Programmering i C++
EDA623
Objektorienterad programutveckling

Objektorienterad programutveckling

Innehåll

- Grundläggande begrepp
- Relationer mellan objekt
- Grafisk representation
- Implementering i C++
- Objektorienterad programmering

Objekt

- Representation av verkligt föremål
- Egenskaper - Attribut
- Operationer - Metoder

Klass

- Beskrivning av en viss typ av objekt
- Ett objekt sägs vara en *instans* av en klass

Association känner till

Bil —— Person (ägare)

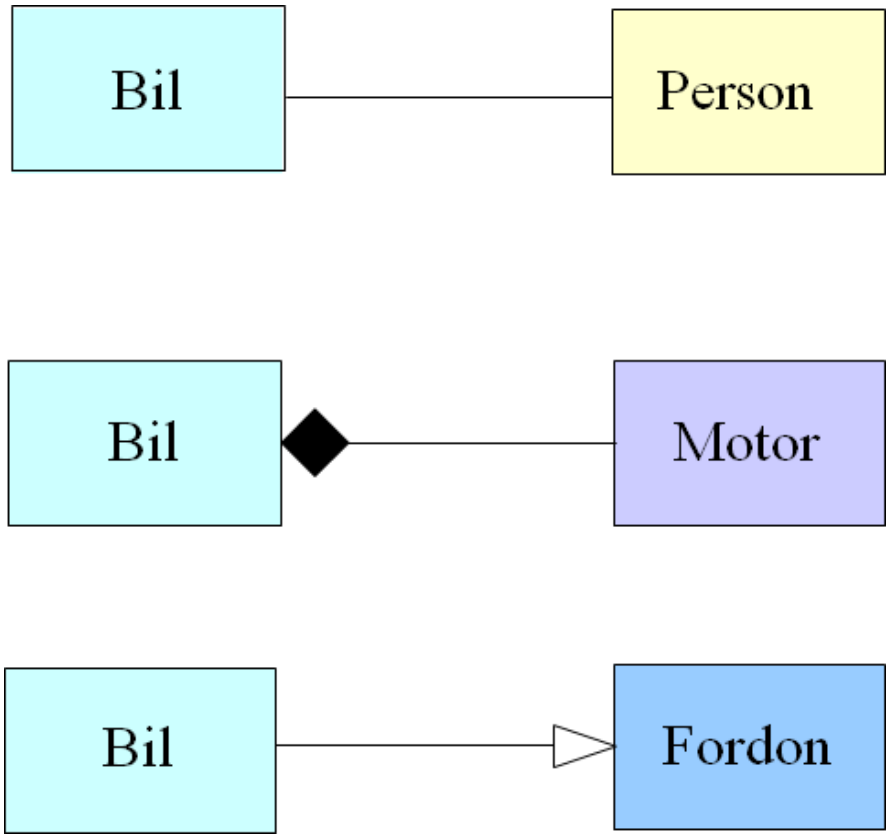
Komposition har

Bil —— Motor

Generalisering är

Bil —— Fordon

UML (Universal Modeling Language)



Association känner till

```
Person *owner;
```

Komposition har

```
Motor m;
```

Generalisering är

```
class Bil : public Fordon {
...
};
```

Inkapsling (information hiding)
Samla alla objektegenskaper på ett ställe i programmet.
Gömma ett objekts detaljer för användaren

Arv Återanvända tidigare objektspecifikationer (=klasser) för att specialisera genom tillägg av egenskaper

Polymorfism *Virtuella* funktioner
Överlagring av funktioner (el. operatorer)
Generiska programenheter (*mallar* (*templates*) i C++)

Polymorfism

Virtuella funktioner Betydelsen hos en funktion bestäms först vid exekvering (*dynamisk el. sen bindning* (*late binding*))

Överlagring av funktioner Betydelsen hos en funktion bestäms av antal och typ av parametrarna. Detta bestäms vid kompileringen (*statisk binding* (*early binding*))

Generiska programenheter (mallar i C++) Enheter där ett antal av de ingående typerna är generella (typerna anges som indata vid användning av enheten).
Ex.: En vektor där elementen är av godtycklig typ (statisk bindning)

Klasser

- Klassdefinitioner
- Placering av klasser
- Konstruktorer
- Destruktorer

Klassdefinitioner

Klass Egendefinierad typ

Objekt Instans av en klass (ett slags variabel)

Typisk klassdefinition:

```
class Klassen {
public: // Den utåt synliga delen
    // Medlemsfunktioner (dekl.)
    // och ev. datamedlemmar
private: // Den dolda delen
    // Datamedlemmar och
    // ev. medlemsfunktioner
}; // Alltid semikolon efter klassdef.
```

Klassdefinitioner

Medlemsfunktioner (<==> metoder i Java)

```
class   Klassen {
public:
    int fun(int, int);  // Deklaration
    // ...
};  // Obs! Semikolon här   ...

//  Definition av funktionen:
int Klassen::fun(int    x, int  y) {
    // ...
} // ... men inget semikolon här!
```

Klassdefinitioner

Inline-definition av en funktion

- Koden läggs ut direkt på det ställe anropet görs (inget hopp till funktionskod på annat ställe)
- Lämpligt endast för mycket enkla funktioner
- Anges genom att skriva funktionsdefinitionen direkt i klassdefinitionen
- Om funktionen läggs utanför klassdefinitionen (via ::-notation) används nyckelordet `inline` framför

Klassdefinitioner

Inline i klassdefinitionen:

```
class Klassen {
  public:
    int getValue() {return value;}
  // ...
};
```

Inline utanför klassdefinitionen:

```
inline int Klassen::getValue() {
    return value;
}
```

Placering av klasser

Klassdefinitionen läggs i en headerfil (.h)

- För att inte riskera att definiera en klass mer än en gång används direktiv:

```
#ifndef KLASSEN_H
#define KLASSEN_H
//...
class Klassen {
//...
};
#endif

// Visual Studio använder istället:
#pragma once
```

Medlemsfunktioner läggs i en ”vanlig” fil (.cpp)

Konstruktorer

Initiering på "vanligt" sätt fungerar ej (före C++11):

```
class Klassen {
public:
    //...
private:
    int value=0; //Ger kompileringsfel (före C++11)
    //...
};
```

Initiering sker istället via en speciell medlemsfunktion kallad *konstruktor* vilken har samma namn som klassen (som i Java)

Konstruktorer

Initiering möjlig i C++11:

```
class Klassen {
public:
    //...
private:
    int value{0}; //Endast C++11
    //...
};
```

Exempel: Bankkonto

Klassdefinition

```
class Bankkonto {
public:
    // Konstruktorer läggs här
    int insattning(int belopp);
    int uttag(int belopp);
    void kontobesked();
private:
    int saldo;
    char kontonr[10];
};
```

Exempel: Bankkonto

Deklaration av konstruktorer

```
// Placeras i klassdefinitionen
Bankkonto();
Bankkonto(char* knr);
Bankkonto(char* knr, int saldo);
```

Exempel: Bankkonto

Definition av konstruktorer

```
Bankkonto::Bankkonto() {
    strcpy(kontonr,"saknas");
    saldo = 0;
}

Bankkonto::Bankkonto(char* knr) : saldo(0) {
    strcpy(kontonr, knr);
}

Bankkonto::Bankkonto(char* knr, int sald) : saldo(sald) {
    strcpy(kontonr, knr);
}
```

Exempel: Bankkonto

Definition av medlemsfunktioner

```
int Bankkonto::insattning(int belopp)
{
    if (belopp > 0) {
        saldo += belopp;
        return 1;
    }
    else
        return 0;
}
```

Exempel: Bankkonto

Definition av medlemsfunktioner

```
int Bankkonto::uttag(int belopp)
{
    if (belopp > 0 && belopp < saldo) {
        saldo -= belopp;
        return 1;
    }
    else
        return 0;
}
```

Exempel: Bankkonto

Definition av medlemsfunktioner

```
void Bankkonto::kontobesked() {
    cout << "KONTOBESKED" << endl;
    cout << "Kontonr: " << kontonr <<endl;
    cout << "Saldo: " << saldo << endl << endl;
}
```

Exempel: Bankkonto

Hantering av objekt

```
Bankkonto konto("1-234-567"), *pkonto;

konto.insattning(20000);
konto.kontobesked();
pkonto = new Bankkonto("3-425-167");
pkonto->insattning(25000);
konto.uttag(1200);
pkonto->kontobesked();
konto.kontobesked();
```

Exempel: Bankkonto

Hantering av objekt – Utskrifter

```
KONTOBESKED
Kontonr: 1-234-567
Saldo : 20000

KONTOBESKED
Kontonr: 3-425-167
Saldo : 25000

KONTOBESKED
Kontonr: 1-234-567
Saldo : 18800
```

Konstruktorer

Defaultkonstruktor

- Definieras automatiskt om det inte finns någon konstruktor alls definierad
- Svarar mot en konstruktor utan parametrar
- Får ha parametrar med defaultvärden

```
// I klassdekl.
Bankkonto(char* knr="?", int saldo=0);
// fungerar även som defaultkonstruktor

// ...Anrop av defaultkonstruktorn ...
Bankkonto mitt_konto;   // ... sker här

Bankkonto *pkonto;
pkonto = new Bankkonto; // ... och här
```

Konstruktorer

Kopieringskonstruktor

- Anropas vid t.ex. initiering av ett objekt då detta ska bli en kopia av ett annat objekt av samma typ
- Anropas *inte* vid tilldelning
- Kan definieras, men i annat fall definieras automatiskt en standardvariant av kopieringskonstruktor

```
Bankkonto konto1("5-346-712", 12500);

Bankkonto konto2 = konto1; // Här anropas
                           // kopieringskonstruktorn ...
Bankkonto konto3;
konto3 = konto1;           // ... men inte här
```

Konstruktorer

Typomvandlingskonstruktor

- Anropas då konstruktorn anropas med uttryck av annan typ

```
class KomplextTal {
public:
    KomplextTal():re(1),im(0) {}          //Default
    KomplextTal(const KomplextTal& c);  //Kopiering
    KomplextTal(float x):re(x),im(0) {} //Typomv.
    //...
private:
    float re, im;
}
```

Destruktorer

När minnesutrymme allokeras dynamiskt inne i en konstruktor (med `new`) måste detta avallokeras igen så småningom. Det finns en "motsats" till konstruktor som anropas vid avallokering (t.ex. vid `delete`). Denna kallas *destruktor* (namnet inleds med tilde ("~"))

```
~Bankkonto(); // Dekl. av destruktor
```

Exempel: Vektor

Klassdefinition

```
class Vektor {
public:
    Vektor(int lng = 10);     // Även defaultkonstruktor
    Vektor(const Vektor& v); // Kopiering!
    ~Vektor(); // Destruktor
    int lngd() {return ant;} // Inline!
    int get_elem(int ind);
    void set_elem(int ind, int val);
private:
    int *p; // Själva vektorn
    int ant;// Antalet element
};
```

Exempel: Vektor

Konstruktorer

```
Vektor::Vektor(int lng) : ant(lng) {
    assert(ant>=0);      // Indexkontroll
    p = new int[ant];    // Dynamisk allokering
}

//Kopieringskonstruktor
Vektor::Vektor(const Vektor& v) : ant(v.ant) {
    p = new int[ant];
    for (int i=0; i<ant; i++)
        p[i] = v.p[i];
}
```

Exempel: Vektor

Övriga medlemsfunktioner

```
int Vektor::get_elem(int ind) {
    assert(ind>=0 && ind<ant);
    return p[ind];
}
void Vektor::set_elem(int ind, int val) {
    assert(ind>=0 && ind<ant);
    p[ind] = val;
}
```

Exempel: Vektor

Destruktorn

```
//Deallokera allt dynamiskt minnesutrymme
Vektor::~Vektor() {
    delete [] p;
}

// Passar egentligen bättre
// med inline i klassdeklarationen:
// ...
   ~Vektor() {delete [] p;}
// ...
```

Exempel: Vektor

Skapande och destruktion i ett block

```
{
    Vektor v; // Konstruktor anropas
    Vektor *pv;
    pv = new Vektor; // Konstruktoranrop
    // ...
    delete pv; // Destruktor anropas
}   // Destruktor anropas även här
    // då vi lämnar blocket
    // för att ta kål på v (lokal var.)
```