

# HOOFDSTUK 4

## OGP IN C++

**Helga Naessens**

# Inhoud

- **Klassen in C++**
- Werking constructor-destructor
- Copy-constructor
- Separate compilatie
- Objecten als attributen
- Friend functies en klassen
- Operator overloading
- Klasse-templates

# Klassen in C++

- klassendeclaratie ≠ klassendefinitie
  - declaratie: attributen + signatuur van lidfuncties  
Vergeet afsluitende ; niet na } van declaratie!
  - definitie lidfunctie buiten declaratie met **scope-operator ::**  
(vlak voor de naam van de lidfunctie)
- in klassendeclaratie **private:** en **public:** 1x vermelden
  - default: alles private
- **default waarden mogen slechts 1x vermeld worden**  
(meestal in klassendeclaratie)

```
class voorbeeld {  
    public:  
        void set_a(int = -1);  
    private:  
        int a, b;  
};  
  
void voorbeeld::set_a(int waarde) {  
    a = waarde;  
}
```

- Gewone lidfuncties kan men onderverdelen in 2 groepen:
  - **mutators**: wijzigen (attributen) huidige object (vb setters)
  - **accessors**: wijzigen (attributen) huidige object niet (vb getters)

In C++ kan hierin expliciet een onderscheid gemaakt worden (zodat compiler kan controleren).

Deze '**const**' **lidfuncties** vermelden het sleutelwoord **const** na hun parameterlijst, om te garanderen dat ze geen enkel attribuut wijzigen.

**Als de definitie apart gebeurt, moet const daar herhaald worden,** want het behoort tot de signatuur van de lidfunctie.

ENKELE eventueel toe te voegen bij LIDfuncties!!!

```
class voorbeeld {  
    public:  
        void set_a(int = -1);  
        int get_a() const;  
    private:  
        int a, b;  
};  
  
int voorbeeld::get_a() const {  
    return a;  
}
```

Opmerking: indien je van de setter een const-lidfunctie zou maken, zal de compiler reclameren!!

- **Constructor:**     *kLassenaam([parameterLijst])*
    - wordt automatisch opgeroepen bij creatie object  
=> maakt een correct geïnitieerd object
    - heeft geen return-value (zelfs geen void)
    - constructor overloading bestaat  
maar uiteraard zijn ook default-parameters toegestaan
    - indien de klasse geen constructor bevat, wordt een automatische default constructor voorzien:  
  
primitief type: rommel                      klasse: default constructor
- Van zodra de klasse een constructor bevat, is er geen automatische default constructor meer.

- **Constructie van objecten** in C++

```
// constructie bij declaratie
```

```
student std1;
```

```
//NIET: student std1();
```

```
//WEL OK (zie later): student std1{};
```

```
student std2("Tom");
```

```
std1 = student("Jan"); // herinitialisatie achteraf
```

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



- **Destructor:** `~kLassenaam()`
  - heeft geen return-value (zelfs geen void)
  - heeft geen argumentenlijst
  - géén destructor overloading
  - wordt automatisch opgeroepen bij “out of scope”
  - destructor enkel noodzakelijk indien dynamisch aangemaakte componenten (met new)
  - Voorbeeld: **student.cpp** (map Vb1Algemeen)

# Inhoud

- Klassen in C++
- **Werking constructor-destructor**
- Copy-constructor
- Separate compilatie
- Objecten als attributen
- Friend functies en klassen
- Operator overloading
- Klasse-templates

# Werking constructor-destructor

```
class myclass {  
    int i;  
    public:  
    myclass(int i);  
    ~myclass() {  
        cout << "Destructor object " << i << endl;  
    }  
};  
  
myclass::myclass(int i) {  
    cout << "Constructor object "<< i << endl;  
    this->i = i;  
}
```

pointer **this** verwijst naar het huidige object

# Output?

```
int main() {  
    myclass a(1);  
    for (int i=2 ; i<4 ; i++)  
        myclass b(i);  
    return 0;  
}
```

```
Constructor object 1  
Constructor object 2  
Destructor object 2  
Constructor object 3  
Destructor object 3  
Destructor object 1
```

- Voorbeeld: **demoConstrDestr.cpp**  
(map Vb2ConstrDestr)

- **Objecten op de runtime stack**

**<klassenaam> <identifier>;**

Voorbeeld: myclass a;

allocatie geheugen object met naam a

oproep default constructor voor a

**<klassenaam> <identifier>(<argumentlijst>;**

Voorbeeld: myclass a(1);

allocatie geheugen object met naam a

oproep constructor met argumenten voor a

Bij **out of scope** gaan van deze objecten wordt de destructor opgeroepen en wordt het geheugen op de runtime stack vrijgegeven.

- **Objecten op de heap**

```
new <klassenaam>(<argumentlijst>opt);
```

Voorbeeld: myclass \*a = new myclass;

myclass \*a = new myclass(1);

Bij **out of scope** gaan van deze objecten wordt de destructor **NIET** opgeroepen en wordt het geheugen op de heap **NIET** vrijgegeven.

```
delete <identificer>;
```

Voorbeeld: delete a;

roept destructor op en geeft geheugen op de heap vrij

```
int main() {  
    myclass *a = new myclass(1);  
    unique_ptr<myclass> up = make_unique<myclass>(11);  
    for (int i=2 ; i<4 ; i++)  
        myclass *b = new myclass(i);  
    delete a;  
    return 0;  
}
```

Output?

```
Constructor object 1  
Constructor object 11  
Constructor object 2  
Constructor object 3  
Destructor object 1  
Destructor object 11
```

- Voorbeeld: **DemoHeap.cpp** (map Vb2ConstrDestr)

# Delegerende constructoren



- Vanaf C++11 mag een **constructor** een andere constructor van dezelfde klasse oproepen via de **initializer list**

```
class voorbeeld {  
    public:  
        voorbeeld(int _a, int _b) {  
            a = _a; b = _b;  
        }  
        voorbeeld() : voorbeeld(1, 2) {}  
    private:  
        int a, b;  
};
```



# Inhoud

- Klassen in C++
- Werking constructor-destructor
- **Copy-constructor**
- Separate compilatie
- Objecten als attributen
- Friend functies en klassen
- Operator overloading
- Klasse-templates

# Copy-constructor

- wordt in C++ standaard gebruikt in 2 situaties
  - `student std2(std1);`
  - `void proc(student st) //st is value-parameter`
- is standaard **steeds** aanwezig  
(zelfs bij aanwezigheid van andere constructoren)  
standaard copy-constructor kopieert geen dynamisch aangemaakt geheugen → shared structure
- zelf definiëren is dus soms noodzakelijk!!
- syntax: `A(const A &a)` //NIET: `A(A a)`
- Voorbeeld: `student_oud/nieuw.cpp` (map Vb3CopyConstr)

# Inhoud

- Klassen in C++
- Werking constructor-destructor
- Copy-constructor
- **Separate compilatie**
- Objecten als attributen
- Friend functies en klassen
- Operator overloading
- Klasse-templates

# Separate compilatie

- indien **binnen project** klasse nodig is in meerdere klassen:  
niet broncode includeren in elke klasse, wel header-file
- werkwijze: aparte files
  - header-file (student.h): klassendeclaratie met compiler directieven

```
#ifndef STUDENT_H
#define STUDENT_H
...
#endif
```
  - bronfile van klasse (student.cpp): gecompileerd tot object-file
  - bronfile van toepassing (demo.cpp): gecompileerd en gelinkt
- Voorbeeld: project **project1** (map Vb4SepComp)

# Inhoud

- Klassen in C++
- Werking constructor-destructor
- Copy-constructor
- Separate compilatie
- **Objecten als attributen**
- Friend functies en klassen
- Operator overloading
- Klasse-templates

# Objecten als attributen

- wordt **associatie/aggregatie/compositie** genoemd
- wanneer het buitenste blok van een constructor betreden wordt, zijn de attributen reeds aanwezig
  - ⇒ primitieve attributen bevatten rommel
  - ⇒ objectattributen werden geconstrueerd met hun default constructoren (als die bestaat, anders: **FOUT!!**)
- met een **initializer list** kan men de constructie en initialisatie van attributen laten gebeuren vooraleer het buitenste blok betreden wordt.
- syntax: `A(const B &b, int i) : attr1(b), attr2(i) { ... }`
- voorbeeld: project **ProjectInitList** (map Vb5InitList)

# Inhoud

- Klassen in C++
- Werking constructor-destructor
- Copy-constructor
- Separate compilatie
- Objecten als attributen
- **Friend functies en klassen**
- Operator overloading
- Klasse-templates

# Friend functies

- Friend functie van een klasse
  - géén lidfunctie!
  - heeft directe toegang tot private attributen en methoden
    - ⇒ heeft zelfde rechten als lidfuncties
- Declaratie friend functie: keyword **friend**
  - gedeclareerd in de klassendeclaratie
  - buiten klasse gedefinieerd
    - **niet in scope van klasse** ⇒ géén *klassemnaam::*
    - **géén herhaling sleutelwoord friend**



- Voorbeeld: **DemoFriendFunctie.cpp** (map Vb6Friend)


```
class A {
    int i;
public:
    A(int _i=0) : i(_i) {}
    int get_i() const;
    friend int fr(const A &);
};

int A::get_i() const { return i; }
int fr(const A &a) { return a.i; }

int main() {
    A a(7); cout << a.get_i() << " " << fr(a); // 7 7
    return 0;
}
```

# Friend klassen

```
class Y { ... };  
class Z {  
    friend class Y;  
    ...  
};
```



Klasse Y is friend van klasse Z:

- ⇒ class Y can now access data of class Z directly
- ⇒ NIET omgekeerd
- ⇒ friendship wordt toegekend, niet genomen

- Voorbeeld: **DemoFriendClass.cpp** (map Vb6Friend)

- Concept friend in OO:
  - Volgens stricte OO principe dienen alle functies lidfuncties te zijn  
⇒ concept friend schendt eigenlijk de OO basisprincipes!
  - Toch gebruikt in C++
    - vooral bij **operator overloading**
    - voor **efficiëntieverhoging**: directe toegang tot private attributen van een klasse  
(dus geen overhead voor toegang tot private attributen via publieke member-functies (spaart extra functie oproepen))

# Inhoud

- Klassen in C++
- Werking constructor-destructor
- Copy-constructor
- Separate compilatie
- Objecten als attributen
- Friend functies en klassen
- **Operator overloading**
- Klasse-templates

# Operator overloading

- standaard zijn in C++ op objecten 2 operatoren gedefinieerd: `=` en `&`

## Voorbeeld

```
tijd t1, t2(12,36,25), *pt;
```

```
t1 = t2;    pt = &t2;
```

- deze zijn te herdefiniëren, net als alle andere operatoren
- syntax:

```
return_type operatornaam([par_lijst]) [const] ;
```

Voorbeeld:    `t1 = t2 + t3`    wordt gecompileerd als

`t1.operator=(t2.operator+(t3))` (indien lidfuncties) of  
`operator=(t1,operator+(t2,t3))` (indien externe functies)

- Overzicht interessantste te herdefiniëren operatoren:

+	-	*	/	%	
<	<=	>	>=	==	!=
+=	-=	*=	/=	%=	=
++	--	<<	>>	[]	

### Let op:

- de compiler genereert automatisch de **= operator** (hierbij wordt een **ondiepe kopie** van de attributen genomen).  
 ⇒ om ervoor te zorgen dat een diepe kopie genomen wordt,  
 moet de = operator soms overschreven worden  
 (soms = indien de klasse pointer-attributen bevat)

- Voorbeeld:

```
int main() {  
    tijd t0, t1(1,51,51), t2(5,30,11), t3;  
    t0 = t1+t2; t3 = t2*2;  
    if (t1<t2)  
        cout << "t1 < t2";  
    return 0;  
}
```

Welke operatoren moeten overschreven worden?  
Geef de signatuur van deze operatoren/lidfuncties.

```
class tijd {  
    private:  
        int uur, min, sec;  
        void herbereken();  
    public:  
        ...  
        tijd operator+(const tijd &) const;  
        tijd operator*(int) const;  
        bool operator<(const tijd &) const;  
};
```

**tijd1.cpp**  
(map Vb7OperOverl)



```
tijd tijd::operator+(const tijd &t) const {  
    tijd som(uur+t.uur, min+t.min, sec+t.sec);  
    som.herbereken(); return som;  
}  
  
tijd tijd::operator*(int factor) const {  
    tijd tmp(uur*factor, min*factor, sec*factor);  
    tmp.herbereken(); return tmp;  
}  
  
bool tijd::operator<(const tijd &t) const {  
    return uur*3600+min*60+sec <  
        t.uur*3600+t.min*60+t.sec;  
}
```

- **Unaire operator:** heeft één operand

Voorbeeld: `t1 = -t2;`

```
tijd operator-() const {  
    return tijd(-uur, -min, -sec);  
}
```

Kan bestaan naast:

```
tijd operator-(const tijd &t) const
```

- **Overloaden van een toekenningsoperator:**

=      +=      -=      \*=      /=      %=

Enkel de =operator is standaard voorzien!!!

Voorbeeld:

```
int a=b=123;
```

```
(a=5)++; // resultaat van (a=5) moet a zijn!!
```

```
(b+=3)*=2; // resultaat van (b+=3) moet b zijn!!
```

resultaat is:

- geen tijdelijk object, geen kopie, maar **het linkerlid zelf**
- **return by reference: *type&***  
=> **return \*this;**

```

class tijd {
    ...
    tijd& operator+=(const tijd &t);
};

tijd& tijd::operator+=(const tijd &t) {
    sec += t.sec; min += t.min; uur += t.uur;
    herbereken(); return *this;
}

int main(){
    tijd nu; tijd t(1,0,0);
    (nu += t) += t;
    ...
}

```

- Overloaden van prefix en postfix operatoren ++ en --

- prefix operator is default

```
tijd t(20,30,0);  
++(++t); //verhoog telkens met 1 sec
```

=> signatuur prefix ++: tijd& operator++();

- welk returntype voor postfix operator?

```
tijd t(20,30,0);  
tijd t2 = t++; //verhoog met 1 sec
```

=> kopie nodig van vorige staat van object

=> tijd operator++();

**FOUT: verwarring met prefix operator!!**

- onderscheid nodig in signatuur tussen prefix- en postfix-operator:

```
class A {  
    public:  
        A& operator++();    // ++x  
        A operator++(int); // x++  
        A& operator--();    // --x  
        A operator--(int); // x--  
};
```

⇒ int argument geeft aan dat het om postfix-operator gaat  
(deze waarde wordt NIET gebruikt!!)

**tijd2.cpp**

(map Vb7OperOverl)

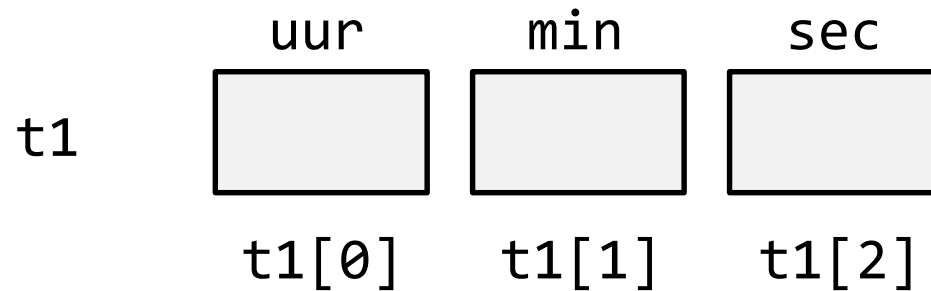
```
class tijd {  
    ...  
    tijd& operator++();  
    tijd operator++(int);  
};  
  
tijd& tijd::operator++() {  
    sec++; herbereken(); return *this;  
}  
  
tijd tijd::operator++(int a) {  
    tijd temp(*this);  
    sec++; herbereken(); return temp;  
}
```

- Voorbeelden van operatoren gedeclareerd in klasse A:

naam	return_type	arg_lijt	const (J/N)
+	A	const A&	J
-	A	const A&	J
- (unair)	A	(geen)	J
==	bool	const A&	J
<	bool	const A&	J
=	A&	const A&	N
+=	A&	const A&	N
++(prefix)	A&	(geen)	N
++(postfix)	A	int	N
[]	<i>type</i> &	int	N



```
class tijd {  
    ...  
    ...    operator[](int);  
};
```



Gebruik:     `t1[2] = t1[1]; t1[0]++;`

⇒ signatuur:

```
int& operator[](int);
```

```
class tijd {  
    ...  
    int& operator[](int);  
};  
  
int& tijd::operator[] (int i) {  
    if (i==0) return uur;  
    else if (i==1) return min;  
    else return sec;  
}  
  
int main() {  
    tijd t2(6,45,51); t2[2]=0; t2[1]++; ...  
}
```

- **Bevriende operatoren:**

`t1 = t2 * 2;`

$\Rightarrow$  via definitie van `tijd operator*(int f) const;`

`t1 = 2 * t2;`

2 opties:

1) via definitie van operator die buiten klasse gedeclareerd is:

toegang tot private attributen via publieke methoden  
(bijv. set/get methodes)

`tijd operator*(int f, const tijd &t);`

2) via definitie van friend operator:

rechtstreekse toegang tot private attributen

`friend tijd operator*(int f, const tijd &t);`

```
class tijd {  
    ...  
    friend tijd operator*(int f, const tijd &t);  
};  
  
tijd operator*(int f, const tijd &t) {  
    tijd tmp = tijd(t.uur*f,t.min*f,t.sec*f);  
    tmp.herbereken(); return tmp;  
}
```

beter (en korter): **return t\*f;**

⇒ operator kan dan als externe operator gedefinieerd worden (niet als friend-operator)

- **overloaden ostream-operatoren** << en >>
  - werken in op cin of ifstream, cout of ofstream (linkeroperand)
    - ⇒ kunnen geen lidfunctie van zelf-gedefinieerde klasse zijn!
    - ⇒ wel friend-functie van zelf-gedefinieerde klasse
  - gebruik:

```
cin >> t1 >> t2;  
cout << "t1 = " << t1 << endl << "t2 = " << t2;
```
  - syntax:

```
friend istream& operator>>(istream& is, A& a)  
friend ostream& operator<<(ostream& os, const A& a)
```

```
class tijd {  
    ...  
    friend ostream& operator<<(ostream& os, const tijd& t);  
    friend istream& operator>>(istream& is, tijd& t);  
};  
  
ostream& operator<<(ostream& os, const tijd& t) {  
    os << setw(2) << setfill('0') << t.uur << ':' << ...  
        << setw(2) << setfill('0') << t.sec;  
    return os;  
}  
  
istream& operator>>(istream& is, tijd& t) {  
    is >> t.uur >> t.min >> t.sec; t.herbereken();  
    return is;  
}
```

# Inhoud

- Klassen in C++
- Werking constructor-destructor
- Copy-constructor
- Separate compilatie
- Objecten als attributen
- Friend functies en klassen
- Operator overloading
- **Klasse-templates**

# Klasse-templates

```
template <typename T> //meerdere typenames mogelijk
class koppel { //hier géén <T> toevoegen!!
    public:
        koppel();
        koppel(T,T);                                (*)
        void set_first(T);
        T get_first() const;                        (*)
        koppel<T> operator+(const koppel<T> &) const;
    private:
        T first, second;
};
```



Elke definitie is op zichzelf een "template"  
=> vereist template prefix voor elke definitie

```
template <typename T>
```

```
koppel<T>::koppel(T _first, T _second)  
    : first(_first), second(_second) {}
```

Voor :: telkens *klasse\_naam*<T>

```
template <typename T>
```

```
T koppel<T>::get_first() const { return first; }
```

Gebruik:

```
koppel<int> p(9,2);  
p.set_first(3);  
cout << p.get_first();
```

# Inhoud

- Klassen in C++
- Werking constructor-destructor
- Copy-constructor
- Separate compilatie
- Objecten als attributen
- Friend functies en klassen
- Operator overloading
- Klasse-templates