Pointer

>> Pointer (deutsch: Zeiger) bieten eine alternative Möglichkeit, Daten zwischen Funktionen auszutauschen.

Bisher haben wir alle Daten als Werte übergeben (*by value*), mit einer Ausnahme (Arrays). Bei der Wertübergabe wird nur eine Kopie der Daten weitergegeben.

Mit Pointern können wir die tatsächliche Variable selbst übergeben. Das bedeutet, dass Änderungen in einer Funktion sich auf eine andere Funktion auswirken können.

Vorher war das nicht möglich!

```c
#include <stdio.h>

void tausche(int *i, int *j)
{
    int temp = *j;
    *j = *i;
    *i = temp;
}

int main(void)
{
    int i = 5, j = 10;
    tausche(&i, &j);
    printf("%i - %i\n", i, j); // gibt 10 - 5 aus
}
```

Speicher-Grundlagen

Bevor wir in die Details von Pointern einsteigen, sollten wir uns ansehen, wie der Speicher in einem Rechner funktioniert.

Jede Datei auf Ihrem Computer liegt auf einer Festplatte (HDD, *hard-disk drive*) oder SSD (*solid-state disk*) mit Platz für viele Gigabyte (GB) an Daten.

Solche Platten bieten nur Speicherplatz; direktes Arbeiten mit Dateien ist dort nicht möglich. Datenmanipulation und -nutzung findet nur im Arbeitsspeicher (RAM, *random access memory*) statt. Deshalb müssen wir Daten dorthin bewegen, wenn wir sie benutzen wollen.

Der Arbeitsspeicher (z.B. 16 GB groß) ist im Prinzip ein großes Array aus Bytes.

Datentyp	Größe (Byte)
int	4
char	1
float	4
double	8
long long	8
string	???

Speicher als Array

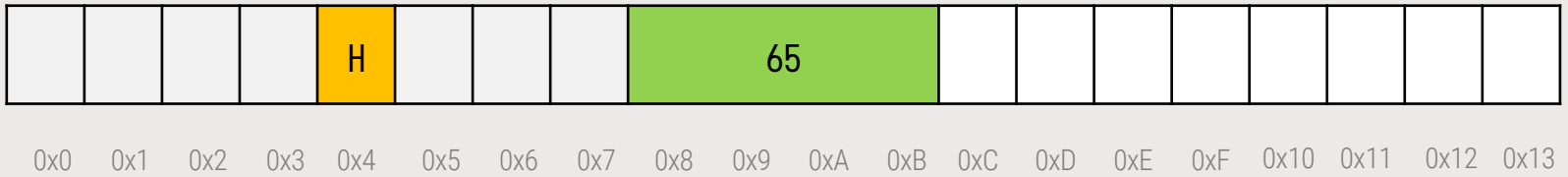
Zurück zur Vorstellung vom Speicher als großes Array aus Byte-großen Zellen.

Wie bei Arrays, die wir bereits kennen, erlaubt dies direkten Zugriff.

Wir können einzelne Elemente des Arrays durch ihren Index ansprechen – darüber haben wir direkten Zugriff auf jedes beliebige Element (sog. *random access*; anders als bei einer verketteten Liste, später mehr dazu).

Analog dazu hat jede Stelle im Speicher eine Adresse.

Beispiel




```
char c = 'H';
int speedlimit = 65;
```

POINTER SIND NUR ADRESSEN!

k

```
int k;
```

k

```
int k;
k = 5;
```

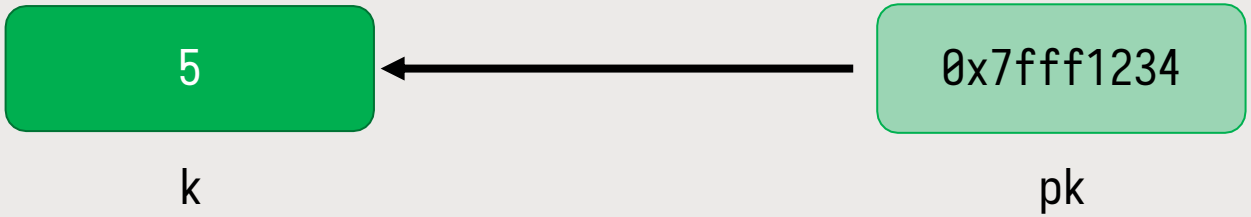
k

```
int k;
k = 5;
```

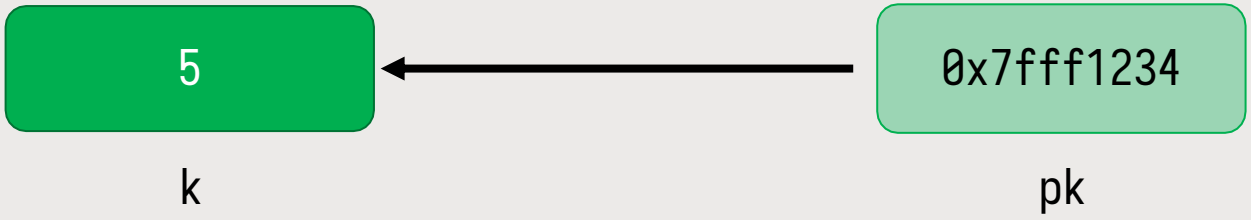
k

pk

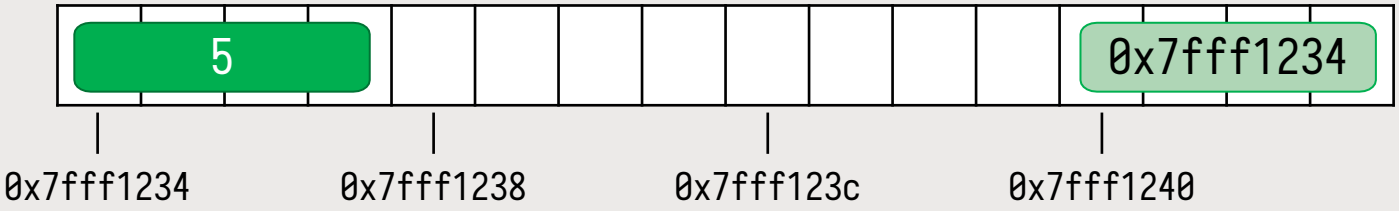
```
int k;
k = 5;
int* pk;
```

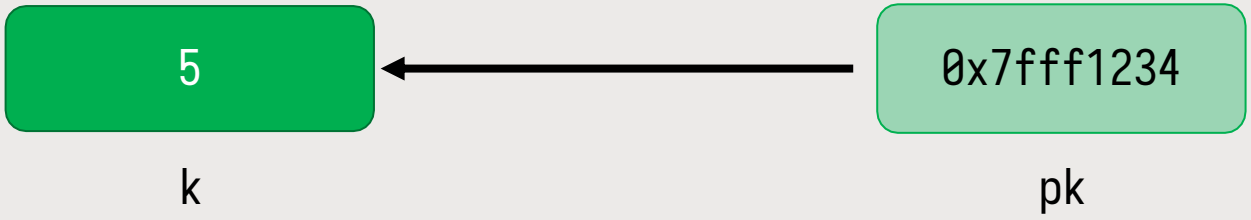



```
int k;
k = 5;
int* pk;
pk = &k;
```



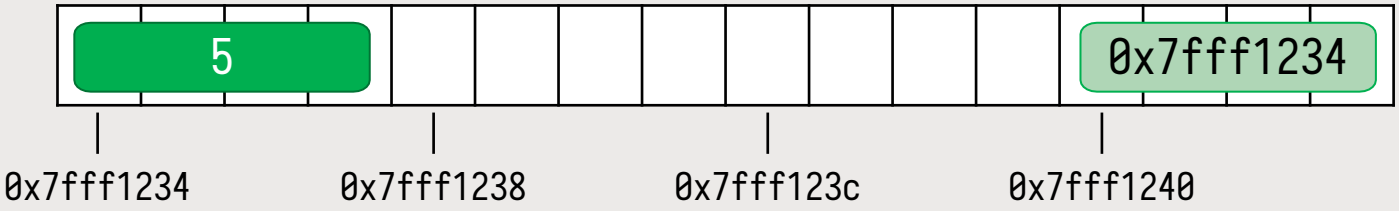

```
int k;
k = 5;
int* pk;
pk = &k;
```



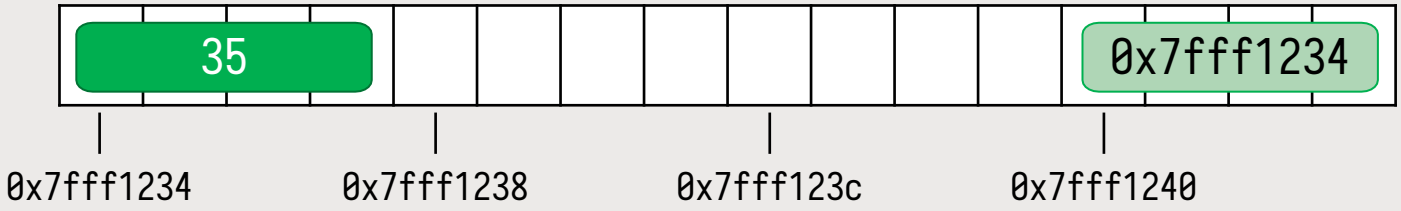


*pk ist 5 *(pk wird „dereferenziert“)*

```
int k;
k = 5;
int* pk;
pk = &k;
```




```
int k;
k = 5;
int* pk;
pk = &k;
*pk = 35; // ändert k (!) auf 35
```



NULL-Pointer

Der einfachste Pointer in C ist der NULL-Pointer (Adresse 0x0).

Er zeigt auf … nichts. Wozu nützlich?

Versucht man, von Adresse 0 zu lesen oder zu schreiben wird das Programm sofort beendet.

Wenn Sie einen Pointer erstellen und nicht sofort initialisieren, sollten Sie ihn auf NULL setzen

Sie können mit dem Gleichheitsoperator (==) prüfen, ob ein Pointer NULL ist.

Eine Möglichkeit, einen Pointer zu erstellen, ist die Adresse einer bereits existierenden Variable zu extrahieren (`&`).

Wenn `x` eine Variable vom Typ int ist, dann ist `&x` ein *Pointer to int*, dessen Wert die Adresse von `x` ist.

Pointer müssen nicht so erstellt werden!

Arrays als Pointer

Wenn `arr` ein Array von *doubles* ist, dann ist `&arr[i]` ein *Pointer to double*, dessen Wert die Adresse des *i*-ten Elements von `arr` ist.

Der Name eines Arrays ist also eigentlich nur ein Pointer auf sein erstes Element.

Sie haben die ganze Zeit schon mit Pointern gearbeitet!

```
double arr[4] = {1.0, 2.0, 3.0, 4.0};
// arr     ist äquivalent zu &arr[0]
// arr + 1 ist äquivalent zu &arr[1]
// arr + i ist äquivalent zu &arr[i]
```

Arrays als Pointer

Wenn `arr` ein Array von *doubles* ist, dann ist `&arr[i]` ein *Pointer to double*, dessen Wert die Adresse des *i*-ten Elements von `arr` ist.

Der Name eines Arrays ist also eigentlich nur ein Pointer auf sein erstes Element.

Sie haben die ganze Zeit schon mit Pointern gearbeitet!

Operationen

& ist der Adressoperator, der die Adresse einer Variable liefert. Diese kann man in einem Pointer speichern.

***** ist der *Dereferenzierungsoperator*; wird auf einen Pointer angewendet. Damit „geht“ man an die Stelle im Speicher und greift auf die Daten dort zu , entweder lesend oder schreibend (bei Zuweisung).

```
char* pc = &c;
// pc speichert die Adresse von c

char x = *pc;
// x enthält Wert an Adresse pc

*pc = 'A';
// Ändert x und c
```

```
double arr[4] = {1.0, 2.0, 3.0, 4.0};
// arr     ist äquivalent zu &arr[0]
// arr + 1 ist äquivalent zu &arr[1]
// arr + i ist äquivalent zu &arr[i]

arr[i] ist eigentlich *(arr + i)
```

Arrays als Pointer

Wenn `arr` ein Array von *doubles* ist, dann ist `&arr[i]` ein *Pointer to double*, dessen Wert die Adresse des *i*-ten Elements von `arr` ist.

Der Name eines Arrays ist also eigentlich nur ein Pointer auf sein erstes Element.

Sie haben die ganze Zeit schon mit Pointern gearbeitet!

Vorsicht: Besonderheit der Syntax

```
int* px, py, pz;  // Was bewirkt dies?
```

Eine weitere Besonderheit bei der Arbeit mit *:
es ist ein Teil des Typnamens aber auch des Variablennamens.

Besser:

```
int *pa, *pb, *pc;
```

Noch besser:

```
int* pa;
int* pb;
int* pc;
```

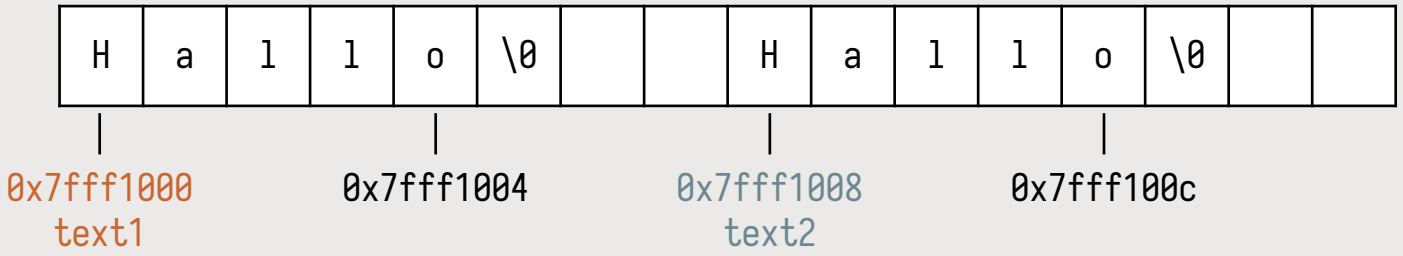
Datentyp	Größe (Byte)
int	4
char	1
float	4
double	8
long long	8
string	???

Datentyp	Größe (Byte)
int	4
char	1
float	4
double	8
long long	8
char*	???

Datentyp	Größe (Byte)
int	4
char	1
float	4
double	8
long long	8
char*	4 oder 8

Datentyp	Größe (Byte)
int	4
char	1
float	4
double	8
long long	8
char*, int*, …	4 oder 8

```c
char *text1 = "Hallo"; // enthält Adresse 0x7fff1000
char *text2 = "Hallo"; // enthält Adresse 0x7fff1008


if (text1 == text2) { // Vergleicht Adressen, nicht Inhalte!
    printf("Gleich");
}
```

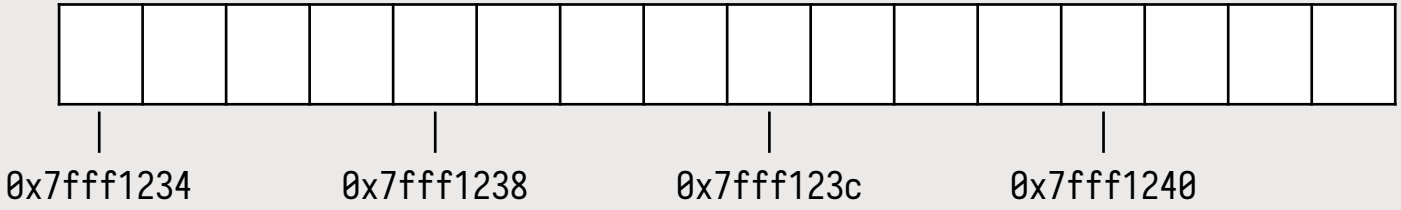


Pointer versus Arrays

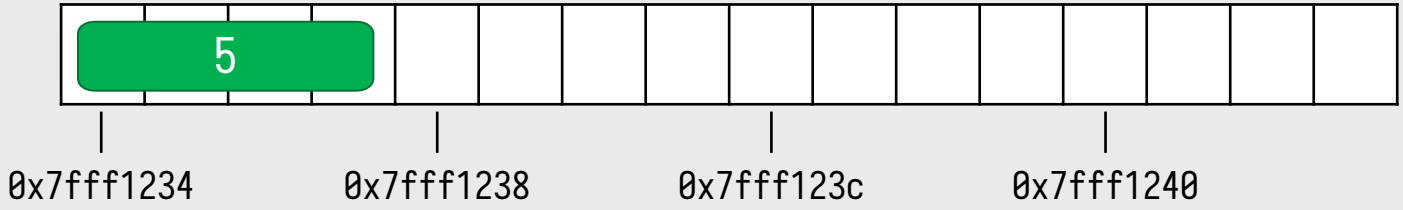
Der Name eines Arrays ist ein Pointer auf sein erstes Element und die Schreibweise `arr[i]` ist äquivalent zu `*(arr + i)`.

```
int zahlen[4] = {10, 20, 30, 40};
zahlen[2] = 35;
// ist das gleiche wie
*(zahlen + 2) = 35;


void array_func(int arr[]) {
  // arr ist eigentlich ein int*
  // Änderungen wirken sich auf Original aus
}
```

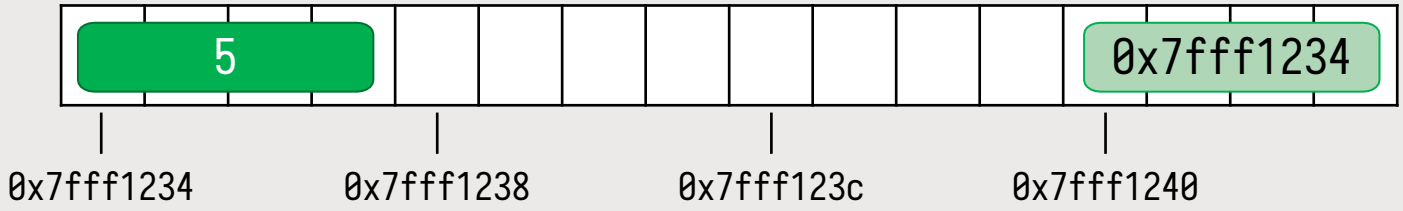
```c
#include <stdio.h>

void tausche(int *i, int *j)
{
    int temp = *j;
    *j = *i;
    *i = temp;
}

int main(void)
{
    int i = 5, j = 10;
    tausche(&i, &j);
    printf("%i - %i\n", i, j); // gibt 10 - 5 aus
}
```



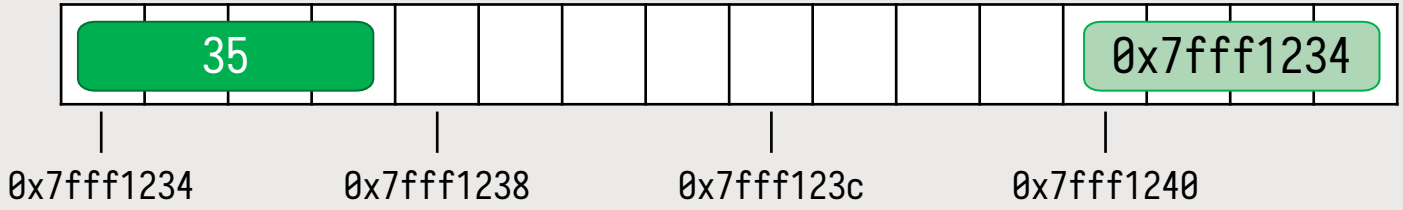

```
int k = 5;
int *pk = &k;
*pk = 35;

int m = 4;
pk = &m;
```



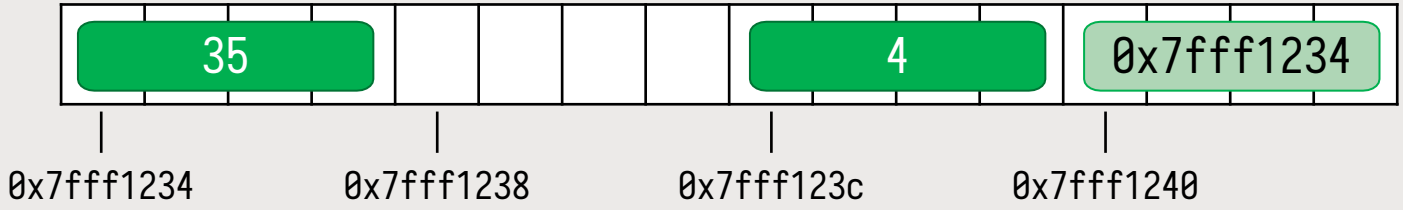

```
int k = 5;
```



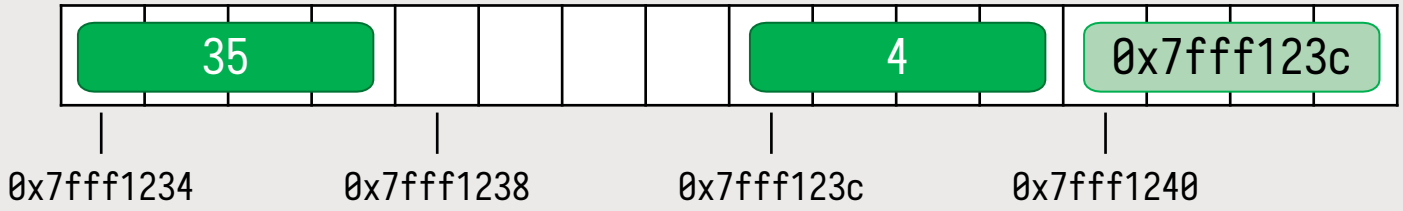

```
int k = 5;
int *pk = &k;
```




```
int k = 5;
int *pk = &k;
*pk = 35;
```




```
int k = 5;
int *pk = &k;
*pk = 35;

int m = 4;
```




```
int k = 5;
int *pk = &k;
*pk = 35;

int m = 4;
pk = &m;
```

Ein Pointer ist nichts anderes als eine Adresse

Pointer ermöglichen direkten Zugriff auf Speicherstellen

Mit & erhalten wir die Adresse einer Variable

Arrays und strings sind intern auch nur Pointer

Mit * greifen wir auf den Wert an einer Adresse zu

NULL-Pointer schützen vor undefinierten Speicherzugriffen