Mittuniversitetet
MID SWEDEN UNIVERSITY

# Lektion 1 – OOP, UML samt repetition

*DT060G Objektorienterad programmering i C++*

Innehåll:

- OOP-terminologi samt introduktion av UML.
- Repetition av
  - allokering / deallokering av objekt
  - defaultversioner av medlemsfunktioner
  - operatoröverlagring

# Lektion 1 – OOP, UML och repetition

Vi börjar med repetition av några grundläggande begrepp.
*Objekt*:
Ett objekt är en unik instans av en bestämd datatyp (klass). Alla objekt av samma datatyp har samma uppsättning data (attribut), däremot har varje enskilt objekt sina egna *värden* på attributen. De operationer som kan utföras på ett objekt är de operationer som definierats för den datatyp (klass) som objektet tillhör.
I C++ kallas attributen datamedlemmar och operationerna utförs genom medlemsfunktioner.

*Klass*:
En klass definierar en datatyp med tillhörande attribut och operationer. De publika medlemmarna utgör tillsammans klassens publika gränssnitt (public interface), dvs de medlemmar som kan accessas av klienter till klassen. Instanser av en klass kallas objekt.

I ett exekverande program kommunicerar olika objekt genom att skicka *meddelanden* (messages) till varandra. Objekt A skickar ett meddelande till objekt B genom att anropa en medlemsfunktion för objekt B. Om meddelanden kan skickas mellan två objekt så existerar en relation mellan dessa objekt. En relation kan vara enkel- eller dubbelriktad.

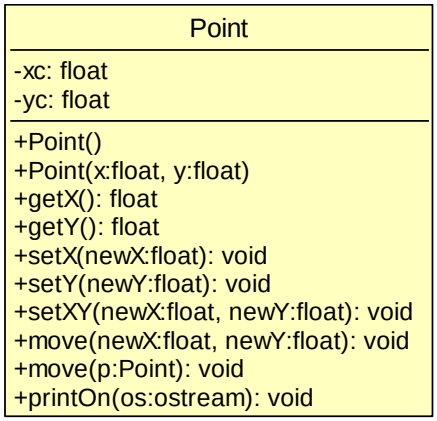
UML (Unified Modeling Language) är ett modelleringsspråk för att grafiskt beskriva objektorienterade system genom ett antal olika diagramtyper.

Den statiska strukturen hos en klass kan beskrivas i ett *klassdiagram* som utöver klassens namn bl.a. kan visa
- datamedlemmar (attribut)
- klassmedlemmar (static) visas understrukna
- medlemsfunktioner
- 'visibility'/access

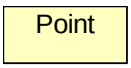
Exempel – klassen Point

```
class Point {
public: // Public interface

    // Constructors
    Point();    // default constructor
    Point(float,float); // overloaded constructor

    // Getters & setters
    void setX(float newX) { xc = newX; } // set x-coord
    void setY(float newY) { yc = newY; }    // set y-coord
    Point setXY(float,float); // set x- and y-coord, returns old Point
    Point setXY(const Point &); // set x and from argument
                                // returns old Point
    Point move(const Point &p); // move this point as indicated by p
                                // returns old Point
    float getX() const { return xc; } // get x-coord
    float getY() const { return yc; } // get y-coord

    // Facilitators
    void printOn(ostream &os=cout) const;   // print coordinates on os

private:    // Private members
    float xc,yc;
};
```

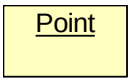
Motsvarande UML klassdiagram:




Visibility/Access markeras med
- \- för *private*
- \+ för *public*
- \# för *protected*

Samma klassdiagram utan attribut och operationer kan skrivas 


En *instans* av klassen Point, dvs ett *objekt som existerar under exekveringen* modelleras med




**Relationer**

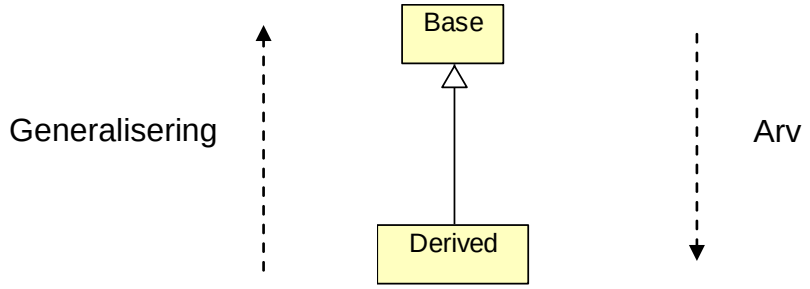
Det finns fem sorters relationer mellan klasser:
- En *generalisering* beskriver att en klass är en subklass till en annan.
- En *realisering* beskriver att en klassimplementerar ett gränssnitt (interface).
- En *association* beskriver att en klass har ett attribut av en annan klass eller omvänt.
- En *inkapsling* beskriver en situation där en klass är deklarerad inuti en annan för att göra den osynlig för andra klasser.
- Ett *beroende* beskriver en relation mellan två klasser av annat slag än de ovanstående som medför att den beroende klassen måste modifieras om den andra klassen förändras.

I ett klassdiagram visas relationer som linjer som förbinder klasser. Olika sorters associationer använder dekorationer som pilar, stereotyper och streckning. Att det finns en relation mellan två klasser syns i koden genom att klassnamnet för den ena klassen återfinns i den andra. Detta innebär att den senare klassen inte kan kompileras utan att den förstnämnda finns tillgänglig.

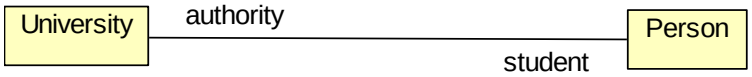
Generalisering (generalization)
Arv motsvaras i UML av relationen *generalisering.* Observera att begreppen arv och generalisering används för samma mekanism fastän de har olika utgångspunkter:

- Arv utgår från basklassen: en deriverad klass ärver basklassen.
- Generalisering utgår däremot från den deriverade klassen: basklassen är en generalisering av den deriverade klassen vilket också är innebörden av *substitutionsprincipen* som säger att en deriverad klass ska kunna ersätta sin basklass (men inte tvärtom).

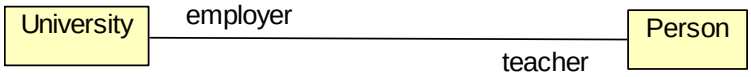
UML använder en ofylld triangel med en spets mot basklassen som symbol för generalization:




Association (association)
När en klass innehåller ett attribut (variabel/objekt) av en annan klass finns det en relation mellan klasserna som kallas association. Den mest allmänna formen av association ritas med ett streck mellan klasserna. Vid associationens ändar kan man ange ”roller”, som ofta överensstämmer med attributnamn i programmet.




De två klasserna kan även vara associerade på andra sätt:

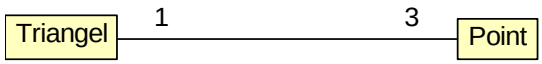



Kardinalitet, eller multiplicitet, anger antalet objekt av de olika typerna i en association. Kardinalitet är heltal, som 1 eller 2, eller ett intervall, som 0..2. Man använder symbolen * för ett obegränsat antal, ensamt eller i intervall som 1..*.
Rollnamn och kardinalitet som används i en klass skrivs närmast den andra klassen och där är placeringen fri. Normalt sätter man ett namn på associationen ovanför eller på linjen som klargör relationen. Man kan också tillfoga en fylld triangel vid namnet som visar i vilken riktning namnet skall tolkas i förhållande till klasserna.
Nedanstående diagram visar en association, ’employment’, mellan University i rollen som arbetsgivare och Person i rollen som lärare. Innebörden är att universitetet kan ha godtyckligt många (men minst en lärare) och att en lärare bara kan vara anställd vid exakt ett universitet.

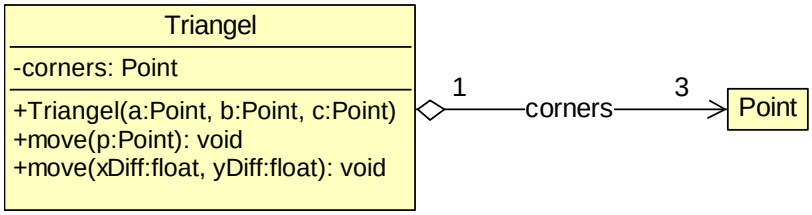
En triangel har som bekant tre hörn och vi kan alltså utnyttja Point i en Triangel-klass:

```
/*     File:  Triangle.h
       Topic: Definition of class Triangle
*/

#ifndef TRIANGLE_H
#define TRIANGLE_H

#include "Point.h"
#include <iostream>

using std::ostream;

class Triangle {
private:
    Point p1,p2,p3;
public:
    /* Constructors */
    Triangle(Point a, Point b, Point c);

   /* Getters & setters */
    void move(Point);
    void move(float,float);

    Point tp() const;  // Tyngdpunkt - Center of gravity

   /* Facilitators */
    void printOn(ostream &os=cout) const;
};

#endif
```

Relationen mellan Triangle och Point kan enklast beskrivas som



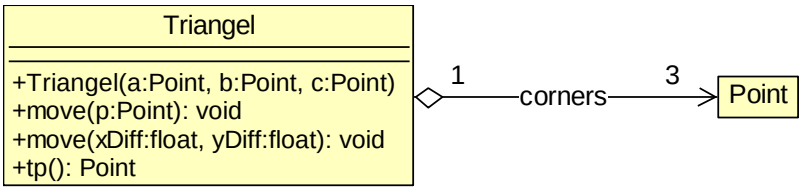

och vi kan säga att Triangel är en *klient* till Point eftersom Triangel utnyttjar tjänster hos Point. En mer detaljerad bild kan vara




där datamedlemmen/attributet `corners` alltså är någon form av kontainer för Point-objekt.

*Eftersom en association innebär att en klass har ett attribut av den andra klassens typ så är det vanligt att inte visa denna datamedlem i klassrutan utan låta den utgöra namnet på associationen.* Detta gäller inte primitiva typer t.ex. int och float och inte heller klassen String, dessa visas i stället i klassrutan.

Detta innebär att ovanstående klassdiagram kan förenklas till

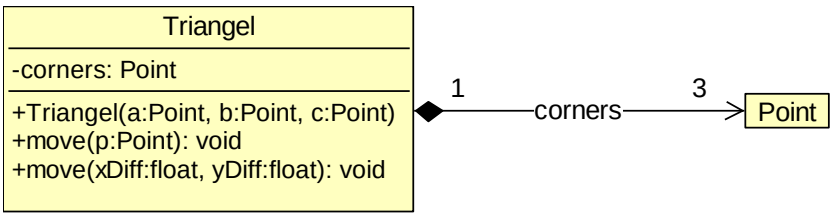



Ovanstående klassdiagram uttrycket ytterligare två saker:

- Den öppna pilen i riktning mot Point innebär att *navigering* endast är möjlig i den riktningen, dvs Triangel kan skicka meddelanden till sina Points men att Point inte känner till Triangel. Om pilen saknas förutsätts att navigeringen är dubbelriktad.

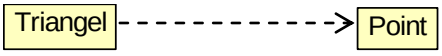
- Den ofyllda romben i anslutning till Triangel anger att associationen är en *aggregation*, dvs en relation av typen **'uses-a'**. Triangel använder Point-objekt och accessar dessa via *referenser eller pekare* som lagras i datamedlemmen corners. Objekten skapas utanför Triangel och skickas som argument till konstruktorn. Triangel 'äger' alltså inte dessa objekt. De kan i princip användas även av andra objekt.

En starkare form av koppling mellan Triangel och Point där Triangel skapar sina egna Point-objekt och där dessa endast lever inom Triangelobjektets scope och livslängd kallas *composition*, de skapas och dör med det omgivande objektet. Detta uttrycks med en fylld romb:




Denna typ av relation uttrycker alltså ett **'has-a'**-förhållande.

En annan aspekt på relationen mellan klasserna Point och Triangle är att Triangle är *beroende* av Point eftersom ändringar av det publika interfacet i Point tvingar fram förändringar av klassen Triangle. I UML kan vi uttrycka detta beroende (eng. dependency) på följande sätt:



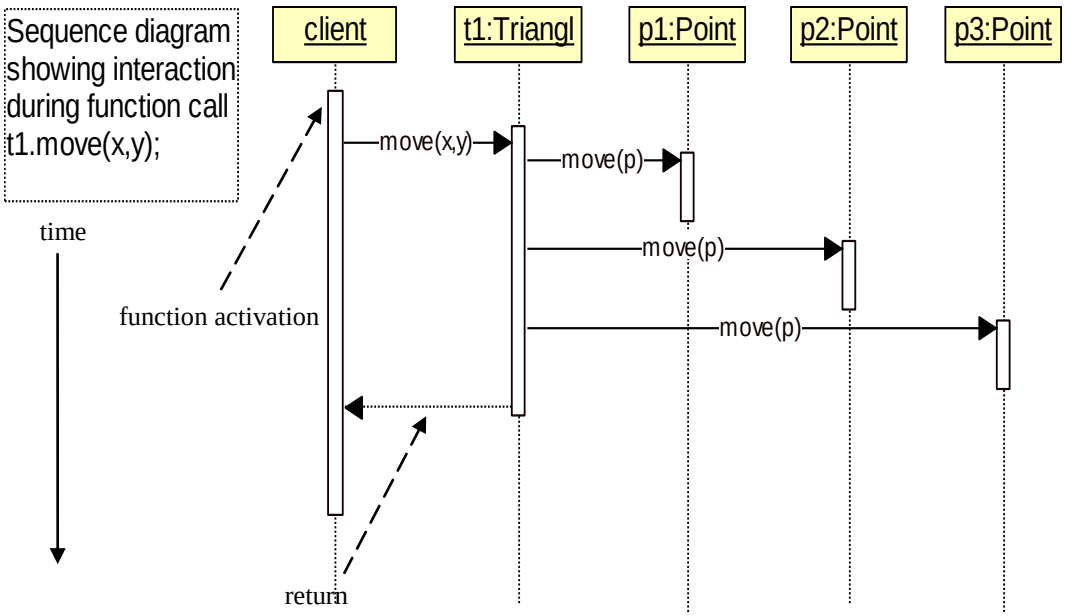

## Sekvensdiagram

Klassdiagram uttrycker endast struktur, dvs *statiska* egenskaper, inte hur objekt samarbetar under exekvering, *dynamiska* egenskaper. Detta uttrycks med hjälp av interaktionsdiagram. Ett mycket användbart sådant diagram är *sekvensdiagrammet.* Programmet TestTriangle (exTriangle.zip) skapar ett Triangle-objekt och utför operationer på detta.

Sekvensdiagrammet nedan visar i tidsordning de anrop som görs när objektet client (main-funktionen) anropar funktionen

```
void Triangle::move(float x, float y) {
    p1.move(Point(x,y));
    p2.move(Point(x,y));
    p3.move(Point(x,y));
}
```

för Triangelobjektet t1.




Mer om UML kommer senare i kursen.

## Initierings- och destruktionsordning för objekt

Antag att vi har en Triangel-klass som som har tre Point-objekt som datamedlemmar (composition). När ett Triangel-objekt ska initieras måste de ingående Point-objekten skapas/initieras först.

Regel:

När ett objekt skapas så skapas först de ingående datamedlemmarna. Detta sker i den ordning datamedlemmarna står uppräknade i klassdefinitionen. Initieringen av varje datamedlem sker i två steg:

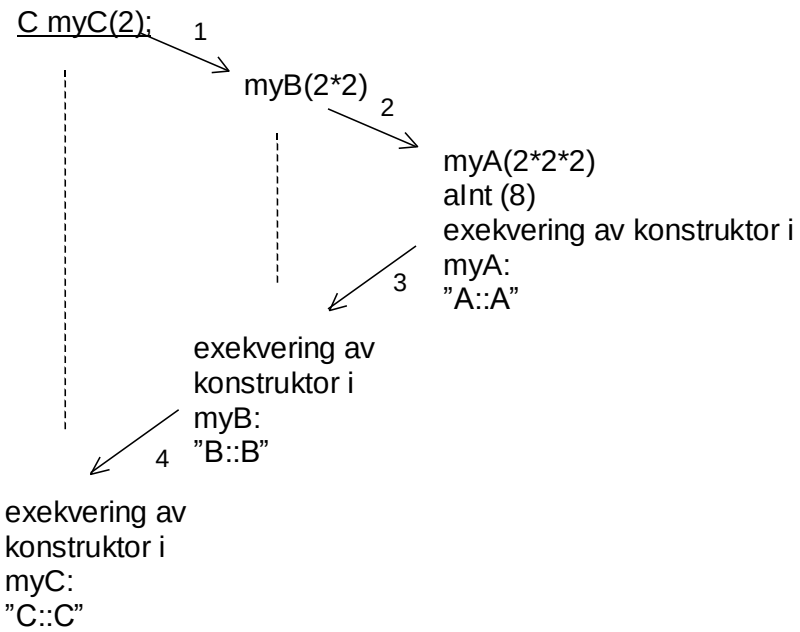
1. Värden för datamedlemmarnas initiering kopieras från konstruktorns initieringslista.
2. Exekvering av datamedlemmens konstruktor om datamedlemmen är av klasstyp.

När samtliga datamedlemmar är initierade exekveras konstruktorns kropp för det aktuella objektet.

Om en datamedlem själv innehåller klassobjekt som attribut tillämpas denna regel rekursivt.
Exempel

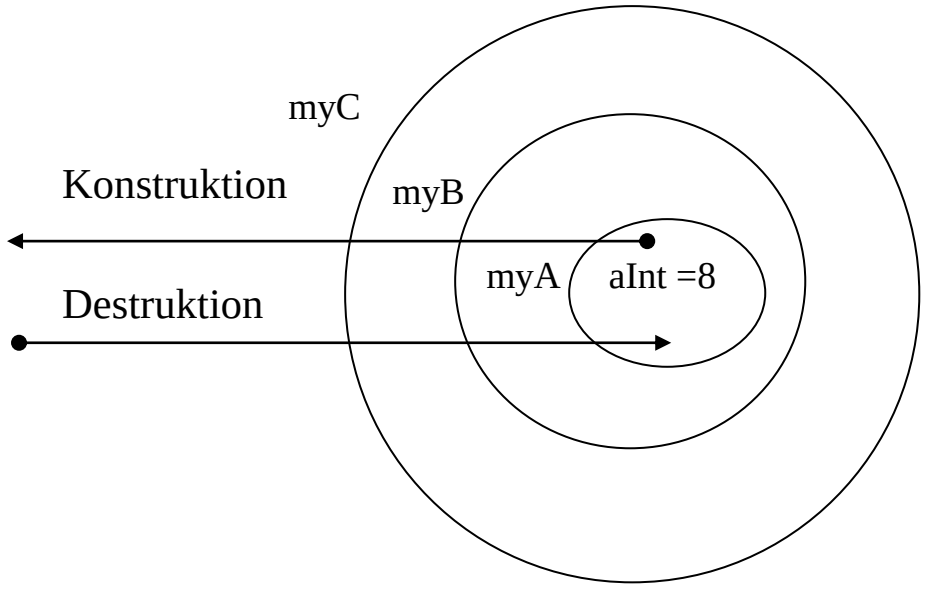
```
class A {
private:
	int aInt;
public:
	A(int a) : aInt(a){
		cout << "A::A" << endl;
	}
};

class B {
private:
	A myA;
public:
	B(int b) : myA(2 * b){
		cout << "B::B" << endl;
	}
};

class C {
private:
	B myB;
public:
	C(int c) : myB(2 * c){
		cout << "C::C" << endl;
	}
};
```

Om vi definierar ett objekt av typen C:
`C myC(2);` så får vi följande serie av initieringar:




myC initierat!

Initieringsordningen kan alltså sägas vara inifrån och ut. Datamedlemmarna i ett objekt kan därmed garanteras existera innan konstruktorn för objektet exekveras.
Destruktionsordningen är den motsatta. Skälet till detta är att inget objekt ska kunna kunna accessa datamedlemmar som inte existerar.




## Default-versioner av medlemsfunktioner och operatorer

Om vissa medlemsfunktioner / operatorer inte är definierade i en klass genereras default-versioner av C++-kompilatorn.

Ex.

```
class MyClass {

};
```

Följande medlemsfunktioner genereras:

- parameterlös konstruktor MyClass( ); } Utför ingenting
- destruktorn ~MyClass( );

- tilldelningsoperatorn operator=(const MyClass &) } Utför medlemsvis kopiering av datamedlemmar
- kopieringskonstruktorn MyClass(const MyClass &)

## Överlagring av operatorer

Ex. Klass Complex för representation av komplexa tal på formen c = a + b*i .

```
class Complex {
      float r, i;   // data members: real and imaginary part
public:
      // constructors/destructor
      Complex(float a = 0, float b = 0); // constructor

      Complex(const Complex &); // copy constructor

      // getters
      float re() const { return r; }
      float im() const { return i; }

      // operations
      Complex add(const Complex &) const;  // +
      Complex sub(const Complex &) const;  // -
      Complex mul(const Complex &) const;  // *
      Complex div(const Complex &) const;  // /

      // facilitators
      void printOn(ostream &strm = cout) const;  // insert in strm
      void readFrom(istream &strm = cin); // extract from strm

      bool isEqual(const Complex &) const;  // = =
      bool isNotEqual(const Complex &) const; // !=

      // operators
      Complex& operator=(const Complex &); // assignment
};

// overloaded operators (NB not members!)

Complex operator+(const Complex &c1, const Complex &c2);
Complex operator-(const Complex &c1, const Complex &c2);
Complex operator*(const Complex &c1, const Complex &c2);
Complex operator/(const Complex &c1, const Complex &c2);

bool operator==(const Complex &c1, const Complex &c2);
bool operator!=(const Complex &c1, const Complex &c2);

ostream& operator<<(ostream &, const Complex&); // insertion
istream& operator>>(istream &, Complex&); // extraction
```

Definitionen `Complex c1, c2(2,3), c3(7);`

ger följande Complex-objekt:

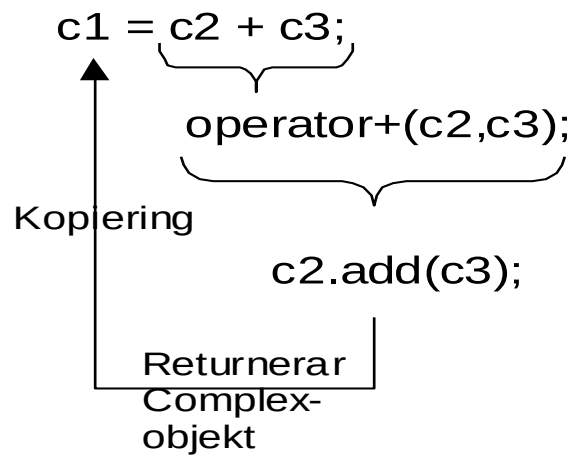
c1 0+0i (deafultvärden i konstruktorn)
c2 2+3i
c3 7+0i implicit konvertering int 7 → float 7 → Complex(7,0)

`c1 = c2 + c3;` översätts av kompilatorn till

```
c1 = operator+(c2,c3);
```

eftersom en sådan version av +-operatorn finns överlagrad. operator+ utnyttjar i sin tur medlemsfunktionen `add`:

```
Complex operator+(const Complex &c1, const Complex
&c2) {
    return c1.add(c2);
}


Complex Complex::add(const Complex &a) const  {
    Complex tmp;
    tmp.r = r + a.r;
    tmp.i = i + a.i;
    return tmp;
}
```




Fråga:
Varför är de aritmetiska operatorerna överlagrade som fristående funktioner och inte som medlemmar?

Svar:
För att få ett symmetriskt interface:

Med operatorfunktionerna som medlemmar får vi första operanden implict via this-pekaren.

`c1 = c2 + 5` tolkas av kompilatorn som

```
c1 = c2.operator+(Complex(5,0)) // OK
```

men

`c1 = 5 + c2` tolkas av kompilatorn som

```
c1 = 5.operator+… Funkar inte!
```

Med operatorfunktionerna som fristående funktioner som tar två explicita argument försvinner detta problem.

## Konverteringsoperator

En viss typ av data t.ex. en skriven textrad kan representeras av olika datayper och klasser, t.ex

- STL-klassen string
- en STL-vector <char>
- en vanlig C-string dvs. en array av typen char[].

Dessa representationer är logiskt nära relaterade till varandra. Önskvärt att kunna blanda olika representationer i samma uttryck. T.ex. sammanslagning av en C-sträng, en string och en vector<char> till en ANSI-string. C++ innehåller mekanismer för att åstadkomma detta. Antag att klasserna A och B är logiskt relaterade:

```
class A {
      ...
};

class B {

public:
      B();  // default
      B(A aObj);        // Implicit konvertering A→ B vid behov
      operator A();     // Implicit konvertering B→ A vid behov
};
```

OBS! C++ kompilatorn utnyttjar konstruktorer som kan ta ETT argument för att utföra implicita typkonverteringar. Denna typ av implicita typkonverteringar kan förhindras m.h.a. nyckelordet *explicit* vid definitionen av konstruktorn.

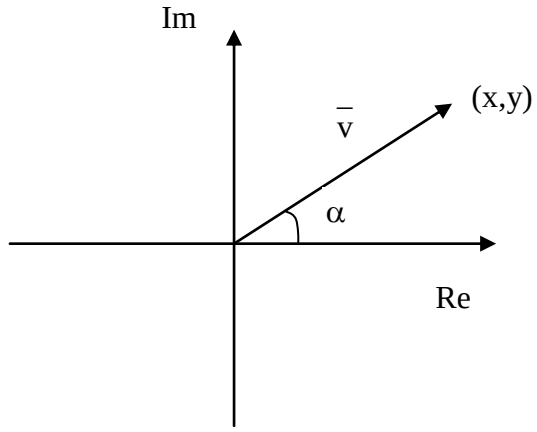
Konstruktorn `B(A aObj)` definierar hur ett B-objekt skapas från ett motsvarande A-objekt.

Konverteringsoperatorn `B::operator A()` definierar hur ett B-objekt konverteras till motsvarande A-objekt.

Med hjälp av dessa kan kompilatorn implicit konvertera A ←→ B.

Ex.

Ett komplext tal kan även representeras på polär form genom längden av en vektor och vektorns vinkel mot reella axeln.




$$|\bar{v}| = \sqrt{x^2 + y^2}$$
$$\alpha = \arctan\left(\frac{x}{y}\right)$$

} klassen / Polar

```
#include<iostream>
#include"Complex.h"

class Polar {
private:

	float ang;		// angle in degrees
	float len;		// length of vector


public:
	// constructor: angle and length
	Polar(float a = 0, float l = 0);

	// constructor: initialize from a Complex,
	// also for implicit conversion Complex => Polar
	Polar(const Complex &c);

	// getters
	float length() const { return len; }
	float angle()  const { return ang; }

	// faciliators
	void printOn(ostream &strm = cout) const;
	void readFrom(istream &strm = cin);

	// conversion operator
	// implicit conversion Polar => Complex
	operator Complex();
};
```

Klassen Polar är logiskt relaterad till Complex eftersom de båda representerar samma underliggande fenomen, ett komplext tal. För att kunna använda Complex och Polar tillsammans utrustar vi klassen Polar med:

1. En konstruktor som tar en Complex som argument. Det innebär att ett Polar-objekt kan initieras från ett Complex-objekt. Konverteringen Complex → Polar sker implicit, dvs ett komplex-objekt kan användas där ett Polar-objekt förväntas eftersom kompilatorn gör konverteringen automatiskt vid behov.

2. En konverteringsoperator (conversion operator), `operator Complex`, som definierar konverteringen Polar → Complex. Används implicit när ett Polar-objekt används som argument där ett Complex-objekt förväntas.

Resultat: Polar- och Complex-objekt kan blandas och ersätta varandra i uttryck. Observera att detta uppnås utan att klassen Complex behöver förändras.

Implementation:

```
const float DEG2RAD = 4 * atan(1) / 180;   // pi/180
const float RAD2DEG = 180 / (4 * atan(1)); // 180/pi

Polar::Polar(const Complex &c) {  // How to make a
Polar from a Complex
      float x = c.re();
      float y = c.im();
      len = sqrt(x*x + y*y);
      ang = RAD2DEG*atan(y / x);
}

Polar::operator Complex() {  // Use a Polar when a
Complex is expected
      float x, y;
      x = len * cos(ang*DEG2RAD);
      y = len * sin(ang*DEG2RAD);
      return Complex(x, y);
}
```

Testkod (fullständig kod finns i exComplex.zip och exPolar.zip):

```
void print(Polar p) {
      cout << "Complex number: " << p << endl;
};

Polar p1(45, 1.41421356); // (1+i)
Complex c1(2, 2);         // (2+2i)
Polar p2(c1); // Initialize a Polar with a Complex
cout << p2 << endl;  // ang=45 len=2.82843
Polar p3(0, 1);

// Usage of Polar where Complex is expected,
/* Complex c2 =
operator+( operator+(c1, p1.operator Complex()),
p3.operator Complex());
*/

Complex c2 = c1 + p1 + p3; // 2+2i + 1+i + 1
cout << c2 << endl;  // 4+3i
Complex c3(0.8660254, 0.5); // (sqrt(3)/2 , 0.5)
print(c3);  // Implicit conversion Complex => Polar
            // using constructor Polar(Complex)
```