TDP004 Objektorienterad Programmering
Fö 2 Objektorientering grunder

Introduktion

- OO är den mest använda programmeringsparadigmen idag, viktigt steg att lära sig och använda OO.
- Klasser är byggstenen i OO, det mesta bygger på dem.
- Vi kommer att introducera grundläggande begrepp inom OO, och återkomma med mera begrepp senare i kursen.
- En ytlig genomgång av minneshantering, mer om detta i en senare föreläsning.

Minneshantering

- Statisk minneshantering, allt minne existerar under hela progammets körning, inget läggs till eller tas bort.
- Dynamisk minneshantering: tar bort och lägger till minne under exekvering. Görs med new.
- Pekar-begreppet:

```
int* a;                 // a är en pekare till en int
int b = 10;
a = &b;                 // a pekar nu på b:s adress

ExampleClass* myClass = NULL;  // Pekaren nollställs
myClass = new ExampleClass();  // Skapa och tilldela
```

- Mer om pekare och minnerhantering senare i kursen.

Klasser

- Klasser är den viktigaste byggstenen inom OO. Mha klasser görs arv, inkapsling, polymorfi, etc…
- Vad är en klass?
  - Tills vidare är det logiskt att tänka på en klass som en representation av ett konkret objekt, tex ett djur, eller en bok etc.
- Ett specifikt ”exemplar” av en klass kallas instans.

Deklaration av klass

I filen ExampleClass.h:

```
class ExampleClass
{
public:                                 // Publik åtkomstdeklaration
    ExampleClass(int defaultID);        // Konstruktor
    ~ExampleClass();                    // Destruktor
    int GetID() const;                  // Medlemsfunktioner.
    float f(float a, float b, float c) const;
private:
    bool SetID(int aID);                // Privat åtkomstdeklaration
    int m_ID;                           // Klassvariabel.
    …
};                                      // Observera ;
```

För det mesta definieras funktionerna i .cc/.cpp-filen.

Deklaration av klass, forts.

- Normalt *deklareras* klassen i en fil med namnet klassnamn.h.
- I filen klassnamn.cc/cpp *definieras* funktionerna.
- Vanligt att man placerar konstruktorn och destruktorn högst upp i .cc-filen.

Konstruktor

- Skapar en instans av klassen.
- Alltid samma namn som klassen.
- Bör anges av programmeraren, *annars skapar kompilatorn en (utan inargument).*
- Anropas med new klassnamn();
- Viktiga medlemsvariabler bör initieras i konstruktorn. Detta kan ske genom tilldelning:

```
ExampleClass::ExampleClass(int defaultID)
{
    m_ID = defaultID;
    …
}
```

Konstruktor, initieringslistor

- I tidigare exempel gjordes tilldelning i konstruktorns funktionskropp.
- Initieringslistor är ett annat sätt att tilldela medlemsvariabler och initiera konstruktorns lokala variabler.
- Syntax:

  klassnamn::klassnamn(…): variabel1(värde1), variabel(värde2), …

Ex:

```
ExampleClass ::ExampleClass(int defaultID) :
    m_ID(defaultID)
{
    …                       // Funktionskropp
}
```

Konstruktor, initieringslistor

- Vaför behövs initieringslistor?
  - Medlemsvariabler som är const och/eller referenser *måste* initieras mha initieringslistor.
  - Ytterligare skäl finns, dessa har med arv att göra.

Multipla konstruktorer

- Möjligt att ha multipla konstruktorer.
- Alla konstruktorer skapar en instans av klassen.
- Kan ta olika inparametrar och göra olika initieringar av medlemsvariabler.
- Tänk på att initiera viktiga medlemsvariabler i *alla* konstruktorer.

Destruktor

- Anropas när man vill ta bort instansen.
- Alltid ~klassnamn.
- Bör anges av programmeraren, *annars skapar kompilatorn en.*
- I destruktorn städar vi upp och lämnar tillbaka minne som klassen har allokerat dynamiskt.
- Anropas med delete instansnamn().

```
ExampleClass::~ExampleClass()
{
   //Städa upp och lämna tillbaka dynamiskt minne, dvs
   minne som har allokerats med new.
}
```

Konstruktor och destruktor

- Funktioner som har speciella uppgifter.
- En pekare till den borttagna instansen kommer inte att peka på något vettigt efter att destruktorn har anropats.
- Tips: sätt pekaren till NULL efteråt.

```
delete myClass;          // Anropa destruktorn.
myClass = NULL;          // Sätt pekaren till NULL
```

Varför sätta variabeln till NULL efter delete?

- Alt 1.

```
delete myClass;
myClass = NULL;
delete myClass;      //OK enligt språkstandarden
```

- Alt 2.

```
delete myClass;
delete myClass;
```

Krasch?!

Åtkomst

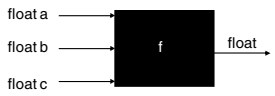
- Med åtkomstdeklarationerna public, protected och private reglerar man åtkomst till klassens data och funktioner för *andra* klasser. Den egna klassen har alltid obegränsad tillgång.
- Public – fritt tillgänglig för läs och skriv.
- Private – ingen tillgänglighet alls.
- Protected – tillgänglig enbart för subklasser. Vi återkommer till dessa i senare föreläsning.
- I class är allt private om inget annat anges.

Åtkomst

- Med tex åtkomstdeklarationen private kan vi skydda data och funktioner så att bara betrodda klasser/funktioner får tillgång.
- Alternativt inte ge någon annan någon tillgång alls.
- *Vi gömmer data som vi inte vill att någon annan ska ha tillgång till samt exakt hur vi implementerar funktionen.*
- Inkapsling (eng. encapsulation).

Inkapsling

- Funktionen float f(float a, float b, float c) gör någon komplicerad beräkning, och vi vill inte att någon annan ska ha tillgång till den data som används, eller veta hur beräkningen görs.



Inkapsling, forts.

- Vi kapslar in beräkningen och får genom detta en ”svart låda” som döljer implementationen och dess data.
- Vad som händer i funktionen/lådan är vår ensak och ingen annan behöver bry sig.
- Vi kan ändra implementationen hur vi vill, så länge vi lämnar funktionsdeklarationen (returvärdet, parameterlistan och funktionsnamnet) intakt.

Friend

- Med friend kan man gå förbi åtkomstdeklarationerna och även få tillgång till *privat* data och funktioner.
- Friend deklareras i den klass som vill ge någon annan klass/funktion tillgång.
- Ex: I någon klass:

```
friend class ExampleClass
friend int ClassA::func(int a);
        //Ger funktionen func i ClassA fri tillgång.
```

- Friend bryter inte mot inkapslingen eftersom det är den givande klassen som bestämmer vilka som ska vara vänner.
- Friend bör trots det användas med försiktighet.

Medlemsfunktioner

- Som de vanliga funktioner vi har använt hittills, men har tillgång till medlemsvariabler.

```
bool  ExampleClass::SetID(int aID)
{
     m_ID = aID;                          //Sätter medlemsvariabel.
     return true;
}

const int ExampleClass::GetID() const  /* En const medlemsfunktion får
                                           inte ändra någon medlemsvaribel */
{
    return m_ID;                          /* Returnerar värdet av en
                                             medlemsvariabel */
}

float ExampleClass::f(float a, float b, float c) const
{
    return a + b + c;
}
```

const-funktioner

- Förra föreläsningen såg vi att inparametrar kunde deklarareras const. Även funktioner kan deklareras const, innebörden av detta är att funktionen inte ändrar någon medlemsvariabel.

```
const int ExampleClass::GetID() const
{
  return m_ID;
}
```

- En const-funktion som får i sin tur enbart anropa andra const-funktioner.
- Bra att använda const så mycket som möjligt, programmeraren får då hjälp av kompilatorn.

Medlemsfunktioner och *this*

- De flesta funktioner har en osynlig parameter, *this*, som parameter till nästan alla funktioner. (undantagen är static-funktioner samt konstruktorer.) Denna pekar på det objekt som funktionen hör till. Vi återkommer till orsaker och konsekvenser av denna i en senare föreläsning.
- Ex:
  MyClass inst =MyClass();
  inst->f(a);
  Översätts till något i stil med:
  MyClass::f(&inst, a);

Medlemsvariabler

- En variabel som hör till en instans av en klass.
- Obegränsad tillgång för alla funktioner i klassen.
- Kan användas för att överföra data mellan funktioner, spara tillståndet hos en instans etc.
- Scope för en medlemsvariabel är så länge instansen existerar.

Varför använda klasser?

- Samla ihop funktioner och den data den använder till en sammanhängande enhet.
- Minska beroendet mellan användare och implementation.
- Förbered för återanvändning.

Användande av klasser

- Att avgöra vilka objekt som är lämpliga att modellera som klasser är ett svårt problem som vi kommer tillbaka till senare i kursen.

Sammanfattning

- Vi har introducerat klassbegreppet.
- Konstruktor, initieringslistor.
- Destruktor.
- Åtkomst: public, protected, private.
- Medlemsvariabler och -funktioner.
- Inkapsling, viktigt begrepp inom OO.
- Vi återkommer till OO senare i kursen.
- Inte trivialt, gå hem och läs, fundera och diskutera!