Kapitel 10

Klasser

10.1 Vägen från C till C++

När människan började lära sig att flyga och gjorde de första försöken råkade hon ut för många smärtsamma olyckor. Man försökte härma fåglarnas flygförmåga genom att hoppa från berg och andra höjder med hjälp av äventyrliga konstruktioner. Då hade man inte insett än att flyg kunde vara en fortsättning på att färdas på jorden: Det räcker i princip med ett par vingar till bilen, mer fart och en startbana för att komma upp i luften. Ungefär så har människan erövrat den tredje dimensionen – en helt ny värld som slutligen öppnat portarna till universum. Att flygplan är mer avancerade än bilar är resultat av forskning och utveckling. Men innovation är otänkbar utan att dra nytta av befintlig kunskap: Flygplansmotorns grundkonstruktion bygger på bilmotorn.

Nu när vi vill öppna portarna till den nya världen av objektorienterad programmering – jämförbart med övergången från bil till flygplan – borde vi dra nytta av det vi lärt oss om procedural programmering. Även om vi tar steget in i en helt ny värld kan vi bygga på befintlig kunskap genom att komplettera den med nya revolutionerande idéer [*]. Det som i programmeringshistorien gjorde att man behövde objektorienterad programmering, var den växande komplexiteten hos program under 70-talet. Programmens storlek var avgörande för den växande komplexiteten: Man insåg att det inte längre räckte till att skriva och testa program som fungerade just då. Det var nödvändigt att med rimliga kostnader kunna även *underhålla* stora program, *förnya* och *vidareutveckla* dem så att de fungerade även i flera år och att de framför allt kunde anpassas till nyuppkomna situationer utan oöverkomliga svårigheter. Det i sin tur krävde att man redan i designstadiet behövde ett annorlunda upplägg. Fokuset förskjöts från problemlösning till modellering av verkligheten. Objektorienterad design (UML) kom in i bilden. Allt detta var endast med procedural programmering inte längre möjligt. Ett *paradigmskifte* hade blivit nödvändigt dvs en ändring av synen på programmering.

Men *på vilket sätt* kan vi nu utnyttja det vi lärt oss hittills för att komma in i den nya filosofin? I ett antal steg ska vi exempelifiera att objektorienterad programmeringens rötter finns i procedural programmering, även om i embryonalt tillstånd. Det avgörande konceptet är modularisering som är centralt inom procedural programmering. Vidareutvecklingen av modulariseringstanken leder till objektorienterad programmering. Om du kommer ihåg kom modularisering in i bilden när vi behandlade funktioner (sid 133). *Funktion* är nyckelkonceptet inom procedural programmering. Motsvarande nyckelkoncept inom objektorienterad programmering är *klass*. Vi börjar med ett program som varken är proceduralt eller objektorienterat.

[*] Eftersom C++ har tagit över allt från C och lagt till en hel del nytt, kan man inte lära sig objektorienterad kod utan en solid kunskap i procedural programmering. Därför har vi i boken först behandlat procedural programmering med C++ utan att explicit belasta framställningen med C-kod och de historiska skillnaderna till C. Det finns en del objektorienterad kod i boken som inte finns i C, t.ex. **`cout`**, **`cin`**, **`//`**, **`namespace`**, **`string`**, **`bool`**, … .

Sedan ska vi steg för steg modularisera det först till ett proceduralt och sedan till ett fullvärdigt objektorienterat program:

```
// All_in_main.cpp
// Programmet är inte objektorienterat då det inte skapas
// några objekt utan endast lokala variabler
// Programmet är inte ens modulariserat eller proceduralt
// då all kod (Input-Bearbetning-Output) står i main()
#include <iostream>
using namespace std;

int main()
{
   float radie, area, omkrets;                // Lokala
                                              // variabler
   cout << "Ange radien till en cirkel: ";
   cin >> radie;                              // Input

   area    = 3.14159 * radie * radie;         // Bearbetning
   omkrets = 2 * 3.14159 * radie;

   cout << "\nEn cirkel med radien  " << radie  // Output
        << "\nhar arean             " << area
        << "\noch omkretsen         " << omkrets << "\n\n";
}
```

Detta är ett typiskt nybörjarprogram som har all sin kod i **`main()`**. Därför är det varken proceduralt eller objektorienterat. Inget fel på det, om målet verkligen är att räkna ut cirkelarean och omkretsen när man matar in radien. Men vi vill använda exemplet för att lära oss att modularisera dvs separera bearbetningsdelen och skriva den i funktioner. Gör man det får man följande procedural variant där beräkningarna är flyttade från **`main()`** till separata *moduler* **`area()`** och **`omkrets()`**. Det modulariserade programmet består av tre moduler. De två sista skriver vi som funktioner in en separat headerfil:

```
// Procedure.h
// Funktioner som beräknar cirklens area och omkrets
// Anropas från main() lagrad i filen Procedure.cpp

float area(float r)                           // Modul area()
{
   return 3.14159 * r * r;
}

float omkrets(float r)                        // Modul omkrets()
{
   return 2 * 3.14159 * r;
}
```

För att dessa nya moduler ska kunna kommunicera med **`main()`** – alla deras variabler är ju lokala och gäller endast i respektive block – måste de ha parametern **`r`**

som får sitt värde överfört från **main()**:s variabel **radie** vid funktionsanrop. Parameteröverföring från **main()** till funktionerna **area()** och **omkrets()**, i praktiken en kopiering av den aktuella parametern **radie** till den formella parametern **r**, är alltså priset man måste betala för modulariseringen. Funktionsanropet står i den tredje modulen **main()** som står i följande separat fil och som måste inkludera headerfilen för att kunna kompileras:

```
// Procedure.cpp
// Innehåller modulen main(), inkluderar funktionerna area()
// och omkrets() i en headerfil och anropar dem härifrån
// Programmet är modulariserat men inte objektorienterat:
// Modulariseringen är på funktionsnivå: "modul" = funktion
// En modularisering på klassnivå kallas objektorienterad
#include <iostream>
using namespace std;

#include "Procedure.h"                    // area(), omkrets()

int main()                                // Modul main()
{
   float radie;                           // Lokal variabel

   cout << "Ange radien till en cirkel: ";
   cin >> radie;                          // Input
                                          // Output:
   cout << "\nEn cirkel med radien  " << radie
        << "\nhar arean              " << area(radie) // Anrop
        << "\noch omkretsen          " << omkrets(radie)
        << "\n\n";
}
```

Här är bearbetningsdelen flyttad till externlagrade *funktionerna* **area()** och **omkrets()** som returnerar de uttryck som i det icke-modulariserade programmet **All_in_main** tilldelades *variablerna* **area** och **omkrets**. Självklart kan man även separera in- och output-delen, men typiskt är att man separerar beräkningar. Nu finns i **main()** kvar input, output samt anropet av **area()** och **omkrets()**. Fokuset vid denna modularisering är på hur man löser problemet att beräkna cirkelns area och omkrets. Problemlösning är den styrande synen på progarmmering: Problemets lösning skrivs i en funktion eller procedur som är den mer generella termen. Därmed är programmet proceduralt men inte objektorienterat. Här har modulariseringen skett på funktionsnivå dvs *modul* är en funktion. Först när ett program är modulariserat på klassnivå – där *modul* är en klass – kan det kallas objektorienterat. Men vad är en klass?

Vår första klass

En nackdel av programmet **Procedure** ur modelleringssynpunkt är att **main()**:s lokala variabel **radie** är separerad från funktionerna **area()** och **omkrets()** – ett resultat av modulariseringen. Nackdel, därför att alla tre är saker och ting som

är relaterade till cirkeln, därför säger vi också ”ur modelleringssynpunkt”. Ur problemlösningssynpunkt är det nämligen ingen nackdel alls: Man har fått moduler som löser beräkningsproblemet. Men ur modelleringssynpunkt är det en nackdel: Ser man på datorprogram som en modell av verkligheten och i vårt fall på ”verkligheten” *cirkel* ska man helst avbilda en modell av cirkeln i sitt program. Man ska *beskriva* cirkeln så generellt som möjligt så att koden kan användas på alla tänkbara konkreta cirklar i hela världen. Denna modellering eller beskrivning av cirkeln måste vara så modulär och så generell att ingen – inklusive oss själva – skulle behöva återuppfinna hjulet och skriva kod för cirkeln i framtiden. Man ska kunna ta vår cirkelmodul och använda den när man behöver en cirkel i sitt progam. En sådan modul kallas för klassen **`Circle`**. I objektorienterad programmering kallar man kod som på ett generellt och modulärt sätt beskriver en kategori av saker och ting ur den reala världen, för *klass*. För att kunna skapa klassen **`Circle`** räcker det inte att flytta funktionerna **`area()`** och **`omkrets()`** från **`main()`** och separera dem från all annan kod utan även variabeln **`radie`** som på ett naturligt sätt tillhör modellen eller klassen **`Circle`**. Vi måste samla allt som är relevant för alla cirklar i en sådan klass. Följande kod som vi skriver i en headerfil och kommer att inkludera i programmet **`Obj_Oriented`** (längre fram) implementerar denna idé som är central för objektorienterad programmering:

```
// Circle.h
// Klass som beskriver kategorin "cirkel" som en abstrakt idé
// Modulariseringen är på klassnivå: "modul" = klass
// Innehåller all kod som är relaterad till en cirkel: data-
// medlemmen radie och metoderna area() och omkrets()
// "Metod" = funktion som är medlem i en klass (medlemsfunk-
// tion). Används som en mall i main() i filen ObjectOrien-
// ted.cpp för att skapa objekt (exemplar) av klassen Circle
class Circle                              // Modul Circle
{
public:
   float radie;                           // Datamedlem

   float area()                           // Metoden area()
   {
      return 3.14159 * radie * radie;
   }

   float omkrets()                        // Metoden omkrets()
   {
      return 2 * 3.14159 * radie;
   }
};
```

Om man jämför denna kod med headerfilen **`Procedure.h`** (sid 207) som var ett resultat av modularisering, ser man att det endast är några få rader som har kommit till: Variabeln **`radie`** har flyttats från **`main()`** och deklarerats här och det hela har fått en överordnad ”ram” med **`class`**. Ändå utgör dessa få ändringar ett sådant kvalitativt steg att det innebär övergången från procedural till objektorienterad pro-

grammering. Avgörande för det här steget är det reserverade ordet **`class`** som i C++ inleder deklarationen till en klass. Att vi kallar det *deklaration* och inte definition beror på att koden ovan inte reserverar en enda byte minne. Klassdeklarationen ovan beskriver *kategorin* cirkel som en abstrakt idé utan att skapa en verklig cirkel. Den är en *mall* för att skapa verkliga cirklar, en föreskrift om hur en verklig cirkel med en viss radie *skulle* se ut och hur dess area och omkrets *skulle* beräknas *om den skapades*. En verklig cirkel kallas för *objekt*. Det är objektet som behöver minnesutrymme för att lagras. Klassen definierar inga objekt utan ställer bara till förfogande modellen för framtida objektdefinitioner. Om man byter ut cirklar mot pepparkakor kan man säga att pepparkaks*formen* är klassen och själva pepparkakorna är objekten. Formen behöver ingen pepparkaksdeg – motsvarigheten till minne – den framställs bara en gång medan kakorna kan bakas i tusentals. Även klassen skriver man bara en gång, objekt kan skapas hur många som helst.

Vi har kallat klassen ovan för **`Circle`** – ett namn vi hittat på. Här följer vi förstås de vanliga namngivningsreglerna för identifierare som ställdes upp tidigare (sid 53). Självklart följer vi även rekommendationen vi gav där att välja namn som är beskrivande och återspeglar identifierarens roll i programmet. Men vi introducerar här ytterligare en konvention som vi kommer att använda i fortsättningen och som de flesta programmerare följer, nämligen att inleda klassnamn med versaler för att skilja dem från andra identifierare som variabler, funktioner osv. Så kommer namnet **`Circle`** till. Det reserverade ordet **`public:`** försett med kolon står i början av klassens kropp (utan indrag) för att kunna komma åt klassens innehåll från **`main()`**. Mer om detta senare.

Man skulle kunna se på klassen **`Circle`** som en fortsättning på modulariseringstanken: Inte bara funktionerna **`area()`** och **`omkrets()`** utan även variabeln **`radie`** har flyttats från **`main()`**. Anmärkningsvärt är att inte själva inläsningen av ett värde till **`radie`** (input) har flyttats till **`Circle`** utan deklarationen av variabeln **`radie`**. Inläsningen kan inte flyttas till klassen pga klassens karaktär som en mall för framtida cirklar. Alla cirklar ska ju inte ha samma radie. Detta visar att modularisering endast är en aspekt av objektorientering. Här kommer en annan aspekt in i bilden. Det är modelleringsaspekten – den nya synen på program som en modell: Varje cirkel *har* en radie. Radien är en beståndsdel av cirkeln. Att ha en radie är en *egenskap* eller ett attribut av alla cirklar. Egenskaper eller attribut av saker och ting som ska modelleras som en klass, brukar man kalla för klassens *datamedlemmar*. I denna bemärkelse modellerar vi **`radie`** som datamedlem i klassen **`Circle`**, dvs deklarerar **`radie`** inte längre som lokal variabel i **`main()`** utan flyttar deklarationen – utan att ändra syntaxen – till **`Circle`**. Så uppstår en datamedlem. Självklart kommer vi i fortsättningen inte längre gå omvägen över **`main()`** utan direkt modellera datamedlemmarna som egenskaper till den sak som ska skrivas som klass. Naturligtvis har en cirkel även andra beståndsdelar (egenskaper, attribut) som t.ex. medelpunkt eller, om man vill rita den på skärmen, färgen osv. som man skulle kunna ta in som datamedlemmar i klassen. Men just nu nöjer vi oss för enkelhetens skull med datamedlemmen **`radie`**.

Nästa fråga som modelleringen ställer är: Vad kan man *göra* med en cirkel? Vilka *operationer* är typiska för en cirkel? Vi har valt att modellera operationerna att beräkna arean och omkretsen. Även här är andra operationer tänkbara som t.ex. att förskjuta medelpunkten (positionen) till ett annat ställe eller att konstruera tangenten vid en viss punkt av cirkeln osv. Operationer eller funktioner som kan tillämpas på en sak som ska modelleras som en klass, brukar man kalla för klassens *metoder*. I denna bemärkelse definierar vi **`area()`** och **`omkrets()`** som metoder i klassen **`Circle`**. Som en naturlig konsekvens av modelleringen involverar metoderna datamedlemmarna: area- och omkretsberäkningen involverar radien.

Klasskonceptet har förutom modelleringsaspekten följande fördelar i koden:

1. **`area()`** och **`omkrets()`** kan komma åt **`radie`** direkt då alla är medlemmar i samma klass.
2. **`area()`** och **`omkrets()`** har inga parametrar. Parametrarna som var ett pris man fick betala för modulariseringen, behövs inte längre.

Generellt: Överföring av data mellan **`main()`** och funktioner som bearbetar data, behövs inte längre då funktionerna blivit metoder och kommer åt data direkt när dessa är medlemmar i klassen och inte längre lokala variabler.

Test av klass

Klassen **`Circle`** vore ingenting värt om man inte använde den i ett program. Klassen själv är inget program och kan inte köras. I följande program används klassen genom att skapa ett **`Circle`**-objekt av den läsa in ett värde till den samt anropa metoderna **`area()`** och **`omkrets()`** som beräknar just denna konkreta cirkelns area och omkrets:

```
// Obj_Oriented.cpp
// Programmet är objektorienterat då det skapas ett objekt av
// klassen Circle som deklarerats som en extern modul i
// headerfilen Circle.h och inkluderas nu här

#include <iostream>
using namespace std;
#include "Circle.h"                          // Klassen Circle

int main()                                   // Modul main()
{
   Circle myCircle;                         // Objekt myCircle

   cout << "Ange radien till en cirkel: ";
   cin >> myCircle.radie;                       // Input
                                                 // Output:
   cout << "\nEn cirkel med radien  " << myCircle.radie
        << "\nhar arean             " << myCircle.area()
        << "\noch omkretsen         " << myCircle.omkrets()
        << "\n\n";
}
```

Satsen som skapar objektet – den verkliga cirkeln **myCircle** – är:

```
Circle myCircle;
```

Syntaxen påminner om definitionen av en variabel. Vi kommer i nästa avsnitt att diskutera den här frågan och komma fram till att satsen ovan *är* en definition av variabeln **myCircle** vars datatyp skall vara klassen **Circle**. Men hur kan en klass vara en datatyp? Även detta tas upp i nästa avsnitt. Det som är relevant för oss nu är att satsen ovan allokerar minnesutrymme åt variabeln **myCircle** av den storlek som datatypen föreskriver. Men datatypen är ju klassen **Circle** som i sin tur har en datamedlem **radie** av typ **float**. Alltså allokeras 4 bytes för en **float** vilket gör det möjligt att läsa in och lagra ett värde till **radie** med satsen:

```
cin >> myCircle.radie;
```

Den här syntaxen för att komma åt ett objekts datamedlem som också förekommer i programmets **cout**-sats för metoderna, kallas *punktnotation* (sid 220).

Programmet **Obj_Oriented** ger samma utskrift när det körs, som **Procedure** och även **All_in_main**:

```
Ange radien till en cirkel: 1

En cirkel med radien  1
har arean             3.14159
och omkretsen         6.28318
```

Objekt och klass

Det är viktigt att inte blanda ihop dessa två begrepp. Deras skillnad kan jämföras med skillnaden mellan *variabel* och *datatyp*. Nästa avsnitt går närmare in på denna analogi. För att förstå skillnaden följer här en kort sammanfattning av det vi hittills lärt oss om objekt och klass:

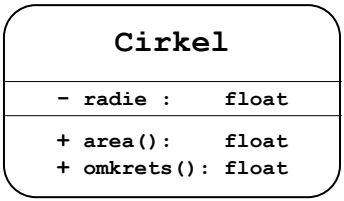
Ett av syftena med objektorienterad programmering är att efterlikna verkligheten så mycket som möjligt. Man vill kunna avbilda den reala världen, konstruera en modell av den i sina datorprogram för att kunna simulera verkligheten så noggrant som det går. Den reala världen, åtminstone den del som tillåter datorisering, består kort sagt av *objekt* och allt man kan *göra* med dessa objekt. *Hur* man gör och framför allt hur man beskriver det som skall göras, är föremål för algoritmer vilket behandlats tidigare. Men *vad* man sysslar med, dvs *objekt* som är involverade i algoritmerna och hur man beskriver dessa objekt, är en annan aspekt på saken. Objektorienterad programmering prioriterar objektaspekten framför algoritmaspekten. Visserligen kan man skriva **Circle**:s data **radie** på *ett* ställe och det man kan *göra* med cirkeln, metoderna **area()** och **omkrets()**, på ett *annat* ställe i sitt program. Men objektorienterad programmering gör inte så, utan sammanför till *en* **class** allt som är relevant för en cirkel, allt som är gemensamt för alla cirklar, allt som man skulle ta upp i en ordbok för att definiera *begreppet* cirkel. Man bildar

kategorin eller *klassen* **`Circle`** som på så sätt kan användas som modell, som form, som föreskrift, som mall för att i olika sammanhang skapa *objekt* av typen **`Circle`**. På samma sätt kan en enda pepparkaksform (klass) producera tusentals pepparkaksgubbar (objekt). Gubbarna kan skiljas från varandra i vissa detaljer, t.ex. materialet, smaken osv. Man kan t.o.m. måla dem i olika färger eller modifiera på annat sätt efteråt. De förblir pepparkaksgubbar av den ursprungliga formen. I formen ingår det som är gemensamt hos alla pepparkaksgubbar. Man har, när man framställde formen, bortsett från oväsentliga skillnader och tagit hänsyn endast till det väsentliga, det gemensamma.

Det handlar om samma tankeprocess som vi, när vi introducerade objektorienterade programmeringens termer, kallade för *abstraktion* som leder till *begrepps*bildning, till *klass*ificering eller *kategori*sering av den reala världen. När vi deklarerar klassen **`Circle`** abstraherar vi, tar upp endast det som är gemensamt hos *alla* verkliga cirklar i världen och bortser från deras skillnader. Gör man det blir kvar bara idén, begreppet, kategorin eller klassen ”cirkel”. Därför kallas klasser även för *abstrakta datatyper*. Tanken är inte ny. De som designat språket C (och andra språk) har redan använt den för att definiera språkets datatyper. Det nya är nu att även vi som programmerar, kan skapa våra egna datatyper för att beskriva nya objekt i den värld vi vill modellera och datorisera. Sammanfattningsvis:

> En klass är en ny sammansatt datatyp som skapas med **`class`**.

Det finns ingen begränsning på vilka, hur många eller vilka kategorier av datatyper man kan förena i en klass. Man kan t.o.m. involvera andra klasser i sin klass. Allt beror på modellen av den reala miljön man vill återspegla i sitt program. En sådan modell brukar man rita upp som ett diagram för att få en översikt över miljön samt som en plan för programmeringen. Ett verktyg som har utvecklats speciellt för objektorienterade modelleringar är UML som står för *Unified Modeling Language*. Enligt det här modelleringsspråket skulle vår första klass framställas så här:



I UML-diagram sätts symbolen **–** framför datamedlemmar och **+** framför metoder. En gång skapad kan en klass användas för att göra hur många objekt som helst av den:

> Ett objekt är en variabel vars datatyp är en klass.

En synonym till objekt är *instans*. En instans av klassen **`Circle`** är ett exemplar av kategorin, av typen **`Circle`**.

10.2 Klass som egendefinierad datatyp

Här följer vi den röda tråd som under begreppet *datatyp* går igenom hela boken. I nästan alla program hittills har vi använt datatyper för att skapa variabler. Att vi kunde göra det berodde på att det redan fanns *fördefinierade* datatyper i C++ som **int**, **char**, **float**, **double**, …. Vi började med dessa enkla och fortsatte sedan med sammansatta datatyper som arrays och pekare. Även de sista byggde i sin tur på fördefinierade datatyper. Nu får vi med klassbegreppet möjligheten att definiera helt nya egna datatyper och på så sätt utvidga språket. Men vad har klassbegreppet som i förra avsnitt introducerades som ett modulariserings- och modelleringskoncept, att göra med *datatyp*? På vilket sätt är den moderna synen på programmering förknippad med ett gammaldags verktyg som används för att skapa variabler?

Låt oss besvara frågan genom att gå tillbaka och ta upp den röda tråden från den första datatyp vi använde, nämligen **int**. Vad är det som gör **int** till en datatyp? Definitionen av *datatyp* säger att det handlar om hur en viss typ av data ska lagras i datorn, hur mycket minne den tar och vilka operationer man får utföra med data skapade med datatypen, här konkret **+**, **-**, *****, **/** och **%** (sid 51). Även explicit typkonvertering av en **float** till en **int** eller någon annan enkel datatyp, är en operation som är implementerad i datatypen (sid 81). Förutom minnesstorleken som är ett fast värde i antal bytes som måste lagras som ett konstant värde i datatypen, betyder allt annat vad som får *göras* med värden av en viss datatyp och *hur* allt detta ska göras. Det är – uttryckt i den objektorienterade programmeringens tremer – inget annat än *metoder* som är definierade för datatypen. Att samla data och metoder som är relaterade till dessa data, i en enhet, är samma koncept som ligger bakom klassbegreppet.

Man får akta sig att härifrån dra slutsatsen att alla datatyper är klasser. T.ex. är alla enkla datatyper inga klasser. Av alla fördefinierade datatyper som vi använt hittills är endast **string** en klass i C++. Däremot gäller det omvända: Varje klass definierar en ny datatyp. T.ex. visar satsen **Circle myCircle;** i programmet **Obj_Oriented** (sid 211) att **myCircle** skapas som en **Circle**, vilket är både deklaration och definition av variabeln **myCircle**. Förutsättning är förstås att man definierat klassen **Circle** innan. Om vi sedan pratar om *objektet* **myCircle** av *klassen* **Circle** eller om *variabeln* **myCircle** *av datatypen* **Circle** är bara två olika talessätt för en och samma sak. Men satsen **int no;** som också är både deklaration och definition av variabeln **no** skapar inte ett objekt, därför att **int** inte är en klass. Vi har att göra med två olika typer av variabler, sådana av enkel datatyp som **no** och sådana av klasstyp som **myCircle**.

Efter enkla datatyper behandlade vi arrays som är sammansatta datatyper. Både array och **class** sammansätter datatyper till en ny enhet. Dock har array vissa begränsningar som inte finns i **class**. En av dem är att man inte kan gruppera element av *olika* typer till en array. En annan är att man inte kan tilldela en array *direkt* utan måste göra det elementvis. Den största begränsningen dock är att man är tvungen att hålla sig till befintliga, fördefinierade datatyper, när man bildar en ar-

ray. Alla dessa begränsningar faller bort i **class**. Med **class** kan man gruppera *medlemmar* – motsvarigheten till element i array – av *olika* typer, närmare bestämt data av olika typer. Dessutom kan medlemmarna vara metoder som inte finns alls i array. **class** definierar *nya* datatyper. Tre steg måste tas när man använder **class**:

1. Deklaration av en klass
2. Definition av ett objekt
3. Åtkomst till objektets medlemmar

Med deklaration av en klass menas själva *koden* man skriver för klassen. Denna kod allokerar inget minne utan introducerar ett nytt begrepp i koden, namnet på en ny datatyp: Deklarationen av en klass definierar en ny datatyp. Med denna nya datatyp – låt oss kalla den för *klasstyp* – kan man sedan definiera variabler av klasstyp vilket till skillnad från klassdeklarationen allokerar minne. Vi går igenom alla tre steg:

1. Deklaration av en klass

Med hjälp av det reserverade ordet **class** kan en klass generellt deklareras så här:

```
class Datatyp
{
    Deklaration av datamedlemmar;
    Deklaration eller definition av metoder(;)
};          // OBS! Semikolon obligatoriskt
```

Datatyp är ett namn som vi kan välja fritt med hänsyn till de kända regler och rekommendationer som gäller för all namngivning (sid 53) samt konventionen att inleda det med en versal. Sedan kan vi använda namnet som datatyp i programmet – med alla de ”rättigheter” som de fördefinierade datatyperna har. Med koden ovan *skapas* den nya datatypen. Pga denna speciella styrka betecknas **class** själv inte längre som datatyp utan som *datastruktur* då den fungerar på en kvalitativt högre nivå. Observera att hela klassdeklarationen efter den avslutande klammern avslutas med semikolon vilket visar att vi har att göra med en deklarations*sats*. Till skillnad från funktionsdefinitioner ersätter den avslutande klammern inte semikolonet: Utelämnas semikolon får man kompileringsfel. Med det semikolonet däremot som står inom runda parenteser menas att det beror på om man skriver en deklaration eller en definition av en metod. Deklaration kräver semikolon, definition inte. I vårt första klassexemplet **Circle** (sid 209) hade vi valt att skriva metodernas definition. I nästa exempel kommer vi att deklarera en metod.

I filen **Employee.h** (nästa sida) definieras den nya datatypen eller klassen **Anstalld** som har fyra datamedlemmar **namn**, **personnr**, **timlon** och **antalTimmar** där den första är en pekare, den andra en array av **int** med **2** element, den tredje och fjärde vanliga **double**:s. Observera att syntaxen för deklarationen av datamedlemmarna är precis som i vanliga deklarationssatser för variabler. Man ser också att man i en klass – till skillnad från array – kan blanda data av helt olika datatyper, både enkla, sammansatta och pekartyper. Man kan t.o.m. ha datamedlem-

mar som i sin tur är av egendefinierade typer dvs objekt. Sedan har den nya datatypen även en metod **lon()** som endast är deklarerad (med huvudet). Kroppen definieras separat. Datamedlemmarna och metoden står inom klamrarna enligt den generella beskrivningen ovan med avslutande semikolon.

```
// Employee.h
// Deklarerar klassen Anstalld med 4 datamedlemmar och 1 me-
// tod. Metoden lon() deklareras här och definieras i Emplo-
// yee.cpp. Samtidigt definieras den nya datatypen Anstalld
// som användsmi programmet EmployeeTest för att skapa varia-
// bler (objekt) av typen Anstalld

class Anstalld
{
public:
   char *namn;
   int personnr[2];
   float timlon, antalTimmar;

   double lon();
};
```

Varför står här endast deklarationen av metoden **lon()**? I modulariseringens anda brukar man faktiskt ofta separera metodernas deklaration från deras definition, speciellt i större applikationer för att kunna använda samma deklaration med olika definitioner i olika program. Deklarationen skrivs och lagras i headerfiler, medan definitionen skrivs separat och lagras i en **cpp**-fil i vilken klassheaderfilen inkluderas. Med det här förfarandet kan man separatkompilera koden i **cpp**-filen, vilket har praktiska fördelar, när man utvecklar stora program och vill testkompilera sina moduler en i taget. Återstår frågan: Hur hittar metodernas kroppar sina huvud? Svaret är: Med hjälp av *räckviddsoperatorn* **::** . Så här kan det se ut:

```
// Employee.cpp
// Definierar metoden lon() deklarerad i klassen Anstalld
// Klassens namn hämtas med räckviddsoperatorn
// Kan kompileras separat

#include "Employee.h"                    // Inkluderar klassen

double Anstalld::lon()                   // Samma metod som i
{                                        // klassen Anstalld
   double overtid = (antalTimmar - 180) * timlon * 1.5;
   if (antalTimmar <= 180)               // Ingen övertid
      return timlon * antalTimmar;
   else
      return 180*timlon + overtid;       // Övertid
}
```

Metoden **lon()** som deklareras i klassen **Anstalld** på förra sidan definieras separat genom att inkludera klassens headerfil och lägga till metodens både huvud och

kropp, men i huvudet specificera att det är samma metod som deklarerats i klassen. Detta gör man genom att med räckviddsoperatorn hämta metodens namn från klassen **`Anstalld`** som anses här som ett slags övre block då det är deklarerat i det s.k. globala namnutrymmet utanför **`main()`**. Därför ser huvudet till metoden **`lon()`**:s definition ut så här:

```
double Anstalld::lon()
```

Man kan också tvärtom säga att räckviddsoperatorn lyfter metodens definition in i klassen **`Anstalld`** och förbinder det med huvudet som finns där. Observera att returtypen kommer som vanligt först och sedan metodens namn som får ”prefixet” **`Anstalld::`**

Det som står i kroppen är en löneberäkningsrutin som man brukar använda på timanställda för att ta hänsyn till ev. övertidsarbete. Antar man månaden som en tidsenhet för löneutbetalning och **`180`** timmar för en normal arbetstid per månad, har i rutinen ovan allt som överstiger denna tid, ansetts som övertid som betalas med en **`1.5`** gånger så stor timlön som den ordinarie timlönen.

2. Definition av ett objekt

När en klass definierar en ny datatyp kallas den nya datatypen för en *klasstyp*. Objekt av denna klass kan då anses som *variabler av klasstyp*. Att definiera ett objekt är således samma sak som att definiera variabler av klasstyp. Följande program demonstrerar definition av objekt genom att definiera variabler av klasstypen **`Anstalld`**:

```
// EmployeeTest.cpp
// Använder klassen Anstalld för att skapa en anställd, ändra
// dess lön och skriva ut den gamla & nya lönen & skillnaden
// Definition och initiering av objekt i en och samma sats
// Mäter storleken till datatypen Anstalld med sizeof
#include <iostream>
using namespace std;

#include "Employee.cpp"                  // Inkluderar cpp-fil
                                         // som i sin tur in-
int main()                               // kluderar headerfil
{
   Anstalld anst;                        // Definierar objekt
   anst.namn        = "Anders Larsson"; // Initierar objekt
   anst.personnr[0] = 590714;
   anst.personnr[1] = 2493;
   anst.timlon      = 110;
   anst.antalTimmar = 190;

   Anstalld copy = anst;                 // Definition och
                                         // initiering i en
                                         // och samma sats
   copy.timlon = anst.timlon * 1.15;     // Löneförhöjning
```

```
   cout << "\n\t" << anst.namn << ", personnr "
        << anst.personnr[0] << "-" << anst.personnr[1]
        << "\n\n\t" << "Gammal lön:\t" << anst.lon()
        << "\n\n\t" << "Ny lön:\t\t" << copy.lon()
        << "\n\n\tVåra lönekostnader kommer att öka med "
        << copy.lon() - anst.lon() << " kr\n\n";

   cout << "Klassen Anstalld föreskriver "<< sizeof(Anstalld)
    << " bytes för sina objekt, däför att \n"
    << "den sammansätter datatyperna:\t"
    << "char* med   " << sizeof(char*) << " bytes\n\t\t\t\t"
    << "int[2] med  " << sizeof(int[2]) << " bytes\n\t\t\t\t"
        << "2xfloat med " << 2*sizeof(float) << " bytes\n\n";
}
```

Programmet **EmployeeTest** skapar två objekt av den nya, egendefinierade datatypen **Anstalld**: det första kallas **anst**, det andra **copy**. Båda är framhävda med vit bakgrund i programmet. Det första skapas med satsen:

```
Anstalld anst;
```

som kan jämföras med:

```
int a;
```

Här kan man direkt se analogin mellan *datatyp* och *klass*. Datatypen **int** skapar variabeln **a**. Då den är fördefinierad i C++ behöver vi inte göra det. Satsen allokerar minnesutrymme åt variabeln **a**, där **int** föreskriver storleken. På samma sätt skapar datatypen **Anstalld** variabeln **anst**. Då den inte är fördefinierad har vi definierat den på sid 216. Satsen **Anstalld anst;** allokerar minnesutrymme åt variabeln **anst**, där **Anstalld** föreskriver storleken. Ändå är den enkla datatypen **int** ingen klass, men klassen **Anstalld** är en datatyp som har exakt samma ”rättigheter” som vilken annan datatyp som helst. Man kan definiera variabler av denna klasstyp som blir objekt. Generellt gäller:

> Objekt kan skrivas överallt i ett program där en vanlig variabel kan stå.

T.ex. kan man slå ihop definition och initiering av ett objekt till en och samma sats:

```
Anstalld copy = anst;
```

precis som hos vanliga variabler:

```
int b = a;
```

I kapitel 4 (sid 57) hade vi introducerat denna teknik för enkla datatyper, vilket inte kunde utvidgas till arrays då de måste tilldelas elementvis. Men nu återupptas den röda tråden av objekt. En förutsättning för satserna ovan är förstås att variablerna höger om tilldelningstecknet är definierade, vilket räcker till för kompilering. Om **anst** och **a** dessutom är initierade kommer de vid exekvering att föra över sina värden till **copy** och **b**, annars blir det skräpvärden som tas över från de oinitierade

variablerna (sid 58). Prova gärna genom att bortkommentera datamedlemmarnas initiering: Man kan fortfarande kompilera, men får skräpvärden vid exekveringen.

Datatyptest med `sizeof`

När vi nu i fortsättningen pratar om *variabler av datatypen* **`Anstalld`** eller om *objekt* av *klassen* eller *typen* **`Anstalld`**, då är det bara olika sätt att uttrycka en och samma sak: För att skapa ett objekt (en variabel), måste *minnesutrymme reserveras* av den storlek som klassen (datatypen) föreskriver. Detta för att kunna lagra *värden* i objektet, för det är det enda som är praktiskt relevant: Existensen av en variabel eller ett objekt i programmet är identisk med existensen av minnesutrymme i datorns RAM. För att få information om hur mycket minne t.ex. ett objekt av typ **`Anstalld`** behöver, måste vi titta – och det gör även kompilatorn – i klassen **`Anstalld`**:s deklaration (sid 216) . Där finns som datamedlemmar pekaren **`namn`** som tar 4 bytes, två **`int`**-arrayelement som båda tar 8 samt två **`float`**:s med 4 var dvs 8 bytes, så att sammanlagt 20 bytes allokeras i minnesutrymme åt objeket **`anst`**. Denna minnesstorlek kan mätas med operatorn **`sizeof`**. **`sizeof`** returnerar antalet bytes operanden tar i minnesutrymme. Operand är det som skrivs i parentesen och kan vara en datatyp, en variabel eller ett uttryck. Med **`sizeof`** kan vi testa det vi sa om klasser, nämligen att de kan skrivas överallt i programmet där även en vanlig datatyp kan stå. Således kan även klasser skrivas som operander i **`sizeof`**. Vir har testat detta i slutet av programmet **`EmnployeeTest`** med:

```
sizeof(Anstalld)
```

Lika bra skulle vi kunna skriva **`sizeof(anst)`** dvs sätta in objektet eller variabeln som operand. I båda fall får vi det förväntade värdet 20 bytes vilket en körning visar:

```
        Anders Larsson, personnr 590714-2493

        Gammal lön:     21450

        Ny lön:         24667.5

        Våra lönekostnader kommer att öka med 3217.5 kr

Klassen Anstalld föreskriver 20 bytes för sina objekt, däför
att den sammansätter datatyperna:   char* med 4 bytes
                                   int[2] med 8 bytes
                                  2xfloat med 8 bytes
```

Detta visar än en gång att klasser är datatyper, sammansatta av alla möjliga datatyper (enkla, arrays, pekare, …) och objekt är variabler av klasstyp. Även klasser kan vara datamedlemmar i en annan klass. Då pratar man om nästling eller *komposition av klasser* vilket kommer att behandlas senare i detta kapitel. Värdet 20 bytes som returneras av **`sizeof`** visar att i det här exemplet storleken som klassen **`Anstalld`** föreskriver för sina objekt är lika med summan av storlekar av klassens datamedlemmar. Men generellt gäller att objektets storlek är *mindre än eller lika*

med summan av alla datamedlemmars storlekar. Dvs C++ kompilatorn reserverar ibland mer minne, vilket har att göra med hanteringen av ordlängder i RAM.

3. Åtkomst till objektets medlemmar

Efter att ha skapat ett objekt vill man kunna arbeta med objektets medlemmar. När man generellt pratar om medlemmar måste man skilja mellan två typer av medlemmar, *data*medlemmar och *metoder* (medlemsfunktioner). Därför skulle rubriken ovan – mer exakt – lyda: Att komma åt objektets datamedlemmar och *att anropa* objektets metoder. För båda ändamål används en och samma teknik som redan nämnts i olika sammanhang och som vi tar upp nu i detalj:

Punktnotation

Som redan tidigare nämnts betyder *notation* sättet att skriva. Sättet att skriva kod för att komma åt både ett objekts datamedlemmar och metoder kallar vi för *punktnotation* [*]. Om vi tar vårt exempel med objektet **anst** av typ **Anstalld** har vi redan sett att definitionssatsen **Anstalld anst;** skapar objektet **anst** genom att allokera minne åt det. Vill vi sedan *efter* denna sats fylla minnet med data dvs tilldela objektet **anst**:s datamedlemmar värden, kan vi skriva:

```
anst.namn        = "Anders Larsson";
anst.personnr[0] = 590714;
anst.personnr[1] = 2493;
anst.timlon      = 110;
anst.antalTimmar = 190;
```

Namnet till den anställda **anst** ska vara **Anders Larsson** osv. **namn** är en datamedlem i objektet **anst** och inte en fritt tillgänglig variabel. Objektet **anst** kan jämföras med en behållare som innehåller medlemmar bl.a. medlemmen **namn**. För att komma åt **namn** måste vi först öppna behållaren **anst**. Sättet i koden att komma åt datamedlemmen **namn** är att *först* skriva objektets namn, sedan en punkt och sist medlemmens namn. Samma sak gäller för de andra datamedlemmarna **personnr**, **timlon** och **antalTimmar**. Punktnotation förutsätter förstås objektets existens dvs kan endast användas efter att objektet skapats med:

```
Klassnamn objektnamn;   // Definition av objekt
```

Då ser punktnotation ut så här:

```
objektnamn.datamedlem     // Åtkomst till obj.s medlem
```

Till vänster om punkten måste alltid finnas namnet på ett objekt och till höger någon datamedlem tillhörande *detta* objekt. Punktnotation skrivs för att *referera* till just detta objekts datamedlem och kan därför användas antingen för att tilldela

[*] En annan beteckning av punkten som skiljer objektet från medlemmen, är *medlemsåtkomstoperator* (eng. *member access operator*). Även termen *member selector operator* förekommer i litteraturen. Pga den språkligt lite tunga översättningen föredrar vi dock punktnotation.

den ett värde (skriva till minnescellen) eller för att hämta värdet (läsa från minnescellen). Satserna ovan är rena tilldelningssatser, medan punktnotationen som står i programmets första **cout**-sats hämtar värdena och skriver ut dem.

Vid sidan av punktnotation finns det andra sätt att initiera objekt direkt vid skapandet. Ett av dem är *konstruktorn* som kommer att behandlas i övernästa avsnitt.

När en datamedlem är en array kombineras punktnotation med arraynotation på ett naturligt sätt och får följande form:

```
anst.personnr[0]
anst.personnr[1]
```

Hakparenteserna beror på att datamedlemmen **personnr** enligt definition av datatypen **Anstalld** (sid 216) – som **anst** är ett objekt av – är en array med 2 element. Index **0** refererar till det första element i objektet **anst**:s array-datamedlem **personnr** osv.

Anrop av metoder

Samma teknik används i princip på ett objekts metoder. När objektet **anst** av typ **Anstalld** skapats kan vi anropa metoden **lon()** även med punktnotation. Skillnaden är bara att efter punkten skrivs ett vanligt anrop av metoden istället för datamedlemmen:

```
anst.lon()
```

Metoden **lon()** anropas i objektet **anst** enligt definitionen i klassen **Anstalld**. Då **lon()** är en metod och inte en fritt tillgänglig funktion, måste man först (*före* punkten) referera till objektet för att sedan (*efter* punkten) kunna anropa metoden i detta objekt. Generellt har anropet av en metod i ett objekt som redan skapats, en syntax som liknar den för åtkomst av datamedlemmar:

```
objektnamn.metodanrop   // Anrop av obj.s metod
```

Körresultatet av programmet **EmployeeTest** (sid 219) visar att metoder inte allokerar minnesutrymme i objektet. När objektet skapas allokeras minne endast för datamedlemmar, inte för metoder. De är bara *deklarerade* i klassen och deklarationen skapar inget minne. Först när funktionen anropas, allokeras minne åt de parametrar och variabler som är involverade i metoden. Men detta sker inte i objektet utan i det program som anropar metoden. En närmare titt på metoden **lon()**:s definition (sid 216) visar att **lon()** inte har några parametrar. Men den har en lokal variabel **overtid** som definieras i metoden, dvs skapas vid varje anrop och ”dör” direkt efter anropet. Dessutom involverar metoden **lon()** datamedlemmarna **timlon** och **antalTimmar** som vid anropet tas från objektet **anst**. Därför säger vi att metoden **lon()** anropas *i* objektet **anst** och har därmed direkt tillgång till datamedlemmarna. Det är också därför de får skrivas i metoden **lon()**:s kropp utan punktnotation. Båda befinner sig inuti objektet och har tillgång till varandra direkt. De är medlemmar i samma klubb – ”insiders” så att säga – och kan därför

hälsa varandra utan att ange klubbens namn. Även om de hade förekommit i parameterlistan hade de angetts utan punktnotation. Punktnotation måste och får användas endast utanför objektet.

Då **`lon()`** är en metod med returvärde, måste ett meningsfullt anrop bakas in antingen i en **`cout`**- eller tilldelningssats. I **`EmployeeTest`** finns anropet i en **`cout`**-sats.

Diskussionen kring metoder har fler aspekter än de som vi hann ta upp i detta avsnitt. Därför ägnar vi nästa avsnittet åt metoders andra egenskaper. En av dem är att kunna hantera även objekt som parameter och returvärde.

10.3 Metoder

Metoder är funktioner som är definierade i klasser. Det enda som skiljer dem från vanliga funktioner är deras placering i programmet. I C++ kan funktioner stå globalt, precis som globala variabler. Det bästa exemplet är själva **main()**-funktionen. Ett C++ program kan börja med att definiera en funktion. I denna bemärkelse är alltså funktioner helt fristående delar av ett C++ program, vilket man inte kan påstå om metoder. Den enda begränsning som gäller för placeringen av funktioner är att de inte får stå inuti en annan funktion. Deras definitioner får inte nästlas i varandra. Den regeln gäller även för metoder. Att göra nästlade anrop av funktioner är någonting helt annat.

Metoder kallas ibland även *medlemsfunktioner*. De är inte fristående utan delar av en klass och därmed delar av alla objekt som skapas av denna klass, precis som datamedlemmarna. Deras definition är alltid inkapslad i klasser, varför de utanför klassen endast kan anropas med punktnotation. Metoder kan inte definieras globalt i ett C++ program. Däremot är klassen i vilken de är inkapslade, globalt deklarerad.

Exempel på egendefinierade metoder är **lon()** definierad i klassen **Anstalld** samt metoderna **area()** och **omkrets()** i klassen **Circle**. Fördefinierade metoder som vi hittills använt i våra program utan att definiera dem själva, har vi haft många exempel på: **getline()** definierad i **cin** (sid 170), **eof()** och **get()** i **ifstream**, **close()** i både **if-** och **ofstream** osv. Det gemensamma hos alla dessa metoder var att de hade – om överhuvudtaget några – endast enkla datatyper som parametrar och returvärden. För att studera metodernas ytterligare objektorienterade egenskaper ska vi nu ta upp ett exempel där en metod både tar in ett objekt som parameter och returnerar ett objekt. Att det är möjligt, är ytterligare ett exempel på att objekt kan skrivas överallt i programmet där även en vanlig variabel kan stå, därmed även i en funktions parameterlista och som funktionens returvärde. Låt oss se på denna möjlighet i en ny klass:

```
// TravelTime.h
// Deklarerar klassen Restid med 2 datamedlemmar och en metod
// sum() vars parameter & returvärde är objekt av typ Restid
class Restid
{
public:
   int tim, min;

   Restid sum(Restid t)
   {
      Restid temp;
      temp.min = (min + t.min) % 60;
      temp.tim = (tim + t.tim) + (min + t.min)/60;
      return temp;
   }
};
```

Objekt som parameter och returvärde

Klassen Restid modellerar tiden, närmare bestämt restiden där det – t.ex. för en handelsresande – kan vara relevant att summera sina restider under en viss period. I detta sammanhang är det inte av betydelse att modellera restidens sekunder. Så, klassen Restid har endast datamedlemmarna **`tim`** och **`min`**. Summering av tider görs i metoden **`sum()`**.

Två programmeringstekniskt nya moment kan observeras här:

1. Metoden **`sum()`** har både som parameter och som returvärde ett objekt.
2. Dessa objekt är av typ **`Restid`**, dvs samma nya datatyp som vi håller på att definiera.

Det första följer av egenskapen att objekt kan skrivas överallt i programmet där även en vanlig variabel kan stå (sid 218). Variabler har hittills varit det vanliga som parametrar och returvärden. Så, varför inte objekt? Från applikationens synpunkt verkar det vara naturligt att restider kommer in som summander (input) i en algoritm som summerar tider och att även resultatet dvs summan av två restider blir en restid (output) eller åtminstone en tid, dvs ett objekt bestående av datamedlemmarna **`tim`** och **`min`**.

Punkt 2 är mindre självklart, särskilt med tanke på att vi befinner oss i klassen **`Restid`** när vi i metoden **`sum()`**:s parameterlista med **`Restid t`** skapar objektet **`t`** och i metodens kropp skapar objektet **`temp`**. Man kan ju misstänka att den nya datatypens definition inte är klar än, hur kan man då använda den redan? Svaret är: De avgörande byggstenarna vid definition av en ny datatyp är datamedlemmarna, inte metoderna. När vi definierar metoden **`sum()`** finns datamedlemmarna **`tim`** och **`min`** redan. Därför kan vi skapa objekt av typ **`Restid`** i metoden **`sum()`**. Vi skulle inte kunna göra samma sak bara en rad ovanför metoden **`sum()`**:s definition, bland datamedlemmarna. Ett försök att t.ex. med **`Restid s;`** skapa ett objekt som en ny datamedlem i klassen skulle generera följande kompileringsfelmeddelande:

. . . 's' : uses 'Restid', which is being defined.

vilket visar att klassens – dvs datamedlemmarnas – konstruktion inte är klar än i det här stadiet. Vi kan inte använda modulen **`Restid`** som vi håller på att bygga. Men sedan, när listan över alla datamedlemmar är komplett, kan vi använda den nya datatypen **`Restid`** i metoden **`sum()`**. (Testa gärna!)

Metoden **`sum()`** adderar klassens datamedlemmar med parametern **`t`**:s datamedlemmar och lägger resultatet i ett temporärt objekt **`temp`** av typ **`Restid`** som sedan returneras. Algoritmen som används för summeringen simulerar det man gör när man adderar två tider manuellt: Först adderas minuterna: **`(min + t.min) % 60`**, men för att hamna under **`60`** tar vi bort de hela timmarna från minuternas summa genom att räkna modulo **`60`** (sid 66). Sedan adderas timmarna: **`(tim + t.tim)`**. Sist lägger vi till de hela timmar som tagits bort från minuternas summa **`(min +`**

t.min)/60 där **/** utför heltalsdivision pga datatypen **int** på bägge sidor av operatorn **/**.

Man kan ju undra, varför metoden **sum()** endast har en parameter. Med tanke på att den adderar två tider borde den ta in två restider som input via två parametrar. Summan tas sedan hand av returvärdet som output. Men ett sådant resonemang tar inte hänsyn till att **sum()** är en metod och ingen funktion. Resonemanget är typiskt för procedural programmering. Hade **sum()** varit en fristående funktion, hade den säkert behövt två parametrar för summans två summander. Men metoden **sum()** kan inte anropas fristående utan bara i ett objekt av typ **Restid**. Detta objekt måste vara initierat när **sum()** anropas dvs dess datamedlemmar måste redan ha värden. Objektet kan användas som summans ena summand. Den andra summanden kan skickas som parameter. Därför räcker det med en parameter.

Att det också är rimligt att förse **sum()** med endast en parameter visar följande program som testar klassen **Restid** och anropar **sum()** för att addera fler än två restider:

```
// TravelTimeTest.cpp
// Använder klassen Restid för att skapa 3 objekt av den. De
// tre restiderna adderas genom att anropa metoden sum() två
// gånger. Alternativt: Nästlat anrop av sum()
#include <iostream>
using namespace std;
#include "TravelTime.h"

int main()
{
   Restid tisdag;
   Restid onsdag;
   cout << "Ange timmar och minuter till tisdagsresan: ";
   cin >> tisdag.tim >> tisdag.min;
   cout << "Ange timmar och minuter till onsdagsresan: ";
   cin >> onsdag.tim >> onsdag.min;

   Restid tvadagsresa = tisdag.sum(onsdag)// 1:a anrop av sum
   cout << "\n\tTvå dagars resa tog " << tvadagsresa.tim <<
        " timmar och " << tvadagsresa.min << " minuter.\n\n";

   Restid torsdag;
   cout << "Ange timmar och minuter till torsdagsresan: ";
   cin >> torsdag.tim >> torsdag.min;

   Restid tredagsresa = tvadagsresa.sum(torsdag);// 2:a sum-
   // Alternativt: Nästlat anrop                    anrop
   // Restid tredagsresa = tisdag.sum(onsdag.sum(torsdag));

   cout << "\n\tTre dagars resa tog " << tredagsresa.tim <<
        " timmar och " << tredagsresa.min << " minuter.\n\n";
}
```

Här skapas först två objekt **tisdag** och **onsdag** av typ **Restid** och initieras genom att läsa in värden till deras datamedlemmar **tim** och **min**. Att skapa och initiera objekt i två separata steg är inte optimalt, men vi gör det än så länge tills vi i nästa avsnitt lär oss att skriva objektets definition och initiering i en enda sats. När objekten **tisdag** och **onsdag** är väl definierade adderas de i följande anrop av metoden **sum()**:

```
tisdag.sum(onsdag)
```

analogt till:

```
tisdag "+" onsdag
```

som om man adderade två vanliga tal med varandra. Man ser direkt att det här skrivsättet ökar kodens läslighet avsevärt och bekräftar att det var rimligt att förse metoden **sum()** med endast en parameter. Resultatet av ”additionen” dvs **sum()**:s returvärde, läggs i ett tredje objekt **tvadagsresa** av typ **Restid** i samma sats som anropet sker:

```
Restid tvadagsresa = tisdag.sum(onsdag);
```

Det kan vi göra därför att metoden **sum()** enligt definition (sid 223) returnerar ett objekt av typ **Restid**. Här sker definitionen och initieringen av objektet **tvadagsresa** i en enda sats precis som man definierar och initierar vanliga variabler i en enda sats (sid 57). Det är möjligt, därför att **tvadagsresa** tar emot ett helt objekt: returvärdet av **sum()**.

När man skriver en klass som ska modellera restider vill man ju att den är så generell som möjligt så att den t.ex. kan addera inte bara två utan flera restider. Det gäller även för vår klass **Restid**. Faktiskt kan metoden **sum()** addera flera restider fast den adderar två restider åt gången. Det finns generellt två möjligheter att låta en funktion som verkar på två operander, att göra det även på flera:

1. Upprepat eller kedjeanrop.
2. Nästlat anrop.

1. Den första möjligheten används i programmet **TravelTimeTest** genom ett andra anrop av metoden **sum()** i satsen:

```
Restid tredagsresa = tvadagsresa.sum(torsdag);
```

där **torsdag** är ett **Restid**-objekt som skapats och initierats innan. Vi anropar **tvadagsresa**-objektets **sum()**-metod för att lägga till den första summan som hade bildats vid **sum()**:s första anrop, **torsdag**ens restid. Den fösta summan hade adderat **tisdag**ens och **onsdag**ens restider och lagt resultatet i **tvadagsresa**. Nu adderar vi **tvadagsresa**ns och **torsdag**ens restider och lägger resultatet i **tredagsresa**.

2. Den andra, alternativa möjligheten för att addera flera restider med **sum()** är nästlat anrop som i programmet **TravelTimeTest** är bortkommenterat och skulle ge exakt samma resultat som upprepat eller kedjeanrop. Det skulle se ut så här:

```
Restid tredagsresa = tisdag.sum(onsdag.sum(torsdag));
```

analogt till: `tisdag "+" onsdag "+" torsdag`

Testa gärna detta alternativ i programmet **`TravelTimeTest`** för att få en förståelse om hur nästlade anrop generellt fungerar och hur de måste kodas. Det viktigaste i funktionssättet av nästlade anrop är regeln:

> Nästlade anrop av funktioner eller metoder exekveras alltid "inifrån".

Dvs först läggs ihop restiderna **`onsdag`** och **`torsdag`** i satsen ovan. Sedan adderas restiden **`tisdag`** till denna summa. Anledningen till den här ordningsföljden är att metoden **`sum()`** måste ha ett värde i sin parameterlista för att kunna utföras. Således måste det inre anropet vara klart innan det ytre anropet kan ge resultat.

En körning av programmet **`TravelTimeTest`** kan ge följande utskrift:

```
Ange timmar och minuter till tisdagsresan: 3 45
Ange timmar och minuter till onsdagsresan: 5 25

        Två dagars resa tog 9 timmar och 10 minuter.

Ange timmar och minuter till torsdagsresan: 2 57

        Tre dagars resa tog 12 timmar och 7 minuter.
```