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, 0);`):

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

Anstelle des `...` in der Definition eines Iterators müssen andere Iteratoren stehen (z.B. `a.begin()`).

```
// Example 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, 0);
for (std::vector<int>::iterator i = a.begin(); i < a.end(); ++i)
    std::cin >> *i; // read into object of iterator

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

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

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 Objekt* 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.at(0))
```

Bereichsbasierte for-Schleife	

Sequenzielle Iteration mittels eines Iterators über einen `llvec` (const-Iteratormöglich; andere Container möglich):

```
llvec v(3); // v == 0, 0, 0
for (llvec::iterator it = v.begin(); it != v.end(); ++it) {
   std::cout << *it; // 000
}
```

Kann alternativ auch wie folgt geschrieben werden:

```
for (int i : v) std::cout << i; // 000
```

Wird dann zu Iterator-basierter Schleife übersetzt.

Modifizierender Zugriff ist auch möglich:

```
for (int& i : v) i += 3;
for (int i : v) std::cout << i; // 369
```

Datentypen

`set`	Datentyp für **Mengen** (jedes Element kommt nur einmal vor).

Erfordert: `#include<set>`

Wichtige Befehle (Sei `b = some_vec.begin(); e = some_vec.end();`):

Definition: `std::set<int> my_set (b, e);`
(Initialisiert `my_set` mit den Werten im Bereich `[b,e)`.)

Die **Iteratoren** der `set`s funktionieren wie die Iteratoren der Vektoren, **aber:**

Keine: `[], +, -, <, >, <=, >=, +=, -=`
Zum **Verschieben** nur: `++..., ...++, --..., ...--, =`
Zum **Vergleichen** nur: `==, !=`

```
// Determine All Occurring Numbers
std::cout << "Enter 100 numbers:\n";
std::vector<int> nbrs (100);
for (int i = 0; i < 100; ++i)
    std::cin >> nbrs.at(i);

std::set<int> uniques (nbrs.begin(), nbrs.end());

// Output
typedef std::set<int>::iterator Sit;
for (Sit i = uniques.begin(); i != uniques.end(); ++i)
    std::cout << *i << "  ";

// This does not work:
for (int i = 0; i < uniques.end() - uniques.begin(); ++i)
    std::cout << uniques.at(i);
```

Standard-Funktionen auf Arrays, Vektoren, ...

`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 Vit;
std::vector<int> vec (5, 2);
vec.at(3) = -7

// Goal: Find index of -7 in vec: 2 2 2 -7 2
Vit 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 (10, 0);
for (int i = 0; i < 10; ++i)
    std::cin >> vec.at(i);

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