Förord

Detta är del II av ett material avsett att användas på en inledande kurs i programmering med C++. Denna del tar upp klasser, pekare och enkel filhantering.

Materialet ger inte alls en fullständig täckning av språket och den som vill fördjupa sig måste skaffa en mer fullständig bok.

Vissa avsnitt har märkts med asterisk (*) vilket anger att materialet är perifert och kan överhoppas åtminstone på vid första genomläsningen.

En del material som funnits i tidigare versioner har utelämnats bl.a. mallar, operatoröverlagring och exemplen med polynomhantering och klassen `bignum`.

Den här föreliggande versionen skall betraktas som en ”prerelease”. Vi tar tacksamt emot felrapporter och andra synpunkter via email till tom@tdb.uu.se.

Innehåll

Kapitel 9

Introduktion till klassbegreppet

Vi har i de föregående kapitlen diskuterat ett antal grundläggande begrepp såsom variabler av olika datatyper (`int`, `double` mm), operationer på dessa (aritmetik, inläsning, utskrift mm.) Vi har också fört in begreppet funktion som används för att beskriva och namnge mera komplexa operation med hjälp av de grundläggande operationerna. I detta kapitel skall vi ta upp klassbegreppet med vars hjälp man kan bygga nya datatyper utifrån de begrepp (variabler, arrayer, funktioner etc.) som vi tidigare studerat. De nya typerna kan vara tillämpningsorienterade som t ex bankkonton, fartyg eller schackpjäser. De kan också vara avsedda att komplettera programmeringsspråket med begrepp som är användbara i många olika tillämpningar (t ex komplexa tal som saknas i C++) eller ersätta begrepp som man tycker är ofullkomliga i språket (t ex arrayer).

En datatyp karakteriseras dels av vilka värden den kan anta och dels av vilka operationer som man skall kunna utföra på den. Vi använder variabler för att representera värdena och funktioner för att beskriva operationerna. Det ”paketeras” i en *klass*. En viktig egenskap, som inte är given i alla programmeringsspråk, är båda data- och operationbeskrivningar som hör till ett begrepp hålls samman.

9.1 Klassdefinitioner

Vi skall exemplifiera klassbegreppet genom att utveckla ett (mycket enkelt) banksystem med bankkonton där man kan sätta in och ta ut pengar. Vi börjar med att definiera en ny datatyp som vi kallar `Account` som skall representera ett konto i banken. Det absolut minsta vi måste hålla reda på är hur mycket pengar det finns på kontot och vems det är. Det gör vi genom att för varje konto lagra ett *saldo* av typen `double` och innehavarens *namn* av typen `string`. En sådan datatyp kan definieras på följande sätt:

```
class Account {
  string name;     // Innehavarens namn
  double balance;  // Saldo
};
```

Observera att detta bara är en beskrivning av vad som skall finnas med i ett konto. Inga konton har skapats än. För att skapa ett konto deklarerar variabler av typen `Account`:

```
Account x;
Account customers[1000];
```

Variabeln `x` lagrar ett och arrayen `customers` tusen konton där vardera har plats för namn och saldo. Dock har inga namn eller saldon ännu lagrats.

Nu har vi talat om vilken typ av data som klassen `Account` lagrar. För att det skall bli användbart behöver vi också ange vad vi skall kunna göra med typen. Detta görs genom att definiera *operationer* som ofta kallas *metoder*. Exempel på typiska operationer i detta fall är att sätta in och ta ut pengar. Sådan definitioner ser ut som funktionsdefinitioner men är placerade inuti klassdefinitionen:

```
class Account {
 protected:     // Skyddade data
  string name;        // Innehavarens namn
  double balance;     // Saldo

 public:        // Allmänt tillgängliga metoder
  void deposit( double amount ) {
    balance += amount;
  }

  void withdraw( double amount ) {
    balance -= amount;
  }
};
```

Orden `protected` och `public` specificerar åtkomsträttigheter. Saker som är angivna efter `protected:` får endast användas av ”egna” metoder (i detta fall `deposit` och `withdraw`) medan sådant som kommer efter `public:` är tillgängliga även för utomstående. Mer om detta senare.

Än så länge saknar vi möjligheter att lagra namn och startsaldo samt möjligheter att få ut någon information. För att råda bot på detta skapar vi ytterligare två metoder: `input` för att läsa in namn och nollställa saldot samt `print` för att skriva ut informationen om ett konto. Vi lägger också till en `main`-funktion för att testa klassen och dess metoder och får vårt första programexempel med en egendefinierad klass.

Exempel 9.1 Första version av Accountklassen

```
// Filnamn : .../Account0/Account.cc
// Första versionen av ett bankkonto.
#include <iostream>
#include <string>
using namespace std;

// Klassdefinition för hantering av bankkonton.
```

```
class Account {

public:       // Publika metoder
  void input() {
    cout << "Ge namn : ";
    getline(cin, name);
    balance = 0;
  }

  void deposit( double amount ) { balance += amount; }

  void withdraw( double amount ) { balance -= amount; }

  void print() {
    cout << name << " har " << balance
         << " på sitt konto " << endl;
  }

protected:    // Skyddade datafält
  string name;
  double balance;
}; // Här är klassdefinitionen slut


// Ett litet testprogram

int main() {
  // Deklaration av 2 objekt av klassen Account
  Account lisa;
  Account olle;

  lisa.input();
  olle.input();
  lisa.print();
  olle.print();
  lisa.deposit( 500 );
  lisa.withdraw( 350 );
  lisa.print();
  return 0;
}
```

I `main` kan vi se hur vi kommer åt det som finns inuti klassen. Vi i använder en *punktnotation* dvs ett variabelnamn av någon klasstyp följt av en punkt följt av namnet på något inuti klassen. Om det är en metod skall uttrycket följas av metodens parametar på samma sätt som funktionsanrop. Man kan säga att punktnotationen är en slags genitivkonstruktion: ”lisas input”, ”olles print” etc. Det är exakt samma sätt som vi tidigare punktnotationen på `string`-objekt.

Ofta (oftast) vill man separera *implementationen* av klassens metoder från klass-

beskrivningen. Det skulle då få följande utseende:

Exempel 9.2 Account-klassen med separerade implementationer

```
// Filnamn : .../Account1/Account.cc
//
// Implementation av metoderna utanför klassdefinitionen

#include <iostream>
#include <string>
using namespace std;

// Klassdefinition för bankkonton.

class Account {
 public:            // Deklaration av publika metoder
  void input();
  void deposit( double amount );
  void withdraw( double amount );
  void print();

 protected:         // Skyddade datafält
  string name;
  double balance;
}; // Slut på klassdefinition

// Implementationer (definitioner) av metoderna. Observera
// att varje metod föregås av klassnamnet följt av ::

void Account::input() {
  cout << "Ge namn : ";
  getline(cin, name);
  balance = 0;
}

void Account::deposit( double amount ) { balance += amount; }

void Account::withdraw( double amount ) {balance -= amount; }

void Account::print() {
  cout << name << " har " << balance
       << " på sitt konto " << endl;
}

// Testprogram

int main() {
  // Deklaration av 2 objekt av klassen Account
  Account lisa;
  Account olle;

  lisa.input();
```

```
  olle.input();
  lisa.print();
  olle.print();
  lisa.deposit( 500 );
  lisa.withdraw( 350 );
  lisa.print();

  return 0;
}
```

Fördelen med detta är att själva klassdefinitionen blir kortare och därmed överskådligare. Man kan utifrån den avgöra hur den skall användas utan att behöva blanda in implementationsdetaljer.

Det som står i klassdefinition som är precis det som kompilatorn behöver för att kunna hantera kod som använder klassen. Därför placerar man vanligen denna definition på en egen fil som brukar kallas *deklarationsfil*, *header-fil* eller *inkluderingsfil*. Observera att terminologin är något oegentlig: Deklarationsfilen innehåller *klassdefinitionen* (utom definitioner av metoderna).

Implementationerna av metoderna placeras också på en egen fil som kan kompileras separat. Alla programdelar som sedan använder klassen skall inkludera (med `#include`) *deklarationsfilen*.

Vårt lilla program med kontoklassen uppdelas således på tre filer: en `Account.h` med klassdefinitionen, en `Account.cc` med implementationerna av klassmetoderna och en med användningen vilket i detta fall bara är det lilla huvudprogrammet som testar metoderna. Låt oss kalla denna fil för `AccountTest.cc`. Deklarationsfilen har följande innehåll

Exempel 9.3 Deklarationsfil för Account-klassen

```
// Filnamn : .../Account2/Account.h

#ifndef __ACCOUNT__
#define __ACCOUNT__

#include <string>
using namespace std;

// Klassdefinition för hantering av bankkonton.

class Account {
 public:            // Publika metoder
  void input();
  void deposit( double amount );
  void withdraw( double amount );
  void print();
```

```
 protected:          // Skyddade datafält
  string name;
  double balance;
};
#endif
```

Det har tillkommit tre lite mystiska rader: `#ifndef`, `#define` och `#endif`. Dessa är till för att undvika ”dubbeldeklarationer” när man har många `include`-filer som inkluderar varandra. Egentligen behövs de inte nu men det är en god vana att alltid ha med dem. Det som står efter `#ifndef` och `#define` är en symbol som man vanligen väljer lika med filnamnet i versaler omgivet av ett eller två understrykningstecken (_)

Betydelse är ungefär följande: Om symbolen `__ACCOUNT__` inte är definierad så fortsätter vi direkt och definierar symbolen annars hoppas all kod över fram till matchande `#endif`. Det betyder att första gången filen läses under en kompilering kommer koden att gås igenom men alla påföljande gånger hoppas den över.

Filen med implementationer av metoderna i klassen `Account` får följande utseende.

Exempel 9.4 Implementationsfil för Account-klassen

```
// Filnamn : .../Account2/Account.cc
// Implementation av metoderna i klassen Account

#include <iostream>
#include <string>
#include "Account.h"
using namespace std;

void Account::input() {
  cout << "Ge namn : ";
  getline(cin, name);
  balance = 0;
}

void Account::deposit( double amount ) { balance += amount; }

void Account::withdraw( double amount ) {balance -= amount; }

void Account::print() {
  cout << name << " har " << balance
       << " på sitt konto " << endl;
}
```

Observera att vi här inkluderar deklarationsfilen som vi visade ovan. Filnam-

net i `include`-satsen omges av citationstecken i stället för `< >` som vi tidigare använt. Detta anger att filen skall sökas i aktuell katalog i stället för bland de systemdefinierade deklarationsfilerna.

Filen med användning av `Account`-klassen får följande utseende.

Exempel 9.5 Fil som använder Account-klassen

```
// Filnamn .../Account2/AccountTest.cc
// Testprogram för klassen Account

#include <iostream>
#include "Account.h"

int main() {
  // Deklaration av 2 objekt av klassen Account
  Account lisa;
  Account olle;

  lisa.input();
  olle.input();
  lisa.print();
  olle.print();
  lisa.deposit( 500 );
  lisa.withdraw( 350 );
  lisa.print();
  return 0;
}
```

Även här inkluderar vi deklarationsfilen för klassen. Vi skulle också kunna inkludera implementationsfilen om vi alltid vill kompilera om allting. I detta fall är ju filerna så små så det skulle vara enklast att göra så men i stora program vill man bara kompilera om de delar som har ändrats. Hur man rent praktiskt gör detta beror på vilket operativsystem och den programmeringsmiljö man använder och beskrivs inte i detta kompendium.

9.2 Lite terminologi

Sammanfattningsvis är en klass en samling data och funktioner (attribut och metoder). Alla egenskaper för ett konto har *inkapslats* i en klass, vilket på engelska kallas för *encapsulation* och är objektorienteringens första princip.

I satsen

```
Account myAccount;
```

deklarerar vi en variabel av typen `Account`. I den objektorienterade världen säger man att vi har deklarerat ett *objekt* av typen `Account`. Om vi vill vara

riktigt ”inne” säger vi att vi har `instansierat` klassen `Account`. I denna sats kan vi inte se att `Account` är en egendefinierad klass, `Account` kunde lika gärna vara en inbyggd typ som `int`.

Ett anrop till en medlemsmetod kallas ibland för att skickar ett *meddelande* till ett objekt. Huvudprogrammet skickar meddelandet `input` till objektet `lisa`.

9.3 Åtkomstskydd och attributåtkomst

Som vi sett kan komponenter i en klass anges som `protected` eller `public`. Den första (`protected`) innebär att det inte är tillåtet att referera komponenten utifrån dvs med punktnotationen medan den andra (`public`) tillåter det. Det är alltså inte tillåtet att skriva t ex

```
Account myAccount;
myAccount.balance += 100;
```

för att sätta in 100 kronor på ett konto utan vi måste använda de `public`-deklarerade metoderna `deposit` och `withdraw` för att ändra på saldot.

Det är en god programmeringsstil att göra precis som vi gjort dvs att ha datafälten skyddade och tillhandahålla metoder för att manipulera dessa. Fördelen är att man kan använda klassen utan att veta hur den är representerad. Det räcker med att känna till vilka operationer som finns. Man kan t ex ändra representation utan att behöva ändra i de program som använder klassen.

Man kan också vilja förändra metoderna som t ex låta dem hålla reda på vilka transaktioner som gjorts och när de gjorts. Så länge metodanropen ser likadana ut behöver de delar av programmet som använder klassen inte förändras.

Det finns ytterligare en skyddsnivå som heter `private`. Det är också den som gäller om man inte anger någon skyddsnivå. Tills vidare kan vi anse den som liktydig med `protected`.

Att datafälten (attributen) är skyddade gör ju att vi inte kommer åt dem utifrån. Som `Account`-klassen nu är definierad så går det inte att utifrån (från andra programdelar) avläsa saldot eller namnet på kontoinnehavaren — det enda vi kan göra är att öka eller minska saldot samt att skriva ut informationen. För att råda bot på detta brukar man förse klasser med metoder för att avläsa respektive ändra datafälten. Metoder för att avläsa kallas *selektorer* eller *get-metoder* medan metoder för att ändra kallas för *mutatorer* eller *set-metoder*. Metoderna `withdraw` och `deposit` är exempel på mutatorer.

För att vi skall kunna använda klassen `Account` fortsättningsvis skall vi förse den med två selektorer för att avläsa namn- respektive saldofältet. Vi ger dem de naturliga namnen `getName` och `getBalance`. I deklarationsfilen `Account.h` tillfogar vi då raderna

```
double getBalance();
string getName();
```

bland de publika metoderna. Implementationerna i `Account.cc` blir mycket enkla:

```
double Account::getBalance() {
  return balance;
}

string Account::getName() {
  return name;
}
```

När detta är gjort kan man t ex skriva följande kod:

```
Account x, y;
...
if ( x.getBalance() > y.getBalance() )
   cout << x.getName() << " har mer pengar än " << y.getName()
        << endl;
```

9.4 Initiering av objekt

När vi deklarerar variabler har vi tidigare sagt att det är bra att ge dem ett initialvärde genom en tilldelning vid själva deklarationen. På så sätt undviker man att får konstiga ”odefinierade” värden. Man kan i regel inte göra på samma sätt med klassvariabler. Följande deklaration

```
Account myAccount = { "Arthur Dent",
                      1000
                    };
```

är endast korrekt om alla datafälten är publika — dvs precis det som vi avrått ifrån.

Av den anledningen finns det *initieringsmetoder* till klasser. Dessa kallas för *konstruktorer*. Som namnet indikerar är de till för att konstruera ett objekt. Det finns alltid en så kallad *defaultkonstruktor* (alternativt namn är *standardkonstruktor*) utan parametrar som inte gör något. Denna kan man överlagra dvs definiera en egen som då gäller.

Vidare kan en eller flera konstruktorer definieras med parametrar.

Motsatsen till konstruktor heter *destruktor* vars uppgift är att ”förstöra” objektet då det skall försvinna. En destruktor har aldrig några parametrar. Det finns alltid en default destruktor som inte gör någonting. Destruktorers viktigaste användning är i samband med pekare och dynamiskt minne — begrepp som vi ännu inte tagit upp.

Låt oss se på vårt `Account`-exempel med två konstruktorer och en omdefinierad destruktor. Först deklarationsfilen:

Exempel 9.6 Deklarationsfil med konstruktorer och destruktor

```
// Filnamn : .../Account4/Account.h

#ifndef __ACCOUNT__
#define __ACCOUNT__

#include <string>
using namespace std;

// Klassdefinition för hantering av bankkonton.

class Account {

 public:
  Account();                        // Standardkonstruktor.
  Account( string n, double b );    // Konstruktor med parametrar
  ~Account();                       // Destruktor.

  void input();                     // Läs in data.
  void deposit( double amount );    // Sätt in pengar.
  void withdraw( double amount );   // Ta ut pengar.
  void print();                     // Skriv saldobesked.

 protected:
  string name;
  double balance;
};
#endif
```

Observera att destruktorn heter `~Account` och de två konstruktorerna `Account`. Namnen bestäms alltså av namnet på klassen.

Låt oss gå vidare genom att titta på dessa metoder (vi utelämnar de övriga medlemsfunktionerna av utrymmesskäl):

Exempel 9.7 Implementaion av konstruktorer och destruktorer

```
// Filnamn : .../Account4/Account.cc

#include <iostream>
#include <string>
#include "Account.h"
using namespace std;


Account::Account() {
  balance = 0;
  name    = "Arne Anonym";
  cout << "Hello default: " << name
       << "  "  << balance << endl;
}
```

```
Account::Account(string n, double b) {
  balance = b;
  name    = n;
  cout << "Hello : " << name
       << "  "  << balance << endl;
}

Account::~Account() {
  cout << "Bye-bye " << name
       << " med saldot: "  << balance << endl;
}
```

Vi börjar med att titta på `Account()`. I fälten `name` läggs ett ”default-namn” in och saldot nollställs.

Den sista `cout`-satsen behövs egentligen inte alls. Vi har satt in den för att vi senare skall kunna se hur konstruktorer och destruktorer anropas.

I `Account( string n, double b )` kopierar vi strängen från parametern `n` till datafältet `name` samt värdet från parametern `b` till datafältet `balance`

Destruktorn `~Account()` är helt onödig i detta exempel. Vi har tagit med den för att man skall se hur den ser ut. Utskriften gör att vi ser att den verkligen anropas.

Ett huvudprogram kan se ut som:

Exempel 9.8 Program som testar konstruktorer och destruktor

```
// Filnamn : ...Account4/AccountTest.cc

//   Ett litet huvudprogram som testar
//   konstruktorerna och metoderna.

#include <iostream>
#include <string>
#include "Account.h"
using namespace std;

int main() {
  string name = "Ford Prefect";  // Lokal variabel i main
  Account nobody;                // Standardkonstruktor anropas
  Account somebody(name, 100);   // Överlagrad konstruktor anropas

  nobody.print();
  somebody.print();
  nobody.input();
  nobody.print();

  return 0;
```

```
}
```

Så här ser det ut om vi kör programmet

```
Hello default: Arne Anonym 0
Hello : Ford Prefect 100
Arne Anonym har 0 på sitt konto
Ford Prefect har 100 på sitt konto
Ge namn : Prostetnic Vogon Jeltz
Prostetnic Vogon Jeltz har 0 på sitt konto
Bye-bye Ford Prefect med saldot: 100
Bye-bye Prostetnic Vogon Jeltz med saldot: 0
```

I huvudprogrammet `AccountTest.cc` finns det inga utskrifter av typen ”Hello ..” och ”Bye-bye”. Inte heller finns det några explicita anrop till destruktorn eller till konstruktorn `Account()`. Dessa anropas automatiskt av C++.

9.5 Arrayer med objekt och klasser i klasser

En klass kan innehålla andra objekt eller arrayer av objekt. Låt oss nu ta ett exempel med en bank. Vi låter banken vara en klass innehållande en array med konton samt en indikator på hur många konton som finns d v s hur många platser i arrayen som faktiskt utnyttjas.

Exempel 9.9 En bank-klass

```
// Filnamn : .../Bank1/Bank.cc

#include <iostream>
#include "Account.h"
using namespace std;

const int MAX_NR_ACC = 100; // Arraystorlek. Maximalt antal konton

class Bank {
public:
  Bank();              // Standardkonstruktor

  void newAccount();
  void printAllAccounts();

protected:
  string name;         // Banknamn
  int noOfAcc;         // Antal befintliga konton. Fungerar även
                       // som index till första fria position.
  Account accounts[MAX_NR_ACC]; // Array med konton
};
```

```
Bank::Bank() {
  noOfAcc = 0;          // Inga konton från början
  name    = "TDBank";
}

void Bank::newAccount() {
  // Kontroll om fler konton får plats
  if (noOfAcc >= MAX_NR_ACC)
    cout << "Inga fler konton får plats!" << endl;
  else {
    accounts[noOfAcc].input();
    noOfAcc++;
  }
}

void Bank::printAllAccounts() {
  cout << endl << "Följande konton finns i "
       << name << " : \n";

  for (int i=0; i<noOfAcc; i++)
    accounts[i].print();
}

// Ett litet testhuvudprogram
int main() {
  int nr;
  Bank myBank;

  myBank.printAllAccounts();
  cout << "Hur många konton ska läggas in : ";
  cin >> nr;
  cin.get();   // Läs bort returtecknet

  for(int i=1; i<=nr; i++)
    myBank.newAccount();
  myBank.printAllAccounts();

  return 0;
}
```

Vi har denna gång valt att lägga hela programmet på en fil men vi utnyttjar klassen `Account` som vi redan skapat genom att inkludera dess deklarationsfil. (Vi har tagit bort destruktorn som inte fyller någon funktion samt utskrifterna från konstruktorerna.)

Konstruktorn `Bank()` sätter ett namn på banken samt initierar antalet konton till noll.

Metoden `newAccount` kontrollerar att det finns plats för ytterligare ett konto och läser in den genom att anropa `input`-metoden i `Account`-klassen.

Metoden `printAllAccounts` itererar över arrayen och skriver ut alla befintliga konton genom anrop till `Accounts` `print`-metod.

Ett viktigt skäl till att vi skapade en `Account`-klass var att vi skulle kunna skapa många konton. Detta gäller dock knappast för klassen `Bank`. Att vi ändå gjort `Bank` som en klass beror på att det är ett bra sätt att samla ihop allt (data och operationer) som hör till en bank.

För att få lite vana att hantera klasser och objekt skall vi nu utvidga klassen `Bank` med diverse metoder. Vi tänker oss att programmet skall användas av en ”kundmottagare” i kassan. Den enda operation vi hittills kan göra är att skapa nya konton samt att lista alla konton. Därutöver vill vi kunna sätta in och ta ut pengar för en angiven kund. Vi vill också kunna fråga om saldo.

En central funktion för dessa operationer är att leta upp en kund med angivet namn dvs att hitta kundens index i kontoarrayen. För detta ändamål skriver vi en hjälpmetod `searchCustomer` Vi låter också banksystemet han en ”aktuell kund” dvs ett index i arrayen som anger den kund vi just nu arbetar med.

Vi behöver också någon kommandotolk som frågar vad som skall göras och anropar lämplig metod för att få det utfört. Denna metod kallar vi `run`

Det leder till följande utvidgade deklaration av klassen `Bank`

```
class Bank {
public:
  Bank();           // Standardkonstruktor

  // Metoder för olika bankoperationer
  void newAccount();       // Skapa nytt konto
  void setCurAccount();    // Sätt aktuellt konto.
  void printCurAccount();  // Skriv aktuellt konto
  void deposit();          // Insättning på aktuellt konto
  void withdraw();         // Uttag från aktuellt konto
  void printAllAccounts(); // Skriv alla konton

  // Menystyrning av bankoperationer
  void run();

  // Hjälpmetoder
  int searchCustomer(string name);

protected:
  string name;                  // Banknamn
  int noOfAcc;                  // Antal befintliga konton. Fungerar även
                                // som index till första fria position.
  Account accounts[MAX_NR_ACC]; // Array med konton
  int curAccount;               // Index för aktuellt konto
};
```

Innan vi tittar på koden skall vi se hur en del av en körning kan se ut.

```
====================
Vad vill du göra?
1 Skapa ny kund
2 Hitta kund
3 Ge saldobesked
4 Sätta in
5 Ta ut
6 Lista alla konton
7 Sluta
Ditt val: 1
====================
* Skapa nytt konto *
Ge namn : Albus Dumbledore
====================
Vad vill du göra?
1 Skapa ny kund
2 Hitta kund
3 Ge saldobesked
4 Sätta in
5 Ta ut
6 Lista alla konton
7 Sluta
Ditt val: 4
====================
* Insättning *
Kund : Albus Dumbledore
Belopp: 5000
Albus Dumbledore har 5000 på sitt konto
====================
```

Vi börjar med att titta på `run`-metoden som kommunicerar med användaren och fördelar arbetet med hjälp av en meny.

```
void Bank::run() {
  int ans;

  do {
    cout << "====================\n";
    cout << "Vad vill du göra?\n";
    cout << "1  Skapa ny kund\n"
         << "2  Hitta kund\n"
         << "3  Ge saldobesked\n"
         << "4  Sätta in\n"
         << "5  Ta ut\n"
         << "6  Lista alla konton\n"
         << "7  Sluta" << endl;
    cout << "Ditt val: ";
    cin >> ans;
    cin.get();
    cout << "====================" << endl;
    if ( ans==1 )       newAccount();
    else if (ans==2)    setCurAccount();
    else if (ans==3)    printCurAccount();
```

```
    else if (ans==4)    deposit();
    else if (ans==5)    withdraw();
    else if (ans==6)    printAllAccounts();
    else if (ans==7)    ;
    else {
      cout << "Felaktigt kommando. Försök igen" << endl;
    }
  } while (ans!=7);
}
```

Denna funktion skriver alltså ut de alternativ som finns och läser in ett värde som det använder för att styra vilken funktion som skall anropas. Eftersom `run` ligger i klassen `Bank` så kan den anropa de klassens övriga funktioner utan punktnotation. (Observera att felhanteringen inte är fullständig — om användaren svarar med något annat än ett tal så blir det helt fel.)

Metoden `printCurAccount()` är mycket enkel. Den använder index för aktuellt konto och anropar dess `print`-metod:

```
void Bank::printCurAccount() {
  accounts[curAccount].print();
}
```

Att den ändå förtjänar att vara en egen funktion beror dels på en önska om enhetlighet — det skall finnas en funktion för varje operation vi vill utföra. Dessutom kan man vilja lägga till en kontroll på att `curAccount` innehåller ett vettigt värde. Det borde t ex bli fel om vi anropar metoden innan några konton lagts in.

Metoderna för insättning och uttag är snarlika och bygger motsvarande metod i `Account`-klassen:

```
void Bank::deposit() {
  double amount;
  cout << "* Insättning *" <<endl;
  cout << "Kund  : " << accounts[curAccount].getName() << endl;
  cout << "Belopp: ";
  cin >> amount;
  cin.get();
  accounts[curAccount].deposit(amount);
  accounts[curAccount].print();
}

void Bank::withdraw() {
  double amount;
  cout << "* Uttag *" <<endl;
  cout << "Kund  : " << accounts[curAccount].getName() << endl;
  cout << "Belopp: ";
  cin >> amount;
  cin.get();
  accounts[curAccount].withdraw(amount);
  accounts[curAccount].print();
}
```

Observera att de kan ha samma namn som motsvarande metoderna i `Account`-klassen. Systemet håller reda på till vilka klasser metoderna hör. Båda metoderna utför operationen på det aktuella kontot. Även här borde man kontrollera att detta är vettigt satt.

Funktionen för att lokalisera ett konto med angivet namn har följande utseende:

```
void Bank::setCurAccount() {
  string name;
  cout << "* Sök konto *" << endl;
  cout << "Namn: ";
  getline(cin, name);
  int i = searchCustomer(name);
  if ( i<0 )
    cout << "Kunden finns ej" << endl;
  else {
    curAccount = i;
    accounts[curAccount].print();
  }
}
```

Funktionen använder hjälpfunktionen `searchCustomer` som letar efter ett konto med angivet namn och returnerar dess index eller -1 om namnet inte hittades:

```
int Bank::searchCustomer(string name) {
  // Hjälpmetod som söker efter konto med angiven ägare
  for (int i=0; i<noOfAcc; i++)
    if ( accounts[i].getName()==name )
      return i;     // Funnen
  return -1;        // Ej funnen
}
```

Den mest komplicerade operationen är att skapa nya konton vilket beror på att två olika fel kan uppstå. Dels kanske det inte ryms fler konton i konto-arrayen och dels kan det valda namnet redan vara upptaget (i ett riktigt system använder man naturligtvis personnummer för en säkrare identifikation).

```
void Bank::newAccount() {
  cout << "* Skapa nytt konto *" << endl;
  // Kontroll om fler konton får plats
  if (noOfAcc >= MAX_NR_ACC)
    cout << "Inga fler konton får plats!" << endl;
  else {
    Account temp;
    temp.input(); // Läs in ny kund
    // Kontrollera att namnet inte redan finns
    if ( searchCustomer( temp.getName() ) >= 0 )
      cout << "*** Fel: Namnet finns redan!" << endl;
    else {
        accounts[noOfAcc] = temp;
        curAccount = noOfAcc;     // Nya kunden blir aktuell kund
        noOfAcc++;
    }
  }
```

```
}
```

Konstruktor och `printAllAccounts` är oförändrade från det det första exemplet med Bank-klassen.

Övningar

1. Skriv `main`-funktionen till det nu genomgångna banksystemet
2. Modifiera `newAccount` så att konto-arrayen hålls sorterad i bokstavsordning på innehavarens namn
3. Modifiera `Account`-klassen så att den håller reda på de senaste 10 transaktionerna (insättningar och uttag). Se till att de skrivs ut av `print`-funktionen. Hur skall en eventuell selektor för transaktionslistan se ut?

Programmet i sin helhet finns tillgängligt på `.../kap9ex/Bank2/`

9.6 Andra härledda datatyper

9.6.1 Struct

Det finns ett annat begrepp i C++ kallat `struct` som är mycket snarlikt `class`. Den enda skillnaden är att default-skyddet är `public` i stället för `private`. Det verkar således vara ett ganska onödigt begrepp. Orsaken till att `struct` ändå finns är önskemålet att hålla C++ bakåtkompatibelt med C. Genom att tillåta den i C++ kan alltså C-program kompileras med C++-kompilatorer. Det finns egentligen aldrig någon anledning att använda den i C++.

9.6.2 Uppräkningsbara typer

Ibland finns behovet att datatyper med endast ett fåtal värden. Antag t ex att man vill ha en variabel som står för någon av danserna vals, foxtrot, quickstep, tango eller lindy. Detta kan naturligtvis göras så att man ”kodar” olika dansslag som heltal med t ex 0 för vals, 1 för foxtrot, 2 för quickstep etc. För att slippa att komma ihåg vilken kodning man har använt kan man definiera konstanter motsvarande respektive dans:

```
const int WALTZ = 0, FOXTROT= 1, QUICKSTEP=2;
int nextDance = FOXTROT;
```

Detta kan göras något elegantare med `enum` enligt följande programskiss:

Exempel 9.10 Demonstration av enum

```
// File name: .../dance.cc

#include <iostream>
```

```
enum dance {
  WALTZ, FOXTROT, QUICKSTEP, TANGO, LINDY
};

int main() {
  dance first=WALTZ, next=QUICKSTEP, Favourite=TANGO;

  // Calculate next dance ...
  next = TANGO;

  if (next==Favourite)
    cout << "Dags för " << next << endl;
}
```

Tyvärr blir in- och utmatning i numerisk form d v s ovanstående program kommer att skriva

```
Dags för 3
```

Konstanterna kommer alltså att erhålla heltalsvärden med början på noll.

Egentligen tillför inte `enum` så mycket mer än ett naturligare sätt att deklarera variabler samt att få en viss felkontroll gjord. Följande sats ger t ex kompileringsfel

```
next = 23;
```

Man bör dock känna till begreppet eftersom det används ganska ofta.

9.7 Några avslutande kommentarer

9.7.1 Vem är användaren

Normalt menas med användaren den person som kör (använder) programmet när det är klart. Vi har dock talat om användaren till klassen si och så. Då avser vi den programmerare som använder klassen vilket mycket väl kan vara samma person som skrivit klassen. Man kan också avse den *kod* som instansierar klassen och anropar dess metoder.

9.7.2 Synonymtabell

Det förekommer en massa uttryck för samma saker inom programmering. Här kommer ett litet försök att lista en synonymtabell vad avser uttryck som hör samman med klasser och klassers funktioner:

funktioner i en klass	medlemsfunktioner medlemsmetoder operationer
datafält i en klass	attribut datamedlemmar
anropa funktion i en klass	skicka ett meddelande till ett objekt att använda en metod
en variabel av typen klass	ett objekt en instans av en klass

9.8 Kom ihåg

- En klass är en paketering av data och funktioner.
- Skyddsnivåer `public`, `protected` och `private`.
- Data är vanligen skyddat (`protected` eller `private`.
- Metoder (funktioner) är vanligen publika.
- Data + funktioner = inkapsling.
- Objekt = variabel av typen klass.
- Varför inkapsling och `protected`? Jo, frihet att ändra.
- Disciplin påtvingad.
- Initiering med konstruktor. Oskadliggörande med destruktor.
- `struct` är en klass utan åtkomstskydd.
- `enum`

Kapitel 10

Pekare och dynamiska variabler

10.1 Pekarvariabler

Betrakta följande kod

```
double d;
d = 13.78;
cout << d;
```

Vi har här skapat en variabel `d` av typen `double`. Denna variabel får ett utrymme om (vanligen) 8 byte någonstans i datorns primärminne. Man kan säga att `d` är vårt programs namn på denna plats i minnet. Varje byte i minnet har en *adress*. Det betyder att vår variabel har en adress som är adressen till den första byten som hör till variabeln. Som programmerare använder vi i regel variabelnamnet och låter C++ hålla reda på dess adress så att tilldelningssatsen ovan fyller rätt 8 byte med det bitmönster (64 ettor/nollor) som motsvarar talet 13.78. I utskriftssatsen hämtas värdet från samma adress och värdet skrivs ut.

Det går att ta reda på adressen till en variabel med hjälp av *adressoperatorn* `&`:

```
cout << d ;   // Skriver d:s innehåll
cout << &d;   // Skriver d:s adress
```

Det är utomordentligt sällan vi har anledning att skriva ut adressen som vi gjorde i exemplet ovan men däremot kan vi ha stor nytta av att kunna hantera adresser (lagra, jämföra, ...) internt i programmen. Vad skall vi använda för datatyp för detta? Egentligen skulle man kunna använda heltal eftersom minnescellerna (byten) är numrerade från noll och uppåt. Så gör man dock inte utan har en särskild datatyp för detta ändamål som brukar kallas *pekare*.

För att deklarera *pekarvariabler* använder man de vanliga deklarationerna som vi redan sett *men* låter de identifierare som skall vara pekare föregås av en asterisk (`*`). Exempel:

```
double *q, a, *r;
int    x, y, *xp, yp;
```

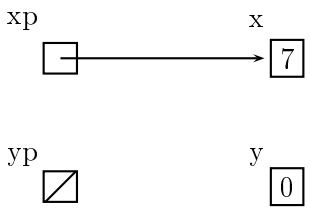
Dessa satser deklarerar två heltalsvariabler, två pekare till heltal, en `double`-variabler och två pekare till `double`-värden. Man måste således skilja på pekare till olika datatyper — en pekare till en `int` kan inte (utan våld) sättas att peka till en `double`.

Pekarvariabler kan ges värdet 0 (skrivs ibland `null`) som betyder ”pekare till ingenting” eller en adress som t ex kan få av adressoperatorn `&`.

Exempel:

```
int x=7, y=0;       // Vanliga int-variabler
int *xp=&x, *yp=0;  // Pekare till int-variabel
```

Situationen kan illustreras på följande sätt



De fyra variablerna reprenteras av var sin ruta med namnet bredvid. I rutorna för heltalsvariablerna har vi placerat deras värden medan pekarvariablernas värden representeras av pilar till det som de pekar på. Pekarvärdet 0 representeras av ett snedstreck i rutan.

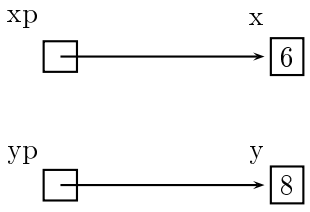
Vi måste också kunna komma åt det ställe som en pekare pekar till. Även till detta används operatorn `*` så att uttrycket

`*`pekarvariabel

står för det som pekaren pekar till. I vårt exempel står alltså `*xp` för `x`. Efter satserna

```
y   = *xp + 1; // y får värdet 8 (*xp är det som xp pekar på)
*xp = 6;       // x får värdet 6
yp  = &y;      // yp sätts att peka till y
```

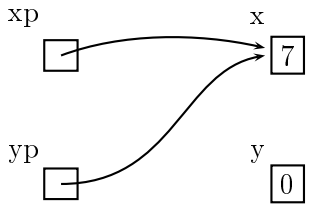
kommer således bilden att vara



Ytterligare manipulationer

```
(*xp)++;       // x får värdet 7
*yp = 0;       // y får värdet 0
yp  = xp;      // yp pekar nu till x
```

ger följande bild.



Det vi gått igenom här har bara varit till för att illustrera hur pekare fungerar. Själva manipulationerna har ju inte tillfört något nytt. Längre fram i kapitlet kommer vi se vad pekare verkligen används till.

10.2 Pekare och arrayer

Det finns ett nära släktskap mellan pekare och arrayer som kommer framgå av följande exempel och diskussion.

Exempel 10.1 Exempel med pekare och arrayer

```
// Filnamn : .../pekArr.cc
#include <iostream>
using namespace std;

int main() {

  const int max = 20;
  double a[max];

  for ( int i=0; i<max; i++ )
    a[i] = 0.1*i;

  double *p = &a[0];
  cout << "p pekar till    : " << *p << endl;

  p = p + 10;
  cout << "p pekar till    : " << *p << endl;

  p++;
  cout << "p pekar till    : " << *p << endl;

  cout << "p+3 pekar till  : " << *(p+3) << endl;
  cout << "Alternativ (fel): " << *p+3 << endl;
```

```
  return 0;
}
```

Utskriften från programmet blir:

```
p pekar till : 0
p pekar till : 1
p pekar till : 1.1
p+3 pekar till : 1.4
Alternativ (fel): 4.1
```

- I exemplet låter vi `p`, som är en pekare till värden av typen `double`, peka på det första elementet i en array med sammanlagt 20 element. Dessa element har värden 0.0, 0.1 o s v.
- Vid den första utskriften skrivs 0 ut eftersom `p` pekar på det 0:e elementet.
- Därefter har vi satsen `p = p+10` vilket innebär att vi låter `p` peka 10 ”doubles” längre fram, d v s 80 bytes längre fram. C++ håller reda på hur många byte varje steg motsvarar. Utskriften blir 1, vilket är det värde det 10:e elementet i `a` har.
- På motsvarande sätt flyttar satsen `p++` fram pekaren ett steg.
- I den näst sista utskriftssatsen skrivs det element som ligger tre steg efter det som `p` för närvarande pekar på, d v s det 14:e elementet eftersom `p` pekar på det 11:e.
- I den sista satsen skrivs 4.1 ut. Kan du lista ut varför? Det har med prioriteten mellan operatorer att göra.

Vi kan alltså adressera oss med hjälp av en pekare med indicering:

`*(`*pekare* + *index*`)`

Nu finns det en kortare variant i C++ för att slippa parentesen och stjärnan:

`*(`*pekare* + *index*`)` = *pekare*`[`*index*`]`

Detta är ju således exakt som adressering i en array!

Arraynamn, `a` i vårt fall, är (eller kan behandlas) som pekare till det första elementet i arrayen. Vi hade därför kunnat ändra tilldelningen

```
double *p = &a[0];
```

till följande

```
double *p = a;
```

I array-fallet tar C++ hand om att allokera tillräckligt med minne och att hantera adressen riktigt. Vi skall senare se att man med hjälp av en pekare själv kan allokera minne.

När vi i kapitlet om arrayer diskuterade hur sådana fungerar som parametrar så lämnade vi frågan varför dessa verkade fungera som referensparametrar trots att de syntaktiskt är värdeparametrar. Varför skapas det inte en kopia av arrayen som vi förväntar oss? Svaret på frågan är att vi faktiskt skickar en kopia av array-variabeln, men denna variabel är ju bara en adress till det första elementet i arrayen. Det spelar ingen roll om vi skickar originaladressen eller bara en kopia av adressen.

Detta om pekare och arrayer. Låt oss nu se varför pekare är användbara!

10.3 Dynamiska arrayer

När vi deklarerar en array så måste vi ange dess storlek i programkoden. Om vi inte på förhand vet hur många element som får vi ta till ”värsta fall”. Låt oss ta ett exempel:

Exempel 10.2 Exempel med ”värsta fall”

```
// Filnamn : .../worstCase.cc
#include <iostream>
using namespace std

int main() {
  const int max = 1000;           // worst case
  double dArray[max];

  int num,i;
  double sum,mean;

  cout << "For how many numbers do you want the mean calculated: ";
  cin >> num;
  if ( (num<1) || (num>max) ) return 0;

  sum = 0.0;
  for ( i=1; i<=num; i++ ) {
    cout << "Give number " << i << " : ";
    cin >> dArray[i-1];
    sum += dArray[i-1];
  }
  mean = sum/num;
  cout << "The values divided by the mean are:" << endl;
  for ( i=1; i<=num; i++ )
    cout << dArray[i-1]/mean << endl;
  return 0;
}
```

Här har vi förutsatt att ingen orkar knappa in fler än 1000 tal.

Detta är ett slösaktigt sätt att hantera minne utan att det ändå kunna garanterar att det alltid räcker. Det finns därför ett sätt att bestämma hur stor arrayen skall vara när programmet *körs* i stället för när det *skrivs*. Man säger att man *allokerar minne dynamiskt* och här kommer pekare in för att hålla reda på sådant minne. Om vi använder det i samma problem som i ovanstående exempel ser det ut på detta sätt:

Exempel 10.3 Exempel med dynamisk array

```
// Filnamn : .../dynArr.cc
#include <iostream>
using namespace std;

int main() {
  int num,i;
  double sum,mean;

  cout << "For how many numbers do you want the mean calculated : ";
  cin >> num;
  if (num<1) return 0;

  double *dPointer = new double[num];     // allokering

  sum = 0.0;
  for ( i=1; i<=num; i++ ) {
    cout << "Give number " << i << " : ";
    cin >> dPointer[i-1];
    sum += dPointer[i-1];
  }
  mean = sum/num;
  cout << "The values divided by the mean are:" << endl;
  for ( i=1; i<=num; i++ )
    cout << dPointer[i-1]/mean << endl;

  delete[] dPointer;                      // återlämning
  return 0;
}
```

- Vi talar om med `new double[num]` hur många element vi vill ha i arrayen.
- `new` reserverar (allokerar) tillräckligt med minne (om möjligt) och tilldelar pekarvariabeln adressen till det första elementet.
- Minne lämnas tillbaka med `delete`. När pekaren, som i detta fall, refererar till en array så måste `delete` följas av `[]` — i annat fall lämnas bara första elementet tillbaka!
- Det vore elegantare att låta programmet själv räkna hur mycket använ-

daren skriver in i stället för att fråga om det i förväg. En lösning på det problemet demonstreras i exemplet med den länkade listan längre fram i detta kapitel.

En pekare kan användas flera gånger i samma program med olika ”storlekar” som följande kodskiss visar

```
double *dP1, *dP2;
                    // Allokera två minnesareor
dP1 = new double[100];
dP2 = new double[200];

.... kod som använde dessa två arrayer

delete[] dP1;       // Återlämning av areorna
delete[] dP2;
                    // Två nya areor
dP2 = new double[2000];
dP1 = new double[20];

... kod som använder dessa två arrayer

delete[] dP1;       // Återlämning
delete[] dP2;
```

Vi använder `new` och `delete` fleras gånger i programmet och låter på så sätt arrayerna ändra storlek under körningen.

Att hantera storleken på arrayer på detta sätt under programmets exekvering kallas för att minnet hanteras dynamiskt (föränderligt), till skillnad från de statiska arrayerna.

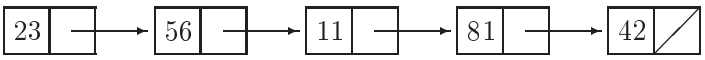
Övning

- Ändra i bankontoklassen i föregående kapitel så att kontoarrayen hanteras dynamiskt.
- Modifiera `void Bank::newAccount` i föregående kapitel så att den allokerar en ny, större kontoarray om ett nytt konto inte får plats.

10.4 Pekare till objekt

Naturligtvis kan man ha pekare till klassobjekt och dessa kan också innehålla pekare. En användning av detta är att skapa länkade strukturer d v s en struktur med flera objekt som pekar till varandra. Det enklaste exemplet på en sådan är en linjär lista. Antag att vi, som i exemplet med den dynamiska arrayen, vill läsa in och lagra ett antal tal men att man inte vet i förväg hur många det blir. I detta exempel antar vi att det är positiva heltal och att vi använder noll som slutmarkering. Då kan vi bygga upp en kedja av objekt där var och ett

innehåller ett tal och en pekare till nästa. Detta kan åskådliggöras på följande sätt:



Vi måste också ha en variabel som refererar första elementet i listan. Snedstrecket i sista rutan symboliserar null-pekaren d v s en pekarvariabel som inte pekar till något.

Exempel 10.4 Länkad lista med heltal

```
// Filnamn : .../linkedList.cc
#include <iostream>
using namespace std;

class listElement {
public:
  listElement();
  listElement(int newKey, listElement *follower);
  ~listElement();
  void printList();
protected:
  int key;
  listElement *next;
};

listElement::listElement() {
  key = 0;
  next = 0;    // Null-pekaren
}


listElement::listElement(int newKey, listElement *follower) {
  key = newKey;
  next = follower;    // Null-pekaren
}

listElement::~listElement() {
  if (next) delete next;
}

void listElement::printList() {
  cout << key;
  if (next)
   (*next).printList(); // recursive call to print the rest
}
```

```
int main() {
  listElement *first = 0;   // Null pointer initially
  cout <<"Give positive integers. Terminate with zero" << endl;
  int number;
  cin >> number;
  while (number > 0) {
    first = new listElement(number,first);
    cin >> number;
        }
  (*first).printList();
}
```

- Klassen har två konstruktorer även om vi bara använder den ena i exemplet. I den konstruktorn så ger vi heltal och pekare till efterföljare.
- Observera att 0 användes som nullpekare!
- Destruktorn ser till att efterföljande element, om det finns,t blir destruerat d v s om `next` inte är en null-pekare. När efterföljaren skall destrueras så destrueras först dess efterföljare — en slags rekursion med andra ord!
- `printList` är också rekursiv: den börjar med att skriva ut sig själv och om det sedan finns någon efterföljare så skrivs den ut o s v
- När vi har en pekare till ett objekt så skriver man

  (**pekare*).*komponent*

  för att nå en komponent. Eftersom detta är en mycket vanlig situation har man fört in en särskild operator `->` för detta. Vi skulle således kunnat skriva `first->printList()` i stället för `(*first).printList()`.
- Talen kommer att skrivas ut i omvänd ordning d v s med det sist inläst först. Hur skall man göra för att få det först inlästa utskrivet först?
- Testprogrammet förutsätter att minst ett tal större än noll matas in. Om så ej sker kommer anropet till `printList` att ge fel. (Vad blir det för fel? Hur gör man för att undvika det?)

10.5 Några avslutande anmärkningar

10.5.1 Arrayparametrar och pekare, en gång till

Vi har tidigare sagt att den formella parametern för en array måste se ut på följande sätt för att C++ ska förstå att det är en array:

```
int sumArray(const int num, double data[])
```

Nu vet vi bättre. Eftersom en array egentligen är en pekare kan vi istället skicka pekaren som parameter:

```
int sumArray(const int num, double *data)
```

Detta är ett vanligt sätt att deklarera arrayer som formella parametrar och det är därför viktigt att du känner igen skrivsättet.

10.5.2 Typedef

När flera pekare deklareras i samma sats måste `*` anges för varje pekarvariabel:

```
double *dP1, *dP2;
```

Satsen

```
double* dP1, dP2;
```

deklarerar således en pekare-variabel `dP1` och en `double`-variabel `dP2` även om det kan se ut som att man deklarerar två pekarvariabler.

För att undvika dessa fel kan en ny typ, en ”pekare till double”-typ, definieras på följande sätt:

```
typedef double *doublePointer;
doublePointer dP1, dP2;
```

Med `typedef` har vi skapat ett alias för `double *` och kan på så sätt undvika problemet ovan.

10.5.3 Pekare är en vanlig orsak till fel

Pekare är en stor källa till svårfunna fel i program. Det beror på att systemen inte kontrollerar att pekarna har vettiga värden och att felen kan uppstå på helt andra ställen i programmen än där de verkligen finns. Se till att du förstår hur pekar fungerar och var extra noga vid användning av dem! Tänk särskilt på

- att inte använda pekare som inte fått värde,
- att inte försöka följa noll-pekaren,
- att inte lämna tillbaka utrymme som används (t ex om två pekare pekar till samma ställe),
- att bara lämna tillbaka utrymme som allokorats med `new` (dvs *inte* lämna tillbaka utrymme som fått pekare till t ex genom att använda adressoperatorn)
- att det *inte* är pekarvariablerna som lämnas tillbaka med `delete` utan det som de pekar på
- att när ett dynamiskt allokerat utrymme lämnats tillbaka så är det illegalt att använda pekaren utan att ge den ett nytt värde.

10.5.4 Vad används pekare till?

I programmeringsspråket C används ofta mycket pekare bl a eftersom referensanrop inte finns som parameteröverföringsmetod. I C++ behöver man inte lika ofta använda pekare. De två viktigaste användningarna har demonstrerats i detta kapitel: dynamisk allokering av minne och länkade strukturer.

10.6 Kom ihåg

- Pekare
- Adressoperatorn `&` och ”följa pekaroperatorn” `*` Båda dessa används ju också för helt andra operationer — det är sammanhanget som avgör vad de betyder!
- Pekare kontra arrayer och arrayer som formella parametrar.
- `new`, `delete` för att skapa och lämna tillbaka dynamiska variabler.
- `delete[]` för att återlämna dynamiska arrayer.
- länkade strukturer
- `typedef`.

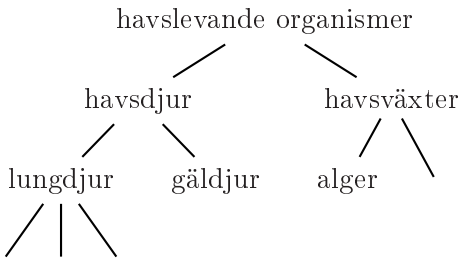
Kapitel 11

Klasshierakier

I förra kapitlet introducerade vi begreppet klass. En klass inkapslar ett objekts egenskaper både vad avser data och metoder. En klass påtvingar också programmerarna mer disciplin i och med att det är möjligt att skydda medlemmar av objekten från yttre påverkan. Den som utformar en klass bestämmer vad som får göras med den. Denna disciplin skall inte ses som något negativt utan som en hjälp att hålla ordning och reda i programmet.

11.1 Taxonomier

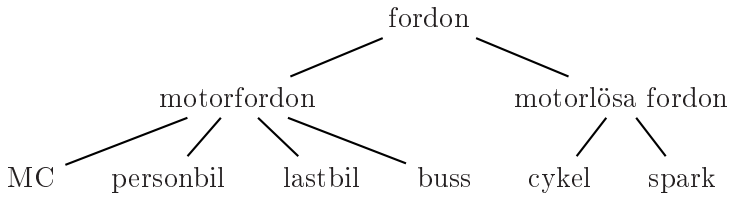
I detta kapitel skall vi utveckla klassen mot ännu mer ordning och reda. Ett sätt att skapa ordning är att införa organisation och struktur. Carl von Linné införde struktur genom att indela organisera djurvärlden i familjer, arter osv. Låt oss titta på ett exempel på organisation av sjölevande organismer, fiskar, växter m m i följande figur.



Exemplet är väl inte biologiskt välgrundat utan mer tänkt som ett tankeexperiment. Så dumt är det väl inte då man betänker att Linné lade in stenar som levande varelser i sina system.

En organisation av detta slag kallas för *taxonomi*. Ni har säkert sett dem på många ställen.

Låt oss ta ett icke-biologiskt exempel:



Här låter vi fordon vara den mest generella klassen. Det finns vissa saker som är gemensamma för alla fordon (t ex vikt) medan andra egenskaper som motorstyrka bara finns hos motorfordon.

För att räkna ut fordonsskatt så används olika formler för olika typer av fordon. Det är exempel på en metod som är finns för alla motorfordon men som har olika implementationer för olika typer av sådana.

På detta sätt bygger vi upp en hierarkisk struktur, ett arvsträd, av olika fordonstyper. Ju längre ner vi kommer ju mer specialiserade blir beskrivningarna. Uppåt i trädet höjs abstraktionsnivån. En arvinge har alltid alla de egenskaper som föräldern har, som i sin tur har sin förälders egenskaper.

I C++ finns kan man beskriva hierarkista strukturer som ovan med hjälp av *klasshierarkier* Givet en viss klass — en så kallad *basklass* (t ex fordon) så kan man definiera *subklasser* (*underklasser*) som specialiseringar av basklassen (t ex motorfordon och motorlösa fordon). Dessa får då alla fordonens egenskaper (attribut och metoder) samt de extra egenskaper som behövs för just den subklassen. Man säger att subklassen *ärver* alla basklassens egenskaper. Arv är objektorienteringens andra huvudprincip.

11.2 Basobjektet

Låt oss nu gå över till att se hur dessa arv går till i C++. Som exempel har vi valt att göra ett personregister. I registret ska vi kunna ha ett stort antal olika personkategorier, t ex studenter, lärare, idrottare m m. Vår första uppgift blir att hitta en lämplig stamfader till alla dessa olika typer.

Processen att skapa en lämplig taxonomi är inte självklar eller entydig. Någon kan hävda att i fordonsexemplet skulle fordon delas in i hjulfordon och bandfordon och hjulfordon i sin tur i motorhjulfordon osv..

Här bestämmer vi att stamfadern skall vara en klass som heter `person`. En person har endast två datafält — ett för förnamn och ett för efternamn. (I verkligheten skulle man säkert ha mera data som personnummer, adress . . .)

Låt oss titta på deklarationsfilen till `person`.

Exempel 11.1 Deklarationsfil till klassen person

```
// Filnamn : .../person.h
#ifndef __person__
```

```
#define __person__

#include<string>
using namespace std;

class person {
 public:
  person();               // defaultkonstruktor
  person(string fn, string ln); // överlagrad konstruktor
  ~person();              // destruktor

  void report();          // Skriv all
  void writeName();       // Skriv namnet
  void input();           // Läs in person

 protected:
  string fname;
  string lname;
};
#endif
```

De två datamedlemmarna är som vanligt deklarerade som `protected`.

Klassen `person` har två konstruktorer, en default och en med parametrar samt en destruktor. Det finns också tre medlemsfunktioner, en för att skriva ut persondata (`report`), en för att skriva ut enbart namnet och en för att läsa in data från användaren (`input`).

Låt oss för fullständighetens skull titta på implementationen av dessa metoder:

Exempel 11.2 Implementationsfil för klassen person

```
// Filnamn : .../person.cc
#include<iostream>
#include<string>
#include "person.h"
using namespace std;

// Konstruktorer och destruktor

person::person() {
  fname = "Arne";
  lname = "Anonym";
  cout << "Välkommen " << fname << endl;
}

person::person(string fn, string ln) {
  fname = fn;
  lname = ln;
  cout << "Välkomen " << fname << endl;
```

```
}

person::~person() {
  cout << "Ajöss " << fname << " " << lname << endl;
}


// Publika metoder

void person::report() {
  cout << fname << " " << lname << endl;
}

void person::writeName() {
  cout << fname << " " << lname;
}

void person::input() {
  cout << "Förnamn: ";
  getline(cin, fname);
  cout << "Efternamn: ";
  getline(cin, lname);
}
```

Vi har sett liknande medlemsmetoder i förra kapitlet. Vad vi noterar är följande:

- Vi behöver egentligen inte definiera någon destruktor eftersom vi inte använder `new` och därför inte behöver använda `delete`. Vi har dock lagt till en utskrift för att vi skall kunna följa programmets förlopp.
- De två konstruktorerna innehåller inga överraskningar. De avslutande utskrifterna är helt onödiga och endast till för ”programförföljelse”.

Ett huvudprogram som använder `person` kan se ut som:

Exempel 11.3 Program som använder person

```
// Filnamn : .../personTest1.cc

#include <iostream>
#include "person.h"
using namespace std;

int main() {
  person someOne( "Richard", "III" );
  person anotherOne("Katarina", "Jagellonica");
  person noOne;

  someOne.report();
  anotherOne.report();
  noOne.report();
```

```
  cout << "Hej ";
  someOne.writeName();
  cout << ", trevligt att se dig!" << endl;

  return 0;
}
```

Slutligen ett körexempel:

```
Välkommen Richard
Välkommen Katarina
Välkommen Arne
Richard III
Katarina Jagellonica
Arne Anonym
Hej Rickard III, trevligt att se dig!
Ajöss Arne Anonym
Ajöss Katarina Jagellonica
Ajöss Richard III
```

Vi har markerat utskrifterna från konstruktorer och destruktorer med kursiv stil. Exemplet är så långt mer eller mindre repetition från föregående kapitel. Låt oss fullända repetitionen med ytterligare ett program som utnyttjar person, fast med pekare:

Exempel 11.4 Program som utnyttjar person mha pekare

```
// Filnamn : .../personTest2.cc

#include <iostream>
#include "person.h"
using namespace std;

int main() {
  person *me;
  person *partner = new person("Eliza","Doolittle");
  me = new person;
  *me = person("Henry","Higgins");

  partner->report();
  me->report();

  delete partner;
  delete me;

  return 0;
}
```

Utskriften från programmet blir:

```
Välkommen Eliza
Välkommen Arne
Välkommen Henry
Eliza Doolittle
Henry Higgins
Ajöss Eliza Doolittle
Ajöss Henry Higgins
Ajöss Henry Higgins
```

Utskriften

```
Välkommen Arne!
```

kommer från

```
me = new person!
```

och

```
Välkommen Richard!
```

från konstruktorn

```
person("Richard", "III")
```

Objektet stannar kvar under programmet som en anonymt objekt. Därför har vi tre anrop till destruktorn och inte bara de två som motsvarar våra `delete`-satser.

11.3 Arv

Nu går vi vidare och definierar en arvinge till person, en *idrottare*. Innan vi definierar en idrottare ska vi fråga oss om det är lämpligt att en idrottare skall vara en arvinge till en person. Testet på lämpligheten som arvinge i C++ är att arvingen kan uttryckas i en mening som börjar på:

En idrottare är en person som ..

Nyckelordet är att vi skall ha en relation: *är en*. I vårt fall definierar vi en idrottare som en person med ”tilläggsegenskapen” att ha en *idrottsgren* och en uppgift om hur många priser han vunnit.

Exempel 11.5 Deklarationsfil för sportsman

```
// Filnamn : .../sportsman.h

#ifndef __SPORT__
#define __SPORT__

#include <string>
#include "person.h"
using namespace std;
```

```
class sportman : public person {
 public:
  sportman();                // Standardkonstruktor
  sportman( string fn, string ln,
     string br, int nfp );
                              // Överlagrad konstruktor
  ~sportman();               // Destruktor

  void report();              // Skriver data på skärmen
  void input();               // Tar in data från användaren
  void firstPrize();          // Räkna upp antal förstapris

  // Selektorer
  string getBranch();
  int getNrFirstPrize();

 protected:
  string branch;              // Idrottsgren
  int nrFirstPrize;           // Antal förstapris
};
#endif
```

Det första vi noterar i definitionen av `sportsman` är texten `: public person`. Detta anger att `sportsman` är en arvinge till `person` dvs en *utbyggnad* eller *specialisering* av den klassen. Ordet `public` innebär att attribut som är tillgängliga (`public`) i klassen `person` kommer att vara tillgängliga även i klassen `sportsman` vilket är det normala sättet.

En `sportsman` har fått två nya dataattribut jämfört med `person`: `branch` och `nrFirtsPrize`. Klassen `sportsman` har därmed fyra datafält: `fname` och `lname` som ärvs från `person` och de ”egna” `branch` och `nrFirstPrize`. Alla dessa fält är `protected` ty `person` hade deklarerat sina fält som `protected`.

För `sportsman` har vi definierat två konstruktorer och en destruktor som vi skall se mer på strax.

Vidare har vi deklarerat fyra vanliga medlemsmetoder. Metoderna `report` och `input` fanns också hos `person` men vi vill omdefiniera dem så att de hanterar de tillagda attributen. Metoderna `firstPrice`, `getBranch` och `getNrFirstPrize` har ingen motsvarighet i `person` eftersom de skall hantera de nya datafälten. Observera att vi inte har omdefinierat metoden `writeName` som fanns i `person`. Denna metod kan på så sätt användas för ett `sportsman`-objekt eftersom dessa ärver alla egenskaper som `person` har och som inte omdefinierats.

Låt oss titta på implementationen av dessa metoder:

Exempel 11.6 Implementation av metoder för sportsman

```
// Filnamn : .../sportsman.cc
```

```
#include <iostream>
#include <string>
#include "person.h"
#include "sportsman.h"
using namespace std;

sportsman::sportsman() : person() {
  branch = "";
  nrFirstPrize = 0;
  cout << "Välkommen sportsman " << fname << endl;
}

sportsman::sportsman( string fn, string ln, string br, int nfp )
  : person(fn, ln) {
  branch = br;
  nrFirstPrize = nfp;
  cout << "Välkommen sportsman " << fname << endl;
}

sportsman::~sportsman() {
  cout << "Ajöss sportsman " << fname << " " << lname << endl;
}

void sportsman::report() {
  person::report();
  cout << "och utövar " << branch << ".\n"
       << fname << " har vunnit " << nrFirstPrize
       << " förstapriser" << endl;
}

void sportsman::input() {
  person::input();
  cout << "Ange gren för " << fname << " : ";
  getline(cin, branch);
  cout << "Ange antal förstapriser : ";
  cin >> nrFirstPrize;
  cin.get();
}

void sportsman::firstPrize() {
  nrFirstPrize++;
}

string sportsman::getBranch() {
  return branch;
}

int sportsman::getNrFirstPrize() {
  return nrFirstPrize;
}
```

Den första konstruktorn utan argument har ett huvud som ser ut som

```
sportsman::sportsman() : person()
```

Detta anger att default-konstruktorn skall användas när basklassdelen konstrueras.

I konstruktorn med parametrar använder vi den konstruktor i klassen `person` som har parametrar.

Anledningen till av vi måste ange vilken konstruktor som skall användas för `person`-delen är att det inte alls är självklart vad som skall väljas när det finns flera stycken.

I destruktorn anger vi *inte* att destruktorn i `person` skall anropas. Detta kommer att göras i alla fall. Det finns ju bara en destruktor för varje klass att välja på!

Observera att de avslutande utskriftssatserna i dessa metoder är helt onödiga och endast till för att vi skall kunna följa programmet enklare.

Metoderna `input()` och `report()` ska göra samma sak som som motsvarande metoder i `person` och lite till. Det enklaste sättet är då att först använda `person::input()` respektive `person::report()` för att därefter läsa in/skriva ut det som är unikt för `sportsman`. Observera att vi måste ange `person::input()` för att det inte skall bli ett rekursivt anrop till `sportsman::input()`.

De två sista metoderna definierar ökar respektive returnerar antalet förstapris.

Låt oss titta på ett litet program som utnyttjar `sportsman`.

Exempel 11.7 Testprogram för `sportsman`

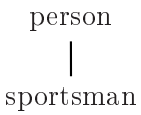
```
// Filnamn : .../sportsmanTest1.cc
#include <iostream>
#include <string>
#include "sportsman.h"
using namespace std;

int main() {
  sportsman anyOne("Kajsa", "Bergkvist", "höjdhopp", 100);

  anyOne.report();
  anyOne.firstPrize();
  cout << "Hej ";
  anyOne.writeName();
  cout << ", du har " << anyOne.getNrFirstPrize()
       << " förstapriser!" << endl;

  return 0;
}
```

Programmet definierar en `sportsman`, skriver ut en rapport, ökar på antaler förstapris och utnyttjar metoderna `writeName()` och `getNrFirstPrize` för att skriva ut ytterligare en hälsning. Observera att `writeName()` behandlas på exakt samma sätt som `getNrOfFirstPrize` trots att `writeName()` är ärvd. Så här ser det ut när programmet körs:

```
Kajsa Bergkvist
och utövar höjdhopp.
Kajsa har vunnit 100 förstapriser.
Hej Kajsa Bergkvist, du har 101 förstapriser!
Ajöss sportsman Kajsa Bergkvist
Ajöss Kajsa Bergkvist
```

Vi ser att när konstruktorn till `sportsman` anropas får vi först ett anrop till personkonstruktorn `person()` och därefter anropas `sportsman()`. Ordningen är logiskt ty stamfadern skall initieras innan de nya fälten initieras. För destruktorn ser vi att anropsordningen är omvänd, först anropas `~sportsman()` och därefter `~person()`. Observera vidare de två raderna som blir resultatet av anropet till `anyOne::report()`. Den första raden kommer från `person::report()` och den andra från `sportsman::report()`.

Vad vi har gjort nu är att skapa en (liten) hierarki av persontyper:



11.4 Mera arv

Vi kan nu bygga ut vår arvshierarki ytterligare. Om vi vill hantera idrottare i olika grenar kan vi definiera t ex definiera en klass för löpare och en för tyngdlyftare som underklasser till `sportsman`.

Löparklassen skulle kunna få följande utseende:

Exempel 11.8 Deklarationsfil för `runner`

```
// Filnamn : .../runner.h

#ifndef __RUNNER__
#define __RUNNER__

#include <string>
#include "sportsman.h"
using namespace std;

class runner : public sportsman {
 public:
  runner();                  // Standardkonstruktor
```

```
  runner( string fn, string ln,
  string br, int nfp, double ci );
                              // Överlagrad konstruktor
  ~runner();                  // Destruktor

  void report();              // Skriver data på skärmen
  void input();               // Tar in data från användaren
  double getConditionIndex();// Selektor
  void setConditionIndex( double ci );// Sätt konditionsindex

 protected:
  double conditionIndex;      // En löpares syreupptagningsförmåga
};
#endif
```

Som framgår av deklarationen är klassen `runner` en arvinge till `sportsman`. Vi har tre nya attribut samt en medlemsmetod som sätter dessa. Metoderna `report()` och `input()` är omdefinierade för att ta hand om nya data. Båda konstruktorerna och destruktorn är nya och vi har dessutom lagt till ytterligare två metoder för att hämta respektive sätta ett *konditionsindex*. Delar av implementationen kan se ut som följer.

Exempel 11.9 Implementationsfil för `runner`

```
// Filnamn : .../runner.cc

#include <iostream>
#include <string>
#include "sportsman.h"
#include "runner.h"
using namespace std;

runner::runner() : sportsman() {
  conditionIndex = 0;
  cout << "Välkommen runner " << fname << endl;
}

runner::runner( string fn, string ln,
  string br, int nfp, double ci )
  : sportsman( fn, ln, br, nfp ) {
  conditionIndex = ci;
  cout << "Välkommen runner " << fname << endl;
}

runner::~runner() {
  cout << "Ajöss runner " << fname << " " << lname << endl;
}

void runner::report() {
  sportsman::report();
```

```
  cout << "Konditionsindex : " << conditionIndex <<endl;
}

void runner::input() {
  sportsman::input();
  cout << "Ge konditionsindex : ";
  cin >> conditionIndex;
  cin.get();
}

double runner::getConditionIndex() {
  return conditionIndex;
}

void runner::setConditionIndex( double ci ) {
  conditionIndex = ci;
}
```

Principen är densamma som tidigare. Utnyttja de metoder som finns och omdefiniera det du behöver.

Antag att vi också definierar en klass `weightlifter` som subklass till klassen `sportsman`. Du får själv utforma klassen som övning!

Låt oss nu bygga ut exemplet med en `student` som arvinge till `person`.

Exempel 11.10 Definition av `student`

```
// Filnamn : .../student.h
#ifndef __student__
#define __student__
#include "person.h"

class student : public person {
 public:
  student();                          // Defaultkonstruktor
  student( string fn, string ln,
   string edu );            // Överlagrad konstruktor
  ~student();                         // Destructor

  void report();                      // Skriv ut data
  void input();                       // Läs in
  void newCourse( string cName, int grade,
  double points );  // Lägg till ny kurs
  double totalPoints();               // Returnera totalpoäng
  int lastgrade();                    // Returnera senaste betyg

 protected:
  double totPoints;
  int lastGrade;
```

```
  string education;
  string lastCourse;
};
#endif
```

med följande implementation

Exempel 11.11 Implementation av `student`

```
// Filnamen : .../student.cc
#include<iostream>
#include<string>
#include "person.h"
#include "student.h"
using namespace std;

student::student() : person() {
  education  = "Kinder garten";
  lastCourse = "";
  lastGrade  = 0;
  totPoints  = 0.0;
}

student::student(string fn, string ln, string edu )
  : person(fn, ln) {
  education  = edu;
  lastCourse = "";
  lastGrade  = 0;
  totPoints  = 0.0;
}

student::~student() { }

void student::report() {
  person::report();
  cout << " studerar " << education << endl;
  cout << " har totalt " << totPoints << " poäng" << endl;
}

void student::input() {
  person::input();
  cout << "Ange utbildning för " << fname << " : ";
  getline(cin, education);
  cout << "Ange poäng för "
       << fname <<  ": ";
  cin >> totPoints;
  cin.get();
  cout << "Ge senaste kurs : ";
  getline(cin, lastCourse);
  cout << " Ange poäng : ";
```

```
  cin >> lastGrade;
  cin.get();
}

void student::newCourse( string cName, int grade, double points ) {
  lastCourse = cName;
  lastGrade  = grade;
  totPoints  += points;
}

double student::totalPoints() {
  return totPoints;
}

int student::lastgrade() {
  return lastGrade;
}
```

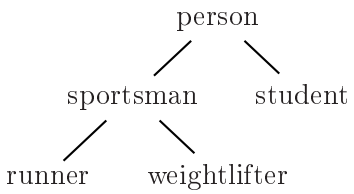
Vi har lagt till en möjlighet att lägga in kurser med studiepoäng och betyg.

Ett program kan se ut så här:

Exempel 11.12 Användning av studentklassen

```
// Filnamn : .../studentTest.cc
#include <iostream>
#include "person.h"
#include "student.h"
using namespace std;

int main() {
  student flitig( "Hermione", "Granger", "häxkonst" );

  flitig.newCourse( "Förvandlingskonst", 3, 5.3 );
  flitig.newCourse( "Programmeringsteknik II ", 4, 4.9 );
  flitig.newCourse( "Genusvetenskap", 5, 3.5 );
  flitig.report();

  return 0;
}
```

Situationen i vår hierarki är nu följande:



11.5 Objekts typkompatibilitet

Objekt i en klasshierarki kan tilldelas varandra efter särskilda regler. Låt oss titta på ett exempel från vår klasshierarki:

Exempel 11.13 Exempel med tilldelning mellan objekt

```
// Filnamn : .../typeCompatibility.cc

#include <iostream>
#include "person.h"
#include "sportsman.h"
#include "runner.h"
#include "student.h"
using namespace std;

int main() {
  person    p("P","P");
  sportsman sp("Sp", "Sp", "", 0);
  runner    ru("Ru", "Ru", "", 0, 0.0);
  student   st("S", "S", "MBL");

  // These assignments are OK
  p  = sp;            // en nivå upp
  p  = ru;            // två nivåer upp
  p  = st;            // en nivå upp
  sp = ru;            // en nivå upp

  // These assignments are NOT OK
  sp = p;
  ru = sp;
  ru = p;
  st = sp;
  return 0;
}
```

Regeln är att det är tillåtet att tilldela uppåt i hierarkin. Detta kan i början verka lite konstigt eftersom arven går åt andra hållet. Förklaringen är att tilldelning nedåt blir en del av objektet odefinierad

När man försöker tilldela ett `sportsman`-objekt värdet av ett `person`-objekt skulle fälten `fname` och `lname` kunna kopieras men däremot har de övriga fälten i `sportsman` ingen motsvarighet i `person`. C++ vägrar att lämna ett fält odefinierat och därför är en tilldelning nedåt ej tillåten.

Om vi tittar på motsatsen, en tilldelning uppåt, så gäller att det inte finns mottagare för alla fält. Det finns helt enkelt ingen plats i `person` att lagra t ex konditionsindex. Tilldelningen går bra att göra men den information som fanns i dessa fält är förlorade (i mottagaren — högerledet påverkas naturligtvis inte).

På samma sätt som det är tillåtet att tilldela objekt uppåt i trädet är det tillåtet att tilldela pekarvariabler uppåt:

```
person    *p;
sportsman *s = new sportsman();
p = s;       // Uppåt är OK
s = p;       // Nedåt är FEL
```

Här är det inga problem med för mycket eller för lite information eftersom det bara är adresser som vi bollar med. Att det ändå inte är tillåtet att tilldela en `person`-pekare till en `sportsman`-pekare beror på att man inte vet om personen är en sportsman med en sportsman alltid är en person.

11.6 Repetition så långt i kapitlet

- Vi har talat om arv mellan klasser (inheritance på engelska). Arv är den andra huvudprincipen i objektorienterad programmering.
- Vi har sett att hierarkier av klasser kan skapas. Denna process skall inte vara konstlad utan avspegla en naturlig är en relation mellan olika klasser.
- En arvinge ärver alla stamfaderns egenskaper, datafält och metoder.
- Hur arvet går till med avseende på skydd av medlemmar beror på deklarationen av arvet (`public`, `protected` eller `private`).
- Konstruktorer och destruktorer anropas uppåt i trädet när en arvinge skall skapas. Vi kan själv avgöra vilken konstruktor som skall användas. Däremot har vi inget val vad avser destruktor eftersom det endast finns en per klass.
- Metoder kan vi efter eget tycke omdefiniera om de skall göra något ”mer” än motsvarande metod hos stamfadern. En god ide är att utnyttja stamfaderns metod till det som den kan och inte återuppfinna hjulet i varje generation!
- Metoder som inte omdefinieras kan användas precis som de definierades hos stamfadern.

11.7 Virtuella metoder

Nu skall vi prata om ett begrepp som är mycket kraftfullt men kan vara lite svårt att ta till sig på en gång. Betrakta följande program som hanterar pekare helt riktigt:

Exempel 11.14 Program som hanterar pekare till objekt

```
// Filnamn : .../nonVirtual.cc
```

```
#include <iostream>
#include "sportsman.h"
#include "student.h"
using namespace std;

int main() {
  person *p;

  int ans;
  cout << "Skapa en idrottare eller en student? " << endl;
  cout << "Svara 1 för idrottare och 2 för student: ";
  cin >> ans;

  if (ans==1)
    p = new sportsman("Calle", "Kula", "kulstötning", 0);
  else
    p = new student("Ginny", "Weasly", "häxkonst" );
  p->report();
  delete p;
  return 0;
}
```

Användaren av programmet avgör när programmet körs om det skall skapa en idrottare eller en student. När objektet är skapat skriver vi ut en rapport om det. Frågan är vad som kommer att skrivas ut. Det ser ut på följande sätt om vi svarar 1:

```
Skapa en idrottare eller en student?
Svara 1 för idrottare och 2 för student: 1
Välkommen Calle
Välkommen sportsman Calle
Calle Kula
Ajöss Calle Kula
```

och på följande sätt om vi svarar 2:

```
Skapa en idrottare eller en student?
Svara 1 för idrottare och 2 för student: 2
Välkommen Ginny
Ginny Weasley
Ajöss Ginny Weasley
```

Som vi ser används rätt konstruktor i båda fallen vilket är naturligt eftersom vi i programmet explicit talar om vilken konstruktor som skall användas i de två olika fallen. Däremot blir anropet `p->report()` i båda fallen till `person::report()` och destruktorn som anropas är även den bara `~person()` och inte vad vi hoppas på `~sportsman()` respektive `~student()`.

Förklaringen ligger i hur ett programmeringsspråk och dess kompilator anropar funktioner. Kompilatorn måste avgöra vilken `report()` som skall användas. Det kan inte C++ göra eftersom tillräckligt med information inte finns vid skrivandet

av programmet. Kompilatorn kan omöjligt avgöra om `sportsman::report()` eller `student::report()` skall användas. Kompilatorn tar nu det säkra före det osäkra och anropar `person::report()` eftersom den metoden alltid kan användas, oberoende hur `*p` skapas. På samma sätt kan kompilatorn inte avgöra vilken destruktor som skall anropas. Fel anropad destruktor kan vara förödande om `sportsman` eller `student` skulle skapa några dynamiska variabler. Dessa skulle i så fall inte tas om hand av sina respektive destruktorer. Det sätt på vilken kompilatorn måste ta beslut när programmet skrivs kallas för statisk bindning (eng. "early binding"). Alla beslut om vilken version av metoden som skall anropas tas av kompilatorn innan programmet egentligen körs.

Det finns dock en lösning på detta problem. Metoder kan deklareras som *virtuella*. Beslutet om vilken metod som skall anropas skjuts då upp till körningstillfället. Man säger att bindningen görs *dynamiskt*

Hur gör vi då för att en metod skall vara virtuell? Jo, vi ändrar i persondeklarationen på följande sätt:

Exempel 11.15 Deklaration av person med virtuella metoder

```
// Filnamn : .../virtual/person.h
#ifndef __person__
#define __person__
#include<string>
using namespace std;

class person {
 public:
  person();
  person(string fn, string ln);
  ~person();

  virtual void report();
  virtual void writeName();
  virtual void input();

 protected:
  string fname;
  string lname;
};
#endif
```

Det enda vi har gjort är att lägga till nyckelordet `virtual` före de metoder vi vet kan omdefinieras av arvingarna. I arvingarna behövs inte motsvarande metoder deklareras virtuella, en gång virtuell alltid virtuell. Samma program som tidigare ger följande utskrift beroende på vad man svarar:

```
Skapa en idrottare eller en student?
Svara 1 för idrottare och 2 för student: 1
Välkommen Calle
Välkommen sportsman Calle
Calle Kula
och utövar kulstötning.
Calle har 0 förstapriser.
Ajöss Calle Kula
```

samt

```
Skapa en idrottare eller en student?
Svara 1 för idrottare och 2 för student: 2
Välkommen Ginny
Ginny Weasley
at häxkonst
har totalt 0 poäng
Ajöss Ginny Weasley
```

Vi kan se att rätt version av `report()` anropas i respektive

Nu kan vi fråga oss varför man inte alltid har metoder som virtuella. Svaret är att den dynamiska bindningen kostar lite datorkraft. Programmet måste ha någon tabell att hitta rätt metod samt något sätt att identifiera `*p` riktigt vilket kostar lite tid och datorminne. Betoningen ligger här på lite, så om det verkar lämpligt ska man inte tveka att deklarera sina metoder som virtuella.

Vi har tidigare sett att `report()` har många olika former beroende för vilket objekt `report()` skall utföras. Många former blir på grekiska *polymorfi*, vilket är objektorienteringens tredje huvudprincip. Äkta polymorfi stöds av virtuella metoder i C++.

11.8 Exempel med ett personregister

Nu ska vi använda vår hierarki för att bygga ett personregister. Vi deklarerar en klass `preg` (personregister) på följande sätt:

Exempel 11.16 Deklarationsfil till personregister

```
// Filnamn: .../virtual/preg.h
#ifndef __preg__
#define __preg__
#include "person.h"

const int maxPerson = 100;

class preg {
protected:
  int numPersons;           // Verkligt antal personer
  person *pers[maxPerson];  // array med pekare till personer
```

```
public:
  preg();
  ~preg();
  void reportAll();
  void inputAll();
  void insertPerson(person *p);
};
#endif
```

Klassen `preg` har ett datafält `numPersons` som talar om hur många personer som finns i registret. Arrayen `pers` (som består av pekare till personer) lagrar de aktuella personerna. I sin nuvarande form klarar registret som mest 100 personer. Vidare finns det en konstruktor och en destruktor deklarerade som vi strax skall titta på. Metoden `reportAll` skall skriva ut en rapport för alla i registret förekommande personer och motsvarande för metoden `inputAll`. Metoden `insertPerson` tar en pekare till en person som inparameter och stoppar in denne i registret.

Låt oss se implementationen av dessa metoder.

Exempel 11.17 Implementation av klassen `preg`

```
// Filnamn: .../virtual/preg.cc
#include<iostream>
#include<string>
#include "person.h"
#include "preg.h"

preg::preg() {     // Konstruktor
  numPersons = 0;
  for (int i=1; i<=maxPerson; i++)
    pers[i-1] = 0;  // Pekare till ingenting
}

preg::~preg() {     // Destruktor
  for (int i=1; i<=numPersons; i++ )
    delete pers[i-1];
}

void preg::reportAll() {
  for (int i=1; i<=numPersons; i++ ) {
    cout << "=============================================" << endl;
    cout << "Person " << i << " : " << endl;
    pers[i-1]->report();
  }
}

void preg::inputAll() {
  for (int i=1; i<=numPersons; i++ )
    pers[i-1]->input();
```

```
}

void preg::insertPerson(person *p) {
  numPersons++;
  if ( numPersons > maxPerson ) {
    cout << "Fö många personer! Programmet avbryts." << endl;
    exit(2);
  }
  pers[numPersons-1] = p;
}
```

- Konstruktorn `preg::preg` sätter antalet aktuella personer till 0 och alla pekare till 0.
- Destruktorn `preg::~preg` gör `delete` på alla pekare.
- Metoden `preg::reportAll` skriver ut en rapport för alla i registret ingående personer genom att anropa respektive `report`-metod. Observera att anropen fungerar tack vara att `report`-metoden är deklarerad som virtuell i person.
- Metoden `preg::insertPerson` tar emot en pekare till en person (eller någon arvinge).

Låt oss se ett program som utnyttjar `preg`.

Exempel 11.18 Program som använder `preg`

```
// Filnamn : .../virtual/pregTest.cc

#include <iostream>
#include <string>
#include "person.h"
#include "sportsman.h"
#include "runner.h"
#include "student.h"
#include "preg.h"

using namespace std;

int main() {
  preg myRegister;
  person *tempP;
  student *tempS;

  tempP = new person("Small", "Person" );
  myRegister.insertPerson( tempP );

  tempS = new student("Ginny", "Weasly", "häxkonst" );
  tempS->newCourse( "C++ kursen", 5, 4.8 );
  myRegister.insertPerson( tempS );
```

```
  tempP = new sportsman("Calle", "Kula", "kulstötning", 0 );
  myRegister.insertPerson( tempP );

  tempP = new runner("Gunder","Hägg", "löpning", 0, 0. );
  myRegister.insertPerson( tempP );

  myRegister.reportAll();
  return 0;
}
```

Programmet skapar objekt av typen person eller av person:s arvingar och lägger in dessa i `myRegister`. Sedan anropas metoden `reportAll()` som skriver ut alla rapporter på det sätt som vi har tänkt oss, se körexempel nedan.

```
  ================================================
  Person 1 :
  Small Person
  ================================================
  Person 2 :
  Ginny Weasley
  studerar häxkonst
  har totalt 4.8 poäng
  ================================================
  Person 3 :
  Calle Kula
  och utövar kulstötning.
  Calle har vunnit 0 förstapriser.
  ================================================
  Person 4 :
  Gunder Hägg
  och utövar löpning.
  Gunder har vunnit 0 förstapriser.
  Konditionsndex : 0
```

Vad som nu skall noteras är:

1. Klassen register har ingen vetskap om att person har arvingar. Om vi tittat i `preg.h` inkluderas enbart `person.h`. Ändå hanterar preg `report`-metoderna helt riktigt! Detta beror på den virtuella deklarationen av `report` i `person`.

2. Vi kan om vi vill lägga till nya arvingar i den existerande hierarkin. Absolut ingenting behöver ändras i `preg` för att `preg` skall kunna ta hand om de nya klasserna.

3. Vi skapar objekt med new i main innan vi skickar dem till `myRegister`. Vi behöver inte (och skall inte!) göra `delete` på dessa objekt i `main`. I och med överlämnandet har `myRegister` ”ansvaret” för alla personer. Destruktorn `preg::~preg` gör `delete` på alla personer i registret. Observera för att det

skall fungera riktigt måste destruktorn vara virtuell. Annars skulle alltid `persons` destruktor användas.

11.9 Aggregat av klasser

Ofta är arvsrelationer ett bra sätt att beskriva samband. Ibland fungerar det dock inte så bra. Låt oss titta på ett exempel där vi deklarerar två klasser för delar till en bil, `tire` och `engine` (däck och motor).

Exempel 11.19 Deklarationsfil till tire-klassen

```
// Filnamn .../tire.h
#ifndef __tire__
#define __tire__

class tire {
protected:
  double diameter;
  double patternDepth;
public:
  tire();
  tire( double d, double pd );
  ~tire();
};
#endif
```

Klassen `tire` är mycket enkel med bara två datamedlemmar, `diameter` som beskriver diametern på däcket och `patternDepth` som lagrar mönsterdjupet i mm. Vi har två konstruktorer och en destruktor men inga andra medlemsmetoder. Du får själv tänka dig några lämpliga metoder för att göra klassen användbar. Implementationen av klassen ser ut som:

Exempel 11.20 Implementation av klassen `tire`

```
// Filnamn : .../tire.cc
#include<iostream>
#include<string>
#include "tire.h"

tire::tire() {
  diameter = 10;
  patternDepth = 10;
}

tire::tire( double d, double pd ) {
  diameter = d;
  patternDepth = pd;
```

```
  cout << "Constructor for tire " << endl;
}

tire::~tire() {
        cout << "Destructor for tire " << endl;
}
```

Inga konstigheter, det enda vi noterar är att vi som vanligt har onödiga utskrifter för att kunna följa programmet. Snabbt vidare till engine-klassen:

Exempel 11.21 Deklarationsfil för engine-klassen

```
// Filnamn .../engine.h
#ifndef __engine__
#define __engine__

class engine {
 protected:
  double horsePower;
  double cylinders;
 public:
  engine();
  engine( double hp, double c );
  ~engine();};
#endif
```

Implementationen av densamma:

Exempel 11.22 Implementation av engine-klassen

```
// Filnamn .../engine.cc
#include<iostream>
#include<string>
#include "engine.h"

engine::engine() {
  horsePower = 10;
  cylinders = 10;
}

engine::engine( double hp, double c ) {
  horsePower = hp;
  cylinders = c;
  cout << "Constructor for engine " << endl;
}

engine::~engine() {
  cout << "Destructor for engine " << endl;
```

```
}
```

Om vi nu vill deklarera klassen `car` i C++ så är en bil varken en naturlig arvinge till `tire` eller till `engine`. Mycket enkelt kan vi säga att en bil består av en `engine` och fyra stycken `tire`. Förhållandet kan uttryckas i C++ genom att en klass kan bestå av andra objekt. Låt oss se hur en `car` kan vara deklarerad i vårt exempel.

Exempel 11.23 Deklarationsfil till klassen car

```
// Filnamn .../car.h
#ifndef __car__
#define __car__
#include "tire.h";
#include "engine.h";

class car {
 protected:
  engine en;              // one engine
  tire lf, lr, rf, rr;    // four tires
 public:
  car();
  car( double hp,
       double cyl,
       double tD,
       double tP );
  ~car();
};
#endif
```

En `car` består av en engine och fyra `tire` i exemplet. Observera relationen *består av* jämfört med relationen *är en* i tidigare exempel. En klass som består av objekt ur andra klasser kallas för ett *aggregat* av andra klasser. Implementationen av `car` ser ut som följer:

Exempel 11.24 Implementationen av klassen car

```
// Filnamn .../car.cc
#include<iostream>
#include<string>
#include "car.h"
#include "tire.h"
#include "engine.h"

car::car() : en(), lf(), lr(), rf(), rr() {}

car::car( double hp,
```

```
          double cyl,
          double tD,
          double tP ) :
  n(hp,cyl), lf(tD,tP), lr(tD,tP), rf(tD,tP), rr(tD,tP) {
        cout << "Constructor for car " << endl;
}

car::~car() {
        cout << "Destructor for car " << endl;
}
```

Vad vi noterar är att efter kolonet i konstruktorerna radas konstruktorerna för de ingående objekten upp separerade av kommatecken. För destruktorn behöver inget anges , C++ vet hur den skall göra iallafall. Låt oss slutligen se ett exempel med en car samt tillhörande körexempel.

Exempel 11.25 Exempelprogram med klassen car

```
// Filnamn : .../carTest.cc
#include <iostream>
#include "car.h"

int main() {
        car Volvo( 150,6,45,12 );
        cout << endl <<" testing the car class" << endl;
        return 0;
}
```

```
Constructor for engine
Constructor for tire
Constructor for tire
Constructor for tire
Constructor for tire
Constructor for car
testing the car class
Destructor for car
Destructor for tire
Destructor for tire
Destructor for tire
Destructor for tire
Destructor for engine
```

Som vi ser av exemplet anropas konstruktorer och destruktorer som vi har vant oss vid.

11.10 Kom ihåg

- Från förra kapitlet. OOP:ns första huvudprincip : inkapsling
- Taxonomier arvsträd - hierarkiska strukturer.
- Basklass, underklass — relationen *är en*
- OOP:ns andra huvudprincip : arv (inheritance).
- `public`, `protected` och `private`.
- Hur konstruktorer och destruktorer anropas vid arv.
- Överlagring av medlemsmetoder.
- Objekts typkompatibilitet uppåt i trädet.
- Virtuella metoder.
- OOP:ns tredje huvudprincip : polymorfa metoder (virtuella metoder).
- Early binding kontra late binding.
- En gång virtuell alltid virtuell.
- Personregisterexemplet, bra att förstå.
- Aggregat av klasser — relationen *består av*

Kapitel 12

Filhantering

I detta kapitel skall vi gå igenom hur vi på enklaste sätt kan lagra data på filer.

När vi kör en editor, t ex emacs, sparar vi vårt arbete med jämna mellanrum på en fil. Filen kan sedan läsas in i emacs på nytt eller användas av ett annat program. Med cat-kommandot listas filen på skärmen. Som du vet är cat ett unix-program som kan läsa filer.

Ett annat exempel på olika program som delar på samma fil är emacs och g++. Ni skriver C++ programmet i emacs och låter sedan g++ översätta till ett exekverbart program.

Filer är således både ett sätt att spara information mellan körningarna av ett program och ett sätt att dela information mellan olika program.

12.1 Vad är en fil?

En fil är en sekvensiellt ordnad mängd data. En fil liknar mycket en array. Skillnaderna är att:

1. En array finns i primärminnet och endast under ”runtime”, dvs den tid under vilket programmet kör. När programmet är slut försvinner arrayen ut i intet. En fil finns i sekundärminnet och dessutom permanent. När programmet slutar (t ex emacs) finns filen kvar på disken och kan användas på nytt vid ett senare tillfälle.

2. En array kan läsas och skrivas samtidigt som i följande sats:

   ```
   arr[1] = arr[2]*1.2;
   ```

   I satsen läses innehållet element 2 och skrivs i element 1. En fil kan antingen läsas eller skrivas, inte båda operationerna samtidigt.

3. I en array kan vi hoppa fram och tillbaka hur som helst med hjälp av index. I en fil kan vi bara läsa/skriva framåt.

I C++ finns det två typer av filer, textfiler och binära filer. I detta kompendium diskuterar vi endast på textfiler.

12.2 Textfiler

En textfil lagrar tecken, dvs char. Heltal et.c. konverteras automatiskt till tecken innan de lagras i en textfil. En textfil som har skapats i ett C++ program är läsbar av andra program som hanterar textfiler. Vidare har en textfil i UNIX ett osynligt tecken för att markera radslut, ett `^J` (control-J), dvs ASCII-kod 10. I andra operativsystem kan det finnas andra sätt att markera radslut. Radslutstecknet representeras dock alltid med `'\n'` i C++.

Om vi har en textfil som uppfattas som:

```
Tom
Bombadil
```

dvs två rader åtskilda av ett radslut. I en textfil (på Unix) representeras detta av sammanlagt 13 tecken inklusive två radslutstecken `'\n'`. Program som t ex emacs, cat och more tyder radslutstecknet på ett riktigt sätt.

Låt oss nu se hur vi kan skapa textfiler i C++. Vi skall skriva ett program som skriver ut ett litet meddelande på en textfil. Därefter skriver vi ett program som kan läsa in meddelandet som det första programmet skrev! Dessutom skall program 2 skriva ut innehållet på skärmen.

Exempel 12.1 Program som skriver ett meddelande på en textfil

```
// Filnamn : .../ex01.cc

#include <iostream>
#include <fstream>
using namespace std;

int main() {
  const double magicNum = 123456789.123456789;
  ofstream outFile( "exempel.dat" );

  outFile << "Hej från program ex01.cc" << endl;
  outFile << "En double ser ut så här: " << magicNum << endl;
  outFile << endl;
  outFile << "Ajöss och tack för fisken!" << endl;

  outFile.close();
  return 0;
}
```

Vi noterar följande i programmet.

1. En ny fil inkluderas, `fstream`. Denna innehåller vad som behövs för filhantering.
2. Ett objekt, `outFile`, deklareras av klasstypen `ofstream`. I konstruktorn anger vi vilket filnamn som skall användas. I `ofstream` står `o` för *output* och `f` för *file*. Vi har alltså en utström till en fil.
3. I de sju satserna som börjar med `outFile <<` skrivs olika ”saker” ut till filen `exempel.dat`.
4. I den fjärde satsen skriver vi ut en `double`. Denna omvandlas till alfanumeriska tecken. Vad som skall observeras är att på en textfil har vi den läsbara representationen av talet `magicNum`, inte de 8 byte som egentligen utgör `magicNum`.
5. Om `outFile` hade bytts ut mot `cout` hade vi fått meddelandet till skärmen istället!
6. Filen stängs för vidare operationer med `outFile.close();`
7. När programmet är klart har vi en ny fil som heter `exempel.dat`.

Filen `exempel.dat` ser nu ut på följande sätt:

```
Hej från program ex01.cc
En double ser ut så här: 1.23457e+08

Ajöss och tack för fisken!
```

Som synes fick vi en liten speciell representation av talet `magicNum`. Utskriften bestäms av `outFile` på precis samma sätt som `cout` gör. Utskriften kan också formateras med exempelvis `outFile.width(12)`.

Nå, nu kan vi skriva ut vad vi vill på en textfil. Låt oss nu se ett program som kan läsa in meddelandet.

Exempel 12.2 Program som skriver en textfil på skärmen

```
// Filnamn : .../ex02.cc

#include <iostream>
#include <fstream>
using namespace std;

int main() {
  ifstream inFile( "exempel.dat" );

  cout << "Filen 'exempel.dat' har följande innehåll:" << endl;
  char ch;

  while ( inFile.get(ch) ) {
    cout << ch;
  }
```

```
  cout << "=== end of file ===" << endl;
  return 0;
}
```

Vi noterar följande i programmet.

1. Vi inkluderar samma fil, `fstream` som i förra exemplet.
2. Ett objekt, `inFile`, av klassen `ifstream` deklareras. Med konstruktorn kopplas `inFile` till diskfilen `exempel.dat`. Bokstaven `i` i `ifstream` står för `i`nput.
3. Därefter läser programmet in tecken efter tecken från filen med `inFile.get()`. Metoden läser in ett tecken och returnerar samtidigt 1 om det gick bra och 0 om det gick dåligt. Med dåligt menas att det inte fanns något tecken kvar att läsa in. While-satsen fortsätter således att läsa in tecken så länge som det finns något kvar i filen.
4. Varje tecken som läses in skrivs ut med på skärmen. Vi får en exakt kopia av filen `exempel.dat` på vår skärm då programmet kör.
5. Det ser ut som jag har glömt att stänga filen i exemplet. Filen stängs automatiskt av destruktorn i `ifstream` om vi inte stänger den själva med `inFile.close()`.

I exemplet läste vi tecken för tecken. Hela rader kan också läsas in från filen på samma sätt som från tangentbordet.

Exempel 12.3 Läsning av hela rader från en textfil

```
// Filnamn : .../ex02b.cc

#include <iostream>
#include <fstream>
using namespace std;

int main() {
  ifstream inFile( "exempel.dat" );
  string line;

  while ( getline(inFile, line) ) {
    cout << line << endl;
  }
  inFile.close();
  return 0;
}
```

12.3 Bank på fil

Vi skall nu återknyta till bank-exemplet i kapitel 9. Ett väsentligt krav på bankprogrammet är naturligtvis att informationen om kontona sparas på filer. Annars går ju all information förlorad om datorn t ex stängs av. För att göra det enkelt för banktjänstemannen så skall vi låta programmet automatiskt läsa in data från en fil när programmet börjar och spara det på en fil när det slutar. Vi kan alltså låta konstruktorn respektive destruktorn (vi hade ingen tidigare) för klassen `Bank` ombesörja detta. Således:

```
Bank::Bank() {
  noOfAcc = 0;        // Inga konton från början
  name    = "TDBank";
  cout << "Läser från fil..." << endl;
  load("bank.dat");
  cout << noOfAcc << " konton lästa" <<endl;
}

Bank::~Bank() {
  cout << "Sparar på fil..." << endl;
  save("bank.dat");
  cout << noOfAcc << " konton sparade" << endl;
}
```

Förutom utskrifterna som berättar vad som pågår så har vi lagt anrop till två hjälpfunktioner för att läsa respektive skriva. Eftersom det är funktionen som *skriver* filen som bestämmer formatet så tittar först på den:

```
void Bank::save(string fileName) {
  ofstream  ofs(fileName.c_str());
  ofs << noOfAcc << '\n';
  for (int i=0; i<noOfAcc; i++)
    ofs << accounts[i].getName() << '\t' << accounts[i].getBalance() << '\n';
}
```

Först på filen skrivs antalet konton ut på en rad. Därefter skrivs konto för konto ett per rad. Vid deklarationen av `ofs` vill vi ge ett filnamn som vi fått som parameter i form av ett `string`-objekt. Konstruktorn för `ofstream` förväntar sig dock en pekare till en teckenarray som parameter. För att åstadkomma detta använder vi metoden `c_str()` i klassen `string` som returnerar just en sådan pekare.

När vi nu vet hur filen ser ut kan vi läsa den med följande funktion:

```
void Bank::load(string fileName) {
  ifstream  ifs(fileName.c_str());
  if (ifs.good()) { // Test om filen hittades
    ifs >> noOfAcc;
    ifs.get();
    for (int i=0; i<noOfAcc; i++) {
      ifs >> accounts[i].name >> accounts[i].balance;
    }
```

```
  } else
    cout << "Ingen indatafil hittades" << endl;
}
```

Eftersom det inte är säkert att filen finns använder vi metoden `good()` för att testa om filöppningen gick bra. I så fall läser vi först antalet och därefter så många konton som antalet anger.

En detalj som borde uppröra läsaren: Eftersom det inte finns några `set`-metoder för namn och saldo i `Account`-klassen så refererar vi dessa attribut direkt! Det får vi ju inte göra eftersom de är deklarerade som skyddade. I stället för att skriva två sådana metoder har vi här valt att lägga deklarationen

```
friend class Bank;
```

i definitionen av `Account`-klassen. Med detta ger vi klassen `Bank` en exklusiv rättighet att använda `Account`-klassens alla skyddade komponenter.

Friend-deklarationer skall endast användas i fall som detta när två klasser hör intimt samman (och kanske inte ens då).

Övning

1. I ett riktigt system skulle man inte vara nöjd med att spara dagens saldo när man slutar programmet. Om det t ex blir ett strömavbrott förloras ju allt som skett under denna körning. Förse programmet med en `logg-fil` där varje transaktion skrivs ut. Skriv de på ett sådant sätt att det lätt går att låta programmet läsa filen och därmed återskapa allt som skett. Tillfoga också en sådan operation till menyn.

12.4 Några avslutande anmärkningar

Det finns inget som kan gå så galet när man programmerar som just filhantering. I detta kapitel har allt gått bra, filerna har funnits, de har innehållit vad jag förväntade mig o.s.v. Detta är dock inte normalfallet och det finns därför ett antal metoder för att testa hur t.ex. en filläsning gick. Dessa metoder heter `fail()`, `good()` och `bad()`. Det finns också en metod som testar filslut, `eof()`. Om du vill veta mer om dessa metoder kan du antingen titta i ett referensverk eller göra den laboration om filhantering som finns till detta kompendium.

12.5 Kom ihåg

- Textfiler, precis som vanlig I/O från tangentbord och till skärm.
- Utfiler är objekt av klassen ofstream.
- Infiler är objekt av klassen ifstream.
- `good()`, `bad()`, `fail()`, `eof()`.

Sakregister

Sakregister