Iteratoren

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 typedef.

// 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

`typedef old new;`	Lange **Datentyp-Namen verkürzen**.

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

// 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 << ",  ";
}
```

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.)

```
typedef std::vector<int>::iterator It;
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.
```