

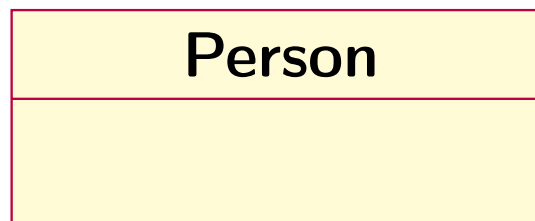
Programmering i C++  
EDAF30  
Arv

## Innehåll

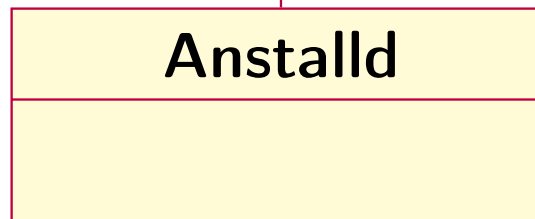
- Härledda klasser
- Konstruktörer och destruktörer vid arv
- Tillgänglighet
- Polymorfism och dynamisk bindning
- Abstrakta klasser
- Multipel ärvning
- Jämförelser med Java

Relationen *är* (generalisering)

Basklass (superklass)



Härledd klass (subklass)



## Basklassens deklaration

```
#include <string>
class Person
{
public:
    Person() {}
    const std::string& heter() const {return namn;}
    void andra_namn(const std::string& n);
    void skriv_info() const;
private:
    std::string namn;
};
```

## Basklassens medlemsfunktioner

```
#include <iostream>
using namespace std;

void Person::andra_namn(const std::string& n) {
    namn = n;
}

void Person::skriv_info() const {
    cout << "Namn : " << namn << endl;
}
```

## Deklaration av härledd klass

```
class Anstalld : public Person
{
public:
    Anstalld() : lon(0) {}
    long tjanar() const {return lon;}
    void andra_lon(long ny_lon);
    void skriv_info() const;
private:
    long lon;
};
```

## Härledda klassens medlemsfunktioner

```
void Anstalld::andra_lon(long ny_lon) {
    lon = ny_lon;
}

// Modifierad skriv_info
// (döljer basklassens skriv_info)
void Anstalld::skriv_info() const {
    Person::skriv_info(); // basklassens info
    cout << "Lön : " << lon << endl;
}
```

## Skapande och hanterande av objekt

```
Person p;  
Anstalld a;  
  
p.andra_namn("Kalle Olsson");  
p.skriv_info();  
a.andra_namn("Nisse Nilsson");  
a.andra_lon(23000);  
long l = a.tjanar();  
a.skriv_info();
```



## Utskrift

Namn: Kalle Olsson

Namn: Nisse Nilsson

Lön: 23000

## Funktionsöverlagring fungerar ej mellan olika nivåer i arvshierarkin

```
class C1 {
public:
    void f(int); // C1::f
};

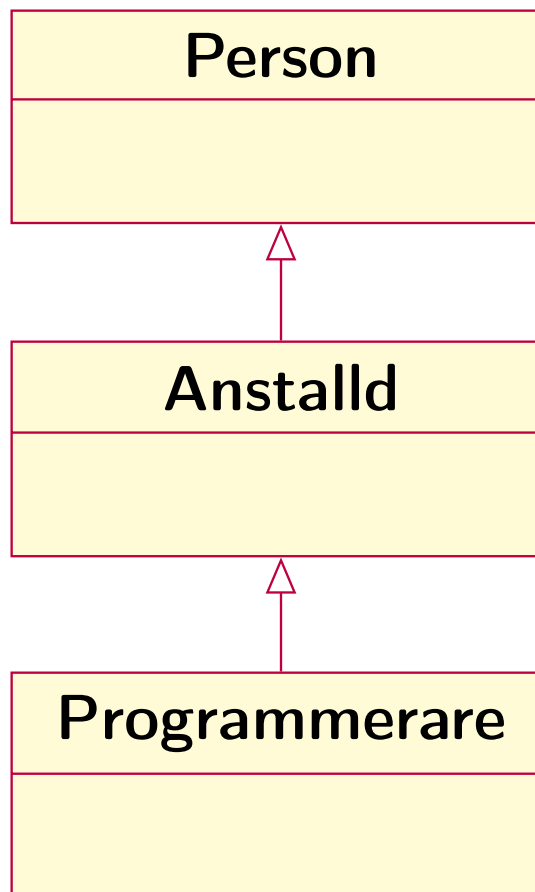
class C2 : public C1 {
public:
    void f(); // C2::f (döljer C1::f)
};

//...
C1 a; C2 b;
a.f(5); // Ok
b.f(); // Ok
b.f(2) // Fel! C1::f är dold!
```

# Härledda klasser

## Grafisk representation

### Arv i flera nivåer



Härledd klass till  
Anstalld

## Arv i flera nivåer – Klassdefinition

```
class Programmerare : public Anstalld {
public:
    Programmerare() {fav[0]='\0';}
    // Programmerare() {}
    const std::string avlas_favorit() const;
    // const std::string& avlas_favorit() const;
    void andra_favorit(const std::string& f);
    void skriv_info() const;
private:
    char fav[20]; // Favoritspråk
                // Typen vald för att passa
                // i senare exempel
    // std::string fav; // Favoritspråk
};
```

## Definitioner av medlemsfunktioner

```
const std::string Programmerare::avlas_favorit() const {  
    return fav;  
} // Kan inte returnera referens här!  
//const std::string& Programmerare::avlas_favorit() const {  
//    return fav;  
//}
```

```
void Programmerare::andra_favorit(const std::string& f) {  
    strcpy(fav, f.c_str());  
}  
//void Programmerare::andra_favorit(const std::string& f) {  
//    fav = f;  
//}
```

```
void Programmerare::skriv_info () const {  
    Anstalld::skriv_info();  
    cout << "Favorit: " << fav << endl;  
}
```

## Användning

```
Programmerare pr, *pp;  
  
pr.andra_namn("Lasse Olsson");  
pr.andra_lon(22000);  
pr.andra_favorit("Visual Basic");  
pr.skriv_info(); // Anstalld::skriv_info()  
  
pp = new Programmerare;  
pp->andra_namn("Anna Nilsson");  
pp->andra_lon(25000);  
pp->andra_favorit("C++");  
pp->skriv_info();
```

## Utskrift

Namn : Lasse Olsson

Lön : 22000

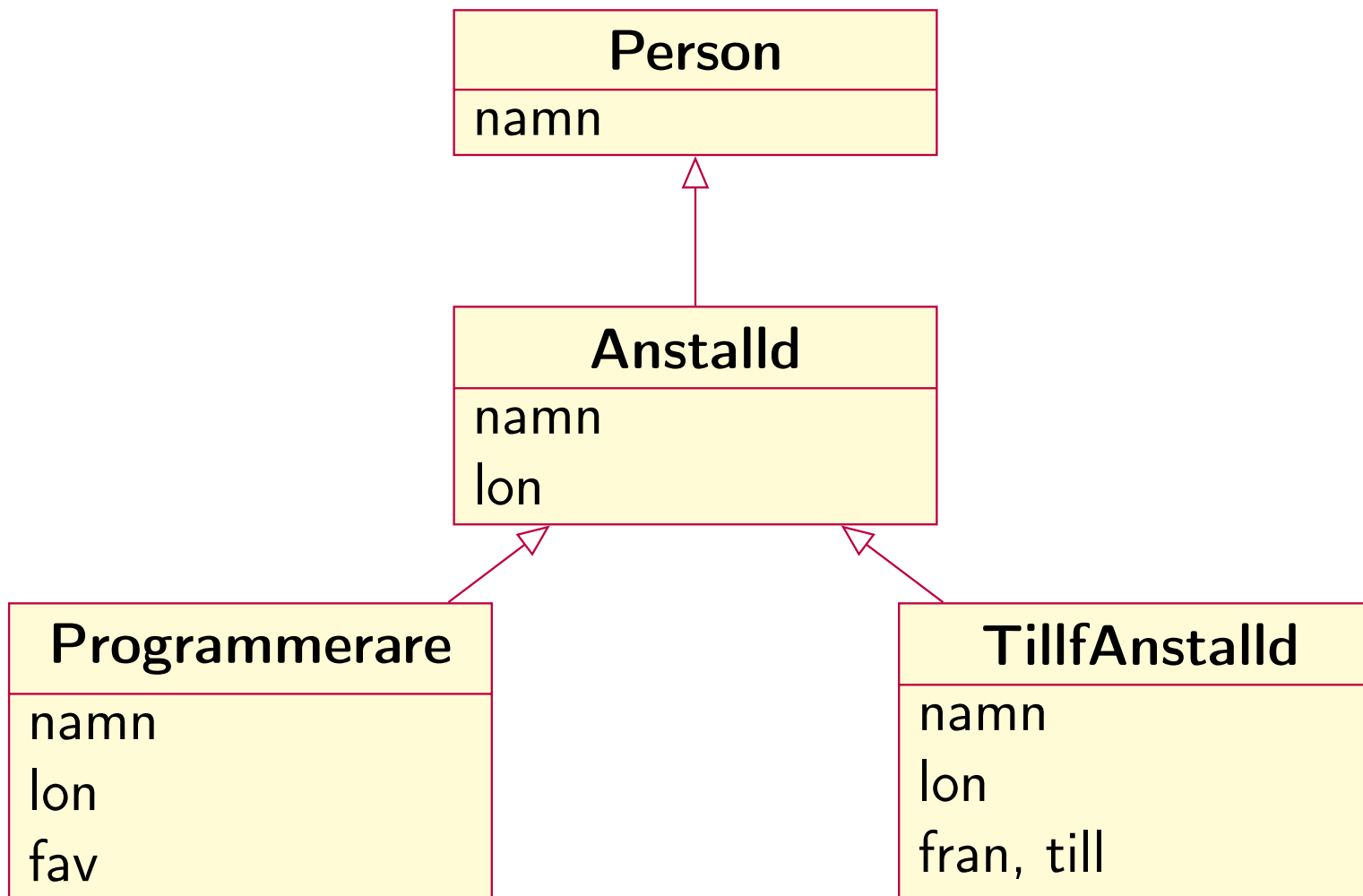
Favorit: Visual Basic

Namn : Anna Nilsson

Lön : 25000

Favorit: C++

### Parallella subklasser





## Arv i flera nivåer – Klassdefinition

```
class Datum {  
public:  
    int ar, man, dag;  
};
```

```
class TillfAnstalld : public Anstalld {  
public:  
    Datum fran, till;  
};
```

## Användning

```
// ...
Programmerare pr, *pp = &pr;
TillfAnstalld ti;
Anstalld *pa;

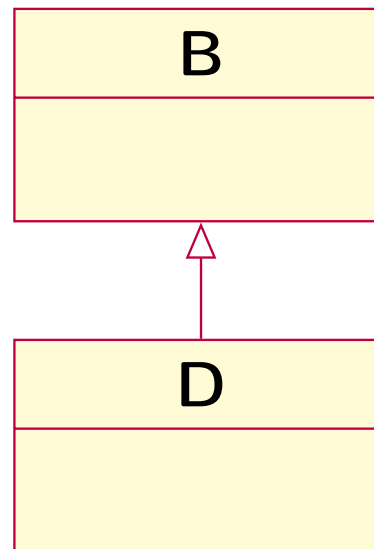
pa = pp; // OK (basklass <-- subklass)
pp = (Programmerare *)pa; // Kan gå bra
pa = &ti; // OK (basklass <-- subklass)
pp = (Programmerare *)pa; // Kan gå illa
pp->andra_favorit("Java"); // Inte bra!
pp->skriv_info(); // Ser ut att funka?
ti.fran.ar = 1999;
pp->skriv_info(); // Vart tog Java vägen?
```

## Initieringsordning i en konstruktor (för härledd klass)

- 1 Basklassens konstruktor anropas
- 2 Datamedlemmar (i den egna klassen) initieras
- 3 Funktionskroppen i konstruktorn exekveras

Explicit anrop av basklassens konstruktor i initieringslistan

```
D::D(param.) : B(param.), ... {...}
```



## Överlagring av konstruktörer

```
class Person
{
public:
    Person() {}
    Person(const std::string& n) : namn(n) {}
};

class Anstalld : public Person
{
public:
    Anstalld() : lon(0) {} // Överlagring!
    Anstalld(const std::string& n, long l) :
        Person(n), lon(l) {}; // Överlagring!
};
```

## Exekveringsordning i en destruktör

- 1 Funktionskroppen i destruktorn exekveras
- 2 Basklassens destruktör anropas

## De olika nivåerna av tillgänglighet

```
class C {  
public:  
// Medlemmar åtkomliga från godt. funktioner  
protected:  
// Medlemmar åtkomliga från medlemsfunktioner  
// i basklassen eller härledda klasser  
private:  
// Medlemmar åtkomliga endast från  
// basklassens medlemsfunktioner  
};
```

## Tillgänglighet vid arv

```
class D : public B { // Publikt arv  
// ...  
};
```

```
class D : protected B { // Skyddat arv  
// ...  
};
```

```
class D : private B { // Privat arv  
// ...  
};
```

## Tillgänglighet vid arv

	Tillgänglighet i B	Tillgänglighet via D
Publikt arv	public protected private	public protected private
Skyddat arv	public protected private	protected protected private
Privat arv	public protected private	private private private

Tillgängligheten inuti D påverkas *inte* av typen av arv



## Polymorfism (mångformighet)

Överlagring

*Statisk bindning*

Generiska programenheter (templates)

*Statisk bindning*

Virtuella funktioner

*Dynamisk bindning*

*Statisk bindning:* Betydelsen hos en viss konstruktion avgörs vid *kompilering*

*Dynamisk bindning:* Betydelsen hos en viss konstruktion avgörs vid *exekvering*

# Polymorfism och dynamisk bindning

- Dynamisk bindning av en funktion (virtuell funktion) markeras med nyckelordet `virtual` före funktionsnamnet i funktionsdeklarationen

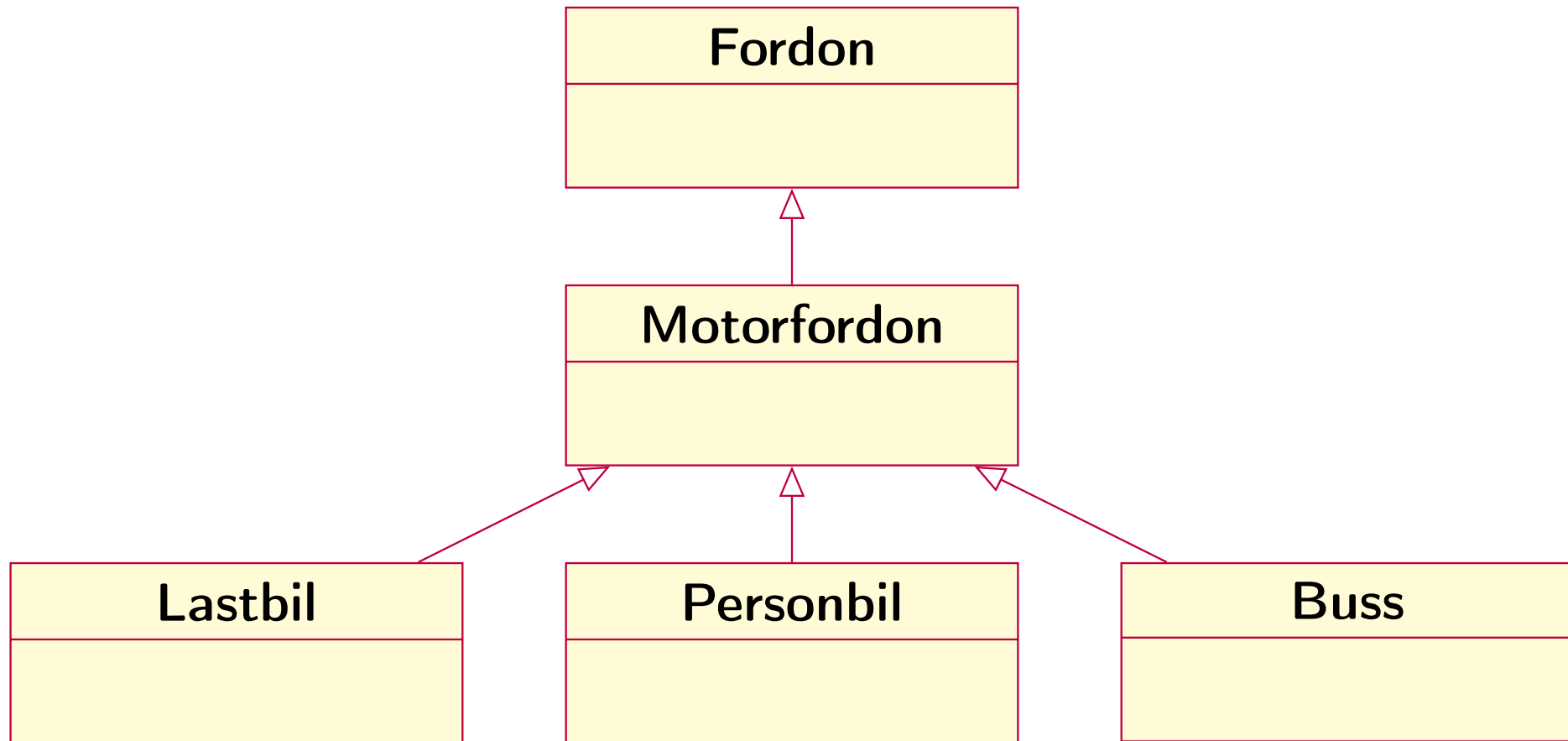
```
class Fordon {  
public:  
    virtual void ge_info();  
};
```

- Klass med minst en virtuell funktion kallas för en *virtuell klass*

# Polymorfism och dynamisk bindning

## Grafisk representation

### Klasser av fordon



## Klasser av fordon

```
class Fordon {
public:
    virtual void ge_info();
};

class Motorfordon : public Fordon {
public:
    Motorfordon(const std::string& nr) : reg_num(nr) {}
    const std::string& nummer() const {return reg_num;}
    void ge_info();
protected:
    std::string reg_num;
};
```

## Klasser av fordon

```
class Personbil : public Motorfordon {
public:
    Personbil(const std::string&nr, int n) :
        Motorfordon(nr), platser(n) {}
    void ge_info();
protected:
    int platser;
};
```

## Klasser av fordon

```
class Lastbil : public Motorfordon {  
public:  
    Lastbil(const std::string& nr, int l) :  
        Motorfordon(nr), max_last(l) {}  
    void ge_info();  
protected:  
    int max_last;  
};
```

## Klasser av fordon

```
class Buss : public Motorfordon {  
public:  
    Buss(const std::string& nr, int n, bool l=false)  
        : Motorfordon(nr), passag(n), luftk(l) {}  
    void ge_info();  
protected:  
    int passag;  
    bool luftk;  
};
```

## Def. av ge\_info för olika klasser

```
void Fordon::ge_info() {
    cout << "Ett fordon" << endl;
}

void Motorfordon::ge_info() {
    cout << "Ett motorfordon" << endl;
    cout << "Reg nr: " << reg_num << endl;
}

void Personbil::ge_info() {
    Motorfordon::ge_info();
    cout << "En personbil" << endl;
    cout << platser << " platser" << endl;
}
```



## Def. av ge\_info för olika klasser

```
void Lastbil::ge_info() {  
    Motorfordon::ge_info();  
    cout << "En lastbil" << endl;  
    cout << "Max last (kg): " << max_last <<endl;  
}
```

```
void Buss::ge_info() {  
    Motorfordon::ge_info();  
    cout << "En buss" << endl;  
    cout << passag << " passagerare" <<endl;  
    if (luftk)  
        cout << "Har luftkonditionering" <<endl;  
}
```

## Skapande av objekt

```
Personbil pb("XYZ 555", 5);
Lastbil lb("ZZZ 222", 10000);
Buss b("CPP 999", 60);

Fordon *fp;
pb.ge_info();
fp = &lb;
fp->ge_info();
fp = new Personbil("AAA 333", 4);
fp->ge_info();
fp = &b;           // Oops, glömde delete fp
fp->ge_info();
```

## Utskrifter

Ett motorfordon

Reg nr: XYZ 555

En personbil

5 platser

Ett motorfordon

Reg nr: ZZZ 222

En lastbil

Max last (kg): 10000

Ett motorfordon

Reg nr: AAA 333

En personbil

4 platser

Ett motorfordon

Reg nr: CPP 999

En buss

60 passagerare

## Avgörande av dynamisk typ med typeid

```
// fp är statistiskt av typ Fordon*  
if (typeid(*fp) == typeid(Personbil))  
    cout << "Fordonet är en personbil";  
else if (typeid(*fp) == typeid>Lastbil))  
    cout << "Fordonet är en lastbil";  
else if (typeid(*fp) == typeid(Buss))  
    cout << "Fordonet är en buss";
```

## Säker dynamisk typkonvertering med `dynamic_cast`

```
// fp är även i detta exempel statistiskt av typ Fordon*
Motorfordon *mp;
if (mp = dynamic_cast<Motorfordon *>(fp))
    cout << mp->nummer() << endl;

// Alternativt sätt via typeid:
if (typeid(fp)==typeid(Motorfordon)) {
    mp = (Motorfordon *) fp;
    cout << mp->nummer() << endl;
}
```

# Virtuella destruktorer

Om en härledd klass introducerar någon ny dynamisk struktur internt så måste en virtuell destruktör användas i denna klass

## Exempel: Virtuellt destruktör

```
class D : public B {
public:
    D() {s = new char [80];}
    virtual ~D() {delete []s;}
private:
    char *s;
};
```

Utan `virtual` ovan deallokeras inte `s` här:

```
B* d = new D; ... delete d;
```

Medlemsfunktion utan implementering (def.) deklareraras som *rent virtuell funktion* genom initiering till 0

## Exempel: Abstrakt klass

```
class Fordon {  
public:  
    virtual void ge_info()=0;  
    //...  
};
```

Används som "markör" i en basklass för att garantera att funktionen finns med i härledda klasser

- En klass med minst en rent virtuell funktion kallas en *abstrakt klass*
- Inga objekt kan skapas från en abstrakt klass
- Härledda klasser utan egna versioner av varje rent virtuell funktion blir också abstrakta



# Multipel ärvning

I C++ kan en klass ha flera basklasser – Detta kallas för *multipelt arv*

## Exempel: Multipelt arv

```
class Bat : public Fordon { //... };  
class Motorbat : public Bat { //... };  
class Amfibiebil  
    : public Motorbat, public Personbil { //... };
```

Om basklassen har medlemsvariabler så kan man specificera om dessa skall komma med en eller flera gånger i den härledda klassen

## Exempel: Virtuellt basklass

```
// Modifiering för entydighet vid ärvning  
class Bat : virtual public Fordon { //... };  
class Motorfordon : virtual public Fordon  
{ //... };
```

- I Java motsvaras en rent virtuell funktion av en abstrakt metod (dekl. med `abstract`)
- Observera även att alla metoder i Java är automatiskt virtuella med C++-terminologi!
- I Java finns inte multipel ärvning. Istället har man infört *interface* vilket ungefär motsvarar en abstrakt klass med samtliga metoder abstrakta men som dessutom helt saknar attribut