FACULTEIT INGENIEURSWETENSCHAPPEN
EN ARCHITECTUUR

HOOFDSTUK 5

OVERERVING

Helga Naessens

UNIVERSITEIT
GENT

Inhoud

- **public versus private overerving**
- constructoren/destructor in afgeleide klasse
- overschrijven toekenningsoperator
- keyword protected
- multiple inheritance
- polymorfisme en dynamic binding
- virtuele destructor
- abstracte klassen

public en *private* overerving

in C++ bestaan er verschillende soorten overerving

1. **public overerving**

```
class student : public persoon {…};
```

2. **private overerving**

```
class student : private persoon {…};
```

```
class student : persoon {…};
```

3. **protected overerving** (zie verder)

=> hebben een fundamenteel verschillend gedrag

private overerving

- **opgelet**: geen vermelding van `public` impliceert `private`
- **publieke leden** van basisklasse **worden private leden** van afgeleide klasse

  $\Rightarrow$ de programmeur van de afgeleide klasse kan gebruik maken van de lidfuncties van de basisklasse, maar deze zijn ontoegankelijk voor de gebruikers van de afgeleide klasse

  $\Rightarrow$ objecten van type `student` kunnen de publieke lidfuncties van `persoon` NIET gebruiken
- is geen echte overerving binnen OGP-termen, want is **geen *is een* – relatie**

public overerving

- is eigenlijke overerving binnen OGP-termen, want is een

  is een – relatie $\Rightarrow$ `student` is een `persoon`
- objecten van type `student` kunnen de publieke lidfuncties

  van `persoon` gebruiken

  $\Rightarrow$ een gebruiker mag met een object van de afgeleide klasse

  doen wat hij met een object van de basisklasse ook kan.

Deze vorm van overerving is de meest gebruikte

en de meest aangewezen.

$\Rightarrow$ vergeet `public` niet na `:`

Voorbeeld *public* overerving

```
class A {
 public:
  void setVarA(int i);
  int getVarA() const;
 private:
  int varA;
};

class B : public A {
 public:
  void setVarB(int i);
  int getVarB() const;
 private:
  int varB;
};
```

Voorbeeld1_basis.cpp

Gebruik:

```
B b;
b.setVarA(8);
b.setVarB(11);
```

Deze opdracht geeft een compileerfout indien gebruik gemaakt wordt van private overerving (geen public na :)

Inhoud

- public versus private overerving
- **constructoren/destructor in afgeleide klasse**
- overschrijven toekenningsoperator
- keyword protected
- multiple inheritance
- polymorfisme en dynamic binding
- virtuele destructor
- abstracte klassen

Constructoren in afgeleide klassen

- Constructoren van de basisklasse **worden NIET overgeërfd**!

  $\Rightarrow$ Indien de afgeleide klasse geen constructoren declareert,
  wordt er een **automatische default-constructor** aangemaakt
  (die de default-constructor van de basisklasse aanroept)

  $\Rightarrow$ Indien de afgeleide klasse geen copy-constructor declareert,
  wordt er een default **copy-constructor** aangemaakt
  (die de copy-constructor van de basisklasse aanroept)

- Met behulp van de **using-declaratie** in de afgeleide klasse kunnen alle constructoren (met uitzondering van default/copy/move-constructor) toch overgeërfd worden.

```
class A {
 public:
   A(int vA1=-1, int vA2=-2);
   int getVarA1() const;
   int getVarA2() const;
 private:
   int varA1, varA2;
};

class B : public A {
 public:
   using A::A; //(*)
   int getVarB() const;
 private:
   int varB;
};
```

gebruik zonder (*):

```
int main () {
   B b0; B b1(b0); …
} //inhoud attributen?
```

gebruik met (*):

```
int main () {
   B b0;
   B b1(4);
   B b2(5,6);
   B b3(b1);
   …
} //inhoud attributen?
```

Voorbeeld2_using.cpp

Opmerkingen omtrent using

- Indien in de afgeleide klasse een overgeërfde constructor overschreven wordt, wordt de nieuwe constructor opgeroepen.

- using kan niet gebruikt worden om een specifieke constructor te erven,

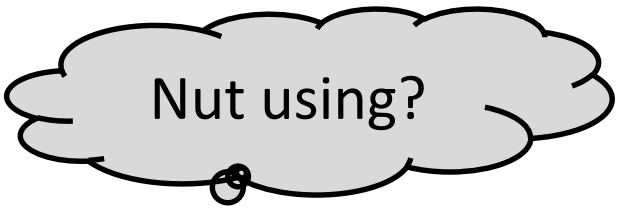
  voorbeeld: ~~`using A::A(int);`~~

  Constructoren kunnen wel gedeleted worden (zie Hoofdstuk 7):

```
class B : public A {
    public:
        using A::A;
        B(int) = delete;
```

Voorbeeld:

copy-constructor deleten



Constructoren in afgeleide klassen

- Indien de constructor van de afgeleide klasse geen constructor van de basisklasse oproept, wordt automatisch de default constructor van de basisklasse opgeroepen
- De constructor van de afgeleide klasse kan de constructor van de basisklasse **enkel oproepen via de initializer list**

```
class A {
 public:
   A(int vA = 5);
   int getVarA() const;
 private:
   int varA;
};

class B : public A {
 public:
   B(int vB = 9);
   B(int vA, int vB);
   int getVarB() const;
 private:
   int varB;
};
```

Implementatie constructoren:

```
A::A(int vA) : varA(vA) {}

B::B(int vB) : varB(vB) {}

B::B(int vA, int vB) :
  A(vA), varB(vB) {}
```

Voorbeeld3_constructor.cpp

Destructor in afgeleide klassen

- Wanneer de destructor van de afgeleide klasse opgeroepen wordt, roept deze AUTOMATISCH de destructor van de basisklasse op

  $\Rightarrow$ hoeft niet expliciet opgeroepen te worden

- Destructor in afgeleide klasse hoeft enkel aandacht te hebben voor extra attributen in de afgeleide klasse

  $\Rightarrow$ vertrouw erop dat destructor van basisklasse correct werkt!

- Volgorde destructor aanroep:

```
class A {
 public:
   ~A(); …
};
class B : public A {
 public:
   ~B(); …
};
class C : public B {
 public:
   ~C(); …
};
```

gebruik:

```
{
   C c; …
} // c out of scope
```

~C , ~B , ~A wordt opgeroepen

(in die volgorde)

$\Rightarrow$ omgekeerde volgorde

van constructor-aanroep

- Voorbeeld: **Voorbeeld4_destructor.cpp**

Merk op:

- Overschrijven copy-constructor:

```
B::B(const B& b) : A(b), … {
   … // kopieer extra attributen van B
}
```

- Om in een subklasse expliciet de methode uit de bovenliggende klasse op te roepen, schrijf je:

```
void B::print() const {
   A::print(); …
}
```

```
B b;
b.print();
b.A::print();
```

Inhoud

- public versus private overerving
- constructoren/destructor in afgeleide klasse
- **overschrijven toekenningsoperator**
- keyword protected
- multiple inheritance
- polymorfisme en dynamic binding
- virtuele destructor
- abstracte klassen

Overschrijven toekenningsoperator

- Toekenningsoperator (=) wordt default aangemaakt indien niet gedeclareerd
- Hoe implementeren/overschrijven?

  Voorbeeld5_probeersel.cpp is niet ok!

```
B& B::operator=(const B &b) {
    if (this != &b) {
      A::operator=(b);
      … // assign extra members of B
    }
    return *this;
}
```

Voorbeeld5_toekenningsoperator.cpp

Inhoud

- public versus private overerving
- constructoren/destructor in afgeleide klasse
- overschrijven toekenningsoperator
- **keyword protected**
- multiple inheritance
- polymorfisme en dynamic binding
- virtuele destructor
- abstracte klassen

Keyword protected

- Laat toegang toe tot attributen (en lidfuncties) in afgeleide klasse
- In de klasse waar het gedefinieerd is:

  zelfde eigenschappen als `private`
- In afgeleide klasse:

  `protected` members van basisklasse zijn eveneens `protected`

  (zodat ook toegankelijk voor verdere afleidingen)
- **Protected overerving**:

  publieke members in basisklasse worden protected in de afgeleide klasse

  $\Rightarrow$ wordt (net als private overerving) zelden gebruikt!

Inhoud

- public versus private overerving
- constructoren/destructor in afgeleide klasse
- overschrijven toekenningsoperator
- keyword protected
- **multiple inheritance**
- polymorfisme en dynamic binding
- virtuele destructor
- abstracte klassen

Multiple inheritance

```
class A { … };
class B { … };
class C : public A,
  public B { … };
```

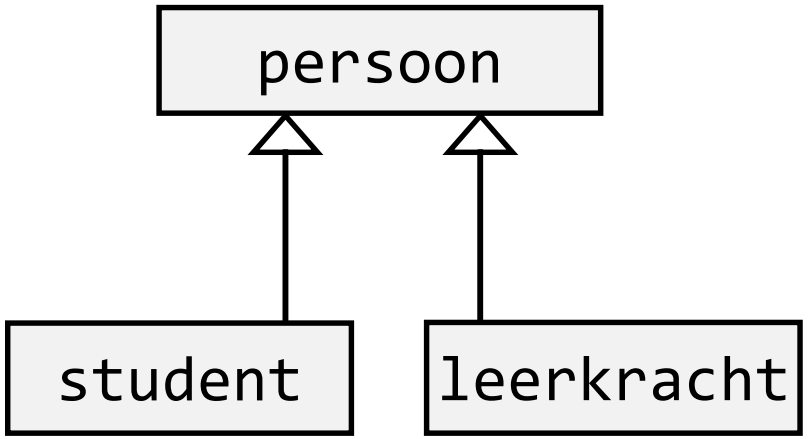
- In tegenstelling tot in Java, kan in C++ een afgeleide klasse meerdere basisklassen hebben
- **Opletten voor dubbelzinnigheden**
  - als A en B een gemeenschappelijke basisklasse hebben, zijn de attributen van deze basisklasse 2x aanwezig in C
    => Welk attribuut wordt bedoeld? (geef expliciet aan met ::)
  - indien A en B dezelfde methode hebben, die niet overschreven is in C, dan is het bij het oproepen van deze methode op een object van de klasse C niet duidelijk welke van de 2 methoden bedoeld wordt (tenzij het expliciet met :: aangegeven wordt).

Inhoud

- public versus private overerving
- constructoren/destructor in afgeleide klasse
- overschrijven toekenningsoperator
- keyword protected
- multiple inheritance
- **polymorfisme en dynamic binding**
- virtuele destructor
- abstracte klassen

Polymorfisme

- bestaat net als in Java
- **opgelet**: **in C++ is een object geen verwijzing**
- polymorfisme is slechts mogelijk bij publieke overerving en bij het gebruik van **pointers/referenties**
- voorbeeld



```
class persoon {
   private:
      string naam;
   public:
      persoon(const string &nm="");
      void print(bool nl=false) const;
};
```

```
persoon::persoon(const string &nm) : naam(nm) {}
```

```
void persoon::print(bool nl) const {
   cout << naam;
   if (nl) cout << endl;
   else cout << " ";
}
```

```
class student : public persoon {
   private:
      string klascode;
   public:
      student(const string &nm="", const string &kl="");
      void print(bool nl=true) const;
};
```

```
student::student(const string &nm, const string &kl) :
            persoon(nm), klascode(kl) {}
```

```
void student::print(bool nl) const {
   persoon::print();
   cout << "zit in klas " << klascode;
   if (nl) cout<<endl;
   else cout<<" ";
}
```

```
class leerkracht : public persoon {
  private:
     string vakgroep;
  public:
     leerkracht(const string &nm="", const string &vg="");
     void print(bool nl=true) const;
};
```

```
leerkracht::leerkracht(const string &nm, const string &vg)
                     : persoon(nm),vakgroep(vg) {}
```

```
void leerkracht::print(bool nl) const {
   persoon::print();
   cout << "uit vakgroep " << vakgroep;
   if (nl) cout<<endl;
   else cout<<" ";
}
```

```
int main() {
    persoon p("Peter");
    p.print(true);
    student s("Silke","1Ba3");
    s.print();
    leerkracht l("Marc","EA20");
    l.print();

    p = l;
    p.print();
    l = p;
    l = (Leerkracht) p;
```

```
Peter
Silke zit in klas 1Ba3
Marc uit vakgroep EA20
```

Let op: alleen het persoon-deel wordt behouden

⇒ **hier is GEEN sprake van polymorfisme!**

```
Marc
```

Compileerfout!

```
persoon &rp = s;
persoon *pp = new leerkracht("Els","Talen");
unique_ptr<persoon> upp = make_unique<student>("An","Ma");
rp.print(); (*pp).print();  upp->print();
```

```
Silke  Els  An
```

er is GEEN dynamic binding

(lidfuncties van basisklasse worden gebruikt)

VbPolymorfisme.cpp ⇒ ook bij de <<-operator is er geen dynamic binding

Dynamic binding

- bestaat nochtans net als in Java
- dynamische binding werkt alleen voor
  - methodes die overschreven zijn
  - methodes die in de basisklasse **virtueel** zijn

```
virtual void print() const;
```

enkel vermelden bij declaratie lidfunctie

- Regels voor dynamic binding:
  - signatuur (= returntype, # en type parameters) van de methode in basisklasse en afgeleide klasse moeten gelijk zijn!
  - methode in basisklasse moet virtueel zijn.
  - methode in afgeleide klasse mag (maar moet niet) virtueel zijn.

- voorbeeld: **VbDynamicBinding[_bis].cpp**
- Virtuele lidfuncties hebben duidelijke voordelen

  $\Rightarrow$ Waarom niet alle lidfuncties virtueel?

  belangrijk **nadeel: tragere uitvoering** (er moet bij uitvoering onderzocht worden of de virtuele lidfunctie uit de basisklasse overschreven is)
- Dus indien virtuele lidfuncties niet nodig zijn: best niet gebruiken
  - In C++: keuze van programmeur
  - In Java: geen keuze (alle methodes zijn virtueel, tenzij `final` gebruikt)

Inhoud

- public versus private overerving
- constructoren/destructor in afgeleide klasse
- overschrijven toekenningsoperator
- keyword protected
- multiple inheritance
- polymorfisme en dynamic binding
- **virtuele destructor**
- abstracte klassen

Virtuele destructor

```
basis *pB = new afgeleid(); //polymorfisme
delete pB;
```

- Voorbeeld: **DemoDestructor1.cpp**

  Indien gewone destructor in klasse basis:

  enkel destructor van klasse basis wordt opgeroepen!!

- Oplossing: maak destructor virtueel

  Voorbeeld: **DemoDestructor2.cpp**

  Het wordt aanzien als een goede methodologie om een **destructor steeds virtueel** te maken!

  ⇨ **Doe dit op het examen!!!**

Inhoud

- public versus private overerving
- constructoren/destructor in afgeleide klasse
- overschrijven toekenningsoperator
- keyword protected
- multiple inheritance
- polymorfisme en dynamic binding
- virtuele destructor
- **abstracte klassen**

Abstracte klassen

- een klasse is abstract als
  - minstens één lidfunctie **puur virtueel** is
  - betekent: lidfunctie heeft geen body (=0)
  - syntax: `virtual void print() const = 0;`
- er kunnen geen objecten worden van aangemaakt

Inhoud

- public versus private overerving
- constructoren/destructor in afgeleide klasse
- overschrijven toekenningsoperator
- keyword protected
- multiple inheritance
- polymorfisme en dynamic binding
- virtuele destructor
- abstracte klassen