

TDP004 Objektorienterad Programmering  
Fö 8 Sammanfattning

## Deklarationer/variabler/konstanter

- Variabler kan initieras  
double salary;  
float distance = 10.542;
- Konstanter och referenser måste initieras  
const double pi = 3.14159;  
int i = 10;  
int& j = i;

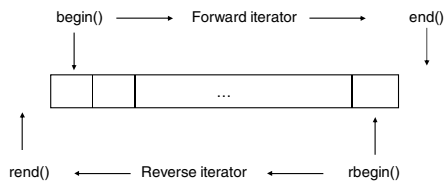
## Typkonverteringar

- Konvertera mellan olika typer med `static_cast<till typ>(variabel)`
- Behövs vid tex  
float a = 7;  
float b = 4;  
int c = a / b; // c = 1 pga trunkering!  
Förväntat resultat fås med:  
int c = static\_cast<float>(a) /  
static\_cast<float>(b);

## STL

- Standard template library.
- Containerar:
  - Sekventiella: vector, list, string.
  - Associativa: map.
- Iteratorer (specialiserad pekare): forward- och reverse-iterator.
- Funktioner: Inbygga funktioner för sortering, sökning, flyttning av element, mm. Alla dessa använder iteratorer.

## Översiktlig bild över iteratorer



## Objektorientering

- **Klasser**
  - Byggstenen inom OO.
  - Åtkomstdeklarationer
    - Public, protected, private.
  - Arv:
    - Public, protected, private.
    - Privat data/funktioner ärvs aldrig.
  - Polymorfi.
  - Inkapsling.
- Struct kan ibland vara ett alternativ till class. I en struct är åtkomsten som default public.

## Exempel på klass

I filen ExampleClass.h:

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

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

## Annat exempel på klass

```
struct Position // Används för att spara en position.
{
    double m_X;
    double m_Y;
    double m_Z;
};

class Character
{
public:
    Character(Position & aSpawnPos);
    ~Character();
    void MoveToPosition(Position & aPosition);
    const Position& GetPosition() const; // OK att returnera en referens till en medlemsvariabel.
                                        // const gör att mottagaren inte kan ändra värdet.
private:
    Position m_Position;
};
```

## 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.
- **I class är allt `private` som default.**

## Arv

- `public`: de ärvda funktionerna/datan behåller den åtkomst de hade i superklassen.
- `protected`: de ärvda funktionerna/datan blir `protected` i subklassen.
- `private`: de ärvda funktionerna/datan blir `private` i subklassen.
- Den ärvande klassen styr hur andra klasser ska komma åt de ärvda funktionerna.

## Konstruktor / destruktör

- Konstruktor skapar objektet
  - Initieringslistor.
    - Behövs för medlemskonstanter och -referenser.
- Destruktor städar upp efter objektet.
- Om programmeraren inte skapar en konstruktor/destruktör skapar kompilatorn dessa.
- Kompilatorgenererad konstruktor använder *inte* initieringslistor.
- Virtuellt destruktör viktigt vid arv.
  - Kompilatorgenererad destruktör är *inte* virtuell.

## Funktioner, forts.

- Deklaration, i h-filen:  
`returvärde funktionsnamn(...);`
- Definition, i cc-filen:  
`returvärde klassnamn::funktionsnamn(...)`

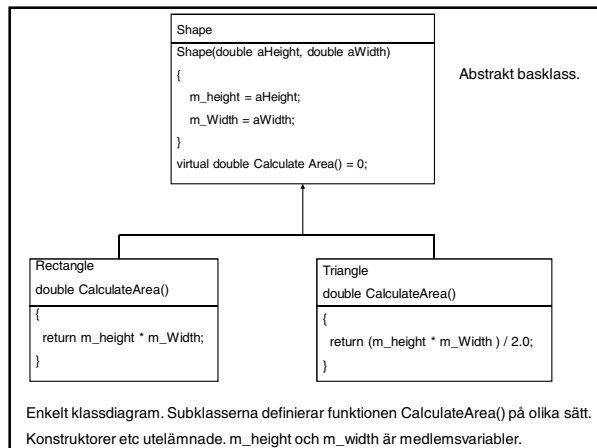
```
{  
    //Implementation  
}
```
- Med `const&` kan vi förhindra kopiering av datan (mha `&`) och hindra att funktionen ändrar på datan (mha `const`).
- Ex:  
`int max(const int& a, const int& b);`
  - Värdet på parametrarna `a` och `b` kan nu inte ändras i funktionen `max`.

## Funktioner

- En funktion gör en sak och bör göra det utan oönskade sidoeffekter.
- Returvärde void eller annan typ, inbyggd eller egendefinerad.
- Parameterlistan är den data som funktionen behöver från den anropande funktionen för att utföra beräkningen.

## Virtuella funktioner

- **virtual** int f(...);
- En virtuell funktion *får* (i C++) implementeras i basklassen, men kan överlagras av basklasserna.
- Virtuella funktioner är grunden för polymorfi. Funktionen implementeras kanske annorlunda i subclasserna än basklassen. Vilken funktion som anropas bestäms vid exekvering (dynamisk/sen bindning).



## Polymorfism

```
vector<Shape*> shapesVector;
Shape* shape1 = new Triangle(...);
Shape* shape2 = new Rectangle(...);
shapesVector.push_back(shape1);
shapesVector.push_back(shape2);
shapesVector.push_back(...);
Ex 1:
for(i = 0; i < shapesVector.size() ; ++i)
{
    cout<< shapesVector[i]->CalculateArea() << " ";
    /* Anropar korrekt CalculateArea(), vi vet
    inte vilken implementation innan anropet sker. */
}
```

```
Ex 2:
shape1->CalculateArea();
//Anropar CalculateArea() i klassen Triangle.
shape2->CalculateArea();
//Anropar CalculateArea() i klassen Rectangle.
```

## Polymorfism, forts.

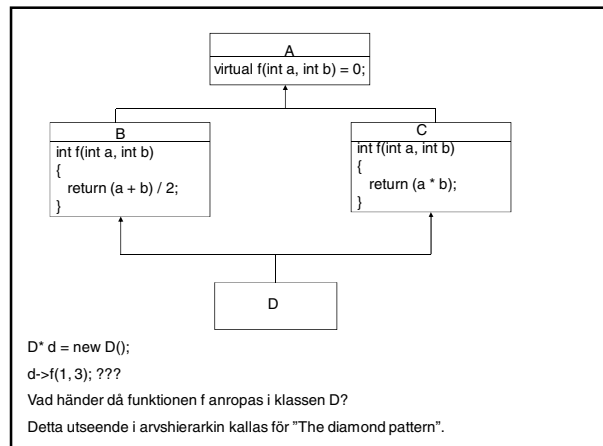
- Vilken funktion som exekveras vid anrop till CalculateArea() avgörs vid exekveringen, sk dynamisk/sen bindning.
- En icke-virtuell funktion har sk statisk/tidig bindning: det bestäms vid kompilering vilken funktion som kommer att anropas.

## Rent virtuella funktioner

- `virtual int f(...) = 0;`
- *Får ej* implementeras i basklassen, *måste* implementeras i subklasserna.
- En klass med minst en rent virtuell funktion blir en abstrakt klass. Det är omöjligt att skapa en instans av en abstrakt klass.

## Multipelt arv

- En klass kan ärva från flera olika basklasser.
- Var försiktig med multipelt arv i allmänhet, pga "diamond pattern".



## Inkapsling

- Data och beräkningar kapslas in i funktioner och implementationen göms.
- Den anropande klassen behöver inte veta några detaljer om implementationen.
- *Privat data och publika interface.*
- Inkapsling är ett viktigt koncept inom OO.

## static

- Static-funktioner hör till klassen, kan anropas även om det inte finns någon instans av klassen. Dessa kan enbart anropa andra funktioner som inte tar this som parameter (ex: Init-create från Fö 9).
- Static-klassvariabler – enbart en upplaga. Initieras utanför alla funktioner. Ex:  
`int ExampleClass::staticVariable = 10;`
- Minne för static medlemsvariabler allokeras enbart en gång.

## Pekare, \*

- Objekt som pekar på ett minne.
- Används bla för dynamisk minneshantering.
- Ex:  
`int* p1;`  
`float* p2;`  
`vector<int*> vp; /* vp är en vector som innehåller pekare till int.*/`  
`ExampleClass* myClass = new ExampleClass();`
- En pekare måste inte, men bör initieras.
- Man kan kontrollera om en pekare är giltig genom jämförelse med NULL.

## Använda pekare

- Om man har en pekare till ett objekt och vill anropa dess funktioner används notationen:  
`variabelnamn->funktionsnamn(parametrar)`

Ex:

```
ExampleClass* myClass = new  
ExampleClass();  
myClass->func(1.1, 3.5, 5.7)
```

- Läser lokal variabel med liknande syntax.

Ex:

```
int id = myClass->m_ID;
```

## Referens &

- Alternativ till pekare.
- "A reference is an alias"
- En referens måste initieras.
- `int i = 1;`
- `int& intRef = i; /*intRef refererar nu till variabeln is värde. */`
- All operationer på en referens är egentligen operationer på den underliggande objektet.
- "A reference is just another name for an object"
- *En referens får inte vara ogiltig.*
- *Det går inte att kontrollera om en referens är giltig.*

## Använda referenser

- Om man har en referens till ett objekt kan man anropa dess funktioner med

`variabelnamn.funktionsnamn(parametrar)`

Ex:

```
ExampleClass myClass2 = &myClass;
```

```
myClass2.func(1.1, 3.5, 5.7);
```

- Läser lokal variabel med liknande syntax.

Ex:

```
int id = myClass.m_ID;
```

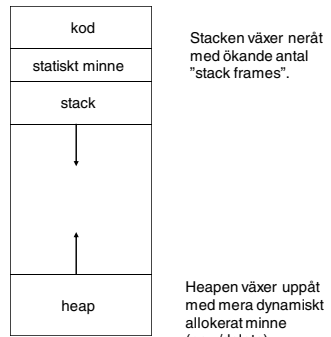
## Minneshantering

- Allokera nytt minne och skapa nya objekt med `new`.  
`MyClass* m_class = new MyClass();`
- Ta bort instansen och återlämna minnet med `delete`.  
`delete m_class;`  
`m_class = NULL; // Behövs inte men ger möjligheten att jämföra med NULL.`
- Vi behöver bara använda `delete` på minne som har allokerats med `new`.

## Minneshantering

- C++ har tre olika sorters minneshantering:
- Statiskt minne
  - Globala och namespace-variabler, static variabler.
- Automatiskt minne/stack
  - Funktionsargument och lokala variabler.
- Free store/heap
  - Minnet allokeras med `new`.
  - Eventuella minnesläckor finns här.

## Översiktlig bild över minnet



## Frågor?

- Resten av tiden till frågor.