Programmierkurs
Speicher und Adressen

Rückblick

VL 0 „Organisation und Inhalt“: Ablauf der Vorlesung, Termine
VL 1 „Hello World“: „Lebenswichtiges“, Programablauf, Programmierablauf, Kompilierung und Ausführung von Programmen
VL 2 „Die ersten Schritte“: Erstes C-Programm, Elementare C-Strukturen, Datentypen, Operatoren, Schleifen
VL 3 „Kontrollstrukturen & Funktionen“: Syntax, Semantik, bedingte Anweisungen, Blöcke, Sichtbarkeit
VL 4 „Rekursive Funktionen & Bibliotheken“: rekursive Funktionsaufrufe, Modularisierung
VL 5 „Typen“: Einfache und strukturierte Datentypen, Wertebereiche, Typendefinition
VL 6 „Speicher und Adressen“: Speicher, Pointer, Funktionsaufrufe „call by value“ vs. „call by reference“
VL 7 „Speicher und Arrays“: Speicher, Arrays, mehrdimensionale Arrays, Arrays und Pointer
VL 8 „Dynamische Speicherverwaltung“: Speicherallokation, Fehlerbehandlung, Rückgabewerte, Arrays/Pointer/Adressen
VL 9 „Strings, Kanäle, Git“: Strings und Arrays, Zeichensätze, Stringlänge, Ein- und Ausgabe, Arbeiten mit git

Speicher

Speicher

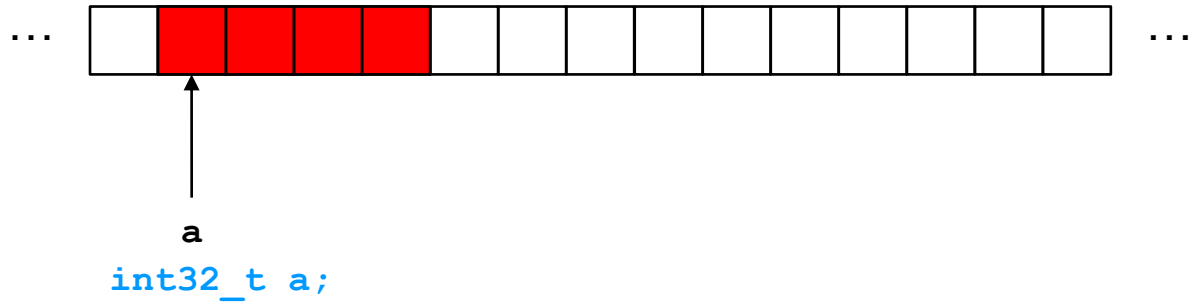
- Speicher besteht aus einer Folge von Bytes

Variablen im Speicher

- Speicher besteht aus einer Folge von Bytes
- Beispiel `int32_t`: wird in 4 aufeinanderfolgenden Bytes gespeichert

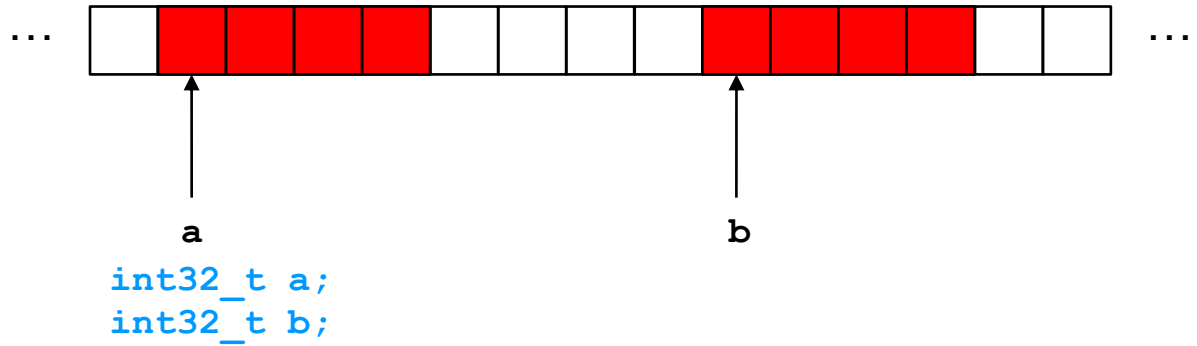
Variablen im Speicher

- Speicher besteht aus einer Folge von Bytes
- Beispiel `int32_t`: wird in 4 aufeinanderfolgenden Bytes gespeichert



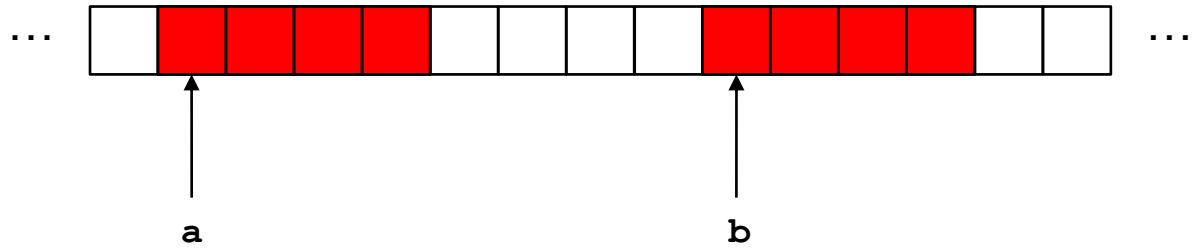
Variablen im Speicher

- Speicher besteht aus einer Folge von Bytes
- Beispiel `int32_t`: wird in 4 aufeinanderfolgenden Bytes gespeichert



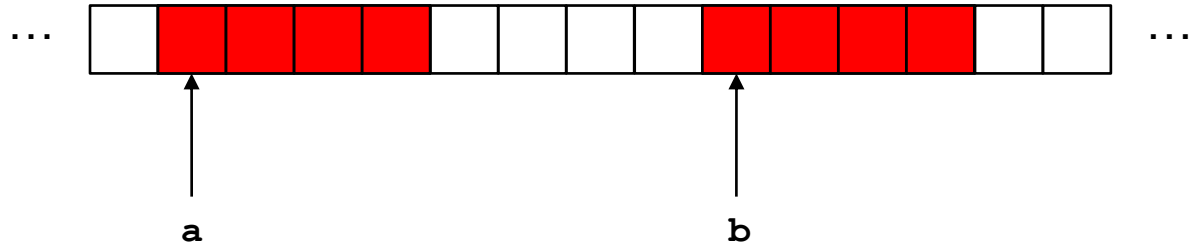
Variablen im Speicher

- Deklaration einer Variablen reserviert Speicher für die Variable



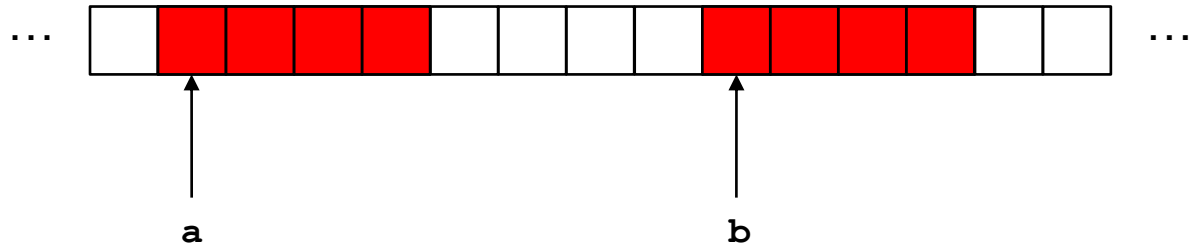
Variablen im Speicher

- Deklaration einer Variablen reserviert Speicher für die Variable
- Zuweisung entspricht Belegung des Speichers mit einem Wert



Variablen im Speicher

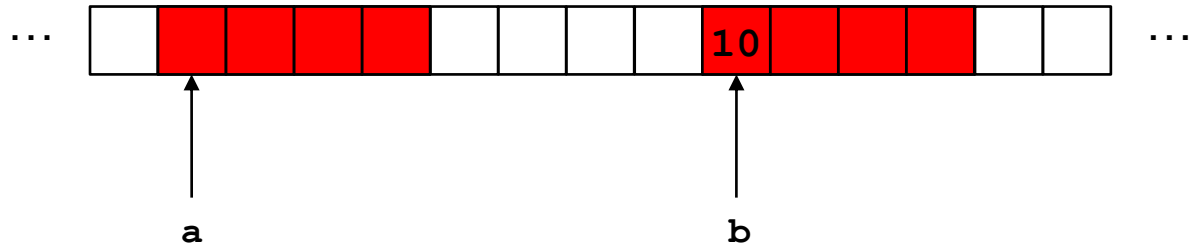
- Deklaration einer Variablen reserviert Speicher für die Variable
- Zuweisung entspricht Belegung des Speichers mit einem Wert




```
b = 10;
```

Variablen im Speicher

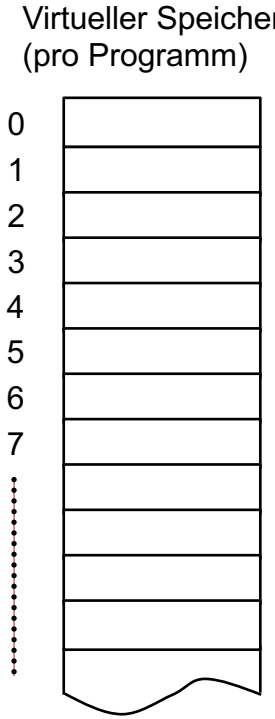
- Deklaration einer Variablen reserviert Speicher für die Variable
- Zuweisung entspricht Belegung des Speichers mit einem Wert




```
b = 10;
```

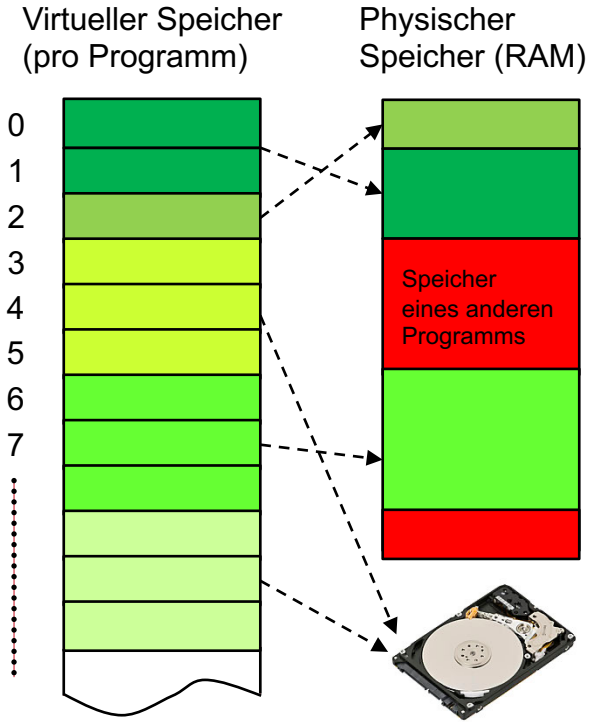
Speicher – Abstraktion

- Virtueller Speicher: Ein Bytearray
- Programmsicht:
  - Jedes Programm hat seinen eigenen Speicher
  - Es hat eine „unbegrenzte Speichermenge“
  - Der Zugriff auf alle Speicherbereiche ist gleich schnell…



Speicher – Realität

- Virtueller Speicher: Ein Bytearray
- Realität:
  - Kein unbegrenzter physikalischer Speicher
    - Alle Programme teilen sich den selben physikalischen Speicher
    - Speicher wird durch das Betriebssystem zugewiesen und verwaltet
    - Viele Anwendungen sind speicherdominiert
  - Es gibt eine Speicherhierarchie
    - Cache, RAM, Festplatte, Netzwerk-Speicher (schnell → langsam)
  - Speicherzugriffsfehler sind besonders schwer zu finden
    - Effekte sind oft weit von der Ursache entfernt



(Speicher-) Adressen von Variablen

Variablen (Wiederholung)

- Variablen
  - Sind Platzhalter für Daten.
  - Geben somit Daten einen „Namen“.
  - Haben einen festgelegten Speicherort an dem der aktuelle Wert gespeichert wird.
  - Der aktuelle Wert ist veränderbar.

```
1.int x;
2.x = 5;
3.int y;
4.x = 6;
```

Zustand	x			
Zustand	x	5		
Zustand	x	5	y	
Zustand	x	6	y	

Adressen von Variablen

- Adressen von Variablen
  - Der Ort an dem Daten tatsächlich gespeichert sind
  - Kann sich somit nicht ändern!
  - Zugriff auf die Adresse mittels `&` vor dem Variablennamen

- Beispiel:

```
int x = 5, y = 3;
printf("The value of x is %d\n", x);
printf("Addresses of x and y are %p %p\n", &x, &y);
x = 6;
printf …
```

Zustand	x	5	y	3

Zustand	x	6	y	3

Adressen von Variablen: Beispiel

```
int x = 5, y = 3;
printf("The value of x is %d\n", x);
printf("Addresses of x and y are %p %p\n", &x, &y);
x = 6;
printf("Addresses of x and y are %p %p\n", &x, &y);
```

Zustand x 5 y 3

Zustand x 6 y 3

Ausgabe:

```
The value of x is 5
Addresses of x and y are 0x7edf04ec 0x7edf04e8
The value of x is 6
Addresses of x and y are 0x7edf04ec 0x7edf04e8
```

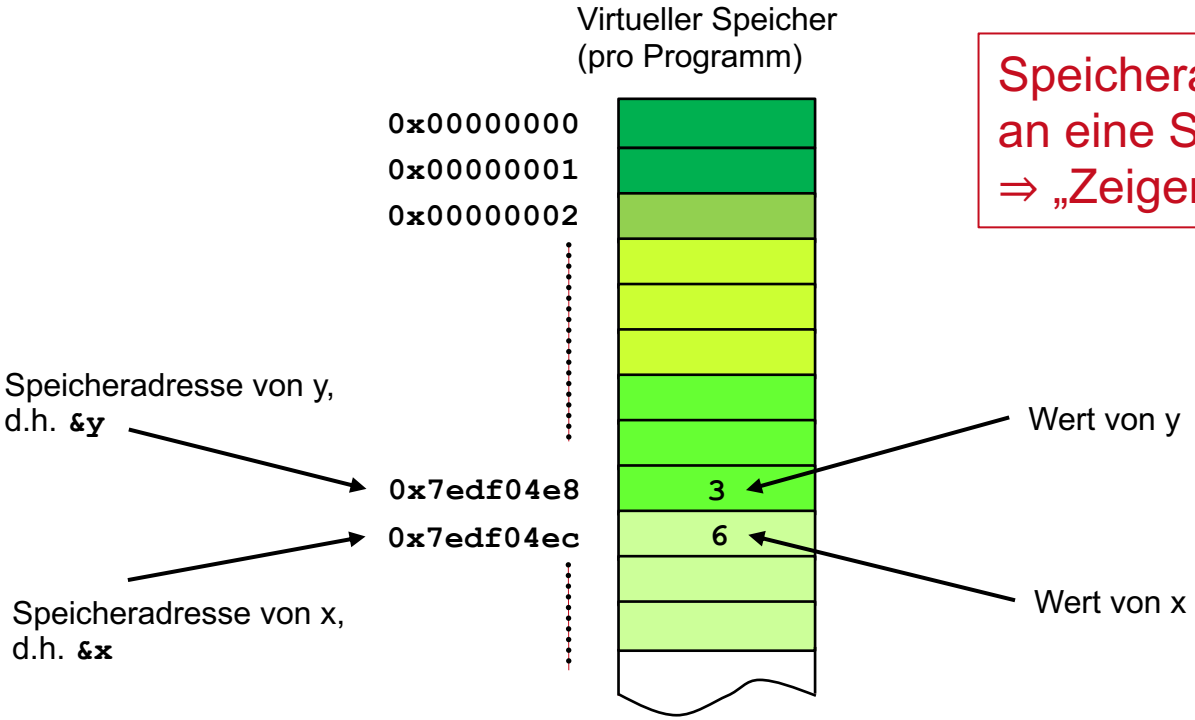
Speicheradresse von x

Speicheradresse von y

Hexadezimal ? Hexadezimal !

- 1 Byte = 8 Bit ⇒ kann 2^8 verschiedene Werte darstellen
- 2^8 Werte = [0, 255] (00000000 – 1111111111)
- Zusammenfassung von 4 Bit zu einer Ziffer (0 – „15“)
  - Ziffern: 0, …, 9, A, B, C, D, F
  - D.h. Zahlensystem zur Basis 16 („hexadezimal“)
  - 2 hexadezimale Zahlen können 1 Byte darstellen:
    00000000 – 11111111 (binär)
    0 – 255 (dezimal)
    00 – FF (hexadezimal)
  - Kennzeichnung: „0x“ vorangestellt, z.B. **`0x7edf04ec`**

Speicheradressen: Beispiel



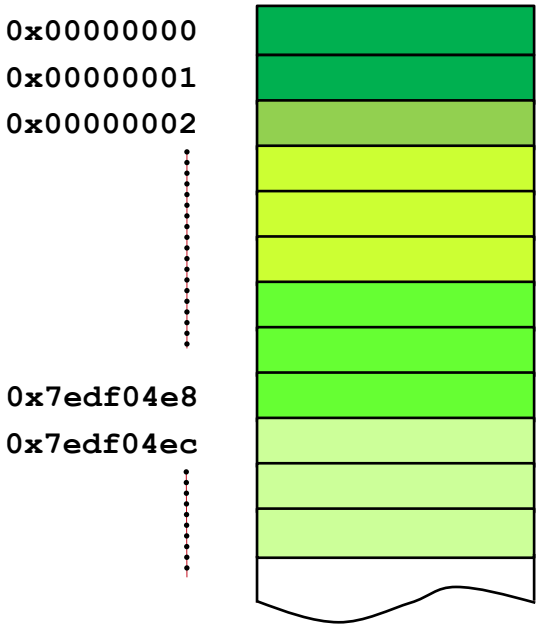
Speicheradressen „zeigen“ an eine Stelle im Speicher
⇒ „Zeiger“ bzw. „Pointer“

Pointertypen

- Pointer: „Eine Variable, die eine Speicheradresse als ihren Wert speichert.“
- Auch Pointer haben einen Typ und werden mit „<Typ>* “ definiert
- Der Typ eines Pointers ist „Pointer auf <Typ>“
- Der Wert auf den ein Pointer zeigt: „*<Pointervariable>“

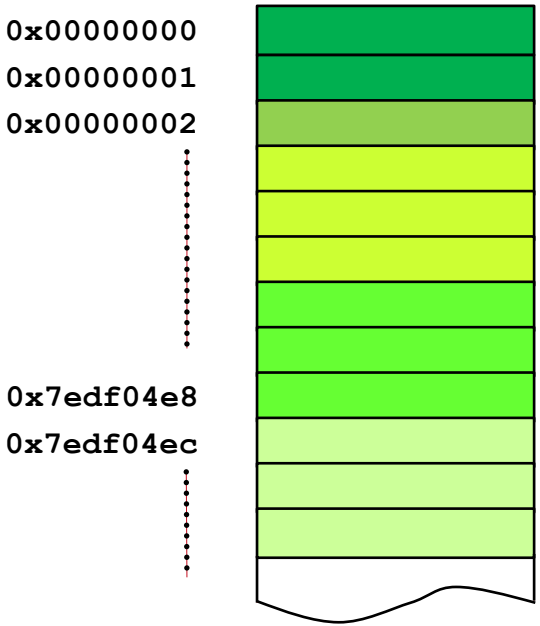
```
int  x = 5;  // declare x as an integer variable
int* p, q;   // declare p and q as pointers to an integer
             // int *p; and int * p; are OK as well
p = &x;      // store address of x in p
int y = *p;  // assign y the value p points to, i.e., x, i.e., 5
q = p;       // q points to the same location as p, i.e., x
printf("Value of x: %d (at address %p)", *p, q);
```

Arbeiten mit Pointern (Zeigern)



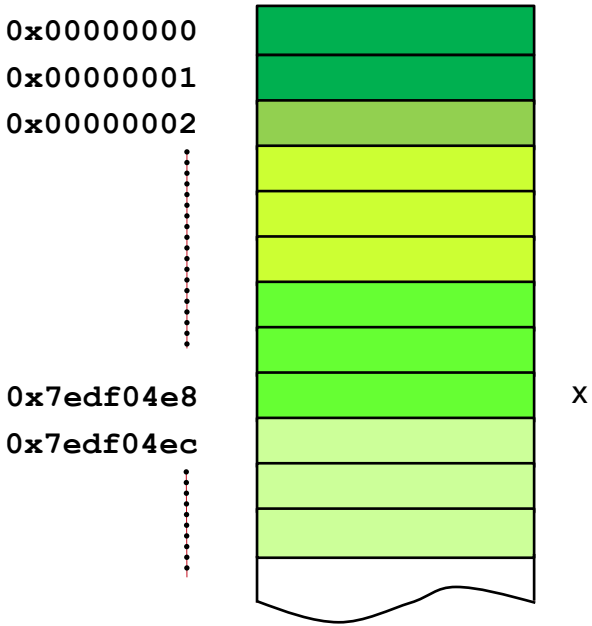
Arbeiten mit Pointern (Zeigern)

```
int  x = 5;
```



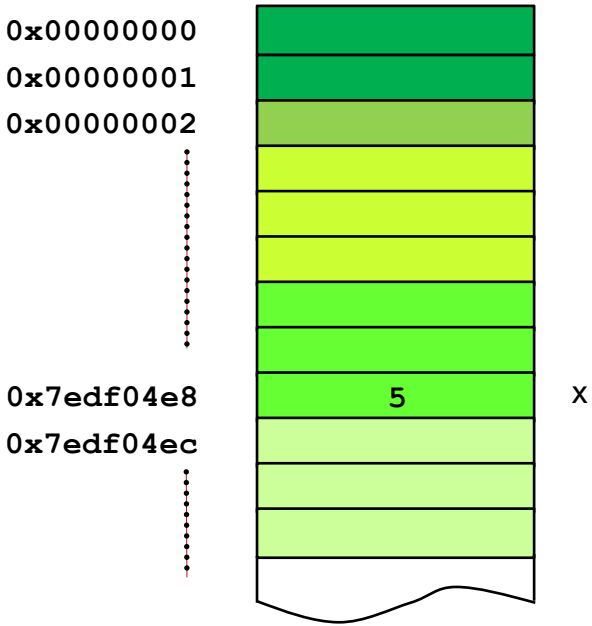
Arbeiten mit Pointern (Zeigern)

```
int  x = 5;
```



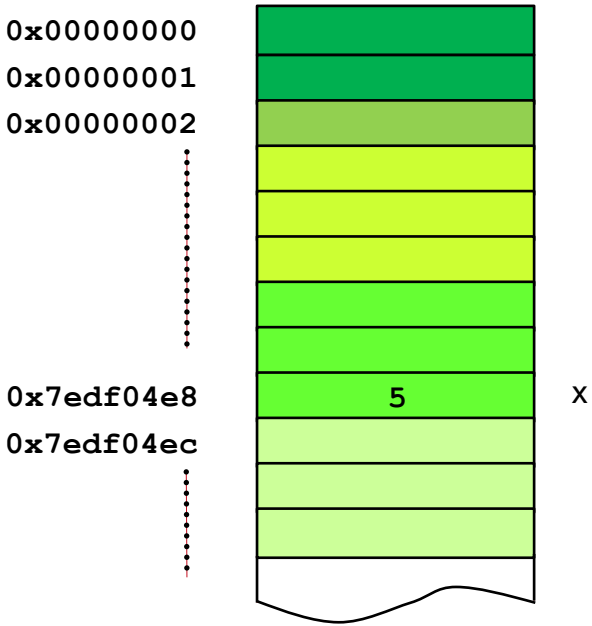
Arbeiten mit Pointern (Zeigern)

```
int  x = 5;
```



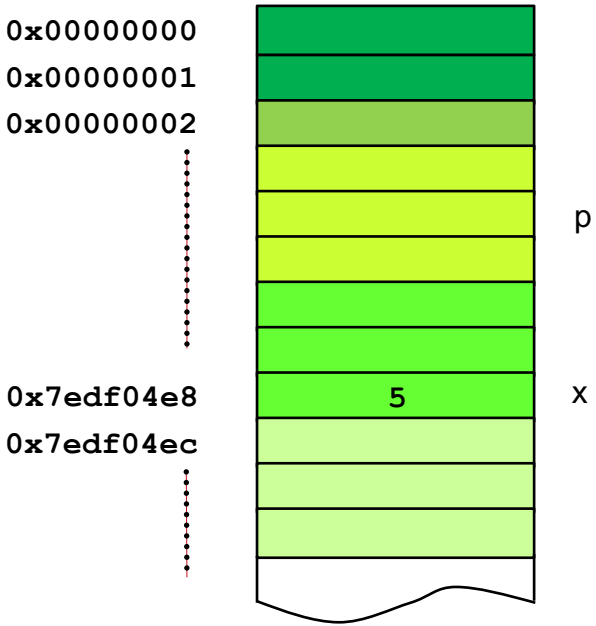
Arbeiten mit Pointern (Zeigern)

```
int  x = 5;
int* p, q;
```



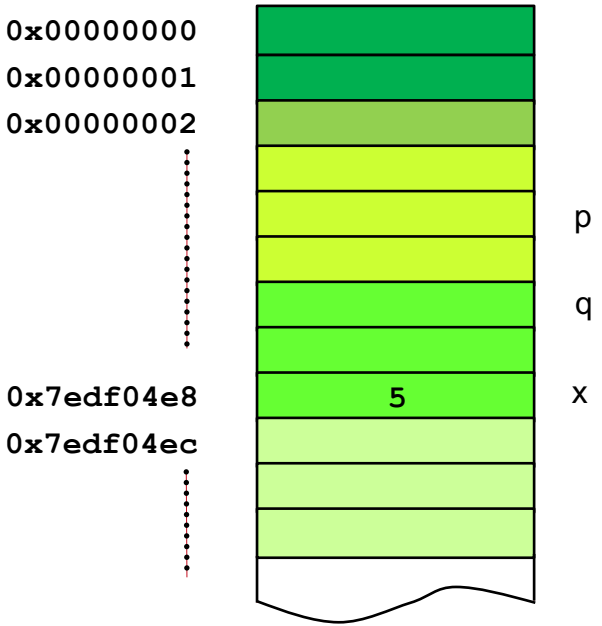
Arbeiten mit Pointern (Zeigern)

```
int  x = 5;
int* p, q;
```



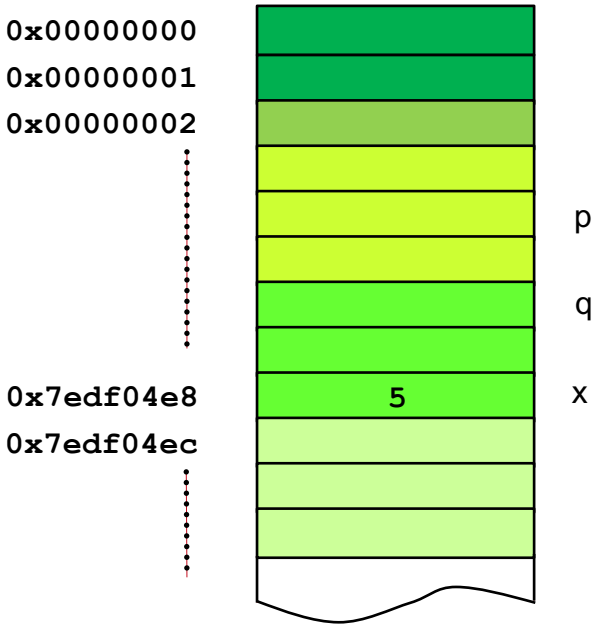
Arbeiten mit Pointern (Zeigern)

```
int  x = 5;
int* p, q;
```



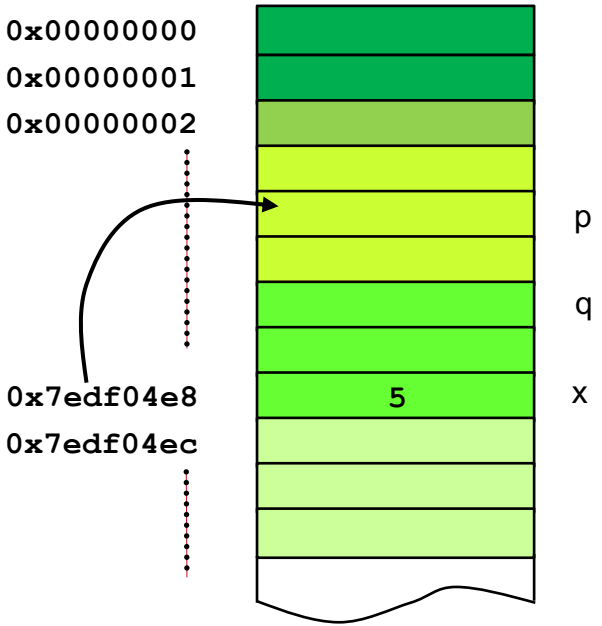
Arbeiten mit Pointern (Zeigern)

```
int  x = 5;
int* p, q;
p = &x;
```



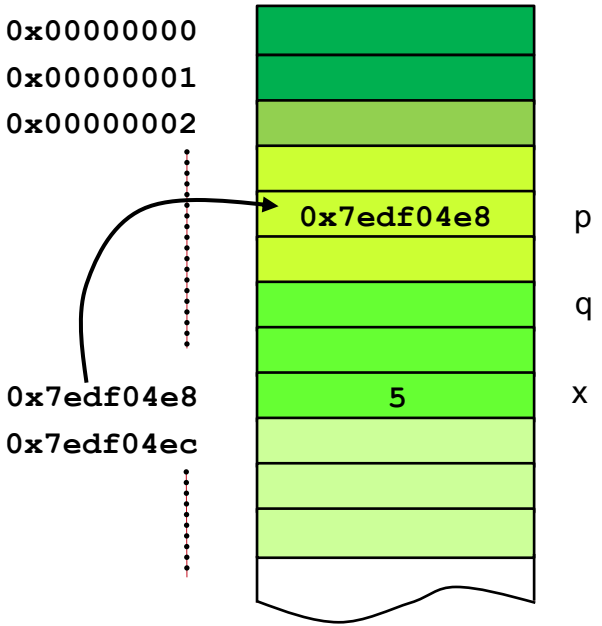
Arbeiten mit Pointern (Zeigern)

```
int  x = 5;
int* p, q;
p = &x;
```



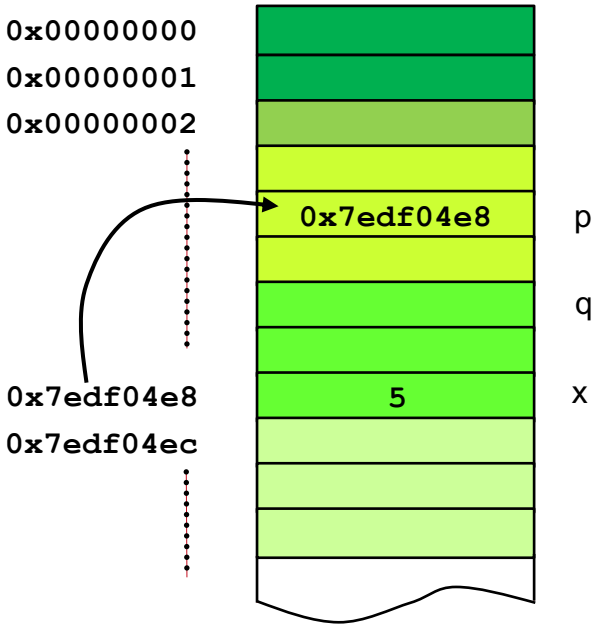
Arbeiten mit Pointern (Zeigern)

```
int  x = 5;
int* p, q;
p = &x;
```



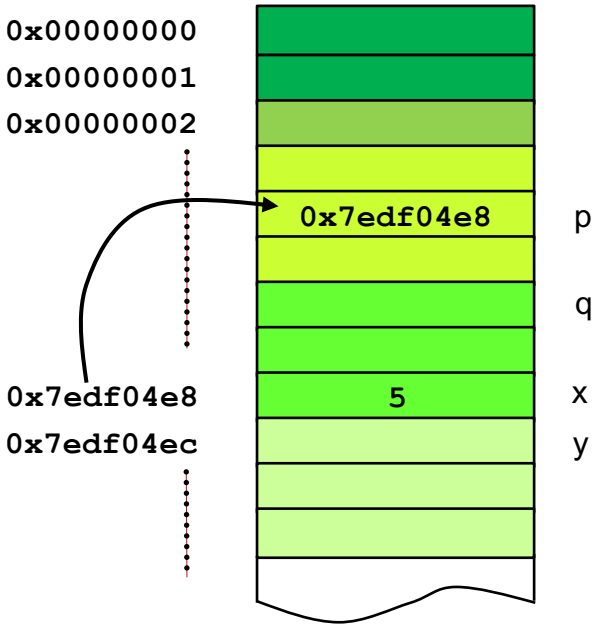
Arbeiten mit Pointern (Zeigern)

```
int  x = 5;
int* p, q;
p = &x;
int y = *p;
```



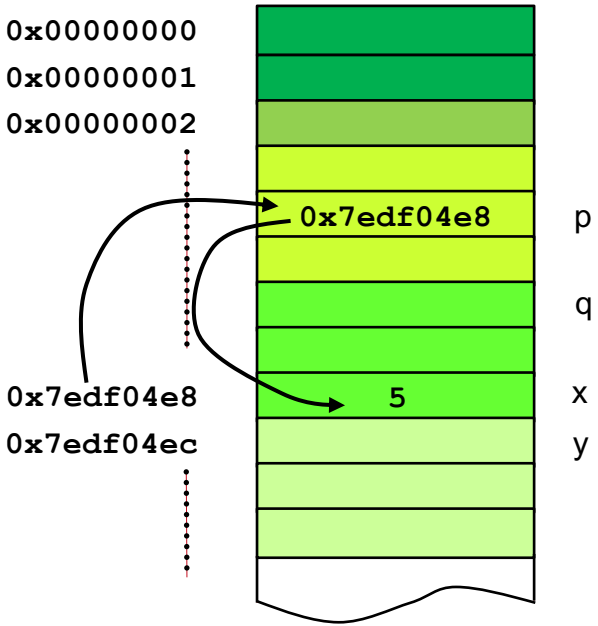
Arbeiten mit Pointern (Zeigern)

```
int  x = 5;
int* p, q;
p = &x;
int y = *p;
```



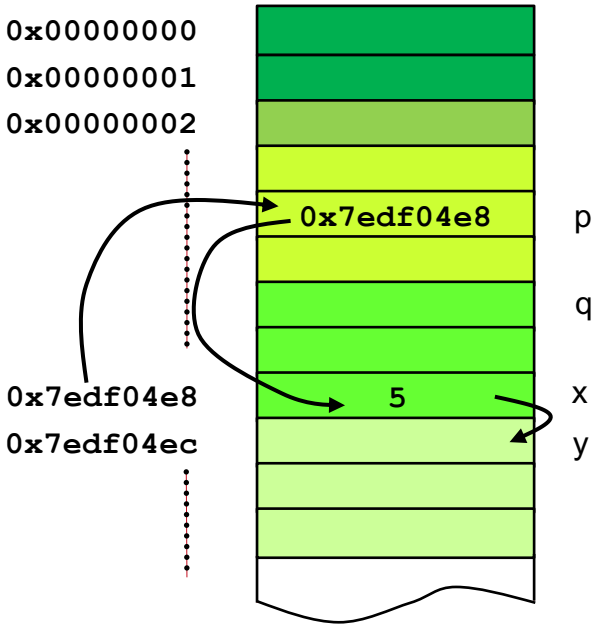
Arbeiten mit Pointern (Zeigern)

```
int  x = 5;
int* p, q;
p = &x;
int y = *p;
```



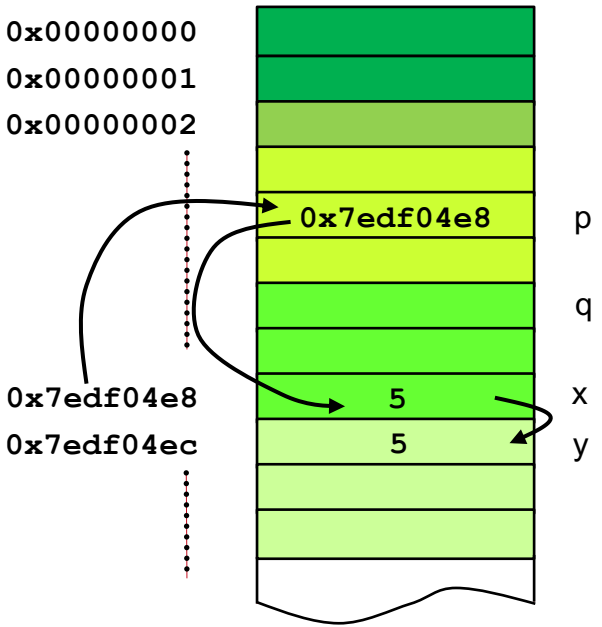
Arbeiten mit Pointern (Zeigern)

```
int  x = 5;
int* p, q;
p = &x;
int y = *p;
```



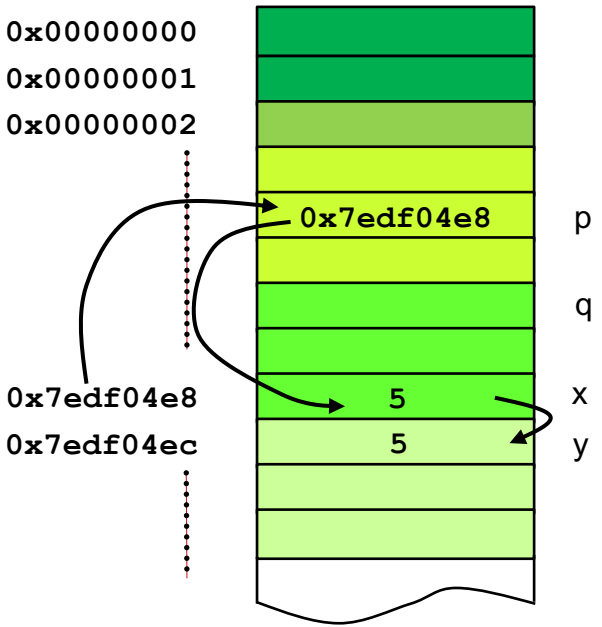
Arbeiten mit Pointern (Zeigern)

```
int  x = 5;
int* p, q;
p = &x;
int y = *p;
```



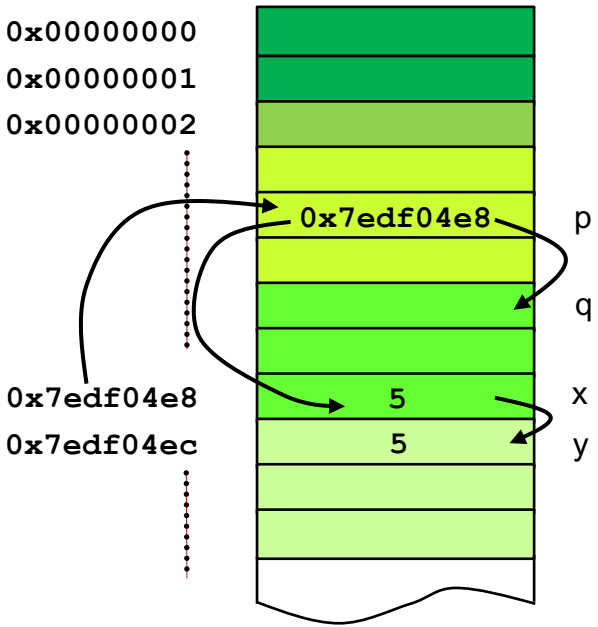
Arbeiten mit Pointern (Zeigern)

```
int  x = 5;
int* p, q;
p = &x;
int y = *p;
q = p;
```



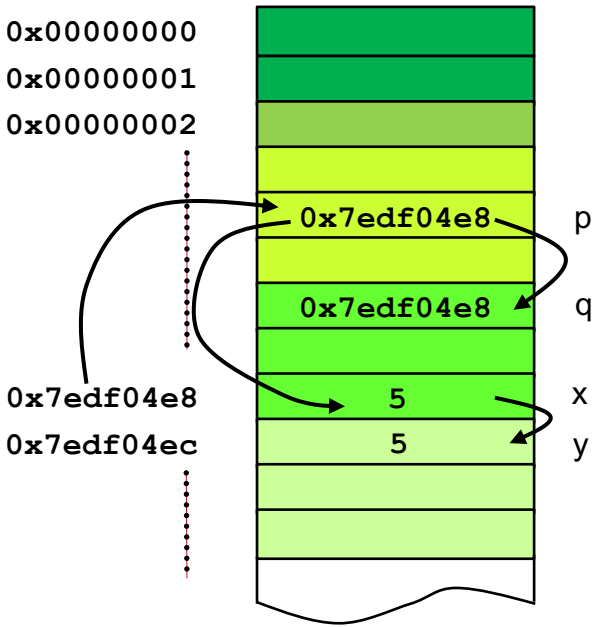
Arbeiten mit Pointern (Zeigern)

```
int  x = 5;
int* p, q;
p = &x;
int y = *p;
q = p;
```



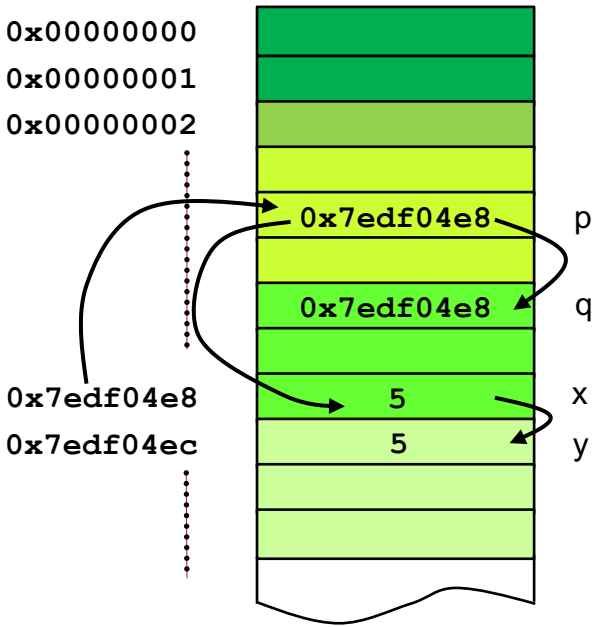
Arbeiten mit Pointern (Zeigern)

```
int  x = 5;
int* p, q;
p = &x;
int y = *p;
q = p;
```



Arbeiten mit Pointern (Zeigern)

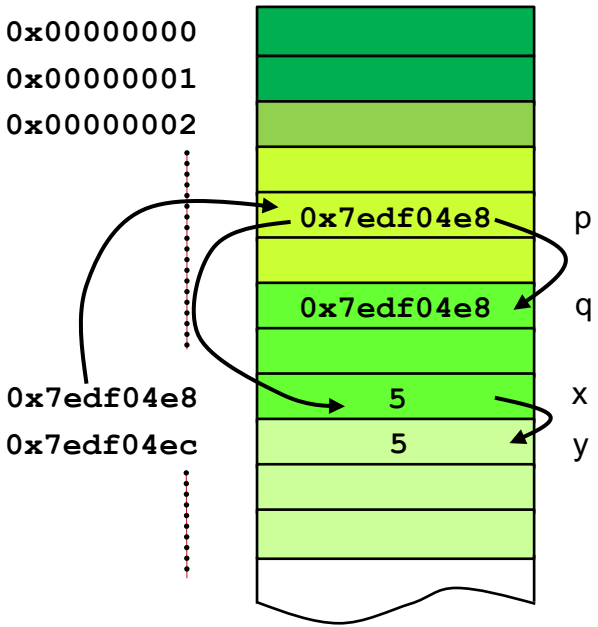
```
int  x = 5;
int* p, q;
p = &x;
int y = *p;
q = p;
```




```
printf("Value of x: %d (at address %p)", *p, q);
```

Arbeiten mit Pointern (Zeigern)

```
int  x = 5;
int* p, q;
p = &x;
int y = *p;
q = p;
```




```
printf("Value of x: %d (at address %p)", *p, q);
```

Ausgabe: `Value of x: 5 (at address 0x7edf04e8)`

Pointer und struct

- Pointer können auf alles zeigen, d.h. auch auf eine `struct`

```
typedef struct Point3d_ {
    float x;
    float y;
    float z;
} Point3d;              // type for 3D points
Point3d my_point;       // a variable of type Point3d
Point3d *p;             // a pointer to type Point3d
p = &my_point;          // p points to myPoint now
```

Pointer und Teile einer struct

```
Point3d my_point = { .x = 0.5, .y = 3.14, .z = -123.4 };
Point3d *p = &my_point;
Printf("x: %f, y: %f, z: %f\n", my_point.x, my_point.y, my_point.z);
Printf("x: %f, y: %f, z: %f\n", p->x, p->y, p->z);
```

Die Ausgabe ist in beiden Fällen:

```
x: 0.5, y: 3.14, z: -123.4
```

Zugriff auf Komponenten einer `struct`:

- Variablen: mit „.“, z.B.: `my_point.x`
- Pointer: mit „->“, z.B.: `p->x`

Bekannt: Rekursive struct unmöglich

```
struct Weird {
    int8_t        i;
    struct Weird w;
};

Weird x = {.i = 0, .w = { .i = 1, .w = {... /*oh no!*/}}};
```

Pointer lösen Rekursionsproblem

```
typedef struct NotWeird_ {
    int8_t                  i;
    struct NotWeird_*   w; // pointer to
                           // struct NotWeird_
} NotWeird;

NotWeird w1;
NotWeird w2 = { .i = 0, .w = &w1 };
```

Funktionsaufrufe und Parameterübergabe

Bespiel: swap

- Aufgabe: Schreibe eine Funktion, die 2 Werte vertauscht

```
// swap exchanges the values of the parameters
int swap (int a, int b){
    int z;

    z = a;
    a = b;
    b = z;
    return 0; // everything went fine
}

int main () {
     int x = 5, y = 3;
     swap(x, y);
     printf("x: %d, y: %d\n", x, y);
}
```

Bespiel: swap

- Aufgabe: Schreibe eine Funktion, die 2 Werte vertauscht

```
// swap exchanges the values of the parameters
int swap (int a, int b){
    int z;

    z = a;
    a = b;
    b = z;
    return 0; // everything went fine
}

int main () {
     int x = 5, y = 3;
     swap(x, y);
     printf("x: %d, y: %d\n", x, y);
}
```

- **Ausgabe:**

Bespiel: swap

- Aufgabe: Schreibe eine Funktion, die 2 Werte vertauscht

```
// swap exchanges the values of the parameters
int swap (int a, int b){
    int z;

    z = a;
    a = b;
    b = z;
    return 0; // everything went fine
}

int main () {
     int x = 5, y = 3;
     swap(x, y);
     printf("x: %d, y: %d\n", x, y);
}
```

- **Ausgabe:** `x: 5, y: 3`

Bespiel: swap

- Aufgabe: Schreibe eine Funktion, die 2 Werte vertauscht

```
// swap exchanges the values of the parameters
int swap (int a, int b){
    int z;

    z = a;
    a = b;
    b = z;
    return 0; // everything went fine
}

int main () {
     int x = 5, y = 3;
     swap(x, y);
     printf("x: %d, y: %d\n", x, y);
}
```

- **Ausgabe:** `x: 5, y: 3`

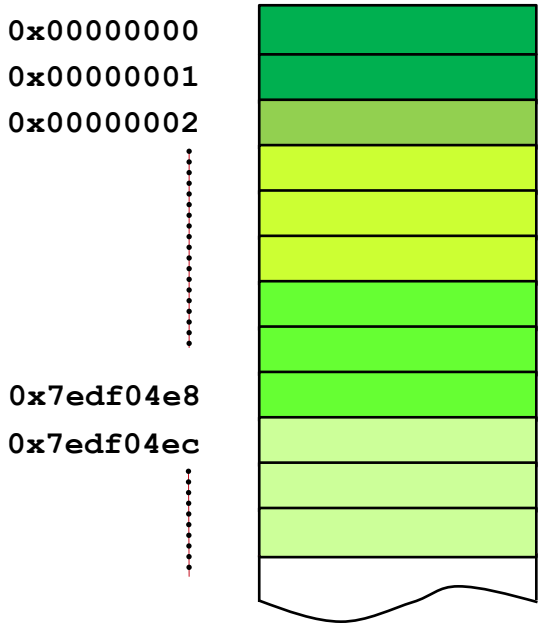
Was ist passiert?

- Aufgabe: Schreibe eine Funktion, die 2 Werte vertauscht

```
// swap exchanges the values of the parameters
int swap (int a, int b){
    int z;

    z = a;
    a = b;
    b = z;
    return 0; // everything went fine
}

int main () {
     int x = 5, y = 3;
     swap(x, y);
     printf("x: %d, y: %d\n", x, y);
}
```



- **Ausgabe:** `x: 5, y: 3`

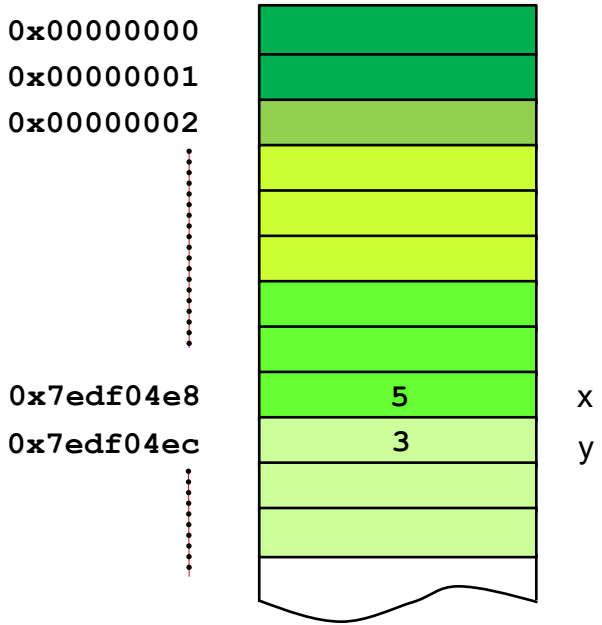
Was ist passiert?

- Aufgabe: Schreibe eine Funktion, die 2 Werte vertauscht

```
// swap exchanges the values of the parameters
int swap (int a, int b){
    int z;

    z = a;
    a = b;
    b = z;
    return 0; // everything went fine
}

int main () {
     int x = 5, y = 3;
     swap(x, y);
     printf("x: %d, y: %d\n", x, y);
}
```



- **Ausgabe:** `x: 5, y: 3`

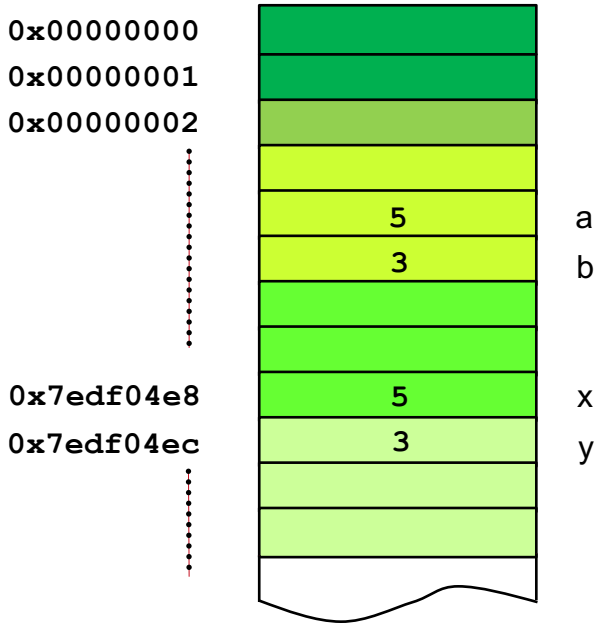
Was ist passiert?

- Aufgabe: Schreibe eine Funktion, die 2 Werte vertauscht

```
// swap exchanges the values of the parameters
int swap (int a, int b){
    int z;

    z = a;
    a = b;
    b = z;
    return 0; // everything went fine
}

int main () {
     int x = 5, y = 3;
     swap(x, y);
     printf("x: %d, y: %d\n", x, y);
}
```



- **Ausgabe:** `x: 5, y: 3`

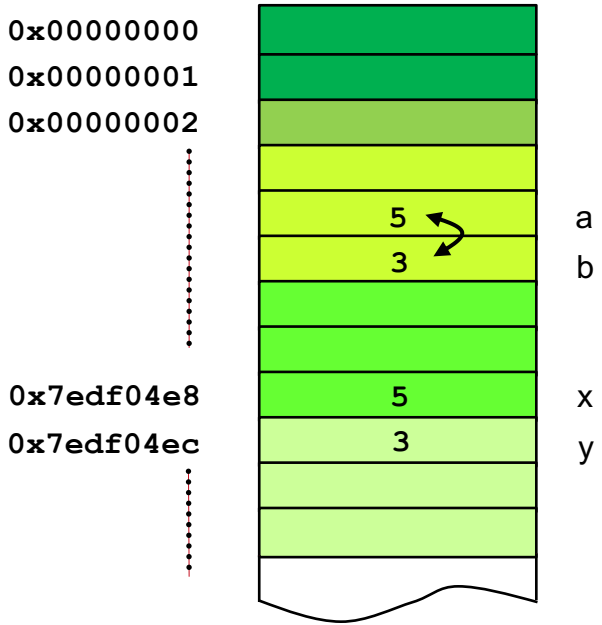
Was ist passiert?

- Aufgabe: Schreibe eine Funktion, die 2 Werte vertauscht

```
// swap exchanges the values of the parameters
int swap (int a, int b){
    int z;

    z = a;
    a = b;
    b = z;
    return 0; // everything went fine
}

int main () {
     int x = 5, y = 3;
     swap(x, y);
     printf("x: %d, y: %d\n", x, y);
}
```



- **Ausgabe:** `x: 5, y: 3`

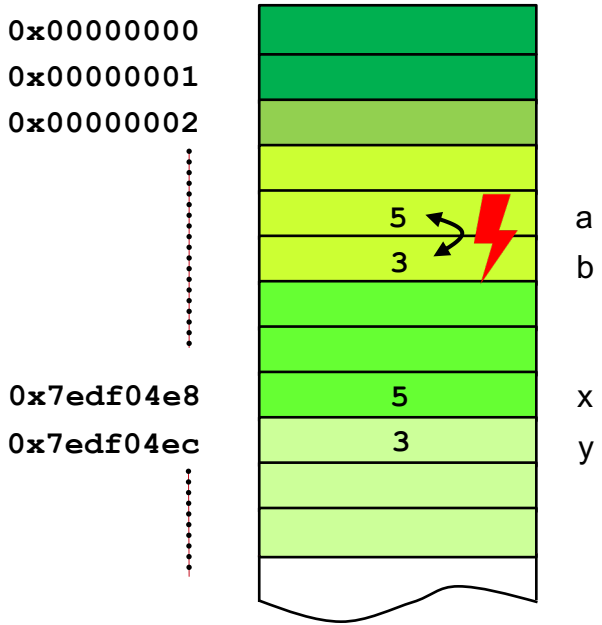
Was ist passiert?

- Aufgabe: Schreibe eine Funktion, die 2 Werte vertauscht

```
// swap exchanges the values of the parameters
int swap (int a, int b){
    int z;

    z = a;
    a = b;
    b = z;
    return 0; // everything went fine
}

int main () {
     int x = 5, y = 3;
     swap(x, y);
     printf("x: %d, y: %d\n", x, y);
}
```



- **Ausgabe:** `x: 5, y: 3`

Was ist passiert?

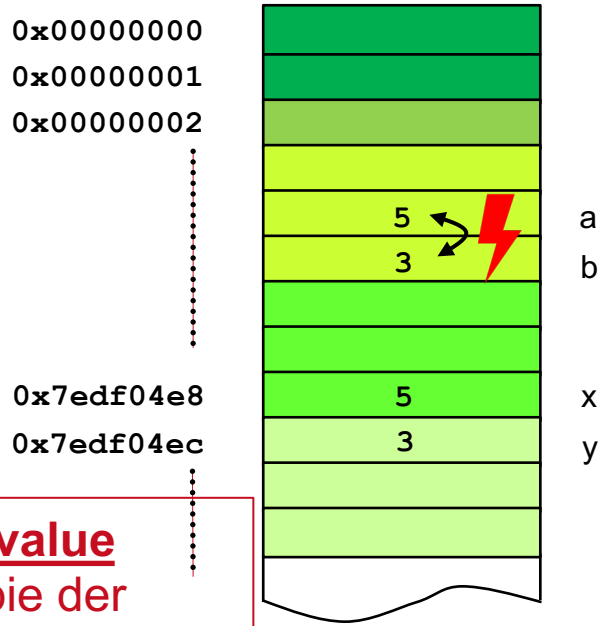
- Aufgabe: Schreibe eine Funktion, die 2 Werte vertauscht

```
// swap exchanges the values of the parameters
int swap (int a, int b){
    int z;

    z = a;
    a = b;
    b = z;
    return 0; // everything went fine
}

int main () {
     int x = 5, y = 3;
     swap(x, y);
     printf("x: %d, y: %d\n", x, y);
}
```

- **Ausgabe:** `x: 5, y: 3`



Call by value
Eine Kopie der Parameter wird angelegt

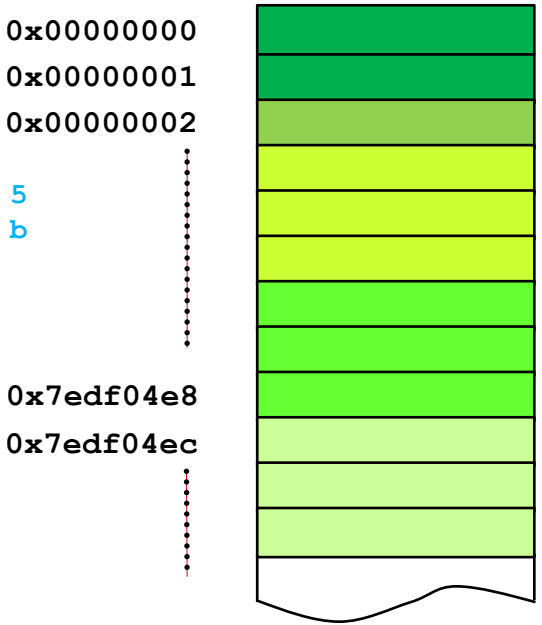
Lösung: Call by reference (pointer)

- Aufgabe: Schreibe eine Funktion, die 2 Werte vertauscht

```
// swap exchanges the values of the parameters
int swap (int* a, int* b){
    int z;

    z = *a;   // z = value a points to, i.e., x, i.e., 5
    *a = *b;  // value at address a = value at address b
    *b = z;   // value at address b = z = 5
    return 0; // everything went fine
}

int main () {
     int x = 5, y = 3;
     swap(&x, &y);
     printf("x: %d, y: %d\n", x, y);
}
```



- **Ausgabe:** `x: 3, y: 5` ✓

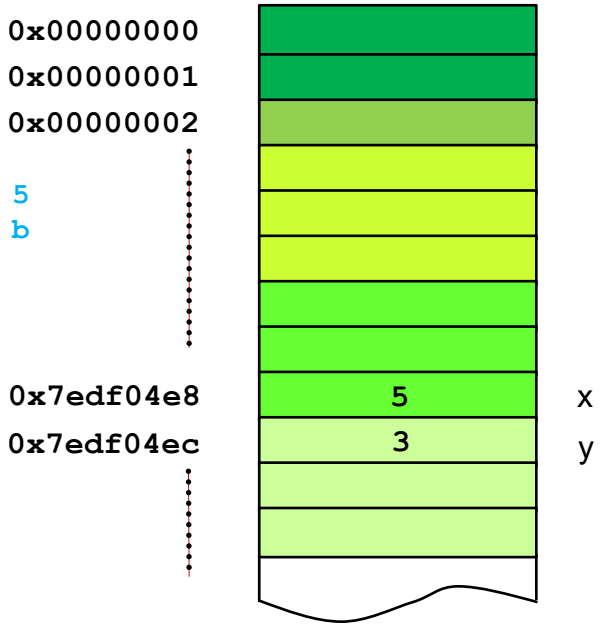
Lösung: Call by reference (pointer)

- Aufgabe: Schreibe eine Funktion, die 2 Werte vertauscht

```
// swap exchanges the values of the parameters
int swap (int* a, int* b){
    int z;

    z = *a;   // z = value a points to, i.e., x, i.e., 5
    *a = *b;  // value at address a = value at address b
    *b = z;   // value at address b = z = 5
    return 0; // everything went fine
}

int main () {
     int x = 5, y = 3;
     swap(&x, &y);
     printf("x: %d, y: %d\n", x, y);
}
```



- **Ausgabe:** `x: 3, y: 5` ✓

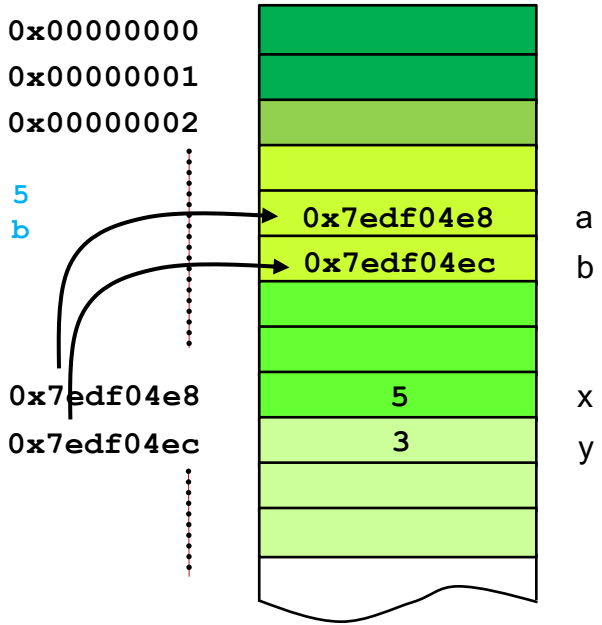
Lösung: Call by reference (pointer)

- Aufgabe: Schreibe eine Funktion, die 2 Werte vertauscht

```
// swap exchanges the values of the parameters
int swap (int* a, int* b){
    int z;

    z = *a;   // z = value a points to, i.e., x, i.e., 5
    *a = *b;  // value at address a = value at address b
    *b = z;   // value at address b = z = 5
    return 0; // everything went fine
}

int main () {
     int x = 5, y = 3;
     swap(&x, &y);
     printf("x: %d, y: %d\n", x, y);
}
```



- **Ausgabe:** `x: 3, y: 5` ✓

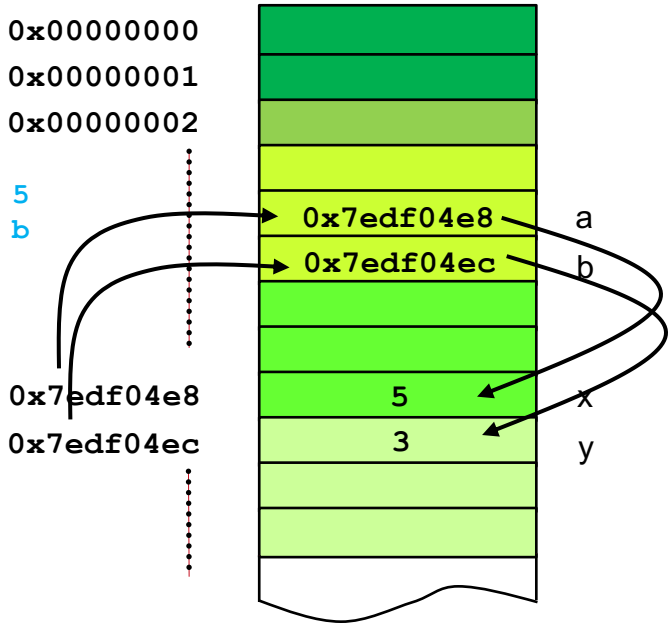
Lösung: Call by reference (pointer)

- Aufgabe: Schreibe eine Funktion, die 2 Werte vertauscht

```
// swap exchanges the values of the parameters
int swap (int* a, int* b){
    int z;

    z = *a;   // z = value a points to, i.e., x, i.e., 5
    *a = *b;  // value at address a = value at address b
    *b = z;   // value at address b = z = 5
    return 0; // everything went fine
}

int main () {
     int x = 5, y = 3;
     swap(&x, &y);
     printf("x: %d, y: %d\n", x, y);
}
```



- **Ausgabe:** `x: 3, y: 5` ✓

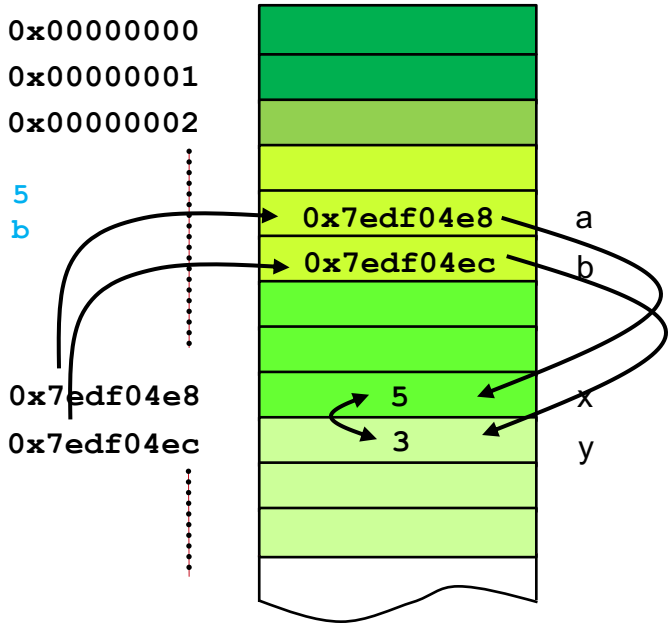
Lösung: Call by reference (pointer)

- Aufgabe: Schreibe eine Funktion, die 2 Werte vertauscht

```
// swap exchanges the values of the parameters
int swap (int* a, int* b){
    int z;

    z = *a;   // z = value a points to, i.e., x, i.e., 5
    *a = *b;  // value at address a = value at address b
    *b = z;   // value at address b = z = 5
    return 0; // everything went fine
}

int main () {
     int x = 5, y = 3;
     swap(&x, &y);
     printf("x: %d, y: %d\n", x, y);
}
```



- **Ausgabe:** `x: 3, y: 5` ✓

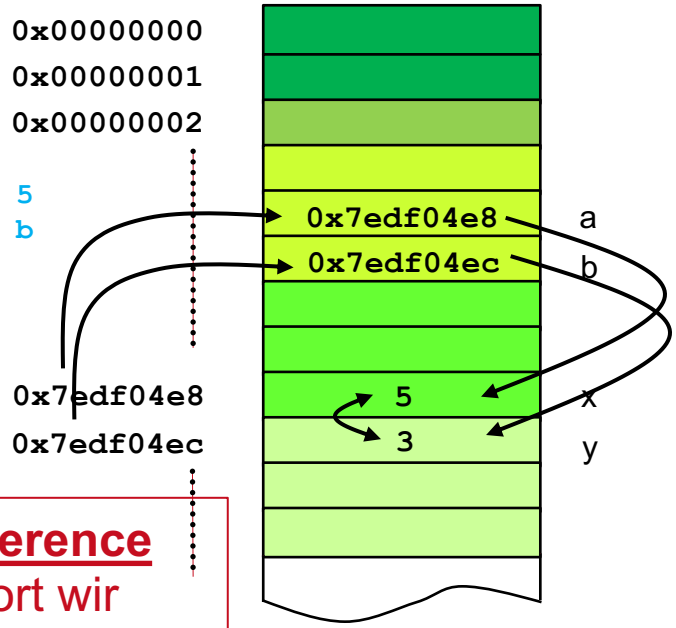
Lösung: Call by reference (pointer)

- Aufgabe: Schreibe eine Funktion, die 2 Werte vertauscht

```
// swap exchanges the values of the parameters
int swap (int* a, int* b){
    int z;

    z = *a;   // z = value a points to, i.e., x, i.e., 5
    *a = *b;  // value at address a = value at address b
    *b = z;   // value at address b = z = 5
    return 0; // everything went fine
}

int main () {
     int x = 5, y = 3;
     swap(&x, &y);
     printf("x: %d, y: %d\n", x, y);
}
```



Call by reference
Speicherort wir übergeben

- **Ausgabe:** `x: 3, y: 5` ✓

Parameterübergabe an Funktionen

- Call by Value
  - Parameterübergabe als Wert.
  - Werte der Variablen werden übergeben.
  - Damit stehen die Werte der Variablen als lokale Kopie zur Verfügung.
  - Konsequenz: Änderungen nur sichtbar **innerhalb der Funktion**

Parameterübergabe an Funktionen

- Call by Value
  - Parameterübergabe als Wert.
  - Werte der Variablen werden übergeben.
  - Damit stehen die Werte der Variablen als lokale Kopie zur Verfügung.
  - Konsequenz: Änderungen nur sichtbar **innerhalb der Funktion**
- Call by Reference
  - Parameterübergabe als Adresse.
  - Adressen der Variablen werden übergeben.
  - Damit steht die Adresse lokal zur Verfügung und es ist der Zugriff auf den Speicherort der übergebenen Variablen möglich.
  - Konsequenz: Änderungen sichtbar **über die Funktion hinaus**, d.h. die Funktion hat **Seiteneffekte**

Ausblick

VL 0 „Organisation und Inhalt“: Ablauf der Vorlesung, Termine
VL 1 „Hello World“: „Lebenswichtiges“, Programablauf, Programmierablauf, Kompilierung und Ausführung von Programmen
VL 2 „Die ersten Schritte“: Erstes C-Programm, Elementare C-Strukturen, Datentypen, Operatoren, Schleifen
VL 3 „Kontrollstrukturen & Funktionen“: Syntax, Semantik, bedingte Anweisungen, Blöcke, Sichtbarkeit
VL 4 „Rekursive Funktionen & Bibliotheken“: rekursive Funktionsaufrufe, Modularisierung
VL 5 „Typen“: Einfache und strukturierte Datentypen, Wertebereiche, Typendefinition
VL 6 „Speicher und Adressen“: Speicher, Pointer, Funktionsaufrufe „call by value“ vs. „call by reference“
VL 7 „Speicher und Arrays“: Speicher, Arrays, mehrdimensionale Arrays, Arrays und Pointer
VL 8 „Dynamische Speicherverwaltung“: Speicherallokation, Fehlerbehandlung, Rückgabewerte, Arrays/Pointer/Adressen
VL 9 „Strings, Kanäle, Git“: Strings und Arrays, Zeichensätze, Stringlänge, Ein- und Ausgabe, Arbeiten mit git