

# Inleiding Programmeren

## Destructor, Inheritance, virtual

Tom Hofkens

# Object-georiënteerd programmeren: begrippen

- Encapsulatie: data en functies die deze data manipuleren met elkaar verbinden en beschermen van inferentie en foutief gebruik van buitenaf. *Data hiding* (beschermen van data door private te maken) is hierbij een belangrijke strategie.
- Invariant: integriteitseigenschap die steeds gegarandeerd moet worden; member functies moeten zodanig geïmplementeerd zijn dat ze invarianten bewaren (voor aanroep geldt invariant → na aanroep ook)
- Compositie: door samenstelling nieuwe, complexere datatypes maken. Dankzij encapsulatie kunnen we de complexiteit van samengestelde datatypes beheersen.

# Let op! Assignment en parameter passing

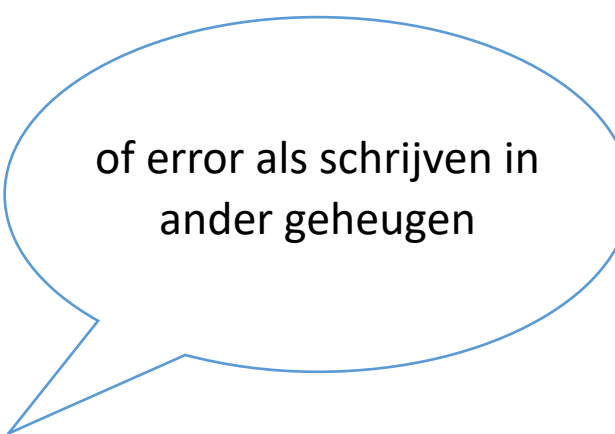
- Bij assignment (=), parameters doorgeven aan functies en return van waarden wordt **standaard** een kopie gemaakt:
  - Voor pointers betekent dit: een *shallow copy*; dat is, de *geheugenlocaties* van pointers worden gekopieerd, de locaties waarnaar ze verwijzen *niet*
- Let dus op bij het doorgeven van objecten by value en assignment van objecten indien die dynamische informatie bevatten.
  - Eigenlijk zou je voor zulke klassen de standaard *copy constructor* en *copy assignment* moeten *overloaden*.
    - Dit is exact wat de klassen in de STL doen (vector, map)

# Let op: destructor en assignment

```
void f(Stack f_stack) {  
    f_stack.push(11);  
    cout << "Laatste lijn van f" << endl;  
}  
  
int main() {  
    Stack main_stack1 = Stack();  
    for (int i = 0; i < 10; i++) {  
        main_stack1.push(i);  
    }  
    Stack main_stack2 = main_stack1;  
    cout << "Laatste lijn van main()" << endl;  
    return 0;  
}
```

# Let op: destructor en assignment

Laatste lijn van main()  
Destructor wordt uitgevoerd  
Weghalen 9  
Weghalen 8  
Weghalen 7  
Weghalen 6  
Weghalen 5  
Weghalen 4  
Weghalen 3  
Weghalen 2  
Weghalen 1  
Weghalen 0  
Destructor wordt uitgevoerd  
Weghalen 17039552  
Weghalen 17068128  
Weghalen 17039552  
Weghalen 17045056  
Weghalen 17068128



of error als schrijven in  
ander geheugen

# Let op: destructor en assignment

```
void f(Stack f_stack) {  
    f_stack.push(11);  
    cout << "Laatste lijn van f" << endl;  
}  
  
int main() {  
    Stack main_stack1 = Stack();  
    for (int i = 0; i < 10; i++) {  
        main_stack1.push(i);  
    }  
    Stack main_stack2 = main_stack1;  
    cout << "Laatste lijn van main()" << endl;  
    return 0;  
}
```

# Let op: destructor & call by value

- By een call by value wordt er een nieuw object aangemaakt dat op het einde terug “destruct” wordt
  - Stack voorbeeld

```
void f(Stack f_stack) {  
    f_stack.push(11);  
    cout << "Laatste lijn van f" << endl;  
}  
  
int main() {  
    Stack main_stack1 = Stack();  
    for (int i = 0; i < 10; i++) {  
        main_stack1.push(i);  
    }  
    f(main_stack1);  
    cout << "Laatste lijn van main()" << endl;  
    return 0;  
}
```

# Let op: destructor & call by value

Laatste lijn van f  
Destructor wordt uitgevoerd  
Weghalen 11  
Weghalen 9  
Weghalen 8  
Weghalen 7  
Weghalen 6  
Weghalen 5  
Weghalen 4  
Weghalen 3  
Weghalen 2  
Weghalen 1  
Weghalen 0  
Laatste lijn van main()  
Destructor wordt uitgevoerd  
Weghalen 16515264  
Weghalen 16543840  
Weghalen 16515264  
Weghalen 16503760

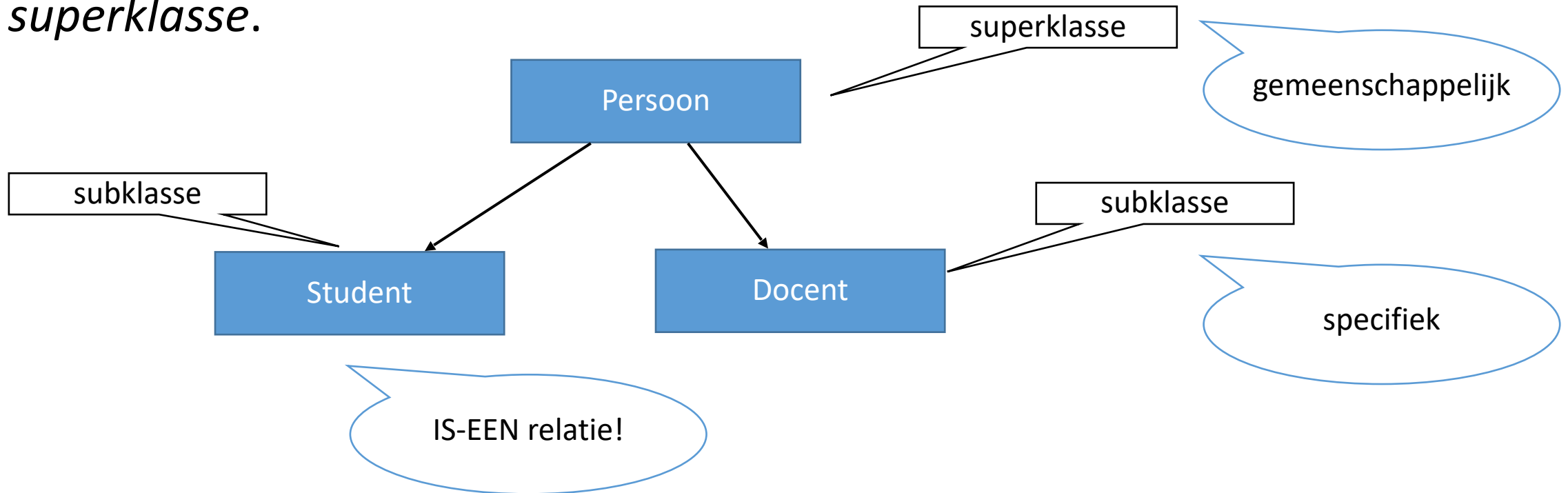


# Overerving

- Specialisatie d.m.v. overerving
- Sub IS Super
- Protected access specifier
- Public/private/protected inheritance
- Wat met constructors en destructors

# 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*.



# Overerving

```
class Super {  
    int x = 1;  
public:  
    int y = 2;  
    void set_x(int xx) {x = xx;}  
    int get_x() const {return x; }  
  
};
```

```
class Sub: public Super {  
public:  
    void print() {  
        // cout << y << "\t" << x << endl;    // werkt niet : x is private  
        cout << y << "\t" << get_x() << endl;    // werkt wel : y en get_x() zijn public  
    }  
};
```

# Specialisatie door overerving

```
enum mv {man, vrouw, X};
```

```
class Persoon {  
    Adres woonplaats;  
    string voornaam;  
    string achternaam;  
    mv geslacht;  
public:  
    Persoon();  
    string to_string() const;  
  
    const Adres &getWoonplaats() const;  
    void setWoonplaats(const Adres &woonplaats);  
    const string &getVoornaam() const;  
    void setVoornaam(const string &voornaam);  
    const string &getAchternaam() const;  
    void setAchternaam(const string &achternaam);  
    mv getGeslacht() const;  
    void setGeslacht(mv geslacht);  
};
```

```
class Student: public Persoon {  
    vector<Course*> follows;  
    int studentnumber;  
public:  
    Student() {}  
  
    const vector<Course *> &getFollows() const;  
    void addCourse(Course* c);  
    void print() const;  
};
```

# Specialisatie door overerving

- Subklasse heeft toegang tot alle publieke variabelen en methods van de superklasse en kan variabelen en methods toevoegen

```
class Persoon {  
    Adres woonplaats;  
    string voornaam;  
    string achternaam;  
    mv geslacht;  
public:  
    Persoon();  
    string to_string() const;  
    ...  
};
```

```
class Student: public Persoon {  
    vector<Course*> follows;  
    int studentnumber;  
public:  
    Student() {}  
  
    const vector<Course *> &getFollows() const;  
    void addCourse(Course* c);  
    void print() const;  
};  
  
void Student::print() const {  
    cout << to_string() << endl;  
}
```

student is  
persoon met nog wat  
extra info en  
functionaliteit

# Overerving: Sub IS Super

- Overerving gaat verder dan “code hergebruiken”
- We kunnen de subklasse overal gebruiken waar de superklasse verwacht wordt

Hier kunnen we enkel methods van  
Persoon gebruiken

```
void printNaam(Persoon& p) {
    cout << p.getVoornaam() << " " << p.getAchternaam() << endl;
}

int main() {
    Persoon p;
    p.setAchternaam("Vermeulen"); p.setVoornaam("Jefke"); p.setGeslacht(man);
    p.setWoonplaats(Adres("Tramezantlei", 122, 2900, "Schoten"));
    printNaam(p);

    Student s;
    s.setAchternaam("Vermeulen"); s.setVoornaam("An"); s.setGeslacht(vrouw);
    s.setWoonplaats(Adres("Tramezantlei", 122, 2900, "Schoten"));
    printNaam(s);
}
```

kan enkel  
bij ref en ptr  
parameters!

Wordt uitgevoerd met het gedeelte dat van  
Persoon komt

14 werkt want Student IS Persoon

# Overerving: Sub-klasse als Super

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

```
Persoon p;  
Student s;  
Persoon& p2 = s; // werkt!  
Persoon* p3 = &s; // werkt!  
Student* s2 = &p; // error!
```

## Voorbeeld 2 : Sub IS Super

- Overerving gaat verder dan “code hergebruiken”
- We kunnen de subklasse overal gebruiken waar de superklasse verwacht wordt
- We kunnen een pointer naar de subklasse overal gebruiken waar een pointer naar de superklasse verwacht wordt

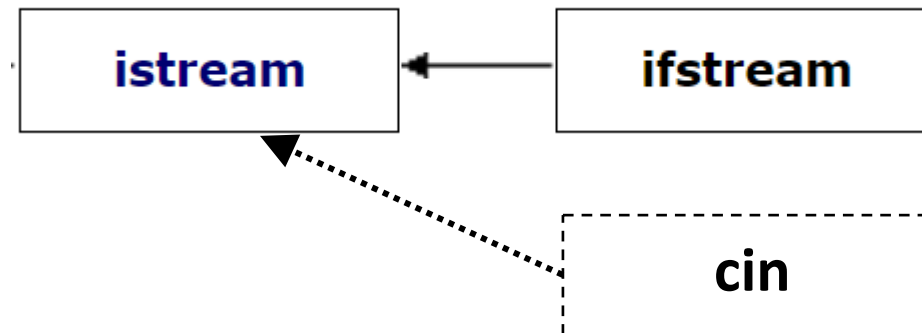
 wat is alternatief?

```
vector<Persoon*> vrijwilligers;  
vrijwilligers.push_back(&s);  
vrijwilligers.push_back(&p);  
  
for (auto x:vrijwilligers) {  
    cout << x->to_string() << endl;  
}
```



# Voorbeeld: Overerving

- In de STL is een input file stream afgeleid van een input stream
- Console input is een object van type istream



- Elk object van klasse ifstream kan dus gebruikt worden waar een (pointer of referentie naar) istream vereist is

# Voorbeeld: Overerving

```
vector<int> getNumbers(istream& is) {  
    vector<int> v;  
    int i;  
    do {  
        is >> i;  
        v.push_back(i);  
    } while (i!=0);  
    return v;  
}
```

```
int main() {  
    ifstream f("test.txt");  
    if (!f.is_open()) {  
        cout << "Opening file failed\n";  
        return -1;  
    }  
    vector<int> v=getNumbers(f);  
    f.close();
```

je zou ook cin  
kunnen doorgeven

# 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

# Enkele voorbeelden (1/4)

```
class Persoon {  
    Adres woonplaats;  
    string voornaam;  
    string achternaam;  
    mv geslacht;  
public:  
    Persoon();  
    Persoon(const string& voornaam,  
            const string& achternaam,  
            mv, const Adres&);  
    ...  
}
```

dan gebruikt die  
default constructor

```
class Student: public Persoon {  
    vector<Course*> follows;  
    int studentnumber;  
public:  
    Student() {}  
    Student(int studentnr,  
            const string& voornaam,  
            const string& achternaam, mv, const Adres&);  
    Student(int studentnr);  
    ...  
}
```

init met  
setters

# Enkele voorbeelden (2/4)

```
class Persoon {  
    Adres woonplaats;  
    string voornaam;  
    string achternaam;  
    mv geslacht;  
public:  
    Persoon();  
    Persoon(const string& voornaam,  
            const string& achternaam,  
            mv, const Adres&);  
    ...  
}
```

```
class Student: public Persoon {  
    vector<Course*> follows;  
    int studentnumber;  
public:  
    Student() {}  
    Student(int snr,  
            const string& vn,  
            const string& an,  
            mv g, const Adres& a) : Persoon(vn, an, g, a) {  
        studentnumber = snr;  
    }  
    Student(int studentnr);  
    ...  
}
```

je kan ook  
constructor expliciet  
aanroepen

bij voorkeur zo!

# Enkele voorbeelden (3/4)

```
class Persoon {
    Adres woonplaats;
    string voornaam;
    string achternaam;
    mv geslacht;
public:
    Persoon(const string& voornaam,
            const string& achternaam,
            mv, const Adres&);
    ...
}
```

wat veel logischer is  
(wat is een persoon zonder  
naam?)


```
class Student: public Persoon {
    vector<Course*> follows;
    int studentnumber;
public:
    Student() {}
    Student(int snr,
            const string& vn,
            const string& an,
            mv g, const Adres& a) : Persoon(vn,an,g,a) {
        studentnumber = snr;
    }
    Student(int studentnr);
    ...
}
```

welke  
constructors werken  
wel/niet?

hier moet je het dus  
zo doen!

# Enkele voorbeelden (4/4)

```
class Persoon {  
    ...  
public:  
    Persoon();  
    ...  
}  
  
class Student: public Persoon {  
    ...  
public:  
    Student();  
}  
  
class PhdStudent: public Student {  
    ...  
public:  
    PhdStudent();  
}
```



Als een nieuw PhdStudent object wordt aangemaakt, dan:

1. Constructor van Persoon
2. Constructor van Student
3. Constructor van PhdStudent

# Overerving: ctors en dtors

```
class A {
public:
    int x;
    A(): x(0) { cout << "A::A()\t" ; }
    A(int xx): x(xx) { cout << "A::A(int)\t" ; }
    ~A() { cout << "A::~~A\t"; }
};

class B: public A {
public:
    int y;
    B(): y(0) { cout << "B::B()\t" ; }
    B(int xx): A(xx), y(0) { cout << "B::B(int)\t" ; }
    B(int xx, int yy): A(xx), y(yy) { cout << "B::B(int,int)\t" ; }
    ~B() { cout << "B::~~B\t"; }
};
```

```
int main() {
    cout << "A o1: \t\t";
    A o1;
    cout << endl << "A o2(3): \t";
    A o2(3);
    cout << endl << "B o3: \t\t";
    B o3;
    cout << endl << "B o4(4): \t";
    B o4(4);
    cout << endl << "B o5(4,7): \t";
    B o5(4,7);
    cout << endl << "=====" << endl;

    return 0;
}
```



# Overerving: ctors en dtors

```
class A {
public:
    int x;
    A(): x(0) { cout << "A::A()\t" ; }
    A(int xx): x(xx) { cout << "A::A(int)\t" ; }
    ~A() { cout << "A::~~A\t"; }
};

class B:public A {
public:
    int y;
    B(): y(0) { cout << "B::B()\t" ; }
    B(int xx): A(xx), y(0) { cout << "B::B(int)\t" ; }
    B(int xx, int yy): A(xx), y(yy) { cout << "B::B(int,int)\t" ; }
    ~B() { cout << "B::~~B\t"; }
};
```

A o1:      A::A()

```
int main() {
    cout << "A o1: \t\t";
    A o1;
    cout << endl << "A o2(3): \t";
    A o2(3);
    cout << endl << "B o3: \t\t";
    B o3;
    cout << endl << "B o4(4): \t";
    B o4(4);
    cout << endl << "B o5(4,7): \t";
    B o5(4,7);
    cout << endl << "=====" << endl;

    return 0;
}
```

# Overerving: ctors en dtors

```
class A {
public:
    int x;
    A(): x(0) { cout << "A::A()\t" ; }
    A(int xx): x(xx) { cout << "A::A(int)\t" ; }
    ~A() { cout << "A::~~A\t"; }
};

class B:public A {
public:
    int y;
    B(): y(0) { cout << "B::B()\t" ; }
    B(int xx): A(xx), y(0) { cout << "B::B(int)\t" ; }
    B(int xx, int yy): A(xx), y(yy) { cout << "B::B(int,int)\t" ; }
    ~B() { cout << "B::~~B\t"; }
};
```

A o1:      A::A()  
A o2(3):    A::A(int)

```
int main() {
    cout << "A o1: \t\t";
    A o1;
    cout << endl << "A o2(3): \t";
    A o2(3);
    cout << endl << "B o3: \t\t";
    B o3;
    cout << endl << "B o4(4): \t";
    B o4(4);
    cout << endl << "B o5(4,7): \t";
    B o5(4,7);
    cout << endl << "=====" << endl;

    return 0;
}
```

# Overerving: ctors en dtors

```
class A {
public:
    int x;
    A(): x(0) { cout << "A::A()\t" ; }
    A(int xx): x(xx) { cout << "A::A(int)\t" ; }
    ~A() { cout << "A::~~A\t"; }
};

class B:public A {
public:
    int y;
    B(): y(0) { cout << "B::B()\t" ; }
    B(int xx): A(xx), y(0) { cout << "B::B(int)\t" ; }
    B(int xx, int yy): A(xx), y(yy) { cout << "B::B(int,int)\t" ; }
    ~B() { cout << "B::~~B\t"; }
};
```

```
A o1:      A::A()
A o2(3):   A::A(int)
B o3:      A::A() B::B()
```

```
int main() {
    cout << "A o1: \t\t";
    A o1;
    cout << endl << "A o2(3): \t";
    A o2(3);
    cout << endl << "B o3: \t\t";
    B o3;
    cout << endl << "B o4(4): \t";
    B o4(4);
    cout << endl << "B o5(4,7): \t";
    B o5(4,7);
    cout << endl << "=====" << endl;

    return 0;
}
```

# Overerving: ctors en dtors

```
class A {
public:
    int x;
    A(): x(0) { cout << "A::A()\t" ; }
    A(int xx): x(xx) { cout << "A::A(int)\t" ; }
    ~A() { cout << "A::~~A\t"; }
};

class B:public A {
public:
    int y;
    B(): y(0) { cout << "B::B()\t" ; }
    B(int xx): A(xx), y(0) { cout << "B::B(int)\t" ; }
    B(int xx, int yy): A(xx), y(yy) { cout << "B::B(int,int)\t" ; }
    ~B() { cout << "B::~~B\t"; }
};
```

```
A o1:      A::A()
A o2(3):   A::A(int)
B o3:      A::A() B::B()
B o4(4):   A::A(int)  B::B(int)
```

```
int main() {
    cout << "A o1: \t\t";
    A o1;
    cout << endl << "A o2(3): \t";
    A o2(3);
    cout << endl << "B o3: \t\t";
    B o3;
    cout << endl << "B o4(4): \t";
    B o4(4);
    cout << endl << "B o5(4,7): \t";
    B o5(4,7);
    cout << endl << "=====" << endl;

    return 0;
}
```

# Overerving: ctors en dtors

```
class A {
public:
    int x;
    A(): x(0) { cout << "A::A()\t" ; }
    A(int xx): x(xx) { cout << "A::A(int)\t" ; }
    ~A() { cout << "A::~~A\t"; }
};

class B:public A {
public:
    int y;
    B(): y(0) { cout << "B::B()\t" ; }
    B(int xx): A(xx), y(0) { cout << "B::B(int)\t" ; }
    B(int xx, int yy): A(xx), y(yy) { cout << "B::B(int,int)\t" ; }
    ~B() { cout << "B::~~B\t"; }
};
```

```
A o1:      A::A()
A o2(3):    A::A(int)
B o3:       A::A() B::B()
B o4(4):     A::A(int)   B::B(int)
B o5(4,7):   A::A(int)   B::B(int,int)
```

```
int main() {
    cout << "A o1: \t\t";
    A o1;
    cout << endl << "A o2(3): \t";
    A o2(3);
    cout << endl << "B o3: \t\t";
    B o3;
    cout << endl << "B o4(4): \t";
    B o4(4);
    cout << endl << "B o5(4,7): \t";
    B o5(4,7);
    cout << endl << "=====" << endl;

    return 0;
}
```

# Overerving: ctors en dtors

```
class A {
public:
    int x;
    A(): x(0) { cout << "A::A()\t" ; }
    A(int xx): x(xx) { cout << "A::A(int)\t" ; }
    ~A() { cout << "A::~~A\t"; }
};

class B:public A {
public:
    int y;
    B(): y(0) { cout << "B::B()\t" ; }
    B(int xx): A(xx), y(0) { cout << "B::B(int)\t" ; }
    B(int xx, int yy): A(xx), y(yy) { cout << "B::B(int,int)\t" ; }
    ~B() { cout << "B::~~B\t"; }
};
```

```
A o1:      A::A()
A o2(3):    A::A(int)
B o3:       A::A() B::B()
B o4(4):     A::A(int)   B::B(int)
B o5(4,7):   A::A(int)   B::B(int,int)
=====
```

```
int main() {
    cout << "A o1: \t\t";
    A o1;
    cout << endl << "A o2(3): \t";
    A o2(3);
    cout << endl << "B o3: \t\t";
    B o3;
    cout << endl << "B o4(4): \t";
    B o4(4);
    cout << endl << "B o5(4,7): \t";
    B o5(4,7);
    cout << endl << "=====" << endl;

    return 0;
}
```

# Overerving: ctors en dtors

```
class A {
public:
    int x;
    A(): x(0) { cout << "A::A()\t" ; }
    A(int xx): x(xx) { cout << "A::A(int)\t" ; }
    ~A() { cout << "A::~~A\t"; }
};

class B:public A {
public:
    int y;
    B(): y(0) { cout << "B::B()\t" ; }
    B(int xx): A(xx), y(0) { cout << "B::B(int)\t" ; }
    B(int xx, int yy): A(xx), y(yy) { cout << "B::B(int,int)\t" ; }
    ~B() { cout << "B::~~B\t"; }
};
```

```
A o1:      A::A()
A o2(3):    A::A(int)
B o3:       A::A() B::B()
B o4(4):    A::A(int)  B::B(int)
B o5(4,7):  A::A(int)  B::B(int,int)
=====
B::~~B  A::~~A
```

```
int main() {
    cout << "A o1: \t\t";
    A o1;
    cout << endl << "A o2(3): \t";
    A o2(3);
    cout << endl << "B o3: \t\t";
    B o3;
    cout << endl << "B o4(4): \t";
    B o4(4);
    cout << endl << "B o5(4,7): \t";
    B o5(4,7);
    cout << endl << "=====" << endl;

    return 0;
}
```

# Overerving: ctors en dtors

```
class A {
public:
    int x;
    A(): x(0) { cout << "A::A()\t" ; }
    A(int xx): x(xx) { cout << "A::A(int)\t" ; }
    ~A() { cout << "A::~~A\t"; }
};

class B:public A {
public:
    int y;
    B(): y(0) { cout << "B::B()\t" ; }
    B(int xx): A(xx), y(0) { cout << "B::B(int)\t" ; }
    B(int xx, int yy): A(xx), y(yy) { cout << "B::B(int,int)\t" ; }
    ~B() { cout << "B::~~B\t"; }
};
```

```
A o1:      A::A()
A o2(3):    A::A(int)
B o3:       A::A() B::B()
B o4(4):    A::A(int)  B::B(int)
B o5(4,7):  A::A(int)  B::B(int,int)
=====
B::~~B  A::~~A  B::~~B  A::~~A
```

```
int main() {
    cout << "A o1: \t\t";
    A o1;
    cout << endl << "A o2(3): \t";
    A o2(3);
    cout << endl << "B o3: \t\t";
    B o3;
    cout << endl << "B o4(4): \t";
    B o4(4);
    cout << endl << "B o5(4,7): \t";
    B o5(4,7);
    cout << endl << "=====" << endl;

    return 0;
}
```



# Overerving: ctors en dtors

```
class A {
public:
    int x;
    A(): x(0) { cout << "A::A()\t" ; }
    A(int xx): x(xx) { cout << "A::A(int)\t" ; }
    ~A() { cout << "A::~~A\t"; }
};

class B:public A {
public:
    int y;
    B(): y(0) { cout << "B::B()\t" ; }
    B(int xx): A(xx), y(0) { cout << "B::B(int)\t" ; }
    B(int xx, int yy): A(xx), y(yy) { cout << "B::B(int,int)\t" ; }
    ~B() { cout << "B::~~B\t"; }
};
```

```
int main() {
    cout << "A o1: \t\t";
    A o1;
    cout << endl << "A o2(3): \t";
    A o2(3);
    cout << endl << "B o3: \t\t";
    B o3;
    cout << endl << "B o4(4): \t";
    B o4(4);
    cout << endl << "B o5(4,7): \t";
    B o5(4,7);
    cout << endl << "=====" << endl;

    return 0;
}
```

```
A o1:      A::A()
A o2(3):    A::A(int)
B o3:      A::A() B::B()
B o4(4):    A::A(int) B::B(int)
B o5(4,7):  A::A(int) B::B(int,int)
=====
B::~~B A::~~A B::~~B A::~~A B::~~B A::~~A ,
```

# Overerving: ctors en dtors

```
class A {
public:
    int x;
    A(): x(0) { cout << "A::A()\t" ; }
    A(int xx): x(xx) { cout << "A::A(int)\t" ; }
    ~A() { cout << "A::~~A\t"; }
};

class B:public A {
public:
    int y;
    B(): y(0) { cout << "B::B()\t" ; }
    B(int xx): A(xx), y(0) { cout << "B::B(int)\t" ; }
    B(int xx, int yy): A(xx), y(yy) { cout << "B::B(int,int)\t" ; }
    ~B() { cout << "B::~~B\t"; }
};
```

```
int main() {
    cout << "A o1: \t\t";
    A o1;
    cout << endl << "A o2(3): \t";
    A o2(3);
    cout << endl << "B o3: \t\t";
    B o3;
    cout << endl << "B o4(4): \t";
    B o4(4);
    cout << endl << "B o5(4,7): \t";
    B o5(4,7);
    cout << endl << "=====" << endl;

    return 0;
}
```

```
A o1:      A::A()
A o2(3):    A::A(int)
B o3:      A::A() B::B()
B o4(4):    A::A(int) B::B(int)
B o5(4,7):  A::A(int) B::B(int,int)
=====
B::~~B A::~~A B::~~B A::~~A B::~~B A::~~A A::~~A
```

# Overerving: ctors en dtors

```
class A {
public:
    int x;
    A(): x(0) { cout << "A::A()\t" ; }
    A(int xx): x(xx) { cout << "A::A(int)\t" ; }
    ~A() { cout << "A::~A\t"; }
};

class B:public A {
public:
    int y;
    B(): y(0) { cout << "B::B()\t" ; }
    B(int xx): A(xx), y(0) { cout << "B::B(int)\t" ; }
    B(int xx, int yy): A(xx), y(yy) { cout << "B::B(int,int)\t" ; }
    ~B() { cout << "B::~B\t"; }
};
```

```
int main() {
    cout << "A o1: \t\t";
    A o1;
    cout << endl << "A o2(3): \t";
    A o2(3);
    cout << endl << "B o3: \t\t";
    B o3;
    cout << endl << "B o4(4): \t";
    B o4(4);
    cout << endl << "B o5(4,7): \t";
    B o5(4,7);
    cout << endl << "=====" << endl;

    return 0;
}
```

```
A o1:      A::A()
A o2(3):   A::A(int)
B o3:      A::A() B::B()
B o4(4):   A::A(int) B::B(int)
B o5(4,7): A::A(int) B::B(int,int)
=====
B::~B A::~A B::~B A::~A B::~B A::~A A::~A A::~A
```

# Overerving constructors

```
1 // constructors and derived classes
2 #include <iostream>
3 using namespace std;
4
5 class Mother {
6 public:
7     Mother ()
8     { cout << "Mother: no parameters\n"; }
9     Mother (int a)
10    { cout << "Mother: int parameter\n"; }
11 };
12
13 class Daughter : public Mother {
14 public:
15     Daughter (int a)
16     { cout << "Daughter: int parameter\n\n"; }
17 };
18
19 class Son : public Mother {
20 public:
21     Son (int a) : Mother (a)
22     { cout << "Son: int parameter\n\n"; }
23 };
24
25 int main () {
26     Daughter kelly(0);
27     Son bud(0);
28
29     return 0;
30 }
```

Wat is de  
output?

# Overriding constructors

```
1 // constructors and derived classes
2 #include <iostream>
3 using namespace std;
4
5 class Mother {
6 public:
7     Mother ()
8     { cout << "Mother: no parameters\n"; }
9     Mother (int a)
10    { cout << "Mother: int parameter\n"; }
11 };
12
13 class Daughter : public Mother {
14 public:
15     Daughter (int a)
16     { cout << "Daughter: int parameter\n\n"; }
17 };
18
19 class Son : public Mother {
20 public:
21     Son (int a) : Mother (a)
22     { cout << "Son: int parameter\n\n"; }
23 };
24
25 int main () {
26     Daughter kelly(0);
27     Son bud(0);
28
29     return 0;
30 }
```

Mother: no parameters

# Overriding constructors

```
1 // constructors and derived classes
2 #include <iostream>
3 using namespace std;
4
5 class Mother {
6 public:
7     Mother ()
8     { cout << "Mother: no parameters\n"; }
9     Mother (int a)
10    { cout << "Mother: int parameter\n"; }
11 };
12
13 class Daughter : public Mother {
14 public:
15     Daughter (int a)
16     { cout << "Daughter: int parameter\n\n"; }
17 };
18
19 class Son : public Mother {
20 public:
21     Son (int a) : Mother (a)
22     { cout << "Son: int parameter\n\n"; }
23 };
24
25 int main () {
26     Daughter kelly(0);
27     Son bud(0);
28
29     return 0;
30 }
```

Mother: no parameters  
Daughter: int parameter

# Overriding constructors

```
1 // constructors and derived classes
2 #include <iostream>
3 using namespace std;
4
5 class Mother {
6 public:
7     Mother ()
8     { cout << "Mother: no parameters\n"; }
9     Mother (int a)
10    { cout << "Mother: int parameter\n"; }
11 };
12
13 class Daughter : public Mother {
14 public:
15     Daughter (int a)
16     { cout << "Daughter: int parameter\n\n"; }
17 };
18
19 class Son : public Mother {
20 public:
21     Son (int a) : Mother (a)
22     { cout << "Son: int parameter\n\n"; }
23 };
24
25 int main () {
26     Daughter kelly(0);
27     Son bud(0);
28
29     return 0;
30 }
```

Mother: no parameters  
Daughter: int parameter  
  
Mother: int parameter

# Overriding constructors

```
1 // constructors and derived classes
2 #include <iostream>
3 using namespace std;
4
5 class Mother {
6 public:
7     Mother ()
8     { cout << "Mother: no parameters\n"; }
9     Mother (int a)
10    { cout << "Mother: int parameter\n"; }
11 };
12
13 class Daughter : public Mother {
14 public:
15     Daughter (int a)
16     { cout << "Daughter: int parameter\n\n"; }
17 };
18
19 class Son : public Mother {
20 public:
21     Son (int a) : Mother (a)
22     { cout << "Son: int parameter\n\n"; }
23 };
24
25 int main () {
26     Daughter kelly(0);
27     Son bud(0);
28
29     return 0;
30 }
```

Mother: no parameters  
Daughter: int parameter

Mother: int parameter  
Son: int parameter



# Overerving constructors

```
1 // constructors and derived classes
2 #include <iostream>
3 using namespace std;
4
5 class Mother {
6 public:
7     Mother ()
8     { cout << "Mother: no parameters\n"; }
9     Mother (int a)
10    { cout << "Mother: int parameter\n"; }
11 };
12
13 class Daughter : public Mother {
14 public:
15     Daughter (int a)
16     { cout << "Daughter: int parameter\n\n"; }
17 };
18
19 class Son : public Mother {
20 public:
21     Son (int a) : Mother (a)
22     { cout << "Son: int parameter\n\n"; }
23 };
24
25 int main () {
26     Daughter kelly(0);
27     Son bud(0);
28
29     return 0;
30 }
```

Mother: no parameters  
Daughter: int parameter

Mother: int parameter  
Son: int parameter

Geen expliciete call naar constructor  
superklasse => default constructor  
zonder parameters

Expliciete call naar constructor Super  
=> de constructor met de juiste  
signatuur wordt aangeroepen

# Access Specifiers

- Een klasse bepaalt zelf hoe en welke van haar members toegankelijk zijn van buiten de klasse
  - Dit houdt ook in: subklasse
- Voorbeeld: klasse gemaakt door Persoon A, gebruikt door Persoon B
  - Persoon A is verantwoordelijk voor:
    - bewaken invariant
    - Constructie / destructie (initialisatie, news, deletes)
  - Daarom: Persoon B heeft niet tot alle onderdelen toegang

blijf van mijn private parts!

volgend jaar: friends with benefits

want het zou een idioot  
kunnen zijn!

invariant ongeldig

memory leaks

# Public, private, protected inheritance

- Een klasse kan zelf beslissen hoe ze geërfdde methods en member variabelen deelt met de buitenwereld en subklassen
- Ze kan echter niet méér toegang geven

```
class Sub: public Super
```

public van Super  
wordt  
**public** in Sub

```
class Sub: private Super
```

public van Super  
wordt  
**private** in Sub

# Waarom Private inheritance?

- Afschermen van methods in de basisklasse
- Voorbeeld: Stack afgeleid van ArrayList
  - We willen niet dat de gebruikers van de Stack de methodes van ArrayList rechtstreeks gaan gebruiken



# Protected

- Er bestaat ook een access modifier *tussen* public en private: **protected** members die protected zijn, kunnen wel door subklassen worden gebruikt, maar niet buiten de klassen en haar subklassen

Access	public	protected	private
members of the same class	yes	yes	yes
members of derived class	yes	yes	no
not members	yes	no	no

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

- Op die manier kunnen we zaken beschikbaar stellen aan de subklasse, maar niet daarbuiten

# Protected: voorbeeld

- Hiervoor: Student erfde over van Persoon
  - voornaam, achternaam, adres waren private
  - In Student moesten we getVoornaam, setVoornaam etc. gebruiken
- Mochten we voornaam, achternaam, adres protected gemaakt hebben, dan was dat niet nodig geweest maar was de data toch beschermd voor buitenaf.

# Public, private, protected inheritance

- Een klasse kan zelf beslissen hoe ze geërfdde methods en member variabelen deelt met de buitenwereld en subklassen
- Ze kan echter niet méér toegang geven

```
class Sub: public Super
```

alles blijft

```
class Sub: protected Super
```

public wordt protected

```
class Sub: private Super
```

public en protected wordt private

# Public, private, protected inheritance

	class Sub: public Super	class Sub: protected Super	class Sub: private Super
public in Super	public in Sub	protected in Sub	private in Sub
protected in Super	protected in Sub	protected in Sub	private in Sub
private in Super	private in Sub	private in Sub	private in Sub

- Dit heeft enkel impact op hoe van *buiten* Sub, geërfde members benaderd kunnen worden; dit heeft geen enkele invloed op de toegangsrechten van members van Sub op door haar geërfde members



# Overerving

```
class Super {
    int x = 1;
public:
    int y = 2;
    void set_x(int xx) {x = xx;}
    int get_x() const {return x; }

};

class Sub: public Super {
public:
    void print() {
        // cout << y << "\t" << x << endl;    // werkt niet : x is private
        cout << y << "\t" << get_x() << endl;    // werkt wel : y en get_x() zijn public
    }
};
```

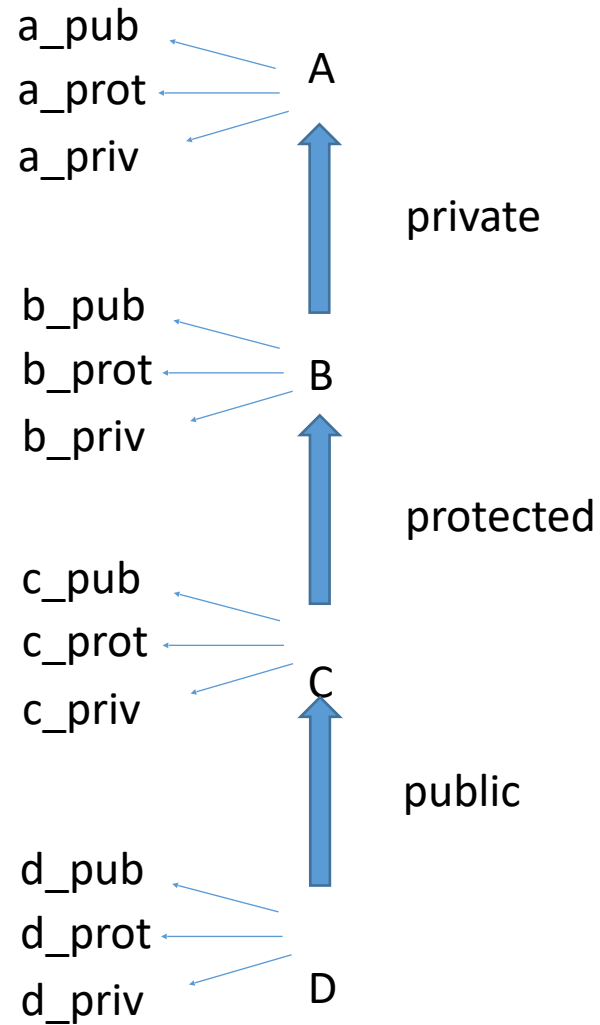
# Overerving

```
class Super {
protected:
    int x = 1;
public:
    int y = 2;
    void set_x(int xx) {x = xx;}
    int get_x() const {return x; }

};

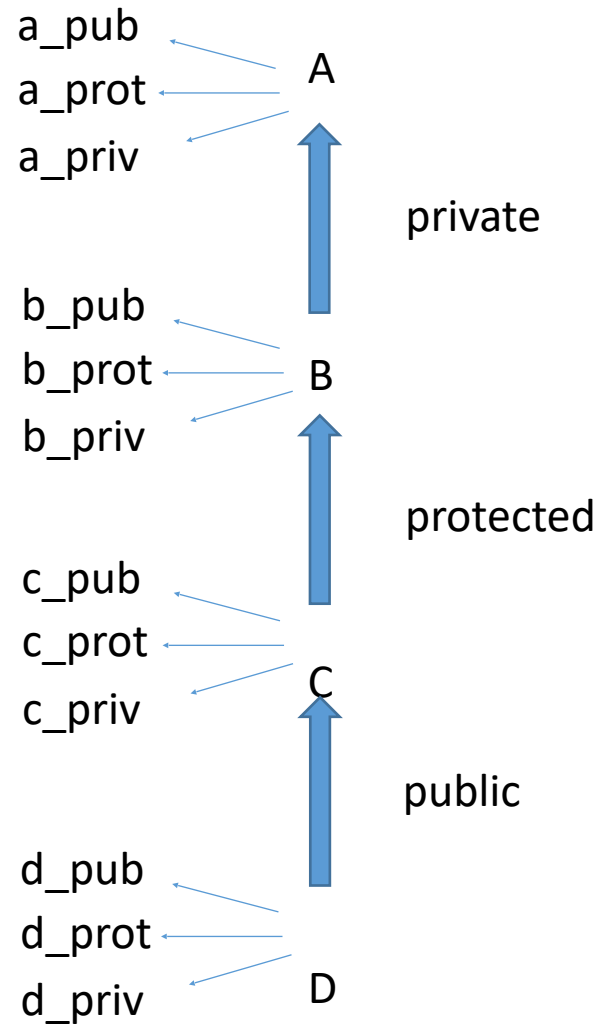
class Sub: public Super {
public:
    void print() {
        cout << y << "\t" << x << endl;          // werkt wel : x is protected
        cout << y << "\t" << get_x() << endl;      // werkt wel : y en get_x() zijn public
    }
};
```

# Voorbeeld



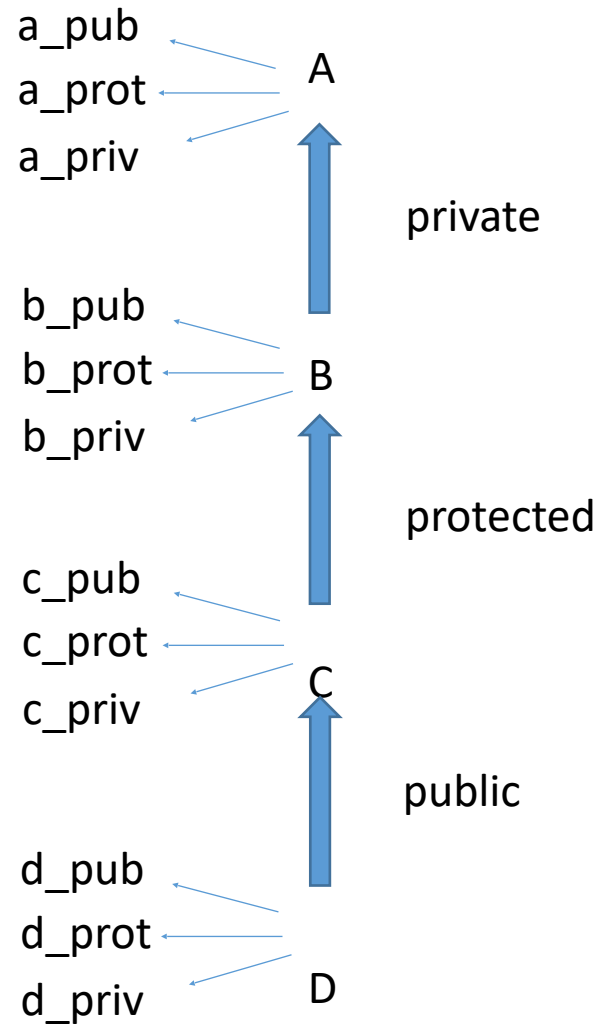
Class	public	protected	private
A			
B			
C			
D			

# Voorbeeld



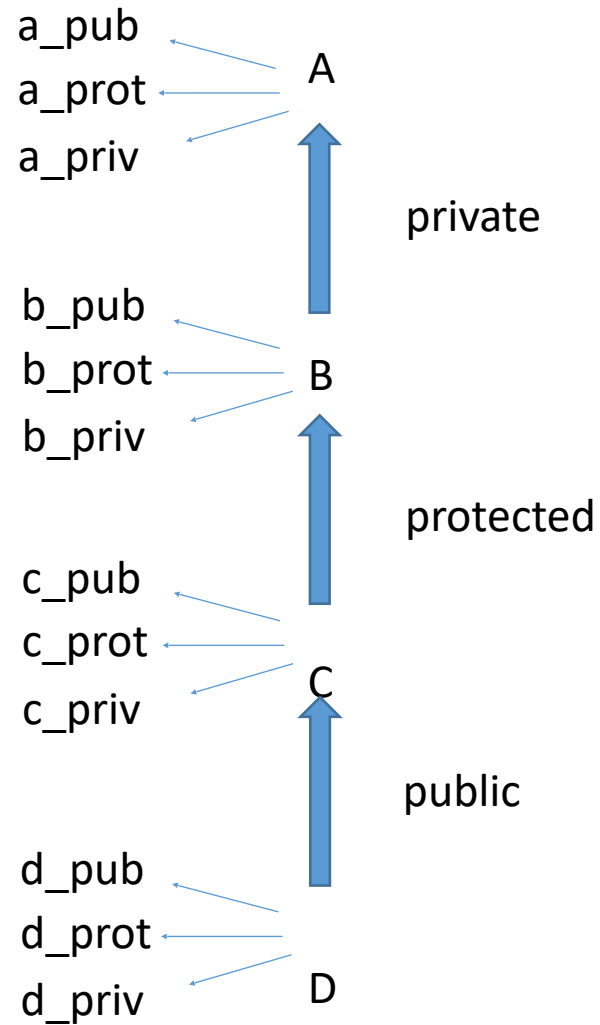
Class	public	protected	private
A	a_pub	a_prot	a_priv
B			
C			
D			

# Voorbeeld



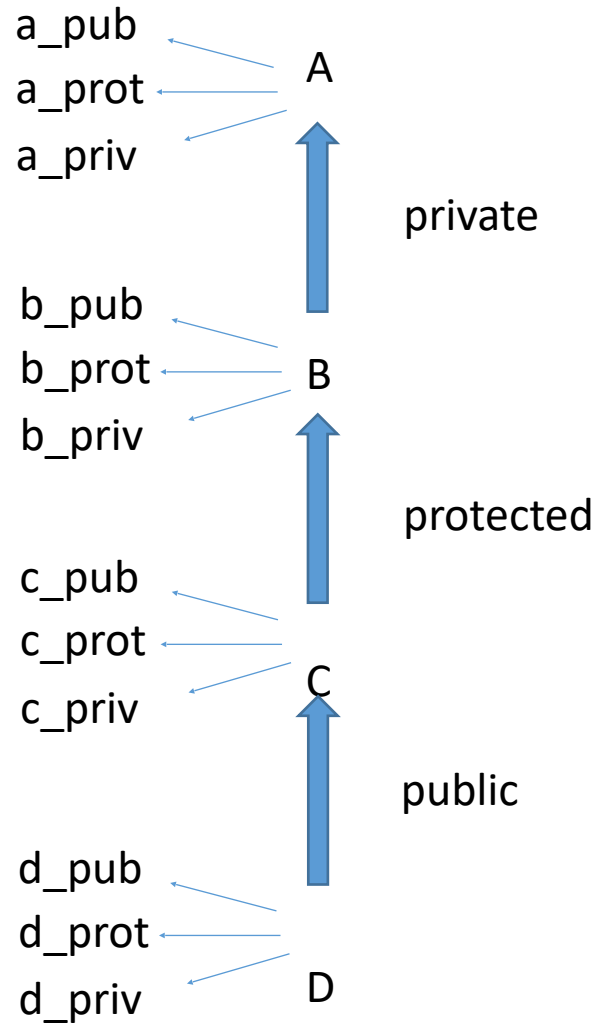
Class	public	protected	private
A	a_pub	a_prot	a_priv
B	b_pub	b_prot	a_* b_priv
C			
D			

# Voorbeeld



Class	public	protected	private
A	a_pub	a_prot	a_priv
B	b_pub	b_prot	a_* b_priv
C	c_pub	b_pub-prot c_prot	a_* b_priv c_priv
D			

# Voorbeeld



Class	public	protected	private
A	<code>a_pub</code>	<code>a_prot</code>	<code>a_priv</code>
B	<code>b_pub</code>	<code>b_prot</code>	<code>a_*</code> <code>b_priv</code>
C	<code>c_pub</code>	<code>b_pub-prot</code> <code>c_prot</code>	<code>a_*</code> <code>b_priv</code> <code>c_priv</code>
D	<code>c_pub</code> <code>d_pub</code>	<code>b_pub-prot</code> <code>c_prot</code> <code>d_prot</code>	<code>a_*</code> <code>b_priv</code> <code>c_priv</code> <code>d_priv</code>