

## Innehållsförteckning

<b>2</b>	<b>INTRODUKTION TILL KLASSER OCH OBJEKT</b>	<b>2</b>
<b>2.1</b>	<b>Inledning</b>	<b>2</b>
2.1.1	Klassdefinition	3
2.1.2	Objekt	3
2.1.3	Inkapsling	4
2.1.4	Medlemsfunktioner	5
2.1.5	Mitt första program med objekt	5
<b>2.2</b>	<b>Set- och getfunktioner</b>	<b>7</b>
<b>2.3</b>	<b>Sammanfattning</b>	<b>11</b>
<b>2.4</b>	<b>Konstruktör, destruktör</b>	<b>12</b>
2.4.1	Skapa konstruktörer och destruktör	12
2.4.2	Användning av konstruktörerna och destruktorn	13
<b>2.5</b>	<b>Placering av klasser i header- och definitionsfil</b>	<b>16</b>
2.5.1	Klientprogram	16
2.5.2	Headerfil	17
2.5.3	Definitionsfil	18
<b>2.6</b>	<b>Skapa en array av objekt</b>	<b>19</b>
<b>2.7</b>	<b>Inline-funktioner i klasser och initieringslistor</b>	<b>21</b>
<b>2.8</b>	<b>Att använda en klass i en annan klass</b>	<b>26</b>
2.8.1	Ett program som använder klassen Flight	28
<b>2.9</b>	<b>Stack som en abstrakt datatyp</b>	<b>31</b>
<b>2.10</b>	<b>Använd stacken i ett program</b>	<b>33</b>

## 2 Introduktion till klasser och objekt

Vi ska nu börja med objektorienterad programmering (OOP). OOP innebär att man får tänka i andra banor än tidigare. Funktionerna byts ut mot objekt. Vad är då ett objekt? Ett sätt är att säga att ett objekt är en enhet som innehåller både data och de operationer som man vill utföra på dessa data. Så kan t.ex. ett objekt vara ett bankkonto vars data är saldo, kontonummer, ägarens namn och adress mm. Vad utför man för operationer på ett bankkonto? De mest grundläggande operationerna är ju att göra insättningar och uttag.

Alltså kan ett objekt vara en enhet som innehåller data i form av saldo, kontonummer, namn och som kan utföra operationer i form av insättning och uttag av pengar.

### 2.1 Inledning

Objektorienterad programmering är en del av det objektorienterade tankesättet. Ska man jobba helt objektorienterat måste man göra

- objektorienterad analys
- objektorienterad design
- objektorienterad programmering
- testning

inom OOP kommer vi att träffa på begrepp som

- klass
- objekt
- datamedlem
- medlemsfunktion
- abstrakt datatyp
- instans
- inkapsling
- arv
- public
- private
- mm

Jag börjar med ett exempel på hur man gör en klass för att hantera ett bankkonto för att du ska få något att hänga upp begreppen på.

Antag att man vill hantera följande data för ett bankkonto:

kontonummer  
förnamn  
efternamn  
saldo

För att hantera dessa data som en enhet har vi tidigare lärt oss att göra en struktur:

```
struct Konto
{
    string kontoNummer;
    string forNamn;
    string efterNamn;
    long saldo;
};
```

Med hjälp av denna struktur kunde vi sedan skapa en variabel som innehåller information för alla datamedlemmarna (kontonummer, förnamn, efternamn och saldo):

```
Konto konto;
```

Åtkomst av datamedlemmarna görs med hjälp av punktnotation:

```
konto.forNamn = "Ola";  
konto.efterNamn = "Karlsson";
```

I C++ är en struct och en klass så gott som samma sak. Det vi inte utnyttjade i structen var att tala om vilka operationer som skulle utföras på dessa data.

Exempel på operationer som kan utföras på ett bankkonto:

- göra insättning
- göra uttag
- ange namn
- ange kontonummer
- visa kontobesked

Vi ska nu göra en klass i vilken vi kan lagra data enligt strukturen Konto och dessutom i klassen ta med funktioner med vilka vi kan göra insättningar och uttag, ange namn och kontonummer mm.

### 2.1.1 Klassdefinition

När man vill skapa en klass skriver man en klassdefinition:

```
// Klassdefinition  
class Account  
{  
    private:                                // kan bara nås via medlemsfunktioner  
        string firstName;                  // datamedlem  
        string lastName;  
        string accountNr;  
        int balance;  
  
    public:                                // definierar publikt gränssnitt  
        void deposit(int amount);          // medlemsfunktion  
        void withdrawal(int amount);       // medlemsfunktion  
        void showAccount();                // medlemsfunktion  
};                                          // avslutar definitionen av klassen Account
```

Vi har nu gjort en klass för att hantera konton. Man kan se klassen som en egentillverkad datatyp.

### 2.1.2 Objekt

När vi vill använda klassen gör vi ett objekt (= en instans) av klassen:

```
Account account
```

Objektet kan jämföras med en variabel av en enkel datatyp (int tal);

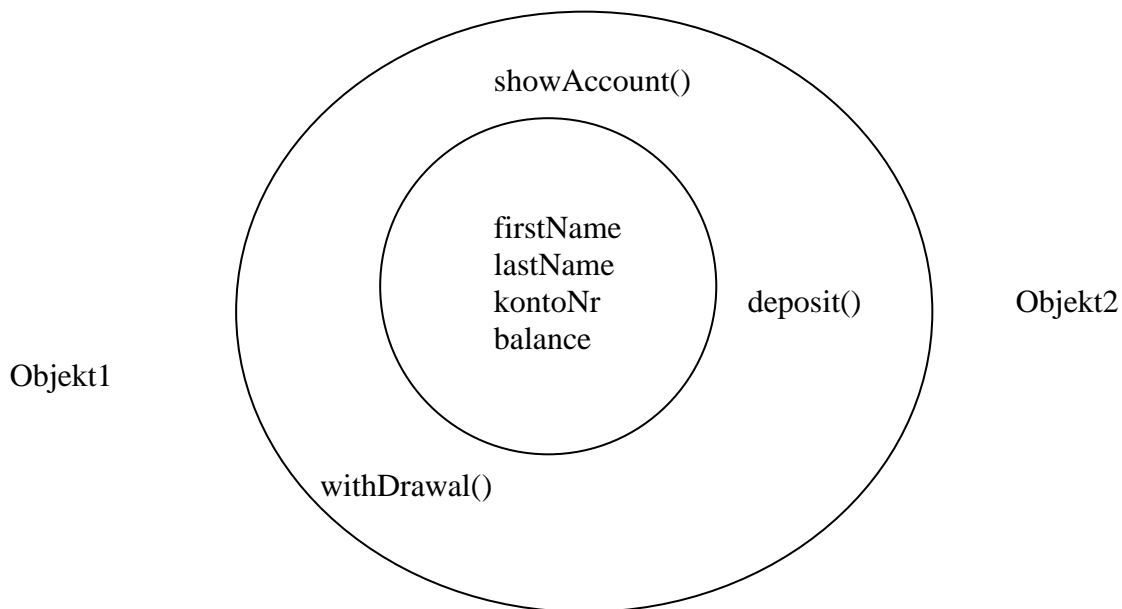
Användning av objektet görs med medlemsfunktionerna:

```
account.deposit(1000);  
account.showAccount();
```

Hur dessa fungerar ska vi visa senare.

### 2.1.3 Inkapsling

Det är inte möjligt att direkt använda de privata medlemsvariablerna. Det är en av de grundläggande tankarna inom OOP att datamedlemmarna ska vara inkapslade, d.v.s. man ska inte ha direktåtkomst till dem. Man kan alltså inte skriva: `account.firstName = "Kalle";`



Figuren visar en kontoklass med medlemsvariabler som man bara kan komma åt via medlemsfunktionerna. I klientprogrammen gör man objekt av klassen för att kunna hantera ett konto.

En liten sammanfattning av det vi gått igenom:

klass	
datamedlemmar	(medlemsvariabler)
medlemsfunktioner	(funktionsmedlem, metod)
private	åtkomliga enbart inom klassen
public	åtkomliga från klientprogrammet
objekt	(instans)

## 2.1.4 Medlemsfunktioner

Implementering av medlemsfunktionerna:

```
//-----  
// Definition av medlemsfunktioner (metoder)  
//-----  
void Account::deposit(int amount)  
{  
    if(amount > 0)                //Kolla att insättningsbeloppet > noll  
        balance += amount;        //Addera beloppet till saldot  
}  
  
void Account::withdrawal(int amount)  
{  
    if( amount>0 && amount <= balance) // Kolla att belopp>0 OCH belopp < saldot  
        balance -= amount;          // Dra belopp från saldot  
}  
  
void Account::showAccount()        // Skriv kontobesked  
{  
    cout << "KONTOBESKED" << endl;  
    cout << "Namn    : " << firstName + " " + lastName << endl;  
    cout << "KontoNr: " << accountNr << endl;  
    cout << "Saldo   : " << balance << endl << endl;  
}
```

### Kommentarer:

- :: = räckviddsoperatoren
- void Account::showAccount() innebär att showAccount hör till klassen Account och kan använda de datamedlemmar och medlemsfunktioner som hör till Account.

## 2.1.5 Mitt första program med objekt

Ett program som använder klassen Account:

```
int main()  
{  
    Account account, a1;        // Skapa objekt (en instans) av klassen Account  
    account.deposit(1000);       // Gör en insättning  
  
    a1.deposit(2000);  
    account.showAccount();       // Skriv kontobesked  
    a1.showAccount();  
    account.withdrawal(300);     // Gör ett uttag  
    account.showAccount();       // Skriv kontobesked  
    a1.withdrawal(300);         // Gör ett uttag  
    a1.showAccount();  
    getch();  
    return 0;  
}
```

Hela filen med klassdeklaration, implementering av medlemsfunktionerna och ett klientprogram ser ut så här:

```
// class_010 Version 10  
// Ett första försök med klasser och objekt  
// Gör en klass för ett bankkonto  
// Per Ekeroot 2014-01-13  
//-----
```

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

//-----
// Klassdefinition
//-----
class Account
{
private:
    string firstName;    // kan bara nås via medlemsfunktioner
    string lastName;     // datamedlem
    string accountNr;    // datamedlem
    int balance ;        // datamedlem
public:
    void deposit(int amount); // definierar publikt gränssnitt
    void withdrawal(int amount); // medlemsfunktion
    void showAccount(); // medlemsfunktion
}; // avslutar definitionen av klassen Account

//-----
// Huvudprogram
//-----
int main()
{
    Account account, al; // Skapa ett objekt (en instans) av klassen Account
    account.deposit(1000); // Gör en insättning
    al.deposit(2000);

    account.showAccount(); // Skriv kontobesked
    al.showAccount();

    account.withdrawal(300); // Gör ett uttag
    account.showAccount(); // Skriv kontobesked

    al.withdrawal(300); // Gör ett uttag
    al.showAccount();

    return 0;
}

//-----
// Definition av medlemsfunktioner
//-----
// deposit - sätt in pengar på kontot
//-----
void Account::deposit(int amount)
{
    if(amount > 0) //Kolla att insättningsbeloppet > noll
        balance += amount; //Addera beloppet till saldot
}

//-----
// withdrawal - ta ut pengar från kontot
//-----
void Account::withdrawal(int amount)
{
    if( amount>0 && amount <= balance) // Kolla att belopp>0 OCH belopp < saldot
        balance -= amount; // Dra belopp från slادت
}
}
```

```
//-----  
// Skriv kontobesked på skärmen  
//-----  
void Account::showAccount()  
{  
    cout << "Account statement" << endl;  
        cout << "Name      : " << firstName << " " << lastName << endl;  
    cout << "AccountNr: " << accountNr << endl;  
    cout << "Balance   : " << balance << endl << endl;  
}
```

#### Kommentarer:

- testa cout << account.firstName; Vad händer?

Körning av programmet ger följande utskrift:

Utskriften ser inte så bra ut! Det verkar som om medlemsvariabeln balance inte nollställs och att firstName, lastName och accountNr (strings) är tomma.



## 2.2 Set- och getfunktioner

För att råda bot på att medlemsvariablerna inte har nollställts bygger vi på klassen med medlemsfunktioner som kan ge medlemsvariablerna värden, s.k. set-funktioner. Vi passar också på att skapa funktioner med vilka vi kan läsa medlemsvariablernas värden, s.k. get-funktioner.

Den utökade klassdefinitionen ser ut så här (class\_020):

```
class Account  
{  
    private:  
        string firstName;           // datamedlemmar  
        string lastName;  
        string accountNr;  
        int balance;  
  
    public:  
        void deposit(int amount);   // medlemsfunktioner  
        void withdrawal(int amount);  
  
        void setFirstName(string aFirstName); // Medlemsfunktioner som sätter  
        void setLastName(string aLastName);   // värden på medlemsvariabler  
        void setAccountNr(string aAccountNr);  
        void setBalance(int aBalance);  
  
        string getFirstName() const;          // Medlemsfunktioner som läser  
        string getLastName() const;           // medlemsvariablernas värden  
        string getAccountNr() const;  
        int getBalance() const;  
};                                             // avslutar definitionen av klassen Account  
  
void showAccount(Account const &account);
```

Implementation av set- och getfunktionerna:

```
//-----  
// setFirstName  
// Datamedlemmen firstName ges värdet aFirstName  
//-----  
void Account::setFirstName(string pFirstName)  
{  
    firstName = pFirstName;  
}
```

**Kommentarer:**

- Antag att vi har gjort ett konto: `Account account`. Då kan vi tilldela objektet `account` förnamnet med set-funktionen så här: `account.setFirstName("Ulla");`
- Argumentet `aFirstName` ges värdet "Ulla". Datamedlemmen `firstName` får sedan också värdet "Ulla" genom tilldelningen `firstName = aFirstName`
- Datamedlemmen `firstName` får inte ha samma namn som parametern `pFirstName`. Jag har valt att skilja dem åt genom att sätta ett "p" (=parameter) först i parameterns namn.

```
//-----  
// setLastName  
// Datamedlemmen lastName ges värdet pLastName  
//-----  
void Account::setLastName(string pLastName)  
{  
    lastName = pLastName;  
}  
//-----  
// setAccountNr  
// Datamedlemmen accountNr ges värdet aAccountNr  
//-----  
void Account::setAccountNr(string pAccountNr)  
{  
    accountNr = pAccountNr;  
}  
//-----  
// setBalance  
// Datamedlemmen balance ges värdet aBalance  
//-----  
void Account::setBalance(int pBalance)  
{  
    balance = pBalance;  
}  
//-----  
// getFirstName  
// Returnera datamedlemmen firstName (string)  
//-----  
string Account::getFirstName() const           // Läs medlemsvariablers värden  
{  
    return firstName;  
}  
//-----  
// getLastName  
// Returnera datamedlemmen lastName (string)  
//-----  
string Account::getLastName() const  
{  
    return lastName;  
}
```



```
//-----  
// getAccountNr  
// Returnera datamedlemmen accountNr (string)  
//-----  
string Account::getAccountNr() const  
{  
    return accountNr;  
}  
//-----  
// getBalance  
// Returnera datamedlemmen balance (int)  
//-----  
int Account::getBalance() const  
{  
    return balance;  
}  
//-----  
// showAccount  
// Skriv ut objektets aktuella data på skärmen  
// Funktionen tillhör INTE klassen  
//-----  
void showAccount(Account const &account)  
{  
    cout << endl;  
    cout << "KONTOBESKED" << endl;  
    cout << "Namn      : " << account.getFirstName() + " " + account.getLastName()  
    << endl;  
    cout << "KontoNr: " << account.getAccountNr() << endl;  
    cout << "Saldo   : " << account.getBalance() << endl << endl;  
}
```

### Kommentarer:

- const efter funktionen ser till att funktionen inte kan ändra värdet på datamedlemmen.
- Funktionen *showAccount()* har nu flyttas så att den är placerad utanför klassen. Anledningen till detta är att man vill att klassen ska vara så **generell** som möjligt. Placerar man *showAccount()* i klassen så lägger man in speciella formateringar i klassen. Det är bättre att göra formateringen i en funktion utanför klassen.
- Parametern i *showAccount()* är referensdeklarerad av effektivitetsskäl. När parametern är referensdeklarerad skickas enbart referensen (=adressen) till objektet och befintliga data används av funktionen. Vid värdeanrop, *showAccount(Account account)*, kopieras objektet vilket kan vara resurskrävande om objekten är stora.

### Klientprogrammet blir nu

```
int main()
{
    // Skapa ett objekt (en instans) av klassen Account
    Account account;
    showAccount(account);

    // Initiera objektet
    account.setBalance(0);
    account.setAccountNr("AA1-122");
    account.setFirstName("Eva");
    account.setLastName("Svensson");
    showAccount(account);
    cout << endl << account.getFirstName() << endl;
    cout << "Press ENTER to continue!";
    cin.get();

    // Gör insättningar och uttag, skriv kontobesked
    account.deposit(1000);
    showAccount(account);
    account.withdrawal(1500);
    showAccount(account);
    cout << "Press ENTER to continue!";
    cin.get();

    return 0;
}
```

### Kommentarer:

- Nu sätts värden på medlemsvariablerna innan insättning och uttag görs.
- Detta kan göras på ett smidigare sätt med konstruktorer, mer om dessa senare.
- Körning av programmet visar att resultatet är OK.
- Öppna class\_020 om du vill se hela programmet. Det som saknas ovan är implementation av deposit(), withdrawal() och showAccount(), dessa finns dock i förra exemplet.

## 2.3 Sammanfattning

Klass : Data och operationer på data samlas i en "datatyp"

Data = egenskaper

Operationer på data = händelser

Data hanteras av datamedlemmar

Operationer utförs av medlemsfunktioner

Objekt: En instans av en klass (en variabel av "datatypen" klass)

Inkapsling: Data kapslas in så att de blir åtkomliga enbart genom klassens medlemsfunktioner.

Abstrakt datatyp:

Ex Ett konto är en abstraktion på så sätt att när man använder begreppet konto så är det underförstått att med begreppet följer ett antal detaljer som man inte anger detalj för detalj.

I C++ kapslar man in data, och gör dessa osynliga för användaren. Data blir synliga genom medlemsfunktionerna. Den som gör klassen bestämmer "användargränssnittet" d.v.s. hur medlemsfunktionerna utformas.

Klassen blir en abstrakt datatyp: Account mittKonto;

Objektet mittKonto innehåller detaljer som inte syns här. Man måste veta hur klassen är utformad för att kunna använda den. Detta görs i klassdefinitionen (**class** Account {...})

Klassdefinitionen är klassens ansikte mot användaren (interface).

Hur medlemsfunktionerna är implementerade är inte intressant för användaren.

## 2.4 Konstruktör, destruktör

I class\_020 gjordes initieringen av medlemsvariablerna med set-funktioner (medlemsfunktioner). Initieringen görs dock smidigare med konstruktörer. En konstruktör är en medlemsfunktion som körs automatiskt när ett objekt skapas. Motsatsen till en konstruktör är en destruktör. Destruktorn körs när ett objekt tas bort.

### 2.4.1 Skapa konstruktörer och destruktörer

I class\_030 bygger vi på klassen Account med två konstruktörer och en destruktör. Konstruktörer och destruktörer kännetecknas av att dom har samma namn som klassen.

```
class Account
{
    private:
        string firstName;           // Datamedlemmar
        string lastName;
        string accountNr;
        int balance;

    public:
        Account();                  //Förvald konstruktör
        Account(string pFirstName, string pLastName, //Konstruktör för initiering
                string pAccountNr,int pBalance);    // av datamedlemmar
        ~Account();                 //Destruktör

        void deposit(int amount);   // Medlemsfunktioner
        void withdrawal(int amount);

        void setFirstName(string pFirstName); // Medlemsfunktioner som sätter
        void setLastName(string pLastName);   // värden på medlemsvariabler
        void setAccountNr(string pAccountNr);
        void setBalance(int pBalance);

        string getFirstName() const;          // Medlemsfunktioner som returnerar
        string getLastName() const;           // medlemsvariablers värden
        string getAccountNr() const;
        int getBalance() const;

};                                           // Avslutar definitionen av klassen Account

// Fristående funktion för utskrift av ett konto
void showAccount(Account const &account);
```

#### Kommentarer:

- en konstruktör har samma namn som klassen
- en konstruktör saknar returdatatyp
- en konstruktör utan parameter kallas förvald konstruktör och körs automatiskt då ett objekt skapas (se nedan)
- en konstruktör med parameter (parameterlista) används för att sätta värden på (initiera) medlemsvariabler
- destruktorn har samma namn som klassen men med ett ~(tilde) som första tecken.
- det går bara att ha en destruktör
- destruktorn körs automatiskt då objektet tas bort (t.ex. går ur scope eller då programmet stängs)

### Implementering av Accounts konstruktörer och destruktör:

```
//-----  
//Förvald konstruktor (Default constructor)  
//-----  
Account::Account()  
{  
    firstName = "";  
    lastName = "";  
    accountNr = "No";  
    balance = 0;  
}  
//-----  
//Konstruktor för initiering av datamedlemmarna  
//-----  
Account::Account(string pFirstName, string pLastName,  
                  string pAccountNr, int pBalance)  
{  
    firstName = pFirstName;  
    lastName = pLastName;  
    accountNr = pAccountNr;  
    balance = pBalance;  
}  
//-----  
//Destruktor  
//-----  
Account::~~Account()  
{  
    cout << "Nu körs destruktorn för kontot med kontoNr: " << accountNr<< endl;  
}
```

### Kommentarer:

- Detta program behöver ingen destruktör. Innehållet i destruktorn är med här enbart i demonstrationssyfte för att du ska få se när den körs. Skriv inte så här i klasserna som du lämnar in i laborationerna!

## 2.4.2 Användning av konstruktörerna och destruktorn

Den förvalda konstruktorn körs med följande anrop:

```
Account account1;
```

”Initieringskonstruktorn” körs med anropet:

```
Account account2("Ewa", "Swenson", "A111-222", 3000);
```

Destruktorn körs när objektet tas bort, i det här fallet är har jag skapat objektens scope med måsvingar, { ... }. När programkörningen går förbi en höger måsvinge, }, tas objektet bort och destruktorn körs.

Hela klientprogrammet ( class\_030.cpp) ser du på nästa sida:

```
int main()
{
    // Skapa ett objekt av klassen Account, den förvalda konstruktorn körs.
    {
        Account account1;
        showAccount(account1);
        cout << "Press ENTER to continue!";
        cin.get();

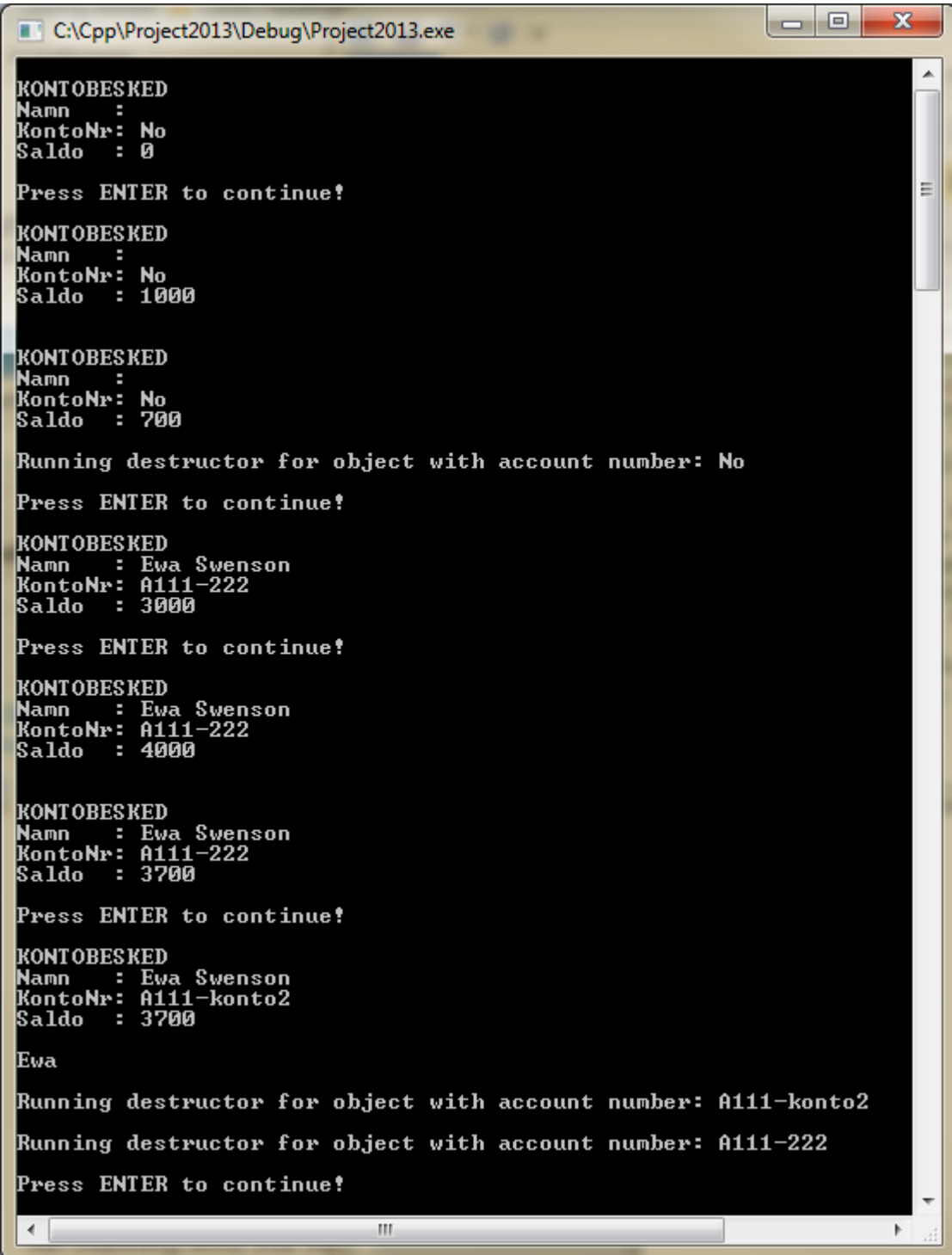
        // Gör insättningar och uttag, skriv kontobesked
        account1.deposit(1000);
        showAccount(account1);
        account1.withdrawal(300);
        showAccount(account1);
    }
    cout << "Press ENTER to continue!";
    cin.get();

    {
        // Skapa ett objekt av klassen Account, initieringskonstruktorn körs.
        Account account2("Ewa", "Swenson", "A111-222", 3000);
        showAccount(account2);
        cout << "Press ENTER to continue!";
        cin.get();

        // Gör insättningar och uttag, skriv kontobesked
        account2.deposit(1000);
        showAccount(account2);
        account2.withdrawal(300);
        showAccount(account2);
        cout << "Press ENTER to continue!";
        cin.get();

        Account account3;
        account3 = account2;
        account3.setAccountNr("A111-konto2");
        showAccount(account3);
        cout << account3.getFirstName() << endl << endl;
    }
    return 0;
}
```

Utskrift från körning på nästa sida:



```
C:\Cpp\Project2013\Debug\Project2013.exe

KONTOBESKED
Namn   :
KontoNr: No
Saldo  : 0

Press ENTER to continue!

KONTOBESKED
Namn   :
KontoNr: No
Saldo  : 1000

KONTOBESKED
Namn   :
KontoNr: No
Saldo  : 700

Running destructor for object with account number: No

Press ENTER to continue!

KONTOBESKED
Namn   : Ewa Swenson
KontoNr: A111-222
Saldo  : 3000

Press ENTER to continue!

KONTOBESKED
Namn   : Ewa Swenson
KontoNr: A111-222
Saldo  : 4000

KONTOBESKED
Namn   : Ewa Swenson
KontoNr: A111-222
Saldo  : 3700

Press ENTER to continue!

KONTOBESKED
Namn   : Ewa Swenson
KontoNr: A111-konto2
Saldo  : 3700

Ewa

Running destructor for object with account number: A111-konto2
Running destructor for object with account number: A111-222
Press ENTER to continue!
```

#### Kommentarer:

- Det körs en destruktör för varje objekt som skapas
- Destruktörerna körs i motsatt ordning mot den ordning de skapades (stack!)

## 2.5 Placering av klasser i header- och definitionsfil

I class\_040 visas hur man delar upp klasser i headerfil och definitionsfil och sedan anropar headerfilen från klientprogrammet.

Uppdelningen görs så här

- lägg klassdefinitionen i headerfilen, här *account.h*
- lägg funktionsdefinitionerna i definitionsfilen, här *account.cpp*
- skriv inkluderingsdirektiv (här: `#include "account.h"`) i klientprogrammet och i definitionsfilen
- inkludera definitionsfilen (här *account.cpp*) i projektet

```
// class_040.cpp

#include "account.h"

main()
{
    Account a1;
    a1.showAccount()
    ...
}
```

```
//account.h

class Account
{
private:
    ...
public:
    ...
};
```

```
//account.cpp
//Definitioner av medlfkner
#include "account.h"

Account::showAccount()
{
    ...
}

...
```

### 2.5.1 Klientprogram

I klientprogrammet kan man sedan koncentrera sig på att använda objekten.

```
// class_040 Version 10
// Per Ekeroot 2014-01-13
//-----
#include <iostream>
#include <iomanip>
#include "account.h"
using namespace std;
//-----

int main()
{
    // Skapa ett objekt av klassen Account, initieringskonstruktorn körs.
    Account account1("Ewa", "Swenson", "A111-222", 3000);
    showAccount(account1);

    // Gör insättningar och uttag, skriv kontobesked
    account1.deposit(1000);
    showAccount(account1);
    account1.withdrawal(300);
    showAccount(account1);

    // Skapa ett nytt objekt, kör den förvalda konstruktorn
    Account account2;

    // Kopiera så att konto2 får samma värden som konto
    account2 = account1;
    cout << endl << "Print account2 which is copied from account 1!" << endl;
    showAccount(account2);
    return 0;
}
```



### Kommentarer:

- Det enda nya i detta program, förutom uppdelning på flera filer, är tilldelning av ett konto till ett annat konto
- `account2 = account1`
- Tilldelning (=kopiering av datamedlemmarna) skapas automatiskt när man skapar ett objekt
- en klass kopieras datamedlem för datamedlem (medlemsvariabel)
- skriv t ex *account1*. (OBS! punkten!) och vänta på hjälpen så syns vilka medlemmar (data & funktioner) som ingår i klassen....

## 2.5.2 Headerfil

Klassdefinitionen headerfil:

```
// account.h Version 10
// Headerfil till klassen Account
// Per Ekeroot 2014-01-13
//-----
#ifndef accountH
#define accountH
    #include <string>
    using namespace std;
//-----
// Klassdefinition
//-----
class Account
{
    private:
        string firstName;           // Datamedlemmar
        string lastName;
        string accountNr;
        int balance;

    public:
        Account();                  //Förvald konstruktor
        Account(string pFirstName, string pLastName, //Konstruktor för ini-
                string pAccountNr,int pBalance);    // tiering av datamedl.
        ~Account();                //Destruktor

        void deposit(int amount);   // Medlemsfunktioner
        void withdrawal(int amount);

        void setFirstName(string pFirstName);    // Medlemsfunktioner som sätter
        void setLastName(string pLastName);      // värden på datamedlemmar
        void setAccountNr(string pAccountNr);
        void setBalance(int pBalance);

        string getFirstName() const;             // Medlemsfunktioner som returnerar
        string getLastName() const;              // datamedlemmars värden
        string getAccountNr() const;
        int getBalance() const;
};                                                // Avslutar definitionen av klassen Account

// Fristående funktion för utskrift av ett konto
void showAccount(Account const &account);
#endif
```

## 2.5.3 Definitionsfil

Definitionsfilen innehållande medlemsfunktionerna till Account:

```
// account.cpp Version 10
// Definitionsfil till klassen Account
// Per Ekeroot 2014-01-13
//-----
#include "account.h"
#include <iostream>
using namespace std;
//-----
// Definition av medlemsfunktioner
//-----
//Förvald konstruktor (Default constructor)
//-----
Account::Account()
{
    firstName = "";
    lastName = "";
    accountNr = "No";
    balance = 0;
}

//-----
//Konstruktor för initiering av datamedlemmarna
//-----
Account::Account(string pFirstName, string pLastName,
                  string pAccountNr,int pBalance)
{
    firstName = pFirstName;
    lastName = pLastName;
    accountNr = pAccountNr;
    balance = pBalance;
}

//-----
// Destruktor // Bortkommenterad eftersom den egentligen inte behövs.
//-----
Account::~~Account()
{
    //cout << "Running destructor for object with account number: " << accountNr<<
endl << endl;
}

//-----
// deposit
// Sätt in beloppet amount som läggs till saldot balance
//-----
void Account::deposit(int amount)
{
    if(amount > 0)                //Kolla att insättningsbeloppet > noll
        balance += amount;        //Addera beloppet till saldot
}

osv . . .
```

## 2.6 Skapa en array av objekt

Ett objekt är en instans av en klass, detta kan jämföras med variabel av en datatyp.

```
Account account;
```

På samma sätt som man gör en array av heltal så borde man kunna göra en array av objekt.

Och det kan man! Gör så här:

```
const size_t SIZE = 3;  
Account account_arr [SIZE];
```

Observera att den förvalda konstruktorn körs på alla objekt i arrayen (account\_arr [0], account\_arr[1] ..)!

Man **måste** alltså skriva en förvald konstruktor om man vill att alla objekt i arrayen ska initieras.

Så här matar du in och skriver ut data till/från vektorn:

```
// Mata in data  
string s;  
cout << "Förnamn      : ";  
cin >>s;  
account_arr[i].SetFirstName(s); // för konto nr i  
  
// Skriv saldobesked för alla konton  
for(auto idx : account_arr)  
    showAccount(idx);
```

Som alternativ till den förvalda konstruktorn kan initieringskonstruktorn användas:

```
// Skapa konton och initiera dem  
Account account_arr2[SIZE]={ Account("Ewa","Swensson","A123-234",1000),  
                             Account("Ola","Karling","A234-345",500),  
                             Account("Ida","Andersson","A345-456",1000),  
                             };
```

Hela class\_050:

```
// class_050 Version 10  
// Gör en array av objekt  
// Skapa 3 konton (objekt, instanser) av klassen Account  
// Per Ekeroot 2014-01-13  
//-----  
#include "account.h"  
#include <iostream>  
using namespace std;  
  
//-----  
// Huvudprogram  
//-----  
int main()  
{  
    // Skapa en konstant för arrayens storlek  
    const size_t SIZE = 3;  
  
    // Skapa en array med tre objekt av klassen Account  
    //-----  
    // Alternativ 1: Defaultkonstruktor körs  
    //-----  
    Account account_arr[SIZE];  
  
    // Skriv saldobesked för alla konton
```

```
for(auto idx : account_arr)
    showAccount(idx);

// Mata in data till kontona
for(size_t i=0; i<SIZE; i++)
{
    string s;
    cout << "First name  : ";
    cin >>s;
    account_arr[i].setFirstName(s);

    cout << "Last name   : ";
    cin >>s;
    account_arr[i].setLastName(s);

    cout << "Account nr   : ";
    cin >>s;
    account_arr[i].setAccountNr(s);

    int a;
    cout << "Balance     : ";
    cin >>a;
    account_arr[i].setBalance(a);

    cout << endl ;
}
cin.get(); // Läs bort ENTER från inmatningsströmen

// Skriv saldobesked för alla konton
for(auto idx : account_arr)
    showAccount(idx);

cout << "Press ENTER to continue!";
cin.get();
//-----
// Alternativ2: Använd initieringskonstruktorn
//-----
// Skapa konton och initiera dem
Account account_arr2[SIZE]={Account("Ewa", "Swensson", "A123-234", 1000),
                             Account("Ola", "Karling", "A234-345", 500),
                             Account("Ida", "Andersson", "A345-456", 1000)
};

// Skriv kontobesked
for(auto idx : account_arr2)
    showAccount(idx);

cout << "Press ENTER to continue!";
cin.get();

return 0;
}
```

## 2.7 Inline-funktioner i klasser och initieringslistor

Här kommer ett nytt exempel på hur man kan göra en klass. Denna klass ska vara en mall för en klocka i vilken man hanterar tid i form av timmar och minuter. Tiden ska kunna stegas minut för minut.

För att beskriva klassen ritas vi ett klassdiagram enligt UML (Unified Modeling Language). Diagrammet består av en rektangel som är delad i tre delar. Överst skriver man klassens namn, i mitten klassens datamedlemmar och nederst medlemsfunktionerna.

### Datamedlemmar:

timme, minut

### Medlemsfunktioner:

Sätt timme

Sätt minut

Ticka en minut

Läs timme, minut

Klassdiagram (UML)

Time
timme int minut int
Sätt timme Sätt minut Läs timme Läs minut Ticka en minut

### Klassen Time: headerfil

```
// Time.h Version 10
// Headerfil till klassen Time
// Per Ekeroot 2014-01-13
//-----
#ifndef TimeH
#define TimeH

class Time
{
private:
    int hour, min;

public:
    // Konstruktörer
    Time(); // Förvald konstruktor
    Time(int pHour, int pMin=0); // Initieringskonstruktor

    // Sätt värden på datamedlemmar
    void setHour(int pHour);
    void setMin(int pMin);

    // Läs datamedlemmars värden
    int getHour()const {return hour;} // inline - funktion
    int getMin()const {return min;} // inline - funktion

    // Stega fram tiden en minut
    void tic();
};
//-----
// Funktionsprototyp för en funktion som INTE hör till klassen Time
//-----
void showTime(Time const &t);
#endif
```

### Kommentarer:

- Medlemsfunktionerna getHour() och getMin() definieras **inline**

- Detta innebär:
  - kompilatorn ersätter funktionsanropet med motsvarande kod
  - programmet blir snabbare
  - programmet blir längre om funktionen anropas många gånger
  - man frångår principen att medlemsfunktionernas detaljer ska gömmas i implementationsfilen
- Rekommendation: använd inline-funktioner sparsamt och bara för små funktioner.
- I Time(int pHour, int pMin=0) används förvalt värde för andra parametern. Detta innebär att man t ex kan gör anropet Time t(12), vilket ger hour = 12 och min=0.
- Funktionen showTime() används för att ut klassen Time's datamedlemmar på ett enkelt sätt

#### Implementation av övriga medlemsfunktioner:

```
//-----  
//  time.cpp Version 10  
//  Definition av medlemsfunktionerna i klassen Time  
//  Klassen har två konstruktörer  
//  Per Ekeroot  
//  2014-01-13  
//-----  
#include "time.h"  
#include <iostream>  
#include <iomanip>    // setw, setfill  
using namespace std;  
  
//-----  
// Förvald konstruktor  
//-----  
Time::Time()  
{  
    hour = min = 0;  
}  
  
//-----  
// Initieringskonstruktor  
// Använd set-funktioner för tilldelning av värden så kontroll av indata görs  
//-----  
Time::Time(int pHour, int pMin)  
{  
    setHour(pHour);  
    setMin(pMin);  
}  
  
//-----  
// setHour  
// Ange timme  
// Testa så att inmatad timme ligger i intervallet 0..23  
// Returnera -1 om så inte är fallet  
//-----  
void Time::setHour(int pHour)  
{  
    if(pHour >= 0 && pHour < 24)  
        hour = pHour;  
    else  
        hour = -1;  
}
```

```
//-----  
// setMin  
// Ange minut  
// Testa så att inmatad minut ligger i intervallet 0..59  
// Returnera -1 om så inte är fallet  
//-----  
void Time::setMin(int pMin)  
{  
    if(pMin >= 0 && pMin < 60)  
        min = pMin;  
    else  
        min = -1;  
}  
//-----  
// Stega en minut  
//-----  
void Time::tic()  
{  
    min = (min+1) % 60;  
    if (min==0)  
        hour = (hour+1) % 24;  
}  
//-----  
// Funktionsprototyp för funktion som INTE hör till klassen  
//-----  
// showTime  
// Skriver ut tiden på formatet hh:mm  
// Inledande 0:or skrivs alltid  
// setfills() räckvidd är inom {...}  
//-----  
void showTime(Time const &t)  
{  
    cout << setfill('0') << setw(2) << t.getHour() << ':'  
        << setw(2) << t.getMin() << endl;  
}
```

#### Kommentarer:

- setHour() och setMin() testar inmatade data så att de ligger inom tillåtna intervall
- Initieringskonstruktorn anropar setHour() och setMin() vid tilldelning av data till datamedlemmarna

Nu ska vi använda klassen Time i ett program (class\_060).

Skapa ett Time-objekt:

```
Time time;
```

Sätt tiden:

```
time.setHour(5);           // Ange timme  
time.setMin(55);          // Ange minut
```

Skriv ut tiden:

```
showTime(time);
```

showTime() är en funktion som inte hör till klassen. En anledning till detta är att man vill att klassen ska vara så generell som möjligt. Om man lägger in en utskriftsfunktion i klassen så låser man utskriftslayout till klassen och klassen blir mindre generell.

Låt kockan gå 7 minuter

```
for(int i=0; i<7; i++)          // Låt klockan stega 7 minuter
    time.tic();
```

Läs av en datamedlem:

```
cout << time.getHour() << " hours " <<
```

En konstruktor kan användas på olika sätt. Här visas fyra alternativ:

```
// Här används konstruktorn för initiering av datamedlemmar (variant 1)
Time time3(15, 16);
showTime(time3);

// Här används konstruktorn för initiering av datamedlemmar (variant 2)
Time time4 = Time(16,17);
showTime(time4);

// Här används konstruktorn för initiering av datamedlemmar (variant 2)
// Vi utnyttjar att den andra parametern har ett förvalt värde
Time time5 = Time(17);
showTime(time5);

// Här används en konstruktor som tar ett Time-objekt som argument (variant 1)
Time time6(time);
showTime(time6);

// Här används en konstruktor som tar ett Time-objekt som argument (variant 2)
Time time7 = Time(time2);
showTime(time7);
```

Och hela programmet på nästa sida:



```
// class_060 Version 10
// Använd klassen Time
// Per Ekeroot 2014-01-13
//-----
#include "time.h"
#include <iostream>
using namespace std;
//-----
int main()
{
    // Skapa ett Time - objekt, den förvalda konstruktorn körs
    Time time;
    showTime(time);          // Skriv ut timme och minut

    time.setHour(5);         // Ange timme
    time.setMin(55);         // Ange minut
    showTime(time);

    for(int i=0; i < 7; i++) // Låt klockan stega 7 minuter
        time.tic();

    showTime(time);

    // Använd set- och getfunktionerna
    int hh, mm;
    cout << endl<< endl<< "Input time" << endl;
    cout << "Hour    : ";
    cin >> hh;
    cout << "Minute : ";
    cin >> mm;

    Time time2;
    time2.setHour(hh);
    time2.setMin(mm);
    cout << time2.getHour() << " hours " <<
        time2.getMin() << " minutes " << endl << endl;

    // Alternativ för användning av konstruktorer
    // Här används konstruktorn för initiering av datamedlemmar (variant 1)
    Time time3(15, 16);
    showTime(time3);

    // Här används konstruktorn för initiering av datamedlemmar (variant 2)
    Time time4 = Time(16,17);
    showTime(time4);

    // Här används konstruktorn för initiering av datamedlemmar (variant 2)
    // Vi utnyttjar att den andra parametern har ett förvalt värde
    Time time5 = Time(17);
    showTime(time5);

    // Här används en konstruktor som tar ett Time-objekt som argument (variant 1)
    Time time6(time);
    showTime(time6);

    // Här används en konstruktor som tar ett Time-objekt som argument (variant 2)
    Time time7 = Time(time2);
    showTime(time7);

    return 0;
}
```

## 2.8 Att använda en klass i en annan klass

Man kan ju se ett objekt som en variabel av en datatyp. Därför går det alldeles utmärkt att använda ett objekt som medlemsvariabel (datamedlem) i en klass. Vi ska som exempel se hur man använder ett Time-objekt i en klass som hanterar flygförbindelser (flighter). Klassen får namnet Flight. För Flight vill vi att följande ska gälla:

### Datamedlemmar:

flightnummer  
avgångstid  
ankomsttid

### Medlemsfunktioner:

Sätta flightnummer, avgångs- och ankomsttid med konstruktor  
Setfunktioner för alla datamedlemmar  
Getfunktioner för alla datamedlemmar  
Ange försening (avgångs- och ankomsttid)

Flight
flightnummer String avgångstid Time ankomsttid Time
Konstruktorer Setfunktioner Getfunktioner Ange försening

### Klassen Flight:

```
#ifndef flight_2H
#define flight_2H
#include "time.h"
#include <string>
#include <iostream>
using namespace std;

class Flight
{
private:
    string flightNr;
    Time depTime, arrTime;

public:
    Flight();
    Flight(string aFlightNr, Time aDepTime, Time aArrTime);

    // Setfunktioner
    void setFlightNr(string aNr);
    void setDepTime(Time aDepTime);
    void setArrTime(Time aArrTime);

    // Getfunktioner
    string getFlightNr() const { return flightNr; }
    Time getDepTime() const { return depTime; }
    Time getArrTime() const { return arrTime; }

    void delay(int min);
};

void showFlightInfo(Flight const &flight);
#endif
```

### Kommentarer:

- depTime och arrTime är objekt av klassen Time
- Flight() är en förvald konstruktor som nollställer alla medlemsvariabler (datamedlemmar)
- Flight(string aFlightNr, Time aDepTime, Time aArrTime) är en konstruktor som har parametrarna flightNr som en sträng, avgångstid och ankomsttid som Timeobjekt.

- Med delay() kan man ange en förbindelses försening i minuter (avgångs- **och** ankomsttid)
- Getfunktionerna är inline-deklarerade och returnerar string- och Timeobjekt.
- showFlightInfo() är en fristående funktion som skriver flight-data på skärmen

Implementeringen av Flight's medlemsfunktioner:

```
// flight.cpp Version 10
// Definitionsfil till klassen Flight
// Per Ekeroot 2014-01-13
//-----
#include "flight.h"
#include <iostream>
//-----
// Definiering av Flights medlemsfunktioner
//-----
// Förvald konstruktor
//-----
Flight::Flight()
{
    flightNr = "";
}
```

#### Kommentarer:

- Time har en förvald konstruktor som nollställer objekten depTime och arrTime när konstruktorn Flight() körs. Man behöver inte anropa deras förvalda konstruktörer explicit!
- Man behöver egentligen inte heller skriva flightNr="", eftersom string-klassen har en förvald konstruktor som "nollställer" strängen.

```
Flight::Flight(string pFlightNr, Time pDepTime, Time pArrTime)
{
    flightNr = pFlightNr;
    depTime = pDepTime;
    arrTime = pArrTime;
}
```

#### Kommentarer:

- Initieringskonstruktor med parametrar i form av ett stringobjekt och två Time-objekt.

Klassens setfunktioner:

```
// Setfunktioner
void Flight::setFlightNr(string pNr)
{
    flightNr = pNr;
}

void Flight::setDepTime(Time pDepTime)
{
    depTime = pDepTime;
}

void Flight::setArrTime(Time pArrTime)
{
    arrTime = pArrTime;
}
```

Ange försening:

```
void Flight::delay(int min)
{
    for( int i=1; i<= min; i++)
    {
        depTime.tic();
        arrTime.tic();
    }
}
```

#### Kommentarer

- Times's medlemsfunktion tic() körs för depTime resp arrTime.

Skriv information om flygningen i en fristående funktion

```
void showFlightInfo(Flight const &flight)
{
    cout << "Flight Nr   : " << flight.getFlightNr() << endl;
    cout << "Departure time: " ;
    showTime(flight.getDepTime());
    cout << "Arrival time  : " ;
    showTime(flight.getArrTime());
    cout << endl;
}
```

#### Kommentarer:

- Funktionen *showTime()* från time.cpp används för att skriva avgångs- resp ankomsttid.

## 2.8.1 Ett program som använder klassen Flight

Skapa ett Flightobjekt som initieras med initieringskonstruktorn. Använd initieringskonstruktorn från Time-klassen för att kunna använda avgångstimme, avgångsminut, ankomststimme och ankomstminut som argument. Skriv ut flightinfo. Lägg till en försening på 32 minuter och skriv ut flightinfo igen.

```
Flight flight("ABC-223",Time(10,35),Time(11,15));
showFlightInfo(flight);
flight.delay(32);
showFlightInfo(flight);
```

Skapa ett Flightobjekt genom att först skapa två Time-objekt vilka används för att skapa flight-objektet.

```
Time dTime(10,15);
Time aTime(10,55);
Flight flight2("AAA-345",dTime,aTime);
```

Använd klassens setfunktioner:

```
int hh, mm, ft_hh, ft_mm;

cout << endl;
cout << "Input time for departure" << endl;
cout << "Hour   : ";
cin >> hh;
cout << "Minute : ";
cin >> mm;
cout << "Input flight time" << endl;
```

```
cout << "Hour(s)      : ";
cin >> ft_hh;
cout << "Minutes      : ";
cin >> ft_mm;
cin.get();

Flight flight3;
flight3.setFlightNr("BBB123");
flight3.setDepTime(Time(hh,mm));
flight3.setArrTime(Time(hh+ft_hh,mm+ft_mm));
cout << endl << endl;
showFlightInfo(flight3);
```

#### Kommentarer:

- Låt användaren mata in avgångstid och beräknad flygtid
- Eftersom Flight's setDepTime() respektive setArrTime() tar ett Timeobjekt som argument, måste timmar och minuter först läggas in i ett Timeobjekt. Detta kan göras på två sätt:
  - 1) skapa först ett time- objekt  
Time dTime2(hh,mm);  
flight3.setDepTime(dTime2);
  - 2) använd Time's initieringskonstruktor  
flight3.setArrTime(Time(hh+ft\_hh,mm+ft\_mm));

Här kommer hela programmet, se nästa sida:

```
/ class_070 Version 10
// Använd klassen Flight
// Per Ekeroot 2014-01-13
//-----
#include "flight.h"
#include <iostream>
using namespace std;
//-----
int main()
{
    // Skapa ett flight-objekt, lägg in data, ändra data , skriv ut
    Flight flight("ABC-223",Time(10,35),Time(11,15));
    showFlightInfo(flight);
    flight.delay(32);
    showFlightInfo(flight);
    cout << "Press ENTER to continue!";
    cin.get();

    // Alternativ i vilket objekt för avgångs- och ankomsttid
    // skapas innan Flight-objektet skapas och initieras med dessa
    Time dTime(10,15);
    Time aTime(10,55);
    Flight flight2("AAA-345",dTime,aTime);
    cout << endl << endl;
    showFlightInfo(flight2);

    // Använd set- och getfunktioner
    int hh, mm, ft_hh, ft_mm;

    cout << endl;
    cout << "Input time for departure" << endl;
    cout << "Hour    : ";
    cin >> hh;
    cout << "Minute : ";
    cin >> mm;
    cout << "Input flight time" << endl;
    cout << "Hour(s)   : ";
    cin >> ft_hh;
    cout << "Minutes   : ";
    cin >> ft_mm;
    cin.get();

    Flight flight3;
    flight3.setFlightNr("BBB123");
    flight3.setDepTime(Time(hh,mm));
    flight3.setArrTime(Time(hh+ft_hh,mm+ft_mm));
    cout << endl << endl;
    showFlightInfo(flight3);

    cout << endl << endl;
    cout << "Press ENTER to continue!";
    cin.get();
    return 0;
}
```

## 2.9 Stack som en abstrakt datatyp

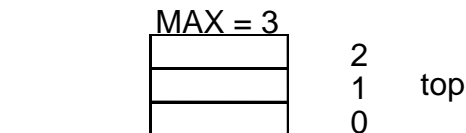
Vi ska implementera en stack som en abstrakt datatyp (ADT). En ADT beskriver en datatyp generellt utan att gå in på detaljer.

En stack

- En stack är ett sätt att lagra data där data läggs i en hög
- Nya data läggs överst i högen
- Data tas ut överst i högen (enbart)
- Sist in först ut (Last In First Out = LIFO) // (Kö FIFO = First In First Out)

En **generell** beskrivning av stacken (på ett abstrakt sätt):

Stack
Array med element
Skapa tom stack Lägg data på stacken Ta data från stacken Skriv antal element



Detta är ett klassdiagram ritat enl. UML.

I C++ gör vi denna beskrivning m.h.a. en klass. I detta exempel har vi valt att använda en array för lagring av elementen i stacken. En array har en bestämd storlek vilket innebär att stacken kan bli full.

En klassdefinition för en stack, stack.h:

```
#ifndef stackH
#define stackH
//-----
#include <iostream>
#include <vector>
using namespace std;

typedef int Item;           // Datatyp för elementen i stacken

class Stack
{
private:
    vector<Item> item;       // En vector med element av datatypen Item

public:
    Stack(){};              // Förvald konstruktor, den är tom!

    void push(Item pItem);   // Lägg ett element på stacken
    Item pop();              // Ta det översta elementet från stacken

    bool isEmpty() const;    // Är stacken tom?
    int getSize() const;     // Returnera aktuellt antal element i stacken
};
#endif
```

**Kommentarer:**

- typedef int Item; innebär att Item ersätts med en int. Detta för att det ska vara enkelt att byta datatyp för det som lagras i stacken.
- Här används en vector lagra data. Det skulle lika gärna kunna ha varit en statisk array eller en länkad lista. Eftersom datamedlemmarna är private-deklarerade ligger detta i linje med ADT, d.v.s. att hur man lagrar data internt är inte intressant utåt.
- Klassen har medlemsfunktioner för kontroll av om stacken är tom samt för att lägga data på och ta bort data från stacken.

Implementering av medlemsfunktionerna:

```
// stack.cpp Version 5
//-----
#include "stack.h"
//-----
// Lägg ett element på stacken
//-----
void Stack::push(Item pItem)
{
    item.push_back(pItem);
}
```

**Kommentarer:**

- push\_back() lägger till data sist i vectorn.

```
//-----
// Ta det översta elementet från stacken
//-----
Item Stack::pop()
{
    Item returnItem = item.back(); // Läs elementet som finns sist i vectorn
    item.pop_back();               // Ta bort elementet som finns sist i vectorn
    return returnItem;             // Returnera läst element
}
```

**Kommentarer:**

- Stackens översta element returneras och tas bort.
- Toppen på stacken ligger sist i vectorn!
- Observera att pop\_back() enbart tar bort det sista elementet i vectorn. För att kunna returnera det läser man det sista elementet med back()!

```
//-----
// Returnera true om stacken är tom annars false
//-----
bool Stack::isEmpty() const
{
    return item.size() == 0;
}

//-----
// Returnera antal elementet
//-----
int Stack::getSize() const
{
    return item.size();
}
```

**Kommentarer:**

- En vector håller själv reda på antal inlagda element. Detta görs med medlemsfunktionen size().



## 2.10 Använd stacken i ett program

I klientprogrammet läggs ett antal heltal på stacken.

Skapa ett stackobjekt så här:

```
Stack stack;
```

Körningen av programmet görs via en meny:

Lägg data på stacken  
Ta data från stacken  
Antal element på stacken  
Sluta

För varje menyalternativ skrivs en funktion som tar hand om in- och utmatning, d.v.s. hanterar gränssnittet mot användaren. Funktionsprototyperna för dessa ser ut så här:

```
void pushData(Stack &stack);  
void popData(Stack &stack);  
void printSize(Stack const &stack);
```

Stacken skickas med som argument till resp. funktion. Parametrarna är referensdeklarerade av effektivitetsskäl och för att funktionerna ska returnera den manipulerade stacken. I *printSize()* är parametern const-deklarerad eftersom stacken inte ändras.

Implementering av "gränssnittsfunktionerna":

```
void pushData(Stack &stack)  
{  
    Item data;  
    cout << "Input data: ";  
    cin >> data;  
    stack.push(data);  
    cout << endl << endl;  
}
```

**Kommentarer:**

- *stack.push(data)* lägger ett tal på stacken.

```
void popData(Stack &stack)  
{  
    if(!stack.isEmpty())  
    {  
        Item data = stack.pop();  
        cout << "Removed data: "<< data << endl << endl;  
    }  
    else  
        cout << "The stack is empty!" << endl << endl;  
  
    wait();  
}
```

**Kommentarer:**

- *stack.isEmpty()* kontrollerar så att stacken inte är tom innan data tas bort från stacken
- *wait()* är en funktion som stoppar upp körningen av programmet och väntar på knapptryckning från användaren för att körningen ska fortsätta. Den finns i biblioteksfilen *PEFuncBibl.cpp*.

```
void printSize(Stack const &stack)
{
    cout << "Number of elements on the stack: " << stack.getSize() << endl << endl;
    wait();
}
```

**Kommentarer:**

- *printSize()* skriver ut antal element i stacken genom att anropa *stack.getSize()*.

Med dessa funktioner bygger vi huvudprogrammet. Fullständig kod finns i *class\_080*.

```
int main()
{
    // Initiera menyn
    const int ITEMNUM = 4;
    string menuItems[ITEMNUM] = {"Put data (int) on the stack",
                                "Remove data from the stack",
                                "Number of elements on the stack",
                                "Quit"};

    // Skapa en tom stack
    Stack stack;

    // Kör menyn
    bool go = true;
    do
    {
        switch(menu( menuItems, ITEMNUM))
        {
            case '1': pushData(stack);    // Lägg data på stacken
                       break;
            case '2': popData(stack);     // Ta data från stacken
                       break;
            case '3': printSize(stack);   // Skriv stackens storlek på skärmen
                       break;
            case '4': go = false;         // Avsluta programmet
        };
    } while(go);
    return 0;
}
```

**Kommentarer:**

- *menu()* en menyfunktion som ligger i biblioteksfilen *PEFuncBibl.cpp*.
- En tom stack skapas med *Stack stack*;
- För varje menyalternativ körs en funktion. Stacken skickas med som en referensparameter. Dessa funktioner kan ses som ett gränsskikt mot användaren. Här skrivs all in- och utmatning till och från skärmen.