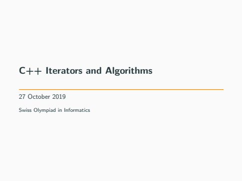
C++ Iterators and Algorithms

27 October 2019

Swiss Olympiad in Informatics



C++ Iterators and Algorithms

27 October 2019

Swiss Olympiad in Informatics

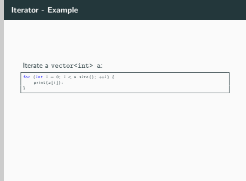
Iterators

Iterators

Iterator - Example

Iterate a `vector<int>` a:

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



Iterator - Example

Iterate a vector<int> a:

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

Iterator - Example

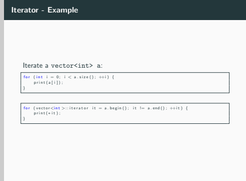
Iterate a `vector<int>` a:

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

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

2019-10-27

C++ Iterators and Algorithms
└Iterators
└Iterator - Example


Iterator - Example

Iterate a vector<int> a:

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

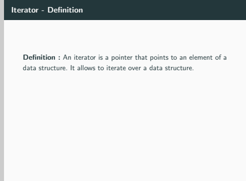
Definition : An iterator is a pointer that points to an element of a data structure. It allows to iterate over a data structure.

2019-10-27

C++ Iterators and Algorithms

└Iterators

└Iterator - Definition



Iterator - Definition

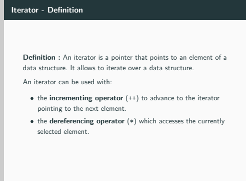
Definition : An iterator is a pointer that points to an element of a data structure. It allows to iterate over a data structure.

Iterator - Definition

Definition : An iterator is a pointer that points to an element of a data structure. It allows to iterate over a data structure.

An iterator can be used with:

- the **incrementing operator** (++) to advance to the iterator pointing to the next element.
- the **dereferencing operator** ($*$) which accesses the currently selected element.



Iterator - Definition

Definition : An iterator is a pointer that points to an element of a data structure. It allows to iterate over a data structure.

An iterator can be used with:

- the **incrementing operator** (++) to advance to the iterator pointing to the next element.
- the **dereferencing operator** ($*$) which accesses the currently selected element.

Iterator - Example

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

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

2019-10-27

C++ Iterators and Algorithms
└─Iterators
└─Iterator - Example

Iterator - Example

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

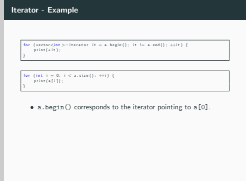
```
for (int i = 0; i < a.size(); ++i) {
    print(a[i]);
}
```

Iterator - Example

```
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()` corresponds to the iterator pointing to `a[0]`.



Iterator - Example

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

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

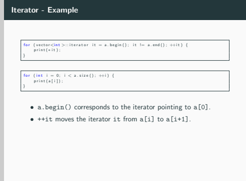
- `a.begin()` corresponds to the iterator pointing to `a[0]`.

Iterator - Example

```
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()` corresponds to the iterator pointing to `a[0]`.
- `++it` moves the iterator `it` from `a[i]` to `a[i+1]`.



Iterator - Example

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

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

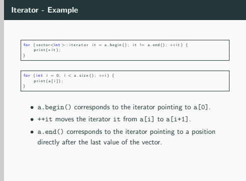
- a.begin() corresponds to the iterator pointing to a[0].
- ++it moves the iterator it from a[i] to a[i+1].

Iterator - Example

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

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

- `a.begin()` corresponds to the iterator pointing to `a[0]`.
- `++it` moves the iterator `it` from `a[i]` to `a[i+1]`.
- `a.end()` corresponds to the iterator pointing to a position directly after the last value of the vector.



Iterator - Example

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

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

- a.begin() corresponds to the iterator pointing to a[0].
- ++it moves the iterator it from a[i] to a[i+1].
- a.end() corresponds to the iterator pointing to a position directly after the last value of the vector.

Iterator - Example

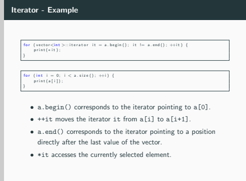
```
for (vector<int >::iterator it = a.begin(); it != a.end(); ++it) {
    print(*it);
}
```

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

- `a.begin()` corresponds to the iterator pointing to `a[0]`.
- `++it` moves the iterator `it` from `a[i]` to `a[i+1]`.
- `a.end()` corresponds to the iterator pointing to a position directly after the last value of the vector.
- `*it` accesses the currently selected element.

2019-10-27

C++ Iterators and Algorithms
└Iterators
└Iterator - Example



Iterator - Example

```
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() corresponds to the iterator pointing to a[0].
- ++it moves the iterator it from a[i] to a[i+1].
- a.end() corresponds to the iterator pointing to a position directly after the last value of the vector.
- *it accesses the currently selected element.

Watch out!

If a is of type vector<vector<int> >:

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

Watch out!

If a is of type vector<vector<int> >:

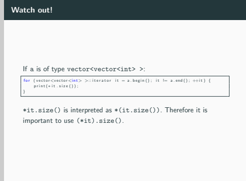
```
for (vector<vector<int> >::iterator it = a.begin(); it != a.end(); ++it) {
    print(*it.size());
}
```

Watch out!

If a is of type `vector<vector<int> >`:

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

`*it.size()` is interpreted as `*(it.size())`. Therefore it is important to use `(*it).size()`.



Watch out!

If a is of type `vector<vector<int> >`:

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

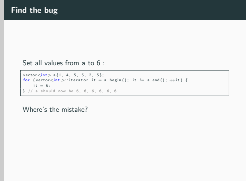
`*it.size()` is interpreted as `*(it.size())`. Therefore it is important to use `(*it).size()`.

Find the bug

Set all values from a to 6 :

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

Where's the mistake?



Find the bug

Set all values from a to 6 :

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

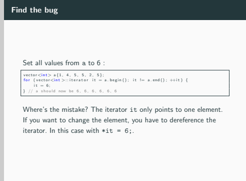
Where's the mistake?

Find the bug

Set all values from a to 6 :

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

Where's the mistake? The iterator `it` only points to one element. If you want to change the element, you have to dereference the iterator. In this case with `*it = 6;`.



Find the bug

Set all values from a to 6 :

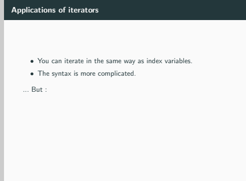
```
vector<int> a{1, 4, 5, 5, 2, 5};
for (vector<int>::iterator it = a.begin(); it != a.end(); ++it) {
    it = 6;
} // a should now be 6, 6, 6, 6, 6, 6
```

Where's the mistake? The iterator `it` only points to one element. If you want to change the element, you have to dereference the iterator. In this case with `*it = 6;`.

Applications of iterators

- You can iterate in the same way as index variables.
- The syntax is more complicated.

... But :



Applications of iterators

- You can iterate in the same way as index variables.
- The syntax is more complicated.

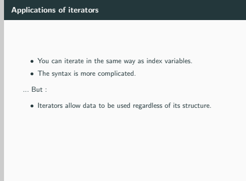
... But :

Applications of iterators

- You can iterate in the same way as index variables.
- The syntax is more complicated.

... But :

- Iterators allow data to be used regardless of its structure.



Applications of iterators

- You can iterate in the same way as index variables.
- The syntax is more complicated.

... But :

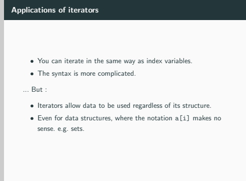
- Iterators allow data to be used regardless of its structure.

Applications of iterators

- You can iterate in the same way as index variables.
- The syntax is more complicated.

... But :

- Iterators allow data to be used regardless of its structure.
- Even for data structures, where the notation `a[i]` makes no sense. e.g. sets.



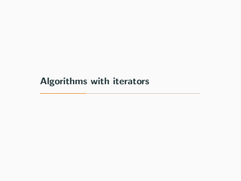
Applications of iterators

- You can iterate in the same way as index variables.
- The syntax is more complicated.

... But :

- Iterators allow data to be used regardless of its structure.
- Even for data structures, where the notation `a[i]` makes no sense. e.g. sets.

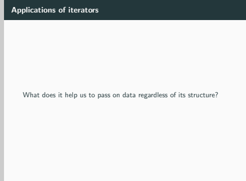
Algorithms with iterators



Algorithms with iterators

Applications of iterators

What does it help us to pass on data regardless of its structure?



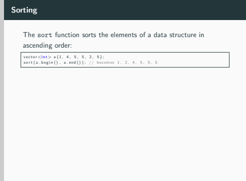
Applications of iterators

What does it help us to pass on data regardless of its structure?

Sorting

The `sort` function sorts the elements of a data structure in ascending order:

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



Sorting

The sort function sorts the elements of a data structure in ascending order:

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

Sorting

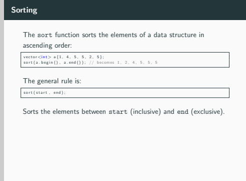
The `sort` function sorts the elements of a data structure in ascending order:

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

The general rule is:

```
sort(start, end);
```

Sorts the elements between `start` (inclusive) and `end` (exclusive).



Sorting

The `sort` function sorts the elements of a data structure in ascending order:

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

The general rule is:

```
sort(start, end);
```

Sorts the elements between `start` (inclusive) and `end` (exclusive).

Sorting

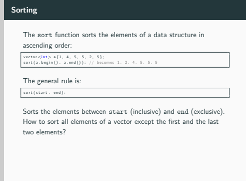
The sort function sorts the elements of a data structure in ascending order:

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

The general rule is:

```
sort(start, end);
```

Sorts the elements between start (inclusive) and end (exclusive). How to sort all elements of a vector except the first and the last two elements?



Sorting

The sort function sorts the elements of a data structure in ascending order:

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

The general rule is:

```
sort(start, end);
```

Sorts the elements between start (inclusive) and end (exclusive). How to sort all elements of a vector except the first and the last two elements?

Sorting

The `sort` function sorts the elements of a data structure in ascending order:

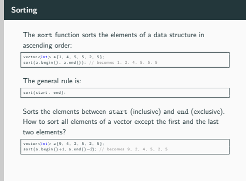
```
vector<int> a{1, 4, 5, 5, 2, 5};
sort(a.begin(), a.end()); // becomes 1, 2, 4, 5, 5, 5
```

The general rule is:

```
sort(start, end);
```

Sorts the elements between `start` (inclusive) and `end` (exclusive). How to sort all elements of a vector except the first and the last two elements?

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



Sorting

The `sort` function sorts the elements of a data structure in ascending order:

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

The general rule is:

sort(start, end);

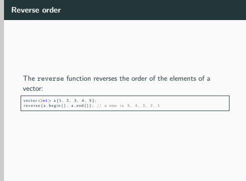
Sorts the elements between `start` (inclusive) and `end` (exclusive). How to sort all elements of a vector except the first and the last two elements?

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

Reverse order

The `reverse` function reverses the order of the elements of a vector:

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



Reverse order

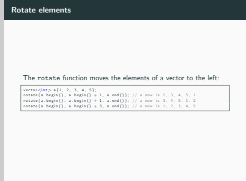
The `reverse` function reverses the order of the elements of a vector:

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

Rotate elements

The `rotate` function moves the elements of a vector to the left:

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



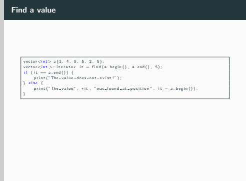
Rotate elements

The rotate function moves the elements of a vector to the left:

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

Find a value

```
vector<int> a{1, 4, 5, 5, 2, 5};
vector<int>::iterator it = find(a.begin(), a.end(), 5);
if (it == a.end()) {
    print("The␣value␣does␣not␣exist!");
} else {
    print("The␣value", *it, "was␣found␣at␣position", it − a.begin());
}
```



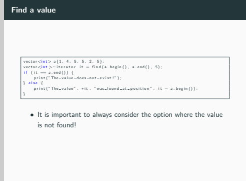
Find a value

```
vector<int> a{1, 4, 5, 5, 2, 5};
vector<int>::iterator it = find(a.begin(), a.end(), 5);
if (it == a.end()) {
    print("The␣value␣does␣not␣exist!");
} else {
    print("The␣value", *it, "was␣found␣at␣position", it − a.begin());
}
```

Find a value

```
vector<int> a{1, 4, 5, 5, 2, 5};
vector<int >::iterator it = find(a.begin(), a.end(), 5);
if (it == a.end()) {
    print("The␣value␣does␣not␣exist!");
} else {
    print("The␣value", *it, "was␣found␣at␣position", it - a.begin());
}
```

- It is important to always consider the option where the value is not found!



Find a value

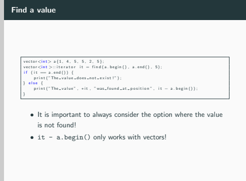
```
vector<int> a{1, 4, 5, 5, 2, 5};
vector<int >::iterator it = find(a.begin(), a.end(), 5);
if (it == a.end()) {
    print("The␣value␣does␣not␣exist!");
} else {
    print("The␣value", *it, "was␣found␣at␣position", it - a.begin());
}
```

- It is important to always consider the option where the value is not found!

Find a value

```
vector<int> a{1, 4, 5, 5, 2, 5};
vector<int>::iterator it = find(a.begin(), a.end(), 5);
if (it == a.end()) {
    print("The␣value␣does␣not␣exist!");
} else {
    print("The␣value", *it, "was␣found␣at␣position", it - a.begin());
}
```

- It is important to always consider the option where the value is not found!
- `it - a.begin()` only works with vectors!



Find a value

```
vector<int> a{1, 4, 5, 5, 2, 5};
vector<int>::iterator it = find(a.begin(), a.end(), 5);
if (it == a.end()) {
    print("The␣value␣does␣not␣exist!");
} else {
    print("The␣value", *it, "was␣found␣at␣position", it - a.begin());
}
```

- It is important to always consider the option where the value is not found!
- `it - a.begin()` only works with vectors!

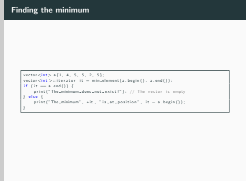
Finding the minimum

```
vector<int> a{1, 4, 5, 5, 2, 5};
vector<int >::iterator it = min_element(a.begin(), a.end());
if (it == a.end()) {
    print("The␣minimum␣does␣not␣exist!"); // The vector is empty
} else {
    print("The␣minimum", *it, "is␣at␣position", it - a.begin());
}
```

2019-10-27

C++ Iterators and Algorithms
└Algorithms with iterators

└Finding the minimum



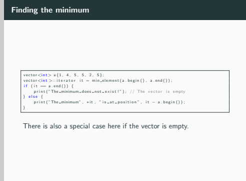
Finding the minimum

```
vector<int> a{1, 4, 5, 5, 2, 5};
vector<int >::iterator it = min_element(a.begin(), a.end());
if (it == a.end()) {
    print("The␣minimum␣does␣not␣exist!"); // The vector is empty
} else {
    print("The␣minimum", *it, "is␣at␣position", it - a.begin());
}
```

Finding the minimum

```
vector<int> a{1, 4, 5, 5, 2, 5};
vector<int >::iterator it = min_element(a.begin(), a.end());
if (it == a.end()) {
    print("The␣minimum␣does␣not␣exist!"); // The vector is empty
} else {
    print("The␣minimum", *it, "is␣at␣position", it - a.begin());
}
```

There is also a special case here if the vector is empty.



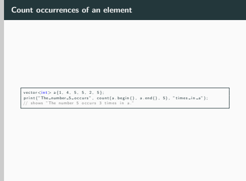
Finding the minimum

```
vector<int> a{1, 4, 5, 5, 2, 5};
vector<int >::iterator it = min_element(a.begin(), a.end());
if (it == a.end()) {
    print("The␣minimum␣does␣not␣exist!"); // The vector is empty
} else {
    print("The␣minimum", *it, "is␣at␣position", it - a.begin());
}
```

There is also a special case here if the vector is empty.

Count occurrences of an element

```
vector<int> a{1, 4, 5, 5, 2, 5};
print("The␣number␣5␣occurs", count(a.begin(), a.end(), 5), "times␣in␣a");
// shows "The number 5 occurs 3 times in a."
```

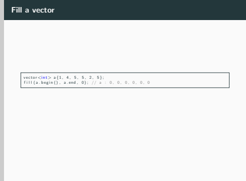


Count occurrences of an element

vector<int> a{1, 4, 5, 5, 2, 5};
print("The␣number␣5␣occurs", count(a.begin(), a.end(), 5), "times␣in␣a");
// shows "The number 5 occurs 3 times in a."

Fill a vector

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



Fill a vector

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

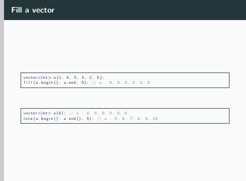
Fill a vector

```
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
```

2019-10-27

C++ Iterators and Algorithms
└─Algorithms with iterators
└─Fill a vector



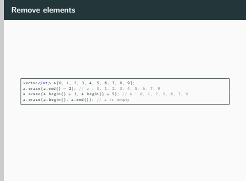
Fill a vector

```
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
```

Remove elements

```
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 is empty
```



Remove elements

```
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 is empty
```

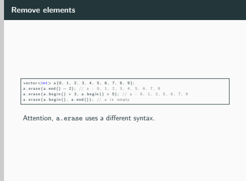
Remove elements

```
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 is empty
```

Attention, `a.erase` uses a different syntax.

2019-10-27

C++ Iterators and Algorithms
└Algorithms with iterators
└Remove elements


Remove elements

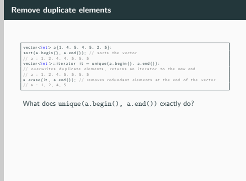
```
vector<int> a{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
a.erase(a.end() - 2); // a : 0, 1, 2, 3, 4, 5, 6, 7, 9
a.erase(a.begin() + 3, a.begin() + 5); // a : 0, 1, 2, 5, 6, 7, 9
a.erase(a.begin(), a.end()); // a is empty
```

Attention, a.erase uses a different syntax.

Remove duplicate elements

```
vector<int> a{1, 4, 5, 4, 5, 2, 5};
sort(a.begin(), a.end()); // sorts the vector
// a : 1, 2, 4, 4, 5, 5, 5
vector<int >::iterator it = unique(a.begin(), a.end());
// overwrites duplicate elements, returns an iterator to the new end
// a : 1, 2, 4, 5, 5, 5, 5
a.erase(it , a.end()); // removes redundant elements at the end of the vector
// a : 1, 2, 4, 5
```

What does `unique(a.begin(), a.end())` exactly do?



Remove duplicate elements

```
vector<int> a{1, 4, 5, 4, 5, 2, 5};
sort(a.begin(), a.end()); // sorts the vector
// a : 1, 2, 4, 4, 5, 5, 5
vector<int >::iterator it = unique(a.begin(), a.end());
// overwrites duplicate elements, returns an iterator to the new end
// a : 1, 2, 4, 5, 5, 5, 5
a.erase(it , a.end()); // removes redundant elements at the end of the vector
// a : 1, 2, 4, 5
```

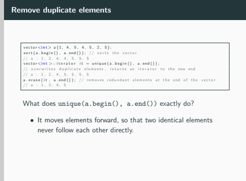
What does unique(a.begin(), a.end()) exactly do?

Remove duplicate elements

```
vector<int> a{1, 4, 5, 4, 5, 2, 5};
sort(a.begin(), a.end()); // sorts the vector
// a : 1, 2, 4, 4, 5, 5, 5
vector<int >::iterator it = unique(a.begin(), a.end());
// overwrites duplicate elements, returns an iterator to the new end
// a : 1, 2, 4, 5, 5, 5, 5
a.erase(it , a.end()); // removes redundant elements at the end of the vector
// a : 1, 2, 4, 5
```

What does `unique(a.begin(), a.end())` exactly do?

- It moves elements forward, so that two identical elements never follow each other directly.



Remove duplicate elements

```
vector<int> a{1, 4, 5, 4, 5, 2, 5};
sort(a.begin(), a.end()); // sorts the vector
// a : 1, 2, 4, 4, 5, 5, 5
vector<int >::iterator it = unique(a.begin(), a.end());
// overwrites duplicate elements, returns an iterator to the new end
// a : 1, 2, 4, 5, 5, 5, 5
a.erase(it , a.end()); // removes redundant elements at the end of the vector
// a : 1, 2, 4, 5
```

What does unique(a.begin(), a.end()) exactly do?

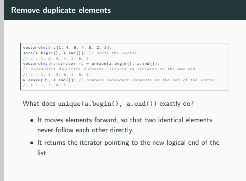
- It moves elements forward, so that two identical elements never follow each other directly.

Remove duplicate elements

```
vector<int> a{1, 4, 5, 4, 5, 2, 5};
sort(a.begin(), a.end()); // sorts the vector
// a : 1, 2, 4, 4, 5, 5, 5
vector<int >::iterator it = unique(a.begin(), a.end());
// overwrites duplicate elements, returns an iterator to the new end
// a : 1, 2, 4, 5, 5, 5, 5
a.erase(it, a.end()); // removes redundant elements at the end of the vector
// a : 1, 2, 4, 5
```

What does `unique(a.begin(), a.end())` exactly do?

- It moves elements forward, so that two identical elements never follow each other directly.
- It returns the iterator pointing to the new logical end of the list.



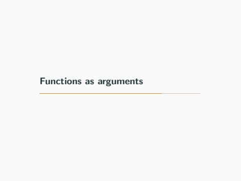
Remove duplicate elements

```
vector<int> a{1, 4, 5, 4, 5, 2, 5};
sort(a.begin(), a.end()); // sorts the vector
// a : 1, 2, 4, 4, 5, 5, 5
vector<int >::iterator it = unique(a.begin(), a.end());
// overwrites duplicate elements, returns an iterator to the new end
// a : 1, 2, 4, 5, 5, 5, 5
a.erase(it, a.end()); // removes redundant elements at the end of the vector
// a : 1, 2, 4, 5
```

What does unique(a.begin(), a.end()) exactly do?

- It moves elements forward, so that two identical elements never follow each other directly.
- It returns the iterator pointing to the new logical end of the list.

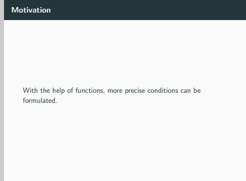
Functions as arguments



Functions as arguments

Motivation

With the help of functions, more precise conditions can be formulated.

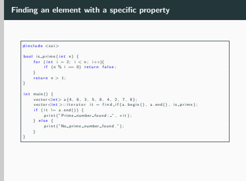


Motivation

With the help of functions, more precise conditions can be formulated.

Finding an element with a specific property

```
#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("Prime␣number␣found:␣", *it);
    } else {
        print("No␣prime␣number␣found.");
    }
}
```



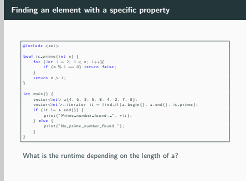
Finding an element with a specific property

```
#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("Prime␣number␣found:␣", *it);
    } else {
        print("No␣prime␣number␣found.");
    }
}
```

Finding an element with a specific property

```
#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("Prime␣number␣found:␣", *it);
    } else {
        print("No␣prime␣number␣found.");
    }
}
```

What is the runtime depending on the length of a?



Finding an element with a specific property

```
#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("Prime␣number␣found:␣", *it);
    } else {
        print("No␣prime␣number␣found.");
    }
}
```

What is the runtime depending on the length of a?

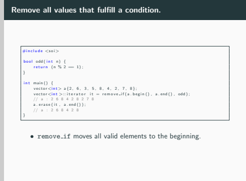
Remove all values that fulfill a condition.

```
#include <soi>

bool odd(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(), odd);
    // a : 2 6 8 4 2 8 2 7 8
    a.erase(it, a.end());
    // a : 2 6 8 4 2 8
}
```

- `remove_if` moves all valid elements to the beginning.

2019-10-27

└Remove all values that fulfill a condition.


Remove all values that fulfill a condition.

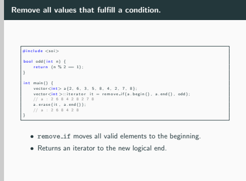
```
#include <soi>

bool odd(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(), odd);
    // a : 2 6 8 4 2 8 2 7 8
    a.erase(it, a.end());
    // a : 2 6 8 4 2 8
}
```

- remove_if moves all valid elements to the beginning.

Remove all values that fulfill a condition.

```
#include <soi>

bool odd(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(), odd);
    // a : 2 6 8 4 2 8 2 7 8
    a.erase(it, a.end());
    // a : 2 6 8 4 2 8
}
```

- `remove_if` moves all valid elements to the beginning.
- Returns an iterator to the new logical end.



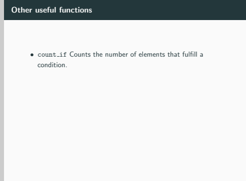
Remove all values that fulfill a condition.

```
#include <soi>

bool odd(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(), odd);
    // a : 2 6 8 4 2 8 2 7 8
    a.erase(it, a.end());
    // a : 2 6 8 4 2 8
}
```

- `remove_if` moves all valid elements to the beginning.
- Returns an iterator to the new logical end.

Other useful functions

- `count_if` Counts the number of elements that fulfill a condition.

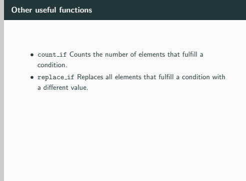


Other useful functions

- `count_if` Counts the number of elements that fulfill a condition.

Other useful functions

- `count_if` Counts the number of elements that fulfill a condition.
- `replace_if` Replaces all elements that fulfill a condition with a different value.

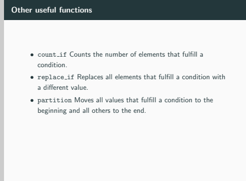


Other useful functions

- count_if Counts the number of elements that fulfill a condition.
- replace_if Replaces all elements that fulfill a condition with a different value.

Other useful functions

- `count_if` Counts the number of elements that fulfill a condition.
- `replace_if` Replaces all elements that fulfill a condition with a different value.
- `partition` Moves all values that fulfill a condition to the beginning and all others to the end.

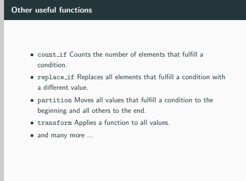


Other useful functions

- `count_if` Counts the number of elements that fulfill a condition.
- `replace_if` Replaces all elements that fulfill a condition with a different value.
- `partition` Moves all values that fulfill a condition to the beginning and all others to the end.

Other useful functions

- `count_if` Counts the number of elements that fulfill a condition.
- `replace_if` Replaces all elements that fulfill a condition with a different value.
- `partition` Moves all values that fulfill a condition to the beginning and all others to the end.
- `transform` Applies a function to all values.
- and many more ...

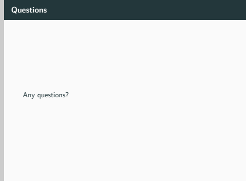


Other useful functions

- `count_if` Counts the number of elements that fulfill a condition.
- `replace_if` Replaces all elements that fulfill a condition with a different value.
- `partition` Moves all values that fulfill a condition to the beginning and all others to the end.
- `transform` Applies a function to all values.
- and many more ...

Questions

Any questions?



Questions

Any questions?