Zeiger

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

Wichtige Befehle:

Definition:	`int* ptr = address_of_type_int;`
(ohne Startwert:	`int* ptr = nullptr;`)
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)	Zeiger **Konstantheit**

Es gibt zwei Arten von Konstantheit:

kein Schreibzugriff auf Target: `const int* a_ptr = &a;`
kein Schreibzugriff auf Zeiger: `int* const a_ptr = &a;`

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

const int* ptr_1 = &a;
*ptr_1 = 3; // NOT valid (change target)
ptr_1 = &b; // valid (change pointer)

int* const ptr_2 = &a;
*ptr_2 = 3; // valid (change target)
ptr_2 = &b; // NOT valid (change pointer)

const int* const ptr_3 = &a;
*ptr_3 = 3; // NOT valid (change target)
ptr_3 = &b; // NOT valid (change pointer)
```

Zeiger	**Iterieren**

Wichtige Befehle:

Zeiger:	`int* ptr = new int[6];`
temporärer Shift:	`ptr + 3`
	`ptr - 3`
permanenter Shift:	`++ptr`
	`ptr++`
	`--ptr`
	`ptr--`
	`ptr += 3`
	`ptr -= 3`
Distanz bestimmen:	`ptr1 - ptr2`
Position vergleichen:	`ptr1 < ptr2` (Sonst: `<=, >, >=, ==, !=`)

Achtung: Die grünen Shifts erzeugen einen neuen (temporären) Zeiger und verschieben `ptr` **nicht**. Die violetten Shifts verschieben aber `ptr`.

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

// 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); // Assert that i+3 stays inside.
    std::cout << (*i + *(i+3)) << ",  ";
}
```

`*this`	Zugriff auf **implizites Argument**

Memberfunktionen einer Klasse haben ein implizites Argument, nämlich das aufrufende Objekt. Und `this` ist ein Zeiger darauf. Via `*this` kann man darauf zugreifen.

Bei Zugriffen von innerhalb einer Klasse aus auf Daten-Member oder Member-Funktionen wird das implizite Argument automatisch verwendet. Man muss es dann also nicht unbedingt explizit angeben (siehe Eintrag `Memberfunktion`). Man muss `*this` aber mindestens explizit verwenden, falls z.B. eine Referenz auf das implizite Argument zurückgegeben werden soll.

```
// General example
class Human {
public:
    void set (const int a) { age = a; } // or (*this).age = a;
    void print1 () const { std::cout << (*this).age; }
    void print2 () const { std::cout << age; } // equivalent
private:
    int age;
};
...
Human me;  me.set(175);
me.print1();  // 175
me.print2();  // 175
-----------------------------------------------------------------------
// Another example
class Complex {
public:
    // Note: In most applications
    // a reference should be returned.
    Complex& operator+= (const Complex& b) {
        real += b.real;
        imag += b.imag;
        return *this;
    }
    ... // other members
private:
    float real;
    float imag;
};
```

Dynamische Datentypen

`new`	**Objekt mit dynamischer Lebensdauer** erstellen.

Mit `new` wird ein Objekt erstellt, indem der nötige Speicherplatz reserviert wird, und dann ein gegebener Konstruktor aufgerufen wird.

Der Rückgabewert von `new` ist ein *Pointer* auf das neu erstellte Objekt.

```
Class My_Class {
public:
    My_Class (const int i) : y (i) { std::cout << "Hello"; }
    int get_y () { return y; }
private:
    int y;
};

...
My_Class* ptr = new My_Class (3); // outputs Hello
My_Class* ptr2 = ptr; // another pointer to the new object
std::cout << (*ptr).get_y(); // Output: 3
...
```

`new ...[]`	**Ranges mit dynamischer Lebensdauer und Länge** erstellen.

```
int n; std::cin >> n;
int* range = new int[n];
// Read in values to the range
for (int* i = range; i < range + n; ++i) std::cin >> *i;
```

Operatoren

`&`	**Adressoperator** (siehe: *Adresse auslesen* unter `Zeiger (generell)`, Summary 8)

Präzedenz: 16 und Assoziativität: `rechts`

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

`...->...`	**Auf einen Member eines Objekts zugreifen**, auf das ein Pointer gegeben ist.
`ptr->mem` macht das Selbe wie `(*ptr).mem`.	

```
struct my_class {
    // POST: "Hi! " is written to std::cout
    void say_hi () const { std::cout << "Hi! "; }
};
...
my_class obj;
my_class* ptr = &obj; // just a pointer to obj
obj.say_hi();         // direct access
ptr->say_hi();        // using ->
(*ptr).say_hi();      // equivalent
```