C++ Iteratoren und Algorithmen

13. Oktober 2019

Swiss Olympiad in Informatics

Iteratoren

Iterator - Beispiel

Einen `vector<int>` `a` iterieren :

```
for (int i = 0; i < a.size(); ++i) {
    print(a[i]);
}
```

Iterator - Beispiel

Einen `vector<int> a` iterieren :

```
for (int i = 0; i < a.size(); ++i) {
    print(a[i]);
}
```

```
for (vector<int>::iterator it = a.begin(); it != a.end(); ++it) {
    print(*it);
}
```

Iterator - Definition

Definition : Ein Iterator ist ein Zeiger, der auf ein Element einer Datenstruktur zeigt. Er erlaubt es, über eine Datenstruktur zu iterieren.

Iterator - Definition

Definition : Ein Iterator ist ein Zeiger, der auf ein Element einer Datenstruktur zeigt. Er erlaubt es, über eine Datenstruktur zu iterieren.

Ein Iterator lässt sich verwenden mit :

- Dem **Inkrementierungsoperator** (++) um zum Iterator, der auf das nächste Element zeigt, vorzurücken.
- Dem **Dereferenzierungsoperator** (*) welcher auf das aktuell selektierte Element zugreift.

Iterator - Beispiel

```
for (vector<int>::iterator it = a.begin(); it != a.end(); ++it) {
    print(*it);
}
```

```
for (int i = 0; i < a.size(); ++i) {
    print(a[i]);
}
```

Iterator - Beispiel

```
for (vector<int>::iterator it = a.begin(); it != a.end(); ++it) {
    print(*it);
}
```

```
for (int i = 0; i < a.size(); ++i) {
    print(a[i]);
}
```

- `a.begin()` entspricht dem Iterator, welcher auf `a[0]` zeigt.

Iterator - Beispiel

```
for (vector<int>::iterator it = a.begin(); it != a.end(); ++it) {
    print(*it);
}
```

```
for (int i = 0; i < a.size(); ++i) {
    print(a[i]);
}
```

- `a.begin()` entspricht dem Iterator, welcher auf `a[0]` zeigt.
- `++it` rückt den Iterator `it` von `a[i]` auf `a[i+1]` vor.

Iterator - Beispiel

```
for (vector<int>::iterator it = a.begin(); it != a.end(); ++it) {
    print(*it);
}
```

```
for (int i = 0; i < a.size(); ++i) {
    print(a[i]);
}
```

- `a.begin()` entspricht dem Iterator, welcher auf `a[0]` zeigt.
- `++it` rückt den Iterator `it` von `a[i]` auf `a[i+1]` vor.
- `a.end()` enspricht dem Iterator, welcher auf eine Position direkt nach dem letzten Wert des Vektors zeigt.

Iterator - Beispiel

```
for (vector<int>::iterator it = a.begin(); it != a.end(); ++it) {
    print(*it);
}
```

```
for (int i = 0; i < a.size(); ++i) {
    print(a[i]);
}
```

- `a.begin()` entspricht dem Iterator, welcher auf `a[0]` zeigt.
- `++it` rückt den Iterator `it` von `a[i]` auf `a[i+1]` vor.
- `a.end()` enspricht dem Iterator, welcher auf eine Position direkt nach dem letzten Wert des Vektors zeigt.
- `*it` greift auf das aktuell gewählte Element zu.

Achtung!

Wenn a vom Typ `vector<vector<int> >` ist:

```
for (vector<vector<int> >::iterator it = a.begin(); it != a.end(); ++it) {
    print(*it.size());
}
```

Achtung!

Wenn a vom Typ `vector<vector<int> >` ist:

```
for (vector<vector<int> >::iterator it = a.begin(); it != a.end(); ++it) {
    print(*it.size());
}
```

`*it.size()` wird interpretiert als `*(it.size())`. Deswegen ist es wichtig, `(*it).size()` zu verwenden.

Finde den Fehler

Setze alle Werte von a auf 6 :

```
vector<int> a{1, 4, 5, 5, 2, 5};
for (vector<int>::iterator it = a.begin(); it != a.end(); ++it) {
    it = 6;
} // a soll jetzt 6, 6, 6, 6, 6, 6 sein
```

Wo ist der Fehler?

Finde den Fehler

Setze alle Werte von a auf 6 :

```
vector<int> a{1, 4, 5, 5, 2, 5};
for (vector<int>::iterator it = a.begin(); it != a.end(); ++it) {
    it = 6;
} // a soll jetzt 6, 6, 6, 6, 6, 6 sein
```

Wo ist der Fehler? Der Iterator `it` zeigt lediglich auf ein Element. Wenn man das Element verändern will, muss man den Iterator dereferenzieren. In diesem Fall mit `*it = 6;`.

Anwendungen von Iteratoren

- Sie können gleich wie Indexvariablen iterieren.
- Die Syntax ist komplizierter.

... Aber :

Anwendungen von Iteratoren

- Sie können gleich wie Indexvariablen iterieren.
- Die Syntax ist komplizierter.

... Aber :

- Iteratoren erlauben es, Daten unabhängig von ihrer Struktur zu verwenden.

Anwendungen von Iteratoren

- Sie können gleich wie Indexvariablen iterieren.
- Die Syntax ist komplizierter.

... Aber :

- Iteratoren erlauben es, Daten unabhängig von ihrer Struktur zu verwenden.
- Selbst für Datenstrukturen, wo die Notierung `a[i]` keinen Sinn macht. z.B. Mengen.

Algorithmen mit Iteratoren

Anwendungen von Iteratoren

Was hilft es uns, Daten unabhängig von ihrer Struktur weitergeben zu können?

Sortieren

Die Funktion `sort` sortiert die Elemente einer Datenstruktur in aufsteigender Ordnung:

```
vector<int> a{1, 4, 5, 5, 2, 5};
sort(a.begin(), a.end()); // wird zu 1, 2, 4, 5, 5, 5
```

Sortieren

Die Funktion `sort` sortiert die Elemente einer Datenstruktur in aufsteigender Ordnung:

```
vector<int> a{1, 4, 5, 5, 2, 5};
sort(a.begin(), a.end()); // wird zu 1, 2, 4, 5, 5, 5
```

Generell gilt:

```
sort(start, ende);
```

Sortiert die Elemente zwischen `start` (inklusive) und `ende` (exklusive).

Sortieren

Die Funktion `sort` sortiert die Elemente einer Datenstruktur in aufsteigender Ordnung:

```
vector<int> a{1, 4, 5, 5, 2, 5};
sort(a.begin(), a.end()); // wird zu 1, 2, 4, 5, 5, 5
```

Generell gilt:

```
sort(start, ende);
```

Sortiert die Elemente zwischen `start` (inklusive) und `ende` (exklusive).
Wie sortiert man alle Elemente eines Vektors ausser dem ersten und den zwei letzten Elementen?

Sortieren

Die Funktion `sort` sortiert die Elemente einer Datenstruktur in aufsteigender Ordnung:

```
vector<int> a{1, 4, 5, 5, 2, 5};
sort(a.begin(), a.end()); // wird zu 1, 2, 4, 5, 5, 5
```

Generell gilt:

```
sort(start, ende);
```

Sortiert die Elemente zwischen `start` (inklusive) und `ende` (exklusive).
Wie sortiert man alle Elemente eines Vektors ausser dem ersten und den zwei letzten Elementen?

```
vector<int> a{9, 4, 2, 5, 2, 5};
sort(a.begin()+1, a.end()-2); // wird zu 9, 2, 4, 5, 2, 5
```

Reihenfolge umkehren

Die Funktion `reverse` dreht die Reihenfolge der Elemente eines Vektors um:

```
vector<int> a{1, 2, 3, 4, 5};
reverse(a.begin(), a.end()); // a ist 5, 4, 3, 2, 1
```

Elemente rotieren

Die Funktion `rotate` verschiebt die Elemente eines Vektors nach links:

```
vector<int> a{1, 2, 3, 4, 5};
rotate(a.begin(), a.begin() + 1, a.end()); // a ist 2, 3, 4, 5, 1
rotate(a.begin(), a.begin() + 1, a.end()); // a ist 3, 4, 5, 1, 2
rotate(a.begin(), a.begin() + 3, a.end()); // a ist 1, 2, 3, 4, 5
```

Einen Wert suchen

```
vector<int> a{1, 4, 5, 5, 2, 5};
vector<int>::iterator it = find(a.begin(), a.end(), 5);
if (it == a.end()) {
    print("Der Wert existiert nicht!");
} else {
    print("Der Wert", *it, "wurde gefunden an Position ", it - a.begin());
}
```

Einen Wert suchen

```
vector<int> a{1, 4, 5, 5, 2, 5};
vector<int>::iterator it = find(a.begin(), a.end(), 5);
if (it == a.end()) {
    print("Der Wert existiert nicht!");
} else {
    print("Der Wert", *it, "wurde gefunden an Position ", it - a.begin());
}
```

- Es ist wichtig, immer die Option zu berücksichtigen wo der Wert nicht gefunden wird!

Einen Wert suchen

```
vector<int> a{1, 4, 5, 5, 2, 5};
vector<int>::iterator it = find(a.begin(), a.end(), 5);
if (it == a.end()) {
    print("Der Wert existiert nicht!");
} else {
    print("Der Wert", *it, "wurde gefunden an Position ", it - a.begin());
}
```

- Es ist wichtig, immer die Option zu berücksichtigen wo der Wert nicht gefunden wird!
- `it - a.begin()` funktioniert nur bei Vektoren!

Das Minimum finden

```
vector<int> a{1, 4, 5, 5, 2, 5};
vector<int>::iterator it = min_element(a.begin(), a.end());
if (it == a.end()) {
    print("Das Minimum existiert nicht!"); // Der Vektor ist leer
} else {
    print("Das Minimum", *it, "ist an Position", it - a.begin());
}
```

Das Minimum finden

```
vector<int> a{1, 4, 5, 5, 2, 5};
vector<int>::iterator it = min_element(a.begin(), a.end());
if (it == a.end()) {
    print("Das Minimum existiert nicht!"); // Der Vektor ist leer
} else {
    print("Das Minimum", *it, "ist an Position", it - a.begin());
}
```

Auch hier gibt es einen Sonderfall, wenn der Vektor leer ist.

Vorkommnisse eines Elements zählen

```
vector<int> a{1, 4, 5, 5, 2, 5};
print("Die Zahl 5 kommt", count(a.begin(), a.end(), 5), "Mal in a vor");
// zeigt "Die Zahl 5 kommt 3 Mal in a vor"
```

Einen Vektor füllen

```
vector<int> a{1, 4, 5, 5, 2, 5};
fill(a.begin(), a.end, 0); // a : 0, 0, 0, 0, 0, 0
```

Einen Vektor füllen

```
vector<int> a{1, 4, 5, 5, 2, 5};
fill(a.begin(), a.end, 0); // a : 0, 0, 0, 0, 0, 0
```

```
vector<int> a(6); // a : 0, 0, 0, 0, 0, 0
iota(a.begin(), a.end(), 5); // a : 5, 6, 7, 8, 9, 10
```

Elemente entfernen

```
vector<int> a{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
a.erase(a.end() - 2); // a : 0, 1, 2, 3, 4, 5, 6, 7, 9
a.erase(a.begin() + 3, a.begin() + 5); // a : 0, 1, 2, 5, 6, 7, 9
a.erase(a.begin(), a.end()); // a ist leer
```

Elemente entfernen

```
vector<int> a{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
a.erase(a.end() - 2); // a : 0, 1, 2, 3, 4, 5, 6, 7, 9
a.erase(a.begin() + 3, a.begin() + 5); // a : 0, 1, 2, 5, 6, 7, 9
a.erase(a.begin(), a.end()); // a ist leer
```

Achtung, `a.erase` verwendet eine andere Syntax.

Doppelte Elemente entfernen

```
vector<int> a{1, 4, 5, 4, 5, 2, 5};
sort(a.begin(), a.end()); // sortiert den Vektor
// a : 1, 2, 4, 4, 5, 5, 5
vector<int>::iterator it = unique(a.begin(), a.end());
//\"uberschreibt doppelte Elemente, gibt einen Iterator auf das neue Ende zur\"
    uck
// a : 1, 2, 4, 5, 5, 5, 5
a.erase(it, a.end()); //entfernt \"uberfl\"ussige Elemente am Ende des Vektors
// a : 1, 2, 4, 5
```

Was genau macht `unique(a.begin(), a.end())`?

Doppelte Elemente entfernen

```
vector<int> a{1, 4, 5, 4, 5, 2, 5};
sort(a.begin(), a.end()); // sortiert den Vektor
// a : 1, 2, 4, 4, 5, 5, 5
vector<int>::iterator it = unique(a.begin(), a.end());
//\"uberschreibt doppelte Elemente, gibt einen Iterator auf das neue Ende zur\"
    uck
// a : 1, 2, 4, 5, 5, 5, 5
a.erase(it, a.end()); //entfernt \"uberfl\"ussige Elemente am Ende des Vektors
// a : 1, 2, 4, 5
```

Was genau macht `unique(a.begin(), a.end())`?

- Es verschiebt Elemente nach vorne, sodass nie zwei gleiche Elemente direkt aufeinander folgen.

Doppelte Elemente entfernen

```
vector<int> a{1, 4, 5, 4, 5, 2, 5};
sort(a.begin(), a.end()); // sortiert den Vektor
// a : 1, 2, 4, 4, 5, 5, 5
vector<int>::iterator it = unique(a.begin(), a.end());
//\"uberschreibt doppelte Elemente, gibt einen Iterator auf das neue Ende zur\"
    uck
// a : 1, 2, 4, 5, 5, 5, 5
a.erase(it, a.end()); //entfernt \"uberfl\"ussige Elemente am Ende des Vektors
// a : 1, 2, 4, 5
```

Was genau macht `unique(a.begin(), a.end())`?

- Es verschiebt Elemente nach vorne, sodass nie zwei gleiche Elemente direkt aufeinander folgen.
- Es gibt den Iterator zurück, der auf das neue logische Ende der Liste zeigt.

Funktionen als Argument

Motivation

Mithile von Funktionen lassen sich genauere Bedingungen formulieren.

Ein Element mit einer bestimmten Eigenschaft finden

```
#include <soi>

bool is_prime(int n) {
    for (int i = 2; i < n; i++){
        if (n % i == 0) return false;
    }
    return n > 1;
}

int main() {
    vector<int> a{4, 6, 3, 5, 8, 4, 2, 7, 8};
    vector<int>::iterator it = find_if(a.begin(), a.end(), is_prime);
    if (it != a.end()) {
        print("Primzahl gefunden: ", *it);
    } else {
        print("Keine Primzahl gefunden.");
    }
}
```

Ein Element mit einer bestimmten Eigenschaft finden

```
#include <soi>

bool is_prime(int n) {
    for (int i = 2; i < n; i++){
        if (n % i == 0) return false;
    }
    return n > 1;
}

int main() {
    vector<int> a{4, 6, 3, 5, 8, 4, 2, 7, 8};
    vector<int>::iterator it = find_if(a.begin(), a.end(), is_prime);
    if (it != a.end()) {
        print("Primzahl gefunden: ", *it);
    } else {
        print("Keine Primzahl gefunden.");
    }
}
```

Was ist die Laufzeitkomplexität?

Alle Werte, die eine Bedingung erfüllen entfernen

```cpp
#include <soi>

bool ungerade(int n) {
    return (n % 2 == 1);
}

int main() {
    vector<int> a{2, 6, 3, 5, 8, 4, 2, 7, 8};
    vector<int>::iterator it = remove_if(a.begin(), a.end(), ungerade);
    // a : 2 6 8 4 2 8 2 7 8
    a.erase(it, a.end());
    // a : 2 6 8 4 2 8
}
```

- `remove_if` verschiebt alle gültigen Elemente an den Anfang.

Alle Werte, die eine Bedingung erfüllen entfernen

```cpp
#include <soi>

bool ungerade(int n) {
    return (n % 2 == 1);
}

int main() {
    vector<int> a{2, 6, 3, 5, 8, 4, 2, 7, 8};
    vector<int>::iterator it = remove_if(a.begin(), a.end(), ungerade);
    // a : 2 6 8 4 2 8 2 7 8
    a.erase(it, a.end());
    // a : 2 6 8 4 2 8
}
```

- `remove_if` verschiebt alle gültigen Elemente an den Anfang.
- Gibt einen Iterator auf das neue logische Ende zurück.

Andere hilfreiche Funktionen

- `count_if` Zählt die Anzahl der Elemente, die eine Bedingung erfüllen.

Andere hilfreiche Funktionen

- `count_if` Zählt die Anzahl der Elemente, die eine Bedingung erfüllen.
- `replace_if` Ersetzt alle Elemente, die eine Bedingung erfüllen, mit einem anderen Wert.

Andere hilfreiche Funktionen

- `count_if` Zählt die Anzahl der Elemente, die eine Bedingung erfüllen.
- `replace_if` Ersetzt alle Elemente, die eine Bedingung erfüllen, mit einem anderen Wert.
- `partition` Verschiebt alle Werte, die eine Bedingung erfüllen, an den Anfang und alle anderen an das Ende.

Andere hilfreiche Funktionen

- `count_if` Zählt die Anzahl der Elemente, die eine Bedingung erfüllen.
- `replace_if` Ersetzt alle Elemente, die eine Bedingung erfüllen, mit einem anderen Wert.
- `partition` Verschiebt alle Werte, die eine Bedingung erfüllen, an den Anfang und alle anderen an das Ende.
- `transform` Wendet eine Funktion auf alle Werte an.
- und viele mehr ...

Fragen

Fragen?