Introduktion till arv

6 INTRODUKTION TILL ARV

Arv
- Generell-Speciell
- Arv för att utnyttja det vi redan gjort
- Återanvändning

Basklass

Härledd klass

Varför arv?
- Inför en subklass för att uttrycka specialisering
- Inför en basklass för att uttrycka generallitet

Olika arv
- Publikt arv - synligt arv
- Exempel med publikt arv
- Initiering av basklassen
- Private basklass, ej synligt arv
- Exempel med privat arv
- Medlemmar som är protected
- Exempel med protected

Abstrakt klass
- Exempel med abstrakt klass

Virtuella funktioner
- Polymorfism
- Varför använda virtuella funktioner
- Exempel med virtuell funktion
- Referens till basklass
- Pekare till basklass

Övningsuppgifter
- Övning 1
- Övning 2
- Övning 3

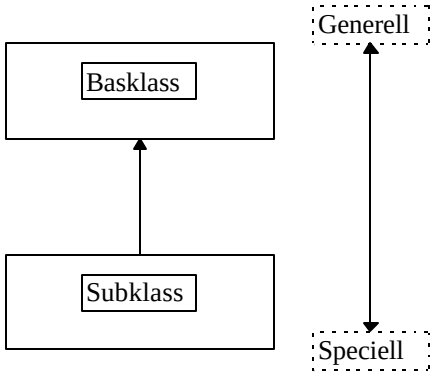
Arv

Arv beskriver att två eller flera klasser har något gemensamt. De är av samma typ men skiljer sig åt i någon eller några detaljer.

Generell-Speciell

Genom dataabstraktion har vi stora möjligheter att komplettera de fördefinierade standardtyperna med egendefinierade klasser. Objektorienterad systemutveckling handlar till stor del om att identifiera, relatera och sedan implementera ett antal klasser inom problemområdet. Vi sysselsätter oss bl.a. med att finna struktur mellan klasser.

En sådan struktur är relationen **generell-speciell** vilket betyder att en klass som ärver av en annan klass alltid skall vara en specialisering av den klass som vi ärver ifrån. Omvänt måste då också gälla att klassen vi ärver ifrån är en generalisering av klassen som ärver.



Arv för att utnyttja det vi redan gjort

Ibland behöver vi en klass som liknar någon som vi redan har. Klassen vi behöver är dock inte riktigt lika som den vi förut konstruerat. Med ett fåtal ändringar så skulle den gamla klassen lösa vårt problem. Vi kan välja att ändra i den gamla klassen och därigenom konstruera en helt ny klass.

Ett bättre alternativ kan i många fall vara att lösa problemet genom att låta en ny klass ärva vissa egenskaper från en befintlig klass. Vi använder oss då av de arv och arvsmekanismer som varje objektorienterat verktyg eller programspråk på något sätt måste stödja.

Genom att vi på detta sätt kan använda de klasser som redan finns kan vi spara mycket tid (och pengar). Programmet kan också bli mindre genom att vissa funktioner bara behöver finnas i basklassen.

Om vi inför arv är det viktigt att den struktur vi därigenom beskriver återfinns i problemområdet

och inte bara blir ett smart sätt att dela kod.

Återanvändning

Återanvändning är ett slagord och något man använder för att marknadsför objektorienterad teknik. Det är ofta genom arv som begreppet återanvändning förverkligas.

Basklass

En klass kan alltså ärva egenskaper från en annan klass. Man säger att klassen är härledd från en basklass. Vi kommer här att använda ordet basklass som beteckning på en klass vars egenskaper ärvs av andra klasser.

I basklassen ska vi försöka samla de egenskaper som är gemensamma för objekt som liknar varandra men som skiljer sig åt i något avseende.

Det finns sällan något i basklassen som säger att den skall användas för arv.

Om en basklass inte själv har någon basklass betyder det att denna klass är den i sammanhanget mest generella.

```
// Eventuell basklass
class Person {
public:
     Person(char *namn, char *adress);
     void print();
private:
     char namn[30];
     char adress[30];
};
```

Härledd klass

En härledd klass (derived) är en specialisering av en annan klass. Om en sublass inte själv har någon subklass betyder det att denna klass är den i sammanhanget mest speciella.

En härledd klass ärver vissa egenskaper från en eller flera basklasser.

Varför arv?

Inför en subklass för att uttrycka specialisering

Man kan inför arv för att uttrycka att en ny klass är ett speciallfall av något som redan finns. Vi inför en **ny subklass**.

Inför en basklass för att uttrycka generallitet

Man kan införa arv därför att man funnit att två eller flera klasser som redan finns har något

gemensamt. Då samlar man de gemensamma egenskaperna i en **ny basklass**.

Olika arv

När vi konstruerar den härledda klassen kan vi välja att låta basklassen vara public eller private (vilket den blir om vi inte anger något).

Dvs som default gäller att basklassen blir private. Om den skall vara public så måste ange detta explicit.

Publikt arv - synligt arv

Vi kan deklarera basklassen som public. Det innebär att de av basklassens medlemmar som är public blir public också i den härledda klassen.

```
class AnstelldPerson: public Person {
public:
     AnstelldPerson(char *namn, char *adress,long loen);
     void visaloen();
private:
     long loen;
};
```

Exempel med publikt arv

Lägg märke till att man har tillgång till den ärvda metoden **print** utan att ange att den är ärvd.

```
// arv1.cpp
#include <iostream.h>
#include <string.h>

class Person {
public:
     Person(char *namn, char *adress);
     void print();
private:
     char namn[30];
     char adress[30];
};

Person::Person(char *n, char *a)
{
     strncpy(namn,n,sizeof(namn));
     strncpy(adress,a,sizeof(adress));
}
void Person::print()
{
     cout << "Namn  :" << namn << endl;
     cout << "Adress:" << adress << endl;
}

class AnstelldPerson: public Person {
public:
     AnstelldPerson(char *namn, char *adress,long loen);
     void visaloen();
private:
     long loen;
};

AnstelldPerson::AnstelldPerson(char *n, char *a, long l)
: Person(n,a),loen(l)
{
}
void AnstelldPerson::visaloen()
{
     cout << "Lön  :" << loen << endl;
}

main()
{
     Person alfa("Arne Karlsson","Gata 4, 123 45 STAD");
     alfa.print();

     AnstelldPerson beta("Karin Karlsson","Gata 6, 123 45 STAD",12000);
     beta.print();         // Ärvd metod
     beta.visaloen();      // subklassens metod...

     return 0;
}
```

Initiering av basklassen

Om basklassens konstruktor kräver parametrar måste denna anropas i en initieringslista i subklassens konstruktor. Lägg märke till hur parametrarna **n** och **a** vidarebefordras till basklassens konstruktor. Det går också att initiera klassens egna datamedlemmar i samma lista (loen).

```
AnstelldPerson::AnstelldPerson(char *n, char *a, long l)
: Person(n,a),loen(l) // Initieringslista
{
}
```

Private basklass, ej synligt arv

Vi kan låta basklassen vara private. Det betyder att den härledda klassen måste förse oss med metoder som gör att vi indirekt kan komma åt data och metoder i basklassen.

De funktioner i basklassen som är public är synliga för den härledda klassens medlemsfunktioner.

Privat arv används ibland för att på ett smart sätt dela kod med andra klasser. Genom att dölja att detta görs blir det privata arvet en del av implementationen och motsvaras ofta inte av någon relation mellan klasser i problemdomänet.

Vår rekomendation är att privat arv inte skall användas.

```
class AnstelldPerson: private Person {
public:
       AnstelldPerson(char *namn, char *adress,long loen);
       void print();
       void visaloen();
private:
       long loen;
};
```

Exempel med privat arv

Lägg märke till att subklassen måste införa en egen printfunktion. den ärvda är ju inte synlig utifrån.

```
// arv2.cpp
#include <iostream.h>
#include <string.h>

class Person {
public:
       Person(char *namn, char *adress);
       void print();
private:
       char namn[30];
       char adress[30];
};

Person::Person(char *n, char *a)
{
       strncpy(namn,n,sizeof(namn));
       strncpy(adress,a,sizeof(adress));
}
void Person::print()
{
       cout << "Namn  :" << namn << endl;
       cout << "Adress:" << adress << endl;
}

class AnstelldPerson: private Person {
public:
       AnstelldPerson(char *namn, char *adress,long loen);
       void print();
       void visaloen();
private:
       long loen;
};

AnstelldPerson::AnstelldPerson(char *n, char *a, long l)
: Person(n,a),loen(l)
{
}

void AnstelldPerson::print()
{
       Person::print(); // Anropa basklassens metod med samma namn...
       visaloen();  // Anropa en annan medlemsfunktion i samma klass....
}

void AnstelldPerson::visaloen()
{
       cout << "Lön   :" << loen << endl;
}

main()
{
       Person alfa("Arne Karlsson","Gata 4, 123 45 STAD");
       alfa.print();
```

```
        AnstelldPerson beta("Karin Karlsson","Gata 6, 123 45 STAD",12000);
        beta.print();       // Egen metod, ej ärvd.

        return 0;
}
```

Medlemmar som är protected

Vi kan också låta vissa medlemmar i basklassen vara protected. Dessa är då synliga för den härledda klassens medlemmar.

Genom detta hindras programmeraren att använda basklassen på annat sätt än indirekt via den härleddda klassens medlemmar..

Exempel med protected

Lägg märke till att subklassens metoder har tillgång till de medlemmar i basklassen somn är protected. Dessutom visar vi att subklassens print-funktion kan anropas om vi anger detta explicit.

```
// arv3.cpp
#include <iostream.h>
#include <string.h>

class Person {
public:
     Person(char *namn, char *adress);
     void print();
protected:
     char namn[30];
     char adress[30];
};

Person::Person(char *n, char *a)
{
     strncpy(namn,n,sizeof(namn));
     strncpy(adress,a,sizeof(adress));
}
void Person::print()
{
     cout << "Namn  :" << namn << endl;
     cout << "Adress:" << adress << endl;
}

class AnstelldPerson: public Person {
public:
     AnstelldPerson(char *namn, char *adress,long loen);
     void print();
private:
     long loen;
};

AnstelldPerson::AnstelldPerson(char *n, char *a, long l)
: Person(n,a),loen(l)
{
}

void AnstelldPerson::print()
{
     cout << "Namn  :" << namn << endl;
     cout << "Adress:" << adress << endl;
     cout << "Lön   :" << loen << endl;
```

```
}

main()
{
     Person alfa("Arne Karlsson","Gata 4, 123 45 STAD");
     alfa.print();

     AnstelldPerson beta("Karin Karlsson","Gata 6, 123 45 STAD",12000);
     beta.print();         // Egen metod, ej ärvd.

     beta.Person::print(); // Anropa den ärvda metoden explicit...

     return 0;
}
```

Abstrakt klass

Om vi deklarerar konstruktorn i en klass som **protected** så kan denna endast anropas av medlemmar i en subklass. det innebär att klassen inte går att instansiera utan indirekt arv och vi har därigenom infört en **abstrakt klass**. Abstrakta klasser definierar bl.a. om en klass är **för** generell och inte kan fungera utan att ha kompletterats genom arv.

Exempel med abstrakt klass

Lägg märke till att det inte går att skapa ett objekt av typen **Person**. Däremot går det bra att ärva från denna klass.

```
// arv4.cpp
#include <iostream.h>
#include <string.h>

// Abstrakt klass
class Person {
public:
     void print();
protected:
     Person(char *namn, char *adress);
private:
     char namn[30];
     char adress[30];
};

Person::Person(char *n, char *a)
{
     strncpy(namn,n,sizeof(namn));
     strncpy(adress,a,sizeof(adress));
}
void Person::print()
{
     cout << "Namn  :" << namn << endl;
     cout << "Adress:" << adress << endl;
}

class AnstelldPerson: public Person {
public:
```

```
    AnstelldPerson(char *namn, char *adress,long loen);
    void visaloen();
protected:
    long loen;
};

AnstelldPerson::AnstelldPerson(char *n, char *a, long l)
: Person(n,a),loen(l)
{
}

void AnstelldPerson::visaloen()
{
    cout << "Lön  :" << loen << endl;
}

main()
{
//  Person alfa("Arne Karlsson","Gata 4, 123 45 STAD"); Går EJ
//       alfa.print();  Går EJ

    AnstelldPerson beta("Karin Karlsson","Gata 6, 123 45 STAD",12000);
    beta.print();        // ärvd metod
    beta.visaloen();

    return 0;
}
```

Virtuella funktioner

Polymorfism

Inom objektorienterad systemutveckling använder man begreppet **polymorfism**. Med detta menas att ett och samma meddelande ger olika resultat beroende på vilket objekt som tar emot meddelandet. I C++ har man implementerat **polymorfism** genom att införa **virtuella funktioner** och **sen bindning**.

Ordet **polymorf** kommer från grekiskan och betyder där **månggestaltad.**

Inom kemin innebär **Polymorfism** att två eller flera mineral med exakt samma kemiska sammansättning ändå kan olika struktur och egenskaper, t.ex. kol som förekommer både som diamant och grafik.

Inom biologin använder man **polymorfism** för att beteckna strukturell eller funktionell variation mellan individer av samma art. Två svampar kan t.ex. tillhöra samma art men ändå ha olika form och färgnyans.

Varför använda virtuella funktioner

Begreppet **virtuell funktion** är starkt förknippat med begreppen **klass** och **arv**. Skall vi ha

någon användning av en virtuell funktion i C++ så förutsätter detta att:

```
1   Det finns en basklass med minst en medlemsfunktion
2   Det finns en subklass med minst en medlemsfunktion
3   Minst en av subklassens medlemsfunktioner har
    identisk prototyp som den virtuella funktionen i
    basklassen
```

Funktioner som subklasser ofta omdefinierar skall deklareras som virtuella så att subklassens funktion anropas när vi faktiskt refererar till eller pekar på ett objekt av subklasstyp.

Om man konstruerar en klass och deklarerar en medlemsfunktion som virtuell (virtual) så kan de klasser som använder klassen som basklass själva innehålla en egen version av den virtuella funktionen.

Vi kan sedan låta en pekare eller referenser till basklasstypen peka på objekt som är av basklasstyp eller någon typ som är härledd från basklassen. Om vi anropar en virtuell funktion med hjälp av en sådan pekare eller referens så är det objektet (det vi pekar på) som bestämmer vilken av de virtuella funktionerna som skall anropas.

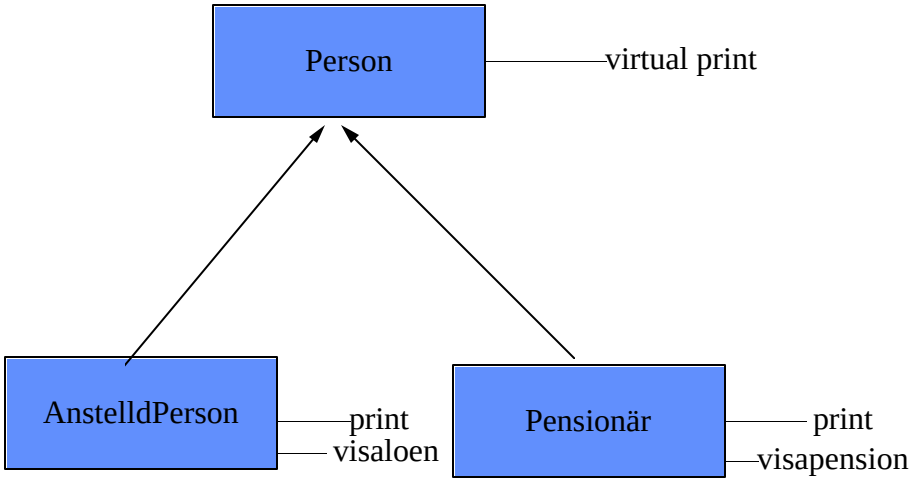
Eftersom kompilatorn vid kompileringen inte säkert att kan avgöra vilken typ av objekt som en pekar- eller referensvariabel kommer att peka i olika skeden under exekveringen av programmet måste en tabell genereras. Denna används sedan under exekveringen av programmet för att vid varje tillfälle anropa (slå upp) rätt funktion. Detta kallas för **sen bindning**.

För en virtuell funktion gäller:

- att den är deklarerad som **virtual** i basklassen
- funktionen returnerar samma typ i både bas- och subklass
- funktionen har samma antal argument i både bas- och subklass
- argumenten har samma typ i både bas- och subklass
- funktionen kan men behöver inte deklareras som **virtual** i subklassen
- om subklassen inte innehåller en egen version av en virtuell funktion i basklassen anropas alltid basklassens funktion

Exempel med virtuell funktion

Antag att vi har konstruerat en basklass Person samt en subklass AnstelldPerson som är härledd från Person. Vi låter båda klasserna ha en medlemsfunktion med namnet **print**. Eftersom **print** omdefineras i subklassen är det viktigt att den deklareras som **virtual** i basklassen.



Observera att funktionen automatiskt blir virtuell i den härledda klassen (AnstelldPerson och Pensionaer) utan att vi explicit behöver deklarera den som sådan.

Funktioner som inte skall omdefinieras i någon subklass deklareras inte som virtuell.

```
// arv5.cpp
#include <iostream.h>
#include <string.h>

class Person {
public:
       Person(char *namn, char *adress);
       virtual void print();  // OBS Virtuell funktion i basklass
protected:
       char namn[30];
       char adress[30];
};

Person::Person(char *n, char *a)
{
       strncpy(namn,n,sizeof(namn));
       strncpy(adress,a,sizeof(adress));
}
void Person::print()
```

```
{
       cout << "***** Person *****\n";
       cout << "Namn  :" << namn << endl;
       cout << "Adress:" << adress << endl;
}

class AnstelldPerson: public Person {
public:
       AnstelldPerson(char *namn, char *adress,long loen);
       void print();              // Blir virtuell
       void visaloen();  // Egen funktion ej virtuell....
private:
       long loen;
};

AnstelldPerson::AnstelldPerson(char *n, char *a, long l)
: Person(n,a),loen(l)
{
}

void AnstelldPerson::print()
{
       cout << "***** Anstelld *****\n";
       cout << "Namn  :" << namn << endl;
       cout << "Adress:" << adress << endl;
       visaloen();  // Anropa en annan medlemsfunktion i samma klass....
}

void AnstelldPerson::visaloen()
{
       cout << "Lön   :" << loen << endl;
}


class Pensionaer: public Person {
public:
       Pensionaer(char *namn, char *adress,long pension);
       void print();              // Blir virtuell pga. basklassens funktion...
       void visapension(); // Egen funktion ej virtuell....
private:
       long pension;
};

Pensionaer::Pensionaer(char *n, char *a, long p)
: Person(n,a),pension(p)
{
}

void Pensionaer::print()
{
       cout << "***** Pensionär *****\n";
       cout << "Namn  :" << namn << endl;
       cout << "Adress:" << adress << endl;
       visapension();      // Anropa en annan medlemsfunktion i samma klass....
}

void Pensionaer::visapension()
{
       cout << "Pension   :" << pension << endl;
}
```

```
void showperson(Person& personref)
{
	personref.print(); // Sen bindning och polymorfism....
}

main()
{
	Pensionaer    alfa("Olle Olsson","Gata 10, 123 45 STAD",9953);
	AnstelldPerson beta("Karin Karlsson","Gata 6, 123 45 STAD",12000);
	Person        gamma("Nisse Svensson","Gata 8, 123 45 STAD");

	showperson(alfa);
	showperson(beta);
	showperson(gamma);

	Person  *array[] = { &alfa, &beta, &gamma };
	for(int i=0;i<sizeof(array)/sizeof(array[0]) ;i++)
		array[i]->print();

	return 0;
}
```

Referens till basklass

Lägg märke till att funktionen **showperson** kan anropas med godtycklig typ av objekt så länge dessas typ är **Person** eller av någon typ som är härledd från denna klass.

```
showperson(alfa);
showperson(beta);
showperson(gamma);
```

En variabel av typen **Person&** kan referera till godtycklig typ inom hierarkin. I funktionen binds sedan anropet av print till aktuellt objekts egen printfunktion. Eftersom det kan vara olika typer av objekt vid olika anrop används **sen bindning**, dvs funktionsanropet binds när programmet exekveras.

```
void showperson(Person& personref)
{
	personref.print(); // Sen bindning och polymorfism....
}
```

Pekare till basklass

På samma sätt är det möjligt att låta en pekare av basklasstyp peka på något objekt som är härlett från denna klass. Vi kan t.ex. låta pekarna i ett fält peka på de tre objekten.

```
Person  *array[] = { &alfa, &beta, &gamma };
```

När vi sedan använder pekarna kommer dessa att peka på olika typer av objekt. Anropet av **print** binds under exekveringen till rätt funktion för respektive objekt pga att denna är virtuell.

```
for(int i=0;i<sizeof(array)/sizeof(array[0]) ;i++)
		array[i]->print(); // Sen bindning
```

Övningsuppgifter

Övning 1

Antag att vi skall konstruera ett banksystem och har funnit objekt av typerna **Kapitalkonto** och **Personkonto.** Antag att man får göra högst fyra uttag per år från ett kapitalkonto utan avgift. Efter fjärde uttaget samma år blir avgiften en procent av det uttagna beloppet. Från personkontot får man däremot göra obegränsat med uttag. Alla konton har ränta men denna är givetvis olika stor för olika konton.

Försök att binda dessa klasser till en och samma hierarki (de har något gemensamt) genom att designa minst tre klasser. Pröva dina klasser i ett fungerande program. Fundera själv ut vilka metoder och attribut som behövs och deras placering.

Övning 2

Antag att vi har studerat ett problemdomän och funnit objekt av typerna **kvadrater**, **cirklar**, **parallellogram, ellipser** och **rektanglar.** Gör en analys och designa sedan en hierarki i C++ där motsvarande klasser ingår. Du kan behöva införa någon klass för att få in alla i samma hierarki.

Önskvärda egenskaper för objekten i det system som skall konstrueras är att de skall ha en viss position (x,y), storlek, färg samt riktning. Man skall kunna be ett objekt att förflytta sig till viss position, att röra sig i aktuell riktning samt visa eller gömma sig. Objekten skall därför hålla reda på om det för närvarande är synligt eller inte.

(Om du kan använda grafik kan du välja att låta objekten visa sig på skärmen annars får du nöja dig med textutskrifter)

Övning 3