

EMS31 Kwartaal 4 Week 5: Datastructuren

Leerdoelen kwartaal 4 week 5. Je leert hoe je:

- een aantal **datastructuren** uit de standaard C++ library kunt gebruiken;
- een aantal **algoritmen** uit de standaard C++ library kunt gebruiken.

Standaard C++ library is zeer uitgebreid:

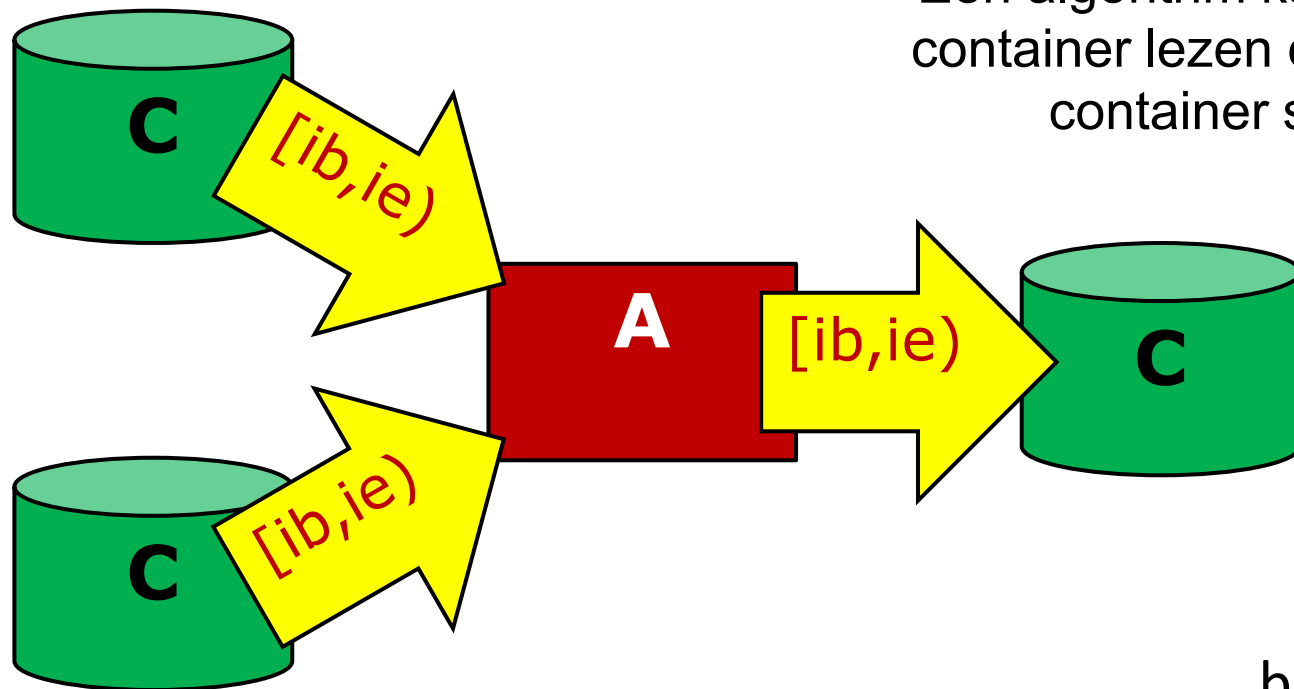
- **dictaat** hoofdstuk **10 t/m 13** en 20.
- <https://en.cppreference.com/w/cpp>

Wij beperken ons tot:

- **Containers** (datastructuren)
- **Iterators** (om door een container te 'lopen')
- **Algorithms** (generieke algoritmes)

- Het gebruik van iterators maakt **generiek** programmeren (generic programming) mogelijk. Een generiek algoritme kan op **verschillende** datatypes (containers) toegepast worden.
- Met behulp van iterators kun je een algoritme op de objecten in een container uitvoeren.
- De containers en algoritmen zijn zo **efficiënt** mogelijk (geen inheritance en virtuele functies gebruikt).

Containers, Iterators and Algorithms



Een algorithm kan ook uit **één** container lezen en in **dezelfde** container schrijven

C = container
A = algorithm
i = iterator

b = begin, e = end

[ib, ie) = range van ib **tot** ie

Iterator voorbeeld

Met behulp van een **iterator** kun je door een string **'lopen'** op dezelfde manier als dat je met een pointer door een `char[]` kunt 'lopen'.

```
char naam1[] {"Roy"};
for (const char* p {naam1}; *p != '\0'; ++p) {
    cout << *p << ' ';
}
cout << '\n';
string naam2 {"Harry"};
for (string::const_iterator i {naam2.cbegin()}; i != naam2.cend(); ++i) {
    cout << *i << ' ';
}
cout << '\n';
```

`cbegin()` geeft een `const_iterator` terug

Iterator voorbeeld met auto

Met behulp van een **iterator** kun je door een string **'lopen'** op dezelfde manier als dat je met een pointer door een `char[]` kunt 'lopen'.

```
char naam1[] {"Roy"};
for (const auto* p {naam1}; *p != '\0'; ++p) {
    cout << *p << ' ';
}
cout << '\n';
string naam2 {"Harry"};
for (auto i {naam2.cbegin()}; i != naam2.cend(); ++i) {
    cout << *i << ' ';
}
cout << '\n';
```

- Sequentiële containers
 - []
 - string
 - array
 - vector
 - `forward_list` (singly linked list)
 - `list` (doubly linked list)
 - `deque` (double ended queue, eigenlijk een devector)
 - `bitset` (slaan wij over)
 - adapters (in combinatie met `vector`, `list` of `deque`):
 - `stack`
 - `queue`
 - `priority_queue`

- Associatieve containers (elementen op volgorde van sleutelwaarde opgeslagen)

- **set**

- Verzameling van sleutels (keys).
- Sleutel moet origineel zijn.

- **multiset**

- Verzameling van sleutels (keys).
- Sleutel kan meerdere malen voorkomen.

- **map**

- Verzameling van paren (sleutel, waarde) pair<const key, value>.
- Sleutel moet origineel zijn.

- **multimap**

- Verzameling van paren (sleutel, waarde) pair<const key, value>.
- Sleutel kan meerdere malen voorkomen.

Geïmplementeerd met
een *binary search tree*

- Associatieve containers (elementen in willekeurige volgorde opgeslagen)

- `unordered_set`

- Verzameling van sleutels (keys).
- Sleutel moet origineel zijn.

- `unordered_multiset`

- Verzameling van sleutels (keys).
- Sleutel kan meerdere malen voorkomen.

- `unordered_map`

- Verzameling van paren (sleutel, waarde) `pair<const key, value>`.
- Sleutel moet origineel zijn.

- `unordered_multimap`

- Verzameling van paren (sleutel, waarde) `pair<const key, value>`.
- Sleutel kan meerdere malen voorkomen.

Geïmplementeerd met
een *hash table*

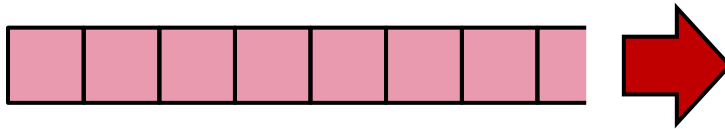
- De containers regelen hun eigen **geheugenbeheer**.
- De containers maken **kopietjes** van de elementen die erin gestopt worden. Als je dat niet wilt dan moet je een container met pointers gebruiken.
- Alle objecten in een container zijn van **hetzelfde type**. (Dat kan wel een *polymorf* pointertype zijn!)

Sequentiële containers

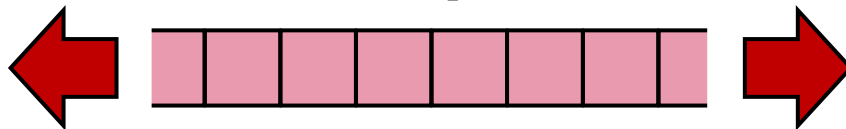
array



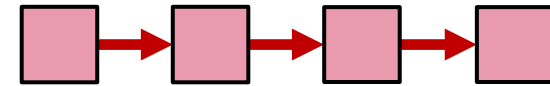
vector



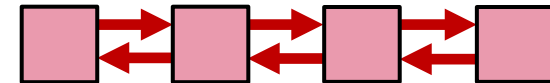
deque



forward_list



list



Container adapters passen de interface van een vector, deque of list aan:

- `stack`
- `queue`
- `priority_queue`

```
stack<vector<int>> s1; // stack implemented with vector
stack<deque<int>> s2; // stack implemented with deque
stack<list<int>> s3;  // stack implemented with list
stack<int> s4;      // using deque by default
```

Zie dictaat H9 voor een **alternatieve implementatie** en H8 voor (verrassende) **toepassingen** van een stack

Associatieve containers

- `set` (verzameling)
 - `multiset` (bag)
 - `map` (1:N relatie)
 - `multimap` (M:N relatie)
- } Bevat elementen van het type:
`const Key`
- } Bevat elementen van het type:
`pair<const Key, Value>`

Mogelijkheden:

- `zoeken` op `Key`
- doorlopen op `volgorde` van `Key`

Implementatie: *binary search tree*

Associatieve containers

- `unordered_set`
- `unordered_multiset`
- `unordered_map`
- `unordered_multimap`

Bevat elementen van het type:
`const Key`

Bevat elementen van het type:
`pair<const Key, Value>`

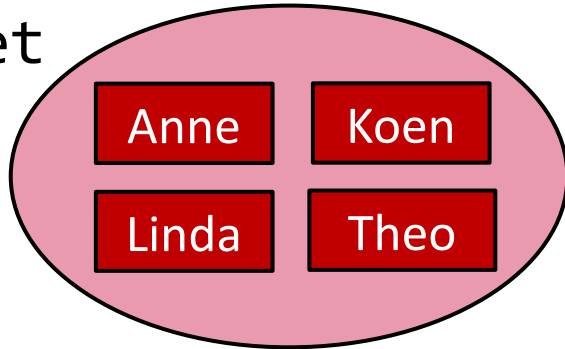
Mogelijkheden:

- **zoeken** op Key

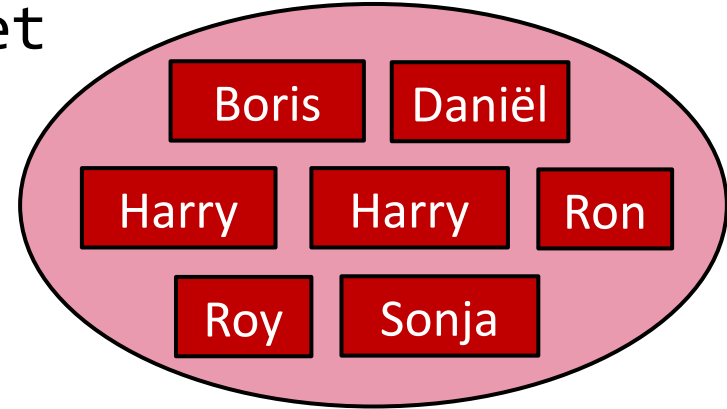
Implementatie: *hash table*

Associatieve containers

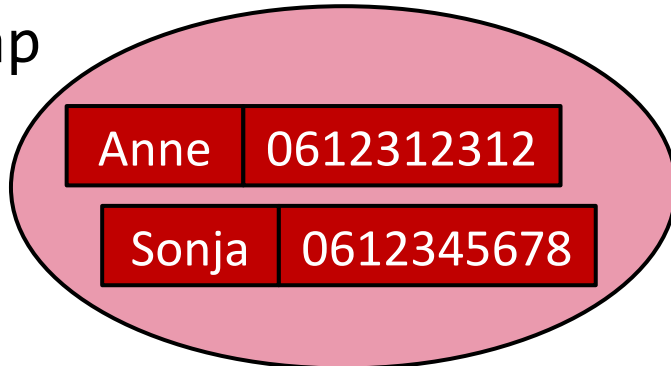
set



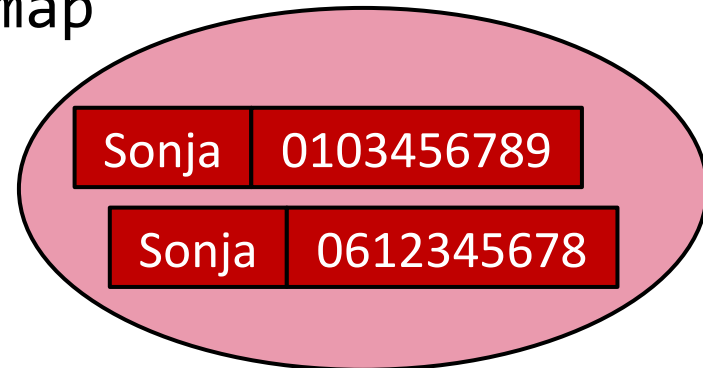
multiset



map



multimap



set (verzameling)

- Toevoegen met **insert**
 - Een element wordt automatisch op de 'goede' plaats ingevoegd.
 - Returntype is een `pair<iterator, bool>`. De `bool` geeft aan of het invoegen gelukt is en de `iterator` geeft de plek aan waar ingevoegd is.
- Verwijderen met **erase**
 - Een te verwijderen element wordt automatisch opgezocht.
 - Je kunt ook een iterator meegeven of een range (2 iterators).
- Zoeken met **find**
 - Geeft een iterator terug naar de plaats waar het element staat en geeft `end()` terug als element niet gevonden is.
- Zoeken met **count**
 - Geeft het aantal keer dat een element voorkomt (0 of 1).

multiset (bag)

- Toevoegen met **insert**
 - Een element wordt automatisch op de 'goede' plaats ingevoegd.
 - Returntype is een `iterator`. De `iterator` geeft de plek aan waar ingevoegd is.
- Verwijderen met **erase**
 - Een te verwijderen element wordt automatisch opgezocht. Alle gevonden elementen worden verwijderd.
 - Je kunt ook een `iterator` meegeven of een `range` (2 `iteratoren`).
- Zoeken met **find**
 - Geeft een `iterator` terug naar de plaats waar het eerste element staat en geeft `end()` terug als element niet gevonden is.
- Zoeken met **count**
 - Geeft het aantal keer dat een element voorkomt (≥ 0).

map (1:N relatie)

- Elementen zijn: `pair<const Key, Value>`
- Interface gelijk aan set:
 - `insert`
 - `erase`
 - `find`
 - `count`
- Extra `operator[]`
 - Met `operator[]` kun je een `key als index` gebruiken.
 - Als de key al in de map zit wordt een reference naar de bijbehorende value teruggegeven.
 - Als de key niet aanwezig in de map dan wordt deze key toegevoegd met de default value (default constructor).

multimap (1:N relatie)

- Elementen zijn: `pair<const Key, Value>`
- Interface gelijk aan `multiset`:
 - `insert`
 - `erase`
 - `find`
 - `count`
- **Geen** `operator[]`

Voorbeeld met map

```
#include <iostream>
#include <fstream>
#include <string>
#include <map>
using namespace std;
int main() {
    string w;
    map<string, int> freq;
    cout << "Geef filenaam: ";
    cin >> w;
    ifstream fin {w};
    while (fin >> w) {
        ++freq[w];
    }
    for (const auto& wc: freq) {
        cout << wc.first << " " << wc.second << '\n';
    }
}
```

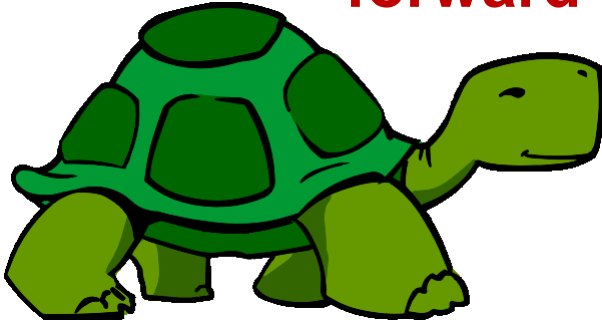
Iteratoren (er zijn er meer)

Een **iterator** is geen class maar een (interface) **afsprakenk**. Elke class die aan deze afspraak voldoet is als iterator te gebruiken.

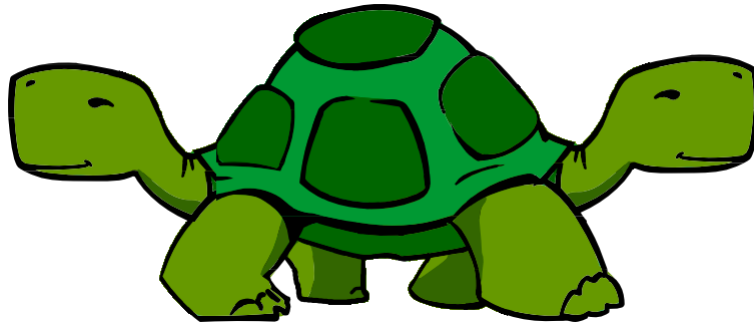
- **forward** iterator
 - voorwaarts * ++ == !=
- **bidirectional** iterator
 - voor-/achterwaarts * ++ -- == !=
- **random access** iterator
 - met sprongen * ++ -- += -= + - == != > >= < <= []
 - een gewone pointer is een random access iterator

Iteratoren (er zijn er meer)

forward iterator



Bron: <https://pixabay.com/nl/vectors/schildpad-carapax-groene-wandelen-303732/>



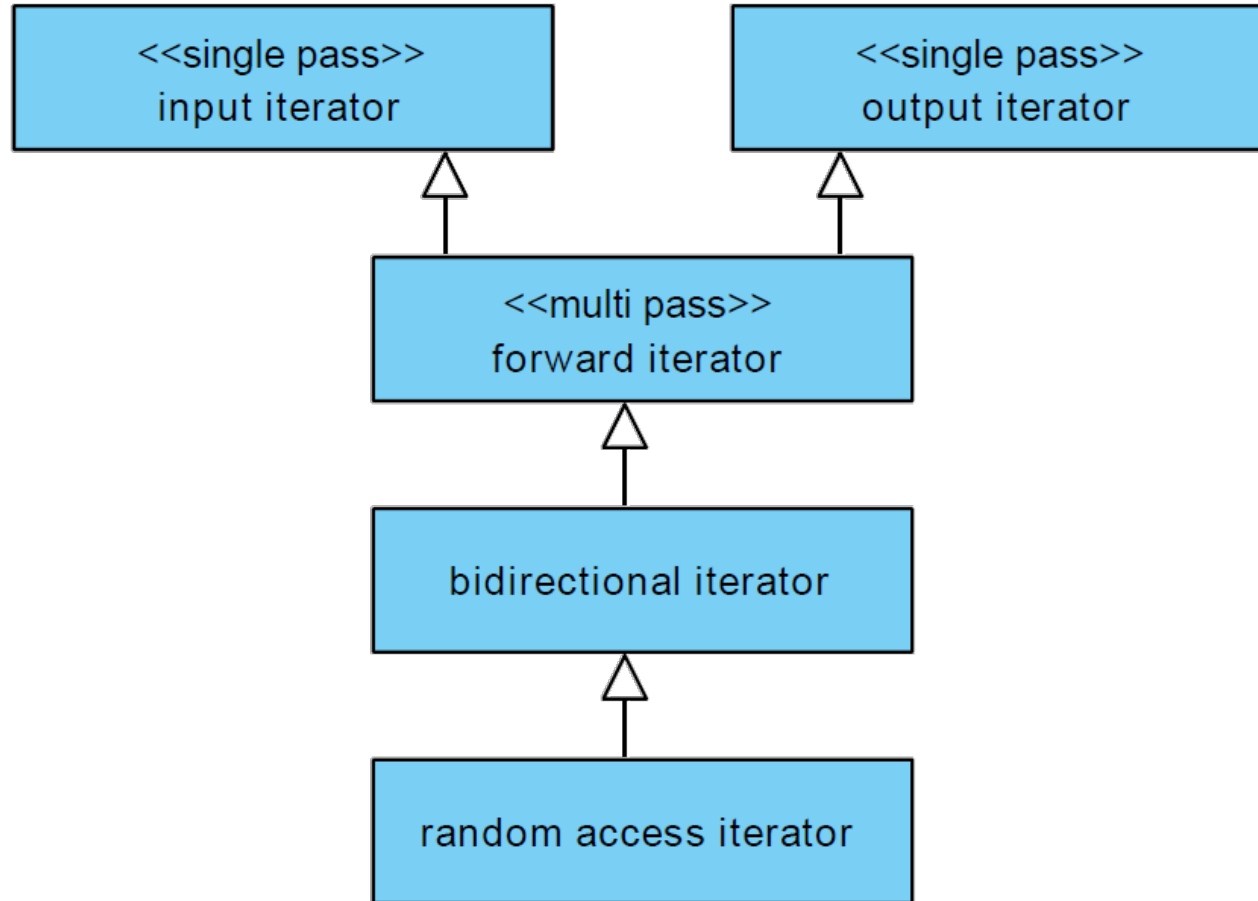
bidirectional iterator



Bron: https://www.pngkey.com/detail/u2q8w7r5o0i1e6u2_disneys-roo-clip-art-winnie-the-pooh-roo/

random access iterator

Iteratoren



Relatie met containers

- **forward iterator**
 - `forward_list`, `unordered_set`, `unordered_multiset`, `unordered_map` en `unordered_multimap`
- **bidirectional iterator**
 - `list`, `set`, `multiset`, `map` en `multimap`
- **random access iterator**
 - `vector` en `deque`

Relatie met algorithms

- **input iterator** bijvoorbeeld `find`
- **output iterator** bijvoorbeeld `copy`
- **forward iterator** bijvoorbeeld `replace`
- **bidirectional iterator** bijvoorbeeld `reverse`
- **random access iterator** bijvoorbeeld `sort`

Reverse-iteratoren (van achter naar voor).

- `rbegin()`, `crbegin()`, `rend()` en `crend()` geven een reverse-iterator

Insert-iteratoren (een soort cursor in insert i.p.v. override mode).

- `insert_iterator`
- `back_insert_iterator`
- `front_insert_iterator`

Er zijn functie beschikbaar om deze insert-iteratoren aan te maken, zie dictaat paragraaf 12.3.

Dit zijn wel echte classes!

Stream-iteratoren (koppeling tussen `iostream` library en algorithms).

- `istream_iterator`
- `ostream_iterator`

Dit zijn wel echte classes!

Voorbeeld met speciale iterators

```
#include <...>
using namespace std;

int main() {
    vector<int> rij;
    ifstream f_in {"getallen_ongesorteerd.txt"};
    if (!f_in)
        return 1; // kan f_in niet openen
    istream_iterator<int> i_in {f_in}, einde;
    copy(i_in, einde, back_inserter(rij));
    sort(rij.begin(), rij.end());
    ofstream f_out {"getallen_gesorteerd.txt"};
    if (!f_out)
        return 2; // kan fout niet openen
    ostream_iterator<int> i_out {f_out, " "};
    copy(rij.cbegin(), rij.crend(), i_out);
}
```

Een generiek algoritme werkt op een **range**. Een range wordt aangegeven door **twee iterators** $i1$ en $i2$. De range loopt van $i1$ tot (dus niet t/m) $i2$. In de wiskunde zouden we dit noteren als $[i1, i2)$.

```
int main() {  
    vector<int> v = {12, 18, 6};  
    sort(v.begin(), v.end());  
    for (auto e: v) {  
        cout << e << " ";  
    }  
}
```

Algoritmen versus memberfuncties

Een **generiek algoritme** werkt op meerdere container types. Sommige containers hebben voor bepaalde bewerkingen **specifieke memberfuncties**.

- Omdat het generieke algoritme krachtigere iterators nodig heeft dan de container kan leveren.
- Omdat de specifieke memberfunctie **sneller** is dan het generieke algoritme.

Kijk dus voordat je een generiek algoritme gebruikt altijd eerst of er een specifieke memberfunctie is.

Algoritmen versus memberfuncties

```
list<int> l {12, 18, 6};  
sort(l.begin(), l.end()); // Error!
```

heeft random access iterators nodig `list`
levert bidirectionele iterators

GCC error (flink aantal regels) belangrijkste mededeling:
In instantiation of 'void
std::__sort(_RandomAccessIterator,
_RandomAccessIterator, _Compare): error: no match for 'operator-'

```
l.sort(); // Ok!
```

Algoritmen versus memberfuncties

```
#include <...>
using namespace std;

int main() {
    set<int> s {12, 6, /*... ,*/ 18}; // Hele grote set

    auto i1 {find(s.cbegin(), s.cend(), 6)}; // Redelijk snel O(n)
    if (i1 != s.cend()) {
        cout << *i1 << " gevonden\n";
    }

    auto i2 {s.find(6)}; // Sneller O(log n)
    if (i2 != s.end()) {
        cout << *i2 << " gevonden\n";
    }
}
```

Er zijn heel veel algoritmen beschikbaar in de standaard C++ library.

- Zie dictaat hoofdstuk 13
- Zie <https://en.cppreference.com/w/cpp/algorithm>

find

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main() {
    list<int> l {-3, -4, 3, 4};
    auto r {find(l.cbegin(), l.cend(), 3)};
    if (r != l.cend()) {
        cout << "Element " << *r << " is gevonden\n";
    }
}
```

find_if

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
```

```
bool is_pos(int i) {
    return i >= 0;
}
```

```
int main() {
    list<int> l {-3, -4, 3, 4};
    auto r {find_if(l.cbegin(), l.cend(), is_pos)};
    if (r != l.cend()) {
        cout << "Het eerste positieve element is: " << *r << '\n';
    }
}
```

Functie **is_pos** wordt gebruikt
als predicate

Wat als we de eerste waarde
>=1 in de lijst l willen zoeken?

- Een **lambda-functie** is een anoniem functie-object (functor).
- De lambda-functie wordt gedefinieerd op de plaats waar het functie-object nodig is.
- Een lambda-functie kan dus als predicate gebruikt worden.



find_if

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
```

```
int main() {
    list<int> l {-3, -4, 3, 4};
    auto r {find_if(l.cbegin(), l.cend(), [](int i) {
        return i >= 0;
    })};
    if (r != l.cend()) {
        cout << "Het eerste positieve element is: " << *r << '\n';
    }
}
```

Lambda-functie

```
[](int i) { return i >= 0; }
wordt gebruikt als predicate
```

find_if

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
```

```
int main() {
    list<int> l {-3, -4, 3, 4};
    auto r {find_if(l.cbegin(), l.cend(), [](auto i) {
        return i >= 0;
    })};
    if (r != l.cend()) {
        cout << "Het eerste positieve element is: " << *r << '\n';
    }
}
```

Lambda-functie

```
[](auto i) { return i >= 0; }
wordt gebruikt als predicate
```

- Een lambda-functie kan gebruik maken van variabelen die buiten de lambda-functie gedeclareerd zijn. Zo'n lambda-functie-object wordt een **closure** genoemd. Nederlands: (in)sluiting. De variabelen die ingesloten moeten worden in de closure worden tussen [en] opgegeven.
- [a, &b] a wordt 'by value' en b wordt 'by reference' ingesloten.
- [&] elke gebruikte variabele wordt 'by reference' ingesloten.
- [=] elke gebruikte variabele wordt 'by value' ingesloten.
- [=, &c] c wordt 'by reference' ingesloten en alle andere gebruikte variabelen worden 'by value' ingesloten.
- [] er worden géén variabelen ingesloten.



Voorbeeld closure

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    vector<int> v {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int som_even {0};
    for_each(v.cbegin(), v.cend(), [&som_even](auto i) {
        if (i % 2 == 0) {
            som_even += i;
        }
    });
    cout << "De som van even getallen in v = " << som_even << '\n';
}
```

[&] werkt ook.
Wat is beter?

Algoritmen, performance en big-O-notatie



Bron: <https://media-cdn.tripadvisor.com/media/photo-p/1b/34/dc/f3/big-o-logo.jpg>

Aan de slag!

Aan de slag met [Opdrachten_Week_4.5.pdf](#)

