Föreläsning Modul 6 – Introduktion till objektorienterad programmering med C++

Programmering med C: Grundkurs, DA105A

Malmö University

OOP Introduktion

- Denna föreläsningen kommer att ge en introduktion till objektorienterad programmering (förkortas oftast OOP) med C++.
- OOP innebär att vi måste tänka i andra banor än vid c-programmeringen som vi ägnat oss åt tidigare i kursen.
- I C så tänker vi i funktioner.
- I C++ så tänker vi i objekt istället.
- Ett *objekt* är en enhet som innehåller både data (kallas även egenskaper hos objekten, eller *attribut* hos objekten) samt även de operationer som man vill utföra på denna data.

OOP

- Den grundläggande ideen inom OOP är att man (i sitt datorprogram) låter ett antal definierade objekt samarbeta för att gemensamt lösa en angiven uppgift.
- Man kan säga att man som OOP-programmerare i sitt datorprogram simulerar ett förlopp med målet att lösa den givna uppgiften.
- Låt oss nu anta att vi har fått till uppgift att skriva ett datorprogram som ritar upp det geometriska objektet kvadrat på vår datorskärm samt skriver ut viktiga egenskaper såsom sidlängd och area hos kvadraten vid förfrågan om dessa ifrån användaren. Det ska också vara möjligt för användaren att förflytta kvadraten en given sträcka.

OOP

Vi ska nu titta närmare på hur vi löser denna ovannämda uppgift genom att tillämpa *OOP modellering*, vilket består av följande tre steg:

1. **Identifierar relevanta objekt i vår omgivning för vår givna uppgift**. Ett exempel på ett objekt för just vår uppgift är en kvadrat. Generellt kan man säga att objekten till vårt program väljs av programmeraren bland de ting, saker som kan observeras i vår omgivning. I vårt fall är ju en kvadrat ett ting som vi har erfarenhet av ifrån vår omgivning, denna är ju en geometrisk figur som består av fyra lika långa sidor och vi kan t ex se kvadraten framför oss när vi ritar den på datorskärmen.

OOP

2. **Vilka egenskaper har nu våra identifierade objekt som är av relevans för att vi ska kunna lösa den givna uppgiften?** När vi betraktar vårt kvadrat objekt så är ju t ex sidlängd en relevant uppgift eftersom denna efterfrågades i uppgiften, men även kvadratens position är relevant eftersom en användare ska kunna förflytta kvadraten en given sträcka. Vi ser att en kvadrat kan ha olika positioner; den kan vara uppritade mitt på datorskärm, eller i högra hörnet, och etc.

Vi säger följaktigen att kvadratens *data* (även kallade datamedlemmar eller attribut) är sidlängd och position.

OOP

3. **Vilka operationer ska man kunna utföra på våra objekt?**

De mest grundläggande operationerna på vårt kvadrat objekt är att:

- rita upp kvadraten t ex på datorskärmen,
- förflytta en redan utritad kvadrat ifrån en initial position på datorskärmen till en ny postion på datorskärmen,
- fråga efter kvadratens sidlängden, och
- fråga efter kvadratens area.

OOP

Sammanfattningsvis:

- Ett objekt (t ex en kvadrat) är en enhet som innehåller data (t ex i form av sidlängd och position) och som kan utföra operationer (såsom förflytta sig ifrån en initial position till en ny position t ex på datorskärmen, rita upp sig på datorskärmen, berätta för oss om sin sidlängd och sin area, o s v).

OOP

- Objektorientad programmering är att tillämpa ett objektorienterat tankesätt för att lösa ett problem med datorns hjälp.
- Att tillämpa ett objektorienterat tankesätt är att använda sig av objekt som samarbetar för att tillsammans lösa en given uppgift.
- Dessa objekt samt deras egenskaper och operationer identifierar vi enligt de tre ovannämnda stegen. Dessa tre stegen är en del av *objektorienterad analys och design*.

OOP

- Ska man jobba objektorienterat så inkluderar detta följande aktiviteter:
  1. objektorientad analys
  2. objektorienterad design
  3. objektorienterad programmering (med detta menar vi själva kodningen.)
- I denna introduktionsföreläsning till objektorienterad programmering med C++ och tillhörande inlämningsuppgift nr 6 så ska vi främst fokusera på punkt 3 ovan, d v s den objektorientade programmeringen och delvis på objektorienterad analys och design, d v s steg 1. och 2. ovan.

OOP

- Vi kommer att gå igenom följande grundläggande och viktiga begrepp inom objektorienterad programmering
  - klass
  - objekt
  - datamedlem
  - medlemsfunktion
  - instans
  - inkapsling
  - public
  - private

Begreppet Klass

Följande exempel visar hur man kan göra en *klass* för att hantera uppritningen av en kvadrat på datorskärmen.

- Antag att vi vill hantera följande data för en kvadrat:
  - position på datorskärmen för uppritning, och
  - sidlängden.

Begreppet Klass

I C skulle vi ha gjort en struct för att hantera denna data enligt följande:

```
struct Square{
    int xpos;
    int ypos;
    int sideLength;
};
```

Begreppet Klass och Datamedlem

Med hjälp av denna struktur kan vi sedan skapa en variabel som lagrar informationen för alla datamedlemmar enligt följande:
Square square;
Antag nu att vi vill komma åt datamedlemmen sideLength (d v s sidlängden) i square. Då använder vi, som vi sett tidigare, punktnotation enligt följande:

```
square.sideLength = 10;
```

Begreppet Klass och Datamedlem

- Nu ska vi titta på hur vi hanterar denna data hos en kvadrat i C++.
- Generellt kan vi säga att i C++ så gör vi en *klass* av vår struct istället. I vårt kvadrat exempel så skapar vi en klass som vi kallar Kvadrat för att hålla samma data som vår struct gjorde i C.
- Och nu tar vi de definierade datamedlemmarna i vår struct i C (d v s x-positionen, y-positionen och sidlängden) och dessa blir istället *attribut* i vår klass Kvadrat i C++.
- Den stora skillnaden mellan struct i C och klasser i C++ är att man i klasser i C++, förutom datamedlemmar associerade till vår datatyp (d v s struct:en), också definierar vilka operationer som man kan göra på dessa datamedlemmar.

Begreppet Klass och dess Operationer

- Exempel: I vår Kvadrat klass i C++ så talar vi även om vilka operationer man ska kunna göra på en kvadrat, och dessa var ju enligt följande:
  - förflytta kvadraten,
  - ta reda på kvadratens sidlängd, och
  - ta reda på kvadratens area.

Klassdefinition

- Vi ska nu se hur vi kan göra en *klass* (eng. *class*) av vår kvadrat och dess datamedlemmar (som ju lagrar information om klassen) och dess associerade operationer.
- När man skapar en klass skriver man en *klassdefinition*.
- Nedanstående är definitionen av en kvadrat (eng. square) klass.

Klassdefinition

```
// Class definition
class Square
{
    private:      // variables are typically declared private
        int xpos;         //datamedlem, or attribute
        int ypos;         //datamedlem, or attribute
        int sidelength;   //datamedlem, or attribute

    public:     //methods are typically declare public
        void move(int dx, int dy); //medlemsfunktion, or method
        int getSideLength();       //medlemsfunktion, or method
        int getArea();             //medlemsfunktion, or method
} //closes the class definition of the class Square
```

Klassdefinition och dess Användning

- Ovanstående var ett exempel på en klassdefinition med några operationer för att hantera en kvadrat. Givetvis kan vi lägga till ytterligare operationer men även ytterligare datamedlemmar.
- Frågan är nu hur vi använder klassen i vårt datorprogram.
- Vi sa tidigare att i OOP så löser vi ett problem genom att vi låter en eller flera *objekt* samarbeta för att leverera lösningen vi söker. Notera här att ovannämda problemet kan vara att t ex utföra en beräkning eller rita upp en kvadrat på datorskärmen.

Objekt och Instans av en Klass

- Efter att vi har definierat en klass så kan vi börja skapa *objekt* av denna klassen.
- Vi skapar ett objekt, vi säger också att vi skapar en *instans* av klassen, i vårt datorprogram enligt följande deklaration:

```
// ett objekt square1 av klassen Square skapas
Square square1;
// ytterligare ett objekt square2 av klassen Square skapas
Square square2;
```

- Notera att square1 och square2 är två olika instanser av samma klass!

Deklaration av Objekt

- Deklarationen av square1 och square2 ovan kan jämföras med en variabeldeklaration som vi känner till sedan tidigare i C.
- Exempelvis deklarationen Kvadrat kvadrat;
  vilken deklarerade en kvadrat av vår egendefinierade datatyp Kvadrat ovan.
- Vi kan alltså se ett objekt som en speciell egendefinierade datatyp.

Deklaration av Objekt

- I och med klassdefinitionen så vet vi att alla objekt som skapas av sin klass har samma egenskaper (d v s samma attribut och samma uppsättning av operationer, d v s alla objekt av en klass uppvisar samma egenskaper). Alla kvadrater har en sidlängd och position och alla kvadrater kan ritas upp och förflyttas, och alla kvadrater kan meddela oss om sin area och sidlängd.

Inkapsling

- Ett annat viktigt och ett av de fundamentala begreppen inom OOP är *inkapsling* av ett objekts datamedlemmar (d v s ett objekts attribut).
- Inkapsling görs genom att deklarera ett objekts attribut med det reserverade ordet *private* i klassdefinitionen.
- Exempel: I vår Square klass har vi deklarerat attributet (datamedlemmen) sideLength med modifieraren private, d v s vi har skrivit enligt följande i vår klassdefinition:

```
private :
int sideLength;
```

Inkapsling

- private deklarationen ovan innebär att vi, för att komma åt information som ett objekt håller, så måste vi skicka ett meddelande till objektet och be objektet skicka ett meddelande tillbaka till oss innehållande den privata (inkapslade) information ifrån objektet som vi söker.

Inkapsling

- Exempel: Vi skickar ett meddelande till vårt kvadratobjekt square1 och ber kvadratobjektet square1 att berätta om sin sidlängd för oss med följande anrop:

```
// detta anrop returnerar objektet square1's sideLength
square1.getSideLength();
```

Inkapsling

- Kvadratens sidlängd kan vi sedan använda oss av i vårt program t ex för att beräkna kvadratens area enligt följande programsnutt:

```
int sqArea;
   ...
\\returns the area of square1
return sqArea = square1.getSideLength()*square1.getSideLength();
   ...
```

- D v s meddelanden skickas med hjälp av klassens medlemsfunktioner.

Inkapsling

- Därmed så ser vi att vi måste definiera objektets operationer, och därmed medlemsfunktionerna i klassdefinitionen, efter hur vi vill att vårt objekt ska uppföra sig, d v s vi vill t ex att en kvadrat ska kunna meddela oss om sin sidlängd och därmed behöver vi definiera metoden som vi kallar getSideLength().
- Sammanfattningsvis: I C++ (och OOP generellt) kapslar man in data och gör denna osynlig för användaren. Datan blir synlig genom medlemsfunktionerna.

Public/Private

- Notera att vi har definierat medlemsfunktionerna i klassen Square är deklarerade *public*.
- Modifieraren public för medlemsfunktionerna innebär att dessa därmed kan nås och användas utanför klassen, d v s square1.getSideLength(), där medlemsfunktionen getSideLength() används för att nå square1's inkapslade data i form av dess sidlängd.
- Notera att medlemsfunktionerna i klassdefinitionen måste implementeras för att klassdefinition ska vara komplett.
- Några av de definierade medlemsfunktionerna (kallas också för metoder) implementeras enligt följande:

Implementation av medlemsfunktioner

```
//medlemsfunktion, or method
void Square::move(int dx, int dy);
{
    xpos=xpos+dx;
    ypos=ypos+dy;
}
//medlemsfunktion, or method
int Square::getSideLength();
{
    return sidelength;
}
```

Implementation av medlemsfunktionerna

```
//medlemsfunktion, or method
int Square::getArea();
{
    return sidelength*sidelength;
}
```

// :: kallas räckviddsoperaorn och t ex Square::move innebär att metoden move tillhör klassen Square.

Konstruktor/Dekonstruktor

- Det enda som återstår nu för att vår klassdefinition av Square ska bli komplett är två specialmetoder som kallas *konstruktor* och *dekonstruktor*.
- En konstruktor är en metod (även medlemsfunktion) som har till uppgift att initiera objektets datamedlemmarna (d v s dess attribut) vid skapandet av objektet. En konstruktor körs automatiskt när ett objekt skapas.

Konstruktor/Dekonstruktor

- En konstruktor har följande egenskaper:
  - en konstruktor har samma namn som klassen och saknar returtyp.
  - en konstruktor utan parametrar kallas ofta default konstruktor medan en konstruktor med parametrar används för att initiera medlemsvariablerna (d v s objektets attribut).
- En dekonstruktor har samma namn som klassen men med ett tilde som första tecken (se nästa sida för ett exempel).
- Man kan bara ha en dekonstruktor i sin klass och denna körs automatiskt när objektet tas bort.

Konstruktor/Dekonstruktor

- Exempel på en default konstruktor och en konstruktor med parametrar samt dekonstruktor för vår klass Square:

```
public:
  Square(); // Default konstruktor
  Square(int ixpos, int iypos, int isideLength); // Konstruktor
  ~Square(); // Dekonstruktor
```

Implementation av Konstruktor

- Implementering av Squares konstruktorer görs enligt följande:

```
Square::Square() // Default konstruktor
{
    xpos = 0;
    ypos = 0;
    sideLength = 0;
}
```

Implementation av Konstruktor

```
Square::Square(int ixpos, int iypos, int isideLength) // Konstruktor
{
    xpos = ixpos;
    ypos = iypos;
    sideLength = isideLength;
}
```

Implementation av Dekonstruktor

Implementering av Squares dekonstruktor görs enligt följande:

```
Square::~Square()
{
    cout << "Now we run the deconstructor" << endl;
}
```

Överlagring av Konstruktorer

- I C++ kan man ha flera medlemsfunktioner och konstruktorer med samma namn så länge dessa har unika parameteruppsättningar (d v s olika antal parametrar eller lika antal men olika parametertyper)
- När en överlagrad medlemsfunktion/konstruktor anropas så väljer C++ kompilatorn rätt m a p parameteruppsättningen.
- Överlagring kan användas t ex för att utföra liknande uppgifter men på data av olika typer.

Överlagring av Operatorer

- I det fall att man vill utföra operationer på objekt med avseende på ett specifikt attribut kan operatorer överlagras till att ge önskad effekt.
- I exemplet nedan överlagras == operatorn till att jämföra arean av två Square objekt.

```
bool Square::operator==(const Square &square) const
{
    return getArea() == square.getArea();
}
```

- I inlämningsuppgift nr 6 kommer vi att överlagra operatorer och konstruktorer.

Set- och Get-funktioner

- I en klass kan vi definiera speciella medlemsfunktioner, så kallade *set-funktioner*, vars uppgift är att ge datamedlemmarna (d v s attributen) värden.
- Vi kan också i klassen definiera speciella medlemsfunktioner, så kallade *get-funktioner*, med vilka vi kan läsa klassens datamedlemmar.

Exempel på en Set-funktion

- Följande är ett exempel på en set-funktion i vår klass Square:

```
// setXPos
// Set xpos, i.e., the value of the x-coordinate
void Square::setXPos(int xPosVal)
{
    xpos = xPosVal;
}
```

Exempel på en Get-funktion

- Följande är exempel på en get-funktion i vår klass Square:

```
// getXPos
// Get xpos, i.e., the value of the x-coordinate
int Square::getXPos() const
{
    return xpos;
}
```

- Notera const deklarationen efter get-funktionen, denna ser till att get-funktionen inte kan ändra värdet på datamedlemmen xpos, vilket är en grundläggande ide inom OOP.

Dynamisk minnesallokering

- I C++ finns det möjlighet för programmeraren att kontrollera allokering och avallokering (frigörande) av minne.
- För detta används operatorerna *new* (allokering av minne) och *delete* (avallokering, frigörande av minne).

Dynamisk minnesallokering

- Betrakta följande exempel:

```
Square *squarePtr;
squarePtr = new Square();
```

- Vad som händer här är att new operatorn allokerar minne av lämplig storlek för objekt av typen Square, därefter anropas default konstruktorn i klassen Square för att initialisera ett Square objekt och returnerar en pekare av typen som är specificerad till höger om new operatorn (d v s i vårt fall en Square*).
- Notera att new operatorn kan användas för att dynamiskt allokera godtyckliga typer.

Dynamisk minnesallokering

- För att ta bort dynamiskt allokerade objekt och frigöra minnet som allokerats för objektet så används *delete* operatorn enligt följande:

```
delete squarePtr;
```

- Vad som händer ovan är att dekonstruktor för objektet som squarePtr pekar på anropas (i detta fallet Square objektet's dekonstruktor) sedan frigörs minnet som är associerat med objektet (här objektet Square). Efter detta kan systemet använda minnet igen för nya ändamål.

Dynamisk minnesallokering

Följande exempel visar hur new operatorn kan användas för att allokera vektorer dynamiskt, vilket vi kommer att använda i inlämningsuppgift nr 6.

- En heltalsvektor med 20 element skapas dynamiskt och tilldelas myArray enligt följande:

```
int *myArray = new int[20];
```

- Vad som händer ovan är att pekaren myArray deklareras och tilldelar den en pekare till första elementet av en dynamiskt allokerad heltalsvektor av storleken 20.

Dynamisk minnesallokering

- För att frigöra minnet för den dynamiskt allokerade vektorn som myArray pekar på så används följande anrop:

```
delete [] myArray;
```

- Notera att anrop ovan avallokerar vektorn som myArray pekar på.
- Om ovannämnda pekare (d v s myArray) skulle peka på en vektor av objekt så skulle först dekonstruktorn för varje objekt i vektorn anropas, innan man frigör vektorn som myArray pekar på.

Placering av klasser i header- och definitionsfil

- I filen Squareclass visas hur man delar upp klasser i headerfil och definitionsfil och sedan anropar man headerfilen från huvudprogrammet, d v s det är i huvudprogrammet som vi sedan använder de redan definierade objekten för att utföra vår uppgift.
- Generellt så gör man följande uppdelning:
  - lägg klassdefinitionen i headerfilen, i vårt fall square.h
  - lägg funktionsdefinitionerna i definitionsfilen, i vårt fall square.cpp
  - skriv inkluderingsdirektiv i definitionsfilen och i huvudprogrammet (här classMyProgram).

Placering av klasser i header- och definitionsfil

```
//classMyProgram.cpp

#include "square.h"

main()
{
   Square square1;
   square1.draw();
   square1.move(1,2);

      . . .
}
```

```
//square.h

class Square
{
 private:
 . . .

 public:
  . . .

} ;
```

```
 //square.cpp

//Def. av medlemsfkn:er
 #include "square.h"

 Square::draw()
 {
    . . .
 }

  . . .
```