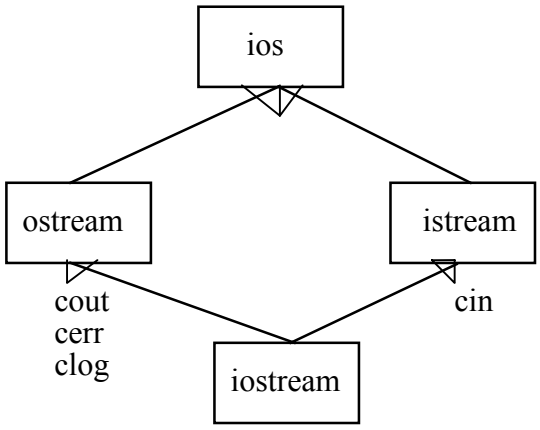
6 In- och utmatning, strömmar

Strömmar används i C för att kommunicera mellan program och yttre enheter som skärm, tangentbord, printer eller sekundärminne. Strömmarna kan man se som *kanaler eller buffertar* som transporterar informationen. I C++ är strömmarna *instansierade objekt* av olika klasser. Det finns *förinstansierade* objekt som cin och cout för att hantera enkel in- och utmatning via tangentbord resp skärm. Det finns klasser för att man ska kunna instansiera egna strömobjekt för att kommunicera med andra enheter.

6.1 Strömmar för enkel in- och utmatning

De fördefinierade objekten cin, cout, cerr och clog tillhör en arvshierarki där det finns multipelt arv enligt följande något förenklade bild :



Objekten cin, cout, cerr och clog finns fördefinierade som globala objekt, om man inkluderar iostream.h.

Objektet cout är en ström mellan program och skärm och cin en ström mellan program och tangentbord. Dessa två objekt har tillstånd, som utskriftsvidd, precision etc och funktioner med vars hjälp man kan transportera data mellan användare och program. För denna transport är det enklast att använda de överlagrade operatorerna för utskrift , << och för inmatning, >>.

Ex :
```
.....
int tal;
.....
cout << "Ge ett tal : ";
cin >> tal;
.....
```

Här anropas funktionen cout.operator<<(char *) för utskrift av strängen och cin.operator>>(int &) för inmatning av tal.

Det finns ett stort antal överlagrade operatorfunktioner av ovanstående typ för *alla de vanliga* enkla variablerna. Man bör också ta till vana att *alltid överlagra* dessa operatorer för sin egendefinierade klasser.

Att använda utskrifts- och inmatningsoperatorerna är det enklaste sättet att göra inmatning och utmatning. Det finns även andra in- och utmatningsfunktioner som exempelvis :

Ex :
```
......
char ch;
......
cin.get(ch);
cout.put(ch);
......
```

Funktionen get läser ett tecken från tangentbordet och put skriver ut detta tecken på skärmen (även blankt).

Ex : Skriv ett program som läser in ett antal heltal (avslutas med enbart RETURN) och skriver ut talens medelvärde.

```
#include <iostream>
using namespace std;

void main()
{
   int  nr = 0;
   float tal, sum = 0;

   cout << "Ge ett tal : ";
   while ( cin.peek() != '\n');
   {
      nr++;
      cin >> tal;
      cin.ignore(80, '\n');
      sum += tal;
      cout << "Ge ett tal : ";
   }
   cout << "Medelvärdet = " << sum / nr << endl;
}
```

Med funktionen cin.peek kontrollerar man nästa tecken i inmatningsbufferten *utan att läsa det*. Funktionsanropet cin.ignore(80, '\n') hoppar över tecken fram till och med första nyrad-tecknet, dock maximalt 80 tecken. Ignore har defaultvärdet 1 och EOF på de två parametrarna, så här hade anropet cin.ignore() varit tillräckligt för att hoppa över ett enda radslutstecken.

Inmatningsoperatorn *läser i fritt format* så länge inmatningen passar ihop med den angivna datatypen. Inläsningen avslutas dock om man hittar något av separeringstecknen *blank, tab eller nyrad* i bufferten. Detta ställer till problem när man ska läsa in strängar som innehåller blanktecken. Det som kommer efter blanktecknet blir kvar i bufferten. Det är bättre att läsa in strängar med funktionen getline enligt :

Ex :

```
#include <iostream>
using namespace std;

void main()
{
   char namn[40];

   cout << "Ge ett namn  : ";
   cin.getline(namn, sizeof(namn), '\n');
   cout << namn << endl;
}
```

Funktionen getline läser här högst 39 tecken från inmatningsbufferten och placerar ett '\0'-tecken sist i strängen namn. Läsningen avslutas dessförinnan om separeringstecknet, som är den tredje parametern, hittas. Separeringstecknet tas bort från bufferten men stoppas ej in i strängen namn. Den tredje parametern kan utelämnas eftersom den har ett defaultvärde i form av '\n'.

Utskriften från ovanstående program blir :

Ge ett namn : Olle Karlsson
Olle Karlsson

Hade man använt cin >> namn istället hade utskriften blivit :

Ge ett namn : Olle Karlsson
Olle

Det finns in- och utmatningsfunktioner som läser in resp skriver ut ett exakt angivet antal byte (tecken) oberoende av blanka, tab, nyrad.

Ex :

```
......

cin.read(namn, 10);
cout.write(namn, 10);
cout.flush();

.....
```

Här läser man in exakt 10 tecken varken mer eller mindre. Utskriften består av 10 tecken. Inget strängsluttecken sätts. Tömning av utmatningsbufferten måste ske explicit med cout.flush eller manipulatorn endl. Read- och write-funktionerna används ofta vid *binär kommunikation* mellan program och sekundärminne(binära filer) men kan också användas, som ovan, vid kommunikation med textfiler.

Tömning av utmatningsbufferten sker med funktionen cout.flush eller med manipulatorn endl. Utmatningsbufferten töms också då utskrift följs av inmatning exempelvis efter ledtext om in- och utmatningskanalerna är ihopbundna, vilket är default. Detta kan ändras med funktionen tie.

Det finns ett antal ytterligare funktioner för att testa eller ändra in- och utmatningsobjektens tillstånd eller status. Dessa kan man utnyttja till att exempelvis få säker inmatning .

Ex : Skriv ett program som läser in reella tal säkert och beräknar medelvärdet av dessa. Avslutning ska ske med bara RETURN.

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

void main()
{
   int nr = 0;
   float tal, sum = 0;

   cout << "Ge ett tal(avsluta med RETURN) : ";
   while ( cin.peek() != '\n' )
   {
      nr++;
      cin >> tal;
      while ( !cin )                    // så länge fel status
      {
         cin.clear();                   // återställ status hos cin
         cin.ignore(80, '\n');          // läs förbi skräpet
         cout << "Inget flyttal!" << endl;
         cout << "Ge nytt tal : ";
         cin >> tal;
      }
      cin.ignore(80, '\n');             // läs förbi ev skräp
      sum += tal;
      cout << "Ge ett tal(avsluta med RETURN) : ";
   }
   cout << "Medel = " << sum/nr << endl;
   getch();
}
```

OBS! Hur man kontrollerar status hos cin med !cin och återställer status med cin.clear. Det finns även andra testfunktioner som cin.fail, cin.good, cin.eof cin.bad etc. Objektet cin kan man kommunicera med genom att använda dess funktioner. Man kan be det att göra vissa saker. Man kan fråga det hur det mår osv. *Detta är objektorienterad programmering*.

6.2 Formatflaggor

När man vill ha annat format på inmatningar och utskrifter än vad som gäller default finns det formateringsoperationer hos objekten som man kan utnyttja.

Ex :

```
....

float x = 1.23456;

....

// default utskrift
cout << x << endl;

// högerjusterat 10 positioner och tre decimaler
cout.setf(ios::fixed, ios::floatfield);
cout.precision(3);
cout.width(10);
cout << x << endl;

// vad gäller nu?
cout << x << endl;

// utfyllnad av blanka med nollor
cout.fill('0');
cout.width(10);
cout << x;
......


Utskriften blir :

1.23456
     1.235
1.235
000001.235
```

OBS! Width-funktionen är den enda formatfunktionen som *inte* ger bestående värde på cout-objektets tillstånd. Alla övriga funktioner måste man själv ställa tillbaka.

Det finns ett antal formateringsflaggor i basklassen ios, som man kan sätta med hjälp av funktionen setf(long set, long alt). *Med det andra argumentet anger man vilka flaggor som ska påverkas och med det första vilka av dessa som ska sättas*. Med funktionen unsetf(long set) nollställs flaggorna. Flaggorna finns definierade i ios-klassen som en egenuppräknad typ enligt :

```
enum {skipws = 0x0001      // strunta i whitespace (blank, tab, nyrad) vid inläsning
      left = 0x0002        // vänsterjustera utskriften
      right = 0x0004       // högerjustera utskriften
      internal  = 0x0008   // vänsterjustera tecken, högerjustera tal (-      123)
      dec = 0x0010         // skriv decimalt, tolka inmatningen som decimaltal
      oct = 0x0020         // skriv oktalt, tolka inmatning oktalt
      hex = 0x0040         // skriv hexadecimalt, tolka inmatning hexadecimalt
      showbase = 0x0080    // skriv inledande 0x för hex och 0 för oct
      showpoint = 0x0100   // skriv alltid decimalpunkt
      uppercase = 0x0200   // versaler för hextal (ABCDEF)
      showpos = 0x0400     // använd plustecken för positiva tal
      scientific = 0x0800  // använd 1.234e2 notation för float
      fixed = 0x1000       // använd 123.4 notation för float
      unitbuf = 0x2000     // flush vid varje skrivning
};
```

Ex : Skriv ett program som läser in ett heltal decimalt och skriver ut talet på hexadecimal form med utskriven bas.

```
#include <iostream>
using namespace std;

void main()
{
   int tal;

   // läs in talet
   cout << "Ge ett tal : "
   cin >> tal;

   // skriv ut talet hexadecimalt
   cout.setf(ios::hex, ios::basefield);
   cout.setf(ios::showbase);
   cout <<  tal;
}
```

OBS! Flaggorna kan anropas antingen som ovan med ios::hex eller med cout.hex eftersom ostream ärver ios och flaggorna är public-definierade.

OBS! Vissa flaggor kan man bara inte sätta utan att ta bort andra. När man sätter hex-flaggan måste man se till att ta bort dec och oct flaggan. Detta gör man genom att ange vilka flaggor som ska påverkas med konstanten basefield = dec | hex | oct och sätta hex. På detta sätt tas dec och oct bort. Det finns också konstanter definierade för adjustfield = left | right | internal och floatfield = fixed | scientific.

6.3 Manipulatorer

Med manipulatorer kan man på ett *enklare sätt sätta formatflaggor* och påverka in- och utmatningsobjekten. Som namnet antyder manipulerar manipulatorerna tillståndet hos objekten. En manipulator kan anges som andra parameter till operatorerna << och >>, som manipulatorn endl enligt :

```
cout << endl;
```

För att *använda manipulatorer med parametrar måste man inkludera iomanip.h* utöver iostream.h. Manipulatorerna är :

dec, oct, hex	sätter resp flagga
ws	sätter skipwhitespace flaggan
endl	radslut skrivs ut och bufferten töms
ends	'\0'-tecknet sätts sist i strängen
flush	tömmer utmatningsbufferten
setfill(char c)	fyller ut med c istället för blanka (jfr funktionen cout.fill)
setw(int w)	sätter fältbredden w (jfr cout.width)
setprecision(int prec)	sätter precisionen prec (jfr cout.precision)
setiosflags(long set)	sätter formatflaggor (jfr cout.setf)
resetiosflags(long set)	nollställer formatflaggor (jfr cout.unsetf)
setbase(int b)	sätter base dec, oct eller hex med 10, 8 eller 16

Ex : Skriv ett program som läser in ett heltal decimalt och skriver ut det hexadecimalt och visar basen. Använd manipulatorer.

```
#include <iostream>
#include <iomanip>
using namespace std;

void main()
{
   int tal;

   // läs in talet
   cout << "Ge ett tal : "
   cin >> tal;

   // skriv ut talet hexadecimalt
   cout << hex << setiosflags(ios::showbase) << tal << endl;

}
```

Alla manipulatorer sätts permanent utom setw som bara gäller nästa utskrift.

Man kan skriva egna manipulatorer. Exempelvis kan man skriva en manipulator som rensar in-bufferten enligt :

```
istream & inflush(istream & in)
{
   in.clear();
   in.ignore(80, '\n');
   return in;
}
```

Denna manipulator-funktion kan man sedan använda i exempelvis den säkra inmatningen ovan enligt :

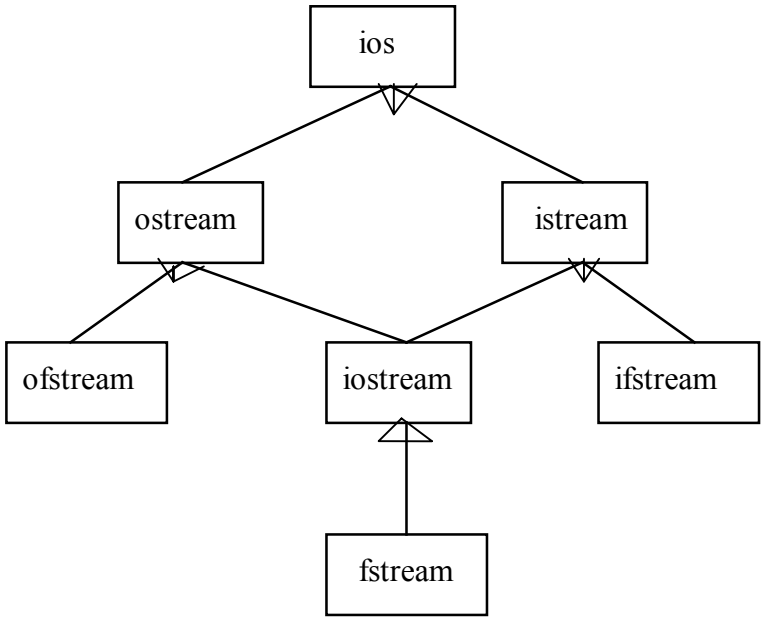
```
#include <iostream>
using namespace std;

void main()
{
   int nr = 0;
   float tal, sum = 0;

   cout << "Ge ett tal(avsluta med RETURN) : ";
   while ( cin.peek() != '\n' )
   {
      nr++;
      cin >> tal;
      while ( !cin )                    // så länge fel status
      {
         cin >> inflush;                // ta bort skräpet
         cout << "Inget flyttal!" << endl;
         cout << "Ge nytt tal : ";
         cin >> tal;
      }
      cin >> inflush;                   // ta bort radslut
      sum += tal;
      cout << "Ge ett tal(avsluta med RETURN) : ";
   }
   cout << "Medel = " << sum/nr << endl;
}
```

6.4 Filhantering

De fördefinierade strömmarna (kanalerna) använder man när man kommunicerar mellan sitt program och tangentbordet eller skärmen. Vill man från sitt program kommunicera med andra enheter, som exempelvis sekundärminne , måste man definiera en egen ström (kanal). Det finns tre olika typer av strömmar nämligen :

in-strömmar som är objekt av klassen ifstream -- *läser data* från annan enhet
ut-strömmar som är objekt av klassen ofstream -- *skriver data* på annan enhet
in-ut-strömmar som är objekt av klassen fstream -- *läser och skriver data* på annan enhet

Klasserna ifstream, ofstream och fstream ärver (något förenklat) enligt :



När man ska använda egna strömmar *inkluderar man alltid fstream.*

Ex : Skriv ett program som läser heltal från en textfil talin.txt som ser ut som :

12 13 23
34 16
3 35 12 15

och skriver ut radsummorna på en annan textfil talut.txt enligt :

48
50
65

```
#include <fstream>
using namespace std;

void main()
{
   int tal, radsum = 0;
   ifstream fin("talin.txt");
   ofstream fout("talut.txt");

   fin >> tal;
   while ( !fin.eof() )
   {
      radsum += tal;
      if ( fin.peek() == '\n' || fin.peek() == EOF )
      {
         fout << radsum << endl;
         radsum = 0;
      }
      fin >> tal;
   }
}
```

eller eftersom fin >> tal returnerar objektet fin kan man istället skriva

```
void main()
{
   int tal, radsum = 0;
   ifstream fin("talin.txt");
   ofstream fout("talut.txt");

   while ( !(fin >> tal).eof() )
   {
      radsum += tal;
      if ( fin.peek() == '\n' || fin.peek() == EOF )
      {
          fout << radsum << endl;
          radsum = 0;
      }
   }
}
```

Operatorerna för in- och utmatning ärvs ner till fstream-objekt. Man kan alltså använda dessa på precis samma sätt som vid in- och utmatning via cin och cout. Definierar man egna överlagrade in- och utmatningsoperatorer för sina objekt kan alltså dessa användas även på fstream-objekt, alltså textfiler.

Ex : Istället för att mata in rationella tal från tangentbordet (cin) kan man läsa dessa från en textfil. Man bör dock ej ha med ledtexten i inmatningsfunktionen. På samma sätt kan man använda utskriftsoperatorn för utskrift på fil istället för skärm. Har man en fil tal.txt enligt :

1/2 1/3 1/4
1/5 1/3 1/2

ger nedanstående program utfilen res.txt enligt :

7/12
11/30

```
#include <iostream>
using namespace std;

class RTAL
{
   private :
      int t;
      int n;
   public :
      friend istream &operator>>(istream &in, RTAL &rt);
      friend ostream &operator<<(ostream &out, RTAL rt);
      .....
};
```

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

istream &operator>>(istream &in, RTAL &rt)
{
   char c;

   in >> rt.t >> c >> rt.n;
   return in;
}


ostream &operator <<(ostream &out, RTAL rt)
{
   out << rt.t << '/' << rt.n << endl;
   return out;
}
```

```
// Huvudprogram -- rtalmain.cpp

#include "rtal.h"
#include <fstream>
using namespace std;

void main()
{
   RTAL a, b, c;
   ifstream fin("tal.txt");
   ofstream fout("res.txt");

   fin >> a >> b >> c;
   while ( !fin.eof() )
   {
      fout << a + b*c;
      fin >> a >> b >> c;
   }
}
```

eller alternativt

```
// Huvudprogram -- rtalmain.cpp

#include "rtal.h"
#include <fstream>
using namespace std;

void main()
{
   RTAL a, b, c;
   ifstream fin("tal.txt");
   ofstream fout("res.txt");

   while ( !(fin >> a >> b >> c).eof() )
   {
      fout << a + b*c;
   }
}
```

OBS! Eftersom fin är ifstream-objekt och ifstream är en subklass till istream kan man anropa operator-funktionen >> med aktuella parametrar i form av ifstream-objekt även om de formella parametrarna är istream-objekt. Detsamma gäller utmatning. Objekt av basklasser kan tilldelas, initieras eller refereras med subklassobjekt men ej tvärtom

Det finns fler parametrar till fstream-konstruktorerna. De formella konstruktorerna med sina parameterlistor ser ut som :

```
ifstream(const char *name,int mode = ios::in,int prot=filebuf::openprot);
ofstream(const char *name,int mode = ios::out,int prot=filebuf::openprot);
fstream(const char *name,int mode,int prot = filebuf::openprot);
```

där prot-parametern anger filskyddet (hidden, system etc.) och mode-parametern anger filtypen som kan vara :

ios::in	läs in från fil till program
ios::out	skriva ut från program till fil
ios::app	skriva till på slutet av utfil
ios::ate	ställa filpekaren på filslut
ios::trunc	ta bort fil om den finns och skapa ny
ios::nocreate	utfilen måste finnas annars felindikering
ios::noreplace	utfilen får ej finnas annars felindikering
ios::binary	binärfil (default är textfil)

Istället för att öppna filen med konstruktorn kan man använda den parameterlösa konstruktorn och öppna explicit med anrop av open-funktionen som har samma parametrar som konstruktorerna ovan.

Ex: Skriv ett program som läser in information från tangentbordet till en binärfil med.dat i form av medlemsposter innehållande medlemsnummer och namn och därefter frågar efter poster som ska uppdateras.

```
// Huvudprogram --filtest3.cpp

#include <fstream>
using namespace std;

void main()
{

   // Öppna fil i binärmode för både in och utmatning

   fstream fil("med.dat", ios::in | ios::out | ios::binary);

   struct medtyp
   {
      int nr;
      char namn[30];
   };

   medtyp med;
   int pnr;
```

```
   // Läsa in poster till fil
   cout << "Ge namn : ";
   cin.getline(med.namn, sizeof(med.namn));
   while ( med.namn[0] != '\0')
   {
      cout << "Ge nr : ";
      cin >> med.nr;
      cin.ignore();
      fil.write((char *)&med, sizeof(medtyp));
      cout << "Ge namn : ";
      cin.getline(med.namn, sizeof(med.namn));
   }

   // Uppdatera post i fil
   cout << endl << "Vilken post ska uppdateras : ";
   cin >> pnr;
   cin.ignore();
   fil.seekg(pnr*sizeof(medtyp));
   fil.read( (char *)&med, sizeof(medtyp));
   cout << "Ge nytt namn : ";
   cin.getline(med.namn, sizeof(med.namn));
   fil.seekp(pnr*sizeof(medtyp));
   fil.write((char *)&med, sizeof(medtyp));

   // skriv ut innehållet i filen
   cout << endl;
   fil.seekg(0);
   while ( !fil.read((char*)&med, sizeof(medtyp)).eof() )
   {
      cout << "Nr   : " << med.nr << endl;
      cout << "Namn : " << med.namn << endl;
   }
}
```

OBS! Binärfiler är byte-orienterade precis som textfiler. Skillnaden är att *i textfiler är informationen ASCII-kodad* och kan skapas med editor och skrivas ut på printer
och skärm.

OBS! Utskrift på fil och läsning från fil av alla bytes i variabeln med anropen:

```
fil.write((char*)&med, sizeof(medtyp));
fil.read((char*)&med, sizeof(medtyp));
```

OBS! Ställa filpekaren på ett visst post-nummer i filen enligt :

```
fil.seekg(pnr*sizeof(medtyp));     // för läsning, get
fil.seekp(pnr*sizeof(medtyp));     // för skrivning, put
```

Ovanstående funktioner räknar byten från filens början. Det finns också överlagrade funktioner som har två parametrar där den första parametern anger byteposition relativt filens början, om andra parametern är ios::beg, aktuell position om ios::cur eller slutet om ios::end.

För att avläsa byteposition finns motsvarande funktioner, fil.tellg och fil.tellp.

Ex : Antag att man vill spara aktuella värden på data för sina ventilobjekt. Man kan skriva binära in- och utmatningsfunktioner i sin ventilklass.

```
// Specifikation av VENTIL-klassen -- ventil1.h

#include <fstream>
using namespace std;

class VENTIL
{
   private :
      char id[10];
      float vp;
      ...
   public :
      friend ofstream &operator<<(ofstream &fout, const VENTIL &v);
      friend ifstream &operator>>(ifstream &fin, VENTIL &v);
      ....
};



// Implementation av VENTIL-klassen -- ventil1.cpp

#include "ventil1.h"

ofstream &operator<<(ofstream &fout, const VENTIL &v)
{
   fout.write((char *)&v, sizeof(VENTIL));
   return fout;
}


ifstream &operator>>(ifstream &fin, VENTIL &v)
{
   fin.read((char *)&v, sizeof(VENTIL));
   return fin;
}



// Huvudprogram -- ventil1m,cpp

#include "ventil1.h"

void main()
{
   VENTIL v1, v2;
   ifstream vfil("ventil1.dat", ios::in | ios::binary);

   vfil >> v1 >> v2;
}
```

OBS! Sizeof(klass) ger storleken i byte på klassens data. Där ingår *ej* this-pekaren.

Har man data i form av pekare måste man vid skrivning till fil och läsning från fil förfara på ett annat sätt när man implementerar de binära ut- och inmatningsoperatorerna.

```
// Specifikation av VENTIL-klassen -- ventil2.h

#include <fstream>
using namespace std;

class VENTIL
{
   private :
      char *id;
      float vp;
      ...
   public :
      friend ofstream &operator<<(ofstream &fout, const VENTIL &v);
      friend ifstream &operator>>(ifstream &fin, VENTIL &v);
      ....
};


// Implementation av VENTIL-klassen -- ventil2.cpp

#include "ventil2.h"
#include <cstring>

ofstream &operator<<(ofstream &fout, const VENTIL &v)
{
   int i = strlen(v.id) + 1;

   fout.write((char *)&i, sizeof(int));
   fout.write(v.id, i);
   fout.write((char *)&v.vp, sizeof(float));
   return fout;
}

ifstream &operator>>(ifstream &fin, VENTIL &v)
{
   int i;

   fin.read((char *)&i, sizeof(int));
   v.id = new char[i];
   fin.read(v.id, i);
   fin.read((char *)&v.vp, sizeof(float));
   return fin;
}


// Huvudprogram -- ventil1m,cpp

#include "ventil2.h"

void main()
{
   VENTIL v1, v2;
   ifstream vfil("ventil2.dat", ios::in | ios::binary);

   vfil >> v1 >> v2;
}
```