

# HOOFDSTUK 7

## NIEUWE FEATURES

### SINDS C++11 (DEEL 2)

# Inhoud

- **automatische type-afleiding (auto)**
- gewijzigde initialisatiesyntax
- *defaulted* en *deleted* lidfuncties
- move constructor en move operator

# Automatische type-afleiding

- Vroeger: bij declaratie steeds type object expliciet speciëren
- Sinds C++11: indien declaratie object **vergezeld** wordt **van initialisatie**, is gebruik van **auto** mogelijk

```
auto a = 0; // int  
auto b = 'a'; // char
```

Nut??

⇒ vooral nuttig bij zeer complexe types (vb bij STL iterators)

```
template <typename T>  
void f(const set<T> &s) {  
    typename set<T>::const iterator ci = s.begin();  
    auto  
}
```

Dit moet er staan!!!

# Inhoud

- automatische type-afleiding (auto)
- **gewijzigde initialisatiesyntax**
- *defaulted* en *deleted* lidfuncties
- move constructor en move operator

# Initialisatiesyntax

- Traditioneel verschillende initialisatienotaties:

- constructor (met haakjes): `string s("hello");`

- accolades (voor structs, arrays): `int t[4] = {0,1,2,3};`

- initializer list bij  
definitie constructoren:

```
class A {  
    int x;  
    public:  
        A() : x(0) {}  
};
```

- In C++11 invoering van uniforme notatie met accolades

```
class A {  
    int x;  
    int y[4];  
public:  
    A(int _x=0) : x{_x}, y{1,2,3,4} {}  
};
```

```
int main() {  
    A a{3};  
    int b{2};  
    int *t = new int[3] {1, 2, 3};  
    vector<int> v = {10,20};  
    return 0;  
}
```

≡ A a(3);

≡ int b = 2;

**Let op:**

A a(); **//FOUT**

A a{}; **//OK**

A a; **//OK**

vóór C++11 kon array die  
gealloceerd is met **new** niet  
geïnitieerd worden!

vervangt push\_back()-lijst

≡ vector<int> v{10,20};

≡ vector<int> v({10,20});

≠ vector<int> v(10,20);

- Sinds C++11: in-klasse initialisatie van attributen is toegelaten

```
class A {  
    int x = 7;  
    int y{9};  
  
    public:  
        A() {}  
};
```

enkel vanaf C++11 mogelijk!!!

Let op:

`int y(9);` **//error**

# Inhoud

- automatische type-afleiding (auto)
- gewijzigde initialisatiesyntax
- **defaulted en deleted lidfuncties**
- move constructor en move operator



# defaulted en deleted lidfuncties

- **defaulted lidfunctie**

```
class A {  
    public:  
        A() = default;  
        A(const A&) = default;  
};
```

instrueert de compiler expliciet om de default implementatie te genereren voor deze functie

- **deleted lidfunctie**

```
class A {  
    public:  
        A(const A&) = delete;  
};
```

verbiedt de compiler automatische generatie van deze functie  
(= tegenovergestelde van default)

```
A a; A b(a);
```

**compilatie-fout!!**

# Inhoud

- automatische type-afleiding (auto)
- gewijzigde initialisatiesyntax
- *defaulted* en *deleted* lidfuncties
- **move constructor en move operator**

# Probleemstelling

- kopiëren kost tijd en geheugen
- Voorbeeld

**vb\_zonder\_move.cpp**

```
int main() {  
    vector<set<int>> v;  
    for (int i=1; i<=3; i++) {  
        set<int> s1, s2;  
        ... // voeg iets toe aan s1  
  
        v.push_back(s1);  
        s2 = s1;  
    }  
}
```

**Er wordt 2 maal een  
kopie genomen van s1**

# move constructor/operator

- copy constructor/toekenningoperator: *kopieert* elk attribuut (default: geen diepe kopie)
- doel move constructor/operator: geen kopie maken van het origineel, maar de attributen van het origineel '**schaken**'
- Hoe move constructor/operator gebruiken?

copy constructor:     A b(a);

toekenningoperator: c = b;

move constructor:    A b(move(a));

move operator:       c = move(b);

# Voorbeeld gebruik move operator

- Voorbeeld

vb\_move1.cpp

```
int main() {  
    vector<set<int>> v;  
    for (int i=1; i<=3; i++) {  
        set<int> s1, s2;  
        ... // voeg iets toe aan s1  
  
        v.push_back(move(s1));  
  
        ... //voeg opnieuw iets toe aan s1  
  
        s2 = move(s1);  
    }  
}
```

sinds c++11 is er een versie van push\_back die move ondersteunt

move constructor/operator “verplaatst” s1  
=> s1 wordt leeg


# implementatie move constructor/operator

- move constructor/operator wordt standaard voorzien
  - ⇒ voert move constructor/operator uit op alle attributen
  - ⇒ maar wat met primitieve types en pointers? zie **move\_test.cpp**
- Hoe move constructor/operator schrijven?
  - attribuut dat geen pointer of primitief type is: kan (meestal) a.d.h.v. de overeenkomstige move constructor/operator van het attribuut
  - attribuut dat pointer is: neem ondiepe kopie (dus geen extra verplaatsing in geheugen) én zet originele pointer op nullptr
  - attribuut dat primitief type is: neem kopie én zet originele variabele op 0

# Implementatie move constructor

## Voorbeeld

```
class A {  
    public:  
        A(const A &);  
        A(A &&);  
        ...  
    private:  
        vector<int> vA;  
        int grA;  
        int *tabA;  
};
```



copy constructor



move constructor

## // copy constructor

```
A::A(const A &a) : vA(a.vA), grA(a.grA), tabA(nullptr) {  
    if (grA > 0) {  
        tabA = new int[grA];  
        for(int i=0 ; i<grA ; i++)  
            tabA[i] = a.tabA[i];  
    }  
}
```

**move constructor is veel  
eenvoudiger + efficiënter  
dan copy constructor**

## // move constructor

```
A::A(A &&a) : vA(move(a.vA)), grA(a.grA), tabA(a.tabA) {  
    a.grA = 0; a.tabA = nullptr;  
}
```



# Implementatie move operator

- Voorbeeld

```
class A {  
    public:
```



toekenningsoperator

```
        A& operator=(const A &);
```

```
        A& operator=(A &&);
```



move operator

```
        ...
```

```
    private:
```

```
        vector<int> vA;
```

```
        int grA;
```

```
        int *tabA;
```

```
};
```

```
// toekenningsoperator
```

```
A& A::operator=(const A& a) {  
    if (this != &a) {  
        vA = a.vA;  
        delete[] tabA;  
        grA = a.grA;  
        if (grA > 0) {  
            tabA = new int[grA];  
            for(int i=0 ; i<grA ; i++)  
                tabA[i] = a.tabA[i];  
        }  
    }  
    return *this;  
}
```

## // move operator

```
A& A::operator=(A&& a) {  
    if(this != &a) {  
        vA = move(a.vA);  
        delete[] tabA;  
        grA = a.grA;  
        tabA = a.tabA;  
        a.grA = 0;  
        a.tabA = nullptr;  
    }  
    return *this;  
}
```

## opmerking

move operator wordt (indien beschikbaar) automatisch opgeroepen als rechterlid 'à la minute' aangemaakt werd

# THE BIG THREE <-> THE BIG FIVE

vóór C++11

## THE BIG THREE

- operator=
- copy constructor
- destructor

sinds C++11

## THE BIG FIVE

- operator=
- copy constructor
- destructor
- **move constructor**
- **move operator**

# Rule of ...

- Vóór C++11: **rule of three**

*If a class requires a user-defined destructor, a user-defined copy constructor or a user-defined copy assignment operator, it almost certainly requires all three.*

- Sinds C++11: **rule of five**

*If you define or =delete any default operation (= destructor, copy-constructor, copy-assignment, move constructor and the move assignment operator), define or =delete them all.*

# Rule of ...

- **Rule of zero**

*Classes that have custom destructors, copy/move constructors or copy/move assignment operators should deal exclusively with ownership (which follows from the Single Responsibility Principle). Other classes should not have custom destructors, copy/move constructors or copy/move assignment operators.*

**=> If you can avoid defining default operations, do**

# Inhoud

- automatische type-afleiding (auto)
- gewijzigde initialisatiesyntax
- *defaulted* en *deleted* lidfuncties
- move constructor en move operator