Egendefinierade betydelse av operatorer

Egendefinierade operatorer
- Operatorer som får överlagras
- Operatorer som inte får överlagras
- Unära operatorer
- Binära operatorer
- Prioritet
- Nya operatorer
- Operatorfunktion som klassmedlem
- Exempel: överlagring av binär operator som klassmedlem
- Överlagring av binär operator som icke-klassmedlem (ex << operatorn)
- Exempel: överlagring av unär operator som klassmedlem
- Övning 1
- Överlagring av tilldelningsoperatorn =
- Överlagring av typomvandlingsoperator
- Exempel: överlagring av typomvandling till const char*
- Överlagring av indexoperatorn []
- Exempel: överlagring av indexoperatorn []
- Övning 2
- Överlagring av -> operatorn
- Överlagring av () (funktionsanrop)
- Functors, objekt som funktioner
- Användning
- Resultat
- Klassdefinition
- Definition av klassmedlemmar

Egendefinierade operatorer

I C++ kan vi omdefiniera de vanliga operatorerna och ge dessa en ny användardefinierad betydelse. Detta sker via överlagring. Följande regler gäller för hur detta får göras:

Operatorer som får överlagras.1 Operatorer som får överlagras.1 Operatorer som får överlagras.1 Operatorer som får överlagras

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

Både den unära och binära formen av +, -,* och & kan överlagras.

Operatorer som inte får överlagras4.2 Operatorer som inte får överlagras4.2 Operatorer som inte får överlagras.2 Operatorer som inte får överlagras

Nedanstående operatorer får inte överlagras:

```
.       .*      ::      ?:      #       ##
```

Unära operatorer.3 Unära operatorer.3 Unära operatorer.3 Unära operatorer

En unär operator X kan överlagras antingen som en medlemsfunktion utan argument eller en vanlig funktion som tar ett argument.

Binära operatorer.4 Binära operatorer.4 Binära operatorer.4 Binära operatorer

En binär operator kan överlagras antingen som en medlemsfunktion med ett argument eller en vanlig funktion som tar två argument.

Prioritet.6 Prioritet.6 Prioritet.6 Prioritet

Operatorerna behåller den precedensordning som de har enligt standarden. Denna kan **inte** förändras. Inte heller kan man få prefix-operator att verka som postfix eller vice-versa.

Nya operatorer.7 Nya operatorer.7 Nya operatorer.7 Nya operatorer

Man kan **inte** införa egna operatorkombinationer som t ex
:=, <> eller |x| (försök att konstruera en operator för absolutbelopp).

Operatorfunktion som klassmedlem4.8 Operatorfunktion som klassmedlem4.8 Operatorfunktion som klassmedlem.8 Operatorfunktion som klassmedlem

Vi kan införa en operatorfunktion som medlemsfunktion i en klass. För en sådan funktion gäller att:

- binära operatorer överlagras med en funktion som har ett argument (t ex == och +)

- unära operatorer överlagras med en funktion som saknar argument (t ex ++ och --)

Exempel: överlagring av binär operator.9 Exempel: överlagring av +.9 Exempel: överlagring av +.9 Exempel: överlagring av + som klassmedlem

Antag att vi konstruerat klassen **Number** och vill införa en betydelse för aritmetiska operatorer för denna klass. Dvs vi vill kunna använda följande syntax:

```
Number  a =  36;
Number  b =  12;

Number  summa =  a + b;
Number  differens = a - b;
Number  produkt = a * b;
Number  kvot =  a / b;

summa.printon(cout);
endl(cout);

differens.printon(cout);
endl(cout);

produkt.printon(cout);
endl(cout);

kvot.printon(cout);
endl(cout);
```

Om vi vill kunna använda ovanstående syntax måste vi först överlagra **+, -, *** och /.

Vi deklarerar först klassen och sedan definierar vi medlemsfunktionerna. Lägg märke till att vi deklarerar funktionerna som:

```
Number operator+(const Number& roper) const;
```

dvs funktionen tar emot en **referens till en const Number** och kan anropas för objekt som är konstanta. Eftersom funktionerna inte manipulerar med operanderna är det lika bra att göra på detta sätt från början:

```
// number1.h (number1.hh)
#ifndef NUMBER1_H
#define NUMBER1_H
class ostream;
class Number {
public:
   Number(double init);
   Number operator+(const Number& roper) const;
   Number operator-(const Number& roper) const;
   Number operator*(const Number& roper) const;
   Number operator/(const Number& roper) const;
   void printon(ostream& os);
private:
   double value;
};
#endif

// number1.cpp ( number1.cc )
#include "number1.h"
Number Number::operator+(const Number& roper) const
{
   return value + roper.value;
}
Number Number::operator-(const Number& roper) const
{
   return value - roper.value;
}
Number Number::operator*(const Number& roper) const
{
   return value * roper.value;
}
Number Number::operator/(const Number& roper) const
{
   assert(roper.value != 0.0);
   return value / roper.value;
}
```

När vi skriver:

```
    Number  summa =  a + b;
```

anropar vi Number::operator +(const Number&);

dvs vi skulle lika gärna kunna skriva

```
    Number  summa =  a.operator+(b);
```

Vi kan själva välja vilket av ovanstående två skrivsätt vi vill använda i vårt program. De flesta anser nog att den **första varianten** ger mer lättläst kod och bättre motiverar varför vi överhuvudtaget bör använda operatoröverlagring.

Överlagring av binär operator som icke-klassmedlem (ex << operatorn)

Antag att vi vill kunna göra på nedanstående sätt. dvs att vi vill kunna skriva ut objekt av typen **Number** på samma sätt som andra typer:

```
main()
{
    Number  a =  36;
    Number  b =  12;

    Number  summa =  a + b;
    Number  differens = a - b;
    Number  produkt = a * b;
    Number  kvot =  a / b;

    cout << "Summa     :" << summa     << endl;
    cout << "Differens :" << differens << endl;
    cout << "Produkt   :" << produkt   << endl;
    cout << "Kvot      :" << kvot      << endl;

    return 0;
}
```

Ett sätt att göra detta möjligt på är att överlagra <<-operatorn för objekt av typen **Number**. Eftersom vi alltid använder ett **stream**-objekt som vänster operand kan vi **inte** överlagra operatorn som medlem i klassen **Number**. Vi vill inte heller överlagra <<-operatorn som medlem i klassen **ostream** eftersom vi självklart inte vill modifiera **iostream.h**. Kvar finns då alternativet att definiera <<-operatorn som **icke-klassmedlem**. En binär operator överlagras som en funktion som tar två argument:

```
ostream& operator <<( ostream& os,const Number& num)
{
    os << num.value;
    return os;
}
```

Eftersom funktionen använder en inkapslad medlem i klassen **Number** måste vi komplettera klassen **Number** med en **friend**-deklaration av funktionen.

```
// number2.h
#ifndef NUMBER2_H
#define NUMBER2_H
class ostream;
class Number {
    friend ostream& operator <<(ostream& os, const Number& num);
public:
    Number(double init);
    Number operator+(const Number& roper) const;
    Number operator-(const Number& roper) const;
    Number operator*(const Number& roper) const;
    Number operator/(const Number& roper) const;
    void printon(ostream& os);
private:
    double value;
};
#endif
```

Exempel: överlagring av unär operator.9 Exempel: överlagring av +.9 Exempel: överlagring av +.9 Exempel: överlagring av + som klassmedlem

Antag att vi konstruerat klassen **Number** och vill införa en betydelse för de unära operatorerna ++ och -- operatorer för denna klass. Dvs vi vill kunna använda följande syntax:

```
Number a = 24;

cout << a << endl;

Number b = a++;

cout << b << endl;

cout << a++ << endl;
cout << ++a << endl;

Number c = 0;

c = ++a + ++b + ++c;

cout << c << endl;
```

och få nedanstående resultat:

```
24
24
25
27
54
```

En unär operator som överlagras som klassmedlem blir en funktion som saknar parametrar. För att kunna skilja på post och prefix-varianterna av ++ och -- har man i standarden infört en extra parameter i postfixvarianten:

```
// number4.h
#ifndef NUMBER4_H
#define NUMBER4_H
class ostream;
class Number {
   friend ostream& operator <<( ostream& os, const
Number& num);
public:
   Number(double init);
   Number operator+(const Number& roper) const;
   Number operator-(const Number& roper) const;
   Number operator*(const Number& roper) const;
   Number operator/(const Number& roper) const;
   Number operator++();
   Number operator++(int);
   Number operator--();
   Number operator--(int);
   void printon(ostream& os);
private:
   double value;
};
#endif
```

Nedan visar vi hur man definierar funktionerna:

```
// number4.cpp
..
Number Number::operator++()
{
   value+=1.0;
   return *this;
}

Number Number::operator++(int)
{
   Number tmp = *this;
   value+=1.0;
   return tmp;
}
Number Number::operator--()
{
   value+=1.0;
   return *this;
}
Number Number::operator--(int)
{
   Number tmp = *this;
   value+=1.0;
   return tmp;
}
...
```

Övning 1

Nedanstående exempel fungerar inte om vi använder klassen **Number** som den har definerats i föregående exempel. Varför då? Förändra eller komplettera **Number** så att nedanstående exempel fungerar.

```
main()
{
   Number a = 24;
   Number b = 12;
   Number c = 0;
   Number d = 0;

   c = b + 10 + a;

   d = 10 + b + c;   // Fungerar inte!

   cout << "c = " << c << endl;
   cout << "d = " << d << endl;

   if (c == d)    // Fungerar inte
   {
       cout << "c == d\n";
   }
   if (a != b)    // Fungerar inte
   {
       cout << "a != b\n";
   }
   if (a > c)     // Fungerar inte
   {
       cout << "a > c\n";
   }
   if (a < d)     // Fungerar inte
   {
       cout << "a < d\n";
   }

   return 0;
}
```

Överlagring av tilldelningsoperatorn =

Tilldelningsoperatorn (=) kan endast överlagras som medlemsfunktion i en klass. Funktionen är den enda som **inte** kan ärvas av en subklass.

Om en klass saknar användardefinierad operatorfunktion för tilldelningsoperatorn så genererar kompilatorn **alltid** en ändå. Denna implementeras då implicit och motsvarar medlemsvis kopiering.

Exempel på default operatorfunktion för tilldelningsoperatorn:

```
X& X::operator =(const X& b)
{
   member1 = b.member1;
   member2 = b.member2;
   ...
   return *this;
}
```

Om klassen har medlemmar som refererar till eller pekar på minne (t.ex. dynamiskt minne) som finns utanför objekten passar det inte alltid att vi använder oss av ovanstående eftersom kopiering av en pekare inte innebär kopiering av det som vi pekar på. Vi kan då själva definiera en tilldelningsoperator för vår klass.

Antag t.ex. att vi infört nedanstående strängklass. Om vi vill kunna hantera tilldelning av **Streng**-objekt utan att det händer olyckor måste vi införa en egen =-operator för klassen.

```
// streng1.h
#ifndef STRENG1_H
#define STRENG1_H
class Streng {
   friend ostream& operator<<(ostream& os,const Streng&
s);
public:
   Streng();
   Streng(char *p);
   Streng(const Streng&);
   ~Streng();
   Streng& operator=(const Streng&);
   int lengd();
   static int antal();
private:
   char *buffer;
   static int ant;
};
#endif
```

Nedan visar vi hur klassens medlemmar definieras:

```
// streng1.cpp
#include <iostream.h>
#include <string.h>
#include <assert.h>
#include "streng1.h"
int Streng::ant;
Streng::Streng()
:buffer(new char[1])
{
   buffer[0] = '\0';
}
Streng::Streng(char *str)
: buffer(strcpy(new char[strlen(str)+1],str))
{
   ant++;
}
Streng::Streng(const Streng& strref)
: buffer(strcpy(new char[strlen(strref.buffer)+
1],strref.buffer))
{
   ant++;
}

Streng::~Streng()
{
   delete [] buffer;
   ant--;
}

Streng& Streng::operator=(const Streng& strref)
{
   if (this != &strref)
   {
       delete [] buffer;
       buffer = strcpy(
           new char[strlen(strref.buffer)+1],
           strref.buffer);
   }
   return *this;
}
```

```
int Streng::lengd()
{
   return strlen(buffer);
}
int Streng::antal()
{
   return ant;
}
ostream& operator<<(ostream& os,const Streng& s)
{
   return os << s.buffer;
}
```

Överlagring av typomvandling.4 Typomvandling.4 Typomvandling.4 Typomvandlingsoperator

Vi vill ibland integrera våra typer med standardtyperna och t.ex. kunna använda de biblioteksfunktioner som finns för dessa. Antag t.ex. att vi vill kunna använda objekt av vår typ **Number** som argument till funktioner som tar en **double** som argument.

```
#include <math.h>
#include "number5.h"

double kvadrat(double a)
{
   return a * a;
}

double max(double a, double b)
{
   return a > b ? a : b ;
}
main()
{
   Number a = 3;
   Number b = 5;
   Number c = 0;

   c = pow(3,5);

   cout << c << endl;
   cout << max(a,b) << endl;
   cout << kvadrat(a) << endl;

   return 0;
}
```

För att ovanstående anrop ska kunna fungera måste vi hjälpa kompilatorn genom att definera hur omvandling från **Number** till **double** skall göras.

Typomvandling från en typ X till en typ Y kan införas genom att implementera en konstruktor för Y med argument av typ X.

```
Y::Y( const X& x);
```

Men denna metod kan inte användas för konvertering till någon standardtyp (t ex int eller float) eftersom dessa inte är klasser och därför inte kan någon konstruktor.

För att lösa detta problem kan vi C++ implementera en s.k. **typomvandlingsoperator** som en

operatorfunktion.

```
// number4.h
#ifndef NUMBER4_H
#define NUMBER4_H
class ostream;
class Number {
   friend ostream& operator <<( ostream& os, const
Number& num);
public:
   Number(double init);
   Number operator+(const Number& roper) const;
   Number operator-(const Number& roper) const;
   Number operator*(const Number& roper) const;
   Number operator/(const Number& roper) const;
   Number operator++();
   Number operator++(int);
   Number operator--();
   Number operator--(int);
   operator double();
   void printon(ostream& os);
private:
   double value;
};
#endif
```

Lägg märke till att funktionen "**operator double ();"** inte får ha någon returtyp (t.ex "**Number operator double ();"**) eftersom funktionen inte får returnera någon annan typ än den som som angetts.

Nedan visar vi hur funktionen definieras:

```
Number::operator double()
{
   return value;
}
```

När uttrycket

```
 c = pow(a,b);
```

exekveras så sker följande:

1 typomvandling från typen Number till double

2 anrop av funktionen pow.

3 omvandling av funktionens returvärde till ett temporärt **Number**-objekt..

4 anrop av klassen **Number**'s =-operator som utför medlemsvis kopiering av det temporära objektet till c.

Det kan beskrivas som nedan:

```
c = Number(pow(a.operator double(),b.operator double()));
```

Exempel: överlagring av typomvandling till const char*

Antag att vi vill kunna anropa funktioner som tar en **const char *** som parameter med ett objekt av typen **Streng**.

```
// strmain2.cpp
#include <iostream.h>
#include <string.h>
#include "streng2.h"
main()
{
   Streng s = "Hej alla programmerare!" ;

   Streng one;
   Streng two;
   Streng three;

   cout << s << endl;

   char *buffer = new char[strlen(s)+1];
   strcpy(buffer,s);

   one = strtok(buffer," ");
   two = strtok(NULL  ," ");
   three = strtok(NULL  ," ");

   delete [] buffer;
```

```
    cout << one << endl;
    cout << two << endl;
    cout << three << endl;

    return 0;
}
```

Genom att vi överlagrar typomvandling till **const char*** för vår klass **Streng** fungerar ovanstående:

```
#ifndef STRENG2_H
#define STRENG2_H
class Streng {
    friend ostream& operator<<(ostream& os,const Streng& s);
public:
    Streng();
    Streng(char *p);
    Streng(const Streng&);
    ~Streng();
    Streng& operator=(const Streng&);
    operator const char*() const;
    int lengd();
    static int antal();
private:
    char *buffer;
    static int ant;
};
#endif
```

Nedan visar vi hur funktionen definieras:

```
...
Streng:: operator const char *() const
{
    return buffer;
}
..
```

Överlagring av indexoperatorn.1 Överlagring av indexoperatorn.1 Överlagring av indexoperatorn.1 Överlagring av indexoperatorn []

Indexoperatorn är en binär operator. Den kan endast överlagras som medlemsfunktion. Om vi har deklarerat en klass X och överlagrat []-operatorn så kommer uttrycket:

```
X  x;

x[y];
```

att motsvara anropet

```
x.operator [](y);
```

C och C++-kompilatorer kontrollerar normalt inte att programmeraren indexerar innanför de gränser som gäller för ett fält. Detta upplevs ofta, speciellt av nybörjare, som en brist gentemot andra språk vars kompilatorer utför denna kontroll.

Genom att överlagra []-operatorn kan vi i C++ avhjälpa detta genom att själva kontrollera att indexeringen sker inom ett tillåtet intervall.

Exempel: överlagring av indexoperatorn []

Antag att vi vill kunna indexera på ett objekt av typen **Streng** på nedanstående sätt.

```
// strmain3.cpp
#include <iostream.h>
#include "streng3.h"

void print(const Streng& s)
{
   int lengd = s.lengd();
   for(int i=0; i < lengd ;i++)
       cout << s[i];
   cout << endl;
}

void fill(Streng& s,char fillchar)
{
   int lengd = s.lengd();
   for(int i=0;i < lengd;i++)
       s[i] = fillchar;
}

main()
{
   Streng s1 = "Hej alla programmerare!" ;

   print(s1);
   fill(s1,'A');
   print(s1);

   return 0;
}
```

För att även kunna indexera en konstant **Streng** inför vi två varianter av []-operatorn.

```
// streng3.h
#ifndef STRENG3_H
#define STRENG3_H
class Streng {
    friend ostream& operator<<(ostream& os,const Streng& s);
public:
   Streng();
   Streng(char *p);
   Streng(const Streng&);
   ~Streng();
   Streng& operator=(const Streng&);
   operator const char*() const;
   char& operator [](int index);
   char  operator [](int index) const;
   int lengd() const;
   static int antal();
private:
   char *buffer;
   static int ant;
};
#endif
```

Lägg märke till att operatorfunktionen returnerar en referens till en char. Det är pga detta som även tilldelningen

```
s[i] = fillchar;
```

är möjlig och innebär ett anrop av operatorfunktionen.

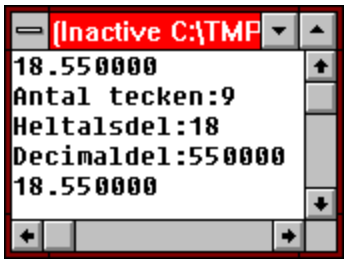
Nedan visar vi hur funktionerna definieras:

```
....
char& Streng::operator [](int index)
{
   assert( index >= 0 && index < strlen(buffer));
        return buffer[index];
}
char Streng::operator [](int index) const
{
   assert( index >= 0 && index < strlen(buffer));
        return buffer[index];
}
....
```

Övning 2

Inför omvandling av **Number** till **const char*** så att nedanstående exempel fungerar:

```
main()
{
   Number a = 3.5;
   Number b = 5.3;
   Number c = 0;

   c = a * b;

   Streng s;

   s = c;

   cout << s << endl;
   cout << "Antal tecken:" << s.lengd() << endl;

   cout << "Heltalsdel:";
   for(int i=0;s[i]!='.' && i< s.lengd();i++)
       cout << s[i];

   cout << endl;
   cout << "Decimaldel:";
   for(i++;i< s.lengd();i++)
       cout << s[i];
   cout << endl;


   char *buffer = new char[strlen(c)+1];
   strcpy(buffer,c);

   cout << buffer << endl;

   delete [] buffer;

   return 0;
}
```


(Inactive C:\TMP
18.550000
Antal tecken:9
Heltalsdel:18
Decimaldel:550000
18.550000

Överlagring av -> operatorn.2 Överlagring av -> operatorn.2 Överlagring av -> operatorn.2 Överlagring av -> operatorn

Operatorn för medlemsaccess är en unär operator. Den kan endast överlagras som medlemsfunktion. Funktionen **måste** returnera en pekare till ett objekt. Om vi har deklarerat en klass X och överlagrat ->-operatorn så kommer uttrycket:

```
X x;
x->y;
```

att motsvara

```
(x.operator ->())-> y;
```

Genom att överlagra -> operatorn kan vi t ex införa **smarta** pekare som hindrar oss att använda pekare som inte är initierade att peka någonstans.

Överlagring av () (funktionsanrop.3 Överlagring av () (funktionsanrop.3 Överlagring av () (funktionsanrop.3 Överlagring av () (funktionsanrop)

Operatorn för funktionsanrop är en binär operator. Den kan endast överlagras som medlemsfunktion. Om vi har deklarerat en klass X och överlagrat ()-operatorn så kommer uttrycket:

```
X x;
x(y);
```

att motsvara

```
x.operator ()(y);
```

Vänster operand är här objektet x. Höger operand är en parameterlista som kan bestå av ett godtyckligt antal parametrar.

Functors, objekt som funktioner

Genom att man överlagrar funktionsanropsoperatorn kan objekt anropas som om de vore funktioner. Här inför vi ett globalt skatteobjekt som man kan anropa när man vill beräkna skatt:

Användning

Lägg märke till hur vi anropar objektet som om det vore en funktion:

```
Skatt skatt;

main()
{
     Number inkomst = 223000.0;

     cout << "Inkomst      :" << inkomst << endl;
     cout << "Kommunalskatt:" << skatt(Skatt::KOMMUN,inkomst) << endl;
     cout << "Statlig skatt:" << skatt(Skatt::STAT,inkomst) << endl;
     cout << "Total skatt  :" << skatt(Skatt::TOTAL,inkomst) << endl;
     cout << "Nettoinkomst :" << inkomst - skatt(Skatt::TOTAL,inkomst) << endl;

     return 0;
}
```

Resultat

Klassdefinition

Lägg märke till hur vi överlagrar ()-operatorn i klassen **Skatt**.

```
// skatt.h
#ifndef SKATT_H
#define SKATT_H
class Number;
class Skatt {
public:
    enum Taxtype { KOMMUN, STAT, TOTAL };
    Skatt();
    Number operator ()( Taxtype typ, const Number& inkomst);
};
#endif
```

Definition av klassmedlemmar

```
// skatt.cpp
Skatt::Skatt()
{
}

Number Skatt::operator ()( Taxtype typ, const Number& inkomst)
{
     const Number brytpunkt     = 196000.0;
     const Number statligskatt  = 0.20;
     const Number kommunalskatt = 0.30;

     Number result = 0;

     switch (typ)
     {
     case KOMMUN:    result = kommunalskatt * inkomst;
                     break;
     case STAT  :    if (inkomst< brytpunkt)
                          result = 0;
                     else
                          result = (inkomst-brytpunkt)* statligskatt;
                     break;
     case TOTAL :    if (inkomst< brytpunkt)
                          result =  kommunalskatt * inkomst;
                     else
                          result =  kommunalskatt * inkomst +
                          (inkomst-brytpunkt) * statligskatt;
                     break;
     default:
                     result = -1;
     }
     return result;
}
```