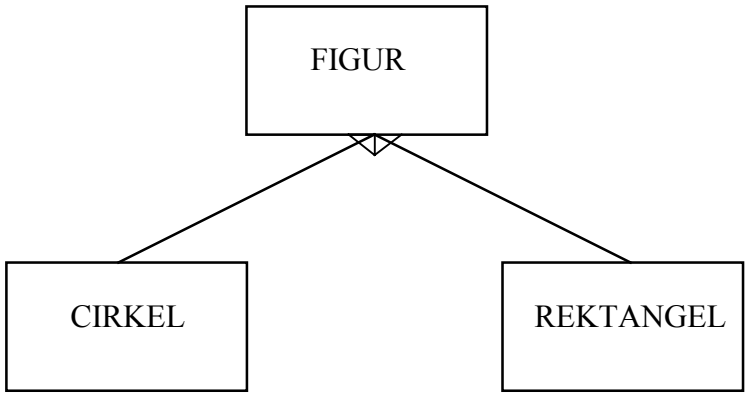
5 Arv och dynamisk bindning

Arv är en av hörnstenarna i objektorienterad programmering. Med hjälp av arv kan man skapa underhållsvänliga och förändringsvänliga system. Att hitta arvsrelationer är en viktig del av systemarbetet. Arvsrelationer kan vara både specialiseringar och generaliseringar. Specialisering innebär att man från en basklass hittar subklasser. Generalisering innebär att man samlar ihop subklasser till en basklass. Arvsrelationen kontrolleras med frågan 'är en'. Elefant är ett däggdjur. Dialogfönster är ett fönster osv. Arv är ett steg mot att programmera med *återanvändbara komponenter*. Man har en färdig fönsterklass (komponent) som man själv kan modifiera till en önskad subfönsterklass genom att *lägga till data och funktioner eller förändra funktionaliteten genom att överlagra funktioner*.

Ex : En rektangel *är en* figur och en cirkel *är en* figur. Ska man använda både rektanglar och cirklar i sitt system är det lämpligt att generalisera och använda en basklass figur som håller reda på sådant som är gemensamt, som exempelvis namnet medan subklasserna håller reda på det som är speciellt, exempelvis sidornas längder för rektangeln och radiens längd för cirkeln.




```
// specifikation av klassen FIGUR -- figur.h

#ifndef FIGURH
#define FIGURH

class FIGUR
{
   private:
      char namn[20];
   public:
      FIGUR(char *namn = "");
      ~FIGUR();
      void skriv();
};

#endif
```

OBS! *Villkorlig kompilering* för att undvika kompileringsfel på grund av dubbla definitioner av klassen FIGUR eftersom figur.h kommer att inkluderas av både rektang.h och cirkel.h.

```
// implementation av klassen FIGUR -- figur.cpp

#include "figur.h"
#include <cstring>
#include <iostream>
using namespace std;


FIGUR::FIGUR(char *namn)
{
   strcpy(this->namn, namn);
}


FIGUR::~FIGUR()
{
}


void FIGUR::skriv()
{
   cout << endl << "Figurnamn : " << namn << endl;
}
```

```
// specifikation av klassen REKTANGEL -- rektang.h

#include "figur.h"

class REKTANGEL : public FIGUR           // OBS! Ärver FIGUR
{
   private:
      float s1, s2;
   public:
      REKTANGEL(char *namn, float s1, float s2);
      ~REKTANGEL();
      float area();
      void skriv();
};
```

```
// implementation av klassen REKTANGEL -- rektang.cpp

#include "rektang.h"
#include <iostream>
using namespace std;


REKTANGEL::REKTANGEL(char *namn, float s1, float s2)
                     :FIGUR(namn) // OBS! Anropar baskonstruktor
{
   this->s1 = s1;
   this->s2 = s2;
}


REKTANGEL::~REKTANGEL()
{
}


float REKTANGEL::area()
{
   return s1*s2;
}


void REKTANGEL::skriv()
{
   FIGUR::skriv();
   cout << "Sida 1 : " << s1 << endl;
   cout << "Sida 2 : " << s2 << endl;
}




// specifikation av klassen CIRKEL -- cirkel.h

#include "figur.h"

class CIRKEL :public FIGUR               // OBS! Ärver FIGUR
{
   private:
      float r;
   public:
      CIRKEL(char *namn, float r);
      ~CIRKEL();
      float area();
      void skriv();
};
```

```
// implementation av klassen CIRKEL -- cirkel.cpp

#include "cirkel.h"
#include <iostream>
using namespace std;

CIRKEL::CIRKEL(char *namn, float r)
               :FIGUR(namn)   // OBS! Anropar baskonstruktor
{
   this->r = r;
}

CIRKEL::~CIRKEL()
{
}

float CIRKEL::area()
{
   const float pi = 3.14159;

   return r*r*pi;
}

void CIRKEL::skriv()
{
   FIGUR::skriv();
   cout << "Radie : " << r << endl;
}
```

OBS! REKTANGEL-klassen och CIRKEL-klassen ärver medlemsvariabeln namn från basklassen. Objekt av klassen CIRKEL kommer alltså att ha värden på både namn och radie. Även om CIRKEL ärver namn från basklassen kan inte CIRKEL använda namn direkt vid exempelvis utskriften. *Även subklasser måste gå via de publika medlemsfunktionerna*. Vill man ge subklasser tillåtelse att använda medlemsvariabler direkt ska man definiera dessa *protected* istället för private i basklassen. *Medlemmar som är protected kan refereras direkt av subklasser men ej av andra*.

OBS! Medlemsfunktionen skriv ärvs också av subklasserna men här ändras funktionaliteten genom att funktionen omdefinieras (överlagras) i subklassen. Då skriv anropas av CIRKEL-objektet kommer denna överlagrade funktion att användas.

OBS! Basklassens konstruktor anropas alltid *före* subklassens. Parametrar till basklassens konstruktor *måste överföras via subklassens konstruktor* (jfr medlemsobjekt). Basklassens destruktor anropas alltid *efter* subklassens. Här finns inga problem med parametrar då destruktorn ej får ha några.

```
// huvudprogram -- figurmai.cpp

#include "cirkel.h"
#include "rektang.h"

void main()
{
   CIRKEL cir("Cirkel", 1.0);
   REKTANGEL rek("Rektangel", 2.0, 4.5);
   FIGUR fig, *fp;

   cir.skriv();
   cout << "Area : " << cir.area() << endl;

   rek.skriv();
   cout << "Area : " << rek.area() << endl;

   fig = cir;
   fig.skriv();

   fp = &cir;
   fp->skriv();
}
```

Utskriften blir :

Figurnamn : Cirkel
Radie : 1
Area : 3.14159

Figurnamn : Rektangel
Sida1 : 2
Sida2 : 4.5
Area : 9

Figurnamn : Cirkel

Figurnamn : Cirkel

OBS! Det går att tilldela objektet cir av subklassen CIRKEL till objektet fig av basklassen FIGUR, men ej tvärtom. Vid tilldelningen tappar man bort radie eftersom objekt av klassen FIGUR ej har någon sådan medlemsvariabel.

OBS! Det går också att tilldela adressen för objektet cir av subklassen CIRKEL till en pekare fp av basklassen FIGUR. Objektet blir dock av typen FIGUR och FIGUR:s skrivfunktion körs. Man kan dock med hjälp av *virtual*-markering framför skrivfunktionen, ange att den ska bindas dynamiskt dvs bindas då pekaren vet vilket objekt den pekar på.

```
class FIGUR
{
   private:
      char namn[20];
   public:
      FIGUR(char *namn = "");
      ~FIGUR();
      virtual void skriv();        // OBS! virtual
};
```

Utskriften blir nu istället :

Figurnamn : Cirkel
Radie : 1
Area : 3.14159

Figurnamn : Rektangel
Sida1 : 2
Sida2 : 4.5
Area : 9

Figurnamn : Cirkel

Figurnamn : Cirkel
Radie : 1 // OBS!

Den sista utskriften anropar subklassens CIRKEL skrivfunktion även om man utnyttjar en pekare till basklassen FIGUR. Genom att vi angett virtual framför skrivfunktionen kommer denna att bindas till aktuellt objekt dynamiskt alltså under körning. Detta kallas dynamisk bindning.

5.1 Dynamisk bindning

Dynamisk bindning innebär att man under programkörningen överlåter åt programmet att leta upp rätt funktion för aktuellt subobjekt. Man definierar en pekare till en basklass och tilldelar denna pekare adressen till olika subobjekt. *Genom dynamisk bindning kommer rätt funktion att anropas vid rätt tillfälle*. Man kan exempelvis köra igenom en hel lista av subobjekt och varje gång anropas korrekt subobjekts medlemsfunktion.

Ofta används *basklassen enbart som en abstrakt klass* dvs man skapar aldrig några verkliga objekt av basklassen utan enbart pekare med vars hjälp man pekar ut subobjekt. Som tillverkare av klasser kan man *tvinga användare* att använda basklassen enbart som en abstrakt klass genom att tilldela de virtuella funktioner, som man vill ska omdefinieras, värdet 0 (NULL) vid specifikationen. Man säger att funktionen nu är *äkta virtuell*. *Definierar ej användaren om äkta virtuella funktioner i sin subklass blir även subklassen abstrakt* och eftersom man ej kan skapa objekt av abstrakta klasser, kan man ej deklarera objekt av subklassen heller o.s.v.

Ex : Gör om klassen FIGUR till en abstrakt basklass genom att göra areafunktionen äkta virtuell.

```
// specifikation av klassen FIGUR

class FIGUR
{
   private:
      char namn[20];
   public:
      FIGUR(char *namn = "");
      virtual ~FIGUR();
      virtual void skriv();
      virtual float area() = 0;    // OBS! Klassen blir abstrakt
};


// huvudprogram -- figurmai.cpp

#include "cirkel.h"
#include "rektang.h"

void main()
{
   CIRKEL cir("Cirkel", 1);
   REKTANGEL rek("Rektangel", 2, 4.5);
   FIGUR fig;     // Kompileringsfel! Kan ej skapa objekt
                  // av abstrakta klasser.
   FIGUR *fp;

   fp = &cir;
   fp->skriv();
   cout << "Area : " << fp->area();

   fp = &rek;
   fp->skriv();
   cout << "Area : " << fp->area();
}
```

Utskriften blir nu :

Figurnamn : Cirkel
Radie : 1
Area : 3.14159

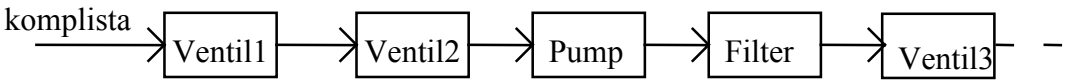
Figurnamn : Rektangel
Sida1 : 2
Sida2 : 4.5
Area : 9

Samma resultat hade man åstadkommit med nedanstående program som allokerar subobjekten dynamiskt :

```
#include "cirkel.h"
#include "rektang.h"

void main()
{
   FIGUR *fp;

   fp = new CIRKEL("Cirkel", 1);
   fp->skriv();
   cout << "Area : " << fp->area();
   delete fp;

   fp = new REKTANGEL("Rektangel", 2, 4.5);
   fp->skriv();
   cout << "Area : " << fp->area();
   delete fp;

}
```

OBS! Destruktorn måste också definieras virtuell eftersom annars körs endast basklassens destruktor. Definierar man destruktorn som virtuell kommer subklassens destruktor att köras först och därefter basklassens.

Ex : I kylsystemet har man ett stort antal VVS-komponenter som man ska löpa igenom och göra beräkningar och regleringar på. Här är det lämpligt att definiera en basklass, VVS och göra en länkad lista av VVS-komponenter som man löper igenom varje cykel. Varje subklass har en dynamikfunktion som utför beräkningarna. Genom dynamisk bindning körs rätt dynamikfunktion vid rätt tillfälle.



Komplista, som blir av typen stack, skapas i konstruktorn för klassen KYLSYSTEM enligt:

```
class KYLSYSTEM
{
   . . . .

   private:
      VVS *komplista;
      NOD *np1, *np2, . . .
      . . . .
};


KYLSYSTEM::KYLSYSTEM()
{
   . . . .
   komplista = NULL;
   np1 = new NOD(komplista, . . .  .);
   . . . .
}
```

I de olika händelsefunktionerna skaffar man sig en lokal iterator kp med vars hjälp man löper igenom listan med komponenter enligt exempelvis:

```
void KYLSYSTEM::timer_Tick(Object *Sender, eventargs *e)
{
      VVS *kp = komplista;

      while (kp)
     {
         kp->dynamik();
         kp->display();
         kp = kp->get_next();
     }

}
```

jälva hoplänkningen av komponenterna sker i konstruktorn för VVS-klassen enligt:

```
class VVS
{
   private:
      VVS *next;                    // pekare till nästa komponent
   protected:
      char namn[10];                // namn på komponenten
      int x, y;                     // koordinater
   public:
      VVS( VVS *&next, char *namn, int x, int y);    // konstruktor
      virtual void dynamik(){}      // dummy dynamik
      virtual void display(){}      // dummy display
      VVS *get_next();              // nästa komponent
      . . . . .

VVS::VVS(VVS *&next, char *namn, int x, int y)
{
   this->next = next;
   next = this;
   strcpy(this->namn, namn);
   this->x = x;
   this->y = y;
}


VVS *VVS::get _next()
{
   return next;
}
```

OBS! *& är en *referens till en pekare*. Detta måste man ha för att kunna ändra pekarvärdet next då nya komponenter stoppas in.

OBS! Medlemsvariablerna namn, x och y är märkta *protected* vilket innebär att de kan användas direkt av en subklass som VENTIL (se nedan) men däremot ej av

instansierade objekt..

OBS! Hur man utnyttjar *this-pekaren* för att länka ihop komponenterna och ge next-parametern ett nytt värde. Alla objekt har en this-pekare som pekar på sig själv.

OBS! Här har man valt att *ej* definiera dynamikfunktionen *äkta virtuell*. Detta innebär att VVS-klassen *ej blir abstrakt*. Man kan alltså skapa objekt av klassen VVS och det kan finnas subklasser, som ej har någon överlagrad dynamikfunktion och som man också kan skapa objekt av.

VENTIL-klassen ärver all data från VVS-klassen och man lägger till de data och de funktioner som är speciella för ventiler. Man ska också förändra VENTIL-objektens funktionalitet gentemot VVS-objekten genom att definiera en ny överlagrad dynamik-funktion istället för den dynamikfunktion i VVS som inte gör någonting. Detsamma gäller display, rita, klick etc.

```
class VENTIL : public VVS          // OBS! Ärver från VVS
{
   private:
      float f;          // flödet i ventilen
      NOD *in;          // in-nod
      NOD *ut;          // ut-nod
      .....

   public:
      VENTIL(VVS *&next, char *namn, int x, int y, NOD *in, NOD *ut);
      void dynamik();
      .....
};
```

```
VENTIL::VENTIL(VVS *&next, char *namn, int x, int y,
               NOD *in, NOD *ut) : VVS(next, namn, x, y)
{
   this->in = in;
   this->ut = ut;
}
```

OBS! Hur man överför parametrar till basklassens konstruktor genom att i subklassens initieringslista vissa parametrar slussas vidare till anropet av basklassens konstruktor. Basklassens konstruktor körs alltid innan subklassens.

En klass kan ses som en server som kan ha två olika typer av klienter:

1) *Användarklient* : En klient som via objekt eller pekare till objekt av klassen *använder* de publika medlemmarna.

2) *Arvsklient* : En klient som skapar nya subklasser genom att *ärva* data och funktioner från klassen.

En tillverkare av en server-klass bestämmer själv hur medlemmarna ska skyddas mot klienterna genom att markera medlemmarna som *private, protected eller public*. För en *användarklient* innebär detta att enbart de publica medlemmarna kan användas emedan en *arvsklient* kan utnyttja både protected- och public-medlemmar.

En arvsklient kan dessutom ange hur den ska ärva egenskaper *public, protected eller private* enligt :

```
class SUBKLASS : public BASKLASS
```

eller

```
class SUBKLASS : protected BASKLASS
```

eller

```
class SUBKLASS : private BASKLASS
```

Hur man som arvsklient ärver inverkar på fortsättningen av arvskedjan och användarkedjan.

Ärver man public så får subklassen *samma* skydd mot nya klienter som basklassen.

Ärver man protected ändras skyddet för de ärvda publica medlemmarna från basklassen till protected i subklassen och dessa kan alltså *ej användas av användarklienter till subklassen utan bara av arvsklienter*.

Ärver man private blir alla ärvda medlemmar från basklassen private i subklassen och kan ej användas av några klienter alls utan det är bara subklassen själv som kan utnyttja de ärvda medlemmarna. På detta sätt kan man bryta en arvskedja och stoppa vidare arv.

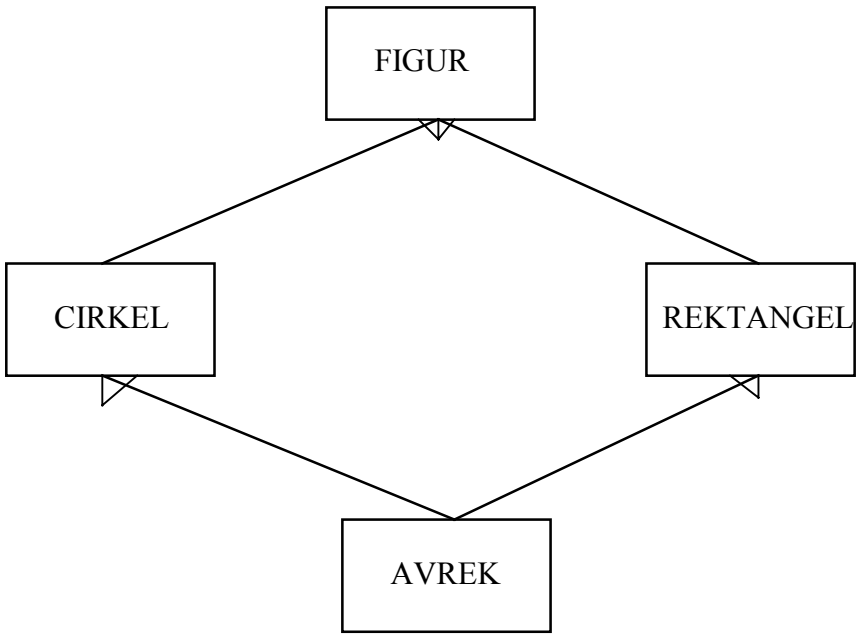
5.2 Multipelt arv

Arv kan man utnyttja i många steg. En kvadrat är en rektangel och en rektangel är en figur. Man kan också i C++ ha *multipelt arv* dvs man kan ärva från flera klasser. Här ska man egentligen fråga sig om ett subobjekt *är ett* basobjekt 1 *och ett* basobjekt 2.

Ex: En student 'är både en' kårmedlem och en låntagare på biblioteket och ärver exempelvis medlemsnummer från kårmedlem och låntagarnummer från låntagare.

Ofta använder man också multipelt arv då en klass ärver egenskaper från flera andra klasser utan att man egentligen kan säga att objekt av klassen är objekt av båda de klasser som man ärver ifrån.

Ex: En avrundad rektangelklass, AVREK, som består av en rektangel med cirkelrundade kortsidor, ärver egenskaper från både CIRKEL och REKTANGEL enligt :



```
// specifikation av klassen AVREK -- avrek.h

#include "rektang.h"
#include "cirkel.h"

class AVREK : public REKTANGEL, public CIRKEL  // OBS!
{
   public:
      AVREK(char *namn, float sida, float radie);
      float area();
      void skriv();
};


// implementation av klassen AVREK -- avrek.cpp

AVREK::AVREK(char *namn, float sida, float radie)
            : REKTANGEL(namn, sida, 2*radie)
            , CIRKEL(namn, radie)              // OBS!
{
}

float AVREK::area()
{
   return REKTANGEL::area() + CIRKEL::area();
}

void AVREK::skriv()
{
   REKTANGEL::skriv();
}


// huvudprogram -- avrekmai.cpp

#include "avrek.h"

void main()
{
   AVREK avr("AVREK", 100, 25);

   avr.skriv();
   cout << "Area :" << avr.area() << endl;
}
```

Vid multipelt arv uppstår två problem :

1) Om det finns medlemmar med samma namn i de klasser man ärver från protesterar kompilatorn. Detta kan man lösa genom att använda globaloperatorn :: eller genom att definiera om medlemmen inuti den nya klassen. I ovanstående exempel definierar man om funktionen skriv i AVREK-klassen och då har kompilatorn inga problem med att hitta rätt funktion. Hade man inte gjort någon ny skrivfunktion i AVREK skulle man ha fått kompileringsfel vid anropet avr.skriv().

2) Data från bas-bas-klassen FIGUR ärvs dubbelt av AVREK. Detta innebär att objektet avr innehåller namnsträngen två gånger vilket är slöseri med minne. För att slippa detta kan man se till att klasserna REKTANGEL och CIRKEL ärver FIGUR-klassen virtual enligt :

```
class REKTANGEL : virtual public FIGUR   // OBS! virtual
{
   ......

};

class CIRKEL : virtual public FIGUR      // OBS! virtual
{
   ......

};
```

Nu har man bara ett namn i AVREK-klassen men frågan är vilken av klasserna REKTANGEL eller CIRKEL anropar FIGUR:s konstruktor? Svaret är att ingen av dem anropar konstruktorn eftersom de ärver virtuellt. Man måste anropa FIGUR:s konstruktor explicit från AVREK enligt :

```
// implementation av klassen AVREK -- avrek.cpp

AVREK::AVREK(char *namn, float sida, float radie)
            : REKTANGEL(namn, sida, 2*radie)
            , CIRKEL(namn, radie),
            , FIGUR(namn)                // OBS!
{
}

.....
```

Multipelt arv är ett omtvistat kapitel inom objektorientering. Många objektorienterade språk saknar denna möjlighet medan andra, som C++, har den implementerad. Man märker av ovanstående exempel att multipelt arv kan ställa till problem. Regeln som man bör ha är att använda multipelt arv i yttersta nödfall. Kan man undvara multipelt arv så bör man göra det. Ovanstående exempel kunde man ha löst med en aggregatrelation. AVREK består av ett REKTANGEL-objekt och ett CIRKEL-objekt.

Ex: Skriv om klassen AVREK ovan så att den istället för arv använder REKTANGEL-objekt och CIRKEL-objekt som medlemsobjekt.

```
// specifikation av klassen AVREK -- avrek.h

#include "rektang.h"
#include "cirkel.h"

class AVREK
{
   private:
      REKTANGEL rek;
      CIRKEL cir;
   public:
      AVREK(char *namn, float sida, float radie);
      float area();
      void skriv();
};



// implementation av klassen AVREK -- avrek.cpp

AVREK::AVREK(char *namn, float sida, float radie)
            : rek(namn, sida, 2*radie)
            , cir(namn, radie)                 // OBS!
{
}

float AVREK::area()
{
   return rek.area() + cir.area();
}

void AVREK::skriv()
{
   rek.skriv();
}


// huvudprogram -- avrekmai.cpp

#include "avrek.h"

void main()
{
   AVREK avr("AVREK", 100, 25);

   avr.skriv();
   cout << "Area :" << avr.area() << endl;
}
```