3 Klasser och objekt

Med *klassbegreppet* kan man definiera sina egna abstrakta datatyper i C++. Man kan bygga ut språket med egna typer (klasser), utöver de vanliga int, char etc. *Objekten*, med vars hjälp man bygger upp sitt program, är variabler (instanser, förekomster) av dessa klasser. Klassen är en *mall* för hur objekten ska se ut *både vad gäller data och operationer*. Objekten och *samspelet* mellan dessa ska utgöra en *modell av verkligheten och den funktionalitet som man vill ha i sitt system.*

I klassen ska man ange de *data*, som man vill hålla reda på för sina objekt samt de *operationer* som en användare av objektet behöver. Operationerna ska vara på så hög nivå som möjligt så att den interna datastrukturen i klassen *göms* så mycket som möjligt (datagömning eller informationhiding). På detta sätt blir objekten säkrare och också lättare för tillverkaren att ändra i. En tillverkare av klasser kan exempelvis optimera implementationen utan problem, bara gränssnittet mot användaren finns kvar.

Ex : Tittar vi på våra ventiler i kylanläggningen skulle vi kunna definiera en klass VENTIL som utgör en typ för ventilobjekten enligt :

```
// Specifikation av klassen VENTIL -- ventil.h

class VENTIL
{
   private :
      float vp;         // ventilkäglans position mellan 0 och 1
   public:
      void open();      // öppna ventilen
      void close();     // stäng ventilen
};                      // OBS! Semikolonet
```

Specifikationen av en klass består av *privata (private)* och *allmänna (public) medlemmar* . Vilka man skriver först kan vara en smaksak. *Privata medlemmar* kan *ej* nås utifrån utan det är bara de *allmänna medlemmarna* som en användare utifrån kan utnyttja. Via de allmänna medlemsfunktionerna kan man naturligtvis komma åt att avläsa eller ändra värden på de privata medlemsvariablerna.

Medlemmarna kan i sin tur vara både *medlemsvariabler* (medlemsobjekt), som vp ovan eller *medlemsfunktioner* , som open och close. Det är inget som hindrar att man har privata medlemsfunktioner eller allmänna variabler. Allmänna (globala) variabler *bör dock användas sparsamt*.

Som tillverkare av en klass måste man också implementera kropparna till sina funktioner.

```
// Implementation av klassen VENTIL -- ventil.cpp

#include "ventil.h"

void VENTIL::open()
{
   vp = 1;
}


void VENTIL::close()
{
   vp = 0;
}
```

Definitionen av en *medlemsfunktion* skiljer sig från definitionen av en vanlig funktion på så sätt att funktionsnamnet föregås av *klassnamnet och global-tecknet :: (två kolon)*.

När man skriver sina medlemsfunktioner *har man direkt tillgång* till klassens privata medlemsvariabler. Ska man använda objekt av klassen VENTIL i ett program måste man dock gå *via objektets publika medlemsfunktioner,* för att komma åt de privata medlemmarna.

```
// Huvudprogram -- ventilma.cpp

#include "ventil.h"

void main()
{
   VENTIL v1;

   .....
   v1.open();     // öppna ventilen
   .....
   v1.vp = 0;     // OBS! Kompileringsfel, vp privat !!
   .....
   v1.close();    // stäng ventilen
}
```

Man kan ha alla tre delarna specifikation, implementation och huvudprogram i samma fil, men det bästa vid större system är att dela upp sitt program i flera filer, som ovan. Har man allt i samma fil måste naturligtvis ordningen vara densamma som ovan.

I huvudprogrammet kan man naturligtvis definiera flera objekt (variabler) av klassen VENTIL. Man kan arbeta med dessa objekt precis som med vanliga variabler. Vilka operationer som är möjliga bestäms av vad som är implementerat i klassen. Vissa operationer är implementerade *implicit*, dvs de finns redan. De operationer som är fördefinierade för alla objekt är *initiering* och *tilldelning*.

Huvudprogrammet till ovanstående exempel kunde exempelvis ha sett ut som :

```
#include "ventil.h"

void main()
{
   VENTIL v1, v2;


   .....

   v1.open();           // öppna ventilen v1
   VENTIL v3 = v1;      // initiering av v3 med v1, båda har vp=1

   .....

   v2.close();          // stäng ventilen

   .....

   v1 = v2; // tilldelning av v2 till v1, båda har vp=0

}
```

OBS! De *fördeklarerade* operationerna *initiering* och *tilldelning* utför en kopiering av alla datadelarna i det högra objektet till det vänstra objektet. Ibland vill man kanske ej ha kopiering av all data och då får man definiera egna initierings- och tilldelningsoperator, som överlagrar default-operatorerna (se nedan).

Varje objekt har *sina egna värden på data-delarna* och för att förändra, visa, testa etc. dessa data använder man de allmänna operationerna, som är *gemensamma för hela klassen*. Vilket objekt man ska operera på anges med *punktnotation* framför operationen. Man brukar också säga att man skickar ett meddelande till ett objekt att göra en viss operation.

Varje definierat objekt innehåller en pekare (adress), som har namnet *this*, till sig själv. Vid anrop av medlemsfunktioner med punktnotation skickas denna adress, implicit, till den anropade medlemsfunktionen, så att den vet vilket objekts data den ska operera på.

Då medlemsfunktionerna är så korta som ovan kan det vara lämpligt att skriva dessa *inline*. Det enklaste sättet att få dessa inline är att skriva *funktionskropparna direkt i specifikationen* av klassen. Gör man ej detta måste man på vanligt sätt markera inline först i funktionshuvudet, antingen i deklarationen eller i definitionen.

Ex : Här följer två alternativ för att få funktionerna i klassen VENTIL inline.

a)
```
// specifikation av klassen VENTIL -- ventil.h

class VENTIL
{
   private:
      float vp;
   public:
      void open(){ vp = 1; }
      void close(){ vp = 0; }
};
```

b)
```
// specifikation av klassen VENTIL -- ventil.h

class VENTIL
{
   private:
      float vp;
   public:
      void open();
      void close();
};

inline void VENTIL::open()
{
   vp = 1;
}

inline void VENTIL::close()
{
   vp = 0;
}
```

OBS! Man måste implementera inline-funktionerna i header-filen så att funktionernas kod kan stoppas in vid anropen under kompileringen. Alternativt kan man istället för att markera inline framför funktionsdefinitionerna göra detta framför funktionsdeklarationerna.

3.1 Konstruktor och destruktor

De egendefinierade objekten ska likna de fördefinierade så mycket som möjligt. När man definierar en reell variabel så kan man i C och C++ ge den startvärdet direkt vid definitionen. Samma sak skulle man vilja göra för sina objekt. Man kan naturligtvis bland sina medlemsfunktioner i klassen ha en speciell funktion som initierar objekten.

Ex : Skriv om klassen VENTIL så att den innehåller en funktion med vars hjälp man kan ge ett startvärde till admittansen (genomsläppligheten).

```
// specifikation av klassen VENTIL -- ventil.h

class VENTIL
{
   private:
      float vp;
      float adm;  // OBS! Här får man ej ge startvärden
   public:
      void init(float adm);
      void open();
      void close();
};


// implementation av klassen VENTIL -- ventil.cpp

#include "ventil.h"

void VENTIL::init(float adm)
{
   this->adm = adm;  // OBS! this->adm för att skilja medlemsvariabeln

                     // adm från den lokala formella parametern adm.
}


.....

.....


// huvudprogram -- ventilma.cpp

#include "ventil.h"

void main()
{
   VENTIL v1;

   v1.init(10);
   .....
}
```

Att använda en separat operation för initiering *liknar dock ej den normala initieringen* som man kan göra i samband med en vanlig variabeldefinition som exempelvis :

```
.....
float x = 2.3;
.....
```

I C++ har man därför infört en speciell medlemsfunktion, *konstruktor*, som *alltid anropas då ett nytt objekt skapas* och en annan speciell medlemsfunktion, *destruktor*, som *alltid anropas då ett objekt förstörs*. Man kan ha flera överlagrade konstruktorer som skiljs åt av sina parametrar. Destruktorn kan ej ta några parametrar och kan alltså ej överlagras.

Ex : Skriv om klassen VENTIL så att den innehåller en konstruktor utan parameter som sätter admittansen till 10 och en med parameter i form av startvärde för admittansen samt en destruktor som inte gör någonting.

```
// specifikation av klassen VENTIL -- ventil.h

class VENTIL
{
   private:
      float vp;
      float adm;
   public:
      VENTIL();               // konstruktor 1
      VENTIL(float adm);      // konstruktor 2
      ~VENTIL();              // destruktor
      void open();
      void close();
};


// implementation av klassen VENTIL -- ventil.cpp

#include "ventil.h"

VENTIL::VENTIL()
{
   adm = 10;
}

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

VENTIL::~VENTIL()
{
}
.....
```

```
// huvudprogram -- ventilma.cpp

#include "ventil.h"

void main()
{
   VENTIL v1;           // konstruktor 1 anropas, adm = 10
   VENTIL v2(15);       // konstruktor 2 anropas ,adm = 15
   VENTIL v3 = 20;      // konstruktor 2 anropas ,adm = 20

   v1.open();
   v2.open();

   .....

}                       // destruktorn för v3, v2 och v1 anropas
```

OBS! Konstruktorerna och destruktorn *saknar returvärden* (inte ens void).

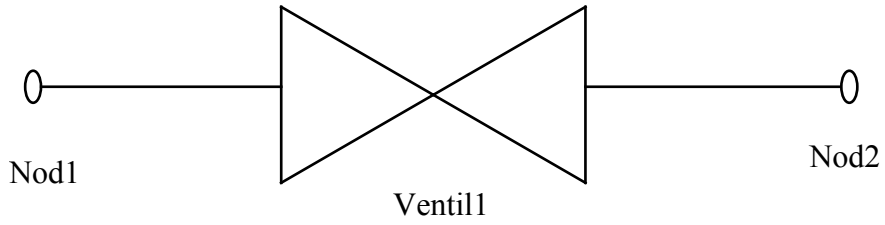
OBS! Har konstruktorn *bara ett argument* kan konstruktorn anropas precis på samma sätt som vid exempelvis initiering av en float-variabeln ovan med tilldelningstecken.

OBS! Konstruktorer kan ha defaultvärden på sina argument precis som vilken annan funktion som helst. Istället för att ha två olika konstruktorer som i exemplet ovan hade man kunnat använda en enda konstruktor :

```
.....

VENTIL(float adm = 10);

.....


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

......
```

Ett anrop av konstruktorn utan parameter ger adm = 10.

Ex : I kylsystemet har varje ventil förbindelse med två noder. Flödet i ventilen bestäms av bland annat tryckskillnaden mellan dessa noder.



Ett första förslag på NOD-klass blir då, eftersom vi är intresserade av trycket i noderna:

```
// specifikation av klassen NOD -- nod.h

class NOD
{
   private:
      float p;
   public:
      NOD(float p){ this->p = p; }
      ~NOD(){}
      float get_p(){ return p; }
};
```

Nu kan VENTIL-klassen kompletteras med anslutningar och början till en dynamik-funktion som beräknar flödet genom ventilen. De aktuella noderna som ansluter till ventilerna kommer in via pekare i VENTIL-klassen.

```
// specifikation av klassen VENTIL -- ventil.h

#include "nod.h"

class VENTIL
{
   private:
      float vp;
      float adm;
      float f;          // Flöde
      NOD *in;          // In-nod
      NOD *ut;          // Ut-nod
   public:
      VENTIL(float adm, NOD *in, NOD *ut);
      ~VENTIL();
      void open();
      void close();
      void dynamik();   // Flödesberäkningsfunktion
};
```

```
// implementation av klassen VENTIL -- ventil.cpp

#include "ventil.h"
#include <cmath>

VENTIL::VENTIL(float adm, NOD *in, NOD *ut)
{
   this->adm = adm;
   this->in = in;
   this->ut = ut;
}

.....


void VENTIL::dynamik()
{
   float dp;

   // beräkna tryckskillnad
   dp = in->get_p() - ut->get_p();

   // beräkna flödet
   if ( dp >= 0 )
      f = vp * adm * sqrt(dp);
   else
      f = - vp * adm * sqrt(-dp);
}
```

Ett huvudprogram som skapar objekten nod1, nod2 och ventil1 enligt ovan och som anropar dynamikfunktionen för ventil1 ser då ut enligt :

```
// huvudprogram till kylsystem -- kylsysma.cpp

#include "ventil.h"

void main()
{
   NOD nod1(2), nod2(1.5);
   VENTIL ventil1(20, &nod1, &nod2);

   ventil1.open();
   ventil1.dynamik();
}
```

En konstruktor anropas *alltid då ett objekt skapas*, oberoende av hur detta sker, globalt, lokalt, statiskt, inuti en klass som medlemsobjekt, dynamiskt etc. Man använder inte konstruktorer enbart till att ge startvärden utan också till att *'duka bordet'* för att kunna använda objektet.

Destruktorn anropas *alltid då ett objekt förstörs*. I destruktorn ska man *'städa bordet'* efter det att man använt objektet.

Har man dynamiskt allokerade medlemsvariabler i sina objekt ska konstruktorn förutom alla tilldelningar av värden också allokera minne och destruktorn ska se till att minnet avallokeras.

Ex : En medlem i en förening, med medlemsnummer och namn, kan avbildas med en klass enligt:

```
// specifikation av klassen MEDLEM -- medlem.h


class MEDLEM
{
   private :
      int mnr;          // medlemsnummer
      char *namn;       // pekare till dynamiskt namn
   public :
      MEDLEM(int mnr = 0, char *namn = "");    // konstruktor
      ~MEDLEM();                               // destruktor
      . . . .
};



// implementation av klassen MEDLEM -- medlem.cpp


#include "medlem.h"
#include <cstring>
#include <cstdlib>

MEDLEM::MEDLEM(int mnr, char *namn)
{
   this->mnr = mnr;
   this->namn = new char[strlen(namn) + 1];
   strcpy(this->namn, namn);
}


MEDLEM::~MEDLEM()
{
   delete [] namn;
   namn = NULL;
}
```

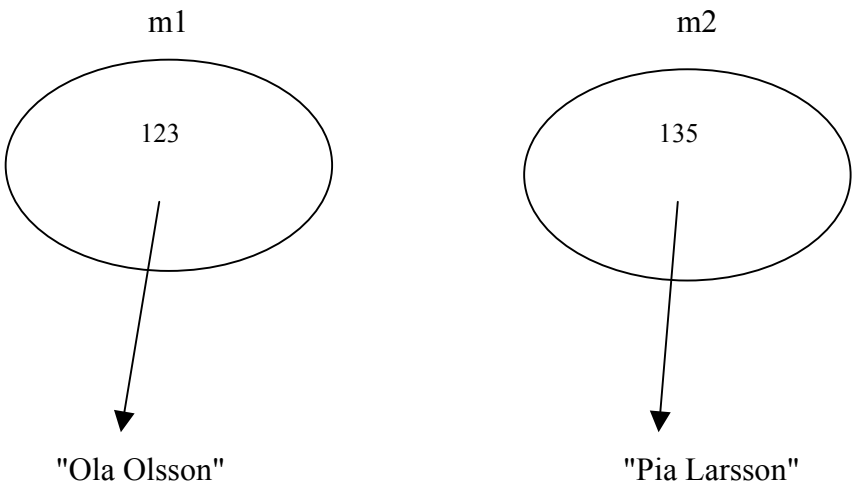
I konstruktorn allokerar man dynamiskt så mycket minne som det aktuella namnet, som kommer in som parameter, kräver.

I destruktorn avallokeras det minne som konstruktorn allokerat.

Ett anropande huvudprogram kan sedan skapa ett antal medlemsobjekt av klassen enligt exempelvis:

```
#include "medlem.h"

void main()
{
   MEDLEM m1(123, "Ola Olsson"), m2(135, "Pia Larsson");

   . . . .

}     // Här anropas destruktorn
```

Programmet skapar två objekt enligt:



Konstruktorn allokerar minne för namnet i objektet och destruktorn avallokerar motsvarande minne.

Har man inte det här med konstruktor och destruktor klart för sig, kan ett C++-program bli mycket svårbegripligt. Det kan finnas en hel mängd *funktionalitet inbakad i konstruktorn resp destruktorn*, som man alltså ej har något explicit anrop till. Konstruktorn för objekt som man definierar på global nivå körs exempelvis redan *innan* första satsen i main-funktionen, vilket kan förbrylla.

Ex: I C++ görs filhantering via så kallade strömmar. När man öppnar en fil för läsning skapar man ett objekt av klassen ifstream och klassens konstruktor körs. Konstruktorn och destruktorn för ifstream skulle kunna vara implementerade enligt :

```
// specifikation av klassen ifstream -- ifstream.h
class ifstream
{
   public:
      ifstream(char *filename);
      ~ifstream();
      int notok();
      .....
   private:
      FILE *file;
      ....
};

// implementation av klassen ifstream -- ifstream.cpp
ifstream::ifstream(char *filename)
{
   file = fopen(filename, "r");
}

ifstream::~ifstream()
{
   fclose(file);
}

int ifstream::notok()
{
   return (file == NULL);
}


// huvudprogram
void main()
{
   ifstream f("namn.txt");

   if (f.notok())
   {
      cerr << "Filen kan ej öppnas" << endl;
      exit(1);
   }
   .....
}
```

OBS! Konstruktorn, som ej kan ha något retur-värde, kan ej meddela om något gått snett när filen ska öppnas, exempelvis att filen ej finns. Man måste istället ha en *extra* medlemsfunktion notok för att kontrollera detta. I verkliga fstream är notok skriven som en överlagrad operator ! och anropas som !f (se nedan).

Ex : Programmet utgör en telefonkatalog. Man matar in en teckensträng från tangentbordet som programmet söker efter i filen telnr.txt. De rader som innehåller den sökta strängen skrivs ut. Programmet är oberoende av om man ger små eller stora bokstäver.

```
// program för sökning i telefonkatalog -- tele.cpp

#include <fstream>      //inkluderar även iostream.h
#include <cctype>
#include <cstring>

using namespace std;

// omvandlar ch till stor bokstav(även å, ä och ö)
char versal(char ch)
{
   if (ch == 'å')
      return 'Å';
   if (ch == 'ä')
      return 'Ä';
   if (ch == 'ö')
      return 'Ö';
   return toupper(ch);
}

// jämför nr st tecken i xstr med ystr
int ustrncmp(char *xstr, char *ystr, int nr)
{
   int i;
   while (nr--)
      if ( (i = versal(*xstr++) - versal(*ystr++)) != 0 )
         return i;
   return 0;
}

// sök i filen name efter strängen str
int grep(char *str, char *name)
{
   char *cp;
   char buf[256];
   ifstream fil(name);
   // kontrollera att filen ok
   if ( !fil )
   {
      cerr << "Kan inte öppna filen !" << endl;
      return 0;
   }
   // för alla rader i filen
   while ( !fil.eof() )
   {
      fil.getline(buf, sizeof(buf));
      cp = buf;
      // sök efter str på raden
      while ( *cp )
         if ( (ustrncmp(str, cp++, strlen(str)) == 0) )
         {
            // bingo, skriv ut raden
            cout << buf << endl;
            break;
         }
   }
   return 1;
}
```

```
// huvudprogram
void main()
{
   char vem[80];

   while ( 1 )
   {
      cout << endl << "Vem söks > ";
      cin.getline(vem, sizeof(vem));
      if ( grep(vem, "telkat.txt") == 0 )
         return;  // om filen ej ok avsluta programmet
   }
}
```

Antag att du har en textfil telkat.txt enligt :

Lektor Hans Rapp 123456
Adjunkt Doris Pers 234567
Frida Pers 345678
Ulla och Kalle Olsson 678901
Conny Äreberg 345234

Programmet gör då följande om man skriver in det understrukna:

Vem söks > doris pers
Adjunkt Doris Pers 234567

Vem söks > kadkslsd kakal

Vem söks > pers
Adjunkt Doris Pers 234567
Frida Pers 345678

Vem söks > RETURN (tom sträng)
Lektor Hans Rapp 123456
Adjunkt Doris Pers 234567
Frida Pers 345678
Ulla och Kalle Olsson 678901
Conny Äreberg 345234

Vem söks > Conny äreBERG
Conny Äreberg 345234

OBS! Objektdefinitionen ifstream fil(name) i grep som anropar en *konstruktor* i klassen ifstream. Konstuktorn öppnar filen. Destruktorn, som stänger filen, anropas automatiskt då grep returnerar.

OBS! Anropet !fil i grep motsvarar funktionsanropet fil.notok() i den tänkta implementeringen av ifstream ovan. Not-operatorn är *överlagrad* i ifstream och ska returnera true om filen ej ok (se nedan om operatoröverlagring).

OBS! Läsning av strängar med getline-funktionen som finns i klassen istream. Denna funktion läser fram till avslutningstecknet, som default är radslut men kan anges som tredje parameter, eller så länge strängen ej överskrider storleken av den deklarerade strängen. Avslutningstecknet *placeras ej i strängen men tas bort från bufferten*.

Definierar man inga egna konstruktorer eller ingen egen destruktor finns det en konstruktor utan parameter och en destruktor *fördefinierade* av kompilatorn, som inte gör någonting alls. Har man definierat någon egen konstruktor eller destruktor själv tillhandahåller ej kompilatorn några konstruktorer eller någon destruktor alls, utan man måste stå för alla dessa själv.

REGEL : Varje klass av objekt ska alltid ha åtminstone en konstruktor.

Varje klass av objekt ska ha en destruktor.

Definierar man ingen egen konstruktor finns en fördefinierad som saknar parametrar och som inte gör någonting.

Definierar man ingen egen destruktor finns en fördefinierad som inte gör någonting.

3.2 Olika sätt att skapa objekt

Det finns olika sätt att skapa objekt. I samtliga fall kommer objektets konstruktor att anropas vid *skapandet* och dess destruktor att anropas då objektet *förstörs*. *Definierar man en vektor av objekt kommer konstruktorn att anropas för varje element i vektorn precis som destruktorn*. Man brukar prata om lokala objekt, globala objekt, statiska objekt, dynamiska objekt och medlemsobjekt.

3.2.1 Lokala objekt

Ett objekt som skapas *inuti* en funktion blir ett lokalt objekt. Objektet skapas och konstruktorn anropas då funktionen anropas. Objektet förstörs och destruktorn anropas då man nått slutet av funktionen eller då funktionen når ett return.

I exemplet telefonkatalog ovan är fil ett lokalt objekt. Objektet fil skapas och konstruktorn anropas på raden

```
ifstream fil(name);
```

Objektet fil förstörs och destruktorn anropas vid avslutande

```
return 0;
```

eller

```
return 1;
```

3.2.2 Globala objekt

Ett globalt objekt är ett objekt som är definierat på yttersta nivån, dvs utanför alla funktioner, även main. Objektet är känt över hela systemet. Alla funktioner kan använda det. Detta bör användas restriktivt.

Det kan vara acceptabelt att ha en global ström om flera olika funktioner ska komma åt att läsa från strömmen eller skriva till strömmen. Annars får man släpa på en extra parameter till alla funktioner som utnyttjar strömmen. Den globala strömmen definieras före main enligt :

```
ofstream tsut("utfil.txt");

void main()
{
   .....
}
```

Koden för konstruktorn körs i detta fall redan innan första satsen i main körs. Koden för destruktorn körs sist av allt.

3.2.3 Statiska objekt

Ett statiskt objekt deklareras inuti en funktion. Konstruktorn körs bara *första* gången funktionen anropas. Objektet placeras i globalt minne men är bara känt inom den funktion som den är definierad i. Objektet blir kvar tills programmet avslutas. Destruktorn körs alltså bara en gång allra sist i programmet.

Ex : Antag att du vill hålla reda på hur många gånger en funktion anropas. Då kan du i funktionen ha en räknare som en statisk variabel enligt :

```
#include <iostream>

using namespace std;

void statictest()
{
   static int nr = 1;

   cout << nr << " ";
   nr++;
}


void main()
{
   for (int i = 1; i <= 10; i++)
      statictest();
}
```

Här skrivs 1 2 3 4 5 6 7 8 9 10 ut.
Tar man bort static framför nr blir utskriften istället 1 1 1 1 1 1 1 1 1 1.

I ovanstående exempel har vi en vanlig variabel som ett statiskt objekt. Men det blir ingen skillnad om det skulle vara ett objekt av någon klass. Konstruktorn (initieringen) körs bara en gång för statiska objekt och detta är vid första anropet. Vid övriga anrop kommer objektet ihåg sitt föregående värde.

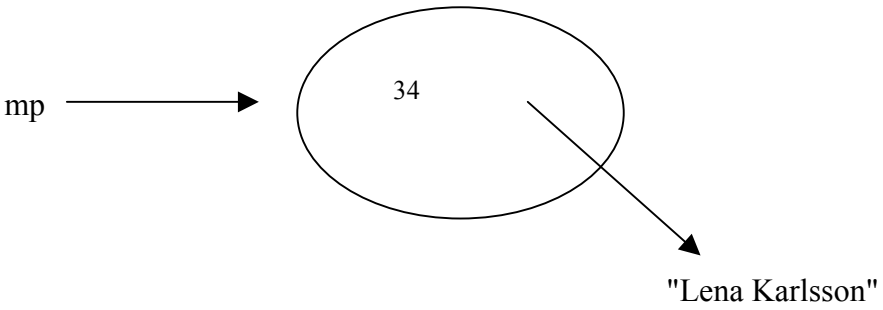
3.2.4 Dynamiska objekt

Ett dynamiskt objekt är ett objekt som skapas med operatorn *new* då man behöver det och tas bort med operatorn *delete* då man ej behöver det längre.

Ex Man kan skapa dynamiska objekt av klassen MEDLEM ovan enligt:

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

void main()
{
   MEDLEM *mp;    // pekare till medlemsobjekt
   int mnr;
   char namn[30];

   cout << "Medlemsnummer ? ";
   cin >> mnr;
   cin.ignore();

   cout << "Medlemmens namn ? ";
   cin.getline(namn, sizeof(namn));

   // skapa dynamiskt ett medlemsobjekt
   mp = new MEDLEM(mnr, namn);

   .....

   // ta bort medlemsobjektet
   delete mp;

}
```

Det objekt som man skapade ovan är anonymt, men man kommer åt det med piloperatorn enligt:



OBS! När man ska skapa något dynamiskt måste man först definiera en pekare och därefter kan man skapa objektet när man vill.

OBS! Konstruktorn till medlemsobjektet anropas vid new-anropet då man också *överför parametrar* till konstruktorn. Destruktorn anropas i samband med delete.

3.2.5 Inre objekt

Medlemsdata i en klass kan förutom vanliga variabler också vara objekt av någon annan klass. Man brukar kalla sådana objekt för inre objekt. Ett problem som uppstår, om man har inre objekt, är hur man ska överföra eventuella parametrar till de inre objektens konstruktorer. Här måste man ta in parametrarna *via det yttre objektens konstruktorer och överföra dessa till de inre eller också använda sig av konstanter.*

Ex: Vårt kylsystem kommer att bestå av ett antal noder och ventiler plus en hel del annat. Noderna och ventilerna kommer då att vara inre objekt i klassen kylsystem.

```
// Specifikation av klassen KYLSYSTEM -- Kylsystem.h

#include "Nod.h"
#include "Ventil.h"


class KYLSYSTEM
{
   private :
      NOD n1, n2, n3;
      VENTIL v1, v2;
      . . . .

   public:

      KYLSYSTEM();
      . . . .
};
```

Det bästa sättet är att initiera de inre objekten i initieringslistan så slipper man att skapa ett temporärt objekt som anropar default-konstruktorn i nod resp ventil-klassen.

```
// Implementation av klassen KYLSYSTEM -- Kylsystem.cpp


#include "Kylsystem.h"

KYLSYSTEM::KYLSYSTEM() : // Initieringslista

                         n1(5, 0, "N1", 100, 400),
                         n2(0, 1, "N2", 300, 400),
                         n3(1, 0, "N3", 500, 400),
                         v1(10, 1, "V1", 200, 400, &n1, &n2),
                         v2(10, 1, "V2", 400, 400, &n2, &n3),
                         . . . . . .



{
}
```

OBS! Parametrarna till konstruktorerna i nod- och ventil-klassen utgörs i detta fall av konstanter men observera att noderna n1, n2 och n3 måste vara definierade innan v1 och v2, eftersom nodernas adresser ska skickas som parametrar till ventil-konstruktorn.

Alternativt kan man istället för automatisk allokering av inre objekt använda dynamisk allokering. Vid dynamisk allokering definierar man först en pekare till klassen och sedan allokerar man med new i konstruktorn och avallokerar med delete i destruktorn enligt:

Ex:

```
// Kylsystem.h

class KYLSYSTEM
{
   private :
      NOD *np1, *np2, *np3;
      VENTIL *vp1, *vp2;
      . . . .

   public:

      KYLSYSTEM();
      . . . .
};

// Kylsystem.cpp

KYLSYSTEM::KYLSYSTEM()
{
   np1 = new NOD(5, 0, "N1", 100, 400);
   np2 = new NOD(0, 1, "N2", 300, 400),
   np3 = new NOD(1, 0, "N3", 500, 400),
   vp1 = new VENTIL(10, 1, "V1", 200, 400, np1, np2),
   vp2 = new VENTIL(10, 1, "V2", 400, 400, np2, np3),
   . . . . .

}

KYLSYSTEM::~KYLSYSTEM()
{
   delete vp1;
   delete vp2;
   delete np1;
   delete np2;
   delete np3;
   . . . .
}
```

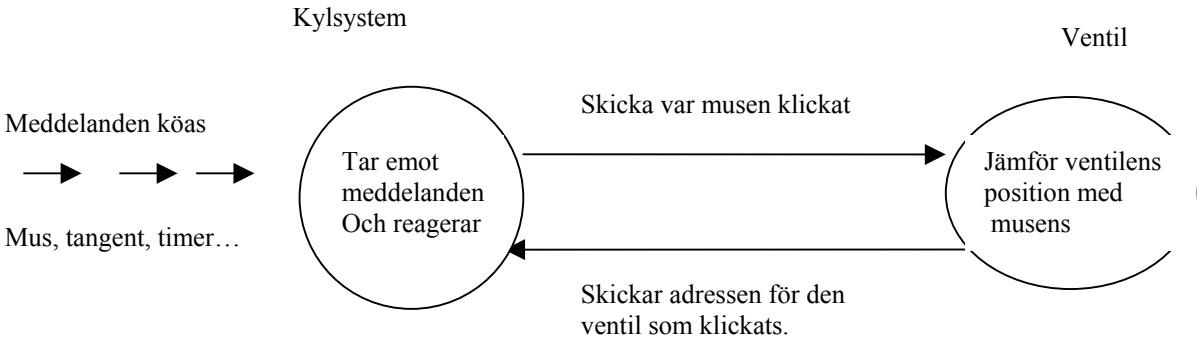
Man kan också ha *statiska medlemmar* i sina klasser. Dessa markeras med static framför och måste initieras globalt eftersom de bara finns i en upplaga för varje klass. Exempelvis kan man i klassen NOD ha en räknare som räknar hur många noder som finns i kylsystemet och en funktion som returnerar antalet noder. Man anropar sedan funktionen inte med ett objekt utan med klassnamnet NOD::antal_noder(). På grund av detta brukar statiska medlemsvariabler även kallas för klassvariabler och statiska medlemsfunktioner för klassfunktioner.

3.3 Anrop och relationer mellan objekt

Ett objektorienterat system byggs upp av ett antal objekt som anropar varandras medlemsfunktioner. Man brukar prata om en *objektsoppa* där *anrop mellan objekten* bygger upp *funktionaliteten* i systemet.

Ett objektorienterat program startar i huvudprogrammet (huvudobjektet) genom att anropa något objekt som man har *definierat där* globalt, lokalt eller dynamiskt. Fortsättningsvis kan medlemsfunktioner i detta anropade objekt anropa ett nytt objekt, som medlemsfunktionen antingen har definierat själv lokalt eller som finns globalt eller som ingår som *medlem* (direkt eller som pekare) i klassen. På detta sätt fortsätter systemet att lösa sina uppgifter genom att objekten samarbetar (soppan kokar).

Ex : Vårt kylsystem är ett Windowsprogram och sådana program är händelsestyrda. Med detta menas att vårt system ska ta hand om ett antal meddelanden som Windows skickar ex. musklick, tangenttryck och bearbeta dessa i händelsefunktioner. Klickar man på en ventil i vårt kylsystem, skickar Windows ett meddelande till vårt program som anger vilken musknapp man klickat med och i vilken position. Detta meddelande ska vårt program ta hand om och reagera på enligt exempelvis :



Kylsystemet, som är huvudobjektet, tar hand om olika meddelanden i sina medlemsfunktioner, så kallade händelsefunktioner och styr med dessa resten av körningen. Händelsefunktionen som tar hand om musklick anropar en klickfunktion i ventil-klassen och skickar med information om var musen klickats. Varje ventil vet sin position och kan på detta sätt skicka tillbaks ja (ventilens adress) eller nej (NULL) alltså om den klickats eller inte.

Ett objektorienterat system byggs upp genom att objekten *använder varandras kunskaper* i form av anrop till varandras medlemsfunktioner. Det hela går ut på att man frågar varandra tills man löst problemet. Man brukar säga att ett objektorienterat system är ett *delegerat* system. Vet man inte själv frågar man ett objekt som vet. Vilka kan man då fråga? På något sätt måste man ha en relation (förhållande) till det objekt som man ska fråga. Man måste ha en kommunikationskanal till det efterfrågade objektet.

Förutom den vanligaste relationen, när man definierar andra objekt globalt, lokalt eller dynamiskt inuti medlemsfunktionerna, finns det två speciella relationer mellan objekt nämligen, *aggregat*- och *associations*relationer.

3.3.1 Aggregatrelation

Två objekt har en aggregatrelation om det första objektet *består av* det andra genom att det andra är inre objekt eller aggregat (automatiskt eller dynamiskt) i det första. Det andra objektet *skapas* av det första.

Ex : Ventilobjekten skapas av ett Kylsystem-objekt. Här har vi en aggregatrelation mellan ventilobjekt och ett kylsystemobjektet.

```
class KYLSYSTEM
{
   private:
      .....
      VENTIL *vp1, *vp2;
   public:
      .....
};

vp1 = new VENTIL(10, 1, "V1", 200, 400, np1, np2),
```

3.3.2 Associationsrelation

Två objekt står i en associationsrelation till varandra om de *känner till eller refererar till* varandra genom att det första objektet som medlemsvariabel har en pekare (adress) till det andra. Det andra objektet *måste vara skapat innan* det första.

Ex: En ventil känner till sina noder. Här är en typisk associationsrelation. I ventil-klassen har vi pekare till de bägge noder som omger ventilen. Noderna skapas inte av ventilobjektet och är alltså inte aggregat utan ventilerna associerar till noder som redan finns.

```
class VENTIL
{
   private:
      .....
      NOD *in, *ut;
   public:
      .....
};

in->get_p();
```