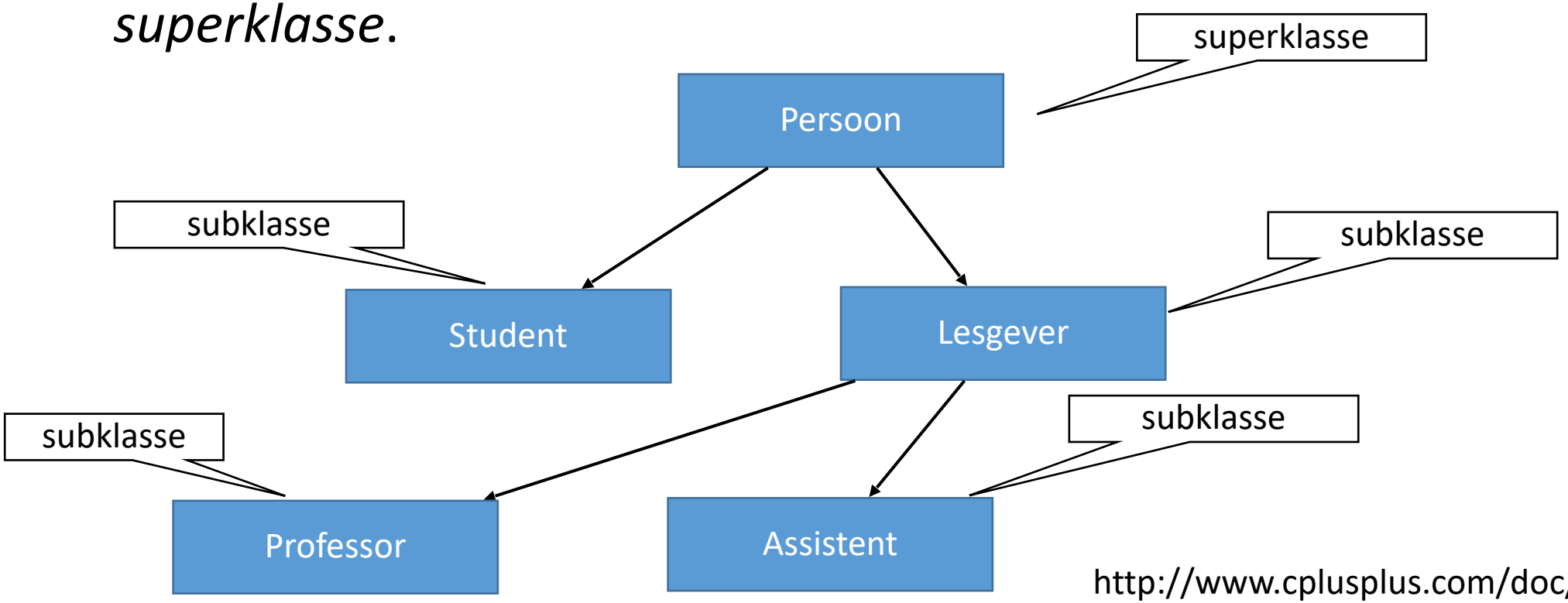
Universiteit Antwerpen

Inleiding Programmeren polymorfisme

Tom Hofkens

Specialisatie door overerving

- Klassen in C++ kunnen uitgebreid worden, wat resulteert in een nieuwe klasse die de eigenschappen van de basis klasse behoudt. We noemen de afgeleide klasse de *subklasse* en de basis klasse de *superklasse*.



http://www.cplusplus.com/doc/tutorial/inheritance/

Wat met constructors en destructors ?

- Persoon is superklasse, Student is subklasse
- Als een nieuwe Student object wordt gemaakt:
  - Eerst wordt constructor van Persoon uitgevoerd
  - Daarna constructor van Student

- Bij een destructor gebeurt net het omgekeerde:
  - Eerst de destructor van de subklasse (Student)
  - Daarna die van de superklasse

Universiteit Antwerpen

Overriding

- Bij inheritance nemen we automatisch alle functies van de superklasse over
- We kunnen functies echter ook herdefiniëren door ze opnieuw op te nemen; dit noemen we *overriding*

Voorbeeld:

Functie toString() in Persoon, Student, professor, …

Virtual functions

- Als B een subklasse is van A, dan kunnen we B gebruiken telkens wanneer een pointer of een reference naar A nodig is:
  - Voordeel: we kunnen klasse B gebruiken overal waar klasse A gebruik wordt
  - Voordeel: code moet niet gekopiëerd worden

- Echter: als we een method van A in B herdefiniëren, hoe weet de compiler welke van de twee uitgevoerd moet worden?
  - Via keyword “virtual”: als een method als “virtual” gedefinieerd is, wordt *at runtime* opgezocht welke versie uitgevoerd moet worden

Voorbeeld met Student

Virtual functions

- Als B een subklasse is van A, dan kunnen we B gebruiken telkens wanneer een pointer of een reference naar A nodig is:

```
class A {
public:
    void identify() {
        cout << "class A at memory location " << this << endl;
    }
};

class B:public A {
public:
    void identify() {
        cout << "class B at memory location " << this << endl;
    }
};

int main() {
    A a;
    B b;
    A& a2=b;
    a.identify();
    b.identify();
    a2.identify();
```

wat is de output?

Virtual functions

- Als B een subklasse is van A, dan kunnen we B gebruiken telkens wanneer een pointer of een reference naar A nodig is:

```
class A {
public:
    void identify() {
        cout << "class A at memory location " << this << endl;
    }
};

class B:public A {
public:
    void identify() {
        cout << "class B at memory location " << this << endl;
    }
};

int main() {
    A a;
    B b;
    A& a2=b;
    a.identify();
    b.identify();
    a2.identify();
```

```
class A at memory location 0x62fe9b
```

Virtual functions

- Als B een subklasse is van A, dan kunnen we B gebruiken telkens wanneer een pointer of een reference naar A nodig is:

```
class A {
public:
    void identify() {
        cout << "class A at memory location " << this << endl;
    }
};

class B:public A {
public:
    void identify() {
        cout << "class B at memory location " << this << endl;
    }
};

int main() {
    A a;
    B b;
    A& a2=b;
    a.identify();
    b.identify();
    a2.identify();
```

```
class A at memory location 0x62fe9b
class B at memory location 0x62fe9a
```

Virtual functions

- Als B een subklasse is van A, dan kunnen we B gebruiken telkens wanneer een pointer of een reference naar A nodig is:

```
class A {
public:
    void identify() {
        cout << "class A at memory location " << this << endl;
    }
};

class B:public A {
public:
    void identify() {
        cout << "class B at memory location " << this << endl;
    }
};

int main() {
    A a;
    B b;
    A& a2=b;
    a.identify();
    b.identify();
    a2.identify();
```

```
class A at memory location 0x62fe9b
class B at memory location 0x62fe9a
class A at memory location 0x62fe9a
```

Virtual functions

- Als B een subklasse is van A, dan kunnen we B gebruiken telkens wanneer een pointer of een reference naar A nodig is:

enkel in de basis klasse noodzakelijk!

in principe overbodig, maar vermeld voor leesbaarheid!

```
class A {
public:
    virtual void identify() {
        cout << "class A at me
    }

class B:public A {
public:
    virtual void identify() {
        cout << "class B at memory location " << this << endl;
    }
};

int main() {
    A a;
    B b;
    A& a2=b;

    a.identify();
    b.identify();
    a2.identify();
```

Wat is de output nu?

Virtual functions

- Als B een subklasse is van A, dan kunnen we B gebruiken telkens wanneer een pointer of een reference naar A nodig is:

```
class A {
public:
    virtual void identify() {
        cout << "class A at memory location " << this << endl;
    }
};

class B:public A {
public:
    virtual void identify() {
        cout << "class B at memory location " << this << endl;
    }
};

int main() {
    A a;
    B b;
    A& a2=b;

    a.identify();
    b.identify();
    a2.identify();
```

```
class A at memory location 0x62fe98
```

Virtual functions

- Als B een subklasse is van A, dan kunnen we B gebruiken telkens wanneer een pointer of een reference naar A nodig is:

```
class A {
public:
    virtual void identify() {
        cout << "class A at memory location " << this << endl;
    }
};

class B:public A {
public:
    virtual void identify() {
        cout << "class B at memory location " << this << endl;
    }
};

int main() {
    A a;
    B b;
    A& a2=b;

    a.identify();
    b.identify();
    a2.identify();
```

```
class A at memory location 0x62fe98
class B at memory location 0x62fe94
```

Virtual functions

- Als B een subklasse is van A, dan kunnen we B gebruiken telkens wanneer een pointer of een reference naar A nodig is:

```
class A {
public:
    virtual void identify() {
        cout << "class A at memory location " << this << endl;
    }
};

class B:public A {
public:
    virtual void identify() {
        cout << "class B at memory location " << this << endl;
    }
};

int main() {
    A a;
    B b;
    A& a2=b;

    a.identify();
    b.identify();
    a2.identify();
```

```
class A at memory location 0x62fe98
class B at memory location 0x62fe94
class B at memory location 0x62fe94
```

Virtual functions: veel gemaakte fouten

- string versie van PhdStudent is zelfde als student

Wat is er mis?

```
class PhdStudent: public Student {
private:
    Docent* promotor;
    string voornaam;
public:
    PhdStudent(Docent *promotor);

    Docent *getPromotor() const;

    void setPromotor(Docent *promotor);

    string toString() const;
};
```

voornaam zit nu 2 keer in geheugen!

toString blijft ongewijzigd, dus hier niet vermelden!

Virtual functions: override

- string versie van PhdStudent is NIET zelfde als student

```
class PhdStudent: public Student {
private:
    Docent* promotor;
public:
    PhdStudent(Docent *promotor);

    Docent *getPromotor() const;

    void setPromotor(Docent *promotor);

    string toString() const override;
};
```

expliciet vermelden dat de virtual function van een basisklasse overriden

zal error geven als er geen virtual staat in een basisklasse

Voorbeeld: figuren

- class Rectangle
- class Square

- Method area

Universiteit Antwerpen

Voorbeeld: figuren

- class Figure
- class Rectangle
- class Square

- Method area

Universiteit Antwerpen

Pure virtual

- De afgeleide klassen MOETEN dit implementeren.
- Contract AFDWINGEN!
- method = 0 zonder implementatie
- methode is pure virtual
- de klasse wordt abstract: je kan geen object aanmaken

Waarom niet alle functies virtual ?

```
class A {
    int a;
public:
    int getA() {return a;}
};

class B {
    int b;
public:
    virtual int getB() {return b;}
};

int main() {
    cout <<sizeof(A) << " " << sizeof(B) << endl;
```

virtual toevoegen betekent extra info bijhouden!

4 8

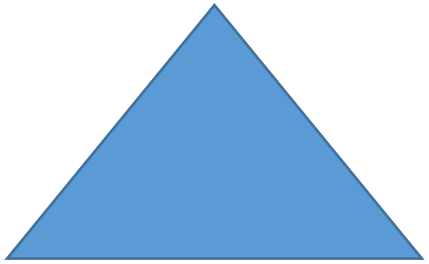
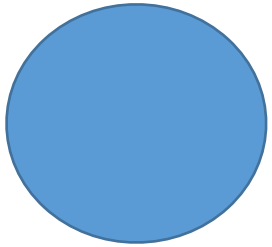
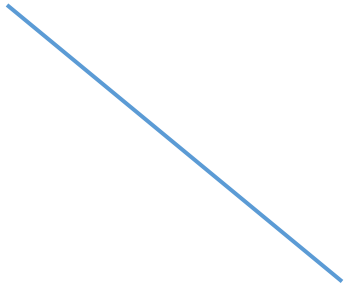
Polymorfisme

- Dankzij inheritance en virtual kunnen we polymorfisme realiseren

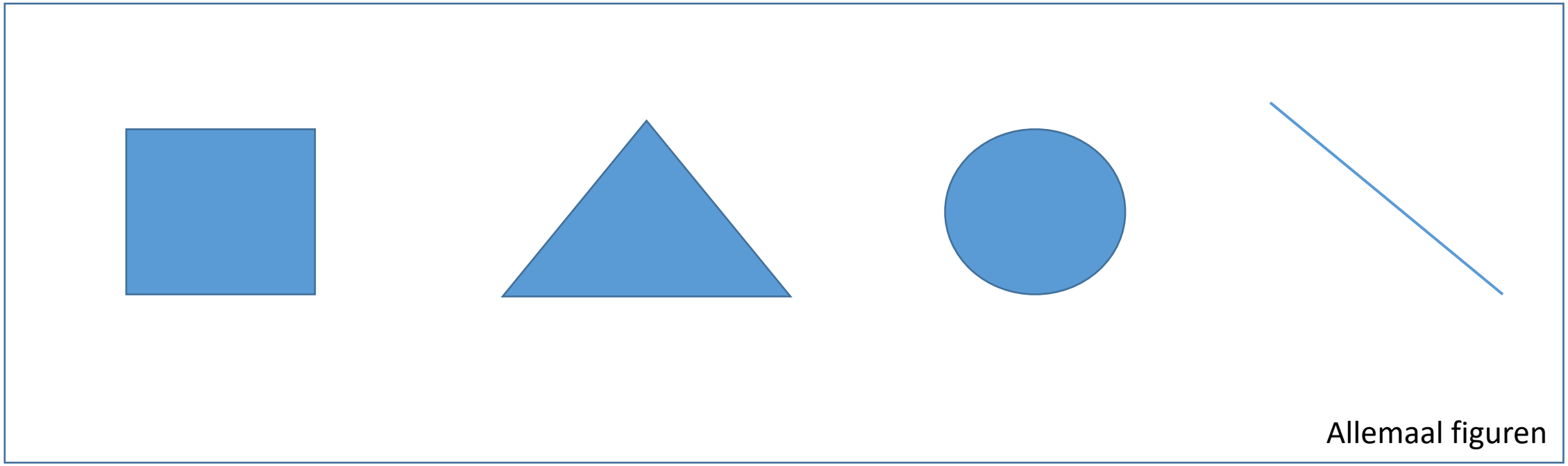
Polymorphisme: een vorm van abstractie waarbij objecten van verschillende klassen een gemeenschappelijke interface hebben en er voor het aanroepende object "gelijkaardig" zijn. Dit kan bijvoorbeeld door verschillende klassen van dezelfde basisklasse af te leiden.

Voorbeeld: vector van objecten

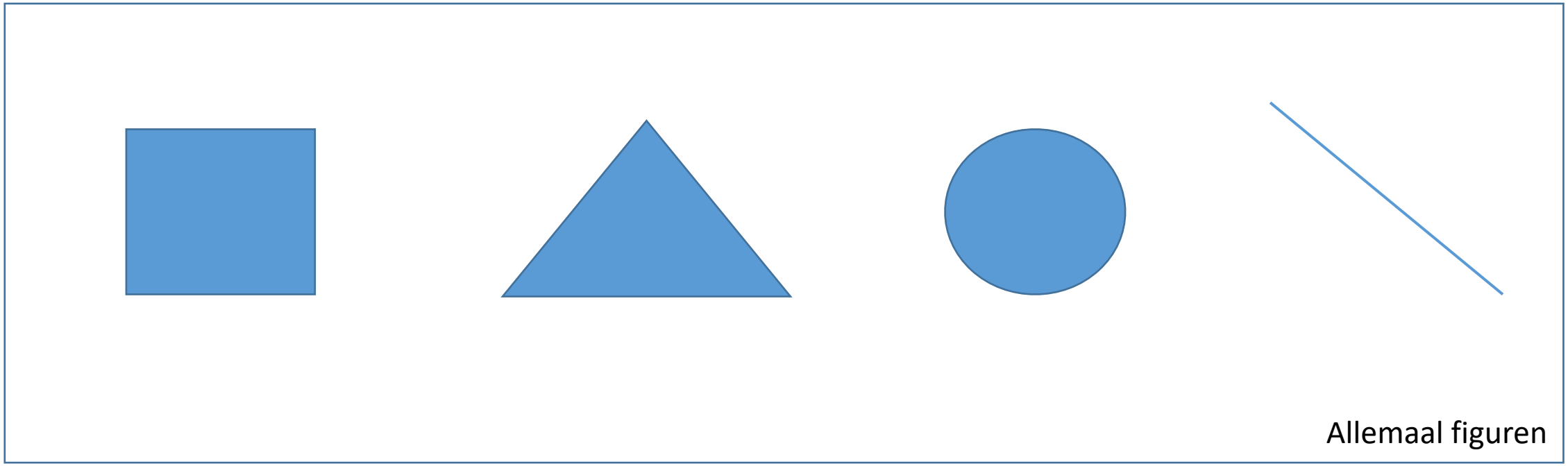
Polymorfisme

Polymorfisme



Polymorfisme

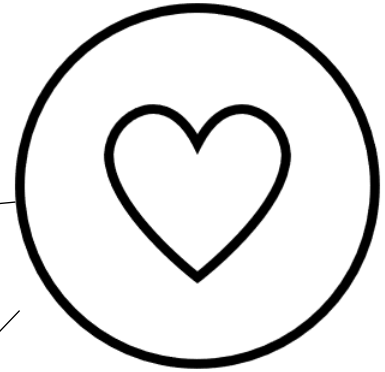


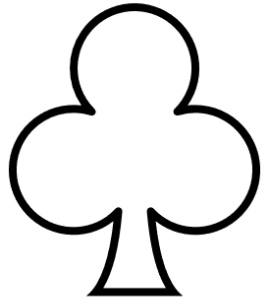
Tekening = lijst van figuren

zonder polymorfisme:
vector<vierkant>, vector<driehoek>, …

Polymorfisme

```
class Vierkant;
class Cirkel;
class Driehoek;
class Lijn;
class Hartje;
class Klaver;
class Figuur {
    vector<Vierkant> Lv;
    vector<Cirkel> Lc;
    vector<Driehoek> Ld;
    vector<Lijn> Ll;
    vector<Hartje> Lh;
    vector<Klaver> Lk;
    void draw_on(Drawing d) const {
        for (Vierkant v : Lv) {
            ...
        }
        for (Cirkel c : Lc) {
            ...
        }
        for (Hartje h : Lh) { ...}
    }   for (Klaver h : Lk) { ...}
```





...

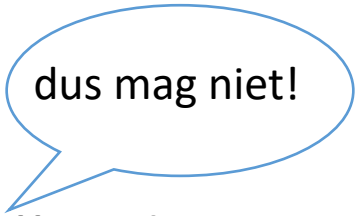
Polymorfisme

```
class Vierkant;
class Cirkel;
class Driehoek;
class Lijn;


class Figuur {
    vector<Vierkant> Lv;
    vector<Cirkel> Lc;
    vector<Driehoek> Ld;
    vector<Lijn> Ll;

    void draw_on(Drawing d) const {
        for (Vierkant v : Lv) {
            ...
        }
        for (Cirkel c : Lc) {
            ...
        }
        ...
    }
```

hoe dan wel?

Polymorfisme

- Waarom pointers?
  - vector<Figuur>: elk element kan een verschillende grootte hebben
  - vector<Figure*>: elk element heeft zelfde grootte



Polymorfisme

- Waarom virtual functies?
  - at runtime niet alleen juiste data, maar ook juiste methode (draw)!

Abstract Base Class

- ABC = klasse die we speciaal gemaakt hebben als superklasse
  - Nutteloos om die te instantiëren
  - Definiëert de interface, maar implementeert hem niet

```
class Figure {
public:
     Figure(int,int);
     virtual void draw(QPainter* p)=0;
     void move(int dx, int dy);
protected:
     int x,y;
};
```

pure virtual

Object Slicing

Voor polymorfisme hebben we nodig:

- Inheritance
- Virtual
- Pointers of referenties naar objecten

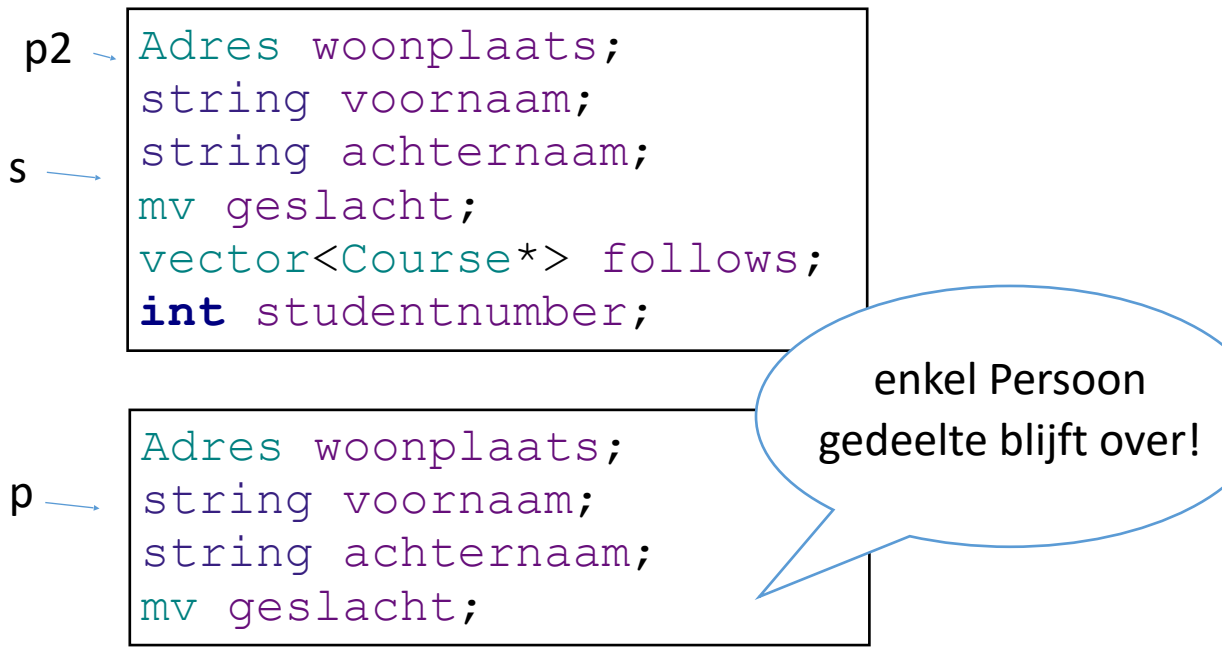
Echter: Indien we een **call by value** of **assignment** doen van de vorm b=a, waarbij b van het basistype is en a het afgeleide type dan gebeurt er *slicing* ; we kopiëren de delen die bij b horen naar a.

https://www.geeksforgeeks.org/object-slicing-in-c/

Let op! Slicing …

```
enum mv {man, vrouw};

class Persoon {
    Adres woonplaats;
    string voornaam;
    string achternaam;
    mv geslacht;
public:
    …
};

class Student: public Persoon {
    vector<Course*> follows;
    int studentnumber;
public:
    …
};
```

```
int main () {
    Student s;
    Persoon p;
    p = s;
    Persoon& p2=s;
}
```



Let op! Slicing …

- Dit gebeurt ook als we Persoon i.p.v. Persoon* als type gebruiken in een vector:

```
vector<Persoon> v;
v.push_back(john);
v.push_back(tom);
for (auto p:v) {
    cout << p.toString() << endl;
}
```

John Doe
Tom Hofkens

```
vector<Persoon*> vp;
vp.push_back(&john);
vp.push_back(&tom);
for (auto pp:vp) {
    cout << pp→toString() << endl
}
```

Student John Doe (12345)
Professor Tom Hofkens

Universiteit Antwerpen

Slicing leidt tot problemen

- Slicing is zelden wat we willen
  - Doet zich voor bij initialisatie
  - Bij assignment
  - Bij return

- Werk bij functies dus bij voorkeur met “call by const reference” of “call by pointer” en “return by reference” of “return by pointer”