Mittuniversitetet
MID SWEDEN UNIVERSITY

Lektion 3 – Arv och komposition

Objektorienterad programmering i C++

Repetition:

- arv utan dynamisk bindning
- relationerna ’is a kind of…’ och ’uses a…’
- jämförelse mellan arv och komposition

Lektion 3 – Arv

Arv utan dynamisk bindning

Arv eller *generalisering* är en mekanism för återvinning av kod. En ny deriverad klass kan skapas genom att uttrycka förändringar i förhållande till en befintlig klass (basklassen). Man kan säga att man gör en specialisering av en basklass.

Ex. En anställd i ett företag kan representeras genom klassen Employee. Alla anställda har ett *namn* och en *lön*, vilket ger oss två datamedlemmar (eller attribut).

```
class Employee {
private:
      int iSalary;
      string iName;

public:
      Employee(string s)
            :iName(s), iSalary(0)  { }

      string getName() const { return iName; }
      void setSalary(int s) { iSalary = s; }
      int getSalary() const { return iSalary; }
};
```

Antalet chefer i ett företag eller organisation brukar vara ansenligt…
Vi deriverar klassen Manager från Employee.
En chef har förutom sin lön i regel en saftig *bonus* → nytt attribut

```
class Manager : public Employee {
private:
      int iBonus;

public:
      Manager(string s)
            :Employee(s), iBonus(0) { }

      void setBonus(int b) { iBonus = b; }
      int getBonus() const { return iBonus; }
};
```

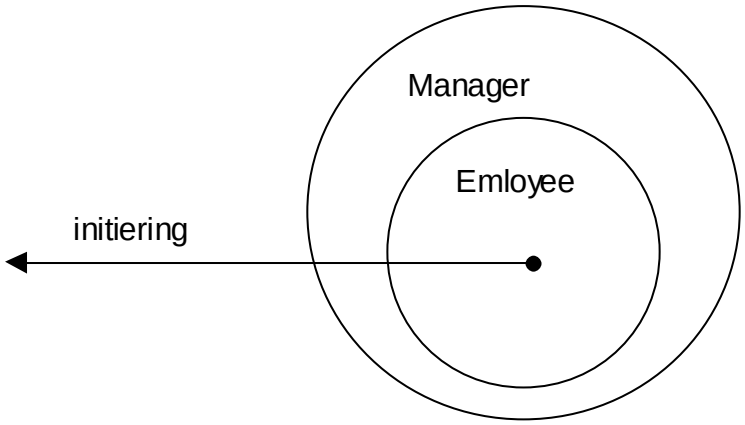
Med publikt arv (`Manager : public Employee`) blir klassen Manager en specialisering eller utvidgning av Employee.

Man kan säga att *”en Manager är en sorts Employee”* . Innebörden i detta är att allt man kan göra med en Employee kan man också göra med en Manager, plus eventuellt lite mer. Detta formuleras i *Substitutionsprincipen* som säger att en deriverad klass alltid kan ersätta sin basklass vid publikt arv. Omvändningen gäller inte!

Om basklassens konstruktor kräver argument så måste den deriverade klassen initiera sin basklass i konstruktorns initieringslista:

```
Manager(string s)
            :Employee(s), iBonus(0) { }
```

Vid instansiering av ett Managerobjekt skapas dess Employee-del först:



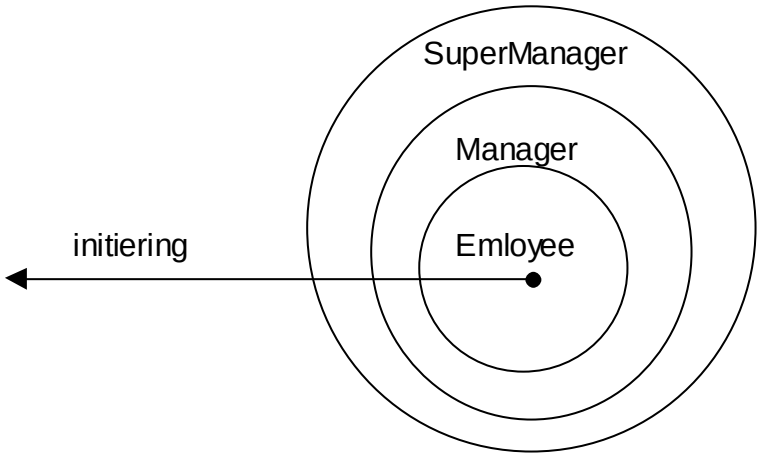
De privata delarna av en basklass kan inte accessas av den deriverade klassen, däremot de delar som är ’protected’. Privat ÄR verkligen privat! Privata delar av basklassen manipuleras mha. basklassens medlemsfunktioner.

En chef med makthunger och ambitioner blir så småningom en Superchef med ett fett fallskärmsavtal (”golden parachute”). Vi deriverar klassen SuperManager från Manager!

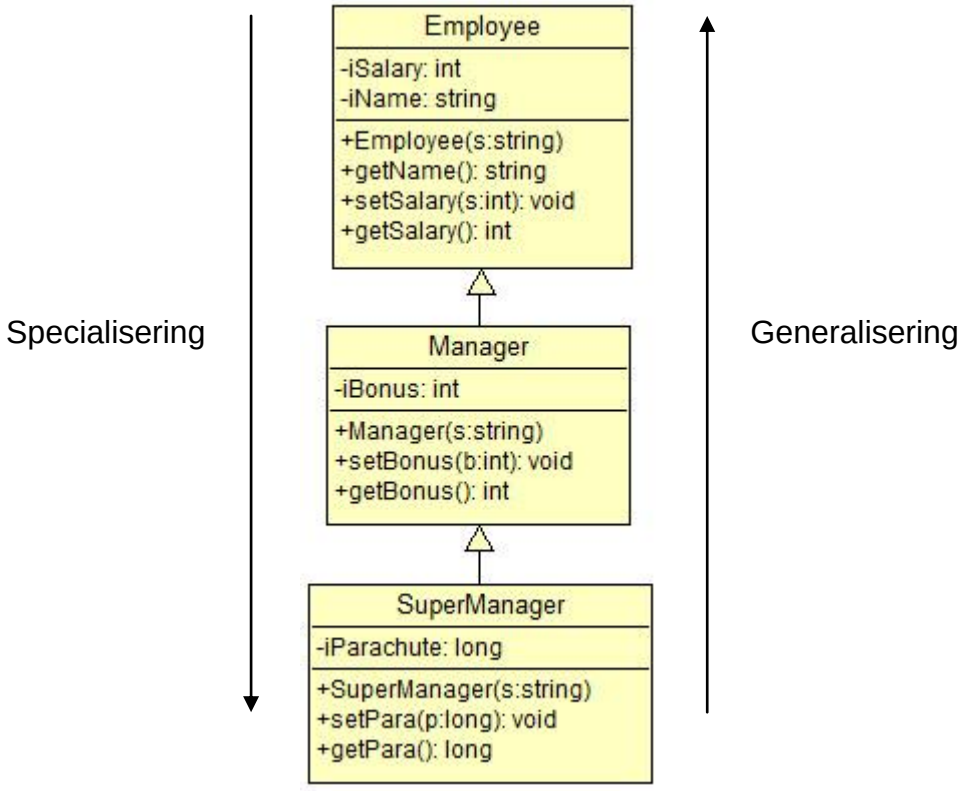
```
class SuperManager:public Manager {
private:
    long iParachute;

public:
    SuperManager(string s)
            :Manager(s), iParachute(1000000) { }

      void setPara(long p) { iParachute = p; }
      long getPara() const { return iParachute; }
};
```

```
SuperManager bigBoss("Bill");
```

kommer att leda till initieringarna



Klasshierarkin uttrycks i UML som



(exEmployee.zip)

Initieringsordningen för ett objekt av en deriverad klass:

- Först initieras basklasserna i den ordning de står i *derivationslistan* i klassdefinitionen
- Därefter initieras datamedlemmarna i den ordning de står i *klassdefinitionen*.

Ovanstående tillämpas rekursivt i en klasshierarki.
Sist exekveras konstruktorns kropp.

Observera att ordningen i konstruktorns initieringslista alltså inte påverkar initieringsordningen.

Exempel (exTrippTrapp):

```
class Trull {
protected:
        int a;
public:
        Trull(int x=0):a(x) {
                cout << "Trull: a=" << a << endl;
        }
};


class Trapp:public Trull {
protected:
        int b;
public:
        Trapp(int x=0):Trull(2*x),b(x) {
                cout << "Trapp: a=" << a << " b=" << b
<<endl;
        }
};


class Tripp:public Trapp {
protected:
        int c;
public:
        Tripp(int x=0):Trapp(2*x),c(x) {
                cout << "Tripp: a=" << a << " b=" << b << "
c=" << c <<endl;
        }
};


int main() {
    Tripp(10);

    return(0);
}
```

Utskrift:

```
Trull: a=40
Trapp: a=40 b=20
Tripp: a=40 b=20 c=10
```

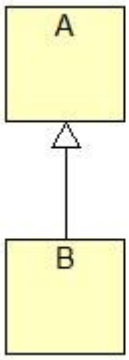
Det är viktigt att påpeka att vi hittills inte har visat arv med virtuella funktioner och dynamisk bindning. Dessa komponenter behövs för att C++ ska uppfylla kraven för ett sant objektorienterat språk. Vi återkommer till detta senare i kursen.

Jämförelse Arv / Komposition

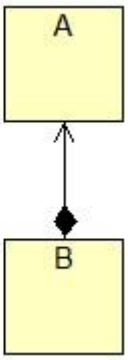
Antag att klass B behöver utnyttja klass A i sin implementation. Hur ska A och B vara relaterade till varandra?

Två tänkbara varianter:

1. Arv



2. Komposition



Vilket är bäst?
Vi ska undersöka saken genom att titta på några exempel.

Antag att vi har en färdig Array-klass för int (exArray.zip) :

```
const int ARRAYSIZE = 10;

class Array {
protected:
        int arrSz;    // Array size
        unique_ptr<int[]> arr; // Dynamic array

public:
        Array(int sz=ARRAYSIZE);

            int size() { return arrSz; }
        int& operator[](int);
            void sort();
};
```

```
Array::Array(int sz)
:arrSz(sz)
{
      arr.reset(new int[arrSz]);
}


int& Array ::operator[](int idx) {

      if (idx >= arrSz || idx < 0) {
       cerr << "\nclass Array ERROR: index " << idx;
       cerr << " out of range. Exiting." << endl;
       exit(1);
   }
   return arr[idx];
}


void Array::sort() {
      // Selection sort
      int i, j, idxSmall, tmp;

      for (i = 0; i < arrSz-1; i++) {
            idxSmall = i; // element to compare to
         for (j = i + 1; j<arrSz-1; j++) //smallest element
                  if (arr[j] < arr[idxSmall])
                        idxSmall = j; // a smaller item
         if (arr[idxSmall] < arr[i]) { // found a smaller
                  tmp = arr[i];         // swap elements
                  arr[i] = arr[idxSmall];
                  arr[idxSmall] = tmp;
            }
      } // for i
}
```

Vi ska utnyttja Array för att skapa en Stack-klass.

Försök 1: Stack får tillgång till Array via *publikt arv*.

(exPubInherit.zip)

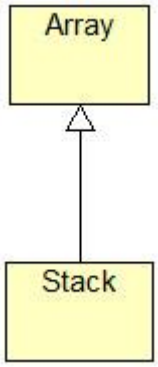
```
class Stack : public Array  {
private:
    int top; // top of stack
    enum {SIZE=100, BOS=-1};
public:
    Stack(int sz=SIZE)
    :Array(sz), top(BOS) { }
    void push(const int &value);
    int pop() ;
    bool empty()const { return top==BOS; }
};

int Stack::pop() {
      return arr[top--];
}

void Stack::push(const int &value) {
      arr[++top] = value;
}
```

Vi testar vår stack med följande kod:

```
  Stack  s(5);
  s.push(2);     // push OK
  s.push(3);
  s.push(4);

  while(!s.empty())
     cout << s.pop() << ' ';  // 4 3 2  OK
  cout << endl;

  s.push(19);
  s.push(13);
  s.push(62);
  s.push(-7);
  cout << s[2] << endl; // 62  Subscripting a stack??
  s.sort();   // What's this? Sorting a stack??

  while(!s.empty())
     cout << s.pop() << ' ';  // 62 19 13 -7
```

Resultatet är lite för bra… Stack-operationerna `push` och `pop` funkar utmärkt *men* det publika arvet från Array möjliggör operationer som t.ex. sortering och indexering vilka inte ska kunna utföras på en stack!

Problemet bottnar i att arvet



inte uttrycker något klart type/subtype-förhållande, dvs en Stack är *inte* en sorts Array (“is NOT a kind of..). Snarare *använder / har* stacken en Array (”uses / has”).

Försök till ändring:
Gör Array till en `private` eller `protected` basklass (exProtInherit).

```
class Stack : protected Array {
private:
    int top; // top of stack
    enum {SIZE=100, BOS=-1};
public:
    Stack(int sz=SIZE)
    :Array(sz), top(BOS) { }

    void push(const int &value);
    int pop() ;
    bool empty()const { return top==BOS; }
};
```

Arrays publika medlemmar blir `protected` i den deriverade klassen Stack. Innebörden är att de kan accessas av Stack men inte av klienter till Stack, i det här fallet `main`-funktionen.

```
Stack myStack;
...
myStack.sort();   // Compile time error:
                  // sort not accessible in Stack
cout << myStack[2]; // Compile time error:
                  // operator [] not accessible in Stack
...
```

Om arvet istället hade gjorts `private` så skulle alla medlemmar som är `public` eller `protected` i Array blivit `private` i Stack.

Slutsats:

- Stack behöver inte Array's *publika* interface
- Stack behöver Array i sin *implementation*
- 'A Stack is not a kind of Array'
- 'A Stack uses/has an Array'

Ett icke-publikt arv, ett så kallat *implementationsarv*, kan alltså användas för att uttrycka relationen 'has/uses'.

Försök2: *Komposition* – en Array är attribut, dvs. datamedlem i Stack (exStackWithArray).

```
class Stack  {
private:
    Array  arr;  // Member!
    int top; // top of stack
    enum {SIZE=100, BOS=-1};
public:
    Stack(int sz=SIZE)
    :arr(sz), top(BOS) { }

    void push(const int &value);
    int pop() ;
    bool empty()const { return top==BOS; }
};
```

```
    Stack  s(5);
    s.push(2);           // push OK
    s.push(3);
    s.push(4);

    while(!s.empty())
      cout << s.pop() << ' ';    // pop OK
    cout << endl;

    s.push(19);
    s.push(13);
    s.push(62);
    s.push(-7);
    // s.sort(); // ERROR: Not a member of Stack

    // cout << s[2] << endl; // ERROR: Not a member..

    while(!s.empty())
      cout << s.pop() << ' ';    // -7 62 13 19
```

Detta fungerar utmärkt!

Slutsatser från de två försöken:

- Relationen “is a kind of…” uttrycks genom *publikt arv*
- Relationen “uses/has…” uttrycks enklast genom *komposition*, eller möjligen via *icke-publikt arv*.

En av många principer inom OOP säger också att man ska försöka välja komposition framför arv om de inte finns starka skäl att inte göra det. Motiveringen till detta hör hemma t.ex. i en designmönsterkurs.