4 Operatoröverlagring

De egendefinierade objekten ska *så mycket som möjligt likna de fördefinierade variablerna* i C++. När det exempelvis gäller heltalsvariabler av typen int kan man utföra addition med hjälp av additionsoperatorn, + enligt :

```
int a = 2, b = 3, c;

c = a + b;
```

Detta skulle man även vilja göra för sina egendefinierade objekt. Har man en klass som ska avbilda rationella tal så vill man använda +-operatorn, *-operatorn osv istället för att anropa funktioner som add, mul.

Ex : Klassen RTAL avbildar rationella tal.

```
// specifikation av klassen RTAL - rtal.h

class RTAL
{
   private :
      int t;           // täljare
      int n;           // nämnare
   public :
      void las();
      void skriv();
      RTAL forkorta();
      RTAL add(RTAL rt);
      RTAL sub(RTAL rt);
      RTAL mul(RTAL rt);
      RTAL div(RTAL rt);
};
```

Ett huvudprogram som exv beräknar 1/2 + 1/3*1/4 ser då ut som,

```
#include "rtal.h"

void main()
{
   RTAL a, b, c, r;

   a.las();
   b.las();
   c.las();

   r = a.add(b.mul(c));

   r.skriv();
}
```

I exemplet ovan skulle man istället vilja beräkna r enligt :

```
r = a + b*c;
```

Ett operatoranrop som (a + b) i C++ är egentligen ett funktionsanrop, operator+(a, b). Man anropar funktionen operator+ med parametrarna a och b. Det finns ett antal *överlagrade* operator+-funktioner för de enkla fördefinierade typerna,

```
int operator+(int, int);
float operator+(int, float);
float operator+(float, int);
.....
```

Här ser man också hur det går till att konvertera resultatet till den ' högre typen'. Exempelvis översätter kompilatorn uttrycket

```
2 + 3.4
```

till ett anrop av den *andra* överlagrade funktionen ovan enligt :

```
operator+(2, 3.4)
```

Funktionen omvandlar sedan 2 till ett flyttal, summerar flyttalen och returnerar *flyttalet* 5.4.

Ex : Skriv om klasssen RTAL så att den använder sig av operatorer för de fyra räknesätten istället för funktioner samt skriv ett huvudprogram som utför någon beräkning med rationella tal.

```
// specifikation av klassen RTAL -- rtal.h

class RTAL
{
   private :
      int t;
      int n;
   public :
      void las();
      void skriv();
      RTAL forkorta();
      RTAL operator+(RTAL rt);
      RTAL operator-(RTAL rt);
      RTAL operator*(RTAL rt);
      RTAL operator/(RTAL rt);
};
```

```
// implementation av klasssen RTAL -- rtal.cpp

#include "rtal.h"
.....

RTAL RTAL::operator+(RTAL rt)
{
   RTAL res;

   res.t = t*rt.n + n*rt.t;
   res.n = n*rt.n;
   return res.forkorta();
}
....

RTAL RTAL::operator*(RTAL rt)
{
   RTAL res;

   res.t = t*rt.t;
   res.n = n*rt.n;
   return res.forkorta();
}

// huvudprogram som exv beräknar 1/2 + 1/3*1/4

#include "rtal.h"

void main()
{
   RTAL a, b, c, r;

   a.las();
   b.las();
   c.las();
   r = a + b*c;
   r.skriv();
}
```

Det finns ett antal regler för hur operatoröverlagring får gå till :

1) Operatorer för fördeklarerade typer som int, float etc *får ej* omdefinieras. Man kan alltså inte definiera om +-operatorn för int till att utföra subtraktion av heltal istället.

2) Man kan bara definiera om *existerande* operatorer. Man kan alltså inte hitta på egna operatorer som att <> skulle betyda 'skilt ifrån'.

3) De regler som gäller för en operators *prioritet* (vad som beräknas först om ej parenteser) och *associativitet* (i vilken ordning om samma prioritet) gäller också för den omdefinierade operatorn. I ovanstående exempel har *-operatorn högre prioritet än +-operatorn och därför utförs multiplikationen först.

4) Vi kan ej påverka antalet operander som en operator tar. En binär operator, som +, måste ha två operander och en unär operator som ! måste ha en operand.

Ex : Tillbaka till den tänkta implementationen av klassen ifstream. Där finns en överlagrad unär operator, !, som testar om en fil går att öppna eller inte.

```
// specifikation av klassen ifstream -- ifstream.h

class ifstream
{
   private:
      FILE *file;
      ....
   public:
      ifstream(char *filename);
      ~ifstream();
      int operator!();
      ....
};


// implementation av klassen ifstream -- ifstream.cpp

#include "ifstream.h"

ifstream::ifstream(char *filename)
{
   file = fopen(filename, "r");
}

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

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


// huvudprogram

#include "ifstream.h"

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

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

4.1 Friendfunktioner

Till en överlagrad binär operator-funktion, som +-funktionen ovan, skickar man bara den högra operanden som argument. Den *vänstra operanden är själva objektet* vars medlemsfunktion man anropar. På motsvarande sätt skickas inget explicit argument till den unära operatorn !. Denna operator tar bara en operand och det är själva objektet vars medlemsfunktion man anropar.

Anropet

```
b*c
```

är egentligen ett anrop som ser ut som

```
b.operator*(c)
```

Ska man överlagra operatorn << och >> för ut- resp inmatning av rationella tal får man problem eftersom den vänstra operanden i exempelvis

```
cout << a   ( eller cout.operator<<(a) )
```

är ett objekt av klasssen ostream. Man måste in i klassen ostream och lägga in en ny överlagring av operatorn << , som gäller för klassen rationella tal. Detta ska man helst undvika eftersom de fördefinierade klasserna ej bör ändras. Istället skulle man vilja ha en överlagrad utskriftsfunktion i klassen RTAL, men då kan den *inte vara en medlemsfunktion eftersom den vänstra operanden ej är ett rationellt tal och är den inte medlemsfunktion så kommer den inte åt data för RTAL.*

Man har på grund av ovanstående infört begreppet *friendfunktion* i C++. I en klass kan man i specifikationen ange vilka funktioner eller operatorer som är friend till klassen. Dessa friendfunktioner blir alltså inte medlemsfunktioner men *de kommer att ha tillgång till klassens privata data*.

Ex : Gör om inläsningsfunktionen las och utmatningsfunktionen skriv i klassen RTAL så att man använder de överlagrade operatorerna >> resp << istället.

```
// specifikation av klassen RTAL --rtal.h
#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);
      RTAL forkorta();
      RTAL operator+(RTAL rt);
      RTAL operator-(RTAL rt);
      RTAL operator*(RTAL rt);
      RTAL operator/(RTAL rt);
};
```

```
// implementation av klassen RTAL -- rtal.cpp

#include "rtal.h"

.....

istream &operator>>(istream &in, RTAL &rt)     // OBS! Ej RTAL::
{
   char c;

   cout << "Ge ett bråk på formen a/b : ";
   in >> rt.t >> c >> rt.n;
   return in;
}


ostream &operator <<(ostream &out, RTAL rt)// OBS! Ej RTAL::
{
   if ( rt.n != 1 )
      out << rt.t << "/" << rt.n << endl;
   else
      out << rt.t << endl;
   return out;
}

......

// huvudprogram som exv beräknar 1/2 + 1/3*1/4

#include "rtal.h"

void main()
{
   RTAL a, b, c;

   cin >> a >> b >> c;
   cout << a + b*c;

}
```

OBS! I inmatningsfunktionen måste vi ha referensparameter (&rt) för det rationella talet som ska läsas in annars läser man in värdet till en kopia. För att spara minne och slippa kopiering kan man även använda referens för det rationella talet i utskriftsfunktionen. Gör man detta bör man för säkerhets skull använda en *const* referens så att argumentet inte kan förändras av misstag inuti funktionen

OBS! Även in- och utmatningsobjekten tas in som referenser eftersom det är viktigt att status hos de aktuella parametrarna blir detsamma som hos de formella. Även funktionernas returvärden är referenser så att man får tillbaka det objekt man anropar med och kan kedja ihop flera >>-operatorer som i huvudprogrammets cin >> a >> b >> c. Här returnerar cin >> a objektet cin.

OBS! En binär operator har *två* argument om den är skriven som friendfunktion.

4.2 Tilldelning

Till alla klasser, som man definierat själv, finns en *fördefinierad tilldelningsoperator* som utför medlemsvis kopiering. I exemplet med de rationella talen ovan användes den i satsen

```
r = a + b*c
```

Här beräknas det högra objektets medlemsvariabler, täljare och nämnare och dessa kopieras över till motsvarande medlemsvariabler i r. Ibland kanske man vill ha eller måste man ha en annan typ av tilldelning där exempelvis bara vissa medlemmar kopieras över. Då får man göra en egen överlagrad tilldelningsoperator.

Ex : Avbilda en medlem med medlemsnummer och namn. Minne för namnet ska allokeras dynamiskt.

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

class MEDLEM
{
   private:
      int mnr;          // Medlemsnummer
      char *namn; // Medlemmens namn

   public:
      MEDLEM(int mnr = 0, char *namn = ””);
      ~MEDLEM();

      ....
};



// skapa två objekt
MEDLEM m1(102, ”A.Andersson”);
MEDLEM m2;


// tilldela m1 till m2
m2 = m1;
```

Nu har man fått två medlemmar som har samma värden på alla medlemsvariablerna. Speciellt kommer pekarna i de båda objekten att peka på samma namn, vilket är *riskabelt då man med den ena pekaren kan avallokera minnesutrymme medan den andra pekaren blir kvar och pekar då på avallokerat minne*. Detta bäddar för problem. Man kan även på detta sätt få system som *äter minne eftersom man ger pekare nya värden utan att ha avallokerat utpekat utrymme först*. För att slippa ifrån detta ska man definiera en egen tilldelningsoperator i MEDLEM-klassen, där man ser till att man tillverkar verkliga kopior även när det gäller namnet.

Regel : **Man ska definiera en egen tilldelningsoperator om man har pekare som pekar på dynamiskt allokerat minne, som medlemsvariabler i sin klass.**

```
// specifikation av klassen MEDLEM – medlem.h
class MEDLEM
{
   private:
      ....

   public:
      ....

      MEDLEM &operator=(const MEDLEM &m);
      ....
};


// implementation av klassen MEDLEM – medlem.h
#include "medlem.h"
#include <cstring>
using namespace std;

.....

MEDLEM &MEDLEM::operator=(const MEDLEM &m)
{
   mnr = m.mnr;
   delete [] namn;
   namn = new char[strlen(m.namn) + 1];
   strcpy(namn, m.namn);
   return *this;
}
.....
```

OBS! Nu avallokeras gammalt namn och nytt namn skapas för m2 vid tilldelningen m2 = m1.

OBS! Tilldelningen m2 = m1 är egentligen ett anrop m2.operator=(m1). Det dolda argumentet, alltså det objekt som anropas, är m2 medan m1 skickas som argument.

OBS! Operatorfunktionen tilldelning ges ett returvärde i form av det vänstra objektet med hjälp av *this-pekaren*. Detta är ej nödvändigt men vill man ha samma funktion på tilldelning som vid tilldelning mellan fördeklarerade variabler så ska en tilldelning som m2 = m1 returnera m2 så att exempelvis tilldelningar av typen

```
m3 = m2 = m1;
```

är möjliga. Eftersom tilldelningsoperatorn är högerassociativ kommer uttrycket att beräknas från höger så att m2 får m1.s värde och returnerar m2-objektet. Sedan får m3 värdet av m2 och returnerar m3-objektet. Hela uttrycket returnerar ett m3-objekt med värden från m1.

4.3 Initiering.

Precis som det finns en fördefinierad tilldelningsoperator finns *initiering fördefinierad* för alla objekt. Initiering är en annan typ av operation än tilldelning. Vid initiering *skapas* ett objekt och *när ett objekt skapas anropas alltid en konstruktor*. Till alla klasser finns alltså en fördeklarerad konstruktor som utför initiering. Denna konstruktor kallas initieringskonstruktor eller *copykonstruktor*. Den fördefinierade copykonstruktorn tar ett objekt av samma klass som argument och returnerar ett nytt objekt med samma värden på alla datamedlemmar. Eftersom denna konstruktor kopierar över alla medlemmar uppstår det samma problem som vid tilldelning, om man har pekare som datamedlemmar. Man får pekare som pekar på redan avallokerat utrymme. Detta kan man undvika om man definierar en egen överlagrad copykonstruktor.

Regel : **Man ska definiera en egen copykonstruktor, om man har pekare som pekar på dynamiskt allokerat minne, som medlemsvariabler i sin klass.**

Ex: Den fördefinierade copykonstruktorn för MEDLEM-klassen utför medlemsvis kopiering dvs alla medlemsvariabler kopieras, även namn-pekarna. Definiera en egen överlagrad copykonstruktor som skapar en riktig kopia även av namnen.

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

class MEDLEM
{
   private:
      int mnr;          // Medlemmens nummer
      char *namn; // Medlemmens namn

   public:
      ....
      MEDLEM(const MEDLEM &m);      // copy-konstruktor
};


// implementation av klassen MEDLEM – medlem.cpp

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

.....

MEDLEM::MEDLEM(const MEDLEM &m)
{
   mnr = m.mnr;
   namn = new char[strlen(m.namn)+1];
   strcpy(namn, m.namn);
}
.....
```

```
.....

// skapa en medlem
MEDLEM m1(106, ”A.Andersson”);


// skapa ny medlem med initiering och anrop av copykonstruktor
MEDLEM m2 = m1;

// skapa ny medlem med initiering och anrop av copykonstruktor
MEDLEM m3(m2);

.....
```

Ovan anropas copy-konstruktorn explicit (synligt). Det finns fall där denna konstruktor anropas implicit (osynligt).

Copy-konstruktorn anropas i följande fall (exempel från klassen rationella tal) :

1) Vid initieringar.

```
RTAL a = b;
RTAL a(b);
```

2) Vid kopiering av en aktuell parameter till en *icke* referensdeklarerad formell parameter
.

Då funktionen

```
RTAL operator*(RTAL rt)
```

anropas med

```
b*c eller b.operator*(c)
```

görs initieringen

```
RTAL rt = c
```

3) Vid kopiering av returvärde till ett icke referensdeklarerat funktionsvärde.

Funktionen

```
RTAL RTAL::mul(RTAL rt)
{
   RTAL res;

   res.t = t*rt.t;
   res.n = n*rt.n;
   return res;
}
```

ger initieringen

```
RTAL mul = res
```

vid return.

4) Vid skapandet av temporära objekt vid beräkningar.

Beräkningen av

```
r = a + b*c
```

skapar ett temporärt objekt

```
RTAL temp
```

som initieras till värdet av b*c.

Ex : Skriv en klass STRANG, som avbildar strängar som innehåller copykonstruktor, tilldelning och utskrift.

```
// specifikation av klassen STRANG -- strang.h

class STRANG
{
   private:
      char *s;    // Pekare till strängstart
      int len;    // Strängobjektets längd

   public:
      STRANG(int len = 80);         // default-konstruktor
      STRANG(char *str);            // sträng-konstruktor
      STRANG(const STRANG &str);    // copy-konstruktor
      ~STRANG();
      STRANG &operator=(const STRANG &str);// tilldelning
      friend istream &operator>>(istream &in, STRANG &str);
      friend ostream &operator<<(ostream &out, const STRANG &str);
};
```

```
//implementation av klassen STRANG -- strang.cpp

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

STRANG::STRANG(int len)
{
   this->len = len;
   s = new char[this->len + 1];     // skapa utrymme
   s[0] = '\0';                     // tom sträng
}

STRANG::STRANG(char *str)
{
   len = strlen(str);
   s = new char[len + 1];
   strcpy(s, str);
}

STRANG::STRANG(const STRANG &str)
{
   len = str.len;
   s = new char[len + 1];
   strcpy(s, str.s);
}

STRANG::~STRANG()
{
   delete [] s;
}

STRANG &STRANG::operator=(const STRANG &str)
{
   // se upp med tilldelning till sig själv
   if ( this != &str)
   {
      delete [] s;
      len = str.len;
      s = new char[len + 1];
      strcpy(s, str.s);
   }
   return *this;
}

istream &operator>>(istream &in, STRANG &str)
{
   cout << "Ge en sträng : ";
   in.getline(str.s, str.len + 1);
   return in;
}

ostream &operator<<(ostream &out, const STRANG &str)
{
   out << str.s << endl;
   return out;
}
```

```
// huvudprogram som testar STRANG-klassen -- strmai.cpp

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

void main()
{
   STRANG s1, s2;                    // Rad 1
   STRANG s3 = "Hej på dig!";        // Rad 2
   STRANG s4 = s3;                   // Rad 3

   cin >> s1 >> s2;                  // Rad 4

   s3 = s1;                          // Rad 5
   s1 = s2;                          // Rad 6

   cout << s1 << s2 << s3 << s4;     // Rad 7
}
```

Vid körning kommer följande att hända rad för rad :

Rad 1: STRANG-objekten s1 och s2 skapas genom att defaultkonstruktorn anropas. Båda objekten har allokerat utrymme för strängar med maximalt 80 tecken och innehåller en tom sträng.

Rad 2: STRANG-objektet s3 skapas och får värdet "Hej på dig!" genom att strängkonstruktorn anropas. Detta är egentligen en typkonvertering (se nedan) från en vanlig sträng till ett STRANG-objekt.

Rad 3: STRANG-objekt s4 skapas och initieras med värden från strang-objekt s3 genom att copyonstruktorn anropas. Copykonstruktorn ser till att vi får nytt allokerat minne och en verklig kopia av strängen..

Rad 4: STRANG-objekten s1 och s2 läses in med överlagrad inläsningsoperator.

Rad 5 : Tilldelningsoperatorn anropas varvid en verklig kopia av s1 skapas i s3. Tilldelningen förstör först det gamla objektet s3 och ger den sedan samma värden som s1. STRANG-objektet s1 lämnas oförändrat.

OBS! Kontrollen för att inte förstöra objektet vid tilldelning till sig självt.

Rad 6 : Som rad 5 men tilldelning av s2 till s1.

Rad 7 : Utskrift av STRANG-objekten med överlagrad utskriftsoperator.

I den nya C++-standarden finns en klass för strängar som heter string. Principen för klassen string är lik den som i ovanstående klass, strang. Klassen string har dock betydligt fler funktioner. Använder man string för strängar slipper man ifrån den något omständiga hanteringen av de vanliga strängarna i C++, som i princip alltid måste hanteras som pekare. De gamla strängarna finns dock kvar och i äldre kompilatorer måste man använda dessa.

Ex : Skriv ett program som läser in två namn i form av objekt av klassen string , byter plats mellan objekten och skriver ut dem på skärmen.

```
#include <iostream>

#include <string>       // Den nya string-klassen

using namespace std;    // Öppna namnrymden std för direktåtkomst

void main()
{
   string anamn, bnamn, tempnamn;

   cout << "Första namnet ? ";
   cin >> anamn;

   cout << "Andra namnet ? ";
   cin >> bnamn;

   tempnamn = anamn;
   anamn = bnamn;
   bnamn = tempnamn;

   cout << "Första namnet är nu : " << anamn << endl;
   cout << "Andra namnet är nu : " << bnamn << endl;
}
```

Man slipper ifrån att tänka på strängar som pekare. Objekt av klassen string kan man som ovan tilldela till varandra och i övrigt också arbeta med dem som vilka objekt som helst.

Vill man så kan man göra om ett objekt av klassen string till en vanlig gammaldags sträng. Vill man göra om objektet anamn ovan till en vanlig sträng anropar man en medlemsfunktion enligt:

```
anamn.c_str();
```

Objekt av klassen string kan läsas in med cin som ovan. Det blir dock problem om objektet innehåller en separator (blank, tab eller RETURN). Precis som för vanliga strängar läser cin bara fram till separatorn. Följande funktion finns som läser hela stringobjektet:

```
getline(cin, anamn);
```

Det finns också ett antal operatorer definierade bl. +-operatorn för konkatenering eller sammanfogning av stringobjekt enligt:

```
tempnamn = anamn + bnamn;
```

som ger tempnamn värdet "Per.Olsson" om anamn är "Per." och bnamn "Olsson".

I kylsystemet och i all Windowsprogrammering har man behov av att göra om tal till strängar och tvärtom. Man kan använda sig av de gamla funktionerna sprintf och sscanf men även här finns nya klasser för att hantera sådana omvandlingar.

Ex : Skissa på ett program som omvandlar ett reellt tal till en sträng och tvärtom.

a) Första metoden

```
#include <stdio.h>

void main()
{
   float = 5.46, y;
   char xbuff[20], ybuff[20] = "y = 2.34";

   sprintf(xbuff, "x = %.2f", x);   // Strängen "x = 5.46" skapas
   printf("%s\n", xbuff);           // På skärmen skrivs x = 5.46

   sscanf(ybuff, "y = %f", &y);     // y är 2.34
   printf("%.2f\n", y);             // På skärmen skrivs 2.34
}
```

b) Andra metoden

```
#include <string>
#include <sstream>
#include <iostream>

using namespace std;

void main()
{
   float x = 5.46, y;
   ostringstream ssout;
   string ybuff = "y= 2.34", slask;
   istringstream ssin(ybuff);

   ssout << "x = " << x;            // Strängen "x = 5.46" skapas
   cout << ssout.str() <<  endl;    // På skärmen skrivs x = 5.46

   ssin >> slask >> y;              // y är 2.34
   cout << y << endl;               // På skärmen skrivs 2.34
}
```

Ex : I kylsystemet har man en grafikyta på formuläret, som man kan rita på med funktioner

som DrawLine, DrawString, DrawEllipse etc. DrawString-funktionen skriver text på grafikytan. För att kunna skriva ut trycket måste man först göra om trycket till en sträng. Detta gör man i en display-funktion som alltid visar aktuell information om exempelvis trycket i en nod. Display-funktionerna anropas med jämna tidsintervaller av kylsystemets händelsefunktion för timern. För exempelvis klassen NOD kan displayfunktionen, som ska visa trycket i nodobjekten, se ut som:

```
void NOD::display()
{
   char buff[20];
   courfont = new Font(CourierNew, 8);
   SolidBrush *sbf = new SolidBrush(Color::Black);
   sprintf(buff, "Tryck : %.1f", tryck);  // Omvandla tryck till
                                          // strängen buff
   canvas->DrawString(buff, sbf, font, x + 5f, y + 10f);
}
```

För att utskriften ej skall bli suddig måste man innan man skriver ut ett nytt värde på trycket sudda ut det gamla trycket genom att skriva ut det gamla trycket med bakgrundsfärg. För att objektet, ex. nod1, ska komma ihåg vilket tryck som skrevs ut senast, måste man definiera en medlemsvariabel, old_buff i klassen enligt:

```
char *old_buff;              // Medlemsvariabler i en
                             // garbagecollection klass(__gc klass)
                             // måste skapas dynamiskt på heapen

old_buff = new char[20];     // I konstruktorn

delete [] old_buff;          // I destruktorn (Behövs ej i en
                             // __gc class eftersom den har en
                             // automatisk skräpsamlare)

void NOD::display()
{
   // Om bakgrundsfärgen är ljusgrå.
   SolidBrush *sbb = new SolidBrush(Color::LightGray);
   // Här skrivs trycket med bakgrundsfärg
   canvas->DrawString(old_buff, sbb, font, x+5f, y+20f);
   // Här skrivs det nya trycket enligt ovan med svart färg
   // Spara undan det aktuella trycket i old_buff.
   strcpy(old_buff, buff);
}
```

Klassen Kylsystem har en händelsefunktion som ska hantera meddelanden från Windows i form av klockpulser. För att köra display-funktionerna exempelvis var 10:de ms skapar man en klocka med namnet timer_Tick, tidsintervallet 10 ms och en händelsefunktion enligt:

```
void Kylsystem::timer_Tick(Object *Sender, eventargs *e)
{
   n1.display();
   n2.display();
   n3.display();
   v1.display();
   v2.display();
}
```

4.4 Egendefinerad typkonvertering

Man kan definiera egna typkonverteringsregler för sina objekt. Dessa regler kommer kompilatorn att använda sig av på samma sätt som den använder konverteringsreglerna för de vanliga fördefinierade typerna i C++, dvs konvertera automatiskt vid behov eller då explicit konvertering begärs. Konvertering åstadkommer man med hjälp av *konstruktorer* och *operatorer* i sina klasser.

Man kan skilja på två olika typer av konvertering :

1) *Konvertering till egendefinierad klasstyp* sker med en konstruktor som tar ett objekt av den typ eller klass som ska konverteras som parameter.

2) *Konvertering till en vanlig fördefinierad typ* sker med en operator som har samma namn som typen.

Ex : Man vill ha en typkonvertering i klassen RTAL från heltal till RTAL så att exempelvis 5 konverteras till 5/1.

```
// specifikation av klassen RTAL -- rtal.h

class RTAL
{
   private :
      int t;
      int n;
   public :
      RTAL(int t = 1, int n = 1){ this->t =t; this->n = n;}

      .....

};



// huvudprogram

#include "rtal.h"

void main()
{
   RTAL a = 2;               // Rad 1
   RTAL b(3);                // Rad 2
   RTAL c(1, 4);             // Rad 3

   cout << a + c;            // Rad 4
   cout << c + 3;            // Rad 5
   cout << 2 + b;            // Rad 6, Kompileringsfel!
   cout << RTAL(2) + b;      // Rad 7

}
```

Rad 1: Typkonvertering av heltal till RTAL genom att konstruktorn anropas med första parametern 2 och den andra får defaultvärdet 1. Heltalet 2 konverteras till bråket a = 2/1.

Rad 2: Samma sak men heltalet 3 konverteras till bråket b = 3/1.

Rad 3: Samma konstruktor anropas men nu skapas bråket c = 1/4.

Rad 4: Alla objekt är RTAL så beräkningar sker med +operatorn i RTAL-klasssen och 9/4 skrivs ut.

Rad 5: Det första objektet är ett RTAL-objekt så +operatorn i RTAL anropas varvid den andra parametern 3 vid anropet omvandlas automatiskt till 3/1 genom att konstruktorn anropas implicit och resultatet 13/4 skrivs ut.

Rad 6: Kompileringsfel! Försöker anropa heltalsoperatorn operator+(2, b) men saknar konverteringregel för RTAL b till heltal så kompilatorn hittar ingen lämplig operator. Vill man att RTAL's +-operator ska anropas kan man göra som på rad 7 eller så måste man göra om +-operatorn i RTAL till en funktion med två RTAL-parametrar (friend-funktion) varvid automatisk konvertering sker av heltals-parametern.

Ex : Man vill ha en typkonvertering i klassen RTAL från RTAL till float så att exv 5/2 konverteras till 2.5.

```
// specifikation av klassen RTAL -- rtal.h

class RTAL
{
   private :
      int t;
      int n;
   public :
      RTAL(int t = 1, int n = 1){ this->t = t; this->n = n;}
      operator float(){ return float(t)/n;}

      .....

};
```

```
// huvudprogram

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

void main()
{
   RTAL a(1, 4);

   cout << 1.5 - a;
}
```

OBS! Vid beräkningen av 1.5 - a kommer kompilatorn att anropa operatorn,

```
float operator-(float op1, float op2);
```

eftersom det nu finns en konverteringsregel från RTAL till flyttal. Vid anropet konverteras a implicit till 0.25 och utskrift sker med utskriftsfunktionen för flyttal.

Överlagring av operatorer eller funktioner i sina egendefinierade klasser kan man antingen göra som medlemsfunktioner eller som friendfunktioner. När ska man göra det ena och när ska man göra det andra.

1) Vissa operatorer *måste* överlagras som *medlemsfunktioner* , nämligen operatorerna =, [], () och ->.

2) Kan man inte modifiera klassen för den *vänstra* operanden vid överlagring måste man använda *friendfunktioner*. Exempel på detta är överlagring av in- och utmatningsoperatorerna där man ej ska gå in i klasserna istream och ostream och ändra utan man istället överlagrar dessa operatorer som friends i den högra operandens klass.

3) Om den vänstra operanden *ej existerar som objekt* kan man med en *friendfunktion* få konvertering av *denna* operand till önskad typ. Exempel på detta är om man vill ha blandade beräkningar av heltal och bråk som 3 + 1/2. Vill man att 3 ska konverteras till ett rationellt tal och beräkning göras med RTAL-operatorn +, måste denna var definerad som friend.

4) Generellt ska man *alltid använda medlemsfunktion* om detta är möjligt.

Ex : Tillbaka till den tänkta implementationen av klassen ifstream. Där kan man istället för den överlagrade unära operatorn, !, ha en typkonvertering enligt :

```
// specifikation av klassen ifstream -- ifstream.h

class ifstream
{
   private:
      FILE *file;
      ....
   public:
      ifstream(char *filename);
      ~ifstream();
      operator int();
      ....
};


// implementation av klassen ifstream -- ifstream.cpp

#include "ifstream.h"

ifstream::ifstream(char *filename)
{
   file = fopen(filename, "r");
}

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

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


// huvudprogram

#include "ifstream.h"

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

   if ( f )
   {
      // filen finns och du kan fortsätta
      .....
   }
}
```

OBS! Här skulle man kunna anropa !f också men då skulle f först konverteras till ett heltal och därefter skulle anropet ske till den vanliga heltalsoperatorn!.