KLASSER

Klasser

Vid objektorienterad systemutveckling är klassbegreppet centralt. Ett objektorienterat programmeringsspråk måste därför stödja detta begrepp.

I C++ har man valt att införa det reserverade ordet **class**. Den som tidigare har erfarenhet av andra språk kan betrakta begreppet **class** som en **utvidgning** av begreppet ”egendefinierad typ” eller ”abstrakt datatyp”. I andra språk används i stället t.ex. begreppet **record** eller **struct**.

Klassen kan betraktas som en mall för objekt som är av samma typ.

Inkapsling

Abstrakt datatyp

Objektorienterad design innebär en utvidgning av begreppet dataabstraktion. En abstrakt datatyp, dvs en klass, ska inte bara dölja den egentliga datastrukturen utan också hur de operationer som kan utföras av objektet är implementerade.

I den objektorienterade modellen finns stöd för begreppet **inkapsling**. Man skiljer på klassens **gränssnitt** och hur detta gränssnitt är **implementerat**.

Public och private

Klassens gränssnitt deklareras som **public** vilket innebär att det är synligt för objekt av andra klasstyper. Hur detta gränssnitt sedan är implementerat kapslas in bl.a. genom att man deklarerar det som inte ingår i gränssnittet som **private**.

Klassmedlemmar

En klass har medlemmar. Huvudsakligen kan dessa vara av två typer; datamedlemmar eller funktionsmedlemmar.

Datamedlemmar

En datamedlem kan vara av någon standardtyp (t.ex. int,long,double), eller en referens till en sådan typ men kan också vara av någon annan klasstyp och motsvarar då ett delobjekt.

Datamedlemmarna håller den information som behöver lagras om varje objekts tillstånd. Som synonymer till begreppet datamedlem brukar man ibland också använda ordet **attribut** eller **tillståndsvariabel**. Vi kommer växelvis att använda även dessa begrepp.

Klassens datamedlemmar brukar alltid ingå i klassens implementation och deklareras därför vanligtvis som **private**.

Exempel

Antag att vi i vårt system behöver objekt som kan representera begreppet **cirkel** och att en cirkels tillstånd i vårt system bestäms av dess position i höjd och sidled samt radie. Vi ger därför vår klass de datamedlemmar som behövs för att hålla reda på cirkelns tillstånd i avseende på position och storlek.

```
class Cirkel {
public:
        // Andra medlemmar .....
        // .....
private:
        // Datamedlemmar
        double radie;
        long    xpos;
        long    ypos;
};
```

Funktionsmedlemmar

I den objektorienterade modellen använder man begreppet **metod, operation, tjänst** eller **meddelande**. Vi kommer att växelvis använda dessa begrepp och betraktar dem i detta sammanhang som synonymer. I C++ motsvaras detta av att en klass har **medlemsfunktioner** och att dessa kan anropas.

Man strävar normalt mot att endast tillåta manipulation av objektets datamedlemmar via dess medlemsfunktioner dvs metoder. Det finns **fyra** standardtjänster;

1. Metoder för initiering av objekt
2. Metoder för att ta del ett objekts tillstånd
3. Metoder för att förändra ett objekts tillstånd
4. Metoder för att avinitiera ett objekt

Initiering av objekt

Om datamedlemmarna i ett objekt inte är initierade så får vi problem när vi anropar en medlemsfunktion som använder dessa på något sätt. Vi får ett resultat som saknar mening och vi riskerar ibland att programmet och datorn hänger sig.

Konstruktor

När vi skapar ett objekt anropas alltid någon av klassens konstruktorer. Om klassen saknar konstruktorer får objektet ett odefinierat tillstånd. Varje klass bör därför ha minst en konstruktor. I klassens konstruktor ser vi till att objektet initieras.

I C++ är en konstruktor en medlemsfunktion som har samma namn som klassen. Om klassen heter **Cirkel** blir därför klassens konstruktor en funktion med samma namn.

Exempel

Antag att vi i vårt system vill undvika cirkelobjekt som har ett odefinierat tillstånd. Vi definierar därför en konstruktor som avkräver användaren av klassen initieringsvärden som ger ett nytt cirkelobjekt ett väldefinierat starttillstånd.

```
class Cirkel {
public:
        Cirkel(long x,long y,double r);  // Konstruktor
        // Andra medlemmar .......
        // .....
private:
        // Datamedlemmar
        double radie;
        long     xpos;
        long     ypos;
};
```

Ta del av objektets tillstånd

I klassen finns ofta metoder som kan användas för att ta del av objektets tillstånd utan att förändra detta.

Exempel

Antag att ett cirkelobjekt i vårt system ska kunna berätta om sitt tillstånd. både grafiskt i form av en figur i en viss position och i textform.Vi ger därför klassen metoderna **show** och **print**.

```
class Cirkel {
public:
        Cirkel(long x,long y,double r);  // Konstruktor
        void show();
        void print(ostream& os);
        // Andra medlemmar .......
        // .....
private:
        // Datamedlemmar
        double radie;
        long     xpos;
        long     ypos;
};
```

Förändra objektets tillstånd

I klassen finns ofta metoder som kan användas för att förändra objektets tillstånd. Det är vanligt, men inte alltid nödvändigt eller önskvärt, att man i klassen definierar en metod för manipulation av varje attribut (datamedlem).

Exempel

Antag att ett cirkelobjekt i vårt system ska kunna ändra position samt radie.Vi ger därför klassen metoderna **moveto, move**, **setradius** och **expand**.

```
class Cirkel {
public:
        Cirkel(long x,long y,double r);  // Konstruktor
        void show();
        void print(ostream& os);
        void moveto(long x,long y);
        void move(long dx,long dy);
        void setradius(double r);
        void expand(double delta);
        // Andra medlemmar .......
        // .....
private:
        // Datamedlemmar
        double radie;
        long     xpos;
```

```
        long    ypos;
};
```

Avinitiera objektet

Destruktor

I C++ kan vi definiera en medlemsfunktion som anropas automatiskt innan ett objekt försvinner. Denna kallas för klassens destruktor. Om vi definierar denna funktion kan vi se till att objekten avinitieras på rätt sätt. Vi kan t.ex. behöva frigöra minnesresurser eller bokföra att objektet försvinner.

En destruktor är en funktion som har samma namn som klassen men med prefixet ~ (tilde).

Destruktorn anropas alltid implicit (automatiskt) när exekveringen går utanför det område där objektet är synligt eller om man använder delete-operatorn på en pekare som pekar på ett objekt som har allokerats med new-operatorn.

Exempel

Vi ger vår klass en destruktor.

```
class Cirkel {
public:
        Cirkel(long x,long y,double r);         // Konstruktor
        ~Cirkel();                              // Destruktor
        void show();
        void print(ostream& os);
        void moveto(long x,long y);
        void move(long dx,long dy);
        void setradius(double r);
        void expand(double delta);
        // Andra medlemmar .....
        // .....
private:
        // Datamedlemmar
        double radie;
        long    xpos;
        long    ypos;
};
```

Funktionsdefinitioner

Vi måste sedan definiera klassens medlemsfunktioner. I funktionsdefinitionen anges i vilken klass som funktionen är medlem genom att funktionsnamnet föregås av klassnamnet samt dubbla kolon (::). En medlemsfunktion deklareras därför på nedanstående sätt. Lägg märke till att punktoperatorn inte behöver användas i en medlemsfunktion för att komma åt de olika medlemmarna.

```
#include <iostream.h>

class Cirkel {
public:
     Cirkel(long x,long y,double r);  // Konstruktor
     ~Cirkel();                                          //
Destruktor
     void show();
     void print();
     void moveto(long x,long y);
     void move(long dx,long dy);
     void setradius(double r);
     void expand(double delta);
private:
     // Datamedlemmar
     double radie;
     long     xpos;
     long     ypos;
};

Cirkel::Cirkel(long x,long y,double r)
{
     xpos     = x;
     ypos     = y;
     radie    = r;
}
Cirkel::~Cirkel()
{
     // Ingen åtgärd än....
}
void Cirkel::show()
{
     // Miljöberoende....
     // Ta reda på hur man ritar en cirkel i din    //
miljö....
}
void Cirkel::print()
{
```

```
     cout << "******** Cirkel ********\n";
     cout << "xpos  = " << xpos << endl;
     cout << "ypos  = " << ypos << endl;
     cout << "radie = " << radie << endl;
}

void Cirkel::moveto(long x,long y)
{
     xpos = x;
     ypos = y;
}

void Cirkel::move(long dx,long dy)
{
     xpos += dx;
     ypos += dy;
}

void Cirkel::setradius(double r)
{
     radie = r;
}

void Cirkel::expand(double delta)
{
     radie += delta;
}
```

Skapa objekt

Man skapar objekt på samma sätt som när man inför andra typer av variabler. Först anger man typen av objekt dvs klassens namn och sedan det namn som skall användas för att referera till objektet (identifierare). Om klassen har en konstruktor som måste ha parametrar måste dessa anges.

Anrop av medlemsfunktion

När vi vill anropa en medlemsfunktion använder vi punkt eller piloperatorn.

Obs!

Vi får här en stark bindning mellan en datatyp (klass) och de funktioner som ska behandla denna datatyp (medlemsfunktioner). Vi kan inte anropa en medlemsfunktion (skicka ett meddelande) utan att markera vilket objekt som ska utföra funktionsanropet (ta emot meddelandet).

En medlemsfunktion kan endast anropas med punkt eller piloperatorn. Med punkt eller piloperatorn markerar vi vilket objekt som ska få vilket meddelande.

Lägg märke till hur klassens funktioner har tillgång till klassens övriga medlemmar utan användning av punkt eller piloperatorn.

Exempel

Nedan skapar vi två objekt av typen Cirkel. Sedan ber vi dessa att skriva ut sig (print), förflytta sig samt ändra storlek samt återigen skriva ut sig.

```
main()
{
     Cirkel alfa(100,100,50);
     Cirkel beta(200,200,50);

     alfa.print();
     beta.print();

     alfa.moveto(200,100);
     alfa.setradius(10);

     beta.move(50,50);
     beta.expand(30);

     alfa.print();
     beta.print();

     return 0;
}
```

Resultat

```
******** Cirkel ********
xpos  = 100
ypos  = 100
radie = 50
******** Cirkel ********
xpos  = 200
ypos  = 200
radie = 50
******** Cirkel ********
xpos  = 200
```

```
ypos  = 100
radie = 10
******** Cirkel ********
xpos  = 250
ypos  = 250
radie = 80
```

Medlemsfunktion som inlinefunktion

Man kan välja att definiera medlemsfunktionerna inuti klassdeklarationen. Dessa blir då implicit deklarerade som inline (dvs utan att man behöver ange detta.

```
class Cirkel {
public:
   Circle(double r)
   {
       radie = r;
   }
   double area()  // Blir inline
   {
       return PI * radie * radie;
   }
   double omkrets()// Blir inline
   {
       return  PI * 2*radie;
   }
private:
   double radie;
};
```

Man bör endast använda inlinefunktioner om de är små, dvs innehåller få rader. Du kan också explicit deklarera en medlemsfunktion som inline utanför klassdeklarationen.

```
inline double omkrets() // Blir inline
{
   return  PI * 2*radie;
}
```

Rekommendation

Inlinefunktioner används av optimeringsskäl när man vill optimera i avseende på tid. Däremot ger överdriven användning av inlinefunktioner stor kodstorlek. Vi rekommenderar dig rent stilmässigt dessutom att aldrig göra definitionen för en inlinefunktion i klassdefinitionen utan göra denna explicit under klassdefinitionen.

Man brukar också rekommendera att man placerar inlinefunktioner i en separat fil som inkluderas i klassens deklarationsfil.

Friend

Funktion som vän

En klass kan deklarera en funktion som **friend**. Denna funktion har fullständig accessrätt till klassens medlemmar.

En vänfunktion anropas som en vanlig funktion.

Klass som vän

En klass kan deklarera en annan klass som **friend**. Medlemmarna i den andra klassen får då full accessrätt.

Konstruktor och destruktor

Destruktorn anropas implicit (automatiskt) när exekveringen går utanför det område där objektet är synligt eller om man använder delete-operatorn på en pekare som pekar på ett objekt som har allokerats med new-operatorn. Den kan också anropas explicit som ett vanligt funktion.

this-pekaren

Varje medlemsfunktion får implicit ett argument som är en pekare till det objekt som gjort anropet. Denna pekare får namnet **this** och kan användas för att t ex returnera en pekare till objektet eller objektet själv från en av klassens medlemsfunktioner.

Dynamisk minneshantering

New

Ett objekt kan också allokeras dynamiskt under programexekveringen. Detta görs med operatorn **new** som returnerar en pekare till det allokerade objektet. Det är den så kallade "heapen" som används. Om klassen saknar konstruktor eller har en konstruktor som inte har några argument kan detta göras på följande sätt:

```
Cirkel *pCirkel   =   new Cirkel;
Record *pRecord   =   new Record;
```

Om klassen har en konstruktor med argument måste dessa anges också vid användning av new:

```
Cirkel *pCirkel   =   new Cirkel(400,500,50);
Record *pRecord   =   new Record(”Pelle”,”Lund”);
```

Delete

Objekt som har allokerats med operatorn **new** finns kvar även efter det block där anropet gjorts. För att frigöra minne måste **delete** operatorn användas. Man måste därför spara en pekare till det allokerade objektet.

Har klassen en destruktor så anropas denna när du använder delete.

delete pCirkel;
delete pRecord;

Exempel med new och delete

Det är lätt hänt att tar minne från "heapen" i anspråk och sedan glömmer att lämna tillbaka detta. Om en klass innehåller pekare till dynamiskt minne och ett temporärt objekt av typen ska plockas bort från stacken, måste man komma ihåg att också deallokera minnet på "heapen".

`Exempel`

Lägg märke till hur klassens konstruktor med hjälp av **delete** lämnar tillbaka det minne som pekarna pekar på. Om man glömmer detta kommer programmet så småningom att krascha när "heapen" är full.

Static

Datamedlem som är static

En klassmedlem som är static delas av alla klassens objekt. Om medlemmen ska initieras så måste detta göras utanför klassdefinitionen.

Man brukar ibland använda begreppet **klassvariabel** om en datamedlem som är deklarerad som **static** och begreppet **instansvariabel** om en datamedlem som inte är staticdeklarerad.

Exempel

En variabel som är deklarerad som static kan t ex användas för att hålla reda på hur många objekt som finns av en viss typ. Det är i så fall en uppgift som delas av alla objekt av samma typ.

Medlemsfunktion som är static

En medlemsfunktion som är static kan anropas utan att man anger något speciellt objekt. I en medlemsfunktion som är static saknas därför **this**-pekaren. Vill man kunna manipulera med något speciellt objekt kan detta anges som argument.

Funktionsmedlemmar som är static kan t ex användas för att behandla datamedlemmar som också är static speciellt om dessa är också är deklarerade som private.

Funktion som anropas före main

Om man väljer att deklarera ett objekt globalt och det finns en konstruktor deklarerad för klassen så kommer denna att anropas före main.

Övningsuppgifter

Uppgift 1

Definiera en klass som representer begreppet **Rektangel**. Antag att man i vårt system vill att rektanglars tillstånd bestäms av dess position och storlek. I objektens gränssnitt skall ingå metoder för att initiera objekt samt förändra och efterfråga objektens tillstånd

Uppgift 2

Antag att vi skall konstruera ett banksystem och att vi skall börja med att designa en klass som skall representera bankkonton. I vårt banksystem är det viktigt att varje kontoobjekt vet sitt kontonummer samt saldo. Dessutom måste man få göra transaktioner samt få ett saldobesked. Gör klassen bankkonto och pröva den sedan i ett program.

Uppgift 3

Antag att vi skall konstruera ett lönesystem och att vi skall börja med att designa en klass som skall representera anställda personer. I vårt lönesystem är det viktigt att varje personobjekt vet sitt namn, adress samt månadslön. Dessutom måste man kunna ge en person högre lön samt kunna begära ett lönebesked. Gör klassen Anstelld och pröva den sedan i ett program.

Uppgift 4