

HOOFDSTUK 3

COLLECTIONS

Helga Naessens

Inhoud

- **Inleiding collections**
- Sequences
- Container adapters
- Associatieve containers

Collections in C++: basisprincipe

- Je hebt een **extra bibliotheek** nodig.

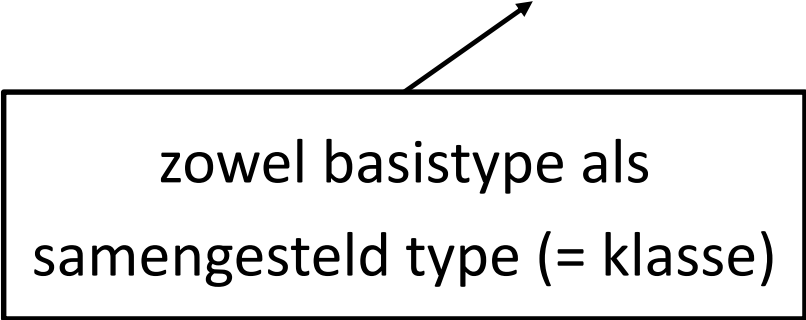
voorbeeld:

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

- Alle elementen in een collection zijn van **hetzelfde type**.

voorbeeld:

```
set<string> s;
```



zowel basistype als
samengesteld type (= klasse)

- De collection is eigenlijk een **klasse** waarvan je objecten aanmaakt.
- Elke collection heeft specifieke **methodes** en operatoren om het gebruik te vereenvoudigen. (zie API)
- De collection weet zelf **hoeveel** elementen hij heeft.

voorbeeld:

```
cout << s.size();
```

- Elke collection heeft **voor- en nadelen**, en is ontworpen met specifieke kenmerken.

Iteratoren

- Bij een aantal collections kan gebruik gemaakt worden van een iterator om alle elementen van de collection te overlopen.
- Een iterator is een pointer naar 1 element van de collection.
- Declaratie van een iterator:

collectiontype::[const_]iterator naam_var;

voorbeeld:

```
list<string> l;
```

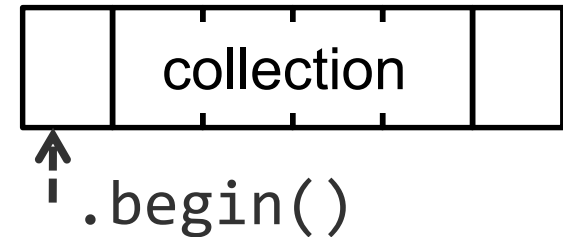
```
list<string>::iterator it;
```

```
typename set<T>::const_iterator cis;
```

- Om de iterator naar het eerste element van de collection te laten wijzen, gebruik je de functie `.begin()`

voorbeeld:

```
it = l.begin();
```



- De iterator verplaatsen kan op twee manieren:

`++it` of `it++` : springt naar het volgende element

`--it` of `it--` : springt naar het vorige element

- Gebruik `*` om het element waar de iterator naar wijst te bekomen (of `->` indien samengesteld type, waarvan men een onderdeel wil selecteren)

voorbeeld:

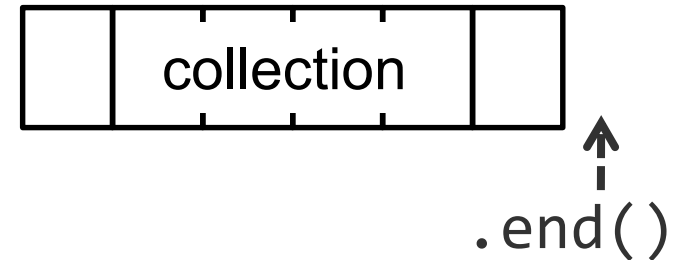
```
cout << *it << endl; *it = "stop";
```

- Om een iterator te bekomen die *voorbij* het laatste element van de collection wijst, gebruik je de functie **.end()**

voorbeeld:

```
list<string>::iterator it2;
```

```
it2 = l.end();
```



- Samengevat: om de volledige collection uit te schrijven:

```
list<string> l;
```

```
list<string>::iterator it = l.begin();
```

```
while (it != l.end()) { //niet: it < l.end()
```

```
    cout << *it << endl;
    it++;
    cout << *it++ << endl;
```

```
}
```

Inhoud

- Inleidend collections
- **Sequences**
- Container adapters
- Associatieve containers

Sequences

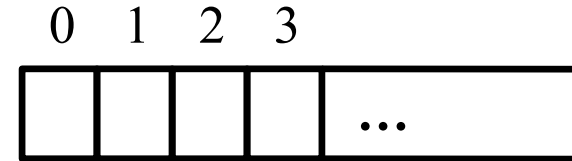
- Sequentiële containers
- Sequenties zijn eendimensionale gegevensstructuren: elk element, behalve het eerste en het laatste, heeft een linkerbuur en een rechterbuur.
- Wanneer men deze containers met een iterator overloopt, gebeurt dat in die lineaire volgorde.

Voorbeelden sequences

- **Groeitabel (string en vector)**
- Gelinkte lijst (list)
- Double-ended queue (deque)

De groeitabel: principe

- Is een **tabelgebaseerde** container die dynamisch (at runtime) kan **groeien**.
- Principe groeitabel:
 - **Bij de declaratie:**
 - wordt het type van de elementen vastgelegd
 - wordt een capaciteit (= aantal beschikbare locaties) voorzien
 - **Bij het toevoegen van een element:**
2 mogelijke scenario's (zie volgende slide)



De groeitabel: toevoegen van een element

- **Scenario 1: Er is nog plaats** in de array, want
 n (= # elementen in de array) < capaciteit
⇒ element komt op index n (en n wordt verhoogd met 1)
- **Scenario 2: Er is geen plaats meer** in de array
⇒ Er wordt een nieuwe array gemaakt met **grotere capaciteit**
⇒ Alle elementen uit de oude array worden gekopieerd naar de nieuwe array
⇒ De oude array wordt *geschrapt*
⇒ Het element komt op index n (en n wordt verhoogd met 1)

De groeitabel: uitbreiding van de array

- Het alloceren van een nieuwe array en het kopiëren van de elementen uit de oude array is een “dure” operatie.
- In welke mate wordt de array best uitgebreid?
Met 1 ~~extra geheugenplaats?~~ → **Er moet te vaak uitgebreid worden**
- In de praktijk: capaciteit bij elke re-allocatie **verdubbelen**.
⇒ naarmate de capaciteit toeneemt, worden de re-allocaties zeldzamer, maar ze worden wel “duurder”
- De gebruiker hoeft zich hier niets van aan te trekken.

Voorbeelden groeitabellen in C++

- `string`
- `vector`

Declaratie (en initialisatie) van een vector

```
#include <vector>
```

```
using namespace std;
```

```
...
```

```
vector<int> v1;
```

v1 ||

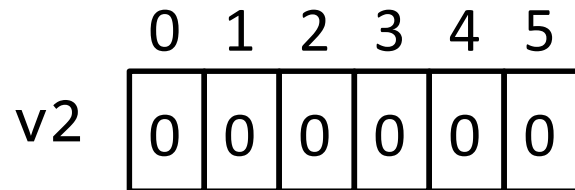


capaciteit en grootte van de array is 0

Merk op: **constructor** oproepen = **géén new** gebruiken!!!

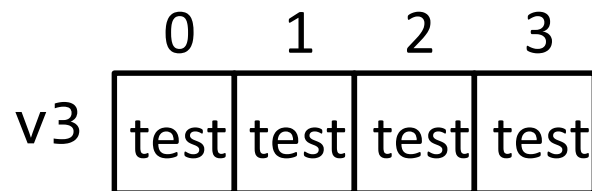
Let op: `vector<int> v2[6];`
//array met 6 vectoren

```
vector<int> v2(6);
```



capaciteit en grootte van v2 is **6**
de geheugenplaatsen zijn opgevuld
met **0** (= default waarde)

```
vector<string> v3(4, "test");
```



capaciteit en grootte van v3 is **4**
opgevuld met ***test***

```
vector<int> v4(v2); //copy-constructor
```


Aanpassen/opvragen van *gealloceerde* inhoud

```
v2[3] = -5;
```

Let op: crash als $\text{index} \geq \text{capaciteit!!}$

```
for (int i=0; i<v2.size(); i++)  
    cout << v2[i] << endl;  
//of: gebruik een for-each-lus
```

gebruik `size()` om de
grootte van de vector op
te vragen

```
cout << v2.capacity();
```

gebruik `capacity()` om de capaciteit
van de vector op te vragen

Achteraan de vector een element toevoegen

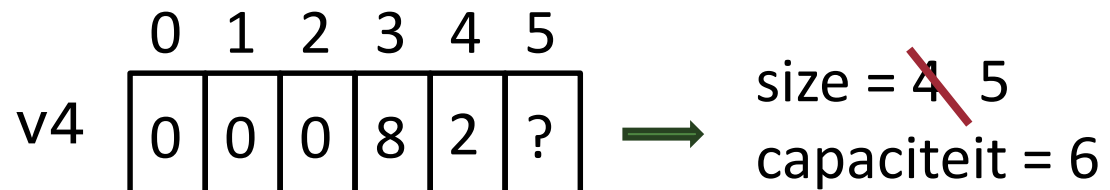
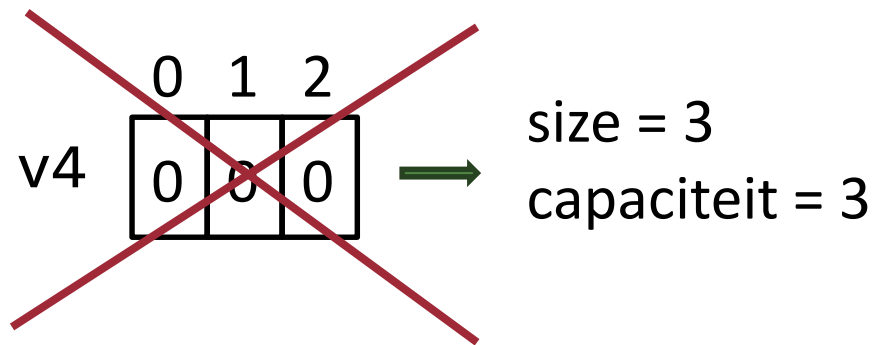
- `void push_back(type e):` voegt het element `e` achteraan toe en verhoogt de grootte (size) van de vector met 1

- Voorbeeld

```
vector<int> v4(3);
```

```
v4.push_back(8);
```

```
v4.push_back(2);
```



Enkele andere methodes voor vectoren

- *ref* **front**() : Geeft referentie naar het 1^e element vd vector terug

Vb: cout << v.front(); v.front() *= 2;

- *ref* **back**() : Geeft referentie naar laatste element vd vector terug

Vb: cout << v.back();

- *ref* **at**(int i) : Geeft referentie naar element op index i terug

Vb: cout << v.at(2)++;



exception als index $i \geq \text{size}!!$

- **bool** **empty**() : Gaat na of de vector leeg is
- **void** **clear**() : Wist de vector (size wordt 0, capaciteit ongewijzigd)

- **void pop_back()**: Verwijdert het laatste element uit de vector en vermindert de grootte (size) van de vector met 1.
Vb: `v.pop_back();`
- **void resize(int n)**: Grootte (size) van de vector wordt n (overtollige elementen worden verwijderd, bijgevoegde elementen worden geïnitieerd)
Vb: `v.resize(5);`
- **void reserve(int n)**: Capaciteit van de vector wordt *minstens* n (capaciteit wordt niet verkleind, inhoud wordt niet geïnitieerd \Rightarrow grootte (size) blijft onveranderd)

Voorbeeld: **vb_vector.cpp**

vector: toepassing 1

- Opgave: Lees een reeks gehele getallen in (stop met -1) en sla alle ingelezen getallen op in een ~~array~~/vector.
- Oplossing:

```
vector<int> v;  
  
int g;  
  
cin >> g;  
  
while (g != -1) {  
    v.push_back(g);  
    cin >> g;  
}
```

vector: toepassing 2

- Opgave: Lees een getal n in, gevolgd door n gehele getallen en sla alle ingelezen getallen op in een array/vector.

- Oplossing 1:

```
int n;  
cin >> n;  
int tabel[n];    int *tabel = new int[n];  
for (int i = 0 ; i < n ; i++) {  
    cin >> tabel[i];  
}  
delete[] tabel;
```

Fout!!! De grootte van een array moet vastliggen bij programmatie!!



- Oplossing 2 (van toepassing 2):

```
int n, g;
```

```
cin >> n;
```

```
vector<int> v;
```

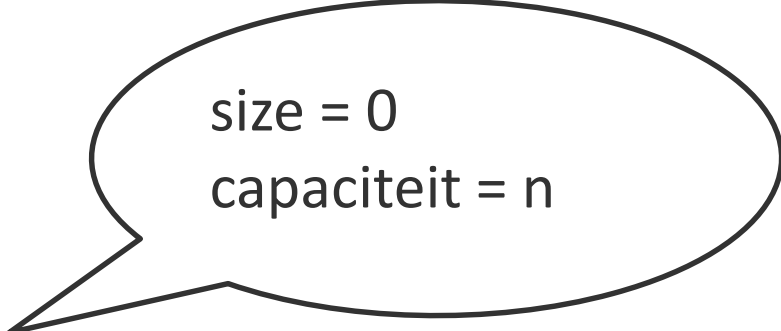
```
v.reserve(n);
```

```
for (int i = 0; i < n ; i++) {
```

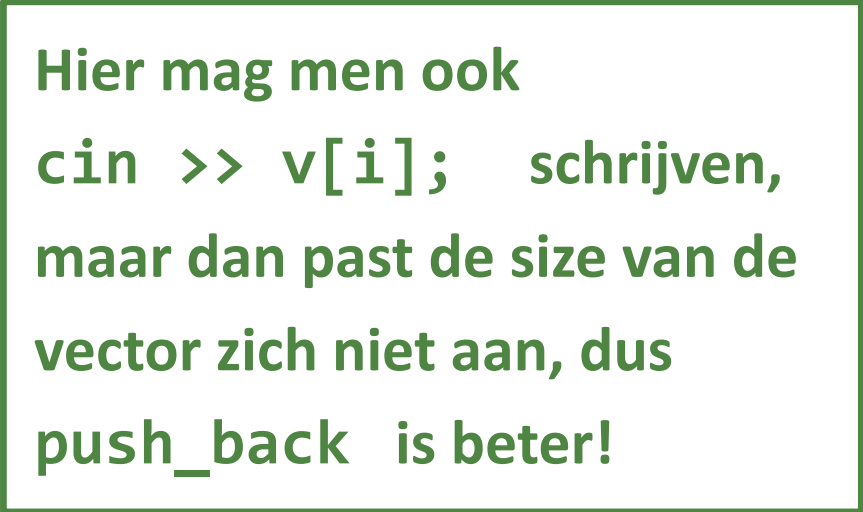
```
    cin >> g;
```

```
    v.push_back(g);
```

```
}
```



size = 0
capaciteit = n



Hier mag men ook
cin >> v[i]; schrijven,
maar dan past de size van de
vector zich niet aan, dus
push_back is beter!

- Oplossing 3 (van toepassing 2):

```
int n, g;
```

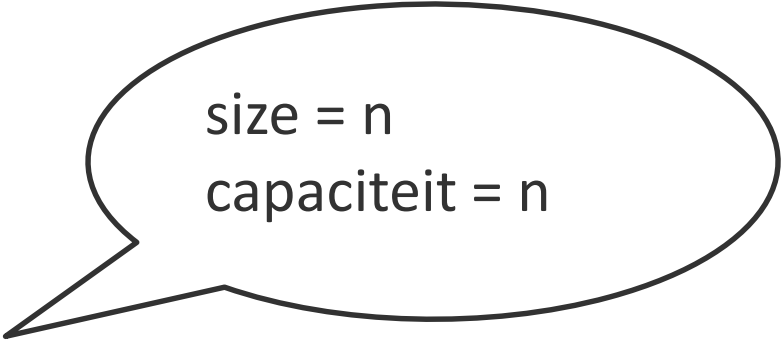
```
cin >> n;
```

```
vector<int> v(n);
```

```
for (int i = 0 ; i < n ; i++) {
```

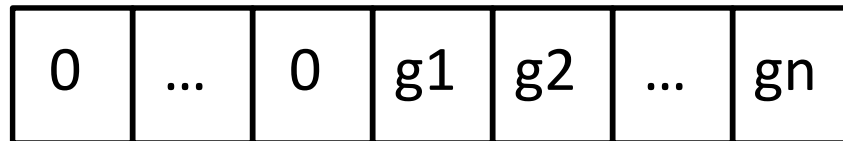
```
    cin >> v[i];
```

```
}
```



size = n
capaciteit = n

Hier gebruik je beter geen `push_back` want dan zou de vector er als volgt uitzien:



n nullen

n ingelezen getallen

vector: voordelen

- elk element in de vector is even gemakkelijk en even snel bereikbaar met []
- je hoeft bij constructie de capaciteit van de vector niet op te geven en deze capaciteit kan nadien nog groeien
- je kan snel achteraan een element toevoegen
- een vector kent haar eigen grootte en capaciteit

vector: nadelen

- er is geen index-checking
- vooraan of tussenin elementen toevoegen of verwijderen is een *dure* (moeilijke?) operatie

Voorbeelden sequences

- Groeitabel (vector)
- **Gelinkte lijst (list)**
- Double-ended queue (deque)

Principe gelinkte lijst: zie gedeelte C

list

- klasse **list** uit <list>: om met gelinkte lijsten te werken
- Enkele methodes:
 - **void push_back(type x) / void push_front(type x):**
Voegt x achteraan/vooraan toe aan de lijst
Vb: 1.push_back(5);
 - **void pop_back() / void pop_front():**
Verwijdert de laatste/ eerste knoop van de lijst
Vb: 1.pop_back();
 - **void clear():** Wist de volledige lijst
Vb: 1.clear();

- *ref* **front()**: Geeft referentie naar 1^e element van de lijst terug
- *ref* **back()**: Geeft ref. naar laatste element van de lijst terug

Vb: cout << l.front(); l.back()++;

- **bool empty()**: Gaat na of de lijst leeg is
- **int size()**: Geeft het aantal elementen van de lijst terug
- Gebruik een for-each-lus of iterators om de lijst te overlopen
⇒ gebruik géén gewone for-lus met indices
(want i^{de} element kan niet opgehaald worden)
- Gebruik **iteratoren** om elementen tussen te voegen of te verwijderen

- Methodes op lijsten die gebruik maken van iterators:
 - `iterator insert(iterator it, type x)`
Voegt knoop met x toe aan de lijst vóór iterator it;
resulterende iterator wijst naar deze nieuwe knoop
 - `iterator erase(iterator it)`
Verwijdert knoop bij iterator it; resulterende iterator
verwijst naar knoop achter verwijderde knoop

Vb: `it = l.insert(it,8); l.erase(it);`

Deze 2 methodes bestaan ook voor een vector,
maar zijn er inefficiënt, zodat hun nut beperkt is.

- Voorbeeld: **vb_list.cpp**

gelinkte lijst: voordelen

- Vooraan of tussenin een element/knoop toevoegen of verwijderen kan sneller dan bij een array.
- Achteraan een knoop toevoegen kan ook snel
(als men over een pointer beschikt naar de laatste knoop)
- Achteraan een knoop verwijderen lukt ook snel
(als men naast de opvolger ook de voorganger van de knoop kent)

gelinkte lijst: nadelen

- De i-de knoop kan men niet direct bekomen. Eerst moeten alle voorgangers overlopen worden.
⇒ indexeren ([]) is een dure operatie
- Een gelinkte lijst verbruikt meestal **iets meer geheugen** dan een groeitabel doordat elke knoop pointers moet bijhouden. Bovendien zorgt de verspreiding in het geheugen voor **performantieverlies**.

Voorbeelden sequences

- Groeitabel (vector)
- Gelinkte lijst (list)
- **Double-ended queue (deque)**

Deque

- Gedefinieerd in de header `<deque>`
- Een deque ('double-ended queue') heeft dezelfde operaties als een **vector**, met uitzondering van `capacity()` en `reserve(...)`
⇒ Indexeren (`[]` en `at`) is toegelaten
- Belangrijkste verschil met een vector:
ook vooraan (efficiënt) toevoegen en verwijderen,
zoals bij een list (`push_front()` en `pop_front()`).

Inhoud

- Inleiding collections
- Sequences
- **Container adapters**
- Associatieve containers

Container adapters

- Worden geïmplementeerd met een van de vorige containers (adapters: interface aanpassen, beperken)
- Beperken het aantal toegankelijke elementen.
 - ⇒ indexeren is onmogelijk + geen iterators
- Definiëren slechts een gering aantal operaties.

Voorbeelden container adapters

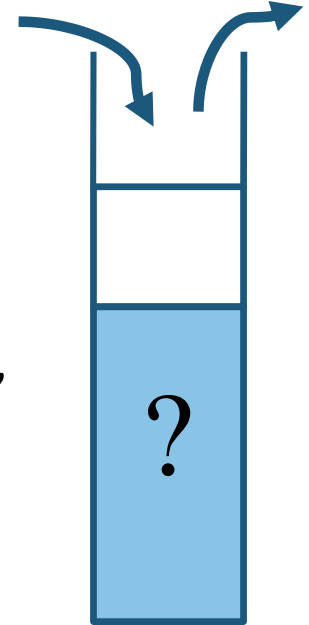
- **Stapel (stack)**
- Wachtrij (queue)
- Prioriteitswachtrij (priority_queue)

De stapel

- **LIFO**-structuur (Last In, First Out):
element dat als laatste op de stapel werd geplaatst (**push**),
wordt er als eerste terug afgehaald (**pop**).

⇒ Bij een herhaaldelijk *pop* worden de elementen
er **in omgekeerde volgorde** afgehaald.
- **Nadeel stapel**: men kan niet aan het i -de element en men kan
de elementen niet overlopen zonder ze te verwijderen.

⇒ geen sequentiële container



Stapel in C++: stack

- De C++ standard library voorziet de klasse **stack** uit `<stack>` om met stapels te werken.
- Enkele methodes:
 - **void push(*type* x):** Voegt x toe aan de stapel
Vb: `st.push("woord");`
 - **void pop():** Verwijdert de top van de stapel
Vb: `st.pop();`
 - **ref top():** Geeft een referentie naar de top van de stapel terug
Vb: `cout << st.top()++;`

– **bool** **empty()**: Gaat na of de stapel leeg is

Vb: if (st.empty())
 cout << "leeg";

– **int** **size()**: Geeft het aantal elementen van de stapel terug

Vb: int grootte = st.size();

- Voorbeeld: **vb_stack.cpp**

Voorbeelden container adapters

- Stapel (stack)
- **Wachtrij (queue)**
- Prioriteitswachtrij (priority_queue)

De wachtrij

- **FIFO**-structuur (First In, First Out):

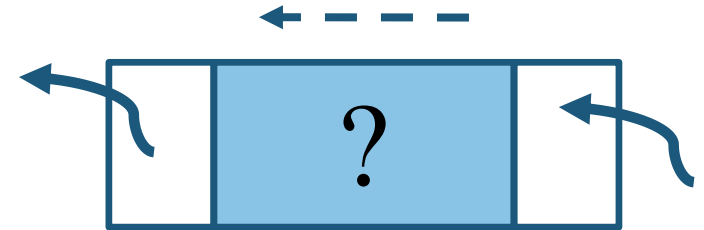
element dat als eerste in de wachtrij

werd geplaatst (**push**), wordt er als eerste terug afgehaald (**pop**).

⇒ Bij een herhaaldelijk *pop* worden de elementen er **in volgorde** afgehaald.

- **Nadeel wachtrij**: men kan niet aan het *i*-de element en men kan de elementen niet overlopen zonder ze te verwijderen.

⇒ geen sequentiële container



Wachtrijen in C++: queue

- De C++ standard library voorziet de klasse **queue** uit `<queue>` om met wachtrijen te werken.
- Enkele methodes:
 - **void push(*type* x)**: Voegt x toe aan de wachtrij
 - **void pop()**: Verwijdert het eerste element van de wachtrij
Vb: `qu.push('a'); qu.pop();`
 - **ref front()**: Geeft referentie naar het eerste element van de wachtrij terug
Vb: `cout << qu.front();`

– **ref** **back()**: Geeft referentie naar het laatst toegevoegde element van de wachtrij terug

Vb: `qu.back() *= 2;`

– **bool** **empty()**: Gaat na of de wachtrij leeg is

Vb: `if (qu.empty()) cout << "leeg";`

– **int** **size()**: Geeft het aantal elementen van de wachtrij terug

Vb: `int grootte = qu.size();`

- Voorbeeld: **vb_queue.cpp**

Voorbeelden container adapters

- Stapel (stack)
- Wachtrij (queue)
- **Prioriteitswachtrij (priority_queue)**

De prioriteitswachtrij

- Is een uitbreiding van de wachtrij: enkel toegang tot element met de met de grootste waarde.
- Elementen worden **geordend** (in dalende volgorde) opgeslagen in de wachtrij.
⇒ de template-parameter moet < implementeren
- De C++ standard library voorziet hiervoor de klasse **priority_queue** uit <queue>.
- Voorbeeld: **vb_priorqueue.cpp**

Inhoud

- Inleiding collections
- Sequences
- Container adapters
- **Associatieve containers**

Associatieve containers

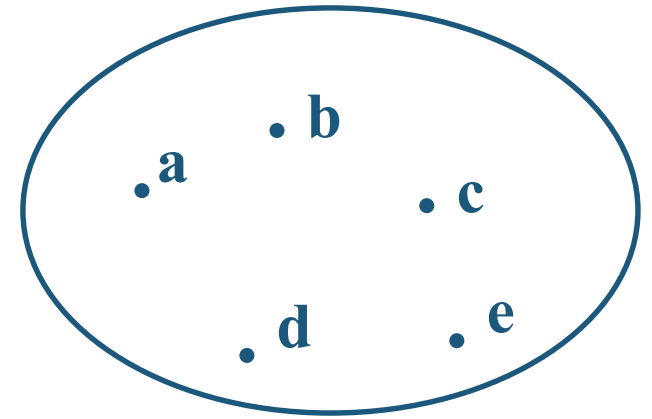
- Slaan gegevens op die bestaan uit een sleutel met bijbehorende informatie.
- Via die sleutel kan men gegevens efficiënt opzoeken om de bijhorende informatie te raadplegen of te wijzigen.

Voorbeelden associatieve containers

- **verzameling en multi-verzameling**
 - `set` en `unordered_set`
 - `multiset` en `unordered_multiset`
- **afbeelding en multi-afbeelding**
 - `map` en `unordered_map`
 - `multimap` en `unordered_multimap`

De verzameling

- Een verzameling (Engels: set) is een container waarin elk element uniek is.
- Is een vereenvoudigde associatieve container: sleutels hebben geen bijhorende informatie
- Volgende operaties zijn mogelijk:
 - een element toevoegen: er gebeurt niets indien dit element al aanwezig is
 - een element opzoeken
 - de elementen overlopen



De verzameling: families

Er zijn 2 families:

- gebaseerd op een **binaire zoekboom**:
 - elementen worden in **gesorteerde volgorde** opgeslagen
 - elementen kunnen **vrij snel** toegevoegd of gevonden worden
- gebaseerd op een **hashtabel**:
 - elementen kunnen **zeer snel** toegevoegd of gevonden worden
 - elementen worden **niet in gesorteerde volgorde** opgeslagen

De verzameling in C++: set en unordered_set

- In de C++ standard library is de klasse **set** een verzameling gebaseerd op een **binaire zoekboom** verzameling.

⇒ de template-parameter moet < implementeren

De klasse **unordered_set** is een gebaseerd op een **hashtabel**.



- Enkele methodes:

– `pair<iterator, bool> insert(type x):`

Voegt x toe (als x nog niet aanwezig is)

Vb: `p.insert(5);`

```
pair<set<int>::iterator, bool> p = s.insert(1);
```

```
if (p.second) cout << "toegevoegd" << endl;
```

```
cout << *p.first << endl; //output = 1
```

– **int** **erase**(*type* **x**): Verwijdert x (als x voorkomt)

Vb: **int** **i** = **s.erase**(1);

 //i = #keer 1 verwijderd

– **int** **count**(*type* **x**): Telt hoeveel keer x voorkomt (0 of 1)

– **void** **clear**(): Wist de volledige verzameling

– **bool** **empty**(): Gaat na of de verzameling leeg is

– **int** **size**(): Geeft het aantal elementen in de verzameling terug

- Gebruik een for-each-lus of iterators om de verzameling te overlopen
⇒ gebruik géén gewone for-lus met indices
- **Iteratoren** kunnen ook gebruikt worden om elementen te zoeken,
toe te voegen of te verwijderen

- **iterator** **begin()/end()**: Geeft een iterator terug naar/net voorbij het eerste/laatste element
 - **iterator** **find(*type* x)**: geeft een iterator terug naar het gezochte element, of de end-iterator indien *x* niet voorkomt
 - **void** **erase(iterator it)**:
Verwijdert het element bij de iterator *it*
 - **iterator** **insert(iterator it, *type* x)**:
Voegt *x* toe na (C++98) / vóór (C++11) de iterator *it*
Vb: `s.insert(it,8);`
- Voorbeeld: **vb_set.cpp**
- Enkel efficiënt indien
iterator goed staat

De multi-verzameling

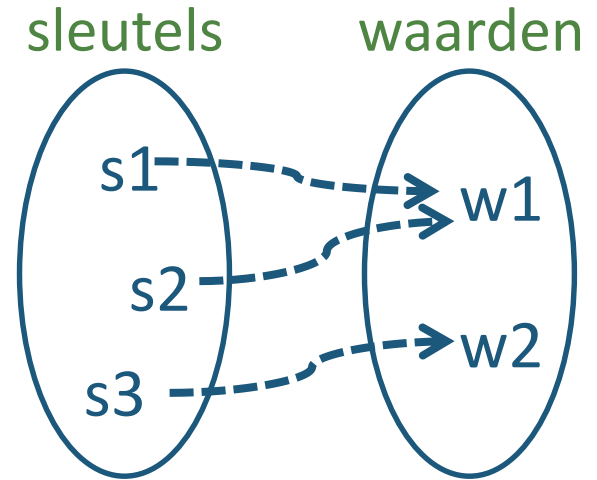
- Is een variant op de verzameling.
- **Duplicaten** worden bewaard.
⇒ een element kan meerdere keren voorkomen
- In de standard C++ library kan de klasse **multiset** of (sinds C++11) **unordered_multiset** gebruikt worden voor een multi-verzameling.
- Voorbeeld: **vb_multiset.cpp**

Voorbeelden associatieve containers

- verzameling en multi-verzameling
 - set en unordered_set
 - multiset en unordered_multiset
- **afbeelding en multi-afbeelding**
 - map en unordered_map
 - multimap en unordered_multimap

De afbeelding

- Een afbeelding (Engels: map) is een verzameling **sleutel-waarde** paren (key/value pairs).
- Bij het toevoegen geef je een sleutel en bijhorende waarde op.



Aan de hand van de sleutel kan een waarde opgezocht worden.

- Een (multi)map is een (multi)set van paren.

De (multi)map in C++: (multi)map en unordered_(multi)map

- In de standard C++ library is de klasse **(multi)map** een op een **binaire zoekboom** gebaseerde (multi)set van paren.
⇒ de paren worden in stijgende sleutelvolgorde bewaard
⇒ sleutels moeten de operator < ondersteunen

Bij de klasse **unordered_(multi)map** is de sleutelverzameling gebaseerd op een **hashtabel**.



- Gebruik **[sleutel]** om in een ~~multimap~~ de bijhorende waarde op te vragen/in te stellen of *sleutel* toe te voegen met default informatie.

- Enkele methodes:
 - `pair<it,bool> insert(pair<type1,type2> p):`
Voegt pair p toe (indien bij map sleutel nog niet voorkomt)
Vb: `m.insert(pair<char,int>('b',3));`
 - `int erase(type x):` Verwijdert sleutel x (als x voorkomt)
Vb: `int i = s.erase('a');`
 - `int count(type x):` Telt hoeveel keer sleutel x voorkomt
 - `void clear():` Wist de volledige afbeelding
 - `bool empty():` Gaat na of de afbeelding leeg is
 - `int size():` Geeft het aantal elementen in de afbeelding terug

- Gebruik een for-each-lus of iterators om de (multi)map te overlopen

Voorbeeld:

```
for (pair<char,int> p : m)
    cout << p.first << "->" << p.second << endl;

map<char,int>::iterator it = m.begin();
while (it != m.end()) {
    cout << (*it).first << "->"
        << it->second << endl;
    it++;
}
```

- **Iteratoren** kunnen ook gebruikt worden om elementen te zoeken, toe te voegen of te verwijderen

- **iterator** **begin()/end()**: Geeft een iterator terug naar het eerste element / net voorbij het laatste element
- **iterator** **find(keytype x)**: geeft een iterator terug naar sleutel x, of de end-iterator indien x niet voorkomt
- **void** **erase(iterator it)**:
Verwijdert het element bij de iterator *it*
- **iterator** **insert(iterator it, pair<type1,type2> p)**:
Voegt pair p toe na (C++98) / vóór (C++11) de iterator *it*
Vb: `m.insert(it, pair<char,int>('b',3));`

- Voorbeeld: **vb_map.cpp**

Enkel efficiënt indien
iterator goed staat

map<int,...> versus vector<...>

Hoe houden we best alle informatie bij van volgende veelterm?

$$8x^{501} - 3x^2 - 5$$

Bij een vector zijn alle elementen $0 \dots \text{size}() - 1$ aanwezig, maar bij een map niet.

⇒ voordeel: plaatsbesparing

⇒ nadeel: [] is bij een map minder efficiënt

Inhoud

- Inleidend collections
- Sequences
- Container adapters
- Associatieve containers