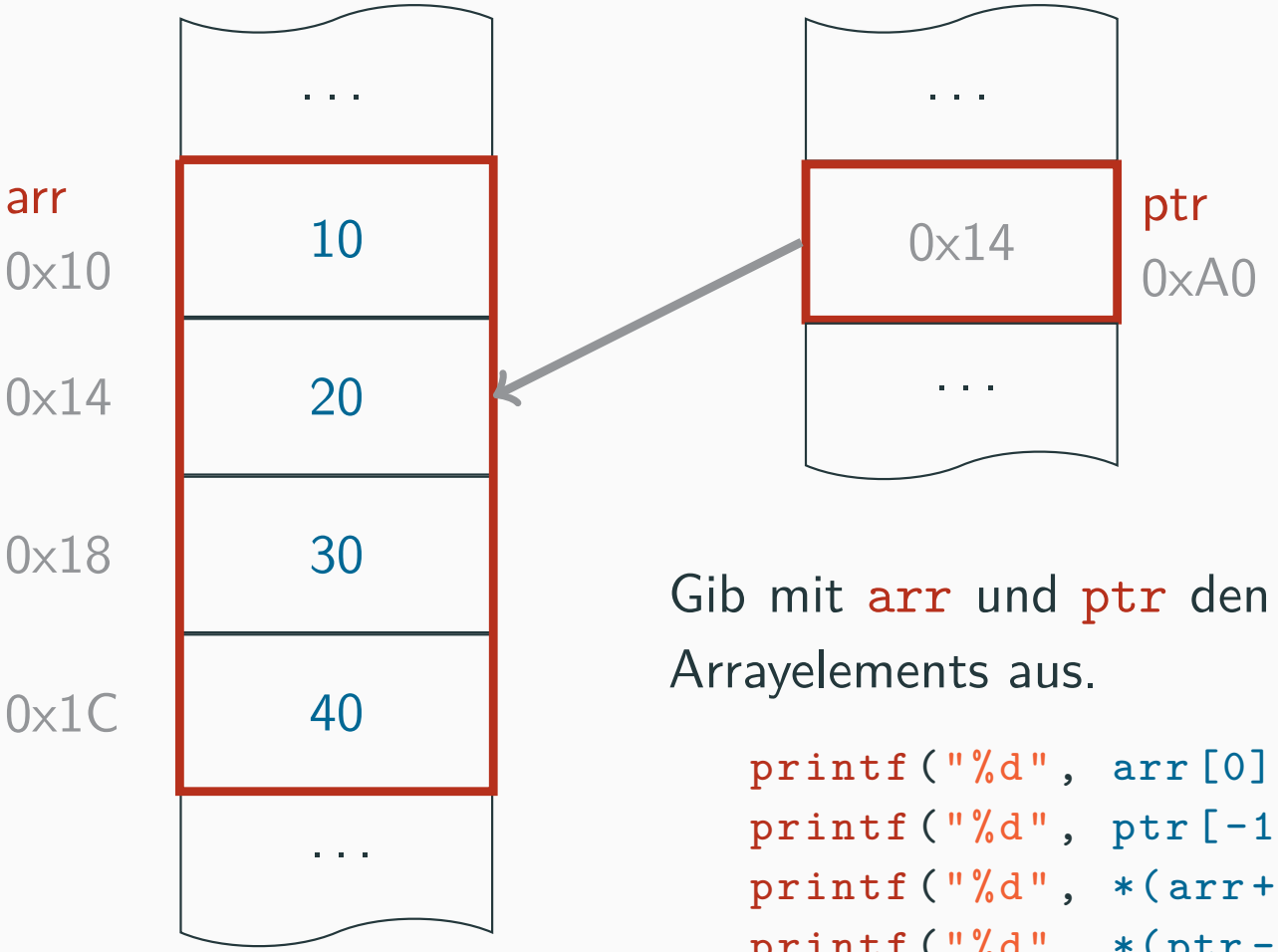
	Pointer-Variante	Array-Variante
Erstes Element	`*arr`	`arr[0]`
	`*ptr`	`ptr[0]`
Element mit Index n	`*(arr+n)`	`arr[n]`
	`*(ptr+n)`	`ptr[n]`

Pointer - Quiz 1



Gib mit `arr` und `ptr` den Inhalt des ersten Arrayelements aus.

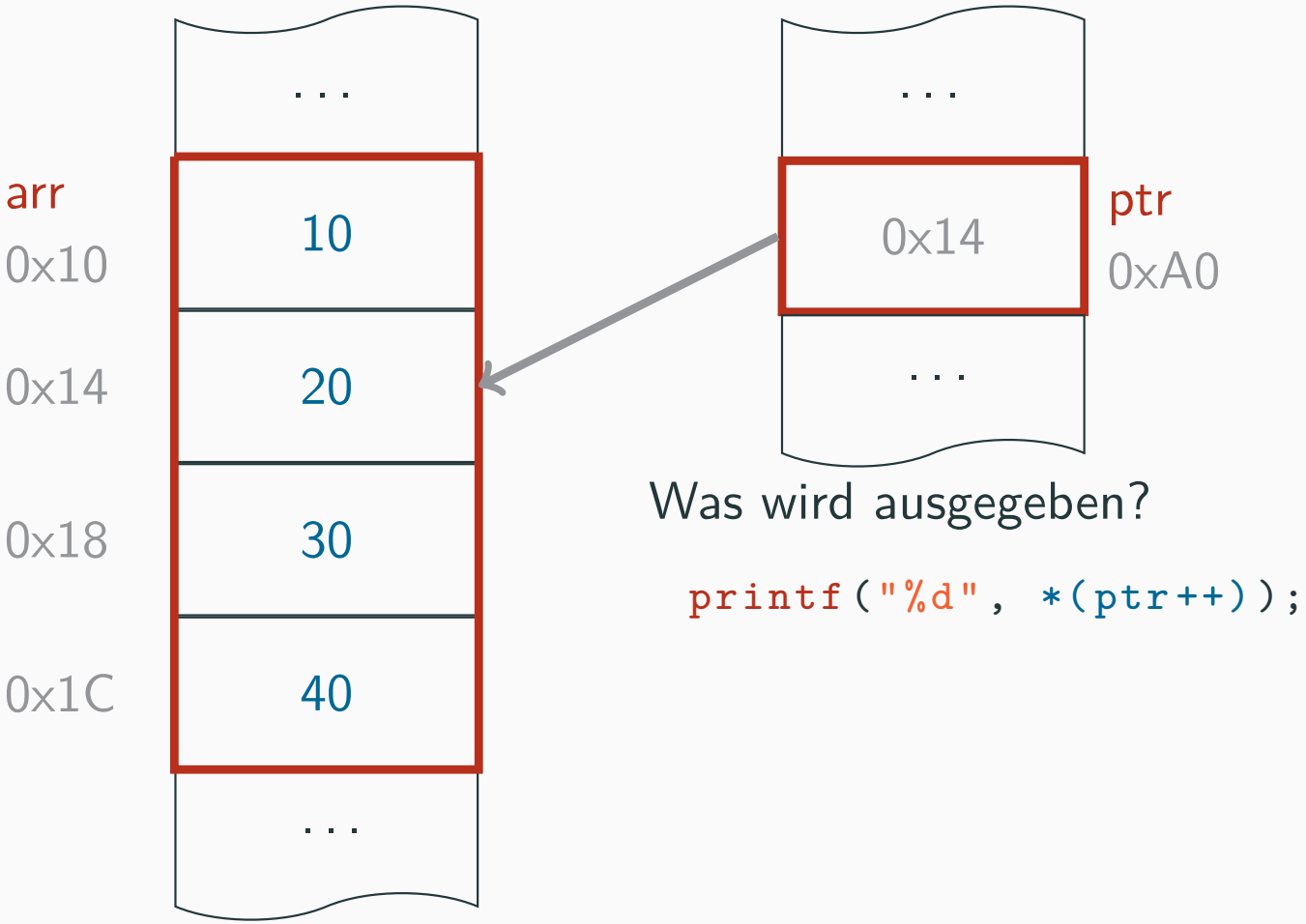
Pointer - Quiz 1



Gib mit `arr` und `ptr` den Inhalt des ersten Arrayelements aus.

```
printf("%d", arr[0]  ); // 10
printf("%d", ptr[-1] ); // 10
printf("%d", *(arr+0)); // 10
printf("%d", *(ptr-1)); // 10
```

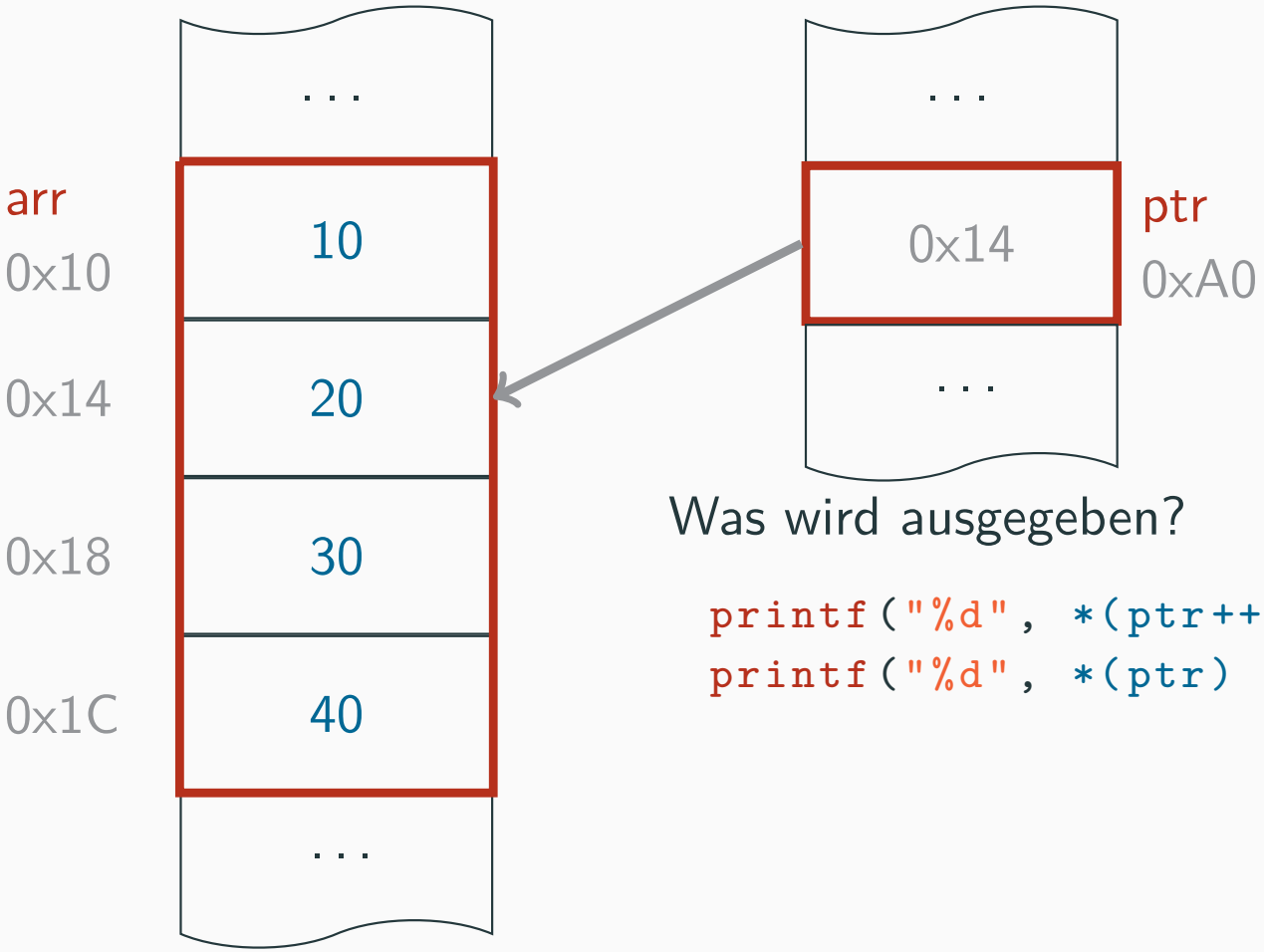
Pointer - Quiz 2



Was wird ausgegeben?

```
printf("%d", *(ptr++));
```

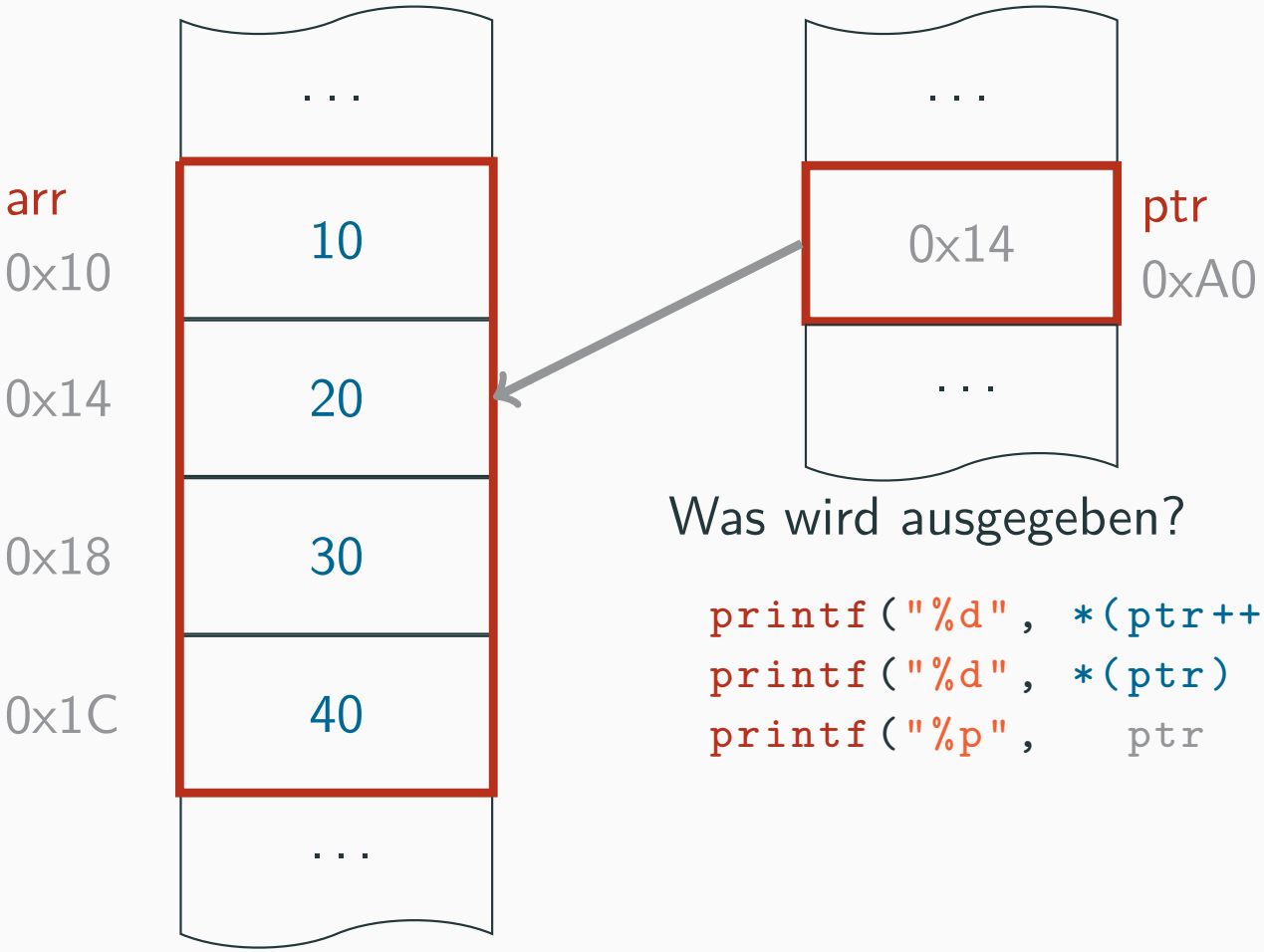
Pointer - Quiz 2



Was wird ausgegeben?

```
printf("%d", *(ptr++)); // 20
printf("%d", *(ptr)   );
```

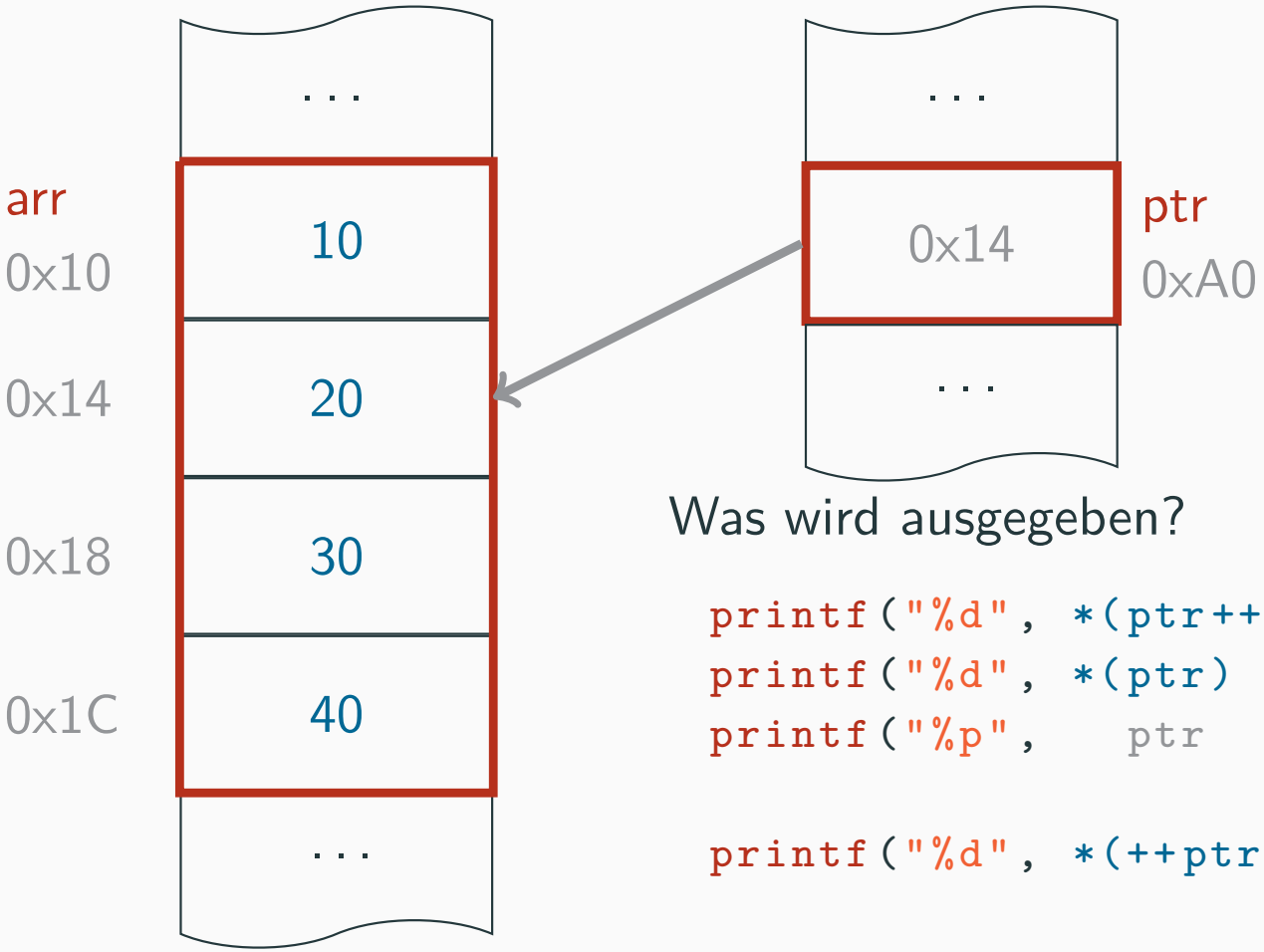
Pointer - Quiz 2



Was wird ausgegeben?

```
printf("%d", *(ptr++)); // 20
printf("%d", *(ptr)   ); // 30
printf("%p",   ptr    );
```

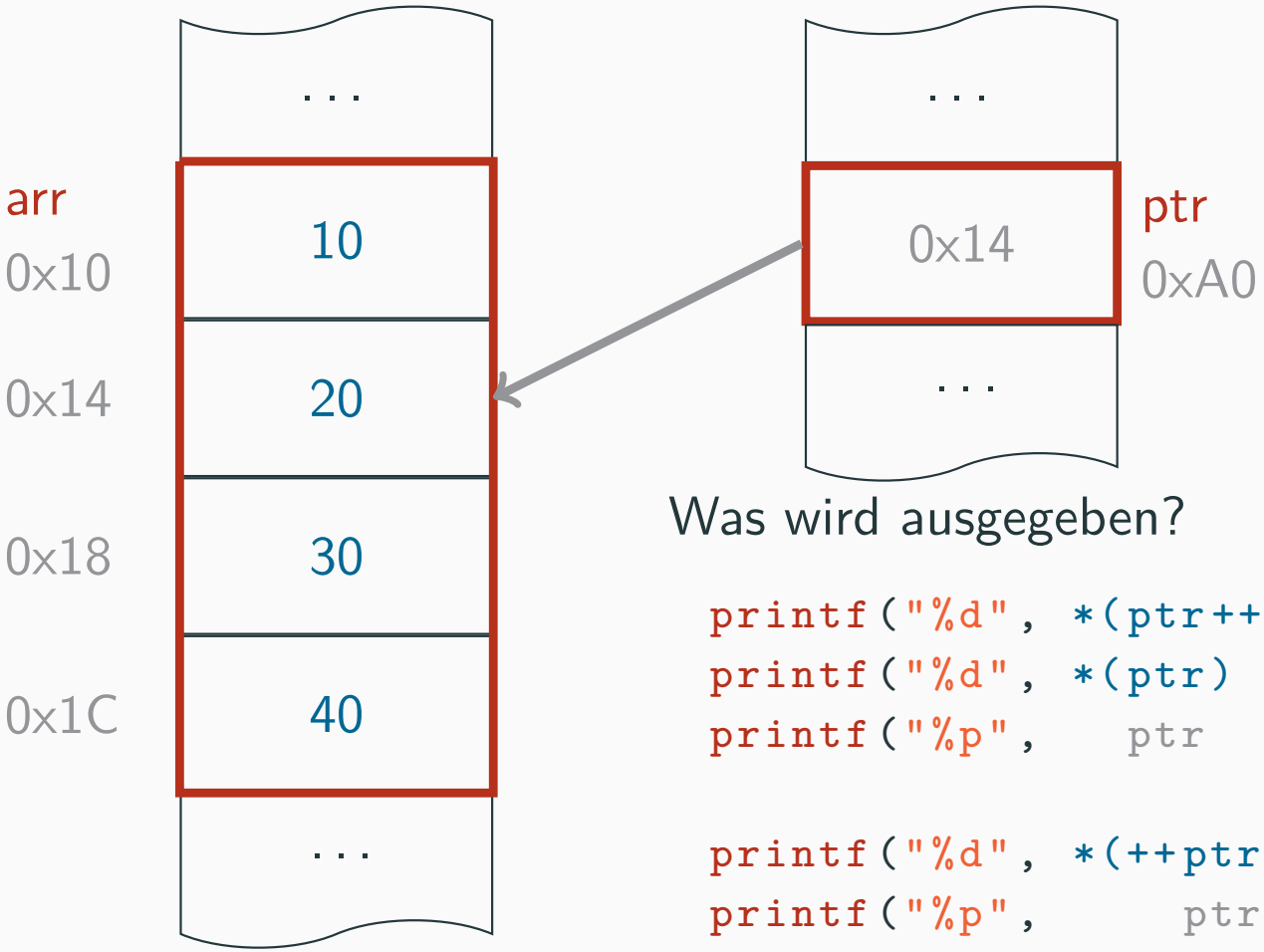
Pointer - Quiz 2



Was wird ausgegeben?

```
printf("%d", *(ptr++)); // 20
printf("%d", *(ptr)   ); // 30
printf("%p",   ptr    ); // 0x18

printf("%d", *(++ptr));
```

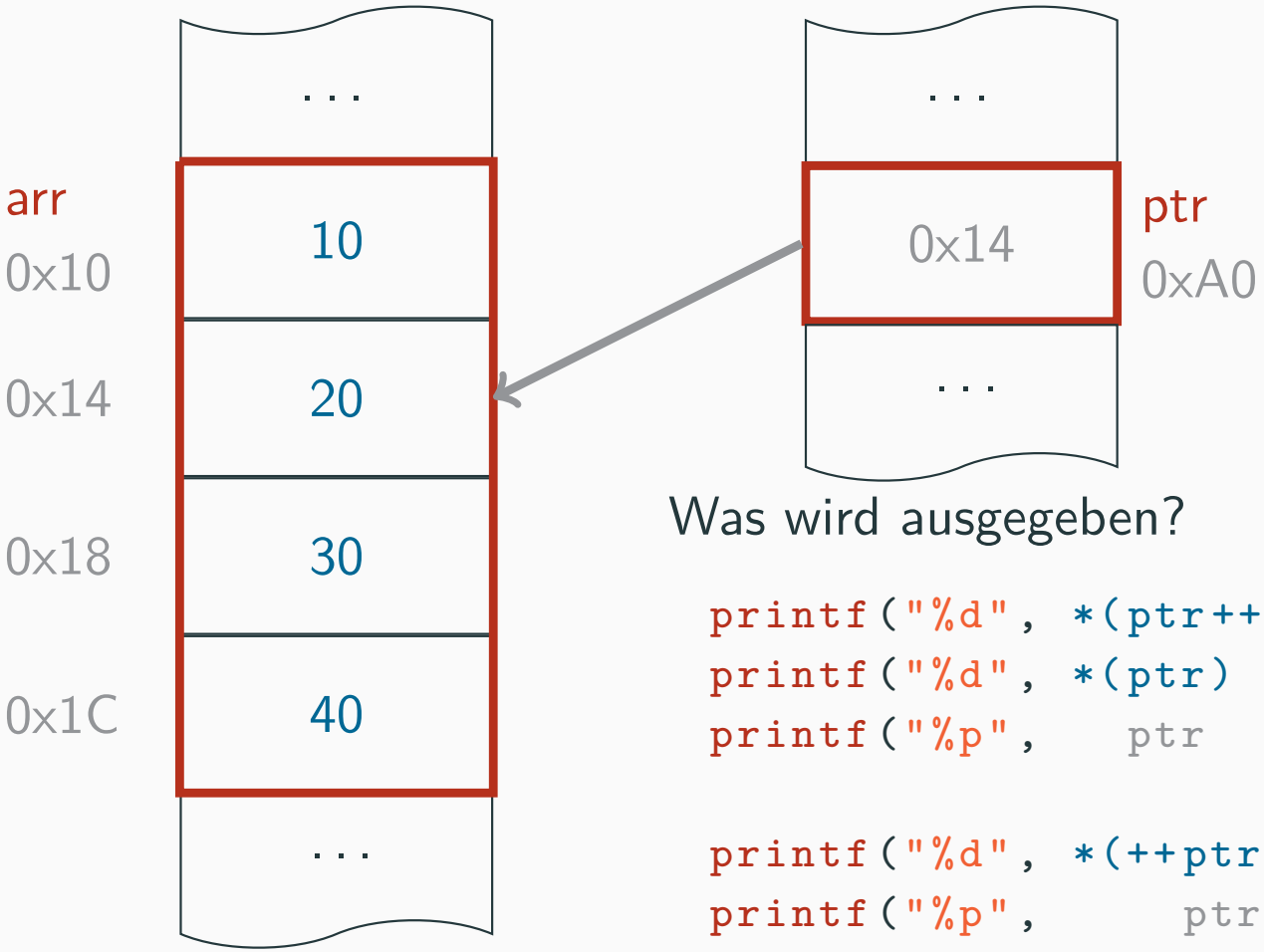
Pointer - Quiz 2



Was wird ausgegeben?

```
printf("%d", *(ptr++)); // 20
printf("%d", *(ptr)   ); // 30
printf("%p",   ptr    ); // 0x18

printf("%d", *(++ptr)); // 40
printf("%p",     ptr );
```

Pointer - Quiz 2

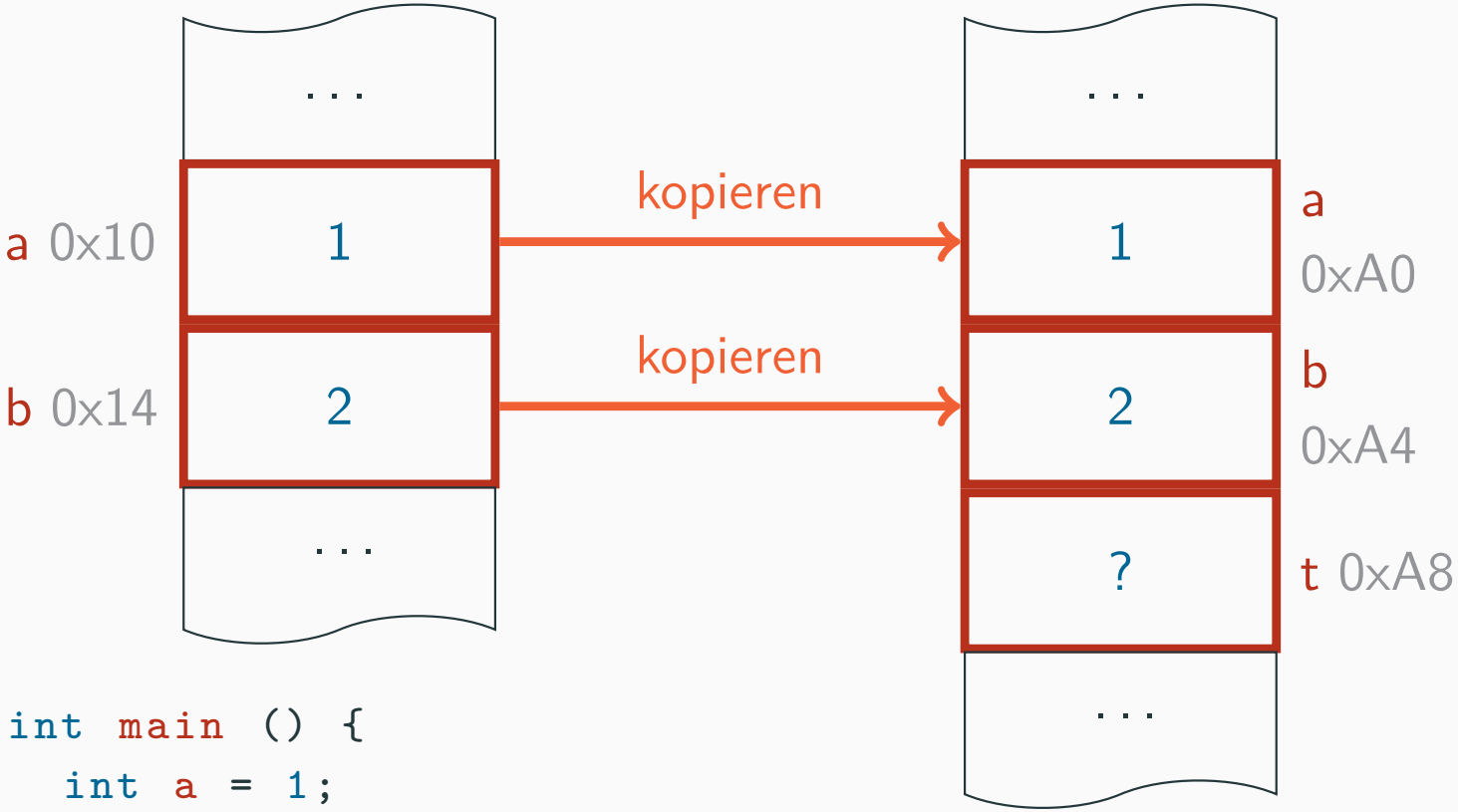


Was wird ausgegeben?

```
printf("%d", *(ptr++)); // 20
printf("%d", *(ptr)  ); // 30
printf("%p",   ptr   ); // 0x18

printf("%d", *(++ptr)); // 40
printf("%p",     ptr ); // 0x1C
```

Call-by-Value und -Reference

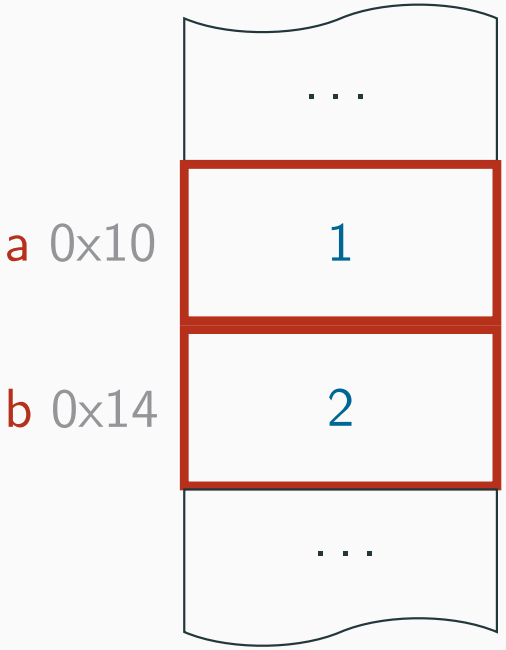
swap mit call-by-value



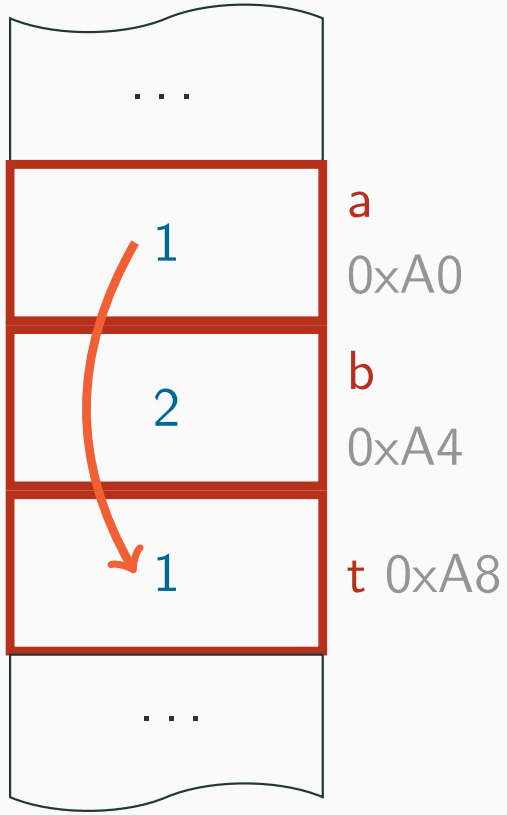

```
int main () {
  int a = 1;
  int b = 2;
  swap(a,b); <--
  return 0;
}
```

```
void swap (int a, int b) {
  int t = a;
  a = b;
  b = t;
}
```

swap mit call-by-value

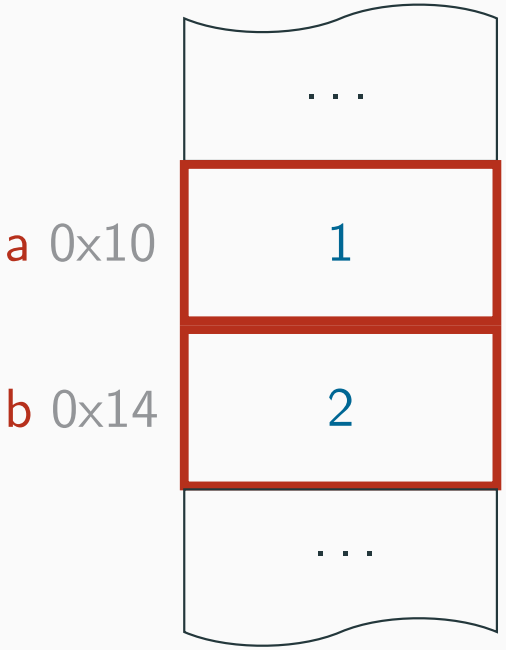



```
int main () {
  int a = 1;
  int b = 2;
  swap(a,b);
  return 0;
}
```

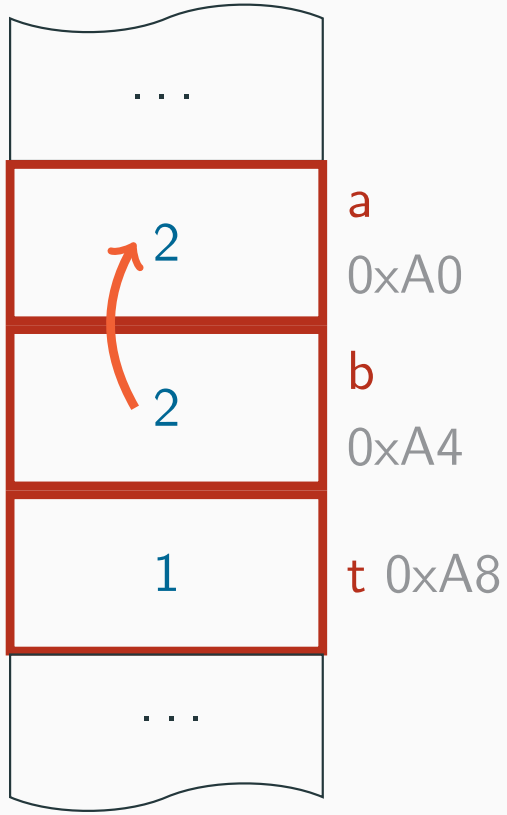



```
void swap (int a, int b) {
  int t = a; <—
  a = b;
  b = t;
}
```

swap mit call-by-value

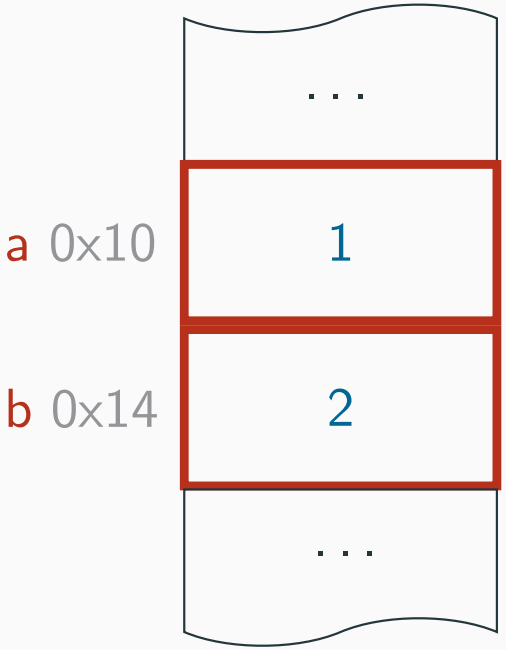



```
int main () {
  int a = 1;
  int b = 2;
  swap(a,b);
  return 0;
}
```

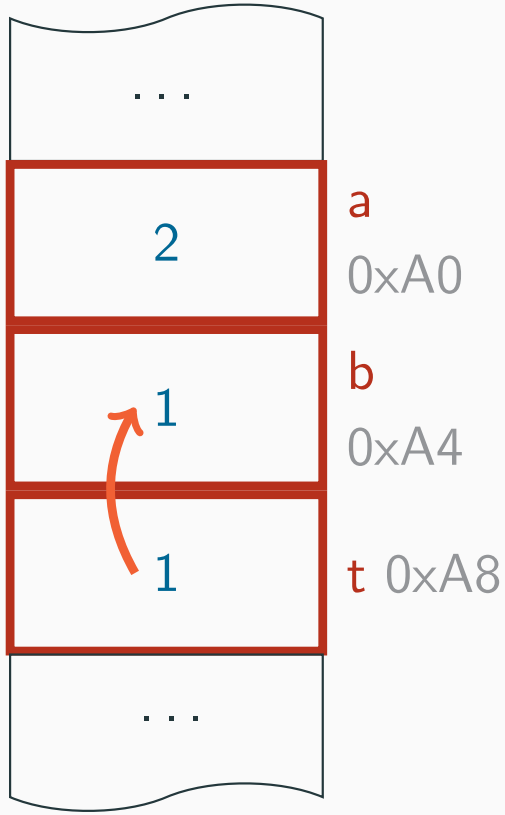



```
void swap (int a, int b) {
  int t = a;
  a = b; <—
  b = t;
}
```

swap mit call-by-value

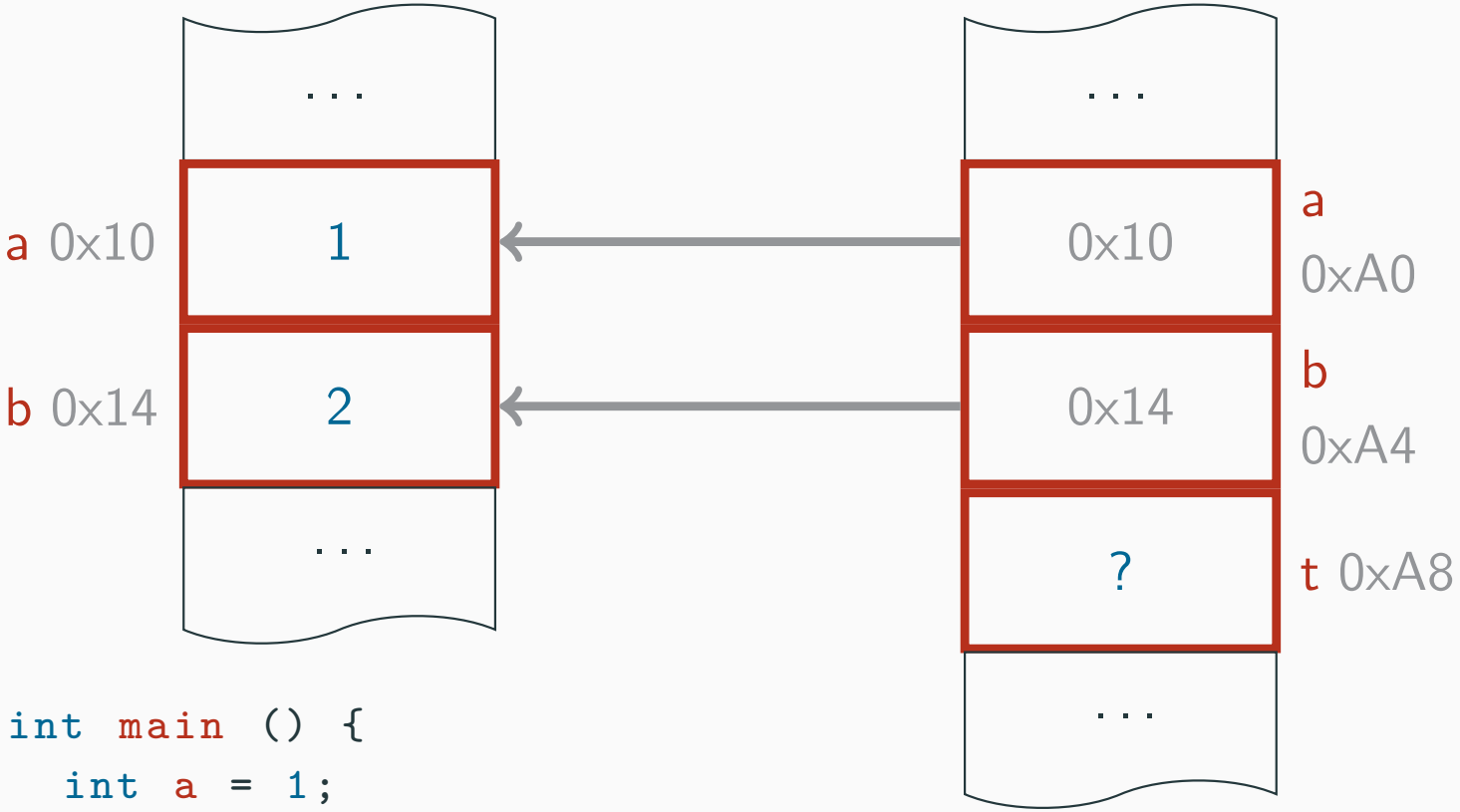



```
int main () {
  int a = 1;
  int b = 2;
  swap(a,b);
  return 0;
} // NIX GETAUSCHT!
```




```
void swap (int a, int b) {
  int t = a;
  a = b;
  b = t; <-- FERTIG
}
```

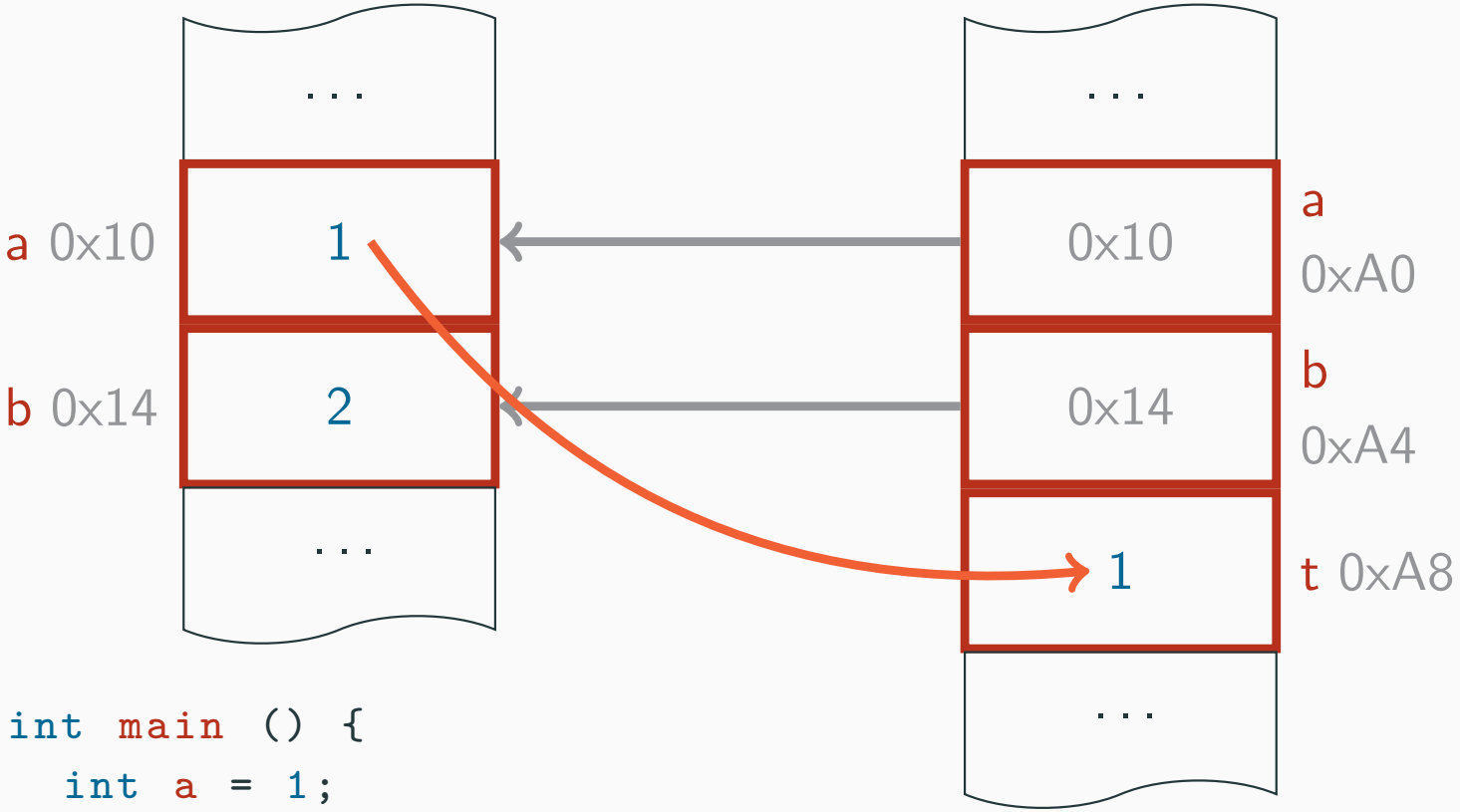
swap mit call-by-reference




```
int main () {
  int a = 1;
  int b = 2;
  swap(&a,&b); <--
  return 0;
}
```

```
void swap (int* a, int* b) {
  int t = *a;
  *a = *b;
  *b = t;
}
```

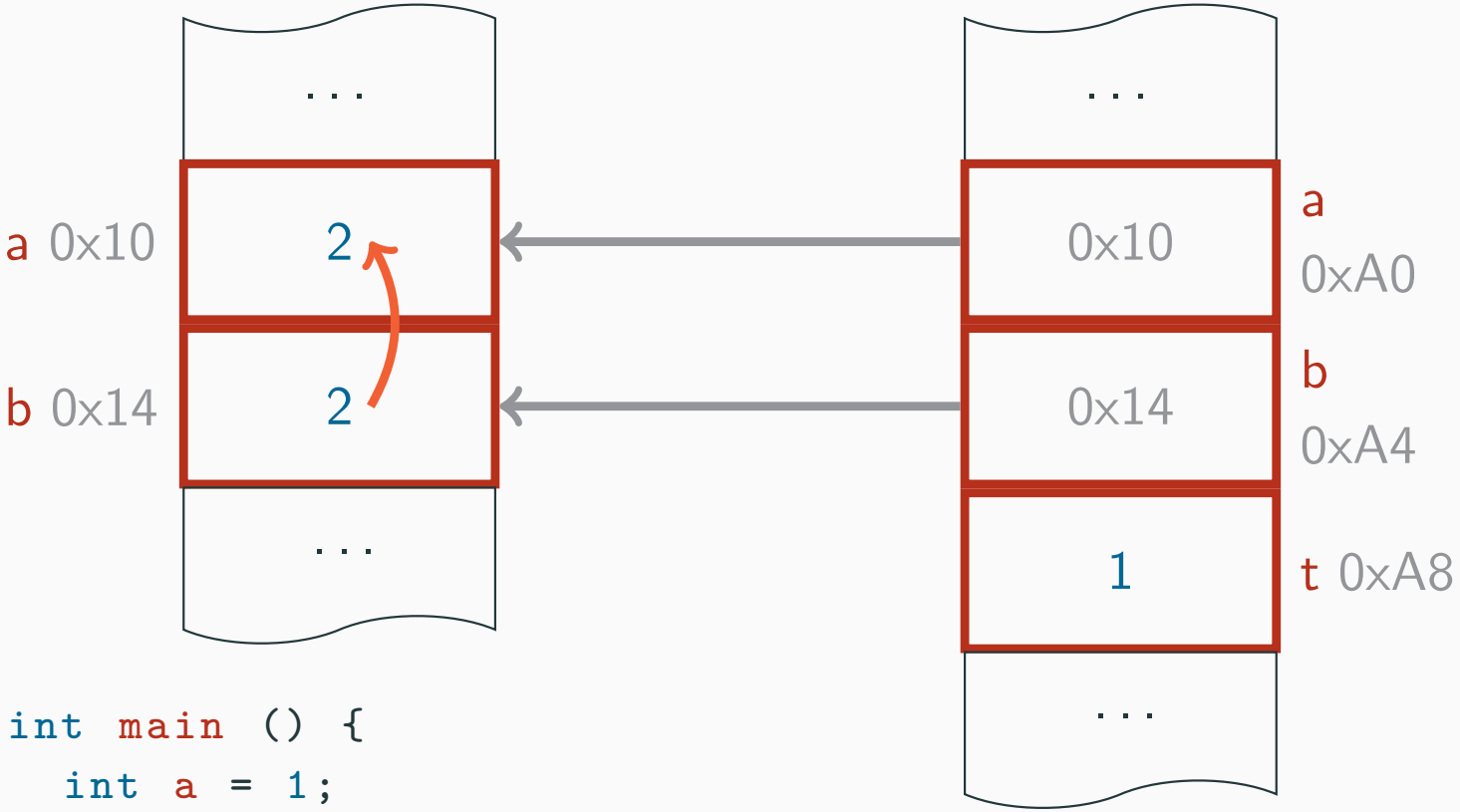
swap mit call-by-reference




```
int main () {
  int a = 1;
  int b = 2;
  swap(&a,&b);
  return 0;
}
```

```
void swap (int* a, int* b) {
  int t = *a; <—
  *a = *b;
  *b = t;
}
```

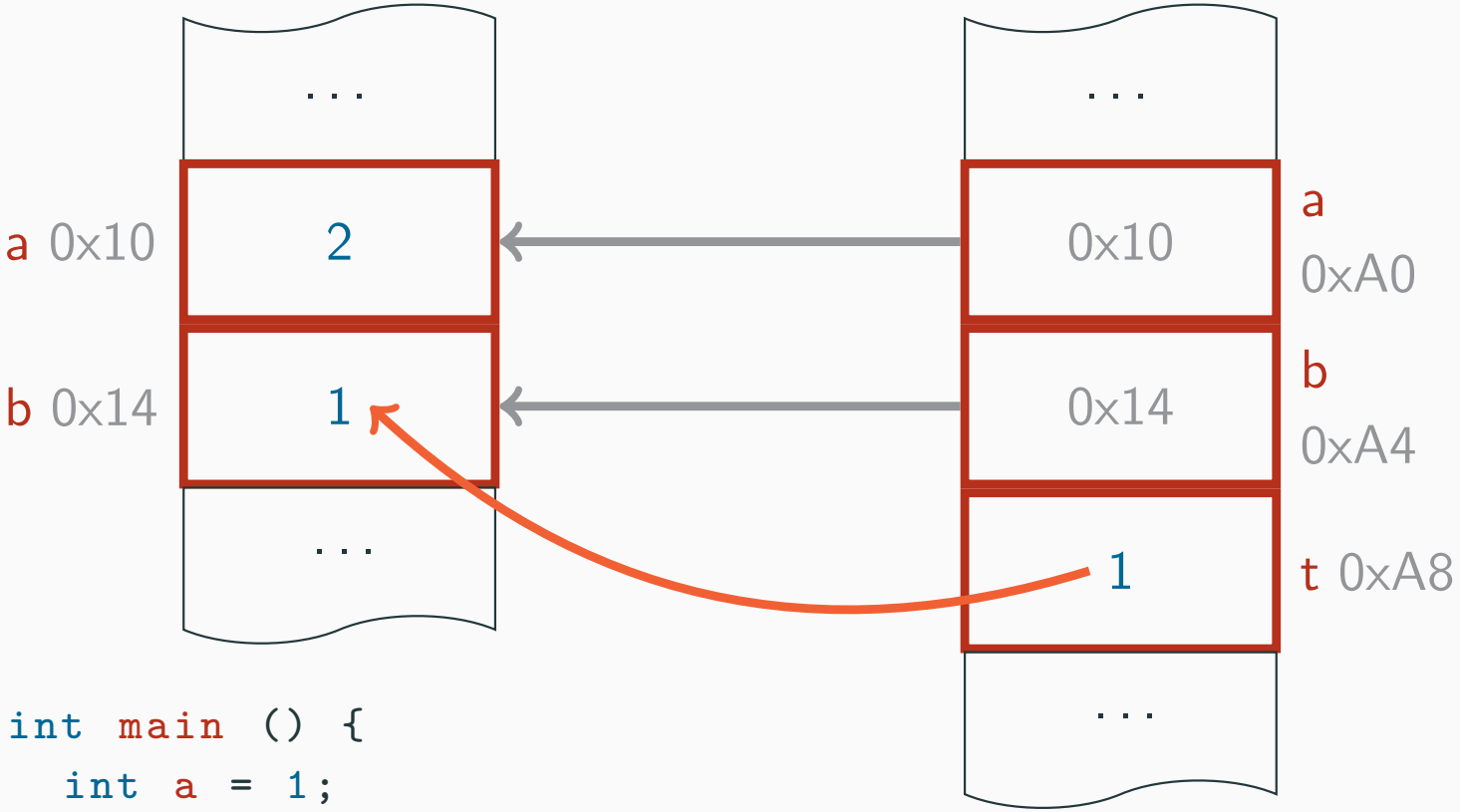
swap mit call-by-reference




```
int main () {
  int a = 1;
  int b = 2;
  swap(&a,&b);
  return 0;
}
```

```
void swap (int* a, int* b) {
  int t = *a;
  *a = *b; <--
  *b = t;
}
```

swap mit call-by-reference



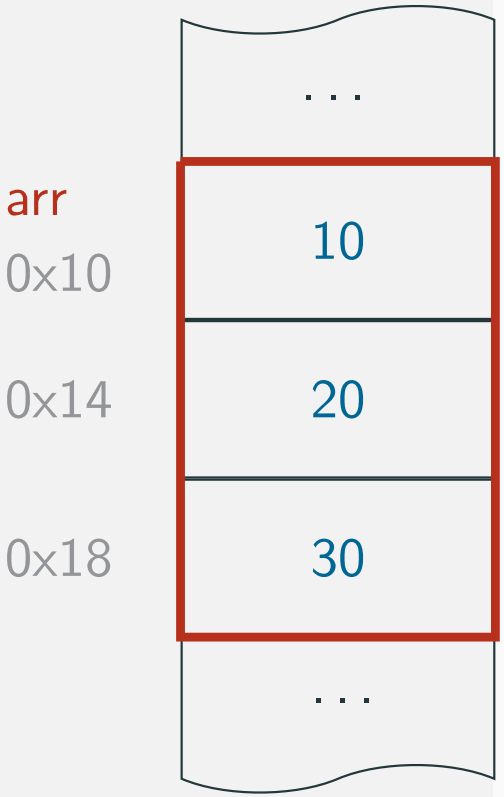

```
int main () {
  int a = 1;
  int b = 2;
  swap(&a,&b);
  return 0;
} // GETAUSCHT!
```

```
void swap (int* a, int* b) {
  int t = *a;
  *a = *b;
  *b = t; <-- FERTIG
}
```

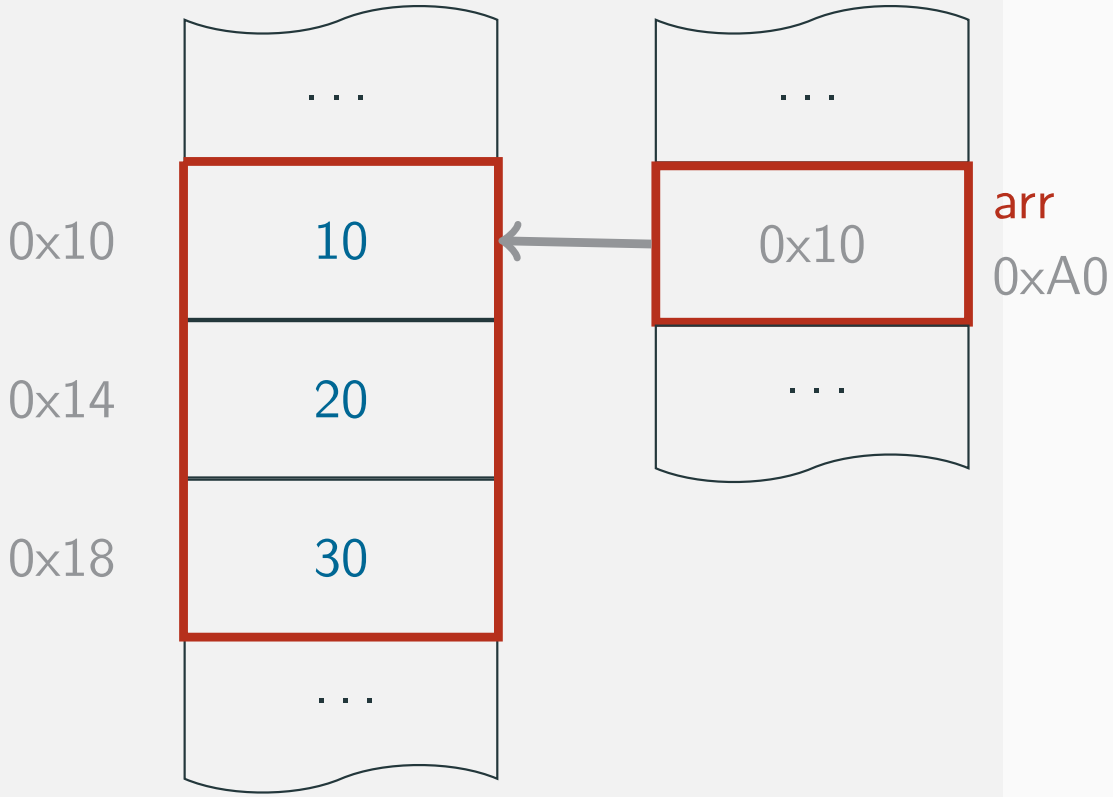
Dynamische Arrays

Statische vs. Dynamische Arrays

Statische Arrays



Dynamische Arrays



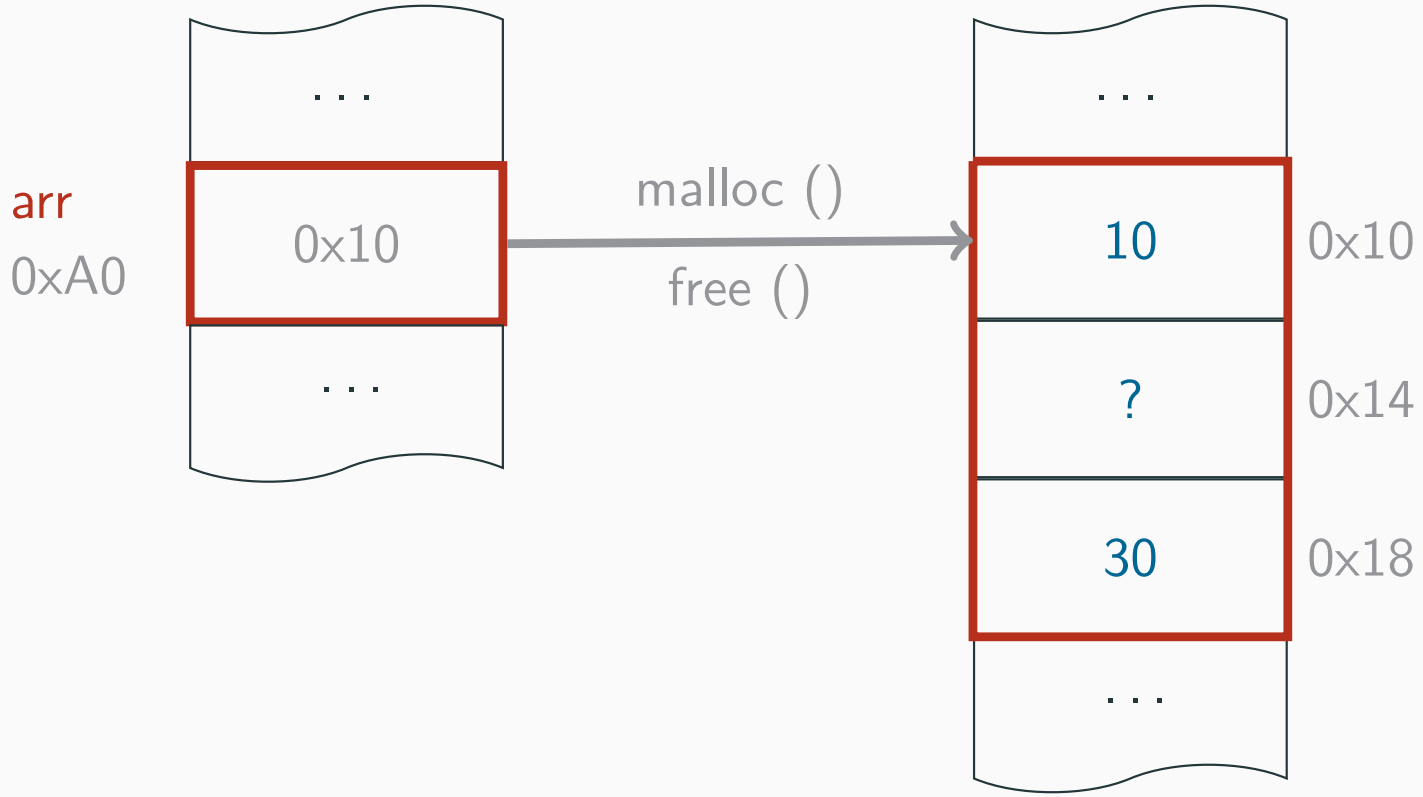
`malloc` und `free` - dynamische Speicherallokation

```c
// Hole Speicher für N Bytes
void*     arr =            malloc (N);

// Hole Speicher für N Elemente vom Typ int (4N Bytes)
int*      arr = (int*)     malloc (N * sizeof(int));

// Hole Speicher für N double (8N Bytes)
double*   arr = (double*) malloc (N * sizeof(double));

// Speicher wieder frei geben
free (arr);
```

Erinnerung: Statische Arrays

```c
char   arr[N]; // N char   (1N Bytes) KEIN void!!!
int    arr[N]; // N int    (4N Bytes)
double arr[N]; // N double (8N Bytes)
```

`malloc` und `free` - dynamische Speicherallokation




```
int* arr = (int*) malloc (3 * sizeof(int));
arr[0] = 10;       // *(arr)   = 10;
arr[2] = 30;       // *(arr+2) = 30;
free (arr);
```

Warum Dynamische Arrays?

Statische Arrays

- Im C89-Standard muss die Größe **vor** dem Kompilieren bekannt sein!
- Ab dem C99-Standard ist das nicht mehr zwingend nötig.

```c
int main () {
  int N;
  scanf("%d", &N);

  int static_arr[N]; // erst mit C99-Standard möglich

  int* dynamic_arr = (int*) malloc (N * sizeof(int));

  return 0;
}
```

Warum Dynamische Arrays?

Statische Arrays

- Der Gültigkeitsbereich ist auf Funktion und aufgerufene Funktionen beschränkt.

```
1  int* alloc_static (int N) { //
2    int arr[N];                // <-- lokales Array
3    return arr;                //
4  }                            // Niemals machen!!
5
6  int* alloc_dynamic (int N) {
7    int* arr = (int*) malloc (N * sizeof(int));
8    return arr;
9  }
10
11 int main () {
12   int* a = alloc_static(10);  // Niemals machen!
13   int* b = alloc_dynamic(10); // Besser
14 }
```

Zusammenfassung

Was kennen Sie jetzt?

- Statische Arrays

```
int arr[N];
int arr[] = {1,2,3};
```

- Pointer

```
int  a   = 4;
int* ptr = &a;
```

- Call-by-value und call-by-reference

```
void swap (int  a, int  b);  // swap ( a,  b);
void swap (int* a, int* b);  // swap (&a, &b);
```

- Dynamische Arrays

```
int* arr = (int*) malloc (N * sizeof(int));
```