4) Operatoröverlagring

(Ex) En klass för komplexa tal

```
// Specifikation -- Komplex.h

class KOMPLEX
{
   private:
      float re, im;   // re + im*j
   public:
      KOMPLEX(float re = 0.0, float im = 0.0)
      void las();
      void skriv();
      KOMPLEX add(KOMPLEX k);
         "    sub    "
         "    mul    "
         "    div    "
};
```

```
// Implementation -- Komplex.cpp

KOMPLEX::KOMPLEX(float re, float im)
{
   this -> re = re;
   this -> im = im;
}
```

```
void KOMPLEX::las()
{
    cout << "Ge komplext tal på formen a+bj: ";
    cin >> re >> im;
    cin.ignore();  // Renca j
}

void KOMPLEX::skriv()
{
    cout << re << '+' << im << 'j' << endl;
}

KOMPLEX KOMPLEX::add(KOMPLEX k)
{
    KOMPLEX temp;

    temp.re = re + k.re;
    temp.im = im + k.im;
    return temp;
}
!
!

// Huvudprogram -- kompmain.cpp

void main()
{
    KOMPLEX a, b, res;
```

```
    a.las();
    b.las();
    res = a.mul(b).div(a.sum(b));// res = a*b/(a+b)
    res.skriv();
    }
}
```

Ej bra! Vill istället använda matematiska operatorer och in-, utmatningsoperatorer enligt:

```
void main()
{
    KOMPLEX a,b,res;

    cin >> a >> b;
    res = a*b/(a+b);
    cout << res;
    }
}
```

Hur få +, -, * och / ?

```
// Komplex.h

class KOMPLEX
{
    private:
      float re,im;
```

```
public:
    KOMPLEX(float re=0.0, float im=0.0);
    :
    !
    KOMPLEX operator+(KOMPLEX k);
        "   operator -    "
        "   operator *    "
        "   operator /    "
};

// Komplex.cpp

KOMPLEX KOMPLEX::operator+(KOMPLEX k)
{
    KOMPLEX temp;

    temp.re = re + k.re;
    temp.im = im + k.im;
    return temp;
}
:
!
!

// Kompmain.cpp

void main()
{
  :
  res = a*b / (a+b); // Samma prioritet som i matte!
  :
```

Hur få << och >> ?

$a + b \Leftrightarrow a.operator+(b)$ } Jämför!

$cin >> a \Leftrightarrow cin.operator>>(a)$ }

↑

Anropsobjektet tillhör klassen istream och därför borde operatorn tillhöra denna klass. Man bör dock ej ändra i de fördefinierade klasserna utan man gör en friend-funktion i klassen KOMPLEX istället!

```
class KOMPLEX
{
    private:
        float re, im;
    public:
      KOMPLEX (float re=0.0, float im=0.0
      friend ostream &operator<<(ostream &ut, KOMPLEX k);
      friend istream &operator>>(istream &in, KOMPLEX &k);
      !
      !
};
```

friend innebär att funktionerna operator<< och >> får direkt tillgång till de privata delarna (re och im) i klassen KOMPLEX.

```
//Komplex.cpp

osteam &operator<<(osteam &ut, KOMPLEX k)
{
    ut << k.re;
    ut.setf(ios::showpos); //Visar +
    ut << k.im << 'j' << endl;
    ut.unsetf(ios::showpos);
    return ut;
}

OBS! Referens för att få tillståndet hon in
OBS! Referens för att få nytt

isteam &operator>>(isteam &in, KOMPLEX &k)
{
    cout << "Ge komplext tal på formen a+bj: ";
    in >> k.re >> k.im;
    in.ignore(); //Läs förbi j
    return in;
}
;
//Kompmain.cpp

void main()
{
    KOMPLEX a,b;

    cin >> a >> b;
    cin >> b
    cin
```

Tilldelning och initiering

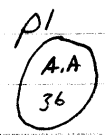
Båda finns fördefinierade för alla objekt!

(Ex) Initiering: `float x = 5.6; // Skapar x och ger den värde`

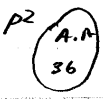
Tilldelning: `x = 8.4; // Ger x ett värde`

(Ex) Personklassen

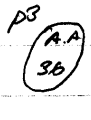
```
// Person.h

class PERSON
{
    private:
        char namn[30];
        int alder;
    public:
        PERSON (char *namn = "", int aldu = 0);
};

PERSON p1("A.A", 36);
```



Initiering: `PERSON p2 = p1;`



Tilldelning: `PERSON p3;`
`p3 = p1;`

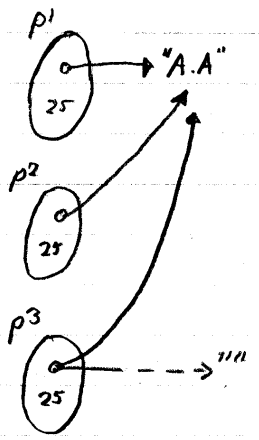


De fördefinierade initiering och tilldelning kopierar över data rakt av vilket kan ställa till problem

(Ex) Personklassen med dynamiskt allokerat namn

```
//Person.h

class PERSON
{
    private:
        char *namn; //OBS! Bara en pekare
        int alder;
    public:
        PERSON(char *namn="", int alder=0);
        ~PERSON();
};

//Person.cpp

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

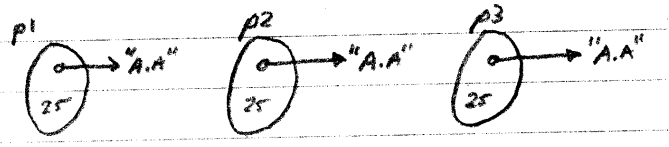
```
PERSON::~PERSON()
{
    delete [] namn;
    namn = NULL;
}
|
|
//Persmain.cpp

void main()
{
    PERSON p1("A.A", 25);

    //Initiering
    PERSON p2 = p1;

    //Tilldelning
    PERSON p3;
    p3 = p1;
    |
}
```



Problem! ① Flera pekare till samma dynamiskt allokerade minnesutrymme! Vid avallokering med en pekare blir de resterande kvar och pekar då på avallokerat utrymme.

② En pekare kan ställas om och kvar blir allokerat minne som aldrig frigörs - äkta minne

Lösa problem! Se till att skapa egen initiering och tilldelning som gör äkta kopior av objekten enligt:



(Ex) Skriv egen initiering och tilldelning till PERSON-klassen.

```
// Persm.h

class PERSON
{
   private:
      char *namn;
      int alder;
   public:
      :
      :
      // Initiering eller kopieringskonstruktor
      PERSON(const PERSON &p);
      :
      :
      // Tilldelningsoperator
      const PERSON &operator=(const PERSON &p);
```

```
// Person.cpp
:
:
PERSON::PERSON(const PERSON &p)
{
    namn = new char[strlen(p.namn) + 1];
    strcpy(namn, p.namn);
    alder = p.alder;
}
```

Går lika bra med PERSON p!

Går lika bra med PERSON!

```
const PERSON &PERSON::operator=(const PERSON &p)
{
  if (this != &p) // Se upp med tilldelning till sig själv
  {
      delete [] namn;
      namn = new char[strlen(p.namn) + 1];
      strcpy(namn, p.namn);
      alder = p.alder;
  }
  return *this;
}
```

① Varför refernsparameter i kopieringskonstruktor?
Jo, annars anropar kopieringskonstruktorn sig själv!

② Varför const referensparameter?
Jo, så att man ej ändrar refparametern av misstag!

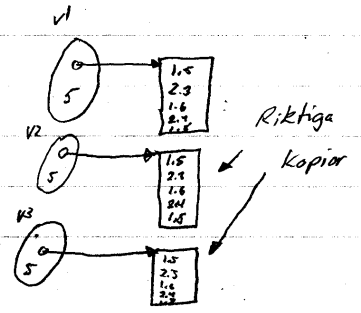
(Ex) En VEKTOR-klass av reella tal

```
//Vektor.h

class VEKTOR
{
    private:
        float *fv;    // Pekare till vektorns första element
        int nr;
    public:
        VEKTOR(int nr = 10);
        ~VEKTOR();
        VEKTOR(const VEKTOR &v);
        const VEKTOR &operator=(const VEKTOR &v);
        |
};

//Vektor.cpp
|
VEKTOR::VEKTOR(int nr)
{
        fv = new float[nr];
    this->nr = nr;
}
VEKTOR::~VEKTOR()
{
        delete [] fv; fv=NULL;
}
```

```
VEKTOR :: VEKTOR (const VEKTOR &v)
{
    fv = new float [v.nr];
    for(int i=0; i < v.nr ; i++)
    {
        fv[i] = v. fv[i];
    }
    nr = v.nr;
}

const VEKTOR &VEKTOR :: operator (const VEKTOR &v)
{
    if (this != &v)  // se upp med tilldelning till sig själv
    {
        delete [] fv;
        fv = new float[v.nr];
        for (int i=0; i<v.nr; i++)
        {
            fv[i] = v.fv[i]
        }
        nr = v.nr;
    }
    return *this;
}
```

Anrop!

```
VEKTOR v1(5);
//Läs v1
VEKTOR v2=v1;
VEKTOR v3;
v3=v1;
```



Typomvandlingar

(Ex)
```
//Komplex.h
class KOMPLEX
{
    public:
        KOMPLEX (float re=0.0, float im=0.0);   (1)
        operator float () { return re; }        (2)
};
```

(1) Från reellt tal till komplext KOMPLEX a(2.5) ⇝ 2.5+0.0j

(2) Från komplext till reellt float(a) ⇝ 2.5

(Ex) 3 typer av strängar C-sträng, string och String

Typomvandling från C-sträng till övriga:
```
char *cnamn = new char[20];
strcpy(cnamn, "Kalle");
string *snamn = new string(cnam);
String *Snamn = new String(cnam);
```

Typomvandling från övriga till C-sträng:
```
strcpy(cnamn, snamn->c_str());
```
vid omvandling från string till c-sträng måste man kopiera fecken för fecken!

Hemuppgift: Lägg till in- och utmatningsoperatorer till klassen PERSON och implementera dessa samt skriv ett huvudprogram som använder dessa operatorer.