Zeiger

Zeiger (generell)	**Adresse** eines Objekts im Speicher

Wichtige Befehle:

Definition:	`int* ptr = address_of_type_int;`
(ohne Startwert:	`int* ptr = 0;`)
Zugriff auf Zeiger:	`ptr = otr_ptr // Pointer gets new target.`
Zugriff auf Target:	`*ptr = 5 // Target gets new value 5.`
Adresse auslesen:	`int* ptr_to_a = &a; // (a is int-variable)`
Vergleich:	`ptr == otr_ptr // Same target?`
	`ptr != otr_ptr // Different targets?`

(Anstatt `int` gehen natürlich auch andere Typen.)
(Eine `address_of_type_int` kann man durch einen anderen Zeiger oder auch mittels dem Adressoperator `&` erzeugen (siehe Beispiel unten).)

Der **Wert** des Zeigers ist die Speicheradresse des **Targets**. Will man also das Target via diesen Zeiger verändern, muss man zuerst "zu der Adresse gehen". Genau das macht der Dereferenz-Operator `*`.

Beispiel: (Gelte `int a = 5;`)

Wert von `a`:	`5`
Speicheradresse von `a`:	`0x28fef8`
Wert von `a_ptr`:	`0x28fef8`
Wert von `*a_ptr`:	`5`

Ein Zeiger kann immer nur auf den entsprechenden Typ zeigen.
(z.B. `int* ptr = &a;` Hier muss `a` Typ `int` haben.)

```
int a = 5;
int* a_ptr = &a; // a_ptr points to a
a_ptr = a;   // NOT valid (same as:  a_ptr = 5; )
             //     5 is NOT an address.
a_ptr = &a;  // valid
*a_ptr = 9; // a obtains value 9
std::cout << "a == " << a << "\n";      // Output: a == 9
std::cout << "a == " << *a_ptr << "\n"; // Output: a == 9
```

`const` (Zeiger)	**kein Schreibzugriff** auf das Target

```
int a = 5;
int b = 8;

const int* ptr = &a;
*ptr = 3; // NOT valid (write to target)
ptr = &b; // valid (write to pointer (i.e. switch target))
```

Zeiger (auf Array)	**Iterieren** über ein Array

Diese Befehle gelten **zusätzlich** zu denen unter Zeiger (generell), **falls** Zeiger auf einem **Array** verwendet werden.

Wichtige Befehle (gelte `int a[6];`):

Zeiger auf `a[0]`:	`int* ptr = a; // Works ONLY if a is ARRAY!`
temporärer Shift:	`ptr + 3`
	`ptr - 3`
permanenter Shift:	`++ptr`
	`ptr++`
	`--ptr`
	`ptr--`
	`ptr_1 += 3`
	`ptr_1 -= 3`
Distanz bestimmen:	`ptr1 - ptr2`
Position vergleichen:	`ptr1 < ptr2` (Sonst: `<=, >, >=, ==, !=`)

Die sogenannte Array-to-Pointer-Conversion erlaubt es, einen (temporären) Zeiger auf das Element beim Index `0` ganz einfach zu bekommen. Beispiele: `int* ptr = a;` oder `a + 3`

Achtung: Die grünen Shifts erzeugen einen neuen (temporären) Zeiger und verschieben `ptr` **nicht**. Die violetten Shifts verschieben aber `ptr` und geben eine Referenz auf ihn zurück. So ist beispielsweise Folgendes möglich: `++(++ptr)`

Achtung: Der Programmierer ist *selbst* dafür verantwortlich, dass Zeiger das Array nicht verlassen. (z.B. `ptr - 1` soll vermieden werden, falls `ptr` auf `a[0]` zeigt). Die einzige erlaubte Ausnahme ist der Past-the-End-Zeiger, der aber **nicht dereferenziert** werden darf.

(...)

(...)

```
// Read 6 values into an array
std::cout << "Enter 6 numbers:\n";
int a[6];
int* pTE = a+6;
for (int* i = a; i < pTE; ++i)
    std::cin >> *i; // read into target

// Output:   a[0]+a[3],  a[1]+a[4],  a[2]+a[5]
for (int* i = a; i < a+3; ++i) {
    assert((i+3) - pTE < 0); // Assert that i+3 stays inside.
                             // Same effect:   assert(i+3 < pTE);
    int sum = *i + *(i+3);
    std::cout << sum << ",  ";
}
```

Datentypen

Array	**"Massenvariable"** eines bestimmten Typs

Wichtige Befehle:

Definition: `int my_arr[5] = {2, 3, 8, -1, 3};`
Zugriff: `my_arr[2] = 8 * my_arr[3];`
(siehe auch `[]`-Operator)

(Anstatt `int` gehen natürlich auch andere Typen.)
(Die Definition kann auch ohne Initialisierung erfolgen: `int my_arr[5];`)

Die Indizes beginnen bei `0`.

Der Programmierer muss **selber** sicherstellen, dass die Indizes nicht über den Array hinausgehen.

Zuweisungen (ausser Initialisierung), Vergleiche, etc. müssen **elementweise** erfolgen.

Die Länge des Arrays **muss zum Kompilierzeitpunkt eindeutig bestimmbar sein.** (z.B. Literal oder `const`-Variable, die mittels Literal eingelesen wurde, etc.)

(...)

(...)

```
float a[10];

for (int i = 0; i < 10; ++i)
    a[i] = i; // a becomes {0 1 2 ... 9}

float b[10] = {2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
a = b; // NOT valid: array-copying is forbidden
       //     (should copy element-wise).
```

Operatoren

`my_array[...]`	**Array-** und **Vektor-Zugriff** (Subskript-Operator)
Präzedenz: 17 und Assoziativität: `links`	

```
int a[] = {1, 2, 3, 4};
a[3] = 5; // a is 1, 2, 3, 5
```

`&`	**Adressoperator** siehe: Adresse auslesen (unter `Zeiger (generell)`)
Präzedenz: 16 und Assoziativität: `rechts`	

`*`	**Dereferenz-Operator** siehe: Zugriff auf Target (unter `Zeiger (generell)`)
Präzedenz: 16 und Assoziativität: `rechts`	