Innehållsförteckning

3 Överlagring av operatorer

Operatoröverlagring (polymorfism) är en teknik som gör det smidigare att jobba med klasser.

Antag att man vill addera elementen i två arrayer. Med den teknik vi känner till måste vi skriva:

```
int v1[10], v2[10], v3[10];

// Fyll v1 och v2 med värden
for(int i=0; i<10; i++)
{
  v1[i] = 1 + 2*i;
  v2[i] = 3 + 7*i*i;
}
// Summering av arrayerna
for(int i=0; i<10; i++)
  v3[i] = v1[i] + v2[i];
```

Man måste alltså hantera vektorerna elementvis vilket är ganska omständligt. Med operatoröverlagring kan man ordna så att man istället kan skriva: v3 = v1 + v2;

För att beskriva hur man gör operatoröverlagring ska vi skapa en klass som hanterar tid i form av timmar och minuter. Med hjälp av denna klass ska vi sedan bl.a. a kunna addera och subtrahera tider.

3.1 Klassen Time

Vi bygger på klassen Time från föregående avsnitt med en medlemsfunktion som adderar två tider, sum().

Klassdefinition

```
class Time
  {
    private:
      int hour, min;
    public:
      // Konstruktorer
      Time();
      Time(int pHour, int pMin=0);

      // Medlemsfunktioner
      // Sätt värden på datamedlemmar
      void setHour(int pHour);
      void setMin(int pMin);

      // Läs datamedlemmars värden
      int getHour()const {return hour;}
      int getMin()const {return min;}

      // Addera tid mha två Time-objekt
      Time sum(const Time &time) const;
      void tic();
  };
  // Funktion för utskrift av ett Time-objekt
  void showTime(Time const &t);
```

Kommentarer:

- I sum skickar man in ett Timeobjekt och returnerar Timeobjekt
- I Time sum(const Time &time) const; görs referensanrop av effektivitetsskäl

Det enda som är tillagt, jämfört med förra time.h, är medlemsfunktionen för summering av två Timeobjekt. Här visas implementationen av denna funktion:

```
Time Time::sum(const Time &time) const
{
  Time tmp;
  tmp.min   = min + time.min;
  tmp.hour  = hour + time.hour + tmp.min / 60;
  tmp.min  %= 60;

  return tmp;
}
```

Kommentarer:

- I `tmp.min = min + time.min` avser `min` ”det anropande objektet” och `time.min` avser objektet i parametern. Jämför: `time = time2.sum(time3);` Här är `time2` det anropande objektet.

Ett klientprogram som använder Time – klassen:

```
//---------------------------------------------------------------------------
// oload_010. Version 10
// Addera tider i klassen Time  (time1.h)
// Per Ekeroot 2014-01-13
//---------------------------------------------------------------------------
#include "time1.h"
//---------------------------------------------------------------------------

int main()
{
  // Skapa tre Time-objekt
   Time time;               // Förvald konstruktor
  Time time2(5, 40);        // Initieringskonstruktor
  Time time3(2, 55);        // Initieringskonstruktor

  showTime(time);           // Visa data (=tid) för de två objekten
  cout << endl << endl << "Time 1          = ";
  showTime(time2);
  cout << endl << "Time 2          = ";
  showTime(time3);

  cout << endl <<  "Time 1 + Time 2 = " ;
  time = time2.sum(time3);  // Addera två tider
  showTime(time);           // Visa summatiden

           cout << endl << endl;
  return 0;
}
```

Kommentarer:

- Addera två tider med `time = time2.sum(time3);`

3.2 Överlagra + operatorn

Det föregående sättet att addera tider är inte så elegant. Vi ska nu visa hur man kan använda operatorn + så att den kan användas tillsammans med klassen Time, vi ska med andra ord överlagra operatorn +.

Klassen justeras på följande sätt:
byt `Time sum(const Time &time) const;` mot
`Time operator+(const Time &time) const;`

```
// time2.h Version 10
// Reviderad för operatoröverlagring av operatorn +
// Per Ekeroot    2014-01-13
//-----------------------------------------------------------------------------
#ifndef time2H
#define time2H
  #include <iomanip>    // setw, setfill
  #include <iostream>
  using namespace std;

  class Time
  {
    private:
      int hour, min;

    public:
      Time();
      Time(int pHour, int pMin=0);

      // Medlemsfunktioner
      // Sätt värden på datamedlemmar
      void setHour(int pHour);
      void setMin(int pMin);

      // Läs datamedlemmars värden
      int getHour()const {return hour;}
      int getMin()const {return min;}

      // Addera tid mha +-operatorn
      Time operator+(const Time &time) const;

      void tic();
  };
  // Funktion för utskrift av ett Time-objekt
  void showTime(Time const &t);
#endif
```

Kommentar:

- Om du jämför med Timeklassen i time1.h så är det är bara **sum** som har bytts mot **operator +** !!!

Motsvarande definitionen av medlemsfunktionen operator+ blir (finns i time2.cpp):

```
Time Time::operator+(const Time &time) const
{
  Time sum;
  sum.min  = min + time.min;
  sum.hour = hour + time.hour + sum.min / 60;
  sum.min  %= 60;
  return sum;
}
```

De övriga medlemsfunktionerna i klassen ändras inte.

Klientprogrammet kan nu skrivas:

```
// oload_020 Version 10
// Addera tider (Time-objekt) med + - operatorn (time2.h)
// Per Ekeroot 2014-01-13
//--------------------------------------------------------------------------
#include "time2.h"
int main()
{
  // Skapa tre Time-objekt
  Time time;                  // Förvald konstruktor
  Time time2(5, 40);          // Konstruktor för initiering
  Time time3(2, 56);          // Konstruktor för initiering

  showTime(time);               // Visa data (=tid) för de olika objekten
  cout << endl << endl << "Tid 1         = ";
  showTime(time2);
  cout << endl << "Tid 2         = ";
  showTime(time3);

  cout << endl <<  "Tid 1 + Tid 2 = " ;
  time = time2 + time3;     // Addera två tider
  showTime(time);              // Visa summatiden
  cout << endl << endl;
  return 0;
}
```

Kommentarer:

- Nu skriver man elegant: `time = time2 + time3;`
- Detta är bara ett annat skrivsätt för `time = time2.operator + (time3)`
- Detta att additionsoperatorn (+) har olika betydelser beroende på sammanhanget benämner man polymorfism. I exemplet nedan adderar '+' – operatorn först heltal och sedan Timeobjekt.

```
int a,b,c
Time t1,t2,t3;
c = a+b;
t3 = t1+t2;
```

3.3 **Operatorer som kan överlagras**

Många men inte alla operatorer kan överlagras. Här följer en lista på de operatorer som kan överlagras.

+	-	*	/	%	^	&
|	-=	!	=	<	>	+=
.=	*=	/=	%=	^=	&=	|=
<<	>>	>>=	<<=	==	!=	<=
>=	&&	||	++	--	,	->*
()	[]	new	delete	new[]	delete[]	->

3.4 Restriktioner vid överlagring

Följande regler gäller när man överlagrar operatorer:

1. Den överlagrade operatorn måste ha minst en operand som är en egendefinierad typ
2. Man kan inte använda en överlagrad operator som bryter mot syntax för den ursprungliga operatorn. T ex gäller prioriteringsreglerna som förut.
3. Man kan inte skapa nya operatorer. T ex **
4. De flesta operatorerna kan överlagras antingen i medlemsfunktioner eller i icke-medlemsfunktioner. För följande operatorer måste man använda medlemsfunktioner:
   = () [] ->

3.5 Överlagra -, *, == och < i klassen Time

Vi ska nu utvidga Time-klassen så att även operatorerna -,*, == och < överlagras.

I time3.h ser Time-klassen ut så här:

```
// time3.h Version 10
// Klassen Time reviderad. Nu ingår även operatoröverlagring av
//   operatorerna +, -,*,  ==, !=, <, > )
// Per Ekeroot  2014-01-13
//---------------------------------------------------------------------------
#ifndef time3H
#define time3H
//---------------------------------------------------------------------------
  #include <iomanip>    // setw, setfill
  #include <iostream>
  using namespace std;

  class Time
  {
    private:
      int hour, min;

    public:
      Time();
      Time(int pHour, int pMin=0);

      // Medlemsfunktioner
      // Sätt värden på datamedlemmar
      void setHour(int pHour);
      void setMin(int pMin);

      // Läs datamedlemmars värden
      int getHour()const {return hour;}
      int getMin()const {return min;}

      // Överlagring av operatorer
      Time operator+(const Time &time) const;
      Time operator-(const Time &time) const;
      Time operator*(int faktor) const;
      bool operator==(const Time &time) const;
      bool operator<(const Time &time) const;

      void tic();
  };
  // Funktion för utskrift av ett Time-objekt
  void showTime(Time const &t);
#endif
```

Implementation av medlemsfunktionerna för överlagring

```
//-----------------------------------------------------------------------------
// Subtraktion
//-----------------------------------------------------------------------------
Time Time::operator-(const Time &time) const
{
  int tot1, tot2;
  tot1 = time.min + time.hour*60;
  tot2 = min + hour*60;

  Time tmp;
  tmp.min  = (tot2 - tot1) % 60;
  tmp.hour = (tot2 - tot1) / 60;

  return tmp;
}
```

Kommentarer:

- Det som skiljer överlagringen av '-' från överlagringen av '+' är beräkningen av resultatet

```
//-----------------------------------------------------------------------------
// Multiplikation
//-----------------------------------------------------------------------------
Time Time::operator*(int faktor) const
{
   int totMin = hour * faktor * 60 + min * faktor;

   Time tmp;
   tmp.hour = totMin / 60;
   tmp.min  = totMin % 60;

   return tmp;
}
```

Kommentarer:

- Multiplikation i det här fallet innebär att man multiplicerar tiden (timmar och minuter) med en heltalsfaktor d.v.s. time1 * 17

```
//-----------------------------------------------------------------------------
// Likhet
//-----------------------------------------------------------------------------
bool Time::operator==(const Time &time) const
{
  return hour == time.hour && min == time.min;
}
```

Kommentarer:

- Med likhetsoperatorn kan man kontrollera om två Timeobjekt är lika: if(time1 == time2) osv.
- Om timmarna **och** minuterna i båda objekten är lika returneras **true** annars false

```
//-----------------------------------------------------------------------------
// Mindre än
//-----------------------------------------------------------------------------
bool Time::operator<(const Time &time) const
{
  return (hour*60 + min) < (time.hour*60 + time.min);
}
```

Kommentarer:

- Räkna om till minuter innan tiderna jämförs

3.5.1 Använd de nya operatorerna

Skapa tre Time-objekt

```
  Time time;                // Förvald konstruktor
  Time time2(5, 45);        // Konstruktor för initiering
  Time time3(2, 55);        // Konstruktor för initiering
```

Använd minusoperatorn:

```
  time = time2 - time3;
```

Använd multiplikationsoperatorn:

```
  time =  time2 * 3;
```

Kontrollera om två Timeobjekt ät lika

```
  cout << "time2 and time3 are ";
  if(time2 == time3)
    cout << " equal";
  else
    cout << "not equal";
```

Använd "mindre-än" – operatorn (<):

```
  showTime(time);
  if(time < time2)
    cout << " is less than ";
  else
    cout << " is bigger than ";
  showTime(time2);
```

Till slut ett helt klientprogram som testar Time-klassen i time3.

```
// oload_030 Version 10
// Använd den reviderade klassen Time  (time3.h)
// Per Ekeroot 2014-01-13
//-----------------------------------------------------------------------------
#include "time3.h"
#include <iostream>
using namespace std;

//-----------------------------------------------------------------------------
int main()
{
  // Skapa tre Time-objekt
  Time time;                // Förvald konstruktor
  Time time2(5, 45);        // Konstruktor för initiering
  Time time3(2, 55);        // Konstruktor för initiering

  // Visa data (=tid) för de olika objekten
  cout << "time  = ";
  showTime(time);
  cout << endl << "time2 = ";
  showTime(time2);
  cout << endl << "time3 = ";
  showTime(time3);

  // Subtrahera två tider
  cout << endl <<"time2 - time3 = ";
  time = time2 - time3;
  showTime(time);

  // Multiplicera en tid med en faktor
  cout << endl << "time2 * 3     = ";
```

```
  time = time2 * 3;
  showTime(time);

  // Testa likhetsoperatorn
  cout << endl;
  cout << "time2 och time3 are ";
  if(time2 == time3)
    cout << " equal";
  else
    cout << "not equal";

  // Testa mindre än-operatorn
  cout << endl << endl;
  showTime(time);
  if(time < time2)
    cout << " is less than ";
  else
    cout << " is bigger than ";
  showTime(time2);

  cout << endl << endl;

  return 0;
}
```

Kommentarer:

- Testa vad som händer om man skriver `time = 2.75 * time2`;
- Det går inte så bra. Men det går att lösa vilket visas längre fram.

3.6 Tider i en array

Vi ska nu använda Time klassen för att skapa en vector med tider (Time-objekt) och sortera tiderna (objekten) i tidsordning.

Skapa en vector med Time-objekt:

```
  vector<Time> times;
```

Kommentarer:

- Förvald konstruktor körs för varje element i arrayen, d.v.s. timme och minut sätts = -1.

Slumpa tider till arrayen i en funktion:

```
  default_random_engine generator(static_cast<unsigned>(time(0))); //Init slumpn.
  uniform_int_distribution<int> randomHour(0,23);  // Slumpvärden för timmar
  uniform_int_distribution<int> randomMin(0,59);   // Slumpvärden för minuter

  for(int i=0; i<numOfTimes; i++)
    times.push_back(Time(randomHour(generator),randomMin(generator)));
```

Kommentarer:

- I varje element i vectorn:
  - får datamedlemmen 'hour' ett värde i intervallet 0 – 23
  - får datamedlemmen 'min' ett värde i intervallet 0 – 59

Sortera vectorn med algoritmen sort():

```
sort(times.begin(),times.end());
```

Kommentarer:

- Detta förutsätter att operatorn < (mindre än) är överlagrad i klassen Time, vilket den ju är!

Skriv ut vectorn med funktionen:

```
void showTimes(const vector<Time> times)
{
  for(auto idx: times)
  {
    showTime(idx);
    cout << endl;
  }
}
```

Kommentarer:

- Den förenklade for-loopen används tillsammans med funktionen `showTime(Time const &t)`

Och här kommer hela programmet:

```
// oload_040 Version 10
// Använd klassen Time  (time3.h)
// Hantera en vector med tider
// Per Ekeroot 2014-01-13
//--------------------------------------------------------------------------
#include "time3.h"
#include <iostream>
#include <vector>
#include <random>
#include <ctime>
#include <algorithm>
using namespace std;

//--------------------------------------------------------------------------
```

```
// Funktionsprototyper
//--------------------------------------------------------------------------
void showTimes(const vector<Time> times);

//--------------------------------------------------------------------------
// Huvudprogram
//--------------------------------------------------------------------------
int main()
{
  // Skapa en konstant som anger antal tider som ska slumpas
  const int numOfTimes = 20;

  // Skapa en vector som innehåller tider, datatyp Time
  vector<Time> times;

  // Fyll vector med slumpade tider
  default_random_engine generator(static_cast<unsigned>(time(0))); // Init slumpn.
  uniform_int_distribution<int> randomHour(0,23);  // Slumpvärden för timmar
  uniform_int_distribution<int> randomMin(0,59);   // Slumpvärden för minuter

  for(int i=0; i<numOfTimes; i++)
    times.push_back(Time(randomHour(generator),randomMin(generator)));

// Skriv ut den osorterade vectorn med tider
  cout << endl << "NOT SORTED TIMES " << endl;
  showTimes(times);
  cout << "Press ENTER to continue!";
  cin.get();

  // Sortera vectorn
  sort(times.begin(),times.end());

  // Skriv ut den sorterade vectorn med tider
  cout << endl << "SORTED TIMES " << endl;
  showTimes(times);

  cout << endl;
  cout << "Press ENTER to continue!";
  cin.get();

  return 0;
}

//--------------------------------------------------------------------------
// Funktionsdefinitioner
//--------------------------------------------------------------------------
// Skriv ut vectorn med tider
//--------------------------------------------------------------------------
void showTimes(const vector<Time> times)
{
  for(auto idx: times)
  {
    showTime(idx);
    cout << endl;
  }
}
```

3.7 Överlagra multiplikation med friendfunktion

I föregående exempel har vi gjort överlagring av operatorer med funktioner som tillhört en klass, dessa funktioner är alltså medlemsfunktioner i klassen. Det går även att göra överlagringar med funktioner som inte tillhör en klass. Vi ska först se exempel på hur man kan använda s.k. friendfunktioner för att komma åt datamedlemmarna i en klass när vi överlagrar en operator med en funktion som inte är en medlemsfunktion.

Exempel när man måste använda icke medlemsfunktioner för överlagring av en operator:

- Det ska vara möjligt att byta plats på operanderna hos binära operatorer som använder olika datatyper . T.ex. multiplikation med en int, Time • int (Jag använder här tecknet • för multiplikation.)
- Överlagring av << och >>

I det första exemplet ska vi titta på hur man kan göra överlagringen av multiplikation så att den blir kommutativ. Kommutativ betyder att 2 • 3 är samma sak som 3 • 2!

Tidigare skrev vi `Time t = t1 • 3`
Men försökte vi skriva `Time t = 3 • t1` sa kompilatorn ifrån eftersom överlagringen av • kräver att det ska vara ett Time-objekt före och en double efter operatorn •.

Jämför följande uttryck:

`t1 • 3` betyder ett anrop av `t1.operator •(3)`
`3 • t1` betyder ett anrop av vadå?? Detta fungerar inte eftersom `3.operator*(t1)` är inte definierad

Vi råder bot på detta genom att utvidga klassen Time igen (time4). Vi skriver ytterligare en överlagring av multiplikationsoperatorn. Funktionsprototypen ser ut så här:

```
Time operator*(int faktor, const Time &time);
```

Här finns två parametrar, en som tar ett heltal och en som tar ett Timeobjekt. Heltalsparametern står först och Timeparametern sedan. Detta innebär att för att använda denna överlagring av '*' skriver man: 23 * time1;

Hela headerfilen (time4.h)ser ut så här (observera att den innehåller även överlagringar av >> och << vilka redovisas senare):

```
class Time
  {
    private:
      int hour, min;

    public:
      Time();
      Time(int pHour, int pMin=0);

      // Medlemsfunktioner
      // Sätt värden på datamedlemmar
      void setHour(int pHour);
      void setMin(int pMin);
```

```
      // Läs datamedlemmars värden
      int getHour()const {return hour;}
      int getMin()const {return min;}


      // Överlagring av operatorer
      Time operator+(const Time &time) const;
      Time operator-(const Time &time) const;
      Time operator*(int faktor) const;
      bool operator==(const Time &time) const;
      bool operator<(const Time &time) const;

      // Nya överlagringar i  time4
      friend Time operator*(int faktor, const Time &time);
      friend ostream &operator<<(ostream &os, const Time &time);
      friend istream &operator>>(istream &is, Time &time);

      void tic();
  };
// Funktion för utskrift av ett Time-objekt
  void showTime(Time const &t);
```

Kommentarer:

- De nya överlagringarna är friend-deklarerade. Vad detta innebär förklaras nedan.

Implementationerna av medlemsfunktionerna för överlagring av multiplikation:
Först den gamla:

```
// Multiplikation  Time * faktor
Time Time::operator*(int faktor) const
{
   int totMin = hour * faktor * 60 + min * faktor;

   Time tmp;
   tmp.hour = totMin / 60;
   tmp.min  = totMin % 60;

   return tmp;
}
```

Sedan den nya som behövs för att vi ska kunna skriva `Time t = 2.75*t1;`
Vi har här använt en **friend**-funktion (se klassdefinitionen) för att komma åt datamedlemmarna på ett **lättvindigt** sätt.

```
// Multiplikation    (int * Time)
Time operator*(int faktor, const Time &time)
{
  int minutes = time.hour * faktor * 60 + time.min * faktor;
  Time tmp;
  tmp.hour = minutes /60;
  tmp.min  = minutes % 60;

  return tmp;
}
```

Kommentarer:

- Man måste använda en fristående funktion eftersom en operator överlagrad med en medlemsfunktion förutsätter att den vänstra operanden har den **egna klassen** som datatyp.
  Ex Time t1, t2(10,14); t1 = t2•17; Eftersom t2•17 betyder t2.operator• (17): Det funkar inte med 17•t2 eftersom 17.operator(t2) inte är definierad.
- Det reserverade ordet **friend** skrivs endast i klassdefinitionen.

- En **friend**-funktion är ”**vän**” med klassen och får fri tillgång till medlemmarna i klassen (även private-medlemarna).
- En **friend**-funktion är **INTE** en medlemsfunktion. Därför skrivs inte `Time::` i definitionsfilen.
- Att använda en friend-funktion stöter sig med OOP-principen att man ska gömma datamedlemmarna.
- Vissa läroböcker tycker dock att det i vissa fall ändå är försvarbart att använda friendfunktioner.
- Man klarar sig utan friendfunktioner, men koden blir lite omständligare.
- Vi ska senare visa hur man kan klara sig utan friendfunktioner.
- I denna kurs kommer jag att undvika friendfunktioner om det går.

Provkör oload_050 första delen.

```
// oload_050 Version 10
// Använder klassen Time  (time4)
// Visa att operatorn * (multiplikation) är kommutativ
// Visa utskrift av Time-objekt med <<
// Visa inmatning av Time-objekt med >>
// Överlagra operatorerna med friendfunktioner
// Per Ekeroot 2014-01-13
//-------------------------------------------------------------------------------
#include "time4.h"
#include <iostream>
using namespace std;
//-------------------------------------------------------------------------
int main()
{
//-------------------------------------------------------------------------
// Testa att överlagringen av * (multiplikation) är kommutativ
//-------------------------------------------------------------------------
  int a = 3;
  Time t(1,45);
  Time t2;

  t2 = a*t ;
  showTime(t2);
  cout << endl;

  t2 = t*a ;
  showTime(t2);

//-------------------------------------------------------------------------
// Testa överlagring av in- och utmatningsoperatorerna
//-------------------------------------------------------------------------
  // Inmatningsoperatorn överlagrad i Time
           cout << endl;
  cin >> t2 ;

  // Utmatningsoperatorn överlagrad i Time
  cout << t2 << endl << endl;

  // In- och utmatning av flera objekt
  Time t3;
  cin >> t2 >> t3;
  cout << t2 << endl << t3 << endl << endl;
  return 0;
}
```

3.8 Överlagring av utskriftsoperatorn

Om vi har ett Timeobjekt deklarerat med satsen `Time t(11,35);` måste vi skriva `showTime(t)` för att skriva ut datamedlemmarna på skärmen. Det skulle vara elegantare om man kunde skriva: `cout<<t;` för att få samma utskrift. Detta kan vi åstadkomma om vi överlagrar vi utskriftsoperatorn (<<). Gör följande funktionsprototyp i klassdefinitionen:

```
friend ostream &operator<<(ostream &os, const Time &time);
```

Och funktionsdefinition:

```
ostream &operator<<(ostream &os, const Time &time)
{
  os << setw(2)<< time.hour << ":" <<setw(2) << time.min ;
  return os;
}
```

Kommentarer:

- cout är ett **ostream**-objekt
- << är överlagrad i klassen **ostream** för alla grundläggande datatyper i C++
- Man måste använda två argument, eftersom om man bara använde ett argument
  `operator<<(ostream & os)`
  skulle man vara tvungen att anropa funktionen med:
  `t << cout` jämför : `t.operator<<(cout)`
  Inte förenligt med det vanliga skrivsättet!!
- Med två argument i ordningen ostream – Time, så kommer vi inte åt datamedlemmarna i Time-parametern. Därför görs operator<< -funktionen som friend-funktion till Timeklassen
- Funktionen behöver **inte** vara friend till ostream eftersom endast hela objekt (os) från ostream används.
- os är referensdeklarerad bl.a. för att vi vill använda objektet **cout** själv och inte en kopia.
- Funktionen har returdatatypen ostream för att man ska kunna koppla ihop flera uttryck med <<-operatorn. Tex `cout << "Tid= " << t;`
  Jämför:
  ```
  int x=5, y=3;
  cout << x << y;
  ```
  betyder
  `(cout<<x)<< y.` När `cout << x` är utförd returneras ett cout-objekt och kvar har vi `cout << y;`

3.9 Överlagra inmatningsoperatorn

Vi vill även kunna skriva `cin >> t;` Därför överlagrar vi inmatningsoperatorn. Funktionsprototyp i Time.h:

```
friend istream &operator>>(istream & is, Time & time);
```

Funktionsdefinition:

```
istream &operator>>(istream & is, Time & time)
{
  cout <<"Hour  : ";
  is >> time.hour;
  cout <<"Minute: ";
  is >> time.min;

  return is;
}
```

Kommentarer:

- Inmatning sker med hjälp av klassen istream
- I övrigt samma kommentarer som för överlagring av <<
- Lägg dock märke till att vi här låser oss vid en layout. Denna överlagring är inte så generell som man skulle vilja ha den. Jag återkommer till en mer generell överlagring av både << och >>.

Provkör oload_050 andra delen, koden finns ovan.

3.10 Överlagring av multiplikation utan friend-funktion

Här kommer en variant på överlagring av multiplikation (double • Time) som klarar sig utan friend-funktion. Exempel oload_060 och time5.h och time5.cpp.

Följande funktion hör **inte** till Time-klassen.
Placera funktionsprototypen i time5.h-filen men utanför klassdefinitionen.

```
Time operator *(int faktor, const Time &time);
```

Funktionsdefinitionen som ligger i time5.cpp ser ut så här:

```
//---------------------------------------------------------------------------
// Multiplikation    (int * Time)  Alternativ utformning av funktionen
//---------------------------------------------------------------------------
Time operator *(int faktor, const Time & time)
{
  return time * faktor;
}
```

Kommentarer:

- Här utnyttjar man den ”första” överlagringen av -operatorn i satsen `return time•mult`

3.11 Överlagring av utskriftsoperatorn utan friendfunktion

Placera funktionsprototypen utanför klassdefinitionen i time5.h

```
  ostream &operator<<(ostream &os, const Time &time);
```

och funktionsdefinitionen:

```
ostream &operator<<(ostream &os, const Time &time)
{
  os << setw(2)<< time.getHour() << ":" <<setw(2) << time.getMin();
  return os;
}
```

Kommentarer:

- Det som skiljer mot tidigare är att man i funktionsdefinitionen måste använda funktionsmedlemmar (getHour() och getMin()) för att komma åt datamedlemmarna hour och min.
- I övrigt gäller samma kommentarer som för friend-alternativet.

3.12 Överlagring av inmatningsoperatorn utan friendfunktion

Placera funktionsprototypen utanför klassdefinitionen i time5.h

```
  istream &operator>>(istream &is, Time &time);
```

och funktionsdefinitionen:

```
istream &operator>>(istream &is, Time &time)
{
  int tempHour, tempMin;
  cout <<"Hour  : ";
  is >> tempHour;

  cout <<"Minute: ";
  is >> tempMin;

  time.setTime(tempHour,tempMin);

  return is;
}
```

Kommentarer:

- Använd de temporära variablerna *tempHour* och *tempMin* eftersom man inte kommer åt datamedlemmarna *hour* och *min* direkt.
- Datamedlemmarna ges värden med medlemsfunktionen *setTime*(,)
- I övrigt samma kommentarer som för friend-alternativet

Här redovisas hela time5.h:

```
// time5.h Version 10
// Headerfil till klassen Time
// Reviderad så att operatorn * (multiplikation) är kommutativ
// Per Ekeroot  2014-01-13
//-----------------------------------------------------------------------
#ifndef time5H
#define time5H
//-----------------------------------------------------------------------
  #include <iostream>
  using namespace std;

class Time
{
  private:
    int hour, min;

  public:
    Time();
    Time(int pHour, int pMin=0);

   // Set och get-funktioner
    void setHour(int pHour);
    void setMin(int pMin);
    int getHour()const {return hour;}
    int getMin()const {return min;}

    // Överlagring av operatorer
    Time operator+(const Time &time) const;
    Time operator-(const Time &time) const;
    Time operator*(int faktor) const;
    bool operator==(const Time &time) const;
```

```
    bool operator<(const Time &time) const;

    void tic();
};
// Funktion för utskrift av ett Time-objekt
void showTime(Time const &t);

// Nya överlagringar i  time5
// Funktioner som inte är medlemmar i klassen
Time operator*(int faktor, const Time &time);
ostream &operator<<(ostream &os, const Time &time);
istream &operator>>(istream &is, Time &time);
#endif
```

och de nya delarna av time5.cpp :

```
// Multiplikation    (faktor * Time)
//-----------------------------------------------------------------------------
Time operator*(int faktor, const Time &time)
{
           return time * faktor;
}
//-----------------------------------------------------------------------------
// Utskrift
//-----------------------------------------------------------------------------
ostream &operator<<(ostream &os, const Time &time)
{
  os << setw(2)<< time.getHour() << ":" <<setw(2) << time.getMin();

  return os;
}
//-----------------------------------------------------------------------------
// Inmatning
//-----------------------------------------------------------------------------
istream &operator>>(istream &is, Time &time)
{
  int tempHour, tempMin;
  cout <<"Hour : ";
  is >> tempHour;

  cout <<"Minute: ";
  is >> tempMin;

  time.setHour(tempHour);
  time.setMin(tempMin);

  return is;
}
```

I klientprogrammet märker man inte om operatorerna är överlagrade med eller utan friendfunktioner så oload_060 ser likadant ut som oload_050.

3.13 Överlagring av tillväxtoperatorn ++

I vår Time-klass skulle det vara elegant att kunna skriva t++ om man vill låta datamedlemmen min öka ett steg och hour ett steg då min har ökat 60 steg. Vi ska i detta exempel titta på hur man kan göra.

Vi skapar ytterligare en ny version av Time-klassen (time6.h, time6.cpp) och använder den i klientprogrammet oload_070. Det nya i headerfilen blir:

```
class Time
{
  . . .
  public:
    // Nya medlemsfunktioner i time6.h
    Time &operator++();    // prefix  ++t
    Time operator++(int);  // postfix  t++
   . . .
};
```

Kommentarer:

- All kod från time5.h ska vara med. Visas inte här av utrymmesskäl!
- Tillväxtoperatorn finns i en prefix- och i en postfixversion
- Utformningen av överlagringsfunktionerna utan och med parameter (`operator++()` och `operator++(int)`) är gjorda på detta sätt bara för att man ska kunna skilja pre- och postfixversionerna åt!!

Det nya i definitionsfilen blir:

```
// ++  prefix ++t
Time &Time::operator++()     // prefix  ++t
{
   tic();
   return *this;             // Returnera en referens till det aktuella objektet
}
```

Kommentarer:

- Returdatatypen är referens till ett Time-objekt
- Datamedlemmarna min och hour räknas upp för det aktuella (det anropande) objektet
- `return *this` innebär att det aktuella objektets värde (innehåll) returneras. Mer om this-pekaren längre fram i kursen.

```
// ++  postfix t++
Time Time::operator++(int)  // postfix  t++
{
  Time tmp(*this);          // Skapa en kopia av det aktuella Time-objektet
  ++(*this);                // Stega upp ett steg
  return tmp;               // Returnera det gamla värdet (OBS! Ingen pekare!!)
}
```

Kommentarer:

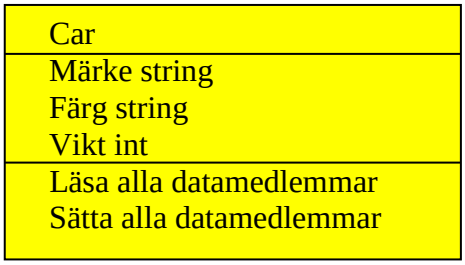
- Eftersom postfixversionen ska returnera det värdet Time-objektet har **innan** ++ operatorn anropas, sparas det aktuella objektets värde i ett temporärt objekt, `Time tmp(*this)`. Konstruktorn körs med det aktuella objektet (*this) som argument .
- Vi utnyttjar att prefixversionen av ++ redan finns, och stegar upp objektet en minut med `++(*this);`
- Det gamla värdet på objektet som lagrats i tmp returneras.

I klientprogrammet load_070 använder vi ++ operatorn i Time-klassen:

```
// oload_070 Version 10
// Använd klassen Time (time6)
// Använd operatorerna t++ och ++t
// Per Ekeroot 2014-01-13
//-----------------------------------------------------------------------------
#include "time6.h"

//-----------------------------------------------------------------------------
// Huvudprogram
//-----------------------------------------------------------------------------
int main()
{
  Time t(10,25);
  t.tic();
  cout << "          cout <<   t => " << t << endl;
  cout << "Prefix:   cout << ++t => " << ++t << endl;
  cout << "          cout <<   t => " << t << endl;
  cout << "Postfix: cout << t++ => " << t++ << endl;
  cout << "          cout <<   t => " << t << endl << endl;

  return 0;
}
```

3.14 Överlaga << och >> för att spara på fil

I detta exempel oload_080 visas hur man kan överlagra strömoperatorerna, << och >>, så att de enkelt kan användas för att spara/läsa objekt på fil. Jag skapar en ny klass Car:



Headerfilen, car.h, ser ut så här:

```
// car.h Version 10
// Headerfil till klassen Car
// Per Ekeroot  2014-01-13
//---------------------------------------------------------------------------
#ifndef carH
#define carH
  #include <iostream>
  #include <string>
  using namespace std;
//---------------------------------------------------------------------------
  class Car
  {
    private:
      // Datamedlemmar
      string mark;
      string color;
      int    weight;

    public:
      // Konstruktorer
      Car();
      Car(string pMark, string pColor, int pWeight);

      // Set-funktioner
      void setMark(string pMark);
      void setColor(string pColor);
      void setWeight(int pWeight);

      // Get-funktioner
      string getMark() const { return mark;}
      string getColor() const { return color;}
      int getWeight() const { return weight;}
  };
  // Överlagring av << och >> för filhantering
  ostream &operator<<(ostream &os, const Car &car);
  istream &operator>>(istream &is, Car &car);

  // Funktioner för inmatning från tangentbord och utskrift till skärm
  void inputCar(Car &car);
  void showCar(Car const &car);
#endif
```

Kommentarer:

- Datamedlemmarna har datatyperna *string* och *int*
- Överlagringarna av << och >> görs i fristående funktioner och utformas för filhantering, dvs för att de ska användas till att spara Car-objekt på och läsa från textfil.

Implementation av medlemsfunktionerna i Car

```
// car.cpp Version 10
// Definitionsfil till klassen Car
// Per Ekeroot 2014-01-13
//--------------------------------------------------------------------------
#include "car.h"
//--------------------------------------------------------------------------
// Förvald konstruktor
//--------------------------------------------------------------------------
Car::Car()
{
  weight=0;
}
//--------------------------------------------------------------------------
// Konstruktor för initiering av datamedlemmarna
//--------------------------------------------------------------------------
Car::Car(string pMark, string pColor, int pWeight)
{
  mark   = pMark;
  color  = pColor;
  weight = pWeight;
}
//--------------------------------------------------------------------------
// setMark
//--------------------------------------------------------------------------
void Car::setMark(string pMark)
{
  mark = pMark;
}
//--------------------------------------------------------------------------
// setColor
//--------------------------------------------------------------------------
void Car::setColor(string pColor)
{
  color = pColor;
}
//--------------------------------------------------------------------------
// setWeight
//--------------------------------------------------------------------------
void Car::setWeight(int pWeight)
{
  weight = pWeight;
}
//--------------------------------------------------------------------------
// Överlagring av << och >>
// << och >> anpassas så att de kan användas för att spara Car-objekt på textfil
// Funktionerna är INTE medlemmar i klassen Car
//--------------------------------------------------------------------------
// Avgränsare mellan datmedlemmar i textfilen
//--------------------------------------------------------------------------
const char DELIM = '|';
//--------------------------------------------------------------------------
// Överlagring av utskriftsoperatorn
//--------------------------------------------------------------------------
ostream &operator<<(ostream &os, const Car &car)
{
  os << car.getMark() << DELIM;
  os << car.getColor()<< DELIM;
  os << car.getWeight();
  return os;
}
//--------------------------------------------------------------------------
```

```
// Överlagring av inmatningsoperatorn
//-----------------------------------------------------------------------
istream &operator>>(istream &is, Car &car)
{
  string tmpString;
  getline(is,tmpString,DELIM);   // Läs märke med getline() fram till DELIM
  car.setMark(tmpString);

  getline(is,tmpString,DELIM);   // Läs color med getline() fram till DELIM
  car.setColor(tmpString);

  int tmpInt;
  is >> tmpInt;
  is.get();               // Läs bort kvarlämnat ENTER från inmatningsströmmen
  car.setWeight(tmpInt);

  return is;
}
//-----------------------------------------------------------------------
// Mata in ett Car-objekt från tangentbordet
//-----------------------------------------------------------------------
void inputCar(Car &car)
{
  string tmpString;
  int tmpInt;

  cout << ** Input a Car ***" << endl;
  cout << "Mark   : ";
  getline(cin,tmpString);
  car.setMark(tmpString);

  cout << "Colour : ";
  getline(cin,tmpString);
  car.setColor(tmpString);

  cout << " Weight: ";
  cin >> tmpInt;
  car.setWeight(tmpInt);
  cout << endl;
}
//-----------------------------------------------------------------------
// Skriv ut ett Car-objekt på skärmen
//-----------------------------------------------------------------------
void showCar(Car const &car)
{
  cout << "Marke  : " <<car.getMark() << endl;
  cout << "Colour : " <<car.getColor() << endl;
  cout << "Weight : " <<car.getWeight() << endl << endl;
}
```

Kommentarer:

- Överlagring av utskriftsoperatorn, strömoperatorn <<.
  - ett objekt per rad på detta sätt: mark DELIM color DELIM weight
  - ingen radframmatning
- Överlagring av inmatningsoperatorn, strömoperatorn >>
  - läs strängar med getline()
  - `getline(is,tmpString,DELIM);` läser fram till DELIM
  - ger datamedlemmarna värden med set-funktioner
  - `is >> tmpInt;` lämnar kvar ett ENTER i inmatningsströmmen. Detta läses bort med `cin.get();`

- Skriv separata funktioner för inmatning från tangentbordet och utskrift på skärmen. Detta är en mer generell lösning jämfört med att överlagra >> och << för inmatning från tangentbord och utskrift till skärm.
- Om man vill använda olika utskriftslayout på olika ställen i programmet är det enkelt att skriva fler utmatningsfunktioner.

Denna klass gör att klientprogrammet blir elegant (oload_80):

```
// oload_080 Version 10
// Använd klassen Car (car.h)
// Överlagra operatorerna << och  >> så att de är anpassade för att spara
// objektet på textfil.
// Per Ekeroot 2014-01-13
//-----------------------------------------------------------------------------
#include "car.h"
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
using namespace std;
//-----------------------------------------------------------------------------
int main()
{
  // Skapa en array och initiera den med tre bilobjekt
  vector<Car> car;
  car.push_back(Car("VW Passat","Gul", 1560));
  car.push_back(Car("Volvo S80","Svart", 1987));
  car.push_back(Car("Toyota Previa", "Ljusbrun",2107));

  // Mata in data för ytterligare ett Car-objekt med hjälp av inputCar()
  car.push_back(inputCar());

  // Spara data för bilobjekten på filen car.txt
  // genom att använda överlagringen av << för att skriva på filen
  fstream outFile("car.txt",ios::out);
  for(auto idx: car)
    outFile << idx << endl;
  outFile.close();

  // Läs in Car-objekt från filen car.txt till vectorn car2
  // genom att använda överlagringen av >> för att läsa från filen
  vector<Car> car2;
  fstream inFile("car.txt",ios::in);
  Car tmpCar;
  while(inFile >> tmpCar )
    car2.push_back(tmpCar);
  inFile.close();

  // Skriv ut vectorn med Car-objekt på skärmen med hjälp av showCar()
  cout << endl << "Car read from the file car.txt" << endl << endl;
  for(auto idx: car)
    showCar(idx);

  return 0;
}
```

Kommentarer:

- Inläsning från fil görs med
  ```
  Car tmpCar;
  while(inFile >> tmpCar )
    car2.push_back(tmpCar);
  ```
- `inFile >> tmpCar` returnerar **true** när inläsningen går bra, annars returneras **false**

3.15 Typomvandling

Först görs en kort repetition av typomvandling mellan float och int sedan visas typomvandling mellan egendefinierade datatyper (klasser).

3.15.1 Automatisk typomvandling

```
int k ;
float x = 3.25;
k = x;
cout << k;  // Skriver ut 3 på skärmen
```

Här sker en automatisk typomvandling.

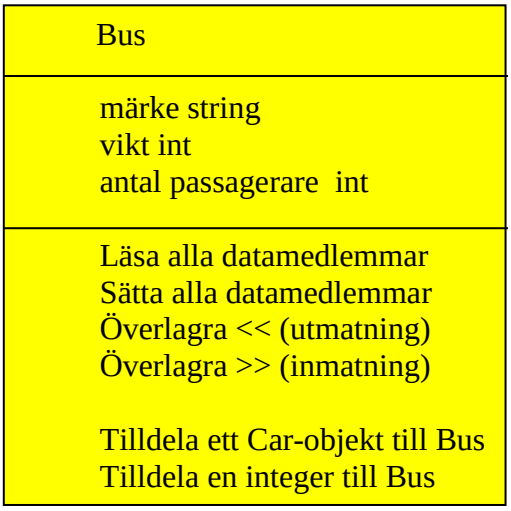
3.15.2 Uttrycklig typomvandling

```
int k=5, p=8;
float x = k / float(p);
cout << x    // Skriver ut  0.625 på skärmen
```

Olika skrivsätt: static_cast<float>(p), float(p) eller (float) p

3.15.3 Typomvandling mellan egendefinierade datatyper.

För att demonstrera typomvandling mellan egendefinierade datatyper (klasser) har jag gjort en klass Bus för at sedan typomvandla ett Carobjekt till ett Busobjekt! Klassen Bus får följande innehåll:



Klassdefinition för Bus:

```
// bus.h Version 10
// Headerfil till klassen Bus
// Visar användning av typomvandlingskonstruktorer
// Per Ekeroot  2014-01-13
//---------------------------------------------------------------------------
#ifndef busH
#define busH
//---------------------------------------------------------------------------
  #include "car.h"
  #include <string>
  using namespace std;

class Bus
{
  private:
    string mark;
    int    weight;
    int    passNum;

  public:
    // Konstruktorer
    Bus();
    Bus(string pModel, int pWeight, int pPassNum);

    // Typomvandlingskonstruktorer
    Bus(Car car);
    explicit Bus(int num);

    // Get-funktioner
    string getMark() const {return mark;}
    int getWeight()  const {return weight;}
    int getPassNum() const {return passNum;}

    // Set-funktioner
    void setMark(string pModel);
    void setWeight(int pWeight);
    void setPassNum(int pPassNum);
};
//---------------------------------------------------------------------------
// Funktioner för utskrift och inmatning av Bus-objekt
//---------------------------------------------------------------------------
void showBus(const Bus &bus);
void inputBus(Bus &bus);
#endif
```

Default-konstruktorn och konstruktorn för initiering av datamedlemmarna är implementerade på samma sätt som motsvarande konstruktorer i Car. Det speciella i klassen Bus är konstruktorerna Bus (Car car) och Bus(int num).

3.15.4 Typomvandlingskonstruktor

Implementation för Bus(Car car) kommer här:

```
// Typomvandlingskonstruktor. Från Car till Bus.
//---------------------------------------------------------------------------
Bus::Bus(Car car)
{
  mark    = car.getMark();
  weight  = car.getWeight();
  passNum = 0;
}
```

Kommentarer:

- Om en konstruktor har **ett** argument av annan datatyp (än klassen) säger man att konstruktorn är en *typomvandlingskonstruktor*.
- I det här fallet sker en typomvandling från Car till Bus
- mark = car.getMark(); man måste använda en ”get” –funktion från Car-klassen för att komma åt medlemsvariablerna
- passNum = 0 eftersom passNum inte finns som datamedlem hos Car.
- Datamedlemmen color hos Car lämnas utan åtgärd eftersom Bus saknar color

I programmet skriver man:

```
  Bus bus;                // Förvald konstruktor körs
  inputBus(bus);
  cout << endl << " Print values: " << endl;
  showBus(bus);

  Car car("Volvo Amazon", "Vit", 1390);
  bus = car;    // Automatisk typomvandling från Car till Bus
                // Datamedlemmarna i bus skrivs över

  cout << " Cast a Car to a Bus" << endl;
  showBus(bus);
```

Provkör i programmet (oload_090).

Den andra typomvandlingskonstruktorn implementeras så här:

```
// Typomvandlingskonstruktor. Från int till Bus.
//-------------------------------------------------------------------------
Bus::Bus(int num)
{
  mark    = "saknas";
  weight  = 0;
  passNum = num;
}
```

Kommentarer:

- Observera funktionsprototypen i klassdefinitionen: `explicit Bus(int num);`
- *explicit* innebär att man måste göra en uttrycklig typomvandling om man vill omvandla från en int till en Bus
- Den enda ”vettiga” datamedlemmen att ge ett heltalsvärde är passNum. De andra ”nollas”!

Programexempel för denna uttryckliga typomvandling:

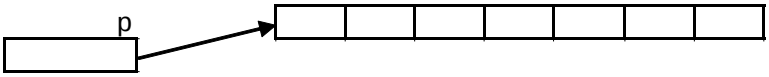
```
// Testa uttrycklig (explicit) typomvandlingskonstruktor
//-------------------------------------------------------------------------
  Bus bus2;

  // bus2 = 34     // Automatisk omvandling fungerar inte
  bus2 = static_cast<Bus>(34); // Uttrycklig typomvandling
  cout << " Cast an integer to a Bus: " << endl;
  cout << bus2 << endl << endl;
```

Kör programmet oload_090.

3.16 Klasser och dynamisk minnesallokering

I de klasser som vi har gjort hittills har datamedlemmarna varit enkla datatyper (int, float), egna klasser eller standardklasser. Nu ska vi göra en klass där en datamedlem är en pekare. Pekaren pekar på data vars plats allokeras när ett objekt, där pekaren ingår som datamedlem, skapas (instansieras).

Mer konkret: vi ska göra en klass som hanterar en array (c-array). Pekaren pekar på arrayens första element. Bland annat ska arayens storlek bestämmas dynamiskt då ett arrayobjekt skapas.



3.17 En arrayklass

Vi ska skapa en arrayklass för heltal (int) med följande innehåll:

IntArray	
pekare till en array med heltal	*int
arrayens maximala storlek	int
aktuellt antal element i arrayen	int
Förvald konstruktor Konstruktor för initiering av datamedlemmar Kopieringskonstruktor Destruktor Lägg till ett värde i arrayen Läs ett värde med hjälp av index Läs arrayens storlek Överlagra tilldelningsoperatorn	

Klassdefinition:

```
class IntArray
{
  private:
    int *arr;             // Pekare till arrayen
     size_t maxSize;     // Arrayens maximala storlek
     size_t size;        // IntArrayens aktuella storlek
  public:
    // ---------------- Konstruktorer och destruktor
    IntArray();
    IntArray (int pNum);
    IntArray (const IntArray &a);
    ~IntArray();

    //------------------ Get- och set-funktioner
    size_t getMaxSize() const {return maxSize;}
    size_t getSize() const {return size;}
    int getValue( int idx) const;
    bool addValue(int value);
    // ----------------- Överlagra tilldelningsoperatorn
    const IntArray &operator=(const IntArray &a);
};
```

3.18 Implementation av medlemsfunktionerna

3.18.1 Förvald konstruktor

```
IntArray::IntArray(): maxSize(0), size(0)
{
  arr = nullptr;
}
```

Kommentarer:

- Initieringslista används för datamedlemmarna `maxSize` och `size` vilka nollställs;
- Pekaren `arr` sätts att peka på `nullptr`.

3.18.2 Initieringskonstruktor

```
IntArray::IntArray (int pMaxSize): maxSize(pMaxSize), size(0)
{
  arr = new int[maxSize];
}
```

Kommentarer:

- Initieringslista används för datamedlemmarna `maxSize` och `size`. `maxSize` initieras medan `size` sätts = 0.
- Det allokeras plats för `maxSize` stycken heltal på heapen. **arr** pekar på detta minnesutrymme

3.18.3 Destruktor

Eftersom vi allokerar plats på heapen, måste vi se till att avallokera platsen när objektet tas bort. Detta görs i destruktorn. Detta är ett av de få fall då vi måste skriva en egen destruktor.

```
IntArray:: ~IntArray()
{
  delete [] arr;
  arr = NULL;
  cout << endl << "Running destructor for IntArray with maxSize= " << maxSize <<
endl;
}
```

Kommentarer:

- Plats allokerad med [] tas bort med [].
- Destruktorn körs när ett objekt tas bort, vanligtvis när programmet stängs eller funktionen avslutas
- Sist skapat objekt tas bort först (stack!)
- Texten är med enbart för att vi ska se när destruktorn körs. Den ska inte vara med när klassen används sedan.

3.18.4 Läs ett värde från arrayen

```
int IntArray::getValue(int idx) const
{
  return arr[idx];
}
```

Kommentarer:

- `arr[idx]` och `*(arr+i)` är olika notation för samma sak
- Man borde testa så att idx ligger i tillåtet intervall, 0..size. Det finns dock inget vettigt värde att returnera om idx ligger utanför tillåtet intervall. Istället bör man skriva ut ett felmeddelande. Felhantering väntar vi med till nästa kurs så vi får använda funktionen med tillåtna värden.

3.18.5 Lägg in ett värde i arrayen

```
bool IntArray::addValue(int value)
{
  bool valueAdded = false;
  if(size < maxSize)
  {
    arr[size] = value;
    size++;
    valueAdded = true;
  }
  return valueAdded;
}
```

Kommentarer:

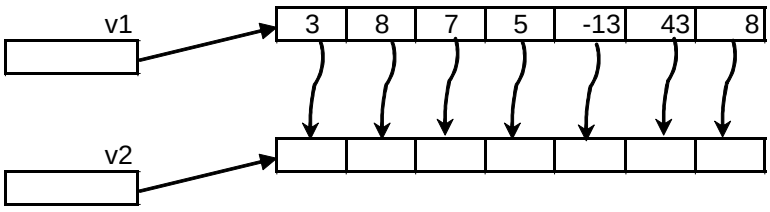
- Ett heltal läggs in på första lediga plats i arrayen
- Test görs så att det finns plats innan talet läggs till i arrayen!
- Om plats finns och talet lagts in returneras *true* annars *false*

3.18.6 Kopieringskonstruktorn

```
IntArray::IntArray (const IntArray &a): maxSize(a.maxSize), size(a.size)
{
  arr = new int[a.maxSize];
  for (int i=0; i<a.size; i++)
    arr[i] = a.arr[i];
}
```

Kommentarer:

- En ny array skapas. Den är en kopia av **a** som ges som parameter.
- Syntax för en kopieringskonstruktor: `KlassNamn(const KlassNamn &kn);`
- Se till att **alla** datamedlemmar kopieras. I detta fall kopieras `size` och `maxSize` i initieringslistan.
- Den nya arrayen tilldelas data element för element, s.k. djup kopiering. Se figur nedan.
- Om man inte skriver en kopieringskonstruktor skapas en kopieringskonstruktor som kopierar objekten datamedlem för datamedlem, s.k. grund kopiering. I det här fallet är det enbart vektorns adress, INTE arrayens värden.
- REGEL: Skapar man utrymme dynamiskt för en datamedlem ska man **alltid** skriva en egen kopieringskonstruktor.
- Exempel på programkod när kopieringskonstruktorn körs:
  - IntArray a2= a1;
  - IntArray a3(a2);
  - IntArray a4 = IntArray(a2);
  - IntArray *a5 = new IntArray(a2);



Figur: Djupkopiering

3.18.7 Tilldelningsoperatorn

Om man överlagrar tilldelningsoperatorn kan man tilldela en array till en annan:

```
a1 = a2 ;
```

Koden för tilldelningsoperatorn för klassen IntArray ser ut så här:

```
const IntArray &IntArray::operator=(const IntArray &a)
{
  if(this != &a)
  {
    delete []arr;
    arr     = new int[a.maxSize];
    maxSize = a.maxSize;
    size    = a.size;
    for (int i=0; i < a.size; i++)
      arr[i] = a.arr[i];
  }
  return *this;
}
```

Kommentarer:

- Kolla första att a1 inte är samma objekt som a2 (adresserna kollas)
- Den befintliga arrayen tas bort, dvs. platsen för det aktuella (vänstra) objektet frigörs
- Skapa plats för ett nytt objekt
- Kopiera datamedlemmarna från ”parameterobjektet” till det aktuella
- Kopiera sedan element för element från ”parameterobjektet” till det aktuella (djupkopiering!)
- Returnera aktuellt objekt (*this –pekaren). Man gör detta för at kunna skriva `a3=(a1=a2)` eller `a3==(a1=a2)`.
- Man returnerar ***this** (objektet som sådant) och inte **this** (adressen till objektet)
- Resultattypen är en konstant till en referens. Detta innebär att en referens till det aktuella objektet kommer att returneras. Man konstantdeklarerar returvärdet för att man inte ska kunna skriva `(a1 = a2) = a3`.
- Resultattypen hade också kunnat varit IntArray, men då hade en kopia av det aktuella objektet returnerats, vilket är mindre effektivt.
- TIPS: Skapar man utrymme dynamiskt för en datamedlem ska man **alltid** skriva en egen överlagring av tilldelningsoperatorn.

3.18.8 This-pekaren

this-pekaren pekar på det anropande objektet (adressen)
***this** avser detta objekt som helhet (innehållet)

Provkör exempel oload_100 med och utan kopieringskonstruktor och tilldelningsoperator.Vad som händer?

Exempel oload_100:

```
// oload_100 Version 10
// En arrayklass för heltal (int). Visar användning av pekare som datamedlem.
// Per Ekeroot 2014-01-13
//---------------------------------------------------------------------------
#include "intarray1.h"
#include <iostream>
using namespace std;
```

```
//-----------------------------------------------------------------------------
// Funktionsprototyper
//-----------------------------------------------------------------------------
int sum(IntArray const &a);
IntArray square(IntArray const &a);

//-----------------------------------------------------------------------------
// Huvudprogram
//-----------------------------------------------------------------------------
int main()
{
  // Deklarera och initiera konstant och variabler
  const size_t MAX_SIZE = 10;
  size_t antal = 0;
  int  x = 0;

  cout << " Input number of values: ";
  cin >> antal;

  IntArray a1(MAX_SIZE);

  cout << " Input values:" << endl;
  for (int i=0; i < antal; i++)
  {
     cout << " Nr " << i + 1 << ": " ;
     cin >> x;
     a1.addValue(x);
  }

  IntArray a2 = a1;     // Kopieringskonstruktorn körs
  a2 = square(a2);      // Kvadrera varje element i a2

  cout << " Sum       = " << sum(a1) << endl;
  cout << " Squaresum = " << sum(a2) << endl;

//-----------------------------------------------------------------------------
// Test av destruktorn
//-----------------------------------------------------------------------------
 {
  IntArray a3(5);
  IntArray a4(7);
  IntArray a;
 }

//-----------------------------------------------------------------------------
// Test av tilldelningsoperatorn
//-----------------------------------------------------------------------------
 IntArray aa;
 aa = a1;  // Tilldelningsoperatorn körs
 cout << endl << "Values assigned with the assignment operator (aa = a1)" << endl;
 for(size_t i=0; i<aa.getSize(); i++)
   cout << aa.getValue(i) << " ";
 cout << endl << endl;

 return 0;
}
// Forts på nästa sida!
```

```
//-----------------------------------------------------------------------------
// Funktionsdefinitioner
//-----------------------------------------------------------------------------
// sum
// Beräkna summa av elementen i arrayen a
//-----------------------------------------------------------------------------
int sum(IntArray const &a)
{
  int s = 0;
  for(size_t i=0; i < a.getSize(); i++)
    s+= a.getValue(i);
  return s;
}
//-----------------------------------------------------------------------------
// square
// Kvadrera varje enskilt element i arrayen a
//-----------------------------------------------------------------------------
IntArray square(IntArray const &a)
{
  IntArray tmpArr(a.getMaxSize());// Skapa en temporär array med maxSize = a.maxSize
  for(size_t i=0; i < a.getSize(); i++)
  {
    int x = a.getValue(i);
    tmpArr.addValue(x*x);
  }
  return tmpArr;
}
```

Kommentarer:

- Inmatning till arrayen görs via en temporär variabel:
  ```
  cin >> x;
  a1.addValue(x);
  ```
- Kopieringskonstruktorn körs då ett nytt objekt skapas och initieras med ett befintligt objekt
  ```
  Array a2 = a1;
  ```
- Ändra sum-funktionen så att ett värdeanrop görs. Vad händer? Jo, destruktorn körs helt plötsligt två gånger. Varför? Jo, vid värdeanrop så görs en kopia av det anropande objektet. Detta objekt lever så länge som funktionen finns till och tas bort då funktionen avslutas. Då objektet tas bort körs destruktorn.

3.19 Implicita (underförstådda) medlemsfunktioner

Implicita medlemsfunktioner skapas automatiskt när man gör en klass. C++ skapar följande medlemsfunktioner automatiskt om man inte har skrivit dem själv:

- Förvald konstruktor
- Kopieringskonstruktor
- Tilldelningsoperator
- Destruktor
- Adressoperator

Vad gör dessa automatiskt skapade medlemsfunktioner:

Medlemsfunktion	**Utför**
• Förvald konstruktor	ingenting, den är tom
• Kopieringskonstruktor	kopierar datamedlemmarna medlem för medlem
• Tilldelningsoperator	tilldelar datamedlemmarna medlem för medlem
• Destruktor	ingenting, den är tom
• Adressoperatorn	returnerar adressen till det aktiverande objektet

(this-pekaren)

3.20 En förbättrad IntArrayklass

Arrayklassen i föregående exempel är ganska klumpig, man måste t.ex. använda getValue(i) för att läsa värden från arrayen. Skulle man inte kunna göra detta elegantare? Jo, vi överlagrar indexoperatorn ([ ]).

Det finns andra klumpigheter som vi också ska försöka råda bot på genom att överlaga ett antal operatorer. Detta visas i exempel oload_110 som använder array2.h och array2.cpp.

Operator		**Användning**
+	addition	a3 = a1 + a2
==, !=	likhet, olikhet	a1 == a2, a1 != a2
[]	indexoperatorn	cout << a1[4], a1[2] = 7

Vi bygger på den gamla klassen med ett antal operatoröverlagringar och får klassdefinitionen för array2.h :

```
//  array2.h Version 10
//  Headerfil till klassen IntArray (uppgraderad version)
//  Per Ekeroot 2014-01-13
//-----------------------------------------------------------------------------
#ifndef IntArray2H
#define IntArray2H
  class IntArray
  {
    private:
      // Datamedlemmar
      int *arr;     // Pekare till arrayen
      int maxSize;  // Arrayens maximala storlek
      int size;     // Aktuellt antal element i arrayen

    public:
      // Konstruktorer och destruktor
      IntArray();
      IntArray (int pMaxSize);
      IntArray (const IntArray &a);
      ~IntArray();

      // get-funktioner
      int getMaxSize() const {return maxSize;}
      int getSize() const {return size;}

      // Lägg in data
      bool addValue(int value);

      // Överlagring av operatorer
      const IntArray &operator=(const IntArray &a);
      IntArray operator+(const IntArray &a) const;
      bool operator==(const IntArray &a) const;
      bool operator!=(const IntArray &a) const;
      int &operator[](int i);
      int operator[](int i) const;
  };

  void showIntArray(const IntArray &a);
  void inputIntArray(IntArray &a);
#endif
```

Implementation av medlemsfunktionerna följer nedan. Provkör exempel oload_110 i anslutning till genomgång av resp. medlemsfunktion.

3.20.1 Addera-operatorn

`a = a1 + a2;`

```
IntArray IntArray::operator+(const IntArray &a)const{
  if(size == a.size)
  {
    IntArray temp(maxSize); //Skapa en tom kopia med samma maxSize som denna array
    for( size_t i=0; i<size; i++)
      temp.addValue(arr[i] + a.arr[i]);

    return temp;
  }
  else
    return *this;// Returnera det aktuella objektet om arrayerna är olika långa}
```

Kommentarer:

- Här returneras ett nytt objekt, summan av a1 + a2
- Kolla först att vektorerna är lika långa
- `IntArray temp(maxSize)` skapar ett nytt objekt (temp), kopiera sedan det aktuella objektet till detta: `temp.addValue(arr[i] + a.arr[i]);`.
- Returnera adressen till det nya objektet, d.v.s. en kopia av temp.
- Det går inte att returnera en referens till temp, eftersom temp endast existerar inne i funktionen.
- Det går inte att returnera en referens till det aktuella objektet eftersom detta inte ska ändras (`a = a1 + a2`).

3.20.2 Likhetsoperatorn

`if(a1==a2)`

```
bool IntArray::operator==(const IntArray &a) const
{
  if(size != a.size)
    return false;
  for ( int i=0; i<size; i++)
    if( arr[i] != a.arr[i])
      return false;
  return true;
}
```

Kommentarer:

- Returnera **false** om arrayerna är olika långa
- Kolla likhet elementvis. Returnera **false** om två element är olika.

3.20.3 Olikhetsoperatorn

`if(a1 != a2)`

```
bool IntArray::operator!=(const IntArray &a) const
{
  return !((*this)==a);
}
```

Kommentarer:

- Utnyttja att likhetsoperatorn (==) redan är definierad i klassen.

3.20.4 Indexoperatorn

Överlagra indexoperatorn för läsning **och** skrivning:

```
int k = a1[2];  a1[1] = 23;
```

```
int& IntArray::operator[] (int idx)
{
  return arr[idx];
}
```

Kommentarer:

- Returtyp är **referens** till int. Detta innebär att man kan skriva a`[i+1] = a[i]`, d.v.s. kopiera data från element till ett element **inom** arrayen. Om man bara returnerat en **int** skulle man ha returnerat en kopia av det utvalda elementet och därmed inte haft möjlighet att ändra det aktuella objektets element.
- I vissa lägen (t.ex. om man vill skriva int `sum(const Array a)`) måste man ha en variant `int Array::operator[] (int idx)` (utan referens) för att kunna const-deklarera arrayen a.

Överlagra indexoperatorn **enbart** för läsning. Denna variant behövs om man vill skicka med en **const** Array som argument till en funktion.

```
int k = a1[3];
```

```
int IntArray::operator[] (int idx)  const
{
  return arr[idx];
}
```

Resterande medlemsfunktioners implementationer utan kommentarer:

```
//--------------------------------------------------------------------------
// Skriv ut arrayen på skärmen
//--------------------------------------------------------------------------
void showIntArray(const IntArray &a)
{
  cout << '[';
  if(a.getSize()>0)
    cout << a[0];

  for (int i= 1; i < a.getSize(); i++)
    cout << ", " << a[i];
  cout << ']';
}
//--------------------------------------------------------------------------
// Mata in data i arrayen
//--------------------------------------------------------------------------
void inputIntArray(IntArray &a)
{
  for(int i=0; i < a.getMaxSize(); i++)
  {
    int tmpInt;
    cout << "Element nr " << (i + 1) << " ";
    cin >> tmpInt;
    a.addValue(tmpInt);
  }
}
```

3.20.5 Testprogrammet oload_110

```
// oload_110 Version 10
// En förbättrad arrayklass. Visar användning av några överlagrade operatorer
// Per Ekeroot 2014-01-13
//-----------------------------------------------------------------------------
#include "intarray2.h"
#include <iostream>
using namespace std;

//-----------------------------------------------------------------------------
// Funktionsprototyp
//-----------------------------------------------------------------------------
int sum(const IntArray &a);

//-----------------------------------------------------------------------------
int main()
{
//-----------------------------------------------------------------------------
// Skapa tre IntArray-objekt
//-----------------------------------------------------------------------------
  IntArray a;        // Förvald konstruktor körs
  IntArray a1(3);    // En array med 3 element skapas
  IntArray a2(3);    //   -"-

//-----------------------------------------------------------------------------
// Mata in data till arrayerna och skriv ut dem
//-----------------------------------------------------------------------------
   cout << "Input a1 :" << endl;
   inputIntArray(a1);

   cout << endl << "Input a2 :" << endl;
   inputIntArray(a2);
            cin.get();

   cout << endl <<"Print the arrays" << endl;
   cout << endl << " a1= "; showIntArray(a1);
   cout << endl << " a2= "; showIntArray(a2);
   cout << endl << " a = "; showIntArray(a);
   cout << endl;

   cout << "Press ENTER to continue!" ;
   cin.get();
   cout << endl << endl;

//-----------------------------------------------------------------------------
// Test av additionsoperatorn
//-----------------------------------------------------------------------------
   a = a1 + a2;
   cout << "a = a1+a2 ger a = ";
   showIntArray(a);
   cout << endl << endl;
   cout << "Press ENTER to continue!" ;
   cin.get();
   cout << endl << endl;

//-----------------------------------------------------------------------------
// Test av likhetsoperatorn
//-----------------------------------------------------------------------------
   cout << endl << "Equal operatorn in work" << endl;
   if(a1==a2)
     cout << " equal" << endl << endl;
   else
```

```
     cout << " not equal" << endl << endl;

   cout << endl << endl;
   cout << "Press ENTER to continue!" ;
   cin.get();
   cout << endl << endl;

//-------------------------------------------------------------------------------
// Test av indexoperatorn
//-------------------------------------------------------------------------------
   cout << endl << "Index operator is working:" << endl;
   cout << "a[2] = "<< a[2] << endl <<endl;

   a[1] = -35;
   cout << endl << "Index operator works again:" << endl;
   cout << "a[1] = " << a[1]<< endl << endl;

   a[2] = a[1];
   cout << "After assignment a[2]= a[1] ger a[2]= " <<a[2];
   cout << endl << endl;

// Testa sum() dvs den 2:a överlagringen av [] som finns i sum()
   cout << endl << "Sum av the elements in a1 = " << sum(a1);
   cout << endl << endl;

   cout << " a1= ";
   showIntArray(a1);
   cout << endl << endl;

//-------------------------------------------------------------------------------
//  Test av att tilldelningsoperatorn (=) returnerar en referens till sig själv
//-------------------------------------------------------------------------------
   a = a1 = a2;
   cout << "a = (a1=a2) ger a = ";
   showIntArray(a);
   cout << endl << endl;
   cout << "Press ENTER to continue!" ;
   cin.get()

   return0;
}

//-------------------------------------------------------------------------------
// sum
// Summera elementen i arrayen
//-------------------------------------------------------------------------------
int sum(const IntArray &a)
{
   int sum =0;
   for(int i=0; i < a.getSize(); i++)
     sum += a[i];
   return sum;
}
```

Kommentarer:

- Funktionen `sum(const Array &v)` kräver att det finns en version av indexoperatorn som är const-deklarerad d.v.s. att det enbart när tillåtet att läsa från vektorn.
- Testa att köra sum( ) **utan** den 2:a varianten av []-överlagringen

3.21 Statiska klassmedlemmar

En statisk klassmedlem är gemensam för klassen, dvs för alla objekt som instansieras av klassen. Både datamedlemmar och medlemsfunktioner kan vara statiska.

Man t.ex. använda en statisk variabel för att räkna antal objekt som har skapats av en klass.
Vi ska använda vår första klass Account i för att visa hur man kan använda statiska medlemmar.

Vi utvidgar klassen med

- en statisk datamedlem som lagrar räntesats.
- en statisk medlemsfunktion för att läsa räntesatsen
- en statisk medlemsfunktion för att sätta räntesatsen

Dessutom lägger vi till några ”vanliga” medlemmar

- en datamedlem som lagrar summerad ränta under året
- en medlemsfunktion som beräknar dagsränta
- en medlemsfunktion som lägger in årsräntan till saldot

Den nya klassen ligger i filerna account2.h och account2.cpp.

```
class Account
  {
    private:
      string firstName;            // Datamedlemmar
      string lastName;
      string accountNr;
      double balance;
//-------------------------------------------------------------------------
// Nytt i account2
//-------------------------------------------------------------------------
      static double interest;
      double interestSum;
//-------------------------------------------------------------------------
    public:
      Account();                                    //Förvald konstruktor
      Account(string pFirstName, string pLastName, //Initieringskonstruktor
              string pAccountNr,double pBalance);

      void deposit(double amount);                  // Medlemsfunktioner
      void withdrawal(double amount);

      void setFirstName(string pFirstName); // Medlemsfunktioner som sätter
      void setLastName(string pLastName);   // värden på datamedlemmar
      void setAccountNr(string pAccountNr);
      void setBalance(double pBalance);

      string getFirstName() const;          // Medlemsfunktioner som läser
      string getLastName() const;           // datamedlemmmars värden
      string getAccountNr() const;
      double getBalance() const;
//-------------------------------------------------------------------------
// Nytt i account2
//-------------------------------------------------------------------------
      void calcDayInterest();
      void calcAnnualInterest();
      static double getInterest() {return interest; }
      static void setInterest(double pInterest);
  };
  void showAccount(Account const &a);
```

Kommentarer:

- Det reserverade ordet **static** används för att definiera en statisk medlem.
- Den ena statiska medlemsfunktionen är inline och den andra definieras i cpp-filen. Jag gör på olika sätt av demonstrationsskäl.

Initiering av den statiska datamedlemmen görs i definitionsfilen (cpp-filen)!! Sedan följer definitionerna av de nya medlemsfunktionerna:

```
//-----------------------------------------------------------------------------
// Definition och initiering av den statiska datamedlemmen 'interest'
// 'interest' deklareras i klassen Account
//-----------------------------------------------------------------------------
double Account::interest = 0.0;

//-----------------------------------------------------------------------------
// Statisk medlemsfunktion som sätter nytt värde på räntesatsen
// Ordet 'static' anges enbart i klassdefinitionen
//-----------------------------------------------------------------------------
void Account::setInterest(double pInterest)
 {
   interest = pInterest;
 }
//-----------------------------------------------------------------------------
// Medlemsfunktion som beräknar och sparar dagsräntan. Ska köras dagligen!
//-----------------------------------------------------------------------------
void Account::calcDayInterest()
{
  interestSum += balance*interest/100/360;
}
//-----------------------------------------------------------------------------
// Medlemsfunktion som "beräknar" årsräntan. Ska köras en gång per år (1 januari)
//-----------------------------------------------------------------------------
void Account::calcAnnualInterest()
{
  balance += interestSum;
  interestSum = 0.0;
}
```

I huvudprogrammet används de nya medlemmarna på följande sätt (oload_120):

```
// oload_120 Version 10
// Visar användning av statiska medlemmar i en klass
// Per Ekeroot 2014-01-13
//-----------------------------------------------------------------------------
#include "account2.h"
#include <iostream>
using namespace std;

//-----------------------------------------------------------------------------
int main()
{
//-----------------------------------------------------------------------------
// Skapa och initiera två konto-objekt
//-----------------------------------------------------------------------------
  Account a1("Eva","Svensson","10 20 30", 1000);
  Account a2("Sven","Eriksson","10 20 33", 10000);

//-----------------------------------------------------------------------------
// Sätt ett värde på räntan (statisk datamedlem)
//-----------------------------------------------------------------------------
  Account::setInterest(3.6);
```

```
//-------------------------------------------------------------------------
// Skriv saldobesked för två konton
//-------------------------------------------------------------------------
  showAccount(a1);
  cout << endl;
  showAccount(a2);
  cout << endl;

//-------------------------------------------------------------------------
// Beräkna ränta dag för dag under ett år och sätt in årsränta på resp konto
//-------------------------------------------------------------------------
  for(int i=0; i<360; i++)
  {
    a1.calcDayInterest();
    a2.calcDayInterest();
  }

  a1.calcAnnualInterest();
  a2.calcAnnualInterest();

//-------------------------------------------------------------------------
// Skriv saldobesked för två konton
// Skriv räntesatsen
//-------------------------------------------------------------------------
  showAccount(a1);
  cout << endl;
  showAccount(a2);
  cout << endl;
  cout << "Interest: " << Account::getInterest()<< endl << endl;

  return 0;
}
```

Kommentarer:

- En statisk medlemsfunktion anropas med klassnamnet som kvalificerare: `Account::setInterest(2.6);` och `Account::getInterest();`
- `showAccount()` är en fristående funktion för utskrift av Account-objekt. Du hittar dess implementation i filen account.cpp

3.22 Pekare till objekt

Följande exempel (oload_130) visas hur man kan

- skapa en pekare som allokerar plats för ett objekt på heapen

I exemplet används klassen Time som finns i time5.h och time5.cpp

Följande kod visar hur man allokerar plats för objekt på heapen.

```
 Time *pTime1 = new Time;           // Default konstruktor körs
 Time *pTime2 = new Time(12,34);    // Initieringskonstruktor körs
```

Kommentarer:

- I första fallet körs default constructor
- I andra fallet körs initieringskonstruktorn

Sedan visar vi hur man hanterar medlemsfunktioner till pekarobjekt. Skapa en pekare till ett Time-objekt och använd medlemsfunktionerna setHour() och setMin().

```
  pTime1->setHour(11);
  pTime1->setMin(12);
  . . .
  cout << "Time2: Hour= " << pTime2->getHour() << " Minute= " << pTime2->getMin();
```

Kommentarer:

- Operatorn `->` används när man vill använda medelmsfunktioner till pekarobjekt. Man skriver alltså: `pTime1->setHour(11);`
- Man kan också använda det ”vanliga” skrivsättet med . (punkt). Men av prioritetsskäl måste man då använda skrivsättet (`*pTime1).setHour(11)`. Man måste använda parenteserna eftersom '**.**' (punkt) har högre prioritet än '*'.

Hela programmet oload_130

```
// oload_130 Version 10
// Använd pekare till objekt.
// Per Ekeroot 2014-01-13
//-----------------------------------------------------------------------------
#include "time5.h"
#include <iostream>
using namespace std;
//-----------------------------------------------------------------------------
int main()
{
//-----------------------------------------------------------------------------
// Använd medlemsfunktioner för pekarobjekt
//-----------------------------------------------------------------------------
  cout << "Use member functions for pointers to objects" << endl;
  Time *pTime1 = new Time;           // Default konstruktor körs
  Time *pTime2 = new Time(12,34);    // Initieringskonstruktor körs
  showTime(*pTime1);
  cout << endl;

  // Använd klassen Times setfunktioner
  pTime1->setHour(11);
  pTime1->setMin(12);
  showTime(*pTime1);

  cout << endl;
  showTime(*pTime2);
```

```
  Time *pTime3 = new Time;
  *pTime3 = *pTime1 + *pTime2;
  cout << endl << "pTime3 = ";
  showTime(*pTime3);
  cout << endl << endl;

  // Använd klassen Times getfunktioner
  cout << "Time2: Hour= " << pTime2->getHour() << " Minute= " << pTime2->getMin();
  cout << endl << endl;

  delete pTime1, pTime2, pTime3;

  return 0;
}
```