Zeiger und Iteratoren

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 << ",  ";
}
```

Iterator (auf Vektor)	**Iterieren** über einen Vektor.

Im Folgenden wird nur auf die Unterschiede zum Zeiger (auf Array) eingegangen. Die restliche Bedienung erfolgt gleich.

Erfordert: `#include <vector>`

Wichtige Befehle (gelte `std::vector<int> a (6);`):

Definition: `std::vector<int>::iterator itr = itr_of_type_int;`
Iterator auf `a[0]`: `a.begin()`
Past-the-End-Iterator: `a.end()`

Vektoren unterstützen **keine *automatische* Konvertierung** zu einem Iterator auf das Element mit Index `0`:

```
int* ptr = arr; // Works ONLY if arr is ARRAY!
std::vector<int> itr = vec.begin(); // Counterpart for VECTORS
```

Dafür haben Vektoren im Gegensatz zu Pointern einen bequemen Schnellzugriff auf den Past-the-End-Iterator: `a.end()`

(...)

(...)

```
// Same example as for arrays, but now for vectors.
// To avoid the lengthy lines see entry on using.

// Read 6 values into a vector
std::cout << "Enter 6 numbers:\n";
std::vector<int> a (6);
for (std::vector<int>::iterator i = a.begin(); i < a.end(); ++i)
    std::cin >> *i; // read into target of iterator

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

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

Vorsicht: Einen `const`-Iterator erzeugt man mittels
`std::vector<int>::const_iterator ...`
und **nicht** mittels
`const std::vector<int>::iterator ...`

Die zweite Version erzeugt einen Iterator, den man nicht herumschieben kann. In dieser Vorlesung gehen wir aber nur auf die Iteratoren näher ein, welche den Schreibzugriff auf das Target verbieten (erste Variante oben).

```
std::vector<int> a (6, -8); // a is: -8 -8 -8 -8 -8 -8

std::vector<int>::const_iterator itr = a.begin() + 3;
*itr = 4;        // NOT valid
itr = a.begin(); // valid (itr now points to a[0])
```

Datentypen

`using new = old;`	Lange **Datentyp-Namen verkürzen**.

```
// Same example as for vectors, but now with using:
using Vec = std::vector<int>;
using It = std::vector<int>::iterator;

// Read 6 values into a vector
std::cout << "Enter 6 numbers:\n";
Vec a (6);
for (It i = a.begin(); i < a.end(); ++i)
    std::cin >> *i; // read into target of iterator

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

Vektoren

`....push_back(v)`	Zusätzliches Element `v` **hinten anhängen**

Erfordert: `#include <vector>`

`push_back` ist sehr stark, wenn man eine unbekannte Anzahl Inputs einlesen soll.

```
// General functionality
std::vector<int> my_vec (5, 0); // my_vec: 0 0 0 0 0
my_vec.push_back(7);            // my_vec: 0 0 0 0 0 7

// Read unknown number of inputs
std::vector<int> input_container (0); // vector of length 0
int i;
while (std::cin >> i)
    input_container.push_back(i);
```

Standard-Funktionen auf Arrays, Vektoren, Sets, Strings, ...

`std::fill(b, p, val)`	Wert `val` in einen **Bereich** `[b,p)` **einlesen**

Erfordert: `#include <algorithm>`

```
// Goal: Generate vector: 4 4 4 2 2
std::vector<int> vec (5, 4);              // vec: 4 4 4 4 4
std::fill(vec.begin()+3, vec.end(), 2);   // vec: 4 4 4 2 2
```

`std::find(b, p, val)`	`val` **suchen** im Bereich `[b,p)`

Erfordert: `#include <algorithm>`

Zurückgegeben wird ein **Iterator** auf das *erste* gefundene Vorkommnis.

Wenn `std::find` nicht fündig wird, gibt es den Past-the-End-Iterator `p` zurück. (Beachte: Past-the-End ist bezüglich Bereich `[b,p)` gemeint.)

```
using It =std::vector<int>::iterator;
std::vector<int> vec = {8, 1, 0, -7, 7};

// Goal: Find index of -7 in vec: 8 1 0 -7 7
It pos_itr = std::find(vec.begin(), vec.end(), -7);
std::cout << (pos_itr - vec.begin()) << "\n"; // Output: 3
```

`std::sort(b, e)`	Bereich `[b, e)` **sortieren**

Erfordert: `#include <algorithm>`

`std::sort` funktioniert nur, wenn Random-Access Iteratoren für `b` und `e` übergeben werden. Somit funktioniert `std::sort` z.B. für Felder und Vektoren, aber nicht z.B. für Sets.

```
std::vector<int> vec = {8, 1, 0, -7, 7};
std::sort(vec.begin(), vec.end()); // vec:  -7 0 1 7 8
```

`std::min_element(b, p)`	**Iterator auf Minimum** im Bereich `[b,p)`

Erfordert: `#include <algorithm>`

Wenn das Minimum nicht eindeutig ist, so wird ein Iterator auf das erste Vorkommnis zurückgegeben.

```
// Goal: Make sure that all inputs are > 0
std::vector<int> vec;
int i;
while (std::cin >> i)
    vec.push_back(i);

assert( *std::min_element(vec.begin(), vec.end()) > 0 );
    // Note: We have to dereference the (r-value-)iterator.
```

Operatoren

`&`	**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`	