Kap 3, Klasser och objekt

datatyp ↓ Klasser, variabel ↓ objekt

```
datatyp     variabel
float       x;
STUDENT     a;
```

(Ex) Klassen VENTIL i projektet kylsim

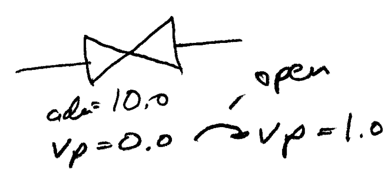
```
//Specifikation -- Ventil.h
class VENTIL
{
  private:
    float vp; //ventilkäglans position 0.0 - 1.0   (Stängd 0.0, Öppen 1.0)
    float adm; //ventilens genomsläpplighet (admittans)
    :
  public:
    void init(float aktvp, float aktadm);
    void open();
    void close();
    :
};
```

OBS! Man bör sätta in villkorlig kompilering i alla h-filer annars får man kompileringsfel, double declaration.

```
#ifndef VENTILH
#define VENTILH
class VENTIL
{
  :
};
#endif
```

OBS! Första gången klassen kompileras definieras VENTILH vilket gör att nästa kompilering hoppar över klassen

```
//Implementation -- Ventil.cpp
#include "Ventil.h"
void VENTIL::init(float aktvp, float aktadm)
{
    vp = aktvp;
    adm = aktadm;
}
void VENTIL::open()
{
    vp = 1.0;
}
void VENTIL::close()
{   vp = 0.0;
}

// Huvudprogram -- Ventmain.cpp
#include "Ventil.h"
void main()
{
    VENTIL v;
    V.init(0.0,10.0)
    V.open();
    V.vp = 0; Fel! vp är private!
```

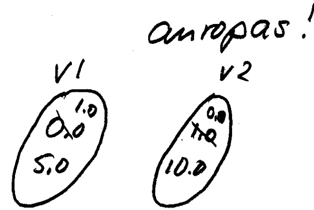


Konstruktor och destruktor

(Ex) Initiering i klassen VENTIL kan göras med en separatfunktion som ovan. Bättre är dock att använda en konstruktor.

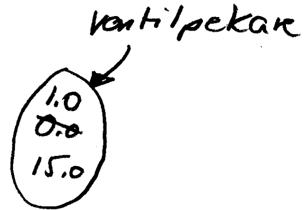
```
//Specifikation -- Ventil.h
class VENTIL
{    private:
        float vp;
        float adm;
        :
     public:
       VENTIL(float aktvp, float aktadm); //Konstruktor
      ~VENTIL(); //Destruktor
       void open();
       void close();
       :
};
// Implementation -- Ventil.cpp
VENTIL::VENTIL(float aktvp, float aktadm)
{    vp = aktvp;
    adm = aktadm;
}
VENTIL :: ~VENTIL()
{
}
   |
   |
   |
   |
```

```
//Huvudprogram -- Ventmain.cpp
#include "Ventil.h"
void main()
{
    VENTIL V1(0.0, 5.0), V2(1.0, 10.0); //Konstruktorn
                                           anropas!
    V1.open();
    V2.close();
    :
} //Här anropas destruktorn!
```



Alternativt main-program!

```
void main()
{
    VENTIL *ventilpekare;

    ventilpekare = new VENTIL(0.0, 15.0); ①
    ventilpekare->open();
    delete ventilpekare; ②
    :
}
```



OBS! ① Konstruktorn körs!
② Destruktorn körs!

En konstruktor körs alltid då ett objekt skapas Destruktorn körs alltid då objektet förstörs

Har du ej definierat en egen konstruktor körs en default konstruktor som inte gör någonting. Detta gäller även destruktorn!

(Ex) VENTIL-klassen

```
:
VENTIL v1(0.0, 5.0), v2(1.0, 10.0), v3;
```

v3 ger fel vid kompileringen, konstruktor saknas!

Åtgärd ① Skriv en egen konstruktor utan parametrar.

Åtgärd ② Skriv en egen konstruktor med defaultvärden på alla parametrar

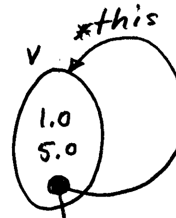
thispekaren

Ett objekt vet sin egen adress — this

(Ex)

```
:
VENTIL v (1.0, 5.0);
:
v.close(); //skickar this
```

för v till close-funktionen så att close vet vilket objekt den ska stänga!

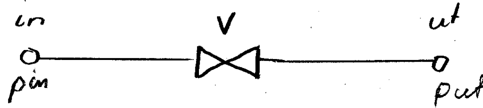


(Ex) Med this kan man skilja på parametrar och medlemsdata som har samma namn.

```
:
VENTIL::VENTIL (float vp, float adm)
{
  this->vp = vp;
  this->adm = adm;
}
```

Relationer

(Ex) En ventil känner till sina noder
— ny relation association



//Specifikation -- Nod.h

Ventilens fysik!

Om pin >= put

$flode = vp * adm * \sqrt{pin - put}$

eller om pin < put

$flode = vp * adm * \sqrt{put - pin}$

```
class NOD
{
    private:
        bool reglerbar;
        float p; //Tryck
        float summaflode;
        :
    public:
        NOD(float tryck, bool reglerbar);
        float get_tryck();
        void add_summaflode(float flode);
        void dynamik(); //Reglera
        :
};
```

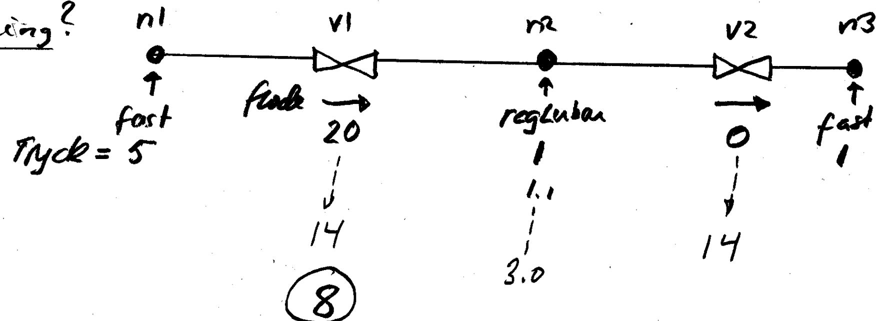
//Implementation -- Nod.cpp

```
NOD::NOD(float p, bool reglerbar)
{
    this->p = p;
    this->reglerbar = reglerbar;
    summaflode = 0.0;
}
float NOD::get_tryck()
{
    return p;
}
```

```
void NOD::addsummaflode(float flode)
{     if(reglerbar)
      {
      addsummaflode += flode;
      }
}

void NOD::dynamik()
{
    if(reglerbar)
    {    if (summaflode > 0)
         {
           p += 0.1;
         }
         else if(summaflode < 0)
         {   p -= 0.1;
         }
         summaflode = 0.0;
    }
}

//Specifikation -- Ventil.h
#include "Nod.h"
class VENTIL
{    private:
        float vp, adm, flode;
        NOD *in, *ut;          //Känner till relation
     public:
        VENTIL(float vp, float adm, NOD *in, NOD *ut);
        void dynamik();
```

```
//Implementation -- Ventil.cpp
#include "Ventil.h"
VENTIL::VENTIL(float vp, float adm, NOD *in, NOD *ut)
{
   this->vp = vp;
   this->adm = adm;
   this->in = in;
   this->ut = ut;
   flode = 0.0;
}
void VENTIL::dynamik()
{
   float dp = in->get_tryck() - ut->get_tryck();
   if (dp >= 0.0)
   {
      flode = vp * adm * sqrt(dp);
   }
   else          OBS!                        OBS!
   {
      flode = - vp * adm * sqrt(-dp);
   }                          OBS!
   in-> add-summaflode(-flode);  //OBS!  } tecken!
   ut-> add-summaflode(flode);   //OBS!
}
```

Reglering?



(Ex) En bil har en motor och en ägare.

Består av relation eller aggregat. Motorn skapas då bilen skapas.

Känner till relation eller association. Ägaren finns redan då bilen skapas.

```
//Spec --Motor.h
class MOTOR
{   private:
        float effekt;
        int antal_cyl;
    public:
        MOTOR(float effekt, int antal_cyl);
        void skriv();
};

//Impl -- Motor.cpp
#include "Motor.h"
#include <iosheam>
using namespace std;
MOTOR :: MOTOR (float effekt, int antal_cyl)
{   this->effekt = effekt;
    this->antal_cyl = antal_cyl;
}
void MOTOR::skriv()
{ cout << "Effekt : " << effekt << endl;
  cout << "Antal cylindrar : " << antal_cyl << endl.
}
```

```
// Spec -- Agare.h
class AGARE
{   private:
        char namn[30];
        char pnr[12];
    public:
        AGARE(char *namn, char *pnr);
        void skriv();
};

// Impl -- Agare.cpp
#include "Agare.h"
#include <iostream>
using namespace std;
#include <cstring>

AGARE::AGARE(char *namn, char *pnr)
{   strcpy(this->namn, namn);
    strcpy(this->pnr, pnr);
}
void AGARE :: skriv()
{       cout << "Namn: " << namn << endl;
        cout << "Personnr: " << pnr << endl;
}
```

```
//Spec -- Bil.h
#include "Motor.h"
#include "Agare.h"
class BIL
{   private:
      char regnr[7];
      MOTOR m;      //Inre objekt Löser aggregat
      AGARE *ap;    // Pekare löser association
   public:
      BIL (char *regnr, float effekt, int antal-cyl, AGARE *ap);
      void skriv();
};

//Impl -- Bil.cpp
#include "Bil.h"
#include <cstring>
#include <iostream>
using namespace std;
BIL::BIL(char *regnr, float effekt, int antal-cyl, AGARE*ap)
        : m(effekt, antal-cyl) //initieringslista, m skapas
{    strcpy(this->regnr, regnr);
     this->ap = ap;
}
void BIL::skriv()
{    cout << "Regnr:" << regnr << endl;
     m.skriv();
     ap->skriv();
}
```

Alternativt måste motorn och regnr allokeras dynamiskt om man exempelvis använda gc-klasser i managed extension i Visual.

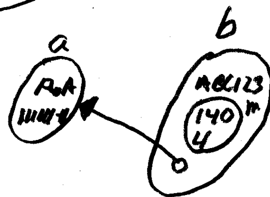
```
class BIL
{
   private:
     char *regnr;
     MOTOR *mp;
     AGARE *ap;
   public:
     BIL(char *regnr, float effekt, int antal_cyl, AGARE *ap);
     void skriv();
};

BIL::BIL(char *regnr, float effekt, int antal_cyl, AGARE *ap)
{
     this->regnr = new char[7];
     strcpy(this->regnr, regnr);
     mp = new MOTOR(effekt, antal_cyl); // Motorn skapas fortfarande här!
     this->ap = ap;
}

void BIL::skriv()
{
     cout << "Regnr: " << regnr << endl;
     mp->skriv();
     ap->skriv();
}
```

//Huvud – Bilmain.cpp

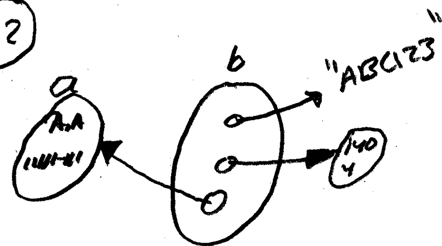
```
#include "Bil.h"
void main()
{
    AGARE a("A.A", "11111-1111");
    BIL b("ABC123", 140, 4, &a);

    b.skriv();

}
```

Fungerar för båda alternativen!!

(Alt 1)



(Alt 2)

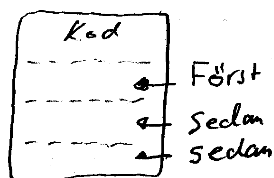


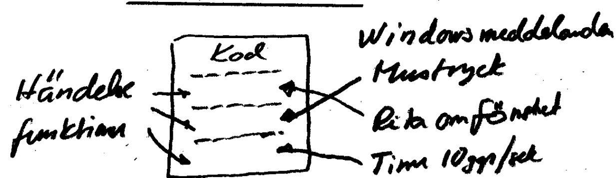
<u>Hemuppgift:</u> Rita ett UML-digram som visar klassdiagram för programet Bilmain ovan.

Windowsprogram

Program som körs i Windows är händelsestyrda till skillnad från program som körs i console som är programstyrda.

Programstyrd | **Händelsestyrd**





(Ex) Kylsim – Testsystemet består av 3 noder och 2 ventiler

```
// spec – Form1.h
__gc class Form1 : public Form
{   private:
      NOD *n1p, *n2p, *n3p;
      VENTIL *v1p, *v2p, *v3p;
      Graphics *canvas; // Rityta
   public:
      Form1()
      {
         canvas = CreateGraphics();
         n1p = new NOD(canvas, 5.0, false, "N1", 100, 400);
         n2p = new NOD(canvas, 1.0, true, "N2", 200, 400);
         n3p = new NOD(canva, 1.0, false, "N3", 300, 400);
         v1p = new VENTIL(canva, 1.0, 10, 150, 400, "V1", n1p, n2p);
         v2p = new VENTIL(canva, 1.0, 10, 250, 400, "V2", n2p, n3p);
      }
```

Mycket fås färdigt när man använder de moderna verktygen för Windowsprogramming. Det som man måste skriva själv är händelsefunktionerna som ska ta hand om Windows-meddelanden.

```
// Händelsefunktion för meddelandet att kylsystemet
// ska ritas om
void Form1_Paint(Objekt *sender, Eventargs *e)
{
        n1p->rita();
        n2p->rita();
        n3p->rita();
        v1p->rita();
        v2p->rita();
}
// Händelsefunktion för timermeddelande
void Form1Clock_tick (Objekt *sender, Eventargs *e)
{
        n1p->dynamik(); n1p->display();
        n2p->dynamik(); n2p->display();
        n3p->dynamik(); n3p->display();
        v1p->dynamik(); v1p->display();
        v2p->dynamik(); v2p->display();
}
```