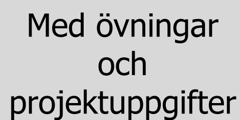
Programmering 2

med C++



Titel: Programmering 2 med C++

Författare: Taifun Alishenas
info@taifun.se

Copyright © 2024 Lieta Förlag AB
All rights reserved

Februari 2024

Kopieringsförbud!

Denna bok är skyddad av *Lagen om upphovsrätt*. Kopiering är förbjuden. Förbudet inkluderar översättning, tryckning, stencilering, kopiering, lagring i elektroniska och digitala media, visning på bildskärm eller via projektor, bandinspelning osv. Dessa förbud gäller även för koden i alla programexempel samt övningarnas lösningar som finns i boken. Den som bryter mot lagen om upphovsrätt kan åtalas av allmän åklagare och dömas till böter eller fängelse i upp till två år samt bli skyldig att erlägga ersättning till upphovsman/rättsinnehavare.

Innehåll

Ämne	Sida	Program
Kapitel 1 Olika programmeringsmiljöer	**7**	
1.1 Om C/C++ och Python	8	
- Standardisering	10	
- Unicodes historia	10	
- Om Python	10	
1.2 En Open Source IDE: Borland C++	12	`CppDemo`
1.3 Att bygga en egen IDE med en editor	15	
- Att koppla Java till TextPad	16	
- Att köra program från en editor	16	`EditDemo`
1.4 Att köra program från kommandofönstret	18	
- Kommandotolken	18	
- Konfiguration av Path	20	
Frågor till kap 1	22	
Övningar till kap 1	24	
Kapitel 2 Fortsättning med C++	**25**	
2.1 Utskrift till en grafisk miljö i C++	26	`MessageBoxB`
2.2 Olika beteenden i olika miljöer	28	`MessageBoxVS`
- Vad är LPCWSTR?	29	
- Parentes om Java	29	
2.3 Unicode	30	`Unicode.java`
- Att arbeta med Unicode	32	`Char2Int.java`
2.4 Svenska tecken i olika miljöer	35	`MessageBoxSw`
- Konsolens hela teckenuppsättning i C++	36	`AsciiFor`
2.5 Radfortsättning	39	`LineContin`
- Radfortsättningstecknet \	39	
2.6 Objektorienterad initiering	41	`ObjInit`
2.7 Automatisk typkonvertering	43	`AssignRule`
- Tilldelningsregeln	43	`Overflow`
- `int`-regeln	46	`IntRule`
- Befordringsregeln	47	`PromotInt/Dec`
2.8 Rekursion	50	`Fibonacci`
2.9 Mer om flervägsval	54	
- Luriga `else`	54	`TrickyElse`
- Korrekt `else`	56	`CorrectElse`
- `switch` med tomma `case`-satser	57	`SwitchInequ`
2.10 Misslyckad modularisering	59	`MiniSort`
- Försök att modularisera `MiniSort`	60	`NoSort`
2.11 Referenser	62	`Reference`
- Vad är en referens?	62	

Ämne		Sida	Program
	- Två olika betydelser av ampersand (&)	63	
	- Referens vs. pekare	64	`PointRef`
2.12	Parameteröverföringsmetoder	66	
	- Värdeanrop (Call by value)	66	`CallByValue`
	- Referensanrop (Call by reference)	68	`CallByRef`
	- Två olika minnesbilder	71	`Swapping`
2.13	In- och utparametrar	72	`change()`
2.14	Överlagring av funktioner	75	`power()`
		77	`Overload`
Övningar till kapitel 2		78	
Kapitel 3	**Klasser**	**79**	
3.1	Vad är objektorienterad programmering (OOP)?	80	
	- OOP:s tre hörnstenar	83	
	- Klassdiagram	84	
3.2	Vägen till objektorienterad programmering	87	`All_in_main`
	- Modularisering på funktionsnivå	88	`Procedure`
	- Modularisering på klassnivå	90	
	- Vår första klass	90	`Circle`
	- Test av klass	92	`CircleTest`
	- Klassbegreppet	93	
	- Objekt och klass	94	
3.3	Inkapsling	96	
	- Åtkomstmodifieraren private	96	
3.4	Konstruktor	98	
	- Klassens konstruktor	98	`CircleConstr`
	- Default konstruktorn	101	`Encapsulation`
	- Flera konstruktorer	102	`Circles`
		104	`MoreConstr`
	- Objektorienterad initiering	105	`ObjInit`
3.5	Accessmetoder	107	`Emp/Access`
3.6	Klass som egendefinierad datatyp	109	
	Deklaration av en klass	110	`Anstalld`
	- Definition av ett objekt	113	`EmployeeTest`
	- Datatyptest med sizeof	114	
3.7	Metoder i OOP	118	`TravelTime`
	- Objekt som parameter och returvärde	118	`Travel_Test`
Övningar till kapitel 3		123	
Kapitel 4	**Logik för blivande programmerare**	**126**	
4.1	Logiska operatorer	127	`AND_OR`
	- Sanningstabeller	129	`Cross`
	- Visualisering av logiska operatorer	131	`NegativeCross`
4.2	Datatypen `bool`	135	`TruthTab`

Ämne	Sida	Program
- Automatisk typkonvertering till `bool`	136	
4.3 NEGATION som logisk operator	137	`GuessNEG`
- Logiska uttryck	138	
4.4 Testa lösenord med logiska lagar	139	`Passwd`
- Caps Lock-problematiken	140	
- De Morgans lagar	142	`PasswdCaps`
Övningar till kapitel 4	144	
Kapitel 5 Filhantering	**147**	
5.1 Att skriva till och läsa från filer	148	`WriteReadFile`
5.2 Append mode	151	`AppendFile`
5.3 Slumplösenord i fil	153	`RandPasswTest`
5.4 Kryptering av filer	157	`EncryptFile`
Övningar till kapitel 5	161	
Kapitel 6 Pekare	**162**	
6.1 Vad är en pekare?	163	
6.2 Deklaration och initiering av en pekare	165	`Pointer`
6.3 Adress- och värdeoperatorn	168	`Value`
6.4 Operatorn `new`	174	`New`
6.5 Pekare och array	178	`PointArray`
- Pekararitmetik	180	`PointArithm`
6.6 Stränghantering med pekare	182	`Initials`
Övningar till kapitel 6	186	
Kapitel 7 Fördjupning i C++ programmering	**187**	
7.1 Array som parameter i funktioner	188	`RefArray`
- Referensanrop med array	190	
7.2 Sökning och sortering	192	`RandArray`
- Slumptal i en array	193	`SearchTest`
	194	`Search`
- Minimax-problemet	196	`Minimax`
- Namngivna konstanter och skalbarhet	198	`MinimaxTest`
- Bubbelsortering	199	`BubbleTest`
7.3 Templates	203	`t_out()`
- Definition av template funktioner	203	`t_BubbleTest`
- Templates = Generics	203	
- Template funktion för bubbelsortering	206	`t_sort()`
7.4 Dynamisk minnesallokering	208	
- Datorns interna minneshantering	208	
- Dynamisk array	210	`Dynamic`
- Variabel arraystorlek	211	
- Operatorerna `new[]` och `delete[]`	212	

Ämne		Sida	Program

	- Nollpekaren	212	
7.5	Dynamisk filkryptering	215	`DynEncryptFil`
7.6	2D arrays	220	`2DArray`
7.7	2D array som parameter i funktioner	224	`TableFile`
	- Tabellhantering i filer	226	`set/writeTable`
		228	`readShowTab`
		228	`updateTab`
7.8	Klasserna **`array`** och **`vector`**	231	
	- Klassen array	231	
	- Arrayens initieringslista	232	`Array_Init`
	- foreach-satsen	233	
	- Klassen vector	235	
Övningar till kapitel 7		237	
Kapitel 8	**Mer om OOP**	**238**	
8.1	Komposition	239	`Date`
	- Komposition av klasser	239	`Employ`
	- Komposition av objekt	242	`Composition`
8.2	Arv	244	`Person`
	- Arvrelationen	246	`EmployeeInh`
		247	`Inheritance`
8.3	Polymorfism	249	`Account`
	- Överskuggning av metoder	251	`MinAccount`
	- Åtkomstmodifieraren protected	252	`CreateAccount`
Övningar till kapitel 8		255	
Sex projektuppgifter		**257**	

Kapitel 1

Olika programmeringsmiljöer

	Ämne	Sida	Program
1.1	Om C/C++ och Python	8	
	- Standardisering	10	
	- Unicodes historia	10	
	- Om Python	10	
1.2	En Open Source IDE: Borland C++	12	**CppDemo**
1.3	Att bygga en egen IDE med en editor	15	
	- Att koppla Java till TextPad	16	
	- Att köra program från en editor	16	**EditDemo**
1.4	Att köra program från kommandofönstret	18	
	- Kommandotolken	18	
	- Konfiguration av Path	20	
Frågor till kap 1		22	
Övningar till kap 1		24	

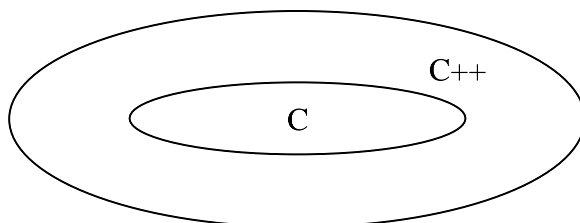
1.1 Om C/C++ och Python

70-
80-talet

C++ är en direkt utvidgning och vidareutveckling av programmeringsspråket ***C*** som **1972** utvecklades av Dennis Ritchie på Bell Laboratories med syftet att skapa ett språk för programmering av operativsystemet *Unix*. I den bemärkelsen är C en biprodukt av Unix. Därför finns många logiska paralleller mellan C och Unix. Idag är inte bara Unix utan även andra operativsystem inkl. Windows skrivna i C. Styrkan i C består av en kombination mellan enkelhet, strukturering och möjligheten att lätt kunna kommunicera med datorns hårdvara. C har bland de moderna språken den bästa förmågan att hantera och kontrollera hårdvaran, vilket favoriserar C som programspråk t.ex. för operativsystem, men även för inbyggda system. Den stora frihet som C erbjuder för hantering av bl.a. datorns primärminne med hjälp av *pekare*, kod som ger åtkomst till den fysiska adressen till data och på gott och ont tillåter manipulationer av minnesadresser.

Det var dansken Bjarne Stroustrup som la grunden till vidareutvecklingen av C. Under 70-talet hade man nämligen konstaterat att *procedural programmering* (Algol, Pascal, C, …) inte längre tillgodoság alla krav som stora komplexa program ställde med avseende på underhåll, förnyelse och ändringsbarhet. Ingen kunde sätta sig in i, ändra och vidareutveckla ett stort program om programmeraren hade lämnat företaget. Det innebar ett enormt slöseri med resurser. Dessutom utvecklades hårdvaruteknologin så snabbt att program som kunde köras på de allt mer avancerade datorerna blev allt större och mer komplexa, speciellt när det gällde grafiska tillämpningar. Mjukvaruteknologin utvecklades inte alls i samma takt. För att lösa alla dessa problem uppkom den nya programmeringsfilosofin *objektorienterad programmering* (OOP) som en vidareutveckling av den traditionella *procedurala programmeringen*.

1983 presenterade Bjarne Stroustrup programmeringsspråket C++. Han bibehöll hela C och la till de nya objektorienterade elementen, bl.a. klassbegreppet, som hade redan funnits t.ex. i *Simula*, ett norskt programmeringsspråk från 1967 som i sin tur var en direkt utbyggnad av *Algol (Algorithmic language)*. Simulas klasser hade ”glömts bort”. Den ovan beskrivna problematiken på 70-talet gjorde att man kom ihåg dem. Förhållandet mellan C och C++ illustrerar bäst den ”nya” filosofin tilläggskaraktär:



Dvs C är en *delmängd* av C++. Därför gäller all C-kod även i C++, men inte tvärtom. En C++ kompilator kan kompilera all C-kod, men inte tvärtom. Så den som lär sig C++ lär sig automatiskt C. Delmängdrelationen mellan C och C++ är unik bland högnivåspråken.

C++ är ett kraftfullt och populärt programmeringsspråk, vars styrka ligger i textbaserade konsolapplikationer. Inte att C++ vore principiellt olämpligt för grafiska tillämpningar, bara att det är lite jobbigt att skriva C++ kod för att åstadkomma grafik. En anledning är att, när C++ skapades, hade grafiska tillämpningar bara en begränsad spridning inom IT. Med utvecklingen av webbens grafiska miljö, med spridningen av Windows och grafiska användargränssnitt blev grafiken dominant. Idag har C++ fått en renässans med uppkomsten av *IoT* och *inbyggda system* p.g.a. sin snabbhet och maskinnära egenskap.

Standardisering

Vad gäller standardiseringen var frågan vad C++ egentligen är, faktiskt inte besvarad förrän den första *internationella standarden för* C++ antogs 1998. I början av 80-talet föreslog Bjarne Stroustrup de objektorienterade tilläggen till C som skapade C++. Sedan dess har olika mjukvaruföretag producerat ”sina” C++ kompilatorer. Stackars nybörjare som just hade börjat lära sig programmering. Ännu större var skadan för industrin som tvingades lägga ner ett enormt extraarbete på att justera dessa skillnader i sina program. Det nödvändiga arbetet med standardiseringen började först 1990. Under arbetets gång slog initiativtagarna ANSI och ISO ihop sina kommittéer. ANSI är det amerikanska och ISO är det internationella organet för standardisering. Resultatet blev ANSI/ISO-*standarden* som i sin tur byggde på ANSI/ISO-standarden för C som skapats 1989. Under tiden har arbetet tagits över av ISO/IEC *(International Electrotechnical Commission)*. Men trots standardisering finns ibland skillnader i hanteringen av C++ mellan olika programmeringsmiljöer, speciellt när det gäller organisationen av de stora programbiblioteken.

Bibliotek

Om man nu tittar på ett C++ program kommer man att upptäcka flera ord som inte finns i ovanstående tabell. Detta beror på att C++ dessutom har ett s.k. *bibliotek* av fördefinierade små program (funktioner och klasser) som man använder i sitt eget program för att åstadkomma vissa rutiner som t.ex. in- och utmatning. Själva språket C++ innehåller inga instruktioner för att skriva ut data till bildskärmen eller läsa in data från tangentbordet. Sådana instruktioner är kodade i biblioteket. Man kan jämföra biblioteksprogram med språkets ”litteratur”.

Dessa små program ligger som ett skal kring den inre kärnan av reserverade ord och använder förstås i sin tur bara nyckelord eller andra fördefinierade program. Om vi vill använda dem i våra egna program måste vi referera till dem med deras namn. Men dessa finns inte bland de reserverade orden.

C++ biblioteket är delvis standardiserat. T.ex. har kommittén för ANSI/ISO-standarden har lagt hela C++ biblioteket i ett s.k. *namnutrymme* för att undvika namnkonflikter med andra bibliotek, både egna och sådana som kommer från tredjehands programtillverkare. Som en utvidgning av C kan C++ även använda C:s standardbibliotek. C-standarden definierar inte bara språket utan också ett C-bibliotek som man utan vidare kan använda i sina C++ program, bara man talar om för kompilatorn namnet och platsen för C-biblioteket. Men C++ biblioteket är i de olika miljöerna för programutveckling (IDE:s) organisersde på olika sätt. Sökvägarna till bibliotekets olika delar är inte lika. Vissa biblioteksprogram saknas i vissa miljöer osv. Så det gäller att först undersöka utvecklingsmiljön, för att inte råka ut för överraskninga.

Unicodes historia

90-talet **Unicode** är inget programmeringsspråk utan en internationell teckenstandard för utvidgning av ASCII-tabellen. Unicode är av betydelse för programmeringshistorien därför att den är en milstolpe mellan textbaserade språk och grafiska tillämpningar. Det är ingen slump att övergången av det textbaserade operativsystemet DOS till det fönsterbaserade Windows faller i samma period. Det blev nödvändigt att ställa upp en teckenkodningsstandard för grafiska miljöer.

För en utförlig behandling av Unicode hänvisas till avsnitt ***2.3 Unicode*** (sid 30).

Om Python

90-talet **Python** skapades år 1989 av Guido van Rossum, en forskare på *National Research Institute for Mathematics and Computer Science* i Amsterdam.

I vissa avseenden är Python revolutionerande inom mjukvaruteknologin. Med små tekniska detaljer har man underlättat kodningen avsevärt. Här följer några egenskaper av Python:

- Språket är *interpreterande*, liknande goda gamla BASIC, vilket gör det möjligt att på ett direkt och lekfullt sätt experimentera med kod. Ingen tung programvara, typ kompilator, behövs för att tolka pythonkod, vilket gör Python lämpligt inte bara för inbyggda system, utan även för nybörjare att lära sig programmering.
- Python är ett *universellt* programmeringsspråk, dvs det kan användas för alla möjliga applikationer. All software kan kodas med Pyhon.
- Python kan enkelt och gratis installeras på alla plattformar utan att man behöver bry sig om licenser.
- Koden är nästan självbeskrivande, ligger nära pseudokod och återspeglar algoritmer på ett enkelt sätt. Därför liknar språket mycket pseudokod.

- Man har avskaffat måsvingarna **{ }** som i andra språk är obligatoriska för att avgränsa block.
- Måsvingarna har ersatts av indragningar som syntax för blockmarkering. Detta gäller inte bara i kontrollstrukturer och funktioner utan även i alla andra delar av programmet.
- De logiska indragningar som gör koden läsligare och tillhörde god programmeringsstil, har man lyft till obligatorisk syntax. På så sätt har god programmeringsstil blivit obligatorisk.
- Det är inte längre nödvändigt att avsluta en sats med semikolon.
- Variabler behöver inte deklareras. Man pratar om *dynamisk variabeldeklaration*: Datatypen tilldelas en variabel automatiskt, när denna initieras till ett värde. Värdets datatyp bestämmer variabelns datatyp.
- Löpande kod och funktioner behöver inte skrivas i klasser, även om man kan deklarera klasser. Detta har man tagit över från C++.

P.g.a. dessa fördelar och sin enkla, smidiga kodningsteknik har Python mer eller mindre konkurrerat bort många språk och kan idag anses som ett av världens mest populära programmeringsspråk, inte minst inom utbildning.

1.2 En Open Source IDE: Borland C++

Visual Studio är en tung och komplex utvecklingsmiljö vars installation och användning kräver handledning. T.ex. är det inte möjligt att skriva kod och testa den, även om koden är korrekt. Man behöver skapa ett *projekt* som en del i en *solution*, spara koden i en fil och inkludera filen i projektet. Dessutom är Visual Studio licensbelagd. Det finns enklare och billigare alternativ. Ett sådant alternativ är *Borland C++ Compiler*, en IDE som hör hemma i *Open Source* världen, en s.k. ***GNU***-produkt. ***GNU*** är en *rekursiv* akronym (förkortning) som står för ***Gnu-is-Not-Unix***. Rekursiv, därför att i förklaringen förekommer själva förkortningen som ska förklaras. ***GNU*** är en sammelbeteckning för *freeware* som utvecklas och sprids över Internet. Filosofin är att idéer och även kod inte borde ägas av någon, utan tillhör mänskligheten, på samma sätt som luften vi andas. *Borland C++* är inte lika avancerad som Visual Studio, men räcker fullt för våra ändamål. I programmen **`MessageBoxB`** (sid 26) och **`MessageBoxVS`** (sid 28) kommer vi att se exempel på skillnaderna mellan Visual Studio och *Borland C++*. För- och nackdelar kan diskuteras.

Borland var ett framgångsrikt mjukvaruföretag på 70-talet som har bytt ägare. Dess största förtjänst var lanseringen av den första integrerade programutvecklingsmiljön (IDE) *Borland Turbo-Pascal* som möjliggjorde kompilering, felsökning, editering och online hjälp i en och samma miljö och p.g.a. kompilatorns snabbhet blev en stor succé. Dessutom inkluderade *Turbo-Pascal* även grafiska applikationer.

Installation av Borland C++

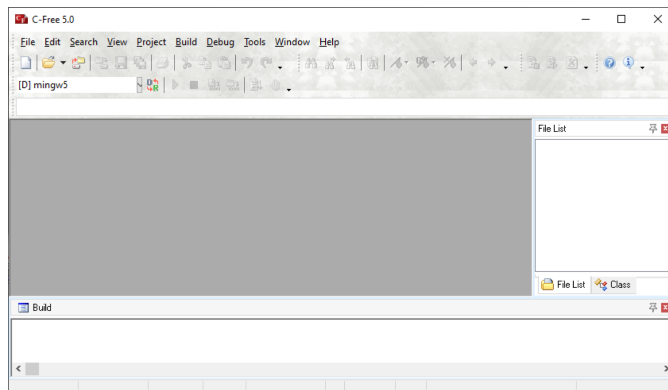
1) Starta din webbläsare och gå till:

   https://developerinsider.co/download-and-install-borland-c-compiler-on-windows-10/

2) Gå till rubriken How to install Borland C++ Compiler och klicka på knappen:

   DOWNLOAD BORLAND C++ COMPILER

3) Den zippade filen BorlandCPP.zip laddas ned. Packa upp den, dubbelklicka på filen C Free.exe. Följ instruktionerna: → Next … → Install → Finish. Du får:



Användning av Borland C++

1) Öppna en ny fil i huvudmenyn File med:

File → New

IDEn ovan delas upp på nytt: Tre fönster, från vänster (ovan): *Symbol Window, Editfönstret* och *File Tree Window* samt nedan: *Message Window.* Mata in följande kod i *Editfönstret*:

Programmet `CppDemo`

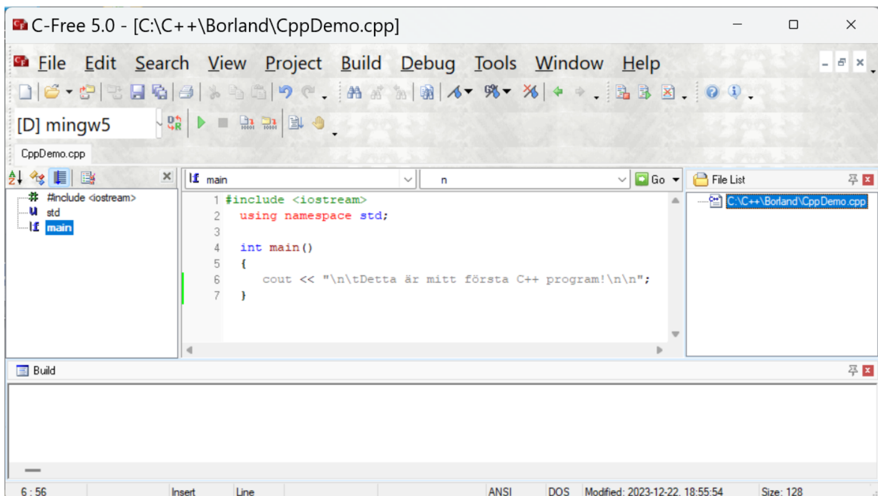
```
#include <iostream>
using namespace std;

int main()
{
    cout << "\n\tDetta är mitt första C++program!\n\n";
}
```

Vi kommer att referera till koden med programnamnet **`CppDemo`**.

2) Spara koden med filnamnet **`CppDemo.cpp`** på en plats på din dator som du själv väljer, genom att från hunvudmenyn File välja:

File → Save As…

Så här borde det nu se ut på din skärm:



3) Kompilera från huvudmenyn Build med:

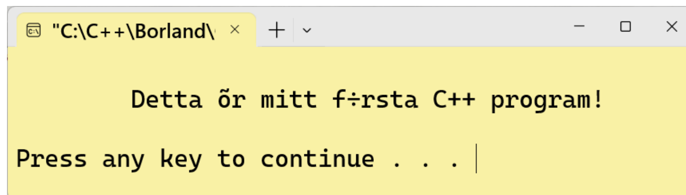
Build → Compile – CppDemo.cpp

Du kan se resultatet i *Message Window* nedan. Har du inga kompileringsfel kan du exekvera.

4) Exekvera koden från huvudmenyn Execute med:

Build → Run

Som utskrift borde du få:



Som du ser: Inget projekt behövs i

Borland's C-Free 5, a professional C/C++ IDE

Vi kommer att testa vissa av våra koder i denna miljö, för att undersöka hur och varför C++ beter sig annorlunda i olika programmeringsmiljöer.

1.3 Att bygga en egen IDE med en editor

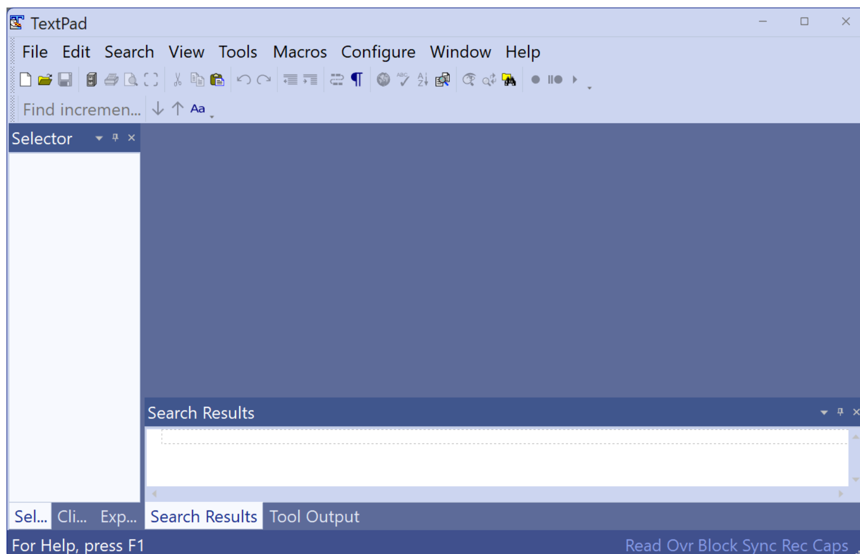
I detta avsnitt vill vi visa hur man kan använda en vanlig texteditor för att konfigurera den till en egen IDE. Detta kräver att vi integrerar i den en kompilator resp. en interpretator för ett programmeringsspråk. Vi kommer sedan att använda denna egenskapade IDE för att skriva och spara våra koder och att kompilera och exekvera dem, utan att behöva byta miljö. Som exempel har vi valt editorn TextPad och programmeringsspråket Java. Bådas programvaror kan gratis laddas ned och enkelt installeras. Sedan behövs det en viss konfiguration för att koppla ihop dem. Det kommer att visa sig att användningen är smidig och att tekniken är överförbar till andra editorer och programmeringsspråk. Det avgörande är att med editorns inställningar ange sökvägen till programmeringsspråkets kompilator resp. en interpretator, i vårt fall att i TextPads grafiska gränssnitt ange sökvägen till javainstallationen Java Development Kit (JDK). Samma sak kan man göra med många andra editorer.

Att ladda ner TextPad

- Starta din webbläsare och gå till: https://textpad.com/download
- Det kommer upp sidan *download*. Halvvägs ner på sidan (scrolla ned) hittar du tabeller och rubriker med grå bakgrundsfärg. Klicka på länken setupv9.exe.
- En exekverbar fil laddas ned. Packa upp om den är zippad.

Installation av TextPad

Dubbelklicka på den just hämtade filen setup.exe. Följ instruktionerna i InstallShield Wizard tills installationen är slutförd och du får t.ex. följande grafiskt gränssnitt:



Stäng fönstret.

Att ladda ner Java Development Kit (JDK)

- Starta din webbläsare och skriv i adressfältet:

  https://www.oracle.com/java/technologies/downloads/

- Under rubriken **Java Development Kit 21.x.x downloads** välj din plattform, t.ex. **Windows**. På raden som inleds med x64 Installer klicka på länken till höger:

  https://download.oracle.com/java/21/latest/jdk-21_windows-x64_bin.exe (sha256)

- Den exekverbara filen jdk-21 windows-x64 bin.exe laddas ned. Placera den i en mapp som reserv för en eventuell ominstallation.

Installation av JDK

Dubbelklicka på den just hämtade filen jdk-21_windows-x64_bin.exe och gå igenom följande dialogrutor:

1) Welcome to the Installation Wizard for Java … : Klicka på knappen Next.
2) Dialogrutan Destination Folder kommer upp. Låt Java installeras i mappen C:\-Program\Java\jdk-21. Klicka på knappen Next utan att ändra något. Vi kommer att referera till denna mapp som JDKs installationsmapp.
3) Klicka på Close i dialogrutan Complete.

Efter lyckad installation har i mappen C:\Program\Java kommit till en undermapp: jdk-21. Öppna den. I den finns bl.a. undermappen bin och några filer.

I princip är JDK-installationen nu klar vars viktigaste ingredienser är Java kompilatorn javac.exe och Java interpretatorn java.exe. Båda filer ligger i undermappen bin. Öppna bin och leta efter dessa två filer. Stäng bin-mappen, efter att du har lokaliserat Javas kompilator och interpretator. Det är de här två programmen som körs när vi sedan i TextPad kommer att ge kommandona Compile Java och Run Java Application. Vi måste se till att TextPad hittar dem.

Att koppla Java till TextPad

1) Öppna TextPad t.ex. från Start-knappen: Start → TextPad

   Mata in följande javaprogram i TextPads vita (stora) editfönster:

```
// EditDemo.java

import javax.swing.JOptionPane;

class EditDemo
{
  public static void main(String[] a)
  {
    JOptionPane.showMessageDialog(null,
                                  "My first Java program!");
  }
}
```

2) Gå till menyraden längst upp och spara filen via File → Save As... som EditDemo.java i en mapp som du kommer ihåg, t.ex. C:\Java.

3) Välj i menynraden:

Configure → Preferences → Tools → Add → Java SDK Commands

Klicka på knappen Verkställ och sedan OK. När vi utför de här konfigurationerna letar TexPad automatiskt efter Javas kompilator javac.exe och interpretator java.exe, hittar dem och lägger deras sökväg bakom sina Tools.

Att köra program från en editor

- Klicka nu på:

  Tools → Compile Java

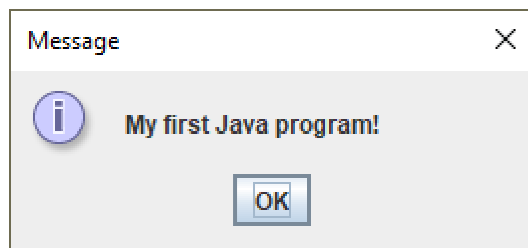
  för att kompilera koden du skrev i punkt 1). Har du skrivit in koden korrekt och sparat den i filen EditDemo.java, kommer javac.exe att kompilera koden och lägga den kompilerade class-filen i samma mapp som filen EditDemo.java ligger, hos oss: C:\Java. Kontrollera om filen EditDemo.class hamnat där du valt att spara din ful EditDemo.java. När du i TextPads Tool Output-fönster får meddelandet Tool completed successfully har du lyckats med kompileringen.

- Efter lyckad kompilering är det dags att *exekvera* programmet. Välj i menyraden:

  Tools → Run Java Application.

  Finns den kompilerade class-filen EditDemo.class i samma mapp som källkodsfilen EditDemo.java, kommer java.exe att exekvera class-filen.

  Följande Message-ruta öppnas och programmets resultat, utskriften **My first Java program!** kan beskådas där:



Klicka på OK i Javas Message-ruta och stäng konsolfönstret, för att avsluta körningen. Stäng TextPad.

1.4 Att köra program från kommandofönstret

Ett alternativ till att kompilera och exekvera program från en IDE, är att *skriva* sina program i en valfri editor och *köra* dem, inte från editorn utan från kommandofönstret. Självfallet är det enklare att göra det från en IDE: man behöver inte byta miljö mellan editering och kompilering/exekvering. Men kommandofönstrets fördel är att det är oberoende av plattformen. Man behöver inte installera en IDE och inte heller bry sig om IDEn är kompatibel med operativsystemet. Alla operativsystem har ett kommandofönster. I Windows är det Kommandotolken (cmd).

Kommandofönstret blir ännu mer aktuellt, när man har en miljö, t.ex. ett inbyggt system, där man inte har tillgång till till en editor, utan endast till koden i någon form och vill få in koden att exekveras i det inbyggda systemet. Dessutom kan det vara i sig intressant att förstå hur man kan köra från kommandofönstret och hur man kan installera och konfigurera program som fungerar på olika plattformar som endast har kommandofönstret som en given arbetsmiljö.

I detta avsnitt går vi igenom hur man kan köra program i Windows' Kommandotolk. Som exempel för kompilering/exekvering väljer vi JDK, som vi redan har installerat i förra avsnitt. Javakoden kan nu skrivas i en valfri editor som inte behöver vara TextPad. Sedan ska vi både kompilera och exekvera den, inte heller från TextPad utan från kommandofönstret.

Java källkodsfil

Gå till mappen där du hade sparat filen **`EditDemo.java`** som vi redan hade skapat tidigare, hos oss C:\Java. Öppna filen i en valfri texteditor, t.ex. Anteckningar.

Ändra koden i filen **`EditDemo.java`**, genom att lägga till den framhävda texten i utskriftssatsen:

```
"My first Java program, changed in NotePad."
```

Stäng editorn. Ta bort class-filen i mappen C:\Java. Nu har vi preparerat Java källkodsfilen, för att sedan både kompilera och exekvera den från kommandofönstret.

Kommandotolken

I Windows är det programmet cmd som kör Kommandotolken vars fil cmd.exe lagras i Windows systemmapp. I princip kan nästan allt som kan göras med musen i Windows grafiska gränssnitt även göras från den textbaserade Kommandotolken, bara att man där måste skriva kommandon i textform. En stor del av dessa kommandon är gamla DOS-kommandon, eftersom Windows är en vidareutveckling av operativsystemet *DOS (Disk Operating System)*. cmd används som synonym till Kommandotolken.

Skapa en genväg till Kommandotolken i Windows så här:

Start → Windows-systemet → Kommandotolken

Högerklicka på Kommandotolken och välj: Mer → Fäst i aktivitestfältet

Vid start kommer prompten stå i mappen C:\Users\ ... , där istället för … står namnet på en mapp, ofta kallad efter ditt användarnamn. Man kan sedan byta till en annan mapp genom att skriva kommandot:

```
cd mappnamn
```

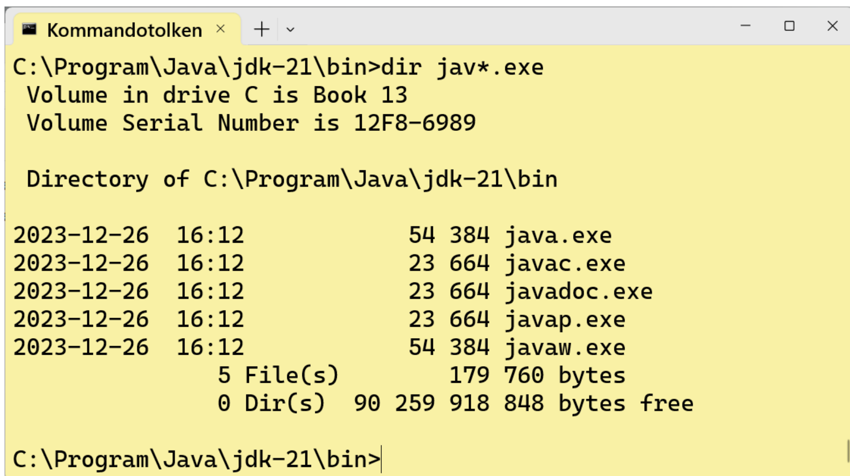
där **`cd`** står för **`c`**hange **`d`**irectory och **`mappnamn`** är namnet på den mapp som man vill byta till. Man kan även skriva en fullständig sökväg till den mapp man vill byta till. T.ex. byter följande kommando till mappen bin som ligger under JDKs installationsmapp. Skriv:

```
cd C:\Program\Java\jdk-21\bin
```

där **`C:\Program\Java\jdk-21`** är Javas *installationsmapp* när vi installerade JDK. Och mappen **`bin`** är den undermapp där bl.a. filerna javac.exe och java.exe finns som innehåller javakompilatorn och javainterpretatorn. Du kan nu efter att ha bytt till **`bin`**-mappen, kontrollera om de finns där genom att ge kommandot **`dir`** som står för **`dir`**ectory (mapp). Du får innehållsförteckningen över aktuell mapp. T.ex. kan du, genom att ge kommandot:

```
dir jav*.exe
```

lista ut alla filer i denna mapp vars namn börjar med **`jav`** och som har filändelse, **`exe`**. Bland dessa finns även filerna javac.exe och java.exe:



Minimera detta kommandofönster. Vi kommer att behöva det lite senare.

Att köra Java i Kommandotolken

Öppna i ett Windows-fönster, inte i Kommandotolken, mappen där du sparat filen **`EditDemo.java`** i, hos oss mappen C:\Java.

Öppna ett andra Windows-fönster och gå (med musen) till mappen:

C:\Program\Java\jdk-21\bin

Dvs till undermappen **bin** i Javas installationsmapp dit vi installerade JDK.

Kopiera (med musen) filen **EditDemo.java** från mappen C:\Java till mappen bin. Om du blir tillfrågad om du vill ange adminrättigheter för att kunna kopiera till denna mapp, klicka på knappen Fortsätt. Stäng båda Windows-fönstren.

Återhämta det minimerade kommandofönstret från förra sidan. Om du har stängt den, öppna cmd och upprepa kommandot:

cd C:\Program\Java\jdk-21\bin

Kontrollera med kommandot:

dir java*.*

att filerna javac.exe, java.exe och **EditDemo.java** finns där. Det borde de göra.

Egentligen ville vi fortsätta så här: Skriv kommandot **javac EditDemo.java** i Kommandotolken som tillämpar javakompilatorn javac.exe på filen **EditDemo.-java** och producerar filen **EditDemo.class**. Men en säkerhetsinställning i Windows tillåter inte sådana aktioner i bin-mappar. Vi kommer att se att man kan göra detta i andra mappar *efter* konfigurationen av Path. Gå istället vidare med att göra samma sak med **class**-filen:

Kopiera (med musen) filen **EditDemo.class** från mappen C:\Java till mappen bin. Om du blir tillfrågad om du vill ange admin-rättigheter för att kunna kopiera till denna mapp, klicka på knappen Fortsätt. Stäng båda Windows-fönstren.

Skriv sedan kommandot **java EditDemo** i Kommandotolken, för att exekvera koden. Kommandot tillämpar javainterpretatorn på class-filen och visar Meddelanderutan med utskriften **My first Java program (, written in NotePad).**

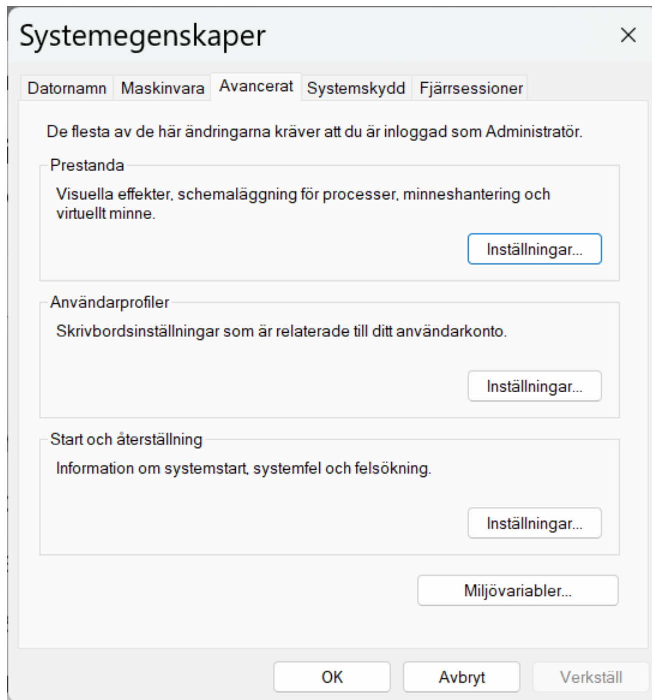
Förfarandet ovan är självklart inte tillfredsställande ur praktisk synpunkt. Man borde kunna kompilera och exekvera javafiler där de finns och inte bara från **bin**-mappen. För att slippa kopieringen av javafiler till **bin**-mappen måste variabeln Path som är en s.k. *miljövariabel* i operativsystemet, konfigureras. Dvs vi måste lägga in **bin**-mappens sökväg i systemets Path, för att cmd kan hitta den.

Konfiguration av Path

För att kunna köra javaprogram i Kommandotolken från andra mappar än bin-mappen, måste bin-mappens sökväg ingå i variabeln Path:s värde. Path är en av operativsystemets variabler vars värde är en sträng bestående av sökvägar till mappar. Vi måste lägga till bin-mappens sökväg till Path.

Så här kan man konfigurera miljövariabeln Path i Windows:

1) Högerklicka på Den här datorn och välj Egenskaper. I dialogrutan Inställningar: Gå till sökfältet Sök efter en inställning och skriv Miljövariabler. Välj Redigera systemets miljövariabler. Dialogrutan Systemegenskaper dyker upp:



2) Klicka på knappen Miljövariabler, så kommer du till dialogrutan Miljövariabler.

3) Den undre delen av dialogrutan heter Systemvariabler Markera där variabeln path och klicka på knappen Redigera… . Klicka på knappen Ny och skriv i textfältet som öppnas, sökvägen till Java installationens bin-mapp där Java kompilatorn, filen **`javac.exe`** och Java interpretatorn, filen **`java.exe`** finns: C:\Program\Java\jdk-21\bin. Lämna dialogrutan genom att klicka på OK.

4) Klicka på OK i dialogrutorna Miljövariabler och Systemegenskaper.

Kontrollera med kommandot path i Kommandotolken om sökvägen ovan lagts till.

Om ja, kan du nu från mappen som innehåller filen EditDemo.java, med kommandot **`javac EditDemo.java`** kompilera och med **`java EditDemo`** exekvera.

Frågor till kapitel 1

Besvara följande frågor om C/C++, Unicode & Pyton (sid 8-10):

1.1 Vad är den traditionella, procedurala synen på programmering som rådde på 60- och 70-talet?

1.2 Vad är den objektorienterade synen på programmering som kom upp på 80-talet?

1.3 När skedde utvidgningen från C till C++ och vem lade grunden till denna utveckling?

1.4 Vad är den viktigaste skillnaden mellan C och C++?

1.5 Varför finns logiska paralleller mellan C/C++ och Unix?

1.6 Vad var anledningen till att man på 80-talet bytte paradigm inom programmering?

1.7 Nämn två operativsystem som är programmerade i C.

1.8 Vad innebär begreppet *pekare* och vilken relevans har det för programutveckling?

1.9 Är *pekare* ett koncept som finns i C eller har man lagt till det senare med C++?

1.10 Vad innebär det att C är en delmängd av C++?

1.11 Vad betyder tillägget ++ vid vidareutvecklingen från C till C++?

1.12 Vad är den historiska orsaken för att C/C++ inte är optimalt för grafiska tillämpningar?

1.13 För vilka teknologier är C/C++ optimalt och vilka av språkens egenskaper är anledning till det?

1.14 Är *Unicode* ett programmeringsspråk? Om ja, är det interpreterande eller kompilerande? Om nej, vad är Unicode då?

1.15 Vilket behov ledde till uppkomsten av Unicode?

1.16 Vad är föregångaren till Unicode?

1.17 Vilken roll spelade tillämpningen av grafik inom IT för uppkomsten av Unicode?

1.18 Är Unicode en delmängd av ASCII eller omvänt?

1.19 Är *Python* ett kompilerande eller interpreterande språk?

1.20 Nämn *en* fördel av interpreterande språk.

1.21 Vad har man ersatt måsvingarna **{ }** med i Python?

1.22 Vilken teknik använder Python för deklaration av variabler?

1.23 Inom vilket område är Python mest populärt?

1.24 Är Python ett *universellt* programmeringsspråk eller ett skriptspråk?

1.25 I vilka avseenden är Python revolutionerande inom mjukvaruteknologin?

Övningar till kapitel 1

1.1 Installera *Borland C++ Compiler* på din dator enligt instruktionerna i avsnitt 2.3 (sid 12). Mata in programmet **`CppDemo`** nedan i denna miljö:

```
// CppDemo.cpp
#include <iostream>
using namespace std;

int main()
{
   cout << "\n\tDetta är mitt första C++ program!\n";
}
```

Kompilera och exekvera.

Fortsätt med att kompilera och exekvera följande program i *Borland C++*:

```
// MySecond.cpp
#include <iostream>
using namespace std;

int main()
{
   cout << "\n\t"
        << "Detta är mitt andra C++ program!\n"
        << "\tDet skriver ut två rader text.\n";
}
```

1.2 Utveckla och testa följande program både i Borland och i Visual Studio. Kan du konstatera nårga skillnader mellan dessa miljöer?

Skriv ett program som läser in två heltal, multiplicerar dem med varandra och skriver ut resultatet blandat med förklarande text. Om du t.ex. matar in **`3`** till det första och **`4`** till det andra heltalet, ska programmet skriva ut: **`3 gånger 4 är 12.`** Vidareutveckla programmet med ytterligare räkneoperationer, kanske så småningom till en liten kalkylator.

1.3 I början av detta kapitel sa vi: ”Det finns enklare och billigare alternativ” (sid 12), och då menade vi ... än Visual Studio, för att utveckla och testa C++ program. Vi presenterade *Borland C++ Compiler*. Ytterligare ett sådant alternativ är IDEn *CLion*. Ladda ned och installera *CLion*. Testa programmen **`MessageBoxB`** (sid 26) och **`MessageBoxVS`** (sid 28) i denna IDE, för att studera skillnaderna med de befintliga miljöerna. Vad är för- och nackdelarna?

Kapitel 2

Fortsättning med C++

	Ämne	Sida	Program
2.1	Utskrift till en grafisk miljö i C++	26	**`MessageBoxB`**
2.2	Olika beteenden i olika miljöer	28	**`MessageBoxVS`**
	- Vad är LPCWSTR?	29	
	- Parentes om Java	29	
2.12	Unicode	30	**`Unicode.java`**
	- Att arbeta med Unicode	32	**`Char2Int.java`**
2.4	Svenska tecken i olika miljöer	35	**`MessageBoxSw`**
2.5	Automatisk typkonvertering	43	**`AssignRule`**
	- Tilldelningsregeln	43	**`Overflow`**
	- **`int`**-regeln	46	**`IntRule`**
	- Befordringsregeln	47	**`PromotInt/Dec`**
2.6	Rekursion	50	**`Fibonacci`**
2.7	Mer om flervägsval	54	
	- Luriga **`else`**	54	**`TrickyElse`**
	- Korrekt **`else`**	56	**`CorrectElse`**
	- **`switch`** med tomma **`case`**-satser	57	**`SwitchInequ`**
2.8	Misslyckad modularisering	59	**`MiniSort`**
	- Försök att modularisera **`MiniSort`**	60	**`NoSort`**
2.9	Referenser	62	**`Reference`**
	- Vad är en referens?	62	
	- Två olika betydelser av ampersand (&)	63	
	- Referens vs. pekare	64	**`PointRef`**
2.10	Parameteröverföringsmetoder	66	
	- Värdeanrop (Call by value)	66	**`CallByValue`**
	- Referensanrop (Call by reference)	68	**`CallByRef`**
	- Två olika minnesbilder	71	**`Swapping`**
2.11	In- och utparametrar	72	**`change()`**
2.12	Överlagring av funktioner	75	**`power()`**
		77	**`Overload`**
Övningar till kapitel 2		78	

2.1 Utskrift till en grafisk miljö

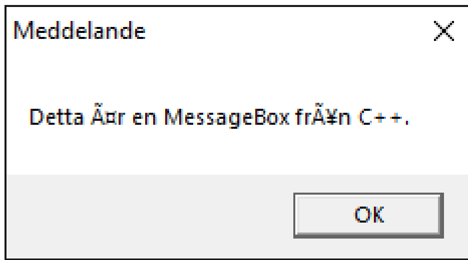
Alla våra program hittills har skrivit ut sina körresultat till konsolen som är en ren textmiljö uppbyggd av rutor där varje ruta rymmer ett tecken, antingen bokstav, siffra eller specialtecken. Till skillnad från en textmiljö är en grafisk miljö uppbyggd av pixlar, där en pixel är bildskärmens minsta ljuspunkt. I en grafisk miljö ”ritas” t.o.m. bokstäver och siffror med pixlar.

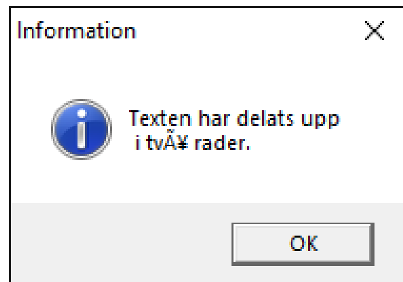
Här ska vi försöka oss på att med så enkla medel som möjligt skriva ut till en grafisk miljö som de olika C++ utvecklingsmiljöerna ställer oss till förfogande. Verktyget som gör detta möjligt är funktionen **`MessageBox()`** ur biblioteket **`windows.h`**. Den genererar en meddelanderuta, en grafisk miljö som till skillnad från konsolen endast tillåter output.

Det är anmärkningsvärt att den grafiska miljön **`MessageBox()`** kan användas i en *Console Application* som ursprungligen är konciperad för textbaserade applikationer. Läs mer om funktionen **`MessageBox()`** på nästa sida.

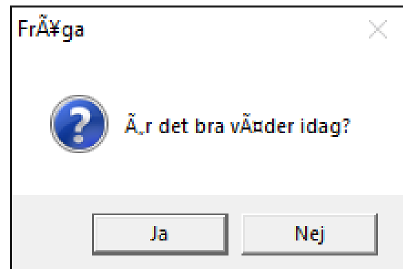
```
// MessageBoxB.cpp     // Detta program kan inte kompileras i
                       // Visual Studio. Kör det i Borland C++
// Skriver ut text till en MessageBox (meddelanderuta)
// Anrop av funktionen MessageBox() som är definierad i
// biblioteket windows.h och har fyra parametrar

#include <windows.h>  // Krävs för funktionen MessageBox()

int main()
{
   MessageBox(NULL,
              "Detta är en MessageBox från C++.",
              "Meddelande",
              NULL);
   MessageBox(NULL,
              "Texten har delats upp \n i två rader.",
              "Information",
              MB_ICONINFORMATION);
   MessageBox(NULL,
              "Är det bra väder idag?",
              "Fråga",
              MB_YESNO | MB_ICONQUESTION);
}
```

Exekveringen av denna kod genererar meddelanderutor som är avbildade på nästa sida. Tittar man noga på programmets fullständiga körresultat kan man även upptäcka det tomma konsolfönstret i bakgrunden som kommer upp först. Detta beror på att programmet **`MessageBoxB`** är en *Console Application*, inte en *Windows Forms Application*. Sedan visas följande meddelanderutor, en i taget. Klickar man på OK-knappen dyker upp nästa:





Slutligen visas rutan med rubriken Fråga. Efter klick på Ja eller Nej återgår programkontrollen till konsolfönstret.



Konsolapplikationen **`MessageBoxB`** visar att man med funktionen **`MessageBox()`** kan skriva ut till en grafisk miljö, närmare bestämt till en meddelanderuta. Anropet av **`MessageBox()`** i sin tur kräver inkluderingen av biblioteket **`windows.h`**.

Funktionen `MessageBox()`

Som körresultatet visar finns det lite olika utseenden på meddelanderutorna som är förstås relaterade till koden på förra sidan. Där hittar vi lite olika varaianter av funktionen **`MessageBox()`**. Tre olika anrop genererar tre olika meddelanderutor. Det är parametrarna som avgör vilken variant man anropar:

1) Första paramatern **`NULL`** bestämmer rutans placering. På denna plats kan man ange i vilken behållare (container) rutan ska placeras. **`NULL`** innebär default-containern som är bildskärmen. Därför placeras rutorna centrerade på bildskärmen.
2) I den andra parametern kan man skriva en sträng som sedan dyker upp som text i rutans innehåll.
3) I den tredje parametern kan man skicka en text till rutans rubrik.
4) Den fjärde parametern specificerar både ikonen i rutan och vilken typ av knappar rutan ska visa. De olika informationstyperna – ikon och knappform – skiljs åt med `|` (eng.: *vertical bar*) – allt inom den fjärde parametern.

Det finns ännu fler varianter av funktionen **`MessageBox()`** än de som visas här. Med F1-tangenten, när markören står på **`MessageBox()`**, får man reda på dem.

<u>Två problem kvarstår:</u>

1. Varför kan **`MessageBoxB`** inte kompileras i Visual Studio och vad är lösningen?

2. Hur kan man få korrekta svenska tecken ä, å, ö, Ä, Å, Ö i utskrifterna?

2.2 Olika beteenden i olika miljöer

Här ska vi besvara fråga **1** som ställdes i slutet av förra avsnitt. Koden i programmet **MessageBoxB** (sid 26) genererar i Visual Studio kompileringsfel. Undersöker man felet närmare hittar man följande beskrivning:

Argument of type "const char *" is incompatible with parameter of type "LPCWSTR"

Och vid lokaliseringen av felet pekar Visual Studio på alla parametrar av funktionen **MessageBox()** som är konstanta strängar omgärdade av citationstecken **" "** , dvs av typ **string**. Närmare bestämt är det den andra och den tredje parametern i funktionen **MessageBox()** som anropas tre gånger i programmet **MessageBoxB**.

Ett sådant fel dyker inte upp i *Borland C++ Compiler* (sid 12). Där kan **MessageBoxB** både kompileras och exekveras. I programmet nedan presenteras lösningen utgående från den information vi fick från Visual Studios felmeddelande ovan. Vi ska sedan ägna oss åt att förklara denna lösning.

```
// MessageBoxVS.cpp
// Gör samma sak som programmet MessageBoxB (sid 26)
// Kan kompileras & exekveras i Visual Studio, inte i Borland

#include <windows.h>     // Krävs för funktionen MessageBox()

int main()
{
   MessageBox(NULL,
        (LPCWSTR) L"Detta är en MessageBox från C++.",
        (LPCWSTR) L"Meddelande",
        NULL);
   MessageBox(NULL,
        (LPCWSTR) L"Texten har delats upp \n i två rader.",
        (LPCWSTR) L"Information",
        MB_ICONINFORMATION);
   MessageBox(NULL,
        (LPCWSTR) L"Är det bra väder idag?",
        (LPCWSTR) L"Fråga",
        MB_YESNO | MB_ICONQUESTION);
}
```

OBS! I koden ovan måste det stora **L** skrivas *utan mellanslag* framför strängkonstanterna. Skillnaden till programmet **MessageBoxB** är de vit framhävda koderna. Koden **(LPCWSTR)** tyder på att det handlar om explicit typkonvertering och att **LPCWSTR** är en datatyp (bortsett från **L** – vi återkommer till det). Om det är så, konverteras de konstanta strängarna till datatypen **LPCWSTR**. En annan information i felmeddelandet som citerades ovan, är att strängarnas datatyp är const char * , dvs konstanta *pekare* till **char**. Om *pekare* läs kap 6, sid 162. Tydligen kan Visual Studio inte automatiskt konvertera datatypen const char * till **LPCWSTR**. Det är nödvändigt att vi gör det explicit – och det har vi gjort i **MessageBoxVS**. Förklaringen är att

MessageBox() är en biblioteksfunktion och att C++ standarden inte gäller för alla bibliotekskoder. Det är ett känt fenomen, men vi hade inte konkret stött på det hittills. För att förstå detta lite bättre ska vi bekanta oss med datatypen **LPCWSTR**.

Vad är LPCWSTR ?

Denna akronym (förkortning) står för *__L__ong __P__ointer to __C__onstant __W__ide __STR__ing* och är beteckningen för en datatyp i C/C++ som representerar pekare till strängar, men inte som **const char *** utan i *Long*-format. Vad betyder det?

Som vi vet representeras strängar i C/C++ med pekare som pekar på adressen till strängens första tecken. Och strängar består i C/C++ av ett antal tecken som avslutas med nolltecknet. Alla dessa tecken är av typ **char**. Men vi vet också att för datatypen **char** allokeras i C/C++ **1** byte dvs **8** bitar. Standard ASCII använder sig av s.k. 7-bitars kodning, vilket innebär att koden till ett tecken placeras som nollor och ettor i 7 bitar av en byte. Den lediga åttonde biten används för felkontroll. Detta innebär den störst möjliga kod man kan uttrycka med 7 bitar är 7 ettor 1111111 som är 127 decimalt. Därför består ASCII-koderna av heltalen mellan 0 och 127. Alla tecken vars koder är större än 127, bland dem de svenska tecknen ä, å, ö, Ä, Å, Ö behöver mer minnesutrymme. En annan teckenstandard som allokerar **2** bytes dvs **16** bitar för att lagra ett tecken, måste användas. En sådan standard är Unicode som behandlas i nästa avsnitt.

I C/C++ kan man dock inte komma ifrån att datatypen **char** har endast **1** byte minne till förfogande. Lösningen för att lagra även koder som överskrider ASCII-gränsen 127, är typkonvertering till ett *Long*-format, dvs en fördubbling av minnesutrymmet. Detta görs för det första med det stora **L** som står för **Long** och skrivs utan mellanslag intill strängkonstanterna. **L** ger **2** bytes till tecknen. Och för det andra omvandlas med explicit typkonvertering till datatypen **LPCWSTR** som skapar en ny typ av pekare som pekar på det första tecknet av strängar vars tecken är i *Long*-format.

Biblioteksfunktionen **MessageBox()** som är förprogrammerad i biblioteket i **windows.h** genererar en meddelanderuta som till skillnad från konsolen är en grafisk miljö. I grafiska miljöer är i regel Unicode den dominerande teckenstandarden. Därför kräver funktionen **MessageBox()** att dess konstanta strängparametrar är pekare till strängar i *Long*-format, dvs av datatypen **LPCWSTR**.

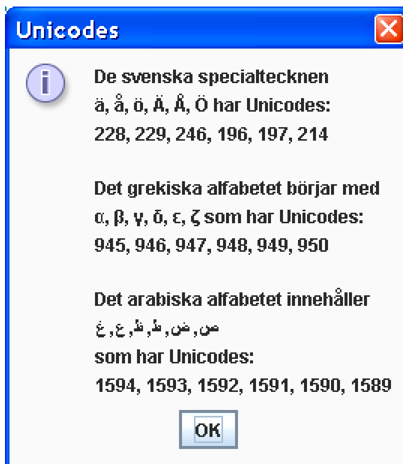
Parentes om Java

Här lägger vi in en parentes om Java som avlutas på sid 33, för att skaffa oss kunskap om Unicode och därmed kunna besvara frågan som ställdes i slutet av förra avsnitt ***3.1 Utskrift till en grafisk miljö*** (sid 27):

2. Hur kan man få korrekta svenska tecken ä, å, ö, Ä, Å, Ö i utskrifterna?

Och detta speciellt i grafiska miljöer. Varför vi använder Java besvaras på sid 33.

2.3 Unicode



Unicode är en internationell teckenkodningsstandard vars första version introducerades 1991 av *The Unicode Consortium*, en non-profit organisation för utveckling av standarder inom IT. Unicode kompletterar ASCII-tabellen och avänds bl.a. i Windows och i många andra grafiska miljöer. Till skillnad från ASCII som lagrar ett tecken i 1 byte (8 bitar) använder sig Unicode av 2 bytes (16 bitar). Därmed kan man koda ett väsentligt större antal tecken, även sådana från andra språk som arabiska, japanska, kinesiska, hebreiska, kyrilliska osv. Unicode har plats för över en miljon tecken varav drygt 100 000 tecken är tilldelade i den senaste versionen. ASCII-standarden är inkluderad, dvs ASCII är en delmängd av Unicode, så att alla ASCII-koder upp till `127` är identiska med Unicode, bara att även dessa tar 2 bytes i Unicode. Utöver koden `127` har man infört nya koder som man kan hitta i ***Unicode-tabellen***. Några av dem ser man i rutan ovan: bokstäver från tre olika språk: svenska, arabiska och grekiska som tagits fram med sina resp. Unicode-koder. Rutan ovan är utskriften av javaprogrammet `Unicode` som tas upp på sid 32.

Unicodes historia

90-talet **Unicode** är dom en internationell teckenstandard för utvidgning av ASCII-tabellen av betydelse för programmeringshistorien därför att den är en milstolpe mellan textbaserade språk och grafiska tillämpningar. Det är ingen slump att övergången av det textbaserade operativsystemet DOS till det fönsterbaserade Windows faller i samma period. Det blev nödvändigt att ställa upp en teckenkodningsstandard för grafiska miljöer.

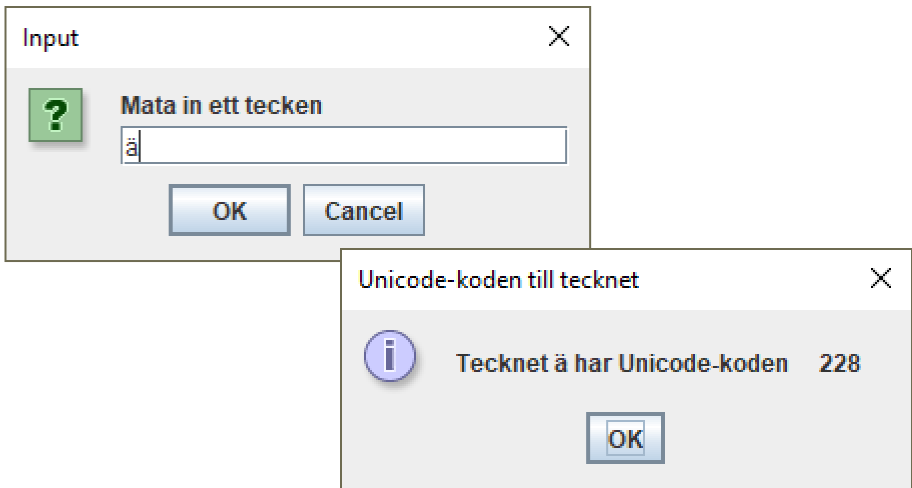
Att få tag i Unicodes

För att kunna använda koderna i progr. på sid 32 kör vi först följande program:

```
// Char2int.java                        // Detta är ett javaprogram.
import javax.swing.JOptionPane;          // Krävs för dialogrutorna
                                         // showInputDialog() och
class Char2int                           // showMessageDialog()
{
  public static void main(String[] a)
  {
```

```
    String str = JOptionPane.showInputDialog(
                                    "Mata in ett tecken");
    char tecken = str.charAt(0); // Tar ut 1:a tecknet ur str
                                 // som returneras som char
    JOptionPane.showMessageDialog(null,
    "Tecknet " + tecken + " har Unicode-koden     " +
    (int) tecken + "     ", "Unicode-koden till tecknet", 1);
  }
}
```

Med programmet **`Char2int`** kan man ta reda på Unicode-koden till vilket tecken som helst, t.ex.:



Hur däremot typomvandlingen från den inlästa strängen till *tecken* genomförs ska vi titta närmare på.

Metoden `charAt()`

Redan sättet att anropa denna metod i satsen

```
tecken = str.charAt(0);
```

visar att den är definierad i klassen **`String`** därför att före punkten står en variabel av typ **`String`** som samtidigt är en klass. I Javas API-dokumentation under klassen **`String`** kan man hitta att den bl.a. har en metod **`charAt()`** som returnerar ett tecken ur den sträng som man anropar den med, dvs den sträng som står före punkten vid anropet. *Vilket* tecken som ska returneras, kan man tala om genom att skicka det önskade tecknets position i strängen som parameter till metoden. Denna position kallas *index* – en slags numrering – som börjar med **`0`**. Index **`0`** innebär strängens första tecken, index **`1`** strängens andra tecken osv. I vårt fall består strängen **`str`** endast av **`A`**. Därför returnerar parametern **`0`** som vi skickar vid anropet

ovan, *tecknet* **A**. Så långt är det OK, vi har lyckats att läsa in **A** och lagra det som tecken, nämligen i variabeln **tecken** av typ **char**. Att vi måste gå omvägen genom en sträng beror på att vi vid inläsning använder den fördefinierade metoden **show-InputDialog()** som returnerar all inmatning som en sträng.

Programmet **Char2int**:s mål var ju att till sist få ut tecknets Unicode. Detta sista steg kan tas med explicit typkonvertering. Det är generellt så att typomvandlingar inom enkla datatyper kan genomföras med explicit typkonvertering på ett mycket enklare sätt än omvandlingar från eller till klasser som **String**. För dem behövs i regel vissa metoder som är definierade i klasser som t.ex. metoden **charAt()** i klassen **String**.

Att arbeta med Unicode

Vi återvänder till rutan **Unicodes** på sid 30 och vill titta på och förstå javakoden som genererar denna utskrift.

Unicode anges som escapesekvensen **\u** följt av ett fyrasiffrigt hexadecimalt tal, dvs så här: **\uAAAA** där **u** står för unicode och **A** för någon hexadecimal siffra. Formatet är föreskrivet vilket bl.a. innebär att ett hexadecimalt tal med mindre än fyra siffror måste inledas med nollor. Alla tecken oavsett skriv- och läsbarhet kan i C++ kod skrivas i detta format. I programmet på nästa sida initieras **char**-variabeln **alpha** till tecknet **\u03B1** som är angivet i Unicode-format där **3B1** är den hexadecimala motsvarigheten till det decimala talet **945** som enligt Unicode-tabellen är teckenkoden till den grekiska bokstaven α vilket man kan övertyga sig om med en blick på Internetlänk. Vi har ingen chans att få in detta tecken i datorn. Det finns inte någon sådan tangent på ett svenskt eller amerikanskt tangentbord. Så, den enda möjligheten att ta fram det är att använda escapesekvensen med Unicode. Apostroferna kring **\u03B1** när det tilldelas **char**-variabeln **alpha** visar att koden **\u03B1** symboliserar endast *ett* tecken – nämligen bokstaven α – och behandlas som vilket tecken som helst. På den nämnda Internet-länken kan man även se att det grekiska alfabetet förekommer som ett sammanhängande block i Unicode-tabellen. Därför kan vi använda teckenaritmetik för att få fram de bokstäver som följer efter α. Samma sak görs i programmet med de arabiska bokstäverna genom att initiera **char**-variabeln **arab** till tecknet **\u0635** där **635** är den hexadecimala motsvarigheten till det decimala talet **1589**, koden till den arabiska bokstaven ص osv. Anmärkningsvärt är också att programmet automatiskt går över till att skriva från höger till vänster när Unicodes till de arabiska bokstäverna skickas till utskrift. För att få ut de svenska specialtecknen behöver vi inte hämta deras koder från Unicode-tabellen, omvandla dem till hexadecimala och skicka dem som escapesekvenser, därför att vi direkt kan skriva in dem från våra svenska tangentbord.

```
// Unicode.java
                                  // Detta är ett javaprogram.
                                  // Kör det i TextPad
Import javax.swing.JOptionPane;   // Krävs för dialogrutan
                                  // showMessageDialog()
```

```
class Unicode
{
  public static void main(String[] a)
  {
    char alpha = '\u03B1', arab = '\u0635';

    JOptionPane.showMessageDialog (null,
      "De svenska specialtecknen\n\u00E4, \u00E5, \u00F6, " +
      "\u00C4, \u00C5, \u00D6 har Unicodes:\n"             +
      (int) '\u00E4' + ", " + (int) '\u00E5'       + ", "  +


      (int) '\u00F6' + ", " + (int) '\u00C4'       + ", "  +
      (int) '\u00C5' + ", " + (int) '\u00D6'       + '\n'  +
      "\nDet grekiska alfabetet b\u00F6rjar med\n"  + alpha +
      ", " + (char)(alpha+1) +", "+ (char)(alpha+2) + ", "  +
      (char)(alpha+3) + ", " + (char)(alpha+4)      + ", "  +
      (char)(alpha+5) + " som har Unicodes:\n"             +
      (alpha+0) + ", " + (alpha+1) + ", "                  +
      (alpha+2) + ", " + (alpha+3) + ", "                  +
      (alpha+4) + ", " + (alpha+5) + '\n'                  +

      "\nDet arabiska alfabetet inneh\u00E5ller\n"   + arab +
      ", "  + (char)(arab+1) + ", " + (char)(arab+2) + ", " +
      (char)(arab+3) + ", " + (char)(arab+4)         + ", " +
      (char)(arab+5) + "\nsom har Unicodes:\n"             +
      (arab+5) + ", " + (arab+4) + ", "                    +
      (arab+3) + ", " + (arab+2) + ", "                    +
      (arab+1) + ", " + (arab+0), "Unicodes", 1);
   }
}
```

Javaprogrammet ovan använder sig av `(char)(alpha+1)` för att skriva ut β när `alpha` har värdet α fast `alpha` redan är av typ `char`. Anledning till den explicita typkonverteringen till `char` är att additionen gör om parentesens resultat till en `int` så att en explicit omvandling tillbaka till `char` blir nödvändigt. Omvänt när vi behöver Unicode-koden utnyttjar vi just den automatiska typkonverteringen t.ex. i `(alpha+0)` som förorsakas av additionen, för att få fram koden `945` utan explicit typkonvertering. Samma teknik används för `(alpha+1)` osv. för att få koderna till β, γ osv. samt för behandlingen av variabeln `arab`.

Varför Java?

Nu kan vi precisera svaret på frågan, varför vi öppnade parentesen om Java (sid 29) som avslutas här. Då sade vi: Utan kunskap om Unicode kan vi inte besvara frågan som ställdes i slutet av avsnitt ***3.1 Utskrift till en grafisk miljö*** (sid 27):

2. Hur kan man få korrekta svenska tecken ä, å, ö, Ä, Å, Ö i utskrifterna?

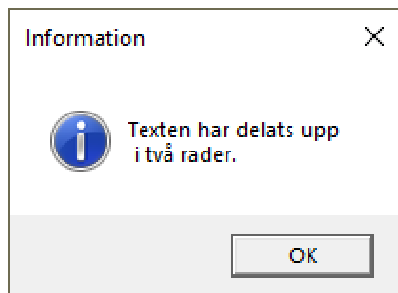
Det slutliga svaret behandlas i nästa avsnitt. Och vi hade inte kunnat exemplifiera Unicode i C++ med konkreta program eftersom C++ som har sina rötter i 70-talets C,

inte har Unicode som standard. På 70-talet fanns inte Unicode än. Java däremot som kommit upp senare, har inte bara Unicode som standard utan framför allt verktyg för att på ett enkelt sätt kunna hantera grafiska miljöer, där endast Unicode gäller. I de två javaprogrammen i detta avsnitt har vi använt oss av ett av dessa verktyg, nämligen **`Swing`**-biblioteket som tillhandahåller kod för de grafiska dialogrutor som visades på sid 30/31, speciellt inmatningsrutan **`showInputDialog()`**. Annars hade vi inte kunnat få tag i de svenska tecknens Unicodes.

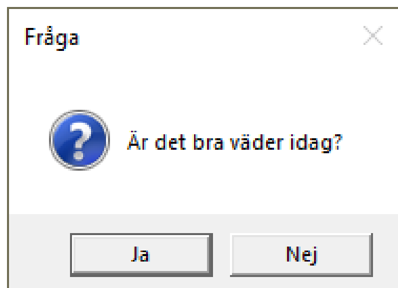
2.4 Svenska tecken i olika miljöer

Vi återvänder nu till C++ och ska besvara fråga **2** ovan, nämligen hur kan man få korrekta svenska tecken ä, å, ö, Ä, Å, Ö i utskriftsrutorna. Detta är möjligt nu eftersom vi i förra avsnitt har fått reda på Unicode-koderna till de svenska tecknen ovan. Och vi vet också att dessa koder fungerar i grafiska miljöer, vilket bekräftas av programmet nedan:

```
// MessageBoxSw.cpp
// Gör samma sak som programmet MessageBoxVS med skillnaden
// att det skriver ut korrekta svenska tecken i MessageBoxen
// Kan kompileras & exekveras i Visual Studio, inte i Borland
#include <windows.h>     // Krävs för funktionen MessageBox()
int main()
{
  MessageBox(NULL,
      (LPCWSTR) L"Detta \u00E4r en MessageBox fr\u00E5n C++.",
      (LPCWSTR) L"Meddelande",
      NULL);
  MessageBox(NULL,
      (LPCWSTR) L"Texten har delats upp \n i tv\u00E5 rader.",
      (LPCWSTR) L"Information",
      MB_ICONINFORMATION);
  MessageBox(NULL,
      (LPCWSTR) L"\u00C4r det bra v\u00E4der idag?",
      (LPCWSTR) L"Fr\u00E5ga",
      MB_YESNO | MB_ICONQUESTION);
}
```



Körresultatet visar korrekta svenska tecken i den grafiska miljön som genereras av funktionen **`MessageBox()`**. Även här måste i koden det stora **`L`** skrivas *utan mellanslag* framför strängkonstanterna. Skillnaden till programmet **`MessageBoxVS`** (sid 28) är de vit framhävda koder, dvs Unicode-koderna till de svenska tecknen som var förvrängda i förra versionen.



För att åtgärda förvrängningen av de svenska tecknen tittar vi på följande tabell som sammanställer:

De svenska tecknens Unicode-koder

Tecknet	Unicode	Hex. kod	Escapesekvens
ä	**228**	**E4**	**\u00E4**
å	**229**	**E5**	**\u00E5**
ö	**246**	**F6**	**\u00F6**
Ä	**196**	**C4**	**\u00C4**
Å	**197**	**C5**	**\u00C5**
Ö	**214**	**D6**	**\u00D6**

Unicode hittar man i Unicode-tabellen på Internet eller genom att köra javaprogrammet **`Char2int`** (sid 30). Hex. kod är den *hexadecimala* framställningen (med basen 16) av de decimala Unicode-koderna. Omvandlingen kan man få med hjälp av en kalkylator. Escapesekvens är den hexadecimala Unicode-koden skriven som escapesekvens, dvs inledd med backslashtecknet **** , följt av ett **u** som står för **u**nicode, och Hex.-koden i fyra siffror med inledande nollor, i fall att denna har mindre än fyra siffror. Escapesekvensen kan i koden antingen infogas i strängkonstanter som är omgärdade av citationstecken eller anges inom apostrofer som enskilda tecken av typ **`char`**.

Unicode-koderna ovan skiljer sig från de ASCII-utvidgningar i textbaserade miljöer som t.ex. konsolen, ja t.om. i samma språk. Medan Unicode-koderna är standard i alla programmeringsspråk och i regel gäller i alla grafiska miljöer, bl.a. i Windows, finns det i regel olika ASCII-utvidgningar i de olika språkens konsoler och andra miljöer. T.ex. gäller i Javas konsolmiljö *inte* Unicode-koderna ovan. Även i C/C++ har ASCII-utvidgningen i konsolen andra koder än i grafiska miljöer. Samma gäller för andra språk. Därför måste man alltid undersöka vad som gäller i den miljö man valt eller är tvungen att arbeta i. Nedan följer en sådan undersökning för C/C++.

De svenska tecknens koder i C++ konsolen

Till att börja med sammanställer vi samma tabell som ovan för de svenska tecknen för deras koder som gäller i textbaserade miljöer som C++ konsolen:

Tecknet	C++ konsolkod	Hex. kod	Escapesekvens
ä	**132**	**84**	**\x84**
å	**134**	**86**	**\x86**
ö	**148**	**94**	**\x94**
Ä	**142**	**8E**	**\x8E**
Å	**143**	**8F**	**\x8F**
Ö	**153**	**99**	**\x99**

Hur får man tag i dessa koder? På *liknande* sätt som vi fick tag i Unicode-koderna för den grafiska miljön, nämligen genom att köra javaprogrammet **`Char2Int.java`** (sid 30). Fast då använde vi oss av Java eftersom vi hade möjligheten att mata och få utskrift i *grafiska* dialogrutor. Men nu gäller det att mata in och få utskrift i en

textbaserad konsol. Därför använder vi oss av C++ programmet **`Char2int`** som vi lärde känna tidigare. Här en påminnelse:

```
// Char2int.cpp
// Ger koden till ett inmatat tecken i C++ konsolen
#include <iostream>
using namespace std;

int main()
{
   unsigned char letter;
   cout << "\n\tMata in ett tecken och tryck på Enter :    ";
   cin >> letter;
   cout << "\n\tDet inmatade tecknet är " << letter <<
           " och har koden    " << (int) letter << '\n';
}
```

Testa gärna själv koden. Du kommer att få bekräftelse för de koder som är angivna i tabellen på förra sidan. Istället vill vi gå vidare och få reda på konsolkoderna även för andra tecken än just de i tabellen ovan.

Konsolens hela teckenuppsättning i C++

Programmet **`AsciiFor`** skriver ut hela ASCII-tabellen genom att låta användaren ange **`start`** och **`end`** på ett kodintervall.

```
// AsciiFor.cpp
// Skriver ut ASCII-tabellen med for-satsen
#include <iostream>
using namespace std;

int main()
{
   int start, end;
   cout << "\n\tAnge början                         (t.ex. 33): ";
   cin >> start;
   cout <<"\toch slutet på ett ASCII-intervall(t.ex. 255): ";
   cin >> end;
   cout << "\n\t";
   for (int code = start; code <= end; code++)
   {
      cout << (char) code << " = " << code << "     ";
      if (code % 6 == 0)
         cout << "\n\t";  // Radbyte & tab var 6:e utskrift
   }
   cout << "\n";
}
```

`if`-satsens roll är att producera radbyte samt tabulator när **`code`**:s värde är jämnt delbart med **`6`** dvs var sjätte utskrift. Vi har gjort så för att få en tabellartad utskrift och för att undika en dubbel- eller nästlad utskrift. Självklart kan även en **`while`**-

loop lika bra åstadkomma samma resultat. Ett körexempel med programmet **AsciiFor** ger följande utskrift som visar alla läs- och skrivbara tecken i ASCII-tabellen, både standard ASCII (upp till **127**) och den utvidgade teckenuppsättningen som C++ använder i textbaserade miljöer, de icke-standardkoderna mellan **128** och **255**:

```
  Ange början                             (t.ex. 33):  33
  och slutet på ett ASCII-intervall (t.ex. 255):  255

! = 33     " = 34     # = 35     $ = 36
% = 37     & = 38     ' = 39     ( = 40     ) = 41     * = 42
+ = 43     , = 44     - = 45     . = 46     / = 47     0 = 48
1 = 49     2 = 50     3 = 51     4 = 52     5 = 53     6 = 54
7 = 55     8 = 56     9 = 57     : = 58     ; = 59     < = 60
= = 61     > = 62     ? = 63     @ = 64     A = 65     B = 66
C = 67     D = 68     E = 69     F = 70     G = 71     H = 72
I = 73     J = 74     K = 75     L = 76     M = 77     N = 78
O = 79     P = 80     Q = 81     R = 82     S = 83     T = 84
U = 85     V = 86     W = 87     X = 88     Y = 89     Z = 90
[ = 91     \ = 92     ] = 93     ^ = 94     _ = 95     ` = 96
a = 97     b = 98     c = 99     d = 100    e = 101    f =102
g = 103    h = 104    i = 105    j = 106    k = 107    l = 108
m = 109    n = 110    o = 111    p = 112    q = 113    r = 114
s = 115    t = 116    u = 117    v = 118    w = 119    x = 120
y = 121    z = 122    { = 123    | = 124    } = 125    ~ = 126
⌂ = 127    Ç = 128    ü = 129    é = 130    â = 131    ä = 132
à = 133    å = 134    ç = 135    ê = 136    ë = 137    è = 138
ï = 139    î = 140    ì = 141    Ä = 142    Å = 143    É = 144
æ = 145    Æ = 146    ô = 147    ö = 148    ò = 149    û = 150
ù = 151    ÿ = 152    Ö = 153    Ü = 154    ø = 155    £ = 156
Ø = 157    × = 158    ƒ = 159    á = 160    í = 161    ó = 162
ú = 163    ñ = 164    Ñ = 165    ª = 166    º = 167    ¿ = 168
® = 169    ¬ = 170    ½ = 171    ¼ = 172    ¡ = 173    « = 174
» = 175    ░ = 176    ▒ = 177    ▓ = 178    │ = 179    ┤ = 180
Á = 181    Â = 182    À = 183    © = 184    ╣ = 185    ║ = 186
╗ = 187    ╝ = 188    ¢ = 189    ¥ = 190    ┐ = 191    └ = 192
┴ = 193    ┬ = 194    ├ = 195    ─ = 196    ┼ = 197    ã = 198
Ã = 199    ╚ = 200    ╔ = 201    ╩ = 202    ╦ = 203    ╠ = 204
═ = 205    ╬ = 206    ¤ = 207    ð = 208    Ð = 209    Ê = 210
Ë = 211    È = 212    ı = 213    Í = 214    Î = 215    Ï = 216
┘ = 217    ┌ = 218    █ = 219    ▄ = 220    ¦ = 221    Ì = 222
▀ = 223    Ó = 224    ß = 225    Ô = 226    Ò = 227    õ = 228
Õ = 229    µ = 230    þ = 231    Þ = 232    Ú = 233    Û = 234
Ù = 235    ý = 236    Ý = 237    ¯ = 238    ´ = 239    - = 240
± = 241    ‗ = 242    ¾ = 243    ¶ = 244    § = 245    ÷ = 246
¸ = 247    ° = 248    ¨ = 249    · = 250    ¹ = 251    ³ = 252
² = 253    ■ = 254      = 255
```

Resultaten bekräftar koderna för de svenska tecknen vi fick tidigare.

2.5 Radfortsättning

Vi vet att escapesekvensen **\n** i koden åstadkommer radbyte i utskriften. Men hur gör man när man vill bryta rad i koden utan att åstadkomma radbyte i utskriften? Detta kan bli aktuellt t.ex. när det inte finns tillräcklig plats på samma rad i edit-fönstret? Eller om man vill bryta rad för bättre läslighet av koden? Ett exempel är:

```
cout << "Detta är en
                   utskriftsrad.";
```

Men denna radbrytning i koden ger kompileringsfel. Anledningen är att den görs mitt i en sträng som inte är avslutad. Det gäller nämligen regeln:

> Mitt i en sträng får man inte utan vidare åtgärd bryta rad i C++ kod.

Generellt kan **Enter**, mellanslag och tabulator vara lämpliga ställen för radbrytning i koden. De kallas *vita tecken.* T.ex. kan man bryta rad i koden på alla ställen där ett mellanslag förekommer. Detta gälller dock *inte* för mellanslag mitt i en sträng. Detta är innebörden i regeln ovan.

Tidigare har vi löst problemet genom *konkatenering*, dvs att dela upp strängen i *två* strängar och – som en ytterligare förenkling – att utelämna utmatningsoperatorn:

```
cout << "Detta är en "
        "utskriftsrad.";
```

I koden ovan har vi *två* strängar som **cout**-satsen konkatenerar automatiskt. Alternativet vore radfortsättning som bibehåller *en* sträng. Detta gör man med:

Radfortsättningstecknet \

Det speciella tecknet för att åstadkomma radfortsättning i C/C++ kod är:

backslash **** *direkt åtföljt av* **Enter** *utan mellanslag*

Kom ihåg att **** *åtföljt av ett tecken* ger en escapesekvens. Om detta tecken är **Enter** betyder escapesekvensen radfortsättning. Vi kan alltså ersätta den inledande satsen oven som gav kompileringsfel med följande sats:

```
cout << "Detta är en \
utskriftsrad.";
```

OBS! Man ser inte **Enter**-tecknet, men vid editering måste **Enter** tryckas direkt efter **** . Allt annat kommer att producera oönskat tomrum i utskriften av strängen. Detta gäller även för den andra kodraden. Därför måste den skrivas längst till vänster för att inte i onödan få ett tomrum. Hur man hanterar indragningen på en ny rad i **main()** är en fråga om läsligheten av kod, vilket borde lösas från fall till fall.

Följande program jämför radfortsättning i koden med konkatenering inom sammanhängande strängar:

```
// LineContin.cpp
// Radfortsättningstecknet:
// backslash \ direkt åtföljt av Enter utan mellanslag

#include <iostream>
using namespace std;

int main()
{                                // Radfortsättning: EN sträng
   cout << "\n\tDetta är endast en sträng kodad med \
radfortsättningstecknet.\n";
                                 // Konkatenering: TVÅ strängar
   cout << "\n\tAlternativet är konkatenering "
           "som skrivs i två strängar i koden.\n";
}
```

Att den andra raden i den första **cout**-satsen inte är indragen är här som en ful, men nödvändig konsekvens av radfortsättning. Annars skulle utskriften producera oönskat mellanrum i strängen på det stället där radfortsättningstecknet finns. En körning av programmet ovan ger utskriften:

```

   Detta är endast en sträng kodad med radfortsättningstecknet.

   Alternativet är konkatenering som skrivs i två strängar i koden.
```

Oönskat mellanrum mellan orden **med** och **radfortsättningstecknet** är resultatet av att den andra raden i den första **cout**-satsen inte är indragen. Efter radfortsättningstecknet har man i koden tryckt **Enter** utan mellanslag.

Radfortsättningstecknet kan användas när det t.ex. inte finns tillräckligt med utrymme på samma rad i editfönstret, vilket är just fallet i exemplet ovan.

2.6 Objektorienterad initiering

Ett helt nytt sätt att skapa initierade variabler av enkel datatyp visar följande program:

```
// ObjInit.cpp
// Initiering av variabler på ett objektorienterat sätt:
// Int-variablerna no1, no2 initieras som om de vore objekt

#include <iostream>
using namespace std;

int main()
{
   int no1(5);                 // Deklaration och initiering
   int no2(3);                 // på objektorienterat sätt

   cout << "\n Summan av " << no1 << " och "
        << no2 << " är " << (no1+no2) << "\n\n";
}
```

Skillnaden till våra program hittills är att tilldelningsoperatorn **`=`** inte längre används. Istället har vi den lite kryptiska koden:

```
int no1(5);
```

som gör exakt samma sak som:

```
int no1 = 5;
```

Båda satser definierar variabeln **`no1`** och *initierar* den samtidigt till värdet **`5`**.

Avsaknaden av tilldelningsoperatorn **`=`** kan vara av fördel i satsen **`int no1(5);`** eftersom definition och initiering smälter ihop till *en* operation: Man skapar en variabel och initierar den samtidigt. Det blir en vana att aldrig definiera en oinitierad variabel. En annan fördel är att inget problem angående operatorprioritet kan uppstå när andra operatorer är inblandade. T.ex. skulle satsen

```
int no1(4+3);
```

skapa variabeln **`no1`** och initiera den till värdet **`7`** utan någon konkurrens om prioritet mellan tilldelnings- och additionsoperatorn, dvs vilken av dem som ska utföras först.

Detta sätt att koda används i objektorienterad programmering. ”Objektet” är i vårt fall variabeln **`no1`** som skapas som ett exemplar av den fördefinierade datatypen (”klassen”) **`int`**. Samtidigt anropas en funktion – parenteserna **`()`** är symbolen för den. Vid anropet skickas initialvärdet **`5`** till variabeln.

Trots fördelarna med detta skriv- och tänkesätt kommer vi att fortsätta använda tilldelningsoperatorn för initieringen av variabler av *enkla* datatyper, men kommer att använda den nya terminologin när vi hanterar objekt. Programmet **`ObjInit`** producerar följande utskrift:

```
Summan av 5 och 3 är 8
```

Som sagt är initieringsproblematiken så generell att man i C++ har konstruerat det automatiska verktyget *konsruktor* för initiering av objekt. ”Automatiskt” därför att den alltid finns med i varje C++ klass, vare sig vi definierar den själva eller kompilatorn gör det åt oss by default.

Jämförelsen av den enkla datatypen **`int`** med en klass är endast en liknelse, av vilken man inte får dra den felaktiga slutsatsen att **`int`** vore en klass i C++ och **`no1`** ett objekt av den. Ändå finns det starka skäl till att fortsätta tänka i liknelsens banor: Alla enkla datatyper kan uppfattas som klasser i embryonalt tillstånd. Det nyare objektorienterade programmeringsspråket C# som utvecklades år 2000 – ca. 20 år efter C++ – har transformerat alla enkla datatyper till klasser och kallat det *Type system unification*.

2.7 Automatisk typkonvertering

C++ är ett *strikt typbestämt programmeringsspråk*, på eng. *strongly typed language*, vilket innebär att all data måste ha en datatyp. Utan datatyp kan data inte bearbetas i ett C++ program. Detta gäller både för enskilda värden och uttryck, dvs när flera värden är inblandade i ett uttryck, både i form av variabler och konstanter. Man kan inte utesluta att de har *olika* datatyper. Om flera olika datatyper är inblandade i ett uttryck, vilken datatyp ska då uttryckets värde (beräkningens resultat) slutligen få?

T.ex. vilken datatyp ska **`a + b`** få om **`a`** är en **`int`**, men **`b`** en **`double`**? För att lösa problem av den här typen tillämpar C++ kompilatorn automatiskt vissa regler.

Explicit typkonvertering innebär att programmeraren själv uttryckligen bestämmer när och hur ett värde ska byta datatyp. Vi använde explicit typkonvertering för att få reda på tecknens ASCII-koder genom att omvandla **`char`** till **`int`** och omvänt. Men typkonvertering kan även ske utan vårt uttryckliga medgivande:

Automatisk typkonvertering görs alltid när olika enkla datatyper är inblandade i en tilldelning eller i ett aritmetiskt uttryck. Även tilldelning, därför att C++ tolkar tilldelningstecknet **`=`** som en *operator* precis som vilken annan operator som helst, t.ex. **`+`** eller **`-`** . I detta avsnitt behandlar vi endast regler för de enkla datatyperna.

Automatisk typkonvertering vid tilldelning

En omvandling av datatypen utförs enligt inbyggda generella regler i C++. Det finns tre regler som kompilatorn använder automatiskt. Tilldelningsregeln tillämpas när olika enkla datatyper förekommer på olika sidor av tilldelningstecknet.

> **1. Tilldelningsregeln:**
>
> Är olika enkla datatyper involverade vid en tilldelning, konverteras alltid till den datatyp som står till vänster om tilldelningstecknet.

I programmet **`AssignRule`** nedan följer ett enkelt exempel på tilldelningsregeln:

I tilldelningen **`b = a`** (andra satsen i **`main()`**) förekommer datatypen **`short`** på vänster och **`int`** på höger sidan av tilldelningstecknet. Detta medför att kompilatorn omvandlar variabeln **`a`**:s **`int`**-värde som är 60 000 till ett **`short`**-värde – allt enligt tilldelningsregeln: det konverteras till den datatyp som står till vänster om tilldelningstecknet. 60 000 lagras ursprungligen i 4 bytes p.g.a. **`int a`**. Det är helt OK eftersom 60 000 ligger inom det tillåtna värdeområdet för **`int`**. Men försöket att stoppa 60 000 i en **`short`**-variabel **`b`** kan inte lyckas, eftersom den har endast har 2 bytes till förfogande p.g.a. **`short b`**. 60 000 överstiger (**`a`**:s värde) datatypen **`short`**:s övre gräns som är 32 767. Lådan **`b`** är för liten för att rymma 60 000.

```
// AssignRule.cpp
// Demonstrerar tilldelningsregeln som leder till overflow

#include <iostream>
using namespace std;

int main()
{
   int a = 60000;
   short b = a;          // Vid denna tilldelning sker automa-
                         // tisk konvertering av int till short
   cout << "\n\t int a = " << a << ", men short b = " << b
        << "\n\n";
   cout << "\t Men a tar fortfarande " << sizeof(a)
        << " bytes.\n";
}
```

Eftersom lådan **b** är för liten för att rymma 60 000, blir det ett felaktigt resultat för **b** när vi kör **AssignRule**:

```
            int a = 60000, men short b = -5536

            Men a tar fortfarande 4 bytes.
```

Programmet kan både kompileras och exekveras, men producerar ett felaktigt resultat. Fenomenet kallas för *overflow*, vilket innebär att minnet inte kan rymma den mängden data vi stoppar i det. Problemet är alltså minnesplatsbrist.

Mer exakt förklarat: Programmet räknar *modulo* 2^n där n är antalet bitar i minnesutrymme som den aktuella datatypen har till förfogande, i vårt exempel n = 2 därför att **short** har 2 bytes dvs 2 x 8 = 16 bitar, till förfogande. Det blir alltså *modulo* 2^{16} där 2^{16} = 65 536. Värdet ”slår runt” och hamnar på andra ändan av det ”cirkulära” talområdet: 60 000 – 65 536 = -5 536. Därför resultatet **-5536** i utskriften ovan.

Den andra raden i **AssignRule**:s körresultat visar att typkonvertering gäller *värdet*, inte variabeln, eftersom **sizeof(a)** returnerar **4** och inte **2**, där **a** är en **int**:

```
            Men a tar fortfarande 4 bytes.
```

All typkonvertering, både explicit och automatisk, tillämpas på *värden* inte på variabler. En variabel kan aldrig få en annan datatyp än den som den är deklarerad till. Även efter automatisk typkonvertering har variabeln **a** fortfarande datatypen **int**. Det är inte variabeln som byter datatyp till **short**, utan värdet. Typomvandlingen sker lokalt i enskilda programsatser i vilka regeln tillämpas, vilket inte påverkar resten av programmet.

Det farliga i programmet **AssignRule** är att kompilatorn inte ger något felmeddelande, inte ens en varning, och att exekveringen inte påverkas av overflow. Om man inte skriver ut **b**:s värde märker man inte ens att ett fel har inträffat.

I programmet **AssignRule** skulle man kunna ifrågasättta varför vi överhuvudtaget använder **short**. Skulle man inte kunna slippa problemet genom att använda **int** istället för **short**? Argumentet är berättigat just här, men inte generellt. För det första kan man undra varför det överhuvudtaget finns datatypen **short**, om man inte ska använda den? För det andra blir man i alla fall inte av med problemet eftersom man vid en tilldelning alltid kan råka ut för att ha en mindre datatyp till vänster om tilldelningstecknet än till höger. **short** var endast ett exempel.

Resultatet av typkonvertering behöver inte alltid bli helt felaktigt. Det kan även leda till förlust av noggrannhet, vilket är ännu svårare att upptäcka än ett helt felaktigt resultat. Programmet **Overflow** demonstrerar ett sådant fall och är ytterligare ett exempel på automatisk typkonvertering vid tilldelning:

```
// Overflow.cpp
#include <iostream>
using namespace std;

int main()
{
   float f = 3.96875;     // Automatisk konvertering av float
   int   i = f;           // till int: Alla decimalerna kapas
   cout << "\n\tfloat f = "<< f <<", men int i = "<< i
                                                  << "\n\n";
   short max = SHRT_MAX;  // Gränsen för datatypen short: ok
   short s = max + 1;     // Autom. konv. av int till short:
                          // s ger fel värde då det överskri-
                          // der gränsen för short: Overflow
   cout << "\tMax. short = " << max << "\n\t short s = " << s
        << " ger fel värde! (tar " << sizeof(s)
        << " bytes)\n\tRätt int-värde = " << max + 1
        << " \t\t(tar " << sizeof(max + 1) << " bytes)\n";
}
```

Ett körresultat av ovanstående program är:

```
          float f = 3.96875, men int i = 3

          Max. short = 32767
          short s = -32768 ger fel värde! (tar 2 bytes)
          Rätt int-värde = 32768          (tar 4 bytes)
```

Detta visar att kompilatorn vid tilldelningen **i = f** omvandlat **float**-värdet **3.96875** till **int**-värdet **3** dvs kapat alla decimaler. Detta beroende på att datatypen **int** endast kan lagra heltal. Den kan varken lagra decimalpunkten eller decimalsiffrorna. I definitionen av datatypen **int** ingår dessa begrepp inte. Därför kan den inte heller avrunda korrekt till värdet 4. En avrundningsalgoritm förutsätter begreppen decimalpunkt och decimalsiffra. Det enda som **int** kan hantera är heltalsdelen av **3.96875** som är **3**. Argumentet att endast använda decimaltalstyperna **float** och **double** för att slippa problemet är inte hållbart då hanteringen av deci-

maltal är mer krävande än hanteringen av heltal. Att generellt använda t.ex. **double** i större program som t.ex. endast bearbetar heltal skulle försämra programmets prestanda avsevärt både ur snabbhets- och minnesekonomisk synpunkt.

I det andra exemplet i **Overflow** tilldelas **short**-variabeln **max** det maximalt tillåtna värdet **SHRT_MAX** vilket går bra då det på gränsen ryms i datatypen **short**:s minnesutrymme 2 bytes. **SHRT_MAX** fyller utrymmet med 16 ettor, 1 för förtecknet och 15 för värdet 32767. Redan **max + 1** ryms inte i 2 bytes utan behöver minst 3 bytes: När det tilldelas **short**-variabeln **s** konverteras värdet enligt tilldelningsregeln till datatypen **short** som p.g.a. platsbrist inte kan lagra värdet. Det blir overflow och ett felaktigt negativt värde lagras i **s**. För att få det korrekta resultatet måste själva konstanten **max + 1** direkt skrivas ut utan att lagras i en **short**-variabel. Att denna konstant är av typ **int** beror på **int**-regeln, se nedan. För att visa det låter vi i **Overflow** med **sizeof** bestämma minnesstorleken av detta **int**-värde. Utskriften ger **4** bytes för **max + 1** och **2** bytes för **short**-värdet **s**.

Automatisk typkonvertering vid aritmetiska operationer

Tilldelningsoperatorn är inte den enda som förorsakar automatisk typkonvertering. Även de aritmetiska operatorerna **+**, **–** , ***** och **/** som tillsammans med variabler och konstanter bildar aritmetiska uttryck kan ge upphov till automatisk typkonvertering. Följande regel gäller och tillämpas av kompilatorn automatiskt:

> **2. int-regeln** (integral promotion):
>
> Är endast datatyperna **char**, **short** eller deras **unsigned**-typer inblandade i ett aritmetiskt uttryck, omvandlas uttryckets värde till datatypen **int**.

Omvandlingen till **int** ger namnet till denna regel som konstaterar att **int** är den *centrala* datatypen bland alla heltalstyper. Konverteringen går uppåt därför att **int** står högre än **char** och **short** i datatypshierarkin, när det gäller minnesstorleken. Omvandling uppåt i datatypshierarkin kallas även *befordring*, på eng. *promotion*. Därav namnet *integral promotion*. Här följer ett exempel på **int**-regeln:

```
// IntRule.cpp
#include <iostream>
using namespace std;

int main()
{
   char  a;
   short b;
   cout << "\n\tSumman av char och short tar " << sizeof(a+b)
        << " bytes.\n\n";
   cout <<"\tHeltalskonstanter tar "<<sizeof(-4)<<" bytes\n";
}
```

När man kompilerar detta program får man två varningar som talar om att variablerna **a** och **b** visserligen är definierade men inte tilldelade några värden fast vi bildar uttrycket **a + b**. Men vi använder ju inte uttryckets värde. Själva värdet är ointressant. Det vi är ute efter är uttryckets minnesstorlek som vi mäter med **sizeof**. I uttrycket **a + b** förekommer endast datatyperna **char** och **short** vilket enligt **int**-regeln gör att uttryckets värde konverteras till **int** fast **a** och **b** inte alls är av typ **int**. Beviset är att **sizeof** returnerar **4** vilket en körning visar:

```
Summan av char och short tar 4 bytes.

Heltalskonstanter tar 4 bytes.
```

I utskriften ovan bekräftas också att heltalskonstanten **-4** tolkas som **int** då den tar 4 bytes. Är datatypen till ett heltal inte specificerad genom definition är standarddatatypen alltid **int**. Hos konstanter som förekommer i koden, kan datatypen inte anges explicit då endast variabler definieras, inte konstanter. Därför:

Heltalskonstanter lagras automatiskt som **int**.

En annan sak är det med namngivna konstanter som definieras med det reserverade ordet **const** framför datatypen. Regeln borde alltså preciseras: Icke-namngivna heltalskonstanter lagras automatiskt som **int**. Både **int** och **long** 4 bytes, men även i andra system där **int** och **long** är av olika längd kommer körningar av **IntRule** eller liknande program att bekräfta **int**-regeln.

int-regeln visar att **int** är en slags favoritdatatyp för heltal. Alla rutiner kring **int** är optimerade så att C++ räknar snabbast med **int**. Ändå är **int**-regelns användningsområde begränsat till datatyperna **char**, **short** och deras **unsigned**-typer.

Vad händer om även andra datatyper decimaltal förekommer i ett uttryck? Vilken datatyp ska **a+b** få om **a** är av typ **int**, men **b** av typ **double**? Regel är:

3. Befordringsregeln:

Är minst en operand i ett aritmetiskt uttryck av typ int eller högre, konverteras uttryckets värde uppåt (befordring) enligt följande hierarki:

char → unsigned char → short → unsigned short → int →
unsigned int → long → unsigned long → float → double → long double

Att konverteringen görs uppåt är logiskt. Man vill ju förhindra overflow genom att ställa till förfogande ett större minnesutrymme för uttrycket än det finns för operanderna. Att det ändå kan uppstå problem p.g.a. skillnaden mellan de teckenlösa **unsigned**-typerna och datatyperna med förtecken visar följande exempel med endast heltalstyper:

```
// PromotInt.cpp
// Automatisk typkonvertering: Befordringsregeln
// Exempel med heltal
#include <iostream>
using namespace std;

int main()
{
   int   i =  5;
   short s = -3;
   cout << "\n          int 5 multiplicerad med short -3 ger "
        << i * s << " \n\n";
   unsigned int ui = 5;
   cout << "unsigned int 5 multiplicerad med short -3 ger "
        << ui * s << "\n";
}
```

Programmet innehåller två exempel varav det ena är OK medan det andra producerar ett felaktigt resultat. En körning visar:

```
         int 5 multiplicerad med short -3 ger -15

unsigned int 5 multiplicerad med short -3 ger 4294967281
```

I det första exemplet omvandlar kompilatorn värdet i uttrycket **`i * s`** enligt befordringsregeln till **`int`** därför att **`int`** står högre än **`short`** i hierarkin bland de inblandade datatyperna. Eftersom **`int`** även kan lagra negativa tal blir resultatet helt korrekt, nämligen **`-15`**, enligt matematiken: $5 \cdot (-3) = -15$. I det andra exemplet däremot omvandlar kompilatorn uttrycket **`ui * s`** enligt befordringsregeln till **`unsigned int`** därför att **`unsigned int`** står högre än **`short`**. Men eftersom **`unsigned int`** inte kan lagra det negativa talet **`-15`** blir resultatet fel.

`PromotInt` demonstrerade befordringsregeln med exempel som endast behandlade heltal. Vi ska även ha ett exempel på befordringsregeln som behandlar decimaltal.

En direkt slutsats av befordringsregeln är: Kan en decimaltalstyp inte specificeras via deklarationen, så är standarddatatypen för decimaltalskonstanter alltid **`double`**:

> Decimaltalskonstanter lagras automatiskt som **`double`**.

Ett exempel på denna slutsats är följande program som demonstrerar befordringsregeln med decimaltal och visar **`double`**:s roll som standarddatatyp för decimaltalskonstanter.

```
// PromotDec.cpp
// Automatisk typkonvertering: Befordringsregeln
// Exempel med decimaltal
#include <iostream>
using namespace std;

int main()
{
   short  s;
   float  f;
   double d;
   cout << "Summan short + float  tar "
        << sizeof(s + f) << " bytes (float).\n\n";
   cout << "Summan short + double tar "
        << sizeof(s + d) << " bytes (double).\n\n";
   cout << "Decimaltalskonstant 4.2  tar " << sizeof(4.2)
        << " bytes (double).\n\n";
   cout << "Decimaltalskonstant 4.2f tar "
        << sizeof(4.2f) << " bytes (float).\n\n";
}
```

Vill man inte ha decimaltalskonstanten som **double** måste man låta konstanten åtföljas av ett **f** som står för **float**. Återigen får man varningar vid kompilering då variablerna **s**, **f** och **d** inte är tilldelade några värden. Vi kan negligera dessa då vi ju bara är ute efter minnesstorleken när vi bildar uttryck med dessa variabler. I uttrycket **s + f** förekommer datatyperna **short** och **float** vilket enligt befordringsregeln gör att uttryckets värde konverteras till **float** därför att **float** står högre i hierarkin. Att befordringsregeln tillämpas här beror på att i uttrycket förekommer **float** som är högre än **int**. Att **sizeof(s + f)** returnerar **4**, minnesstorleken för **float** – den andra operandens datatyp **short** tar ju 2 bytes – visar följande körning:

```
Summan short + float  tar 4 bytes (float).

Summan short + double tar 8 bytes (double).

Decimaltalskonstant 4.2  tar 8 bytes (double).

Decimaltalskonstant 4.2f tar 4 bytes (float).
```

Uttrycket **s + d** däremot med de inblandade datatyperna **short** och **double** konverteras till **double** därför att **double** står högre i hierarkin. Dessutom visar exemplet att decimaltalskonstanten **4.2** lagras som **double** medan **4.2f** lagras som **float**. När det gäller decimaltal har man alltså möjligheten att ändra den automatiska lagringen av konstanter som **double** i 8 bytes till **float** i 4 bytes genom att skriva ett **f** (= **f**loat) utan mellanslag efter konstanten. Även ett heltal med **f** t.ex. **4f** blir omvandlat till **float**, en slags explicit typkonvertering för decimaltalskonstanter.

2.8 Rekursion

Rekursion är ett koncept inom problemlösning och programmering som tillämpar successiv upprepning av delar av en algoritm. Rekursiva metoder och funktioner är sådana som anropar sig själva, ungefär som hundar som bitar sig i svansen. Ordet rekursiv kommer från *recurrere* på latin som på engelska betyder *to run back* eller *to run again* dvs att gå tillbaka och köra igen.

Upprepade aktioner kan implementeras med kontrollstrukturer, främst *loopar*, s.k. *iterativa* lösningar, men även med *rekursiva* algoritmer, främst rekursiva funktioner eller metoder. Dessa kan ersätta loopar och genererar ofta kort och elegant kod, ofta rena översättningar av matematiska *rekursionsformler*.

Ett exempel på problem som kan lösas rekursivt är följande uppgift som den italienske matematikern *Leonardo Pisano Fibonacci* år 1202 formulerade i sin bok *Liber abaci* (Boken om räknekonsten). Den handlar om kaniners fortplantning:

> Ett kaninpar föder från den andra månaden av sin tillvaro ett nytt par varje månad. Samma gäller för de nya paren.
>
> Hur många par kommer att finnas om två år, om Fibonaccis observation är korrekt?

Fibonacci hade väl knappast kunnat drömma om att hans problem skulle bli föremål för datoriserade lösningar med rekursion mer än 800 år senare.

Om vi följer uppgiftens lydelse och räknar fram de första månaderna kan vi efter viss eftertanke ställa upp följande samband mellan antalet månader och antalet kaninpar:

Antal månader	1	2	3	4	5	6	7	8	…
Antal kaninpar	1	1	2	3	5	8	13	21	…

Det uppstår en talsekvens i den andra raden av tabellen som kallas för *Fibonaccis talsekvens* eller kort *Fibonaccitalen*. Så här får man sekvensen:

De två första månaderna finns det 1 kaninpar. De föder sitt första barnpar först efter 2 månader dvs i månad nr 3, varför det finns 2 kaninpar i månad 3. I månad 4 föder det första paret sitt andra barnpar, varför det finns 3 par i månad 4. I månad 5 föder det första paret sitt tredje barnpar, men även deras första barnpar föder ett nytt par, eftersom det har gått 2 månader sedan deras födelse. Därför finns det 5 par i månad 5. Osv. …

Praktiskt taget blir det allt svårare att hålla reda på antalet kaninpar när antalet månader växer. Man måste kanske rita någon sorts diagram och anteckna allt från månad till månad. En utväg ur dilemmat vore att upptäcka ett mönster, en struktur, t.ex.

ett samband mellan antal månader och kaninpar, en slags laglighet i bildandet av fibonaccitalen som kan beskrivas i form av en algoritm för att sedan kunna skrivas som program. Undersöker man tabellen noga kan man se följande enkelt *mönster:* Summan av två på varandra följande fibonaccital ger nästa fibonaccital. Kolla själv! Men hur kav vi beskriva detta mönster?

Vi inför beteckningarna: **n = Antalet månader**
Fn = Antalet kaninpar i månaden n

Mönstret som vi upptäckte ovan kan vi nu beskriva matematiskt så här:

$$\mathbf{F_1 = 1, \qquad F_2 = 1}$$
$$\mathbf{F_n = F_{n-1} + F_{n-2} \text{ för } n = 3, 4, 5, \ldots}$$

Den första raden säger att de första två fibonaccitalen är **1** och **1**. Den andra raden säger att det **n**-te fibonaccitalet är summan av de två föregående, vilket är bara en annan formulering av samma mönster vi upptäckte i tabellen. Formeln ovan kallas *Fibonaccis rekursionsformel.*

Vad är det rekursiva i denna formel? I en vanlig, icke-rekursiv formel står den *sökta* storheten vänster om likhetstecknet och alla *givna* storheter höger om likhetstecknet. Men här står den sökta storheten, fibonaccitalen, på *båda* sidor likhetstecknet, fast för olika månader, för olika parametrar så att säga. För att beräkna ett fibonaccital måste man känna till de två föregående. Men eftersom vi har de två första $\mathbf{F_1}$ och **F2**, s.k. *startvärden*, kan vi beräkna alla andra successivt utgående från dessa startvärden. Att det sökta står på båda sidor likhetstecknet är alltså det rekursiva, vilket, när vi kodar formeln, resulterar i en funktion som anropar sig själv, fast med olika parametrar.

Följande rekursiv funktion implementerar Fibonaccis rekursionsformel:

```
// Fibonacci.h
// Rekursiv funktion fib() som returnerar fibonaccitalen
// Rekursiv därför att funktionen anropar sig själv 2 gånger
// i den andra return-satsen

int fib(int n)                        // Funktionens definition
{
    if (n == 1 || n == 2)
       return 1;
    else
       return fib(n-1) + fib(n-2);   // 2 rekursiva anrop i
}                                    // funktionens definition
```

Koden en ren översättning av Fibonaccis rekursionsformel till C++. Därför är den också väldigt kort och elegant, vilket är typiskt och den stora förelen för rekursiva funktioner. För `n=1` eller `2` returneras `1` som enligt formeln är det första fibonaccitalet. För alla andra `n` returneras summan av de två föregående fibonaccitalen dvs `fib(n-1) + fib(n-2)`. Men de i sin tur är var och en, anrop av `fib()`. Dessa anrop står i funktionens *definition*, vilket är just det *rekursiva*.

Ett anrop av **`fib(4)`** t.ex. resulterar i att **`fib(3)`** och **`fib(2)`** anropas, **`fib(3)`** i sin tur resulterar i att **`fib(2)`** och **`fib(1)`** anropas, osv. Varje anrop av metdoden resulterar i ett stort antal följdanrop. Växer n leder det till en väldigt stor mängd av beräkningar. Det är bara datorn som kan klara av denna beräkningskomplexitet. För stora Fibonaccital är tidsåtgången stor, vilket är priset man måste betala för den komprimerade och eleganta koden i funktionen **`fib()`**. Mer om detta finns på nästa sida under ***Beräkningskomplexitet***.

Vi testar vår rekursiva funktion **`fib()`** i följande program:

```
// FibonacciTest.cpp
// Testar funktionen fib() genom att anropa den för de
// första 30 fibonaccitalen och skriva ut dem
#include <iostream>
using namespace std;
#include "Fibonacci.h"

int main()
{
   cout << "\n\tDe första 30 Fibonaccitalen:\n\n\t";
   for (int i = 1; i <= 30; i++)
   {
       cout << fib(i) << "\t";          // Vanliga anrop
       if (i % 6 == 0)
          cout << "\n\t";
   }
}
```

Funktionen **`fib()`** anropas i **`for`**-satsen, vars räknare **`i`** är funktionens parameter. Anropen genererar de första 30 fibonaccitalen. I var 6:e utskrift läggs in ett radbyte för layoutens skull. Här körresultatet:

```
        De första 30 Fibonaccitalen:
        1       1       2       3       5       8
        13      21      34      55      89      144
        233     377     610     987     1597    2584
        4181    6765    10946   17711   28657   46368
        75025   121393  196418  317811  514229  832040
```

Som man ser växer fibonaccitalen, dvs ökar kaninpopulationen, ganska fort. Så kan vi besvara den inledande frågan: Det kommer att finnas **144** kaninpar om ett år (12 månader). För det andra året (24 månader) växer kaninpopulationen till **46 368** kaninpar.

Beräkningskomplexitet

Man kan ju undra varför vi låter programmet `FibonacciTest` beräkna fibonaccitalen inte längre än 30 styck. Anledningen är att datorn inte klarar av mer, dvs efter 30 blir det väldigt trögt att få fram en utskrift. Efter 40 – självklart beroende på datorns prestationsförmåga – blir det praktiskt taget omöjligt. Det verkar som om datorn dör. I själva verket tar det timmar att beräkna `fib(40)`. Detta beror på ett fenomen som är typiskt för vissa rekursiva funktioner, även om inte för alla. Och det är beräkningskomplexiteten. Å andra sidan lämpar sig sådana rekursiva funktioner för att testa datorns prestationsförmåga.

Fibonacci rekursionen, närmare bestämt funktionen `fib()`, har en stor beräkningskomplexitet. Man pratar om en exponentiellt växande tidskomplexitet av typ 2^n för att beräkna `fib(n)`. Dvs tidsåtgången för beräkningen växer med en faktor 2^n. T.ex. om det tar $2^4 = 16$ nanosekunder för att beräkna `fib(4)`, tar det 2^{40} dvs över 10^{12} nanosekunder (ca. 2½ timmar) för att beräkna `fib(40)`, vilket uppenbart är ineffektivt. Så är man intresserad av mer än 40 Fibonaccital är det effektivare att använda en alternativ icke-rekursiv funktion, t.ex. med hjälp av kontrollstrukturen *repetition*. Att skriva en sådan iterativ variant av Fibonaccis problem är inte svårt, se övn 3.7 på sid 78.

Därmed är det inte sagt att rekursiva funktioner alltid är ineffektiva. Det finns problem som effektivast kan lösas med rekrursiv teknik, t.ex. sortering eller att manipulera datastrukturer som träd och grafer. Ett annat problem är hur svårt det är att beskriva och implementera dessa algoritmer. Man borde alltså avväga från fall till fall om en rekursiv funktion eller en repetitiv kontrollstruktur ska användas. Allt man kan göra med rekursion kan man även göra med repetition.

2.9 Mer om flervägsval

Nästlar man **`if-else`**-satsen spontant utan närmare fundering kan man komma på nedanstående lösning på detta trevägsval. Frågan är: till vilken **`if`** hör **`else`** ?

```
// TrickyElse.cpp
// Testar om det är mindre , större eller lika med 17
// Val mellan 3 alternativ (trevägsval)
// I nästlade if-satser hör else automatiskt till närmaste if
// Dåligt exempel på nästling av if-else-satser
#include <iostream>
using namespace std;

int main()
{
   int guessedNo;
   cout << "Gissa tal mellan 1 och 20: ";
   cin >> guessedNo;
   if (guessedNo <= 17)
   {                    // Utan klamrar: felaktigt körresultat!
      if (guessedNo == 17)
         cout << "\n\tGrattis, du har gissat rätt!\n\n";
   }                    // Utan klamrar: felaktigt körresultat!
   else
      cout << "\n\tFör stort !\n\n";
   if (guessedNo < 17)
      cout << "\n\tFör litet !\n";
}
```

Luriga `else`

Programmet **`TrickyElse`** försöker att programmera ett enkelt ”Gissa tal”-spel. Användaren ska gissa programmets hemliga tal 17 i intervallet 1-20. Vi har alltså tre fall att behandla: mindre än 17, lika med 17 och större än 17. Ett sådant trevägsval kan man koda med nästlade **`if`**- och **`if-else`**-satser. Låt oss anta att en nybörjare har löst problemet att välja mellan tre alternativ, just med **`TrickyElse`** efter att i en lärobok om C++ ha läst följande regel:

> I nästlade **`if`**-satser hör **`else`** automatiskt till närmaste **`if`**.
> Vill man ändra det, måste man använda blockmarkering **`{ }`** .

Regeln ovan har i programexemplet **`TrickyElse`** tillämpats på följande sätt:

```
┌──────► if (guessedNo <= 17)
│        {
│           if (guessedNo == 17)
│              cout<<"\n\tGrattis, du har gissat rätt!\n\n";
│        }
└──────► else
            cout << "\n\tFör stort !\n\n";
```

Blockmarkering med klamrarna **{ }** har gjorts här endast för att para ihop de **`if`** och **`else`** som pilarna ovan pekar på. Annars hade det utpekade **`else`** automatiskt parats med det **`if`** som står inom blocket.

I det första fallet som visas ovan, är **`if-else`**-satsens villkor **`guessedNo <= 17`** och därför utförs satsen efter **`else`** endast om **`guessedNo > 17`**, vilket innebär att **`cout`**-satsen skriver ut **`För stort !`** Och detta är väl sant då i det fallet det inmatade talet **`guessedNo`** är större än **`17`**. Testa gärna!

I det andra fallet som visas nedan, dvs om blockmarkeringen fattas, paras enligt regeln ovan **`else`** automatiskt med det närmaste **`if`**. I så fall uppstår en annan **`if-else`**-sats med villkoret **`guessedNo == 17`** som är nästlad i den enkla **`if`**-satsen med villkoret **`guessedNo <= 17`**:

```
         if (guessedNo <= 17)

┌───────────► if (guessedNo == 17)
│                cout<<"\n\tGrattis, du har gissat rätt!\n\n";
│
└──────► else
            cout << "\n\tFör stort !\n\n";
```

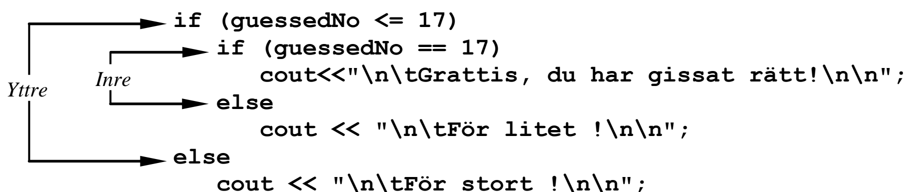
Detta innebär att vi får ingen utskrift alls om vi matar in tal större än **`17`** då villkoret i den överordnade **`if`**-satsen inte släpper in oss. Testa gärna! Och om vi matar in tal mindre än **`17`** får vi utskrifterna **`För stort !`** och **`För litet !`**. Den första utskriften härrör från programmets **`if-else`**-sats och den andra från den sista **`if`**-satsen. Hur som helst blir det uppenbarligen fel. Även testkörningar med olika värden visar att det blir fel svar utan blockmarkering. Låt oss alltså sätta tillbaka klamrarna på rätt plats och testköra **`TrickyElse`** med blockmarkering. Vi kommer att se att programmets svar vid alla inmatningar är korrekta. Ändå har vi i kommentaren betecknat **`TrickyElse`** som ett dåligt exempel på nästling av **`if-else`**-satser. Anledningen är för det första att det är dåligt strukturerat. I algoritmen bakom koden finns ingen systematik. Man anar *Trial and Error*-metoden. För det andra är programmet instabilt i den bemärkelsen att en mycket liten ändring, närmare bestämt borttagandet av klamrarna, får stora konsekvenser och leder till felaktiga resultat vid programmets körning utan att för den skull generera kompileringsfel.

Korrekt else

Nu ska vi förbättra programmet **TrickyElse** genom att skriva en bättre strukturerad nästlad **if-else**-sats så att vi blir av med onödiga klamrar:

```
// CorrectElse.cpp
// Gissa tal-spelet med nästlad if-else-sats
// Korrigerar luriga else med korrekt else
#include <iostream>
using namespace std;

int main()
{
   int guessedNo;
   cout << "Gissa tal mellan 1 och 20: ";
   cin >> guessedNo;
   if (guessedNo <= 17)
      if (guessedNo == 17)
         cout << "\n\tGrattis, du har gissat rätt!\n\n";
      else
         cout << "\n\tFör litet !\n\n";
   else
      cout << "\n\tFör stort !\n";
}
```

För att undvika att något **else** parar sig med ”fel” **if** (utan att använda klamrar) låter vi helt enkelt alla **else** hitta ”rätt” **if** automatiskt genom att behandla **if-else** alltid som par och inte hoppa över något **else**. Man kan jämföra det med parenteser i ett uttryck: Öppnar man en parentes (**if**) måste man även stänga den (**else**). Först behandlas i en yttre **if-else**-sats två fall **<= 17** och **> 17** varav det första i sin tur är sammansatt av två fall: **< 17** och **== 17**. Sedan löses det sammansatta fallet upp i en inre **if-else**-sats som nästlas i den yttre **if-else**-satsens **if**-del:



De tre relevanta testen av programmet **CorrectElse** med gissningar mindre än, större än och lika med 17 ger:

```
Gissa tal mellan 1 och 20: 12

        För litet !
```

```
Gissa tal mellan 1 och 20: 19

        För stort !
```

```
Gissa tal mellan 1 och 20: 17

        Grattis, du har gissat rätt!
```

`switch` *med tomma* `case`*-satser*

I ”Gissa tal”-spelet behandlades flervägsval mellan de tre alternativen **`< 17`**, **`> 17`** och **`== 17`** vilket löstes med nästlad **`if-else`**-sats (sid 56). Kan man inte utnyttja den klara strukturen hos **`switch`** för att behandla även detta och liknande flervägsval? Ja, det går även om **`switch`**-satsen bara testar på likhet. När en likhet inträffar utför **`switch`**-satsen inte bara de satser som omedelbart följer efter **`case`** utan *alla* satser som följer ända tills **`break`** kommer eller **`switch`**-satsen avslutas. Utnyttjar man denna möjlighet genom att skriva ”tomma” **`case`**-satser och utelämna **`break`**-satsen, kan man med **`switch`** lösa även flervägsval med olikheter. Jämförelser med **`<`**, **`<=`**, **`>`** och **`>=`** heter *olikheter*. Följande program demonstrerar denna möjlighet:

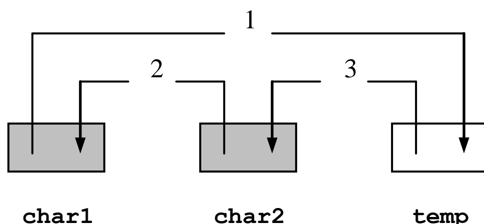
```
// SwitchInequ.cpp
// Gissa tal-spelet med switch-satsen
// Tomma case-satser utan break: flervägsval med olikheter
#include <iostream>
using namespace std;

int main()
{
   int guessedNo;
   cout << "Gissa ett tal mellan 1 och 10: ";
   cin >> guessedNo;
   cout << "\n\t";
   switch (guessedNo)
   {
      case 1:
      case 2:
      case 3:
      case 4:
      case 5:
      case 6:
        cout << "För litet !\n\n";
        break;
```

```
        case 7:
          cout << "\aGrattis, du gissade rätt!\n\n";
          break;
        case 8:
        case 9:
        case 10:
          cout << "För stort !\n\n";
          break;
        default:
         cout << "Du gissade tal utanför intervallet 1-10\n\n";
    }
}
```

För alla inmatningar **`1-6`** skrivs ut **`För litet !`** Inmatningen **`7`** ger **`Grattis, du har gissat rätt!`** Matar man in **`8-10`** får man utskriften **`För stort !`** Alla andra inmatningar ger ett lämpligt ”fel”meddelande. För att koden inte ska bli alltför stor har vi hållit oss till intervallet 1-10. Man kan alltså gruppera *flera* **`case`**-satser till *ett* fall genom att rada upp dem efter varandra, utelämna **`break`** och i den sista skriva det som skall utföras i just detta fall. I den sista får **`break`** förstås inte utelämnas.

2.10 Misslyckad modularisering

Låt oss anta vi har två tecken **char1** och **char2** som vi vill byta pltas på. För att kunna göra det behövs en tredje, temporär plats. Vi börjar med att lägga undan **char1** på den temporära platsen **temp** (steg 1). Sedan byter vi plats på **char2** och lägger det i **char1** som tömdes i steg 1 (steg 2). Och slutligen, i steg 3, lägger vi **char1** som under tiden mellanlagrats i **temp**, in i **char2** som tömdes i steg 2:



Illustrationen beskriver algoritmen för platsbyte av två tecken **char1** och **char2**, där 1, 2 och 3 anger aktionsordningen i algoritmen. En tredje, temporär plats **temp** behövs för att lägga undan det första, ev. felplacerade, tecknet. I följande program implementeras algoritmen:

```
// MiniSort.cpp
// Läser in 2 tecken och sorterar dem i teckentabellens ord-
// ning med hjälp av en algoritm för platsbyte av två objekt
#include <iostream>
using namespace std;

int main()
{
   char char1, char2, temp;
   cout << "\n\tTvå osorterade tecken:\n\n\t"
        << "Ge 2 olika tecken skilda med tabulator:   ";
   cin >> char1 >> char2;

   if (char1 > char2)      // tecknens ASCII-koder jämförs
   {
      temp  = char1;       // Algoritm för platsbyte
      char1 = char2;       // av två tecken
      char2 = temp;
   }

   cout << "\n\tDe inmatade tecknen förekommer"
        << "\n\ti teckentabellen i ordningen:\t\t  "
        << char1 << "\t" << char2 << "\n";
}
```

I följande körexempel byts plats på de inmatade tecknen **Z** och **A** som har blivit inmatade i fel ordning. De sorteras enligt teckentabellens ordning:

```
Två osorterade tecken:

Ge 2 olika tecken skilda med tabulator:    Z     A

De inmatade tecknen förekommer
i teckentabellen i ordningen:              A     Z
```

Algoritmens kärna ligger i **`if`**-satsen med sina tre satser. I den första satsen lägger vi undan **`char1`**:s värde i **`temp`** (steg 1 i bilden ovan). I den andra satsen byter vi plats på **`char2`**:s värde och lägger det i **`char1`** (steg 2). Och slutligen läggs **`temp`** som under tiden har mellanlagrat **`char1`**:s värde, in i **`char2`** (steg 3). Platsbytet på **`char1`** och **`char2`** äger endast rum om de inmatade teckenvärdena är felplacerade dvs endast om **`char1 > char2`**. Annars behåller de sina platser.

I körexemplet ovan jämför **`if`**-satsens villkor **`char1 > char2`** värdena **`Z`** och **`A`** med varandra. Men tecken kan inte sättas i en relation av typ ”större än” till varandra. I själva verket är det Unicode-koderna till **`Z`** och **`A`** som jämförs med varandra. Det är endast tal som kan jämföras med varandra. Jämförelseoperatorn **`>`** behandlar **`char`**-variablerna **`char1`** och **`char2`** som *tal* precis som aritmetiska operatorer gör.

Försök att modularisera MiniSort

I programmet **`MiniSort`** (sid 59) lyckades vi att implementera algoritmen som kan användas för att sortera även större datamängder, eftersom en sådan algoritm bygger på sortering av två objekt. Men för att kunna göra det måste vi separera den från det aktuella program som vi testade algoritmen i, dvs vi måste modularisera den och skriva den som en separat funktion. Detta ska vi försöka göra nu. Så här skulle en sådan funktion se ut, när vi separerar koden som utgör algoritmen från **`MiniSort`**. På engelska kallas denna algoritm för *Swap* eller *Swapping*.

```
// NoSort.h
// Funktionen TrySwap() som tar in 2 tecken t1 och t2
// och byter plats på dem enligt algoritmen MiniSort (sid 59)

void TrySwap(char t1, char t2)
{
   char temp;
   if (t1 > t2)
   {
      temp = t1;                      // Algoritm för platsbyte
      t1   = t2;                      // av de två tecknen
      t2   = temp;                    // t1 och t2
   }
}
```

Algoritmdelen av **`MiniSort`** (sid 59) har flyttats till en funktion där **`t1`** och **`t2`** är funktionens formella parametrar. Funktionen **`TrySwap`** anropas i följande program med de aktuella parametrarna **`char1`** och **`char2`**:

```
// NoSortTest.cpp
// Läser in 2 tecken char1 och char2, skickar dem till meto-
// den TrySwap() i klassen NoSort som ska sortera dem
#include <iostream>
using namespace std;
#include "NoSort.h"

int main()
{
   char char1, char2;
   cout << "\n\tTvå osorterade tecken:\n\n\t"
        << "Ge 2 olika tecken skilda med tabulator:   ";
   cin >> char1 >> char2;

   TrySwap(char1, char2);            // Funktionsanrop

   cout << "\n\tDe inmatade tecknen förekommer"
        << "\n\ti teckentabellen i ordningen:\t\t  "
        << char1 << "\t" << char2 << "\n";
}
```

Att vi kallar filen som lagrar funktionen **TrySwap** för **NoSort** förstår man när man testkör programmet **NoSortTest**. Koden kan både kompileras och exekveras. Det finns inget syntax- eller annat fel i programmet. Det är bara att ingen sortering sker. Tecknen förblir osorterade. Matar man in dem i fel ordning skrivs de ut även i fel ordning – till skillnad från det icke-modulariserade programmet **MiniSort**.

Följande körexempel visar att programmet inte gör som vi vill:

```
        Två osorterade tecken:

        Ge 2 olika tecken skilda med tabulator:   Z     A

        De inmatade tecknen förekommer
        i teckentabellen i ordningen:             Z     A
```

Testa gärna själv. Och om du tror att det beror på att de formella parametrarna **t1** och **t2** i funktionen **TrySwap** har andra namn än de aktuella **char1** och **char2** i programmet **NoSortTest** prova gärna att välja samma namn i båda. Det är inte fel ur varken kompilerings- eller exekveringssynpunkt. Bara att tecknen inte sorteras.

Felet är ett tanke- resp. ett kunskapsfel, om man nu kan beteckna det så. Vi har antagligen inte tillräckliga kunskaper om vad som händer när man lägger koden till *ett* program i *två* olika moduler. Närmare bestämt vet vi inte exakt *hur* parametrarna överförs från den ena till den andra modulen. Därför behandlar vi i nästa avsnitt denna fråga. Det finns nämligen inte bara i C++ utan i alla programmeringsspråk olika metoder för överföring av parametrar mellan en funktions definition och dess anrop. Avgörande för valet mellan dessa metoder är parametrarnas datatyper. Vi ska lösa problemet med *referenser* som är besläktade med pekare. Om pekare läs kap 6, sid 162.

2.11 Referenser

Låt oss börja med ett program som producerar en märklig utskrift:

```
// Reference.cpp
// Demontrerar datatypen referens till int
// Använder en int-variabel och en referens (alias) till den
// Ändrar variabelns värde via referensen
#include <iostream>
using namespace std;

int main()
{
   int  no;                     // En minnescell reserveras
   int& ref = no;               // Ingen minnescell reserveras:
                                // ref endast referens till no
   no = 5;
   ref = 10;                    // Ändring av no:s värde via ref
   cout << "\n\t Summan av " << no << " och "
        << ref << " är " << no + ref << "\n";
}
```

Lite eftertanke behövs för att förstå programkörningen av **`Reference`**:

```
         Summan av 10 och 10 är 20
```

Borde körningen inte ge utskriften: **`Summan av 5 och 10 är 15`** ? Detta med tanke på att variabeln **`no`** tilldelas värdet **`5`** och variabeln **`ref`** värdet **`10`**. Varför blir då summan **`no + ref`** i utskriften **`20`** och inte **`15`**? När man letar efter variabeln **`ref`**:s deklaration lite längre upp, ser man att **`ref`** inte är deklarerad som en vanlig **`int`** utan som en **`int&`**. Men vad betyder detta och vad innebär det i praktiken? Det betyder att **`ref`** är av dataypen *referens till* **`int`**. Och i praktiken innebär detta att satsen **`ref = 10;`** inte tilldelar variabeln **`ref`** värdet **`10`**, utan att den tilldelar variabeln **`no`** värdet **`10`** via *referensen* **`ref`**.

Vad är en referens?

Referenser är i C++ en helt ny kategori av datatyper. De är inte längre enkla datatyper. De uppstår genom att man lägger till ampersand-tecknet **`&`** till en enkel datatyp, t.ex. **`int&`**, **`char&`** eller **`float&`**, **`double&`**,… . På så sätt får man till varje enkel datatyp en ny datatyp som har den allmänna formen **`datatyp&`** och betecknas som *referens* till resp. datatyp: T.ex. är **`int&`** en ”*referens till* **`int`**” dvs ett *alias*, ett andra *namn* till en redan befintlig **`int`**. I exemplet ovan är **`ref`** ett alias, ett andra namn, en referens till variabeln **`no`**. **`ref`** själv har inget (vanligt) värde.

För datatypen referens behövs inte något speciellt reserverat ord som heter *reference* eller liknande för ett sådant finns inte i C++. Istället läggs symbolen **`&`** till en redan känd datatyp för att skapa den nya datatypen. Därför måste det – innan man gör det

– finnas redan en vanlig variabel som referensvariabeln ska referera till. I exemplet ovan definieras först den vanliga variabeln `no` vilket reserverar en minnescell åt ett `int`-värde. Sedan kan referensvariabeln `ref` deklareras och initieras så här:

```
int& ref = no;
```

alternativt:

```
int &ref = no;
```

Båda varianter kan användas. Den första har fördelen att den ur läslighetssynpunkt lätt kan kännas igen som datatypen `int&` dvs *referens till* `int`. Den andra visar tydligare att variabeln `ref` är en referens- och inte en vanlig variabel. En referensvariabel *måste* initieras i deklarationssatsen, dvs deklaration och tilldelning kan inte separeras. En referensvariabel kan endast initieras med ett variabel*namn*, här med `no`, inte med ett *värde* eftersom referensen är endast ett alias. Dvs satserna ovan reserverar *inte* en ny minnescell. Tilldelningen kopierar *inte* något värde från minnescellen `no` till någon minnescell `ref`, för ett sådant finns inte. Det som händer är följande: *Referensvariabeln* `ref` tilldelas den vanliga variabeln `no`:s minnesadress, så att `ref` blir ett nytt namn – ett alias – till `no`. Allt som kan göras med variabeln `no` för att manipulera variabelns värde kan fr.o.m. nu även göras med referensen `ref`. I exemplet tilldelas minnescellen först värdet `5` med `no`, men sedan överskrivs värdet i nästa sats till `10` med referensen `ref`:

```
ref  ──────►  no  [ 5̸ 10 ]
```

Två olika betydelser av ampersand (&)

1. **I deklarationer be**talar ampersand `&` om för kompilatorn att den variabel som följer ska vara en *referensvariabel*. T.ex. betyder alla varianter:

   `int& ref` eller `int & ref` eller `int &ref`

   att variabeln `ref` deklareras till datatypen referens-till-`int` dvs en hänvisning eller ett *alias* till en annan, redan deklarerad `int`-variabel. Förekommer en av varianterna ovan i en funktions parameterlista, skickas den `int`-variabel som `ref` ska referera till, från den anropande funktionen, annars måste `ref` initieras med en `int`-variabel i samma sats som den deklareras. Se nedan och programmet `Reference`.

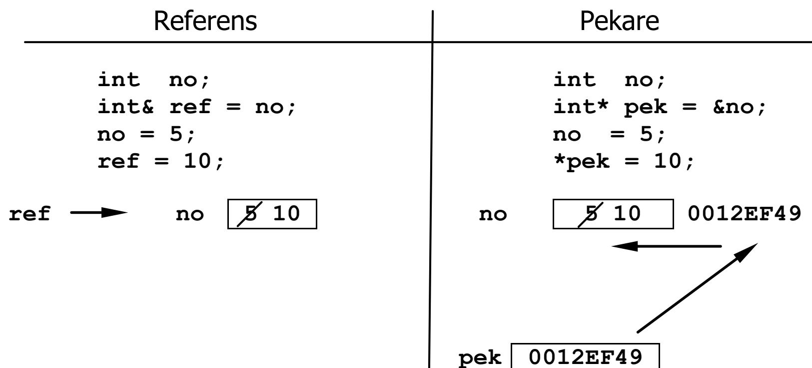
2. **I kod som inte är deklaration** betyder ampersand `&` *adressoperatorn*. När den skrivs framför en redan deklarerad variabel, säg `no`:

   `&no` eller `& no`

   returnerar den *adressen* till `no`. Sedan kan denna adress tilldelas en pekarvariabel. Se nedan och programmet `PointRef`.

Referens vs. pekare

Följande jämförelse ska illustrera släktskapet mellan referens och pekare. Om pekare läs kap 6, sid 162.



Båda kodvarianter gör exakt samma sak: Variabeln **`no`** skapas, initieras till ett värde och överskrivs sedan med ett nytt värde. I det ena fallet görs överskrivningen med en referens-, i det andra fallet med en pekarvariabel. Man ser direkt att referensvarianten är betydligt enklare, renare i kod och även mer läslig om man förstått referenskonceptet. Enkelheten har man uppnått genom att gömma det som man ser med pekarvarianten. Det kan vara av fördel i praktiskt sammanhang om man begränsar sig till detta exempel. Pekarvariantens fördel är att man kan arbeta vidare med adressen då den explicit står till förfogande i pekarens minnescell. Man kan tilldela pekarvariabeln en annan adress osv. vilket inte är möjligt med referensen. Hur som helst, visar jämförelsen ovan att en referens är en mer användarvänlig, automatiserad pekare med vissa begränsningar. Därför pratar man också om datatypen referens som en implicit eller *automatisk pekare*. I vissa tillämpningar räcker det fullt att använda referens istället för pekare. Kodexemplet i jämförelsen ovan ingår i följande program som är en pekarvariant av programmet **`Reference`** (sid 62):

```
// PointRef.cpp
// En vanlig variabel och en pekarvariabel som pekar på den
// Ändrar den vanliga variabelns värde med värdeoperatorn
// Pekare ersätter referensvariabeln i programmet Reference
// Adress- och värdeoperatorn måste användas för att göra
// samma sak som Reference automatiskt gör med referensvar.
// Slutsats: Datatypen referens är en automatisk pekare
#include <iostream>
using namespace std;

int main()
{
   int  no;                    // Vanlig variabel
   int* pek = &no;             // Pekarvariabel som pekar på
                               // vanlig variabel
```

```
   no = 5;                    // Initiering av vanlig variabel
   *pek = 10;                 // Ändring    av vanlig variabel

   cout << "\n\t Summan av " << no << " och "
        << *pek << " är " << no + *pek << "\n";
}
```

Körningen av **PointRef** ger förstås samma resultat som **Reference**:

```
         Summan av 10 och 10 är 20
```

Satsen **`*pek = 10;`** ändrar variabeln **no**:s värde som var **5**, tilldelad i satsen innan, så att det nya värdet blir **10**. Värdeoperatorn tillämpad på pekarvariabeln dvs **`*pek`**, returnerar en referens till den vanliga variabeln **no** då **pek** pekar på **no** p.g.a. **pek = &no**. **cout**-satsen kommer sedan åt det ändrade värdet både direkt med **no** och indirekt med **`*pek`** då båda refererar till samma minnescell. **`*pek`** ersätter referensvariabeln **ref** i **Reference**. På så sätt är datatypen referens en automatiserad pekare.

2.12 Parameteröverföringsmetoder

I det här avsnittet ska vi lära oss *på vilket sätt* parametrar överförs mellan funktioner. Det finns nämligen i C++ olika metoder för parameteröverföring, en av dem är *värdeanrop* (*Call by Value*). Som exempel tar vi det första programmet **`Function`** i början av kapitlet om funktioner. Varför har vi valt där andra namn för de aktuella **`tim, min, sek`** än för de formella parametrarna **`t, m, s`** fast de lagrar samma värden? Båda representerar timmar, minuter och sekunder. Frågan är: Lagras dessa värden i 3 eller 6 minnesceller? Om det är 3 vore valet av samma namn motiverat. Men om det är 6 vore det bättre att återspegla verkligheten även i koden genom att välja olika namn för de aktuella än för de formella parametrarna.

Värdeanrop (Call by value)

```
// CallByValue.cpp
// Värdeanrop: Vid funktionsanrop överförs värdena
// De formella parametrarna ändras i funktionen
// Men ändringen påverkar inte de aktuella parametrarna
#include <iostream>
using namespace std;
/**********************************************************/
int totalsek(int t, int m, int s)     // t, m, s är formella
{                                     // parametrar
   int resultat = 3600*t + 60*m + s; // resultat sparas undan
   t = m = s = 1;                     // Ändring av form. par.
   cout << "\nDe ändrade formella parametrarna " << t
        << " timme, " << m << " minut och " << s << " sek."
        << "\nhälsar från funktionen och ger "
        << 3600*t + 60*m + s << " totalsekunder.\n\n";
   return resultat;                   // resultat returneras
}
/**********************************************************/
int main()
{
   int tim, min, sek;                  // Aktuella parametrar
   cout << "\nGe tim, min, sek (skilda med mellanslag): ";
   cin  >> tim >> min >> sek ;

   cout << "De aktuella parametrarna " << tim << " timmar, "
      << min << " minuter och " << sek << " sekunder\nhälsar"
        << " från main() och ger " << totalsek(tim, min, sek)
        << " totalsekunder via anropet.\n\n";

   cout << "Men även efter anropet är\n"
        << "de aktuella parametrarna " << tim << " timmar, "
        << min << " minuter och " << sek << " sek.\n";
}
```

Parametrar som skrivs i en funktions *anrop* – i vårt exempel **`tim, min, sek`** – är aktuella parametrar, till skillnad från de formella parametrar – i vårt exempel **`t, m, s`** – som skrivs i funktionens *definition*. Med *aktuell* menas att de har aktuella värden som gäller vid anropet för att skickas till funktionens formella parametrar. Därför måste de vara väl definierade variabler eller konstanter. I exemplet ovan läses in de i **`main()`**. De formella parametrarna måste alltid vara variabler som deklareras i funktionen **`totalsek()`**:s parameterlista när denna skapas. Sina värden får de *första gången* inte tilldelade i funktionens kropp utan från de aktuella parametrarna vid funktionens anrop. Sedan ändras deras värden i funktionen: De sätts allihop till 1 för att testa vilken påverkan denna ändring har på de formella parametrarna. Men för att ändå kunna få resultatet med de ursprungliga värdena beräknas antalet totalsekunder och sparas undan i variabeln **`resultat`** som slutligen returneras från funk5tionen. Innan dess skrivs ut de värden som ändrats till 1 med totalsekunder från funktionen som i så fall måste resultera i 3661, nämligen 3600 + 60 + 1.

I **`main()`** skriver vi ut de aktuella parametrarnas värden före och efter anropet av funktionen för att se om de formella parametrarnas ändring i funktionen påverkar de aktuella parametrarna. Följande körexempel visar att detta inte är fallet:

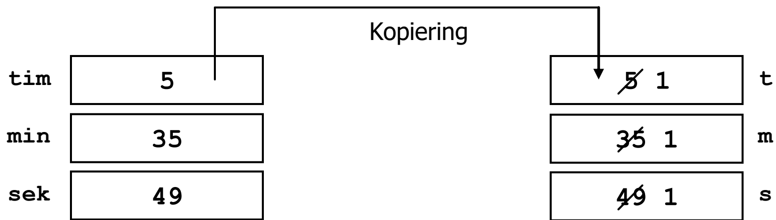
```
Ge tim, min, sek (skilda med mellanslag): 5 35 49

De ändrade formella parametrarna 1 timme, 1 minut och 1 sek.
hälsar från funktionen och ger 3661 totalsekunder.

De aktuella parametrarna 5 timmar, 35 minuter och 49 sekunder
hälsar från main() och ger 20149 totalsekunder via anropet.

Men även efter anropet är
de aktuella parametrarna 5 timmar, 35 minuter och 49 sek.
```

Körexemplet visar att de formella och aktuella parametrarna har var sitt eget liv. Det enda som relaterar dem till varandra är att de tar över värdena från varandra. Ändringen av de formella parametrarna påverkar inte alls de aktuella parametrarna. Av detta kan man dra slutsatsen att **`tim, min, sek`** och **`t, m, s`** är två olika uppsättnigar variabler. De lagras i 6 olika minnesceller. Även om vi skulle välja samma namn för dem – vilket vore tillåtet då de ligger i två olika funktioner och därmed i två olika block – kommer namnen fortfarande beteckna 6 olika minnesceller. För att återspegla denna verklighet vill vi i fortsättningen följa regeln att välja andra namn för de aktuella än för de formella parametrarna. Kodens läsare ska inte luras som om de vore samma variabler p.g.a. namnvalet. Undantag från denna regel kan vara motiverat när en annan princip – nämligen att välja *beskrivande* namn – går emot den.

En annan slutsats av körningen ovan är: Parameteröverföringen mellan funktionerna **`totalsek()`** och **`main()`** realiseras genom kopiering av värdena från de aktuella till de formella parametrarna. Denna parameteröverföringsmetod kallas *värdeanrop*

därför att det är själva *värden* som kopieras över när funktionen aropas. Minnesbilden av värdeanrop ser ut så här:

Värdeanropets minnesbild:



Ändring av kopiorna, de formella parametrarna `t`, `m`, `s`, påverkar inte originalen, de aktuella parametrarna `tim`, `min`, `sek`.

Vid denna parameteröverföringsmetod skapas alltid en *dubbel* uppsättning av minnesceller: 6 om vi har 3 parametrar. Därför leder värdeanrop oundvikligen till fördubblad minnesåtgång. Datatypen till respektive parameter är avgörande för den automatiska tillämpningen av värdeanrop. Det gäller följande regel:

> I C++ väljs automatiskt **värdeanrop** (Call by Value) för parameter-överföring vid funktionsanrop, om parametern är av **enkel datatyp**.

Fördubblingen av minnesåtgången anses inte som problem då enkla datatyper i alla fall tar upp relativt litet minnesutrymme. För datatyper som kräver större minnes-utrymme används en annan teknik som undviker denna fördubbling och som heter *referensanrop*. Det omvända av regeln ovan gäller inte. Dvs värdeanrop används även i andra sammanhang.

Ur minnessynpunkt är förstås fördubblingen av minnesåtgången en nackdel. Men värdeanrop har även fördelen att just p.g.a. minnesbilden ovan de formella och de aktuella parametrarna har var sitt liv och inte påverkar varandra. I vissa samman-hang är detta önskvärt, i andra inte. Så, beroende på applikationen kan man välja bland de två parameteröverföringsmetoderna värde- och referensanrop genom att välja rätt datatyp till sina parametrar. Enkel datatyp leder automatiskt till värdean-rop. Vilken datatyp som automatiskt leder till referensanrop ska vi ta upp nedan:

Referensanrop (Call by reference)

Här ska vi fortsätta diskussionen om de olika metoderna för parameteröverföring vid funktionsanrop. Värdeanrop (sid 62) använder sig av kopiering av parametervärdena till nya minnesceller och tillämpas när parametrarna är enkla datatyper. Nackdelen med värdeanrop är att den medför fördubbling av minnesåtgången. Alternativet till det är *referensanrop* som överför minnesadressen istället för värdet och där man

slipper denna nackdel. Referensanrop är relaterad till datatypen referens som behandlades i förra avsnitt varifrån också namnet härstammar. Anledningen till denna koppling är att parametrarnas datatyp automatiskt styr valet av överföringsmetoden. Det gäller nämligen:

> I C++ väljs automatiskt **referensanrop** (Call by reference) för parameteröverföring vid funktionsanrop, om parametern är av datatypen **referens**.

Samtidigt kommer vi att se att det för vissa problem t.o.m. är nödvändigt att använda referensanrop då det inte går att modularisersa dem med värdeanrop. Man vill t.ex. skicka vissa parametrar till en funktion där de ändras och man vill få tillbaka ändringen till huvudprogrammet. Ta följande exempel: Vi vill skicka två parametrar till en funktion som ska sortera dem. Skickar vi dem i fel ordning ska funktionen ställa dem i rätt ordning och skicka tillbaka dem i den rätta ordningen – grunden till alla sorteringsalgoritmer. Ett exempel på ett sådant problem som vi ska ta upp här, är modulariseringen av program **`MiniSort`** (sid 59). Regeln ovan är till för att för det första undvika fördubbling av minnesåtgång och för det andra för att ta del av den ändring som funktionen gör med parametervärdena, dvs för att kunna utnyttja ändringen även i det program som anropar funktionen – om det nu är önskvärt. Just detta är önskvärt i följande applikation. Programmet **`MiniSort`** som presenterade en algoritm för platsbyte mellan två tecken, ska nu modulariseras. Vi vill skriva själva algoritmen som en funktion med tanke på att den kommer att utvecklas till en allmän sorteringsalgoritm för större datamängder senare.

```
// Swapping.h
// Tar in 2 tecken och sorterar dem i teckentabellens ordning
// De ombytta parametrarna blir även vara ombytta i main()
// Parametrarna är deklarerade som referenser: Referensanrop

void swap(unsigned char& t1, unsigned char& t2)
{
   unsigned char temp;
   if (t1 > t2)
   {
      temp = t1;                      // Algoritm för platsbyte
      t1   = t2;                      // av de två teckenvärdena
      t2   = temp;                    // t1 och t2
   }
}
```

Bearbetningsdelen av **`MiniSort`** (sid 59) har flyttats till en **`void`**-funktion. Parametrarna **`t1`** och **`t2`** är deklarerade som referenser. De tar inte emot några teckenvärden från **`letter1`** och **`letter2`** (se nedan) utan endast deras adresser. **`t1`** och **`letter1`** är två namn till samma värde. Samma sak är det med **`t2`** och **`letter2`**. När värdena ändras i funktionen med **`t1`** och **`t2`** kan ändringen ses i **`main()`** med **`letter1`** och **`letter2`**:

```
// CallByRef.cpp
// Läser in 2 tecken, skickar dem till den externlagrade
// funktionen swap() som sorterar dem i teckentabellens ord-
// ning. Ändringen är synlig även i main() pga referensanrop
#include <iostream>
using namespace std;
#include "Swapping.h"

int main()
{
   unsigned char letter1, letter2;     // Vanliga variabler
   cout << "\nGe 2 olika tecken skilda med tabulator:   ";
   cin >> letter1 >> letter2;

   swap(letter1, letter2);             // Funktionsanropet

   cout << "\nDe inmatade tecknen förekommer \n"
        << "i teckentabellen i ordningen:\t\t  "
        << letter1 << '\t' << letter2 << "\n";
}
```

Funktionen **swap()** ställer i rätt ordning tecken som är inmatade i fel ordning. En körning av programmet **CallByRef** visar detta:

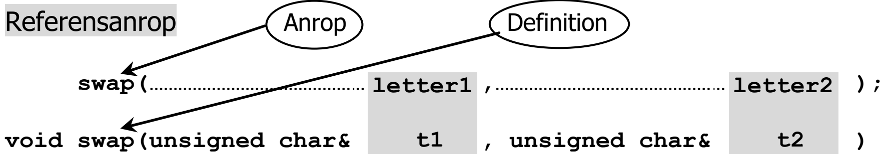
```

Ge 2 olika tecken skilda med tabulator:   Z     A

De inmatade tecknen förekommer
i teckentabellen i ordningen:             A     Z

```

Gör gärna följande test: Ta bort ampersand-tecknet **&** från definitionen av båda parametrarna i parameterlistan av funktionen **swap()**, så att **t1** och **t2** blir vanliga variabler, kompilera och kör. Du kommer inte få tecknen sorterade i rätt ordning om du matar in dem i fel ordning. Anledningen är att genom borttagningen av **&** blir **t1** och **t2** variabler av enkel datatyp så att värdeanrop tillämpas automatiskt. Dvs **t1** och **t2** får sina egna minnesceller vars värden kopieras över från **letter1** och **letter2**. Ändringen av **t1** och **t2** i funktionen kommer inte att påverka **letter1** och **letter2** i **main()**. Så, man ser att det är absolut nödvändigt att använda referensanrop i det här problemet, för det går inte att lösa det med värdeanrop.

Två olika minnesbilder

Här en gång till en jämförelse mellan värde- och referensanrop med hjälp av minnesbilderna. Den första raden är funktionens *anrop* **`swap()`** i det anropande programmet **`main()`**, den andra är huvudet till funktionens *definition:*

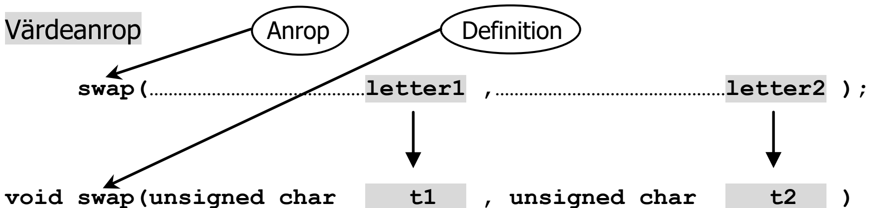
****



`t1` och **`letter1`** är två *olika namn* till *en och samma minnescell*. Båda namn kan användas för att komma åt värdet i den. Även **`t2`** och **`letter2`** är två olika namn till en och samma minnescell. Avgörande för denna minnesbild är att **`t1`** och **`t2`** är referenser.

****

En annan minnesbild uppstår vid värdeanrop. Även här är den första raden funktionens *anrop* i **`main()`**, den andra huvudet i funktionens *definition:*



`t1` *och* **`letter1`** *är två olika minnesceller*. Pilen (med tjockare spetsen) symboliserar kopiering av värdet vid funktionsanrop. Även **`t2`** *och* **`letter2`** *är två olika minnesceller*. Observera att **`t1`** och **`t2`** är variabler av enkel datatyp.

****

2.13 In- och utparametrar

Nu har vi lärt oss en hel del om funktioner, med och utan returvärde, med en, flera eller inga parametrar, värde- och referensanrop osv. Ändå kan vi inte returnera *flera* värden från en funktion. Det beror på att alla funktioner i C++ returnerar endast ett eller inget värde. Men för att vara mer noggrant, borde vi lägga till *med* **`return`**-*satsen*. Begreppet *returvärde* används i programmeringsterminologin endast för värden som skickas med **`return`**-satsen via funktionsnamnet. I denna bemärkelse finns det inga funktioner med *flera* returvärden. Men funktionens gränssnitt mot omgivningen dvs mot andra funktioner är inte begränsad till funktionsnamnet. Även parameterlistan tillhör gränssnittet och kan användas för kommunikation med andra funktioner. Hittills har denna kommunikation varit *enkelriktad*: Våra parametrar importerade data bara in i funktionen. Frågan är: Kan man inte använda dem även för export av data ut ur funktionen? I så fall skulle vi kunna få tillbaka även *flera* värden från en funktion genom att använda *flera* parametrar. Detta är möjligt fast man kallar sådana data inte längre för returvärden då de inte skickas med **`return`**-satsen via funktionsnamnet, utan via parametrarna. De kallas för *utparametrar*. Hittills har vi använt bara *in*parametrar. I detta avsnitt ska vi lära känna *ut*parametrar. Verktyget som behövs för det är datatypen referens som behandlats tidigare (sid 62). Det enda som behövs för att känneteckna en parameter som *ut*parameter är nämligen att deklarera den i parameterlistan som *referens*.

I följande funktion finns det en inparameter **`b`** som tillför funktionen ett värde och fem utparametrar **`t`**, **`f`**, **`e`**, **`h`** och **`r`** vars värden exporteras ur funktionen. De kommer in i funktionen oinitierade, som referenser, beräknas där, men används sedan utanför funktionen i **`main()`**. I själva verket är utparametrarna endast andra namn (alias) till de aktuella parametrarna **`tio, fem, en, halv, ore`** (originalnamn) som är deklarerade, men inte initierade i **`main()`**. Deras initiering sker i funktionen med referenserna. Meningen är att använda *inparametrar* för input och *utparametrar* för output till och från funktionen, även om det är *flera* värden.

```
// Change.h
// Tar in växelbeloppet b och delar upp det i antalet t 10-
// kronor, f 5-kronor, e 1-kronor, h 50-öringar och resten r
// i öre. Endast b är en inparameter (enkel datatyp).
// t, f, e, h och r är utparametrar (referenstyper).

void change(double b, int& t, int& f, int& e, int& h, int& r)
{
   int total = b * 100.001;                        // växel som int
   t =    total / 1000;                            // 10-kronor
   f =   (total % 1000) / 500;                      // 5-kronor
   e =  ((total % 1000) % 500) / 100;              // 1-kronor
   h = (((total % 1000) % 500) % 100) / 50;        // 50-öringar
   r = (((total % 1000) % 500) % 100) % 50;        // rest i öre
}
```

Den reala bakgrunden till funktionen är följande problem: I en automat erbjuds vissa varor. Man väljer en vara och stoppar in en viss summa pengar, i regel mer än varan kostar. Sedan ska automaten ge tillbaka växelpengar vilket endast är möjligt med ett antal myntslag som är föreskrivna i automaten. Låt oss säga det är 10-, 5-, 1-kronor och 50-öringar. I så fall måste växelbeloppet omvandlas till detta mynt"system". Just denna beräkning utförs av **`void`**-funktionen **`change()`** ovan. Men hur genomförs omvandlingen med de uttryck för **`t`**, **`f`**, **`e`**, **`h`** och **`r`** som står i funktionen? Det är en liten algoritm som använder sig av två enkla heltalsoperationer.

Algoritmen för omvandling av ett belopp till olika myntslag

Då denna algoritm endast fungerar för heltal måste växelbeloppet **`b`** som är en **`double`** först konverteras till **`int`**, vilket görs i funktionens första sats. Vi utnyttjar den automatiska typkonverteringens tilldelningsregel (sid 43) för att låta beloppet i kronor och ören konverteras till ett rent örebelopp som lagras i den lokala **`int`**-variabeln **`total`**. Multiplikationen med **`100.001`** istället för **`100`** är ett litet trick som förhindrar oväntade och fula avrundningsfel vid konvertering till heltal. I fortsättningen står alltså det givna växelbeloppet i variabeln **`total`**.

1. För att få antalet 10-kronor divideras **`total`** med **`1000`** då 10-kronor är **`1000`** ören:

```
t = total / 1000;
```

Hur många gånger ryms **`1000`** – eller 10-kronor – i **`total`**? Det antalet tilldelas till **`t`**. Eller med andra ord: **`1000`** dras av från **`total`** så många gånger tills resten blivit mindre än **`total`**. Det antalet som tilldelas till **`t`** blir antalet 10-kronor. Divisionen ovan är inte vanlig division utan heltalsdivision då både **`total`** och **`1000`** är heltal. Dvs **`total`** divideras med **`1000`**, resultatet tas, resten ignoreras, t.ex. **`6975/1000`** ger **`6`**. Se körexemplet på nästa sida. Resten **`975`** ignoreras här, men används i fortsättningen.

2. För att få antalet 5-kronor divideras resten som blev kvar från punkt **1** med **`500`** då 5-kronor är **`500`** ören:

```
f = (total % 1000) / 500;
```

"Resten som blev kvar från punkt **1**" är just **`(total % 1000)`**. Här används en annan operator som är besläktad med heltalsdivision, nämligen modulooperatorn **`%`**. **`%`** har ingenting att göra med procenträkning utan ger *resten* vid heltalsdivision. T.ex. **`6975 % 1000`** ger **`975`**. Efter att ha dragit av alla 10-kronor från **`total`** divideras resten med **`500`** för att få reda på hur många 5-kronor som finns i **`total`**. T.ex. **`975/500`** ger **`1`**. Resultatet av denna division ges till **`f`**, resten ignoreras och används i fortsättningen.

I ytterligare tre steg skulle man kunna förklara de övriga formlerna för beräkning av **`e`**, **`h`** och **`r`**. Men nu har mönstret i algoritmen (förhoppningsvis) trätt fram: Man tar förra stegets formel, ersätter **`/`** med **`%`** och lägger till en heltalsdivision med den nya enhetens örebelopp. I det allra sista steget däremot måste **`%`** användas hela vägen då

man är ute efter allra sista resten i öre. För att testa algoritmen anropas funktionen **change()** i följande program:

```
// ChangeTest.cpp
// Efter inköp av en vara i en automat ska växeln ges till
// bakai form av ett antal föreskrivna myntslag:
// 10-kronor, 5-kronor, 1-kronor, 50-öringar (& rest i öre)
// main() läser in ett växelbelopp, skickar det till den ex-
// ternlagrade funktionen change() som omvandlar växeln till
// mynt. Sedan skrivs ut resultatet av omvandlingen här
#include <iostream>
using namespace std;
#include "Change.h"

int main()
{
   double amount;
   int tio, fem, en, halv, ore;              // Originalnamn
   cout << "\nAnge ett växelbelopp i kronor och ören:  ";
   cin >> amount;

   change(amount, tio, fem, en, halv, ore); // Funktionsanrop
                 // t  , f  , e , h   , r är deras aliasnamn
   cout << '\n' << amount << " kr =\t"      // i change()
        << tio  << " tio-kronor\n\t\t"
        << fem  << " fem-krona\n\t\t"
        << en   << " en-kronor \n\t\t"
        << halv << " femtio-öring\n\n"
        << "Det blir\t" << ore << " ören kvar\n";
}
```

Växelbeloppet läses in. Den externlagrade funktionen **change()** inkuleras och anropas varvid förutom **amount** de aktuella parametrarna **tio**, **fem**, **en**, **halv** och **ore** skickas med sina originalnamn. Dessa tas emot i **change()** av sina aliasnamn **t**, **f**, **e**, **h** och **r**, dvs referenserna till **tio**, **fem**, **en**, **halv** och **ore**. När beräkningen görs där med referenserna kan man komma åt resultaten i **main()** med originalnamnen då **tio** och **t** är två namn till en och samma minnescell. Samma sak är det med de övriga parametrarna. Ett körexempel visar att vi verkligen får tillbaka till **main()** de värden som beräknas i funktionen p.g.a. referensanrop som automatiskt tillämpas vid utprametrar av referenstyp:

```
Ange ett växelbelopp i kronor och ören:  69.75

69.75 kr =      6 tio-kronor
                1 fem-krona
                4 en-kronor
                1 femtio-öring

Det blir        25 ören kvar
```

2.14 Överlagring av funktioner

Överlagring av operatorer har vi nämnt tidigare. Då såg vi att t.ex. symbolen **+** betydde både additions- och konkateneringsoperatorn. Det var sammanhanget där symbolen användes, som avgjorde vilken betydelse den hade. Även operatorn **/** är överlagrad: En gång som symbol för heltalsdivision, en gång för vanlig division. På samma sätt kan funktioner vara överlagrade. Bibliotksfunktionerna **`MessageBox()`** och **`MessageBox.Show()`** med sina olika varianter är exempel på *överlagring av funktioner* (eng. *overloading*). I detta avsnitt ska vi skriva egna överlagrade funktioner.

> Överlagring av funktioner innebär *olika* funktioner med *samma* namn, men olika parameterlistor: antingen olika antal parametrar eller olika datatyper till parametrarna.
> *Signaturen* skiljer åt deras olika varianter.

Signaturen

Det som avgör om två funktioner är *olika* eller *identiska* är funktionens signatur. Signaturen består av följande delar:

- Funktionens namn
- Antal parametrar
- Parametrarnas datatyper

Signaturen är alltså en funktions igenkänningstecken. T.ex. har funktionen **`void search(int t[], int n, int s)`** som vi kommer att använda senare (sid 194), följande signatur:

```
search(int t[], int n, int s)
```

Denna signatur består av namnet **`search`**, antalet tre (parametrar) och datatyperna **`int[]`** till den första och **`int`** till de andra två parametrarna. OBS! Returtypen **`void`** ingår *inte* i signaturen. Funktioner med samma signatur anses vara *identiska*. Funktioner som skiljer sig på *något* av signaturelementen anses vara *olika*. Två eller flera funktioner i ett och samma program kan ha samma namn om deras parameterlistor är olika dvs om funktionerna antingen har olika *antal* parametrar eller lika antal, men olika *datatyper*. Då *överlagrar* de varandra. Ett program däremot med två funktioner som har samma signatur kan inte kompileras.

Överlagring är ett koncept inom programmering som används för att koda funktionaliteter som är besläktade med varandra, men ändå inte är exakt identiska. Verkligheten är full av överlagring. Ta följande exempel: Att bromsa en lastbil görs på ett annat sätt än att bromsa en båt. Det finns ingen anledning att hitta på ett annat namn för funktionaliteten ”att bromsa” hos olika typer av fordon. Tvärtom, det vore t.o.m. förvirrande att använda olika namn. Man vill ju helst slippa att tänka på de tekniska skillnaderna mellan olika typer av fordon när man pratar om bromsning. En och

samma funktionalitet är *realiserad* på olika sätt. Med andra ord, man gör ”samma sak”, fast ändå lite annorlunda. Programmering tar över detta koncept genom att välja ett och samma namn för olika funktioner. C++ biblioteket är fullspäckat med överlagrade funktioner. C++ kompilatorn skiljer åt överlagrade funktioner genom den annorlunda parameterlistan och skickar automatiskt rätt anrop till rätt funktion.

Följande program är ett exempel på överlagring av två egendefinierade funktioner som överlagrar varandra:

```
// power.h
// Överlagring av funktioner:
// Två funktioner ned samma namn men olika parameterlistor
// En beräknar potensen "bas upphöjd till int-exponent"
// Den andra   potensen "bas upphöjd till double-exponent"
#include <cmath>                        // Krävs för exp(), log()

int power(int bas, int exponent)        // Beräknar potensen med
{                                       // en heltalsexponent
   int resultat = 1;
   for (int i=1; i<=exponent; i++)      // Loopen bygger potensen
       resultat *= bas;                 // med upprepad multip-
   return resultat;                     // likation
}

double power(int bas, double exponent)  // Beräknar potensen
{                              // med en decimaltalsexponent med
   return exp(exponent*log(bas));       // en matematisk formel
}
```

C++ kompilatorn skiljer åt överlagrade funktioner genom den annorlunda parameter listan och skickar automatiskt rätt anrop till rätt funktion. I exemplet ovan har de två funktionerna **`power()`** samma namn, men olika datatyper till parametrarna.

Den ena funktionen har **`int`** som datatyp till parametrarna bas och exponent. Denna funktion beräknar potensen "bas upphöjd till exponent" när exponent är heltal, t.ex. 2 upphöjd till 3, dvs $2 \cdot 2 \cdot 2$, genom enkel upprepad multiplikation i en **`for`**-sats som gör samma sak som: **`resultat = bas * bas * bas`** om vi tillämpar exemplet 2 upphöjd till 3. Den andra beräknar **p**otensen när exponent är decimaltal, t.ex. 2 upphöjd till 3.5 genom att använda en avancerad matematisk formel då det är meningslöst att multiplicera 2 med sig själv 3.5 gånger. Man tillämpar två olika metoder för beräkning av potensen beroende på om exponenten är heltal eller decimaltal. Vilken datatyp basen har, är däremot irrelevant för val av metod. Självklart täcker den matematiska formeln även beräkningen av "bas upphöjd till heltal". Men varför göra det komplicerat när det går enklare? Den matematiska formeln kräver inkluderingen av biblioteksfilen **`cmath`** vilket man slipper när man bara multiplicerar upprepade gånger. Dessutom kan man minska risken för avrundningsfel när man använder en enklare beräkningsmetod för den enklare uppgiften. Därför är det motiverat att ställa båda funktioner till förfogande. Överlagring ger oss dessutom möjligheten att döpa

dem till samma namn då det handlar om två funktioner som är besläktade med varandra – båda gör potensiering – men ändå inte exakt identiska.

För att testa överlagring anropar vi båda funktionerna **`power()`** från **`main()`** i följande program:

```
// Overload.cpp
// Anropar den ena potensfunktionen med en int som exponent
// och den andra  (två gånger)  med en double som exponent
#include <iostream>
#include <iomanip>                    // Krävs för setprecision()
using namespace std;

#include "power.h"                    // Innehåller två funk-
                                      // tioner power()
int main()
{
 cout << fixed << setprecision(15) << "\n\t";
 cout << "2 upphöjd till 3   =  "<< power(2, 3)   << "\n\n\t"
      << "2 upphöjd till 3.0 =  "<< power(2, 3.0) << "\n\n\t"
      << "2 upphöjd till 3.5 = " << power(2, 3.5) << "\n";
}
```

Det första anropet **`power(2, 3)`** går automatiskt till den första potensfunktionen med en **`int`** som exponent då heltalskonstanten **`3`** tolkas som en **`int`** (sid 47). De två sista anropen går automatiskt till den andra potensfunktionen med en **`double`** som exponent då decimaltalskonstanterna **`3.0`** och **`3.5`** tolkas som **`double`** (sid 48).

En körning ger:

```
        2 upphöjd till 3   =  8

        2 upphöjd till 3.0 =  7.999999999999998

        2 upphöjd till 3.5 = 11.313708498984759
```

Man ser avrundningseffekten vi nämnde ovan: 2 upphöjd till 3 ger med vanlig upprepad multiplikation exakt 8, medan 2 upphöjd till 3.0 som beräknas med den matematiska formeln, leder till ett litet avrundningsfel p.g.a. datorns begränsade lagringsutrymme och beräkningsformelns komplexitet, närmare bestämt exponentialfunktionen **`exp()`** och logaritmusfunktionen **`log()`** som båda är definierade i bibilioteket **`cmath`**.

En speciell typ av överlagring kommer vi att lära känna senare, när vi tar upp *polymorfism* (sid 249).

Övningar till kapitel 2

2.1 Mata in programmet **MessageBoxB** (sid 26) i Visual Studio. Kompilera. Försök att tolka och åtgärda kompileringsfelet.

Mata in samma program i Borland C++. Kompilera och exekvera.

Hur ska man tolka de olika beteendena i de olika utvecklingsmiljöerna?

2.2 Mata in programmet **MessageBoxVS** (sid 28) i Visual Studio. Kompilera och exekvera.

Mata in samma program i Borland C++. Kompilera. Tolka kompileringsfelet.

2.3 Modifiera programmet **AssignRule** (sid 44) genom att tilldela variabeln **a** värdet 65 536. Förklara varför variabeln **b** får värdet 0. Vilken regel är orsaken till detta felaktiga resultat?

2.4 Ersätt i programmet **IntRule** (sid 46) **sizeof(a + b)** med

sizeof(a * b / 2.0)

Förklara resultat. Vilken regel har tillämpats här? Hur kommer det sig att koden kan kompileras, fast **a** och **b** inte är initierade?

2.5 Ersätt i programmet **PromotInt** (sid 48) datatypen **short** till variabeln **s** till **float**. Vilken regel har tillämpats här? Förklara varför enligt regeln det felaktiga resultatet försvinner?

2.6 Följande algoritm är formulerad i pseudokod och beräknar summan av de första n positiva heltalen $1 + 2 + ... + n$ som en rekursiv funktion:

```
Funktion sum(n)
   OM n = 1
      returnera 1
   ANNARS
      returnera n + sum(n-1)
```

a) Varför är algoritmen rekursiv?

b) I vilken ordning adderar algoritmen de pos. heltalen, fram- eller baklänges? Varför den gör så?

c) Implementera pseudokoden ovan som en rekursiv funktion i C++ och anropa den i ett program för $n = 10$, 100 och 1000, dvs beräkna summorna $1 + 2 + ... + 10$, $1 + 2 + ... + 100$ och $1 + 2 + ... + 1000$.

2.7 Skriv en iterativ variant av *Fibonacci problemet*. Ersätt den rekursiva funktionen **fib()** (sid 51) med en vanlig loop. Jämför båda lösningsvarianterna med varandra, inte minst med avseende på beräkningskomplexiteten.

Kapitel 3

Klasser

	Ämne	Sida	Program
3.1	Vad är objektorienterad programmering (OOP)?	80	
	- Paradigmskifte	80	
	- OOP:s tre hörnstenar	83	
	- Klassdiagram	84	`All_in_main`
3.2	Vägen till OOP	87	`All_in_main`
	- Modularisering på funktionsnivå	88	`Procedure`
	- Modularisering på klassnivå	90	
	- Vår första klass	90	`Circle`
	- Test av klass	92	`CircleTest`
3.3	Inkapsling	96	
	- Åtkomstmodifieraren private	96	
3.4	Konstruktor	98	
	- Objektorienterad initiering	105	`ObjInit`
	- Klassens konstruktor	98	`CircleConstr`
	- Default konstruktorn	101	`Encapsulation`
	- Flera konstruktorer	102	`Circles`
3.5	Accessmetoder	107	`Emp/Access`
3.6	Klass som egendefinierad datatyp	109	
	- Deklaration av en klass	110	`Anstalld/lon()`
	- Definition av ett objekt	113	`EmployeeTest`
	- Datatyptest med sizeof	114	
	- Åtkomst till objektets medlemmar	115	
3.7	Metoder i OOP	118	`TravelTime`
	- Objekt som parameter och returvärde	118	`Travel_Test`
Övningar till kapitel 3		123	

3.1 Vad är objektorienterad programmering (OOP)?

En given definition på programmering är problemlösning med hjälp av datorn. Om man då beskriver problemets lösning i form av en *algoritm* kan man kort säga:

Program = algoritm + data.

Denna definition ställdes upp av Niklaus Wirth på 60-talet och återspeglar den procedurala synen på programmering. Fokuset ligger på *algoritmen* dvs att inte bara hitta utan även *beskriva* tillvägagångssättet (proceduren) för att lösa ett problem. Sedan återstår bara att koda denna beskrivning. En annan definition som kom upp på 80-talet och återspeglar den objektorienterade synen på programmering är:

Program = Modell av verkligheten

Om man i Wirths formel *Program = algoritm + data* lägger vikten på data istället för på algoritmen och inte längre betraktar data som ett slags bihang till algoritmen utan som *objekt,* kommer man till *objektorienterad programmering*. Denna nya programmeringsfilosofi genomsyr C++ med alla sina fördefinierade biblioteksprogram som i allra högsta grad är objektorienterade.

Paradigmskifte

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 kom in i bilden. Allt detta var endast med procedural programmering inte längre möjligt. Ett s.k. *paradigmskifte* hade blivit nödvändigt, dvs en ändring av helhetssynen på programmering.

Objekt, klass, datamedlem och metod

Objektorienterad programmering syftar åt att efterlikna verkligheten. Man vill avbilda den reala världen – åtminstone den del som tillåter datorisering – och konstruera en modell av den i sina datorprogram för att kunna simulera verkligheten genom att testa modellen. För att undvika filosofiska diskussioner kan vi anta att den reala världen består kort sagt av *objekt*. Världen kring oss är full med sådana objekt: Människor, byggnader, bilar, tåg, flygplan, träd, möbler, böcker, butiker, skolor, bibliotek, kontor, anställda, kunder, varor, fakturor, order, bokningar, kurser osv. Objekten kan vara verkliga eller virtuella. Ett datorprogram försöker att beskriva dessa objekt.

Ett *objekt*, t.ex. en bil, har vissa egenskaper. Man kan t.o.m. säga att bilen är summan av alla sina egenskaper. Ett annat ord för egenskap är *attribut*. Summan av alla attribut utgör objektet. Bilen har som attribut: fabrikat, modell, färg, årsmodell, antal körda mil, antal hästkrafter, maximala hastigheten, antal och storlek på cylindrar i motorn osv. Alla dessa data utgör objektet bil och ger svar på frågan ”Vad är det för bil?”. Alla bilar har sådana attribut. Därför abstraherar man – dvs bortser från bilarnas olikheter – och samlar bilarnas gemensamma egenskaper (attribut) i något som man kallar för *klassen Bil*. När man programmerar, deklarerar man klassen *Bil* och skriver upp alla dessa bilattribut som klassens *datamedlemmar*.

Metoder

Men bilden vore ofullständig om vi nöjde oss med dessa intressanta, men statiska datamedlemmar. Vi vill också veta vad man kan *göra* med bilen. Ett objekt med alla sina attribut kan i regel även utföra vissa aktioner eller operationer. I den objektorienterade programmeringens terminologi kallas dessa aktioner för *metoder*. Typiska metoder för en bil är t.ex. att köra fram, att backa, att accelerera, att bromsa, att parkera, att byta olja osv. Den fullständiga definitionen på en bil vore alltså att ange *både* dess attribut *och* metoder. Bilfabrikanten måste förse bilen med alla dessa färdigheter för att kunna sälja den som en bil. Därför går man i bilfabriken efter en plan när man tillverkar bilen. Denna plan för konstruktion av bilen är *klassen Bil*. Konstruktörerna, mest ingenjörer, måste skapa denna plan, innan bilen kan byggas. När vi skriver ett program måste vi först formulera *klassen Bil* för att sedan kunna skapa objekt av den. Klassen skrivs bara en gång, medan objekt kan skapas enligt klassens beskrivning i obegränsat antal. I klassen måste vi ta upp alla attribut och metoder som är relevanta eller av någon anledning önskvärda för en bil.

En *metod* är en funktionalitet som definieras i en klass. Den talar om vad ett objekt av denna klass kan *göra*. Det finns två steg i hantering av metoder: Först definierar man dem dvs skapar man deras kod i en klass. Sedan *anropar* dvs aktiverar man dem i ett objekt av denna klass. Ofta är det första steget redan genomfört av andra, så vi behöver bara anropa en redan *fördefinierad* metod. I klassen *Bil* t.ex. är metoderna att köra fram, att backa, att accelerera, att bromsa osv. definierade i huvuden på bilkonstruktörerna och i deras konstruktionsritningar och dokumentationer. Sedan har man tillverkat massor med objekt av klassen Bil i fabriken och byggt in dessa metoder i alla bilar. Vi behöver bara anropa dem i den bil vi kör. Den bil vi kör är ett specifikt objekt av klassen Bil. Låt oss kalla det för **`minVolvo`**. Objektet **`minVolvo`** har ett antal attribut som t.ex. fabrikat, modell, färg, årsmodell osv., men också ett antal metoder, bl.a. metoden **`kör()`**. Parenteserna i metodens namn brukar man skriva för att karakterisera **`kör()`** som en *metod* och skilja den från klassens attribut. I C++ skriver man ett anrop av metoden **`kör()`** så här:

```
minVolvo.kör();
```

Observera att *före* punkten står ett objekt, inte klassen. Det är ju den specifika bil som jag använder just nu som ska köras. Först *efter* punkten står själva anropet av metoden **`kör()`**. Det här sättet att skriva kallas *punktnotation*. Metoder måste alltid

anropas med punktnotation, vilket har sin grund i att de endast är deklarerade i klasser, så att de endast existerar i objekt av en klass. Till skillnad från fristående *funktioner* kan metoder varken definieras utanför klasser eller anropas utanför objekt. I C++ finns både metoder och funktioner. Om vi bortser från bilexemplet kan det i andra sammanhang även förekomma en klass (istället för objekt) före punkten i anropet av en metod. I så fall är metoden definierad i klassen på ett speciellt sätt nämligen som en *statisk* metod, vilket tas upp senare när vi behandlar metoder i detalj.

En annan variant av metoden **`kör()`** kan anropas på fäljande sätt:

```
minVolvo.kör(40);
```

Det kan t.ex. betyda: Kör bilen med hastigheten **`40`** km/h. Värdet **`40`** kallas då en *parameter* som skickas till metoden när den anropas. I så fall måste även metoden **`kör()`** vara definierad så att den har beredskapen att ta emot denna parameter. Så det kan inte vara samma metod som anropades *utan* parameter. Det måste vara en annan variant av den, exakt talat en annan metod med samma namn. Konceptet kallas *överlagring av metoder* och innebär två eller flera metoder med samma namn, men olika parametrar.

Abstraktion

Det är avgörande att skilja mellan *objekt* och *klass*. Vi tar ett annnat exempel: Pepparkakor är objekt vars klass är pepparkaks*formen*. Klassen är alltså en slags mall, en förskrift för produktion av objekt: En enda pepparkaksform kan producera tusentals pepparkaksgubbar. 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 pepparkaksformen ingår det som är gemensamt hos alla pepparkaksgubbar. Man har, när man byggde formen, bortsett från oväsentliga skillnader och tagit hänsyn endast till det väsentliga, det gemensamma hos alla pepparkakor – samma abstraktionprocess som vi kunde observera hos bilar.

Att *bortse* från skillnader och att bibehålla det gemensamma hos olika verkliga objekt, kallas för *abstraktion*. *Abstrahera* betyder på latin: att ta bort, att dra av. Man tar bort allt som skiljer saker och ting av samma kategori eller typ och kommer på det viset till själva kategorin. Abstraktion leder till *begreppsbildning*, till *klassificering* eller *kategorisering* av den reala världen. Ett växande barn går igenom samma abstraktionsprocess, ser först sina föräldrar (objekt), abstraherar sedan via erfarenhet så småningom till begreppet *människa* (klassen) och inser att sina föräldrar är två konkreta exemplar av den abstrakta klassen människa. Så gör barnet med alla saker och ting omkring sig och lär sig vuxenvärldens begreppsapparat. Det abstrakta begreppet *penna* (klassen) t.ex. bildas efter att man sett hundratals verkliga pennor (objekt). Objektorienterad programmering återspeglar denna naturliga tankeprocess från det konkreta till det abstrakta, från objekt till klass. Därför kallas även en klass för en *abstrakt datatyp* i programmering.

OOP:s tre hörnstenar

Objektorienterad programmering har kommit till för att förverkliga programmeringens gamla önskedrömmar om *modularisering*, *återanvändning av kod* och *strukturering av program*. Allt för att kunna underhålla stora program, förnya och vidareutveckla dem, så att de fungerar över längre tid och snabbt kan anpassas till nyuppkomna situationer.

Objektorienterad programmering bygger på tre hörnstenar:

- Inkapsling
- Arv
- Polymorfism

Vi ska nu få en första inblick i dessa begrepp utan att skriva kod. För att förstå dem bättre behöver vi sedan i alla fall skriva kod och på så sätt få mer detaljerade kunskaper om objektorientering.

Inkapsling och klassens konstruktor

Att låta klassens datamedlemmar vara *privata* och inte åtkomliga från andra klasser kallas för *inkapsling*. Detta gör man bl.a. ur datasäkerhetssynpunkt – den första av tre hörnstenar i objektorienterad programmering. Ändå måste programmeraren kunna komma åt dem för att läsa, ändra och hantera dem i koden. För att kunna göra det måste programmeraren använda sig av ett verktyg som kallas för klassens *konstruktor*. Konstruktorn en speciell metod vars namn är identiskt med klassens namn. Den initierar automatiskt klassens privata datamedlemmar när ett objekt skapas. Konstruktorn är ett programmeringstekniskt koncept för att realisera inkapsling och kommer att behandlas i detalj senare.

Arv

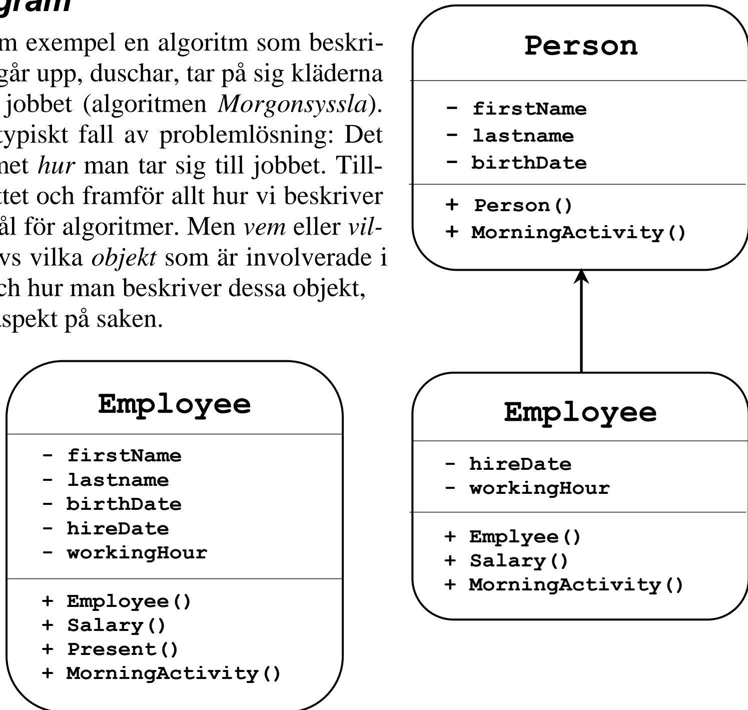
I den reala världen som vi vill efterlikna, finns inga isolerade objekt. Alla objekt är mer eller mindre relaterade till andra objekt. En klok modellering måste dra nytta av de befintliga relationer mellan objekt för att effektivisera och optimera utvecklingsarbetet. En sådan relation är arvrelationen.

Man kan alltid etablera en arvrelation mellan två begrepp om de står i en ”*är*”-relation till varandra. I exemplet ovan kan vi konstatera ett en anställd *är* en person. Därför kan klassen **`Employee`** ärva klassen **`Person`**, närmare bestämt ärver klassen **`Employee`** klassen **`Person`**:s alla datamedlemmar och metoder. Klassen **`Person`** kallas *bas-* eller *superklass*. Klassen **`Employee`** kallas *härledd* eller *subklass*. Subklassen ärver superklassens alla datamedlemmar och metoder, vilket i praktiken innebär att klassen **`Employee`** tar över all kod som redan finns i klassen **`Person`** och lägger till ny kod som närmare specificerar en anställd. På så sätt slipper man skriva om kod som redan finns. T.ex. har en person ett för- och efternamn samt ett födelsedatum. Vid modellering av en anställd ärvs dessa attribut, och man lägger till

de nya attributen hireDate och workingHour som är speciella för en anställd. Klassdiagrammet ovan (till vänster) visar modellen där arvrelationen ritats med en pil riktad mot superklassen. Följer man pilens riktning underifrån kan man avläsa att det är klassen **`Employee`** som ärver klassen **`Person`**.

Klassdiagram

Låt oss ta som exempel en algoritm som beskriver hur man går upp, duschar, tar på sig kläderna och åker till jobbet (algoritmen *Morgonsyssla*). Detta är ett typiskt fall av problemlösning: Det löser problemet *hur* man tar sig till jobbet. Tillvägagångssättet och framför allt hur vi beskriver det, är föremål för algoritmer. Men *vem* eller *vilka* gör det, dvs vilka *objekt* som är involverade i algoritmen och hur man beskriver dessa objekt, är en annan aspekt på saken.



Objektorienterad programmering prioriterar objektaspekten framför algoritmaspekten. Den primära frågan är inte längre vad man *gör* utan vem man *är* dvs *hur kan personen beskrivas?* Hur man gör för att ta sig till jobbet kommer att ingå som en del i denna beskrivning. Algoritmen Morgonsyssla blir en metod i *objektet* Person. Det är objektet som utför metodens instruktioner för att ta sig till jobbet.

Personen kan t.ex. vara en anställd vilket förresten skulle förklara varför han tar sig till jobbet. I så fall är personen ett objekt av kategorin eller klassen *Employee*. Därför definieras en klass som beskriver alla anställda. Personen i fråga görs till ett objekt, ett exemplar av denna klass.

På så sätt kan koden återanvändas även för andra anställda. Återanvändning av kod gör utvecklingsarbetet av programvara effektivare och är en av den objektorienterade synens fördelar. I klassen *Employee* ingår all typ av information som är relevant för en anställd, det vi kallar för attribut, t.ex. för- och efternamn, födelse- och anställningsdatum, arbetstid osv.

Dessutom tar vi upp allt som en anställd kan göra, det vi kallar för metoder, t.ex. att få lön, att presentera sig eller också att ta sig till jobbet. På så sätt blir algoritmen Morgonsyssla i den objektorienterade programmeringens terminologi en metod i klassen *Employee*. Ett verktyg speciellt för objektorienterade modelleringar är UML *(Unified Modeling Language)*. Enligt det här modelleringsspråket skulle klassen *Employee* beskrivas med diagrammet till höger som kallas för *klassdiagram*. Där står tecknet **−** för attribut och **+** för metoder. Andra beteckningar för attribut är *datamedlem* eller *egenskap*. Dessa termer är synonymer. En klass består av datamedlemmar och metoder. Klassen **`Employee`** t.ex. har fem datamedlemmar och tre metoder.

Observera att klassen **`Employee`** inte har två utan fem attribut därför att den via arvrelationen även har **`Person`**-klassens tre attribut. Samma gäller för metoderna: **`Employee`**-klassen ärver metoden **`Present()`** från klassen **`Person`**. Modellen ovan går utifrån att personer presenterar sig på samma sätt som anställda. Sedan har anställda en löneberäkningsmetod som icke-anställda personer saknar. Men varför står metoden **`MorningActivity()`** i båda klasser? Närmare bestämt: Varför förekommer den i **`Employee`**-klassen fast den ärver den från superklassen? Svaret ges av ett annat koncept inom objektorienterad programmering:

Polymorfism

Modellen ovan går utifrån att icke-anställda personer har en annan form av morgonsyssla än anställda. De kanske inte tar sig till jobbet, i alla fall inte alla, utan har en annan morgonsyssla. Så vi har här att göra med två olika morgonsysslor tillhörande två olika klasser, men med samma namn. För objekt av typ **`Person`** kommer den ena och för objekt av typ **`Employee`** kommer den andra att gälla. Men varför har de samma namn? Vore det inte bättre, för att undvika namnkonflikt, att ge dem olika namn, när de ändå är olika metoder? Faktiskt inte!

Anledningen till att de har samma namn är följande: För det första blir det ingen namnkonflikt därför att de tillhör olika typer av objekt. De är inte fristående utan inkapslade i var sitt objekt som skiljer åt dem. För det andra ska vi inte i onödan göra utvecklingsarbetet komplicerat genom att hitta på nya namn på metoder som skiljer sig från varandra endast i detaljer. Ingen människa skulle kunna komma ihåg så många namn. För det tredje vill vi efterlikna verkligheten där det bara kryllar av beteckningar som är identiska, men har olika innebörd i olika sammanhang. Inte heller det vanliga språket har olika namn på dem. Ta följande exempel: Att bromsa en lastbil görs på ett annat sätt än att bromsa en båt. Det finns ingen anledning att hitta på ett annat namn för funktionaliteten "att bromsa" hos olika typer av fordon. Tvärtom, det vore förvirrande att använda olika namn. Man vill ju helst slippa att tänka på de tekniska skillnaderna mellan olika typer av fordon när man pratar om bromsning. En och samma funktionalitet är realiserad på olika sätt. Med andra ord, man gör "samma sak", fast på annorlunda sätt. Objektorienterad programmering tar över detta koncept genom att välja ett och samma namn för olika metoder. När metoderna dessutom finns i klasser som ärver varandra kallas konceptet för *polymorfism*.

Polymorfism modifierar helt eller delvis funktionaliteten hos metoder med samma namn som förekommer i en arvhierarki.

”Poly” betyder *många* och ”morf” är *form* eller *gestalt* på latin och antik grekiska. Polymorfism handlar om en sak som har många olika gestalter, t.ex. ett ord som har många olika betydelser. En metod beskriver alltid någon funktionalitet. Polymorfism förändrar denna funktionalitet genom att definiera en metod i superklassen och definiera om innehållet, men behålla namnet i subklassen.

3.2 Vägen till objektorienterad programmering

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. *Funktion* är nyckelkonceptet inom procedural programmering. Motsvarande nyckelkoncept inom objektorienterad programmering är *klass*.

Utan modularisering

Vi börjar med ett program som varken är proceduralt eller objektorienterat. Sedan ska vi steg för steg modularisera det först till ett proceduralt och sedan till ett fullvärdigt objektorienterat program genom att gå igenom modulariseringsprocessen i två steg:

1. Modularisering på funktionsnivå (procedural)

2. Modularisering på klassnivå (objektorienterad)

```
// All_in_main.cpp
// Programmet är inte objektorienterat då det inte skapar
// några objekt utan endast lokala variabler
// Programmet är inte heller proceduralt (modulariserat)
// eftersom 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";
}
```

Detta är ett typiskt nybörjarprogram som har all sin kod i **`main()`**. Det är varken proceduralt eller objektorienterat. Inget fel på det, om målet endast är att räkna ut cirkelarean och omkretsen när man matar in radien. Men vi vill använda exemplet för att lära oss att modularisera programmet på olika nivåer. Det gör i två steg:

Steg 1: modularisering på funktionsnivå (procedural)

Beräkningarna flyttas från **`main()`** till separata *moduler* **`area()`** och **`omkrets()`** som skrivs som funktioner och lagras externt i headerfilen **`Procedure.h`**:

```
// 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;
}
```

Funktionerna **`area()`** och **`omkrets()`** definieras här och anropas i **`main()`**. Därmed består det nya programmet av tre moduler: **`main()`**, **`area()`** och **`omkrets()`**. En direkt konsekvens av denna modularisering är:

Parametrisering

För att de nya modulerna ska kunna kommunicera med **`main()`** måste vi förse dem med en parameter som överför sitt värde från **`main()`** till dem. Denna parameter döper vi till **`r`** som får sitt värde från **`main()`**:s lokala variabel **`radie`**. Parametriseringen är alltså priset man måste betala för modulariseringen. Vid funktionsanropen kopieras den aktuella parametern **`radie`**:s *värde* till den formella parametern **`r`**. Därför kallar man den här typen av parameteröverföring för *värdeanrop*.

Den tredje modulen **`main()`** placeras i följande separat fil. Headerfilen **`Procedure.h`** måste inkluderas i den för att koppla ihop modulerna:

```
// 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"                    // Externa moduler
                                          // 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 de externa *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 med att beräkna cirkelns area och omkrets.

I den procedurala programmeringen är *problemlösning* den styrande synen på hur man strukturerar koden. Man skriver lösningen i funktioner (procedurer). Därför är programmet **`Procedure`** proceduralt, men inte objektorienterat. Modulariseringen har genomförts på funktionsnivå och *modulerna* är funktioner. Först när ett program är modulariserat på klassnivå – där *modulerna* är klasser – kan det kallas objektorienterat.

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 **`radie`** är ”ur modellerings-synpunkt” relaterad till cirkeln. Samma sak gäller för **`area()`** och **`omkrets()`**. Ur problemlösningssynpunkt är det däremot ingen nackdel alls: Man har fått moduler som löser beräkningsproblemet.

Ser man på datorprogram som en modell av verkligheten – 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. 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. 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 i 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 är en del av modellen **`Circle`**, borde följa med. 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 inkluderas i programmet **`CircleTest`** (längre fram) implementerar denna idé.

Steg 2: modularisering på klassnivå (objektorienterad)

```
// Circle.h
// Klass som beskriver kategorin "Circle" som en abstrakt idé
// Modulariseringen är på klassnivå: "modul" = klassen Circle
// Innehåller datamedlem radie & metoderna area(), omkrets()

class Circle                                // Klassen 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 88) 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 programmering. 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. 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*. Vi definierar **`area()`** och **`omkrets()`** som metoder i klassen **`Circle`**.

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 för att skapa ett objekt av typ **`Circle`**, läsa in ett värde till det samt anropa klassens metoder **`area()`** och **`omkrets()`**:

```
// CircleTest.cpp
// Programmet är objektorienterat därför att det skapas ett
// objekt av klassen Circle
#include <iostream>
using namespace std;
#include "Circle.h"                           // Klassen Circle

int main()                                    // Funktionen main()
{
   Circle myCircle;                           // Objektet 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";
}
```

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

```
Circle myCircle;
```

Syntaxen påminner om deklarationen av en variabel. Vi kommer i nästa avsnitt att diskutera den här frågan och komma fram till att satsen ovan *är* en deklaration 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 115).

Programmet **`CircleTest`** 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
```

Klassbegreppet

Här sammanfattar vi våra observationer till *en* definition på *klass*. Läs på sid 109 en *annan* definition.

> En klass är kod som på ett generellt och modulärt sätt beskriver en kategori av verkliga eller virtuella saker och ting.
> Den består av *datamedlemmar* samt *metoder* och används som en mall för att skapa *objekt* av klassen.

Generell är en klass därför att den beskriver en kategori av saker och ting som är föremål för datorisering. Enligt klassens mall skapas sedan *objekt* av denna kategori. Medan klassen är ett abstrakt begrepp, en abstrakt idé, är objekten verkliga eller virtuella saker och ting i den reala världen.

Modulär är en klass därför att den kodas som en namngiven modul så att den kan användas av vilka andra program som helst. Programmen byggs med dessa moduler som minsta beståndsdelar som sedan kan användas för att konstruera andra program – liknande Lego-principen.

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 syften 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 *begreppsbildning*, 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.

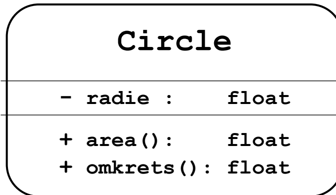
En synonym till objekt är *instans*. En instans av klassen **`Circle`** är ett exemplar av kategorin, av typen **`Circle`**.

Sammanfattningsvis kan vi säga:

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

I nästnästa avsnitt kommer vi att vidareutveckla idén om klassen som en ny sammansatt datatyp, som man kan definiera och gestalta helt självständigt.

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.

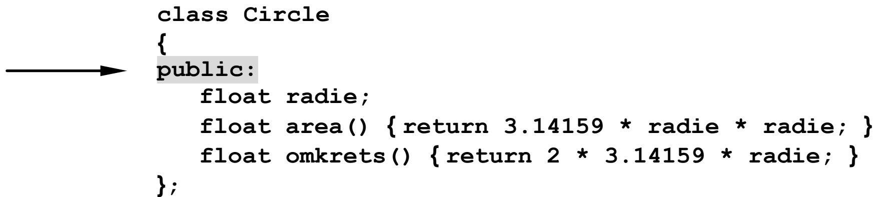
3.3 Inkapsling

Vi återvänder till frågan vi ställde i början av detta kapitel: Vad är objektorienterad programmering? Då besvarade vi den genom att säga att ett program är objektorienterat när det skapas ett objekt i det, eller: ”Först när ett program är modulariserat på klassnivå – där *modul* är en klass – kan det kallas objektorienterat.” (sid 90). Nu ska vi precisera svaret:

Som det nämndes tidigare bygger objektorienterad programmering på tre hörnstenar:

- **Inkapsling**
- **Arv**
- **Polymorfism**

Låt oss återvända till vår första klass (sid 90) och lägga märke på en kod som vi hittills inte gått in på närmare, nämligen det reserverade ordet **`public`**:



```
class Circle
{
public:
   float radie;
   float area() {return 3.14159 * radie * radie; }
   float omkrets() {return 2 * 3.14159 * radie; }
};
```

För att koncentrera oss på **`public`** har vi modifierat layouten genom att skriva metoderna **`area()`** och **`omkrets()`** i kompakt form, annars är det exakt samma klass som förr.

`public` är en *åtkomstmodifierare* som reglerar åtkomsten till klassens medlemmar, närmare bestämt åtkomsten *utifrån*, dvs från andra klasser eller funktioner.

Låt oss göra följande experiment för att förstå **`public`**:s innebörd: Kommentera bort raden med **`public`** och försök att kompilera klassen **`Circle`** genom att spara **`h`**-filen och kompilera **`cpp`**-filen **`CircleTest.cpp`** (sid 92). Resultatet är 4 kompileringsfel när vi försöker. Det första av dem är:

. . . *'radie' : cannot access private member declared in class 'Circle'*

Åtkomstmodifieraren `private`

Även **`private`** är en åtkomstmodifierare. Men vi har inte deklarerat **`radie`** som **`private`**, utan endast tagit bort **`public`**. Ändå säger felmeddelandet att **`radie`** är en *privat medlem* och att programmet inte kan komma åt den. Slutsats:

> I C++ är alla medlemmar i en **`class`** by default **`private`**.

Dvs, anger man inte explicit **`public`**, då blir det automatiskt **`private`**.

Åtkomstmodifieraren **`public`** gjorde att man utifrån klassen **`Circle`** kunde komma åt dess medlemmar, både datamedlemmar och metoder.

Åtkomstmodifieraren **`private`** spärrar åtkomsten till klassens medlemmar utifrån klassen. Räckvidden för en åtkomstmodifierare är därifrån den skrivs till slutet av klassdeklarationen eller tills en annan åtkomstmodifierare upphäver dess giltighet. På så sätt kan man deklarera enskilda, en del eller alla medlemmar som **`private`** eller **`public`**. Möjligheten att ha privata medlemmar samt den fördefinierade default-inställningen **`private`** för alla medlemmar hos **`class`**, är en teknik inom objektorienterad programmering som kallas *inkapsling*. Det man vill åstadkomma med denna teknik är igen att kunna efterlikna verkligheten i sina datorprogram så mycket som möjligt. I verkligheten är det självklart att vissa egenskaper hos objekt är eller ska vara "hemliga". T.ex. vem känner till en persons religion eller politiska inställning när man ser personen? Allt man kan se, är personens offentliga egenskaper, utseendet, hårfärgen, storleken, klädseln osv. Allt annat är okänt – så länge man inte ställer frågor. Och även då är det upp till personen att svara, inte svara eller svara delvis, tala sanning eller ljuga. Egenskaperna kan man jämföra med klassens datamedlemmar. Att "ställa frågor" kan man jämföra med att anropa klassens metoder. Man använder offentliga metoder för att via dem kunna efterfråga de privata datamedlemmarna.

> I objektorienterad programmering brukar man deklarera datamedlemmarna som **`private`** och metoderna som **`public`**.

Observera att detta inte är en regel utan snarare en *attityd* att jobba med klasser i alla objektorienterade språk. Det finns säkert i många specialfall skäl nog att använda inkapsling även på andra sätt. Men gör man det som beskrivet ovan, bildar datamedlemmarna klassens *kärna* som är skyddad mot direkta oönskade tillgrepp vare sig från andra program eller även andra programmerare. Metoderna däremot kan tänkas som ett skal kring kärnan som är till för att hantera klassens datamedlemmar. Man pratar om att metoderna bildar klassens *gränssnitt* mot användaren. Det är via dessa metoder man ska kunna kommunicera med den inkapslade kärnan. Därför måste gränssnittet vara offentligt. Självklart kan man tänka sig även olika grader av inkapsling. Inte alla datamedlemmar måste vara privata. Vissa applikationer kräver kanske mer, andra mindre inkapsling. Detta är av betydelse med tanke på att inkapsling alltid innebär en viss overhead dvs mer programmeringsarbete. På vilket sätt, kommer vi att se i de följande avsnitten.

3.4 Konstruktor

Ett problem som generellt uppstår när man arbetar med klasser med *privata* datamedlemmar är: Hur ska man initiera dessa datamedlemmar när de är oåtkomliga? Svaret ligger i det offentliga gränssnittet, dvs man utnyttjar publika metoder för att initiera klassens privata datamedlemmar. Initieringsproblematiken är redan viktig för enkla datatyper och därför ännu viktigare för mer komplexa objekt. För enkla datatyper hade vi infört konceptet av *väl definierade* variabler. För objekt har man i C++ konstruerat ett automatiskt verktyg som kallas *konsruktor*.

Klassens konstruktor

Vi börjar med den egendefinierade varianten av konstruktorn. Som namnet antyder är konstruktorn en byggare, närmare bestämt en objektbyggare, en av klassens metoder som automatiskt anropas när man skapar objekt av klassen. Dess viktigaste uppgift är att initiera objektets datamedlemmar. Det speciella som skiljer denna metod från klassens alla andra metoder kan beskrivas med konstruktorns följande tre egenskaper:

1. **Namnet** är inte fritt väljbart. Konstrukorn och klassen måste ha samma namn. Om man själv definierar konstruktorn har man inget val än att ge konstruktorn samma namn som klassen.
2. **Returtypen** saknas. Konstruktorns definition får inte börja som hos alla andra metoder med en returtyp. För det första *kan* en konstruktor inte returnera ett värde. För det andra *får* den inte ens ha returtypen **`void`** framför sitt namn som alla andra **`void`**-metoder.
3. **Anropet** av konstruktorn sker i samma sats som objektet skapas. För att initiera objektets datamedlemmar anropas konstruktorn samtidigt som objektet skapas. Man kan varken skapa ett objekt utan att anropa konstruktorn eller anropa konstruktorn utan att skapa ett objekt.

De två första egenskaperna måste beaktas när man definierar en konstruktor i klassen. Den tredje egenskapen måste tillämpas när man utanför klassen anropar konstruktorn och samtidigt skapar ett objekt. Med konstruktorn erbjuds en bekväm möjlighet att förhindra oinitierade datamedlemmar dvs allokera minne åt dem utan att tilldela dem värden. Därmed minskas risken för icke-väl definierade objekt.

```
// CircleConstr.h
// Deklarerar klassen Circle med en privat datamedlem radie
// och tre publika metoder: Circle(), area() och omkrets()
// Circle() är klassens konstruktor med parametern r

class Circle
{
   private: float radie;                // Privat datamedlem

public:
   Circle(float r)                     // Klassens konstruktorn
   {
      radie = r;
   }

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

Klassen **Circle** är en ny variant av den tidigare använda klassen **Circle** (sid 90). Den nya varianten har en egendefinierad konstruktor Med den vill vi testa egenskaperna **1-3** på förra sidan. Klassens konstruktor är framhävd med vit bakgrund. Som man ser är de två första ovannämnda egenskaperna givna: konstruktornamnet **Circle** = klassnamnet och ingen returtyp, inte ens **void**, vilket gör att både kompilatorn och vi kan känna igen **Circle()** som konstruktor och kan skilja den från klassens andra metoder **area()** och **omkrets()**. Varje försök att sätta en datatyp eller **void** framför metodnamnet kommer att leda till kompileringsfel.

Konstruktorn **Circle()** har en formell parameter **r** av typ **float**. Den gör i kroppen inget annat än att vidarebefordra parametervärdet till klassens datamedlem **radie** som by default är **private** medan konstuktorn **Circle()** och de andra metoderna **area()** och **omkrets()** är uttryckligen deklarerade som **public**.

Observera att både konstruktorn **Circle()** och metoderna **area()** och **omkrets()** refererar till datamedlemmen **radie** *utan* punktnotation. Orsaken är att de gör det i klassen där det inte kan råda någon tvivel om att vilken datamedlem som är menad. Alla involverade variabler och metoder är medlemmar i en och samma klass och kan referera till varandra utan punktnotation. Refererar man däremot till ett speciellt *objekts* medlemmar vare sig i eller utanför klassen måste punktnotation användas. Utan objekt är punktnotation meningslös.

```
// Encapsulation.cpp
// Skapar ett objekt av typ Circle och anropar konstruktorn
// med en parameter vars värde läses in
// Circle-objektets radie får detta värde via konstruktorn
#include <iostream>
using namespace std;
#include "CircleConstr.h" // Klassen Circle

int main()
{
   float input;
   cout << "Mata in cirkelns radie: ";
   cin >> input;

   Circle c(input);        // Ett objekt skapas och
                           // konstruktorn anropas som
                           // initierar radie till input

   //  c.radie = input;    // Kompileringsfel: radie är privat

 cout << "\nEn cirkel med radien " << input << " har arean "
      << c.area() << "\n\t\t   och omkretsen " << c.omkrets()
      << "\n";
}
```

Programmet ovan testar konstruktorns tredje egenskap genom att utanför klassen anropa konstruktorn och samtidigt skapa ett objekt. Satsen

```
Circle c(input);
```

gör två saker: Den första delen **`Circle c`** skapar som vanligt objektet **`c`** av typ **`Circle`** *och* den andra delen anropar konstruktorn **`Circle()`** med den aktuella parametern **`input`**. Båda delar – definition av objektvariabeln och anrop av konstruktorn – måste skrivas i *en* sats och kan inte separeras.

I sin struktur liknar satsen ovan det som vi en gång nämnde för enkla datatyper och kallade för objektorienterad initiering (sid 105):

```
int no1(5);
```

som gör exakt samma sak som: **`int no1 = 5;`**

Båda satser definierar **`no1`** som en variabel av typ **`int`** och initierar den samtidigt. Så gör vi också nu: Vi definierar objektet **`c`** av typ **`Circle`** och initierar den samtidigt. Jämförelsen visar än en gång kopplingen mellan objektorienterad och traditionell programmering.

När konstruktorn **`Circle()`** i satsen ovan anropas i **`main()`** importerar den den aktuella parametern **`input`** via sin formella parameter **`r`** in i objektet och tilldelar den till objektets datamedlem **`radie`**, se konstruktorns definition i klassen **`Circle`** (sid 99). På så sätt blir **`radie`** initierad fast den är **`private`**. Konstruktorn tillåter

alltså en *indirekt* initiering av den privata datamedlemmen. Varje försök att initiera den *direkt* – ja överhuvudtaget att referera till den med punktnotation – kommer att leda till kompileringsfel. Detta försök finns som kommentar i programmet **`Encapsulation`**. Testa gärna!

En körning ger följande utskrift:

```
Mata in cirkelns radie: 1.5

En cirkel med radien 1.5 har arean 7.06858
                     och omkretsen 9.42477
```

Default konstruktorn

Låt oss börja även här med ett litet experiment för att komma default konstruktorn på spåret. Låt oss i programmet **`Encapsulation`** ersätta den sats som skapar objektet och anropar konstruktorn:

```
Circle c(input); // Anrop av egen konstruktor
```

med:
```
Circle c;        // Anrop av default konstruktorn
```

Dvs vi försöker att skapa objektet utan att anropa vår egendefinierad konstruktor. Vi sa redan att detta leder till kompileringsfel, men intressant är felmeddelandet:

no default constructor exists for class 'Circle'

Felmeddelandet avslöjar att det finns något som heter *default constructor*, men att en sådan inte finns i vår klass **`Circle`**. Det beror på att vi har definierat en egen konstruktor i **`Circle`** och den slår ut klassens inbyggda default konstruktorn. Men kommenterar vi bort vår egen konstruktor i klassen **`Circle`** (sid 99), kan vi kompilera, men får skräpvärden när vi kör. Hur ser denna default konstruktor ut?

> En default konstruktor är en automatisk konstruktor utan parametrar.

Med ”automatisk” menar vi en sådan som vi inte skriver själva. I varje C++ klass finns det en inbyggd default konstruktor som ser ut så här:

```
Klassnamn()
{

}
```

Dvs den är tom. Definierar man ingen egen konstruktor i sin klass, är denna tomma default konstruktor alltid present, även om man inte ser den. Skriver man däremot sin egen konstruktor sätts default konstruktorn ur funktion. I klassen **`Circle`** har vi definierat en egen konstruktor. Därför: *no default constructor exists for class 'Circle'* när vi med satsen **`Circle c;`** försöker att skapa ett objekt.

Det som sker bakom kulisserna är att vi med denna sats samtidigt anropar default konstruktorn. Men kompilatorn hittar ingen default konstruktor eftersom vi har definierat en egen konstruktor (med parameter) som har satt default konstruktorn ur spel. Hade vi inte skrivit en egen konstruktor i klassen `Circle` hade det gått alldeles utmärkt att skapa objekt med `Circle c;`

Men det är något konstigt med anropet `Circle c;` Enligt konstens regler borde anropet av default konstruktorn se ut så här: `Circle c();` dvs *med* parenteserna som man alltid gör när man anropar en metod (eller funktion). Vid anropet skrivs parenteserna alltid med även om metoden (funktionen) inte har några parametrar. Här har vi att göra med ett äkta undantag i C++ som avviker från regulär syntax:

> Anropet av default konstruktorn skrivs utan parenteser.

Anledningen till detta undantag är att tillåta att skapa objekt i sådana sammanhang där parentesen är omöjligt att skriva, eftersom platsen efter objektnamnet är reserverat för andra saker.

Om man t.ex. har en klass `Tid` och vill skapa en array av `Tid`-objekt. skapas med `Tid arbetstid[5]` en array av 5 objekt av klassen `Tid`. Varje element i denna array är ett objekt. För att reservera minne åt dessa objekt anropas för varje element automatiskt en default konstruktor som kompilatorn ställer till förfogande. Om anropet behövde skrivas med parenteser skulle koden som skapar denna array se ut så här: `Tid arbetstid()[5]`, vilket ser väldigt konstigt ut. Det i sig vore inget problem, men kompilatorn godtar inte koden. Den kan inte tolka sekvensen av de runda och hakparenteserna, vilket är bara ett exempel som motiverar undantaget ovan.

Självklart kan man definiera i sina klasser även en egen default konstruktor som inte är tom. Det rekommenderas också att göra eftersom kompilatorns tomma default konstruktor är inte precis någon intelligent initiering av objekt. Datamedlemmarna blir ju inte ens initierade till `0` eller andra default-värden. De får skräpvärden precis som vanliga oinitierade variabler. Detta är inte precis en optimal lösning. Därför rekommenderas att definiera en egen konstruktor utan parametrar med korrekt initiering av datamedlemmarna, som i så fall kommer att sätta ur funktion kompilatorns default konstruktor, se exemplet nedan.

Flera konstruktorer

En klass kan ha flera konstruktorer. Ja, det är t.o.m. ganska vanligt med flera konstruktorer. Anledningen är att man vill ha möjligheten att initiera sina objekt på olika sätt i olika sammanhang. Man vill inte begränsa sig på endast ett sätt att konstruera objekt. Det som gör det möjligt att definiera flera konstruktorer i en klass, är det programmeringstekniska koncept som även används i C++ prorambibliotek:

Överlagring (eng. *Overloading*)

Överlagring av metoder innebär att *olika* metoder har samma namn, men olika parameterlistor, dvs olika antal parametrar eller olika datatyper till parametrarna.

Klassen **`Circles`** nedan har tre konstruktorer som överlagrar varandra, en med ingen parameter, en med en parameter och en med två parametrar:

```
// Circles.h
// Klass med tre olika konstruktorer som överlagrar varandra
// och har olika antal parametrar: ingen, 1 och 2.

class Circles
{
   float radie;
   string namn;

public:
   Circles()                   // Konstruktor utan parameter
   {
      radie = 1;
      namn  = "Enhetscirkeln";
   }

   Circles(float r)            // Konstruktor med 1 parameter
   {
      radie = r;
   }

   Circles(float r, string n)  // Konstruktor med 2 parametrar
   {
      radie = r;
      namn  = n;
   }

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

Flera konstruktorer är en av de viktigaste tillämpningarna av överlagring. Att ha samma namn följer direkt av nödvändigheten att ha samma namn på sina konstruktorer som klassens namn. Klassen **`Circles'`** konstruktorer är tre *olika* metoder som initierar klassens privata datamedlemmar på *olika sätt*.

Konstruktorn utan någon parameter bygger en s.k. *enhetscirkel* som definieras i geometrin med radien **`1`**. En konstruktor utan parameter behöver inte alltid nollställa datamedlemmarna. Man kan anpassa den till applikationen.

Den andra konstruktorn med en parameter initierar endast **`radie`**. **`namn`** förblir oinitierad. Den kan endast bygga namnlösa cirklar.

Den tredje konstruktorn med två parametrar är en allmän sådan och kan via sina parametrar initiera datamedlemmarna med de värden som skickas. Alla dessa initieringar görs förstås när man skapar objekt av klassen **`Circles`** med någon av dessa konstruktorer, vilket demonstreras i följande program:

```
// MoreConstr.cpp
// Skapar tre objekt av typ Circles med olika initieringar
// av datamedlemmarna genom att anropa olika konstruktorer
#include <iostream>
using namespace std;
#include "Circles.h"        // Klassen Circles

int main()
{
   float input;
   cout << "Mata radien till din cirkel: ";
   cin >> input;
                            // Tre objket skapas med:
   Circles noParam;         // anrop av konstruktorn utan par.
   Circles enParam(2);      //                     med 1 par.
   Circles tvaParam(input, "Min cirkel");    //     med 2 par.

   cout << "\nEnhetscirkeln med radien 1 har arean "
        << noParam.area() << "\n\t\t       och omkretsen "
        << noParam.omkrets() << "\n"

        << "\n      Cirkeln med radien 2 har arean "
        << enParam.area() << "\n\t\t       och omkretsen "
        << enParam.omkrets() << "\n"

        << "\n   Din cirkel med radien " << input
        << " har arean "    << tvaParam.area() << "\n\t\t   "
        << "och omkretsen " << tvaParam.omkrets() << "\n";
}
```

Observera att det första objektet **noParam** skapas genom att konstruktorn utan parameter anropas *utan parenteser*. Att det här objektets radie verkligen initieras till **1** kan man verifiera på areans och omkretsens värden som är beräknade med radien **1**, se körexempel på nästa sida. Det andra objektet **enParam**:s radie initieras till **2**. Det tredje objektet **tvaParam** får det inmatade värdet till **input** för radien och strängen **Min cirkel** för datamedlemmen **namn**. Alla dessa initieringar av de privata datamedlemmarna sker via de offentliga konstruktorerna. Även area och omkrets till de tre objekten skrivs ut genom anrop av de offentliga metoderna.

Resultatet blir:

```
Mata in cirkelns radie: 3

Enhetscirkeln med radien 1 har arean 3.14159
                       och omkretsen 6.28318

      Cirkeln med radien 2 har arean 12.5664
                       och omkretsen 12.5664

   Din cirkel med radien 3 har arean 28.2743
                       och omkretsen 18.8495
```

Nu ser man att det första objektet **`noParam`**:s radie verkligen initierats till **`1`**: areans och omkretsens värden är beräknade med **`1`**. Dvs satsen **`Circles noParam;`** som skapat objektet har samtidigt anropat den konstruktorn i objektet där radiens värde är hårdkodat till **`1`**. Beakta att anropet sker här utan parenteser.

Observera att konstruktorerna tillåter endast initiering, de skickar endast en första gång initialvärden till dem. Där slutar deras mission. Vad som händer efteråt har de ingen möjlighet att påverka. Det behövs andra offentliga metoder som tar hand om att *hämta ut* (exportera) privata datamedlemmar. Även om vi vill *ändra* privata datamedlemmarnas värden efter initieringen behöver vi speciella offentliga metoder för det. Med en exportmetod hade vi kunnat hämta ut värdena till **`radie`** och **`namn`** från **`Circles`**-objekten efter att ha initierat dem med konstruktorerna. Ett alternativ hade varit att flytta utskriftssatsen från **`main()`** till klassen där man direkt har åtkomst till privata datamedlemmar. Frågan är: Vad är rimligt att göra just i det här fallet och finns det ett generellt förfarande som man i regel borde använda? Hela den här problematiken diskuteras i nästa avsnitt.

Innan vi behandlar detta ska vi titta på ett objektorienterat alternativ för initiering av enkla datatyper, vilket påminner om det objektorienterade sättet att initiera klassens datamedlemmar med hjälp av konstruktorn. Dessutom demonstrerar detta släktskapet mellan procedural och objektorienterad programmering:

Objektorienterad initiering

Följande program visar ett nytt sätt att skapa initierade variabler av enkel datatyp:

```
// ObjInit.cpp
// Gör samma sak som programmet DefInitial, men initieringen
// görs på ett objektorienterat sätt: Variablerna deklareras
// och initieras samtidigt som om de vore objekt
#include <iostream>
using namespace std;

int main()
{
   int no1(5);              // Definition och initiering
   int no2(3);              // på objektorienterat sätt

   cout << "\n Summan av " << no1 << " och "
        << no2 << " är " << (no1+no2) << "\n\n";
}
```

Skillnaden till våra program hittills är att tilldelningsoperatorn **`=`** inte längre används. Istället har vi den lite kryptiska koden:

```
int no1(5);
```

som gör exakt samma sak som:

```
int no1 = 5;
```

Båda satser definierar variabeln **`no1`** och *initierar* den samtidigt till värdet **`5`**.

Avsaknaden av tilldelningsoperatorn **=** kan vara av fördel i satsen **`int no1(5);`** eftersom definition och initiering smälter ihop till *en* operation: Man skapar en variabel och initierar den samtidigt. Det blir en vana att aldrig definiera en oinitierad variabel. En annan fördel är att inget problem angående operatorprioritet kan uppstå när andra operatorer är inblandade. T.ex. skulle satsen

```
int no1(4+3);
```

skapa variabeln **`no1`** och initiera den till värdet **`7`** utan någon konkurrens om prioritet mellan tilldelnings- och additionsoperatorn, dvs vilken av dem som ska utföras först.

Detta sätt att koda används i objektorienterad programmering. ”Objektet” är i vårt fall variabeln **`no1`** som skapas som ett exemplar av den fördefinierade datatypen (”klassen”) * **`int`**. Samtidigt anropas en funktion – parenteserna **`()`** är symbolen för den. Vid anropet skickas initialvärdet **`5`** till variabeln.

Trots fördelarna med detta skriv- och tänkesätt kommer vi att fortsätta använda tilldelningsoperatorn för initieringen av variabler av *enkla* datatyper, men kommer att använda den nya terminologin när vi hanterar objekt. Programmet **`ObjInit`** producerar exakt samma utskrift som **`Variable`**.

---

* Detta är endast en liknelse, av vilken man inte får dra den felaktiga slutsatsen att **`int`** *är* en klass och **`tal1`** *är* ett objekt. Ändå finns det starka skäl till att fortsätta tänka i liknelsens banor: Alla enkla datatyper – **`int`** är ett sådant – är klasser i embryonalt tillstånd. Det nya objektorienterade programmeringsspråket C# som utvecklades år 2000 – ca. 20 år efter C++ – har transformerat alla enkla datatyper till klasser (*type system unification*).

3.5 Accessmetoder

Accessmetoder kan delas in i tre grupper: *get-metoder* för att hämta (läsa), *set-metoder* för att ändra (skriva) värden till privata datamedlemmar och *utskrifts- eller strängrepresentationsmetoder* för att kunna visa de privata datamedlemmarna i läsbar textform. Hos den sista gruppen handlar det om att skriva ut klassens data som en sträng, en slags strängrepresentation av klassens objekt. Alla dessa accessmetoder är direkta konsekvenser av inkapsling dvs att kunna ha privata datamedlemmar. Följande program visar exempel på alla tre typer av accessmetoder:

```
// Emp.h
// Deklarerar klassen Emp med tre privata datamedlemmar, en
// konstruktor, en get- och set-metod till datamedl. salary
// samt en metod som skriver ut ett Emp-objekt som en sträng

class Emp
{
   string name;
   int    empNo;
   float  salary;

public:
   Emp(string n, int no, float sal)        // Konstruktorn
   {
      name   = n;
      empNo  = no;
      salary = sal;
   }

   float getSalary()                       // get-metod
   {
      return salary;
   }

   void setSalary(float newSalary)         // set-metod
   {
      salary = newSalary;
   }

   void write()                            // Utskriftsmetod
   {
      cout << "Namn            " << name   << "\n\t\t      "
           << "Anställningsnr " << empNo  << "\n\t\t      "
           << "Lön            " << salary << "\n\n";
   }
};
```

Förfarandet som visas här kan generaliseras, ja t.o.m. automatiseras: Till varje privat datamedlem kan en get- och en set-metod definieras, medan en utskriftsmetod räcker för hela klassen. Om man sedan faktiskt utnyttjar alla dessa verktyg i varje program, måste avvägas från fall till fall. Get-metoder har ett returvärde med samma returtyp

som den privata datamedlemmen, inga parametrar och endast en **return**-sats, som returnerar den privata datamedlemmens värde. Alla get-metoder har detta utssende. Man kan t.o.m. standardisera namngivningen genom att döpa get-metoden till **getX()**, där **X** är den privata datamedlemmens namn som man inleder med en versal. Set-metoden däremot är en **void**-metod med en parameter som har samma datatyp som den privata datamedlemmen och innehåller endast en tilldelningssats som tilldelar parametern till den privata datamedlemmen. Namnet ska vara **setY()** där **Y** är den privata datamedlemmens versala initial. Utskriftsmetoden är av **void**-typ utan parametrar och skriver ut alla privata datamedlemmar på ett användarvänligt sätt. I klassen **Emp** som vi testar i följande program har vi definierat en get- och set-metod endast för den privata datamedlemmen **salary**:

```
// Access.cpp
// Använder klassen Emp för att skapa en anställd, ändra dess
// lön (som är privat) med get- och set-metoden samt skriva
// ut den gamla och nya lönen med utskriftsmetoden
#include <iostream>
using namespace std;

#include "Emp.h"                                // Klassen Emp

int main()
{
   Emp emp("Kalle Karlsson", 349, 22500);   // Skapar objekt

   cout << "Före  löneförhöjning: ";
   emp.write();

   float oldSalary = emp.getSalary();       // Hämtar salary

   emp.setSalary(oldSalary *1.25);          // Ändrar salary

   cout << "Efter löneförhöjning: ";
   emp.write();
}
```

För att få tag i den gamla lönen hämtas först den privata datamedlemmen **salary** med ett anrop av get-mtoden **getSalary()**. Sedan görs ändringen av **salary** via anrop av set-metoden **setsalary()** genom att skicka den gamla lönen höjd med 25% som parameter . Resultatet blir:

```
Före  löneförhöjning: Namn           Kalle Karlsson
                      Anställningsnr 349
                      Lön            22500

Efter löneförhöjning: Namn           Kalle Karlsson
                      Anställningsnr 349
                      Lön            28125
```

3.6 Klass som egendefinierad datatyp

På sid 93 ställdes upp en defnition för klassbegreppet. Här följer en annan:

> En klass är en ny, egendefinierad och sammansatt datatyp som skapas med det reserverade ordet **`class`**.

Kan ett begrepp ha flera definitioner? Ja, om de inte motsäger varandra och belyser olika aspekter av begreppet. Vilken som är relevant i en viss situation avgörs av sammanhanget begreppet används i. Det finns ingen begränsning på vilka, hur många eller vilka kategorier av saker och ting man kan involvera i sin klass, inkl. andra klasser. Allt beror på den konkreta miljön man vill modellera i sitt program.

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 **`%`**. Även explicit typkonvertering av en **`float`** till en **`int`** eller någon annan enkel datatyp, är en operation som är implementerad i datatypen. 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 **`CircleTest`** (sid 92) att **`myCircle`** skapas som en **`Circle`**, vilket är en deklaration av variabeln **`myCircle`**. Förutsättning är förstås att man deklaraerat 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 deklaration av variabeln **`no`** skapar inte ett objekt, därför att **`int`** inte är en klass. Vi har att göra med två olika typer av variabler: enkel datatyp (**`no`**) och klasstyp (**`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 array. 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 kan man deklarera 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 för namngivning samt konventionen att inleda det med en versal. Sedan kan vi använda namnet som datatyp i våra program – med alla de ”rättigheter” som de fördefinierade datatyperna har. Med koden ovan *skapas* den nya datatypen som i sin tur kan skapa objekt. P.g.a. denna speciella styrkan betecknas **`class`** ofta som *abstrakt datatyp* eller *datastruktur*.

Observera att hela klassdeklarationen efter den avslutande klammern avslutas med semikolon: vi har att göra med en deklarations*sats*. Till skillnad från funktionsdefinitioner ersätter den avslutande klammern inte semikolonet: Utelämnas semi-

kolon får man kompileringsfel. Med det semikolonet däremot som står inom runda parenteser (efter *metoder*) menas att det beror på om man skriver en deklaration eller en definition av en metod. Deklaration kräver semikolon, definition inte. I exemplet **`Circle`** (sid 90) hade vi valt att skriva metodernas definition. I nästa exempel kommer vi att deklarera en metod.

I filen **`Employee.h`** 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 andra typer. Man kan t.o.m. ha datamedlemmar som i sin tur är av egendefinierade typer dvs objekt. Sedan har den nya datatypen även en metod **`lon()`** som endast är deklarerad, endast med huvudet. Kroppen definieras separat. Datamedlemmarna och metoderna står inom klamrarna enligt den generella beskrivningen ovan med avslutande semikolon.

```
// Employee.h
// Deklarerar klassen Anstalld med 4 privata datamedlemmar
// och 4 publika metoder, bl.a. konstruktor, get- & set-metod
// Metoden lon() deklareras här och definieras i EmpLon.h

class Anstalld
{
   string namn, personnr;
   float timlon, antalTimmar;
public:                                   // Konstruktor:
   Anstalld(string n, string p, float t, float a)
   {
      namn = n;
      personnr = p;
      timlon = t;
      antalTimmar = a;
   }

   float getTimlon()                      // get-metod
   {
      return timlon;
   }

   void setTimlon(float newTimlon)        // set-metod
   {
       timlon = newTimlon;
   }

   double lon();                          // Här deklareras
};                                        // metoden lon()
```

Varför står i klassen **`Anstalld`** 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 annan headerfil. Detta har praktiska fördelar, när man utvecklar stora program och vill testkompilera sina moduler en i taget.

Återstår frågan: Hur hittar metoden **`lon()`** sitt huvud när kroppen är separerad? Svaret är: med hjälp av *räckviddsoperatorn* **`::`** och genom att vi i huvudprogrammet inkluderar båda headerfiler i rätt ordning, så att deklarationen kommer first och definitionen sedan (se nästa sida). Här följer nu metoden **`lon()`**:s definition:

```
// EmpLon.h
// Definierar metoden lon() som är deklarerad i klassen Em-
// ployee. Hämtar metodens namn från klassen med räckvidds-
// operatorn. Beräknar månadslönen baserad på timlönen och
// övertid. Integrerad beståndsdel av klassen Anstalld

double Anstalld::lon()                      // Här definieras
{                                           // samma metod lon()
                                            // som är deklarerad 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 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, skapar två objekt, ändrar den
// privata datamedlemmen timlon med get- och set-metod
// och skriver ut den gamla och nya lönen samt skillnaden
// Mäter storleken till datatypen Anstalld med sizeof
#include <iostream>
using namespace std;
#include "Employee.h"                       // Inkluderar hea-
#include "EmpLon.h"                         // derfilerna i rätt
                                            // ordning
int main()
{                                           // Definierar objekt:
   Anstalld anst("Anders Larsson", "590714-2493", 110, 190);

   Anstalld copy = anst;                    // Kopierar objekt

   float old_Timlön = anst.getTimlon();     // Hämtar timlön

   copy.setTimlon(old_Timlön * 1.15);       // Ändrar timlön

   cout << "\n\t" << "Anders Larsson" << ", personnr "
        << "590714-2493"
        << "\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, därför \natt den"
        << " sammansätter datatyperna: string med "
        << sizeof(string) << " bytes\n\t\t\t\t"
        << "  float  med  " << sizeof(float) << " bytes\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 111. 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. 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;`**

Tidigare 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. 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`**. Prova gärna genom att bortkommentera datamedlemmarnas initiering.

Datatyptest med `sizeof`

När man pratar om att skapa ett *objekt* av *klassen* **`Anstalld`**, då är det samma sak som att skapa en variabel av *datatypen* **`Anstalld`**. I båda fall måste *minnesutrymme reserveras* av den storlek som klassen resp. datatypen föreskriver. Detta för att kunna lagra *värden* i objektet, för det är det enda som är praktiskt relevant: en variabels eller ett objekts *existens* i programmet är identisk med motsvarande minnesutrymmes existens 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 111). Där finns två datamedlemmar av typ **`string`**: **`namn`** och **`personnr`** som var och en tar **`40`** bytes, samt två **`float`**:s med **`4`** var, så att sammanlagt 2 x 44 = **`88`** bytes allokeras åt objeket **`anst`**. Denna minnesstorlek mäter vi med operatorn **`sizeof`** som returnerar antalet bytes operanden tar i minnesutrymme. Med **`sizeof`** kan vi visa att klasser är egendefinierade datatyper genom att skriva dem som operander i **`sizeof`**. I programmet **`EmployeeTest`** gör vi det med:

```
sizeof(Anstalld)
```

Lika bra skulle vi kunna skriva:

```
sizeof(anst)
```

dvs sätta in objektet som operand. I båda fall får vi det förväntade värdet **88**, vilket även visas i körexemplet.

```
        Anders Larsson, personnr 590714-2493

        Gammal lön:     21450

        Ny lön:         24667.5

        Våra lönekostnader kommer att öka med 3217.5 kr

Klassen Anstalld föreskriver 88 bytes för sina objekt, därför
att den sammansätter datatyperna:  string med 40 bytes
                                   float  med  4 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 **88** 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";
```

[*] 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.

```
anst.personnr    = "590714-2493";
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 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 vi behandlat tidigare.

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** 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 112) 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.

Eftersom **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.

3.7 Metoder i OOP

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 exempel på: **`_getch()`** definierad i **`conio`**, **`setprecision()`** i **`iomanip`** osv. Det gemensamma hos alla dessa metoder var att de hade endast parametrar och returvärden av *enkla datatyper*. För att studera metodernas objektorienterade egenskaper ska vi nu ta upp ett exempel där en metod både tar in ett *objekt* som parameter och returnerar ett *objekt*.

Objekt som parameter och returvärde

```
// 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)      // Metod med objekt som parameter
   {                         //       och objekt som returvärde
      Restid temp;
      temp.min = (min + t.min) % 60;
      temp.tim = (tim + t.tim) + (min + t.min)/60;
      return temp;
   }
};
```

Klassen Restid modellerar tiden, närmare bestämt restiden, där det kan vara relevant att summera sina restider under en viss period, t.ex. för en handelsresande. 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 en parameter **`t`** och ett returvärde **`temp`** som båda är objekt.
2. Dessa objekt är av typ **`Restid`**, dvs samma nya datatyp som vi håller på att definiera.

Punkt **1** följer av egenskapen att objekt kan skrivas överallt i programmet där även en vanlig variabel kan stå (sid 114). 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 med t.ex. satsen **`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`**. 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:

```
// Travel_Test.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";
}
```

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 118) 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. 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 **Travel_Test** 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 **tvadagsresan**s 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 **Travel_Test** 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 **`Travel_Test`** 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 **`Travel_Test`** 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.
```

Övningar till kapitel 3

Besvara följande frågor om objektorienterad programmering (3.1):

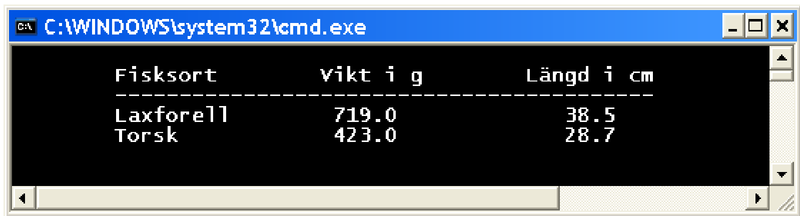
- Vad menas med *paradigmskifte* i programmeringens historia?
- Mellan vilka två programmeringsspråk går historiskt skiljelinjen mellan procedural och objektorienterad programmering? När ungefär inträffade övergången?
- Vad var anledningen till paradigmskiftet inom programutveckling?
- Vilka för- och nackdelar har enligt din åsikt den *procedurala* synen på programmering?
- Vilka för- och nackdelar har enligt din åsikt den *objektorienterade* synen på programmering?
- Är det korrekt att pepparkakor är *klasser* och pepparkaksformen *objekt*?
- Kan man via *abstraktion* komma från objekt till klass eller är det tvärtom?
- Om pennor är objekt var kan man hitta klassen *penna*?
- Av vilka två huvudingredienser består en klass i regel?
- Anta att *Tal* är en klass. Är *addition()* en metod eller en datamedlem i klassen *Tal*?
- Anta att *Bil* är en klass. Är *Motor* en metod eller en datamedlem i klassen *Bil*?
- Vad är skillnaden mellan *funktioner* och *metoder* i C++?
- Vilka är den objektorienterade programmeringens tre hörnstenar?
- Vad innebär modularisering på klassnivå?

3.1 Skriv en klass **`Rektangel`** med datamedlemmarna **`bredd`**, **`höjd`** och metoderna **`area()`**, **`omkrets()`**. Deklarera datamedlemmarna och metoderna som **`public`**. Testa din klass i en separat fil genom att i **`main()`** skapa ett **`Rektangel`**-objekt vars datamedlemmar initieras till konstanta värden i **`main()`**. Skriv ut objektets area och omkrets.

3.2 Modifiera övn 3.1 genom att *läsa in* värden till datamedlemmarna. Efter utskriften av area och omkrets, fördubbla rektangelns längd och bredd. Skriv ut en gång till rektangelns area och omkrets. Med vilken faktor växer arean resp. omkretsen?

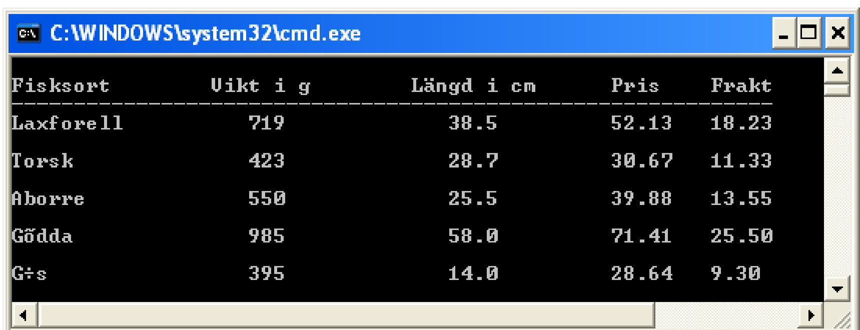
3.3 Skriv en klass **Fish** som beskriver en fisk med datamedlemmarna **fisksort**, **vikt** och **längd** och lagra den i en headerfil, t.ex. **Fish.h**

Testa din klass i en fil **FishTest.cpp** som innehåller **main()**. Skapa i **main()** två objekt av klassen **Fish**. Tilldela det första objektets datamedlemmar värdena *Laxforell*, 719 (gram) och 38,5 (cm). Enheterna gram och cm behöver inte anges. Välj själv andra värden till det andra objektets datamedlemmar. Skriv ut dessa värden till konsolen i en tabell av typ:



Fisksort	Vikt i g	Längd i cm
Laxforell	719.0	38.5
Torsk	423.0	28.7

3.4 Vidareutveckla klassen **Fish** från övn 3.3. Förse klassen med en metod **pris()** som beräknar priset beroende på fiskens vikt, säg 7,25 kr per hekto. Lägg till även en metod **frakt()** som beräknar frakten utifrån fiskens vikt och längd, t.ex. så här: Multiplicera en viss kostnadsfaktor, säg 0,02, med vikten, en annan, säg 0,1, med längden och addera dem. Dessa metoder ska returnera priset och frakten i hela kronor utan ören. Testa klassen **Fish** i **main()** genom att skapa fem **Fish**-objekt vars datamedlemmar samt värden du kan ta från tabellen nedan. Läs in värdena. Pris och frakt till varje **Fish**-objekt ska sedan beräknas genom anrop av metoderna **pris()** och **frakt()**. Lägg till två nya kolumner med resp. rubriker *Pris* och *Frakt* i tabellen ovan och skriv ut deras värden till en ny tabell som kan se ut så här:



Fisksort	Vikt i g	Längd i cm	Pris	Frakt
Laxforell	719	38.5	52.13	18.23
Torsk	423	28.7	30.67	11.33
Aborre	550	25.5	39.88	13.55
Gõdda	985	58.0	71.41	25.50
G÷s	395	14.0	28.64	9.30

3.5 Vidareutveckla klassen **`Fish`** från övn 3.4 (förrförra lektion): Deklarera alla datamedlemmar som **`private`** och alla metoder som **`public`**. Förse klassen med en konstruktor för att initiera de privata datamedlemmarna när du skapar objekt. Annars ska programmet göra samma sak som i övn 3.4.

3.6 Vidareutveckla klassen **`Fish`** från övn 3.5 (ovan): Förse klassen med get- och set-metoder för de privata datamedlemmarna **`vikt`** och **`längd`**. Ändra i huvudprogrammet vikt- och längdvärdena för *Laxforell* till *815* (gram) och *42* (cm). Programmet ska göra samma sak som övn 3.5, bara att den nya tabellen ska skriva ut de nya värdena för Laxforell samt ange de ökade pris- och fraktkostnaderna för Laxforell efter ändringen.

3.7 Vidareutveckla programmet **`TravelTest`** (sid 120) genom att ersätta de tre **`Restid`**-objekten **`tisdag`**, **`onsdag`** och **`torsdag`** med en *array av objekt* med 3 element:

```
Restid tredagar[3];
```

Använd samma klass **`Restid`** (sid 118) för att låta programmet göra samma sak som **`TravelTest`**.

3.8 Använd det du lärt dig om *array av objekt* i övn 3.7 genom att skapa **`Restid`**-arrayen **`vecka`** med 7 element:

```
Restid vecka[7];
```

Mata in timmar och minuter till varje veckodags restid. Använd klassen **`Restid`** (sid 118) och en **`for`**-loop för att summera veckodagarnas restider och skriva ut veckans totala restid.

Kapitel 4

Logik för blivande programmerare

	Ämne	Sida	Program
4.1	Logiska operatorer	127	**AND_OR**
	- Sanningstabeller	129	**Cross**
	- Visualisering av logiska operatorer	131	**NegativeCross**
4.2	Datatypen **bool**	135	**TruthTab**
	- Automatisk typkonvertering till **bool**	136	
4.3	NEGATION som logisk operator	137	**GuessNEG**
	- Logiska uttryck	138	
4.4	Testa lösenord med logiska lagar	139	**Passwd**
	- Caps Lock-problematiken	140	
	- De Morgans lagar	142	**PasswdCaps**
Övningar till kapitel 4		144	

4.1 Logiska operatorer

Begreppet villkor har följt oss från bokens första kapitel: Tidigare såg vi skillnaden mellan instruktion och villkor i algoritmen Morgonsyssla. En instruktion (sats) utförs medan ett villkor testas för att ta ett beslut och träffa ett val mellan två eller flera alternativ. Sådana villkor förekom i kontrollstrukturerna **`if`**, **`if-else`**, **`while`**, **`for`** och **`do`**. Alla villkor vi använt hittills i våra exempel har varit *enkla*. Vad innebär detta och vad är skillnaden mellan enkla och *sammansatta villkor*?

Enkla villkor

Ett villkor heter *enkelt* om dess sanningsvärde – sant eller falskt – kan bestämmas direkt, utan att blanda in andra villkor eller logiska operatorer (se nästa sida). Exempel på enkla villkor är:

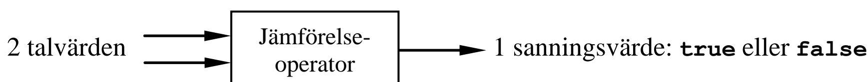
```
no == 0
no != 0
7 > 5
guessedNo <= 17
```

Enkla villkor kan bildas bl.a. med operatorer som jämför värden och därför kallas:

Jämförelseoperatorer

<	mindre än
<=	mindre än eller lika med
>	större än
>=	större än eller lika med
==	lika med
!=	icke lika med

De jämför två talvärden med varandra genom att ta in dem och returnera jämförelsens resultat sant eller falskt, ett s.k. *sanningsvärde*:



Sanningsvärdena **`true`** och **`false`** är de enda värden som villkor kan anta. Observera att de jämförelseoperatorer som är dubbeltecken, inte får innehålla mellanslag. T.ex. är == symbolen för *lika med*. Tidigare pratade vi om skillnaden mellan likhet och tilldelning och poängterade att = i C++ inte betyder likhet utan tilldelning. Här har vi symbolen == för *likhet*. Medan tilldelningsoperatorn = förekommer i instruktioner (satser) används jämförelseoperatorn == i villkor.

Sammansatta villkor

Ett *sammansatt villkor* är en kombination av enkla villkor. Men hur ska man sätta ihop två enkla villkor? Det kan endast göras om det finns något som binder samman dem. Detta ”något” kallas för en *logisk operator*. Exempel på *logiska operatorer* är

det logiska ELLER som kodas med dubbeltecknet **||** och det logiska OCH som symboliseras av **&&**. De opererar på två enkla villkor, ”räknar” på ett visst sätt med dem och returnerar ett sanningsvärde. Med logiska operatorer kan man bilda sammansatta villkor. Nedan följer några enkla exempel på sammansatta villkor:

```
(no == 0)          || (no > 0)
(temp <= 10)       && (temp >= 25)
(guessedNo < 17) || (guessedNo > 17)
```

Nu ser vi att sammansatta villkor är en kombination av enkla villkor, logiska operatorer och parenteser. Att de returnerar ett sanningsvärde beror på att de logiska operatorerna gör *ett* sammansatt villkor av två enkla villkor. Sammansatta villkor kan används för att lösa ett problem med trevägsval. Ett val mellan tre alternativ behöver inte lösas med **switch**-satsen, inte heller med nästlad **if-else**. Här följer ett exemel:

```
// AND_OR.cpp
// Läser in klockslag och avgör om det är dags för lunch
// Enkla villkor sätts ihop till:
// Sammansatta villkor med de logiska operatorerna && (OCH),
//                                                  || (ELLER)
#include <iostream>
using namespace std;

int main()
{
  int clock;
  cout<<"\n\tLäs av klockan och ange klockslag (utan min): ";
  cin >> clock;
  if ((clock >= 11) && (clock <= 14))
     cout << "\n\tDagens Lunch serveras.\n\n";
  if ((clock < 11) || (clock > 14))
  {
     cout<<"\n\tDagens Lunch serveras inte eftersom det är";
     if (clock < 11)
        cout << " för tidigt.\n";
     else
        cout << " för sent.\n";
  }
    cout<<"\n\tDagens Lunch serveras mellan kl 11 och 14.\n";
}
```

Körs programmet för de tre alternativen *före kl 11*, *mellan 11-14* och *efter kl 14* får man de här tre olika utskrifterna:

```
    Läs av klockan och ange klockslag (utan min): 10

    Dagens lunch serveras inte eftersom det är för tidigt.

    Dagens lunch serveras mellan kl 11 och 14.
```

```
Läs av klockan och ange klockslag (utan min): 12
Dagens lunch kan serveras.
Dagens lunch serveras mellan kl 11 och 14.
```

```
Läs av klockan och ange klockslag (utan min): 14
Dagens lunch serveras inte eftersom det är för sent.
Dagens lunch serveras mellan kl 11 och 14.
```

Sanningstabeller

Varje logisk operator definieras med en s.k. *sanningstabell* som bestämmer de sanningsvärden som gäller för just denna operator – jämförbart med de vanliga räkneoperatorerna, t.ex. multiplikationen som definieras med multiplikationstabellen. Innan vi visar tabellerna ska vi diskutera den logiska innebörden hos operatorerna.

Den logiska operatorn OCH

Det första sammansatta villkoret i programmet **AND_OR** är:

```
(clock >= 11) && (clock <= 14)
```

Den logiska operatorn **&&** kombinerar de två enkla delvillkoren **clock >= 11** och **clock <= 14** till ett sammansatt villkor vars sanningsvärde beror på de båda enkla delvillkorens sanningsvärden samt den logiska innebörden av operatorn **&&**. Det hela bildar villkoret i en **if**-sats i programmet **AND_OR**. Parenteserna kring de två enkla delvillkoren kan utelämnas här då de i alla fall evalueras först. Vi har skrivit dem för att vara på den säkra sidan vad gäller prioriteten mellan operatorerna. Den intuitiva innebörden av det logiska OCH i vanligt språk är: Om **clock**:s värde är större än eller lika med **11** *och* samtidigt mindre än eller lika med **14**, så är det sammansatta villkoret sant. Dvs om **clock**:s värde ligger mellan **11** och **14**, är villkoret sant. Det sammansatta villkoret beskriver alltså i det här fallet ett intervall. För att testa om ett värde ligger i ett intervall är ett villkor av sammansatt typ med operatorn **&&** en lämplig konstruktion. I **AND_OR** ska ”Dagens Lunch” serveras mellan klockan 11 och 14. Före kl 11 eller efter kl 14 ska ingen ”Dagens Lunch” serveras. Dvs om bara *ett* enkelt delvillkor **clock >= 11** eller **clock <= 14** är falskt blir också hela det sammansatta villkoret falskt. För att det sammansatta villkoret ska bli sant måste *båda* delvillkoren vara sanna. Dvs klockan måste vara över (eller prick) 11 och samtidigt före (eller prick) 14.

Logiken hos operatorn **&&** kunde i exemplet ovan härledas från det vanliga språkets betydelse för ordet OCH. Men hur avgör datorn som inte förstår vanligt språk,

sanningsvärdet hos ett villkor av sammansatt typ med den logiska operatorn **&&** ? Hur är denna operator definierad? En sådan allmän definition av operatorn **&&** är lagrad i datorn för att kunna bestämma sanningsvärdet till alla villkor som involverar **&&** vilket förstås gäller för alla logiska operatorer. Precis som det finns definitioner för de aritmetiska operatorerna **+**, **–** , ***** och **/** , som datorn använder för att beräkna aritmetiska uttryck, finns även definitioner för de logiska operatorerna, som datorn använder för att evaluera sammansatta villkor. Att *evaluera* ett villkor betyder att bestämma dess sanningsvärde. Den exakta logiska innebörden av operatorn OCH definieras med följande sanningstabell:

OCH:s sanningstabell

```
  p       q             p && q
--------------------------------
true    true            true
true    false           false
false   true            false
false   false           false
```

I sanningstabellen ovan symboliserar **p** *ett enkelt* delvillkor, t.ex. **hour >= 11** och **q** det *andra enkla* delvillkoret, t.ex. **hour < 14**. Då blir **p && q** det sammansatta villkoret, sammansatt av de två enkla delvillkoren med hjälp av operatorn **&&**. Tabellen ska läsas radvis. Första raden (under strecket) säger: Om båda de enkla delvillkoren **p** och **q** har sanningsvärdet **true**, får det sammansatta villkoret **p && q** sanningsvärdet **true**. Den andra raden säger: Om delvillkor **p** har sanningsvärdet **true** och delvillkor **q** sanningsvärdet **false**, får det sammansatta villkoret **p && q** sanningsvärdet **false** osv. Sanningstabellen behandlar alla möjliga kombinationer av värdena **true** och **false** för de enkla delvillkoren **p** och **q**. Det finns sammanlagt fyra sådana kombinationer som är uppställda i tabellens två första kolumner. Resultaten – sanningsvärdena för **p && q** – står i den tredje kolumnen. I och med att tabellen innehåller *alla möjliga* kombinationer, definieras den logiska operatorn **&&** generellt och återspeglar också den intuitiva innebörden av det logiska OCH i vanligt språkbruk, nämligen: Om - och endast om - de *båda* enkla delvillkoren **p** och **q** är sanna, är det sammansatta villkoret **p && q** sant, annars är det sammansatta villkoret falskt.

Liknande gäller för sanningstabellen till den andra logiska operatorn som förekommer i programmet **AND_OR**.

Den logiska operatorn ELLER

Det andra sammansatta villkoret i programmet **OCH_ELLER** är:

```
(hour < 11) || (hour >= 14)
```

Villkoret är sammansatt av de två enkla delvillkoren **hour < 11** och **hour >= 14** med hjälp av den logiska operatorn **||**. Den intuitiva innebörden av det logiska ELLER i vanligt språk är: Om **hour**:s värde är mindre än **11** *eller* större än eller lika med **14**, är det sammansatta villkoret sant, vilket i **AND_OR** innebär att ”Dagens lunch” inte ska serveras före 11 eller efter (eller prick) 14. Det räcker att endast *ett* av delvillkoren antingen **hour < 11** eller **hour >= 14** är sant för att det sammansatta villkoret ska bli sant. Endast om båda är falska, blir resultatet falskt. Man inser att klockan inte samtidigt kan vara före 11 och efter (eller prick) 14, därför används här ELLER och inte OCH. Även här kan logiken hos operatorn **||** härledas från det vanliga språkets betydelse för ordet ELLER, närmare bestämt för ANTINGEN ELLER. Men den exakta logiska innebörden definieras som vanligt av sanningstabellen:

ELLER:s sanningstabell

```
  p       q              p || q
-------------------------------
true    true             true
true    false            true
false   true             true
false   false            false
```

I sanningstabellen ovan står **p** för *ett enkelt* delvillkor, t.ex. **hour < 11** och **q** för det *andra enkla* delvillkoret, t.ex. **hour >= 14**. Operatorn **||** binder samman dessa två enkla delvillkor och bildar det sammansatta villkoret **(hour < 11) || (hour >= 14)**.

Förutom OCH och ELLER är NEGATION en viktig logisk operator. Den negerar sin operand dvs vänder alla dess sanningsvärden till motsatsen. Vi kommer att behandla den logiska operatorn NEGATION senare.

Visualisering av logiska operatorer

Men just nu vill vi fortsätta att öva oss lite mer i de nya begreppen. I övn 4.1 ska man med hjälp av en nästlad **for**-sats skriva ut en rektangel fylld med stjärnor (*), bestående av 9 rader och 20 kolumner (sid 144), se bilden till höger.

```
Microsoft Visual S

x =     01234567890123456789

y=1:    ********************
y=2:    ********************
y=3:    ********************
y=4:    ********************
y=5:    ********************
y=6:    ********************
y=7:    ********************
y=8:    ********************
y=9:    ********************
```

Vi ska modifiera det lite grand för att ur det rektangulära schemat av stjärnor ta ut endast vissa rader och kolumner. Låt oss börja med *en* rad och *en* kolumn. Problemet kan lösas genom att införa en **if-else**-sats

med ett sammansatt villkor i den inre **for**-slingan. Följande program demonstrerar detta:

```
// Cross.cpp
// Ritar ett kors av stjärnor
// Sammansatt villkor med den logiska operatorn ELLER (||)
#include <iostream>
using namespace std;
int main()
{
   cout << '\n';
   for (int y=1; y<=9; y++)
   {
       cout << "y=" << y << ":   ";
       for (int x=1; x<=20; x++)
           if (x==7 || y==5)  // Sammansatt villkor med ELLER
              cout << "*" ;
           else
              cout << " ";
       cout << '\n';
   }
}
```

P.g.a. logiken i **if-else**-satsens villkor producerar programmet följande utskrift:

```
y=1:         *
y=2:         *
y=3:         *
y=4:         *
y=5:   ********************
y=6:         *
y=7:         *
y=8:         *
y=9:         *
```

Det sammansatta villkoret i **if-else**-satsen är: **(x==7 || y==5)**

där **||** är ett dubbeltecken som består av två s.k. *pipes* utan mellanslag. Tecknet *pipe* som har ASCII-koden 124, ser ut som ett vertikalt streck **|** på skärmen och på utskrifter, men på tangentbordet är det representerat med streck delat av ett litet mellanrum i mitten. Som redan nämnt är dubbeltecknet **||** i C++ symbolen för det logiska ELLER. Inlagd i **if-else**-satsen i programmet ovan innebär **(x==7 || y==5)** att det skrivs ut en stjärna om den inre slingans varvräknare **x** är lika med 7 eller den yttre slingans varvräknare **y** är lika med **5**, annars skrivs det ut ett mellanslag. När är **x** lika med **7**? Det är fallet varje gång den inre slingan skriver ut den sjunde stjärnan i en rad dvs alla stjärnor i den sjunde kolumnen. Men programmet skriver även ut en stjärna om den yttre slingans varvräknare **y** är lika med **5**. När **y** är lika med **5**, skrivs ut alla stjärnor på den femte raden. I sin helhet gör alltså det sammansatta villkoret **(x==7 || y==5)** att alla stjärnor på den sjunde kolumnen

och den femte raden skrivs ut. I alla andra fall skrivs mellanslag ut. Så uppstår korset i utskriften av programmet **`Cross`**.

Logiken i ELLER innebär att det med **`||`** sammansatta villkoret är sant om minst *ett* enkelt delvillkor, **`x==7`** eller **`y==5`**, är sant. I så fall skrivs det ut en stjärna. Bara om båda de enkla delvillkoren är falska blir det sammansatta villkoret falskt och då skrivs det ut ett mellanslag. Om vi nu ersätter vårt sammansatta villkor ovan med ett nytt sammansatt villkor:

`(x==7 && y==5)`

där **`&&`** är dubbeltecknet bestående av två **`&`**-tecken som i C++ är symbolen för det logiska OCH, skulle utskriften bestå av en enda stjärna, nämligen där den sjunde kolumnen och den femte raden möts. Detta beror på att logiken i OCH innebär att det med **`&&`** sammansatta villkoret ovan är sant endast om *båda* enkla delvillkoren, **`x==7`** och **`y==5`**, är sanna. I alla andra fall är det falskt. Bara om den inre slingans varvräknare **`x`** är lika med 7 och den yttre slingans varvräknare **`y`** är lika med **`5`**, skrivs det ut en stjärna, annars ett mellanslag. Därför hamnar denna stjärna precis i mötet mellan den sjunde kolumnen och den femte raden då endast denna stjärna uppfyller kravet i det sammansatta villkoret **`(x==7 && y==5)`**. Anmärkningsvärt är att villkoret med OCH producerar *snittet* av den sjunde kolumnen och den femte raden, medan villkoret med ELLER producerar *unionen* av dessa, dvs mängdmässigt sett ”summan” av den sjunde kolumnen och den femte raden. Detta kan testas enkelt genom att i programmet **`Cross`** byta ut **`||`** mot **`&&`**, närmare bestämt i **`if-else`**-satsens villkor. Men vi ska i stället ägna oss åt ett annat logiskt experiment:

Hur kan vi åstadkomma en utskrift med korsets negativa bild, dvs alla stjärnor utom själva korset? Vi vet redan att nyckeln ligger i **`if-else`**-satsens sammansatta villkor. Hur måste detta villkor se ut för att producera en utskrift som består av korsets negativ? Det är klart att villkoret i **`if-else`**-satsen måste negeras. Men det är ju sammansatt. Hur negerar man ett sammansatt villkor? Låt oss börja med att negera det enkla villkor som bildas med likheten == som jämförelseoperator. Bland jämförelseoperatorerna i C++ (sid 127) finns negationen till likhet som kodas med **`!=`** och betyder *icke lika med*. Om vi i de enkla delvillkoren **`x==7`** och **`y==5`** byter operatorn *lika med* till *icke lika med*, får vi negationen till dessa enkla villkor: **`x != 7`** och **`y != 5`**. Men hur ska dessa enkla villkor kombineras för att bilda hela det sammansatta villkorets negation i **`if-else`**-satsen? Svaret är att operatorn **`&&`** och de negerade villkoren producerar korsets negativ:

`(x != 7 && y != 5)`

Detta måste i så fall vara logiskt ekvivalent (likvärdigt) med negationen till **`if-else`**-satsens hela sammansatta villkor i programmet **`Cross`**. Inlagt i ett program blir det:

```
// NegativeCross.cpp
// Ritar korset negativt: alla stjärnor utom korset
// Sammansatt villkor med den logiska opertorn OCH (&&)
#include <iostream>
using namespace std;

int main()
{
   cout << '\n';
   for (int y=1; y<=9; y++)
   {
       cout << "y=" << y << ":   ";
       for (int x=1; x<=20; x++)
           if (x!=7 && y!=5)    // Sammansatt villkor med OCH
              cout << "*" ;
           else
              cout << " ";
       cout << '\n';
   }
   cout << '\n';
}
```

Det nya sammansatta villkoret i **`if-else`**-satsen producerar följande utskrift:

```
y=1:   ****** *************
y=2:   ****** *************
y=3:   ****** *************
y=4:   ****** *************
y=5:
y=6:   ****** *************
y=7:   ****** *************
y=8:   ****** *************
y=9:   ****** *************
```

Varför det sammansatta villkoret **`(x != 7  &&  y != 5)`** är negationen till **`(x==7 || y==5)`** kan förefalla som en gåta vid första anblicken. För att kunna förstå detta behövs mer kunskap om logik, närmare bestämt om logiska operatorer, sanningstabeller, logiska variabler, uttryck och lagar. Allt detta tas upp i de följande avsnitten. Då kommer vi att inse att gåtans lösning grundar sig på en enkel logisk lag. Men för att kunna studera logiska lagar måste vi fördjupa oss lite mer i ämnet. Därför vill vi i nästa avsnitt titta på datatypen **`bool`** i C++ som representerar logiska värden (sanningsvärden) och lagrar dem i logiska variabler – detta för att kunna skriva och testa program som evaluerar komplexa logiska uttryck.

4.2 Datatypen bool

I C++ finns möjligheten att deklarera variabler av typen **bool** som är en *enkel* datatyp och representerar sanningsvärdena sant och falskt. **bool** är namngiven efter den engelske matematikern *George Boole* som verkade på 1800-talet och formulerade logikens lagar genom att använda matematisk notation. Variabler av typen **bool** kan endast anta sanningsvärdet **true** eller **false**. Observera att **true** och **false** är reserverade ord i C++ som representerar sanningsvärdena sant och falskt. De är *logiska konstanter* som kan tilldelas *logiska variabler* av typ **bool**. För att vara bakåtkompatibelt med C finns i C++ möjligheten att representera **false** även med talet **0** och **true** med alla tal skilda från 0. Följande program använder denna möjlighet för att skriva ut sanningstabeller för de logiska operatorerna **&&** :

```
// TruthTab.cpp
// Skriver ut sanningstabellerna till && (OCH), || (ELLER)
// Datatypen bool. Outputmanipulatorn boolalpha gör att bool-
// värden skrivs ut som true, false istället för 1, 0
#include <iostream>
using namespace std;

int main()
{
  bool p, q;            // Deklaration av logiska variabler
  cout << "\n  p \t  q\t\tp && q \t\tp || q\n";
  cout << "-----------------------------------------------\n";
  for (int i=1; i>=0; i--)
      for (int j=1; j>=0; j--)
      {
          p = i;        // Initiering av logiska variabler
          q = j;        // Typkonvertering från int till bool
          cout << boolalpha << p << '\t' << q << "\t\t";
          cout <<   (p && q)  << "  \t\t";
          cout <<   (p || q)  << '\n';
      }
  cout << '\n';
}
```

p och **q** kallas även för *booleska* variabler vilket är synonym med logiska variabler. De är deklarerade som **bool**, kan alltså tilldelas värdena **true** eller **false**. Tilldelningen sker i den nästlade **for**-satsen som skriver ut sanningstabellen. Räknarna **i** och **j** i den yttre och inre **for**-slingan går endast genom värdena **1** och **0**. Så, varje slinga har bara två varv, sammanlagt fyra varv alltså. I varje varv tilldelar **i** och **j** sina värden, **1** eller **0**, till variablerna **p** och **q** i satserna:

```
p = i;
q = j;
```

I dessa satser sker en automatisk typkonvertering från **int** till **bool** enligt tilldelningsregeln:

Automatisk typkonvertering till `bool`

I avsnitt *3.5 Automatsik typkonverering* (sid 43) behandlades

Tilldelningsregeln:

Är olika enkla datatyper involverade vid en tilldelning, konverteras alltid till den datatyp som står till vänster om tilldelningstecknet.

Därför omvandlas de **`int`**-värden som står på höger sidan i tilldelningssatserna i **`TruthTab`** ovan till **`bool`**-värden eftersom variablerna **`p`** och **`q`** som står till vänster om tilldelningstecknet, är deklarerade till datatypen **`bool`**. Vid denna typkonvertering omvandlas alla tal skilda från 0 till **`true`** och värdet 0 till **`false`**. På så sätt får **`p`** och **`q`** sina värden som skrivs ut i sanningstabellens första två kolumner och kombineras sedan med varandra i **`p && q`** samt **`p || q`** vars sanningsvärden skrivs ut i tabellens tredje och fjärde kolumn. Därför får vi, när vi kör programexemplet **`TruthTab`**, följande utskrift:

```
  p       q            p && q         p || q
-------------------------------------------------
true    true         true           true
true    false        false          true
false   true         false          true
false   false        false          false
```

Hade vi inte skickat **`boolalpha`** till **`cout`**, hade sanningstabellen bestått av talen 1 och 0, 1 istället för **`true`** och 0 istället för **`false`**. By default skriver **`cout`** ut de logiska konstanterna numeriskt. Vill man ha dem som ovan måste man initiera **`cout`**-strömmen med **`boolalpha`**, en s.k. *manipulator* som inte skriver ut något utan ändrar **`cout`**-strömmens inställningar. Man kan komplettera programmet **`TruthTab`** med ytterligare sammansatta villkor och låta programmet ställa upp deras sanningstabeller. Det är bara att baka in uttrycket i en **`cout`**-sats som läggs till i den nästlade **`for`**-satsens kropp.

4.3 NEGATION som logisk operator

I de föregående avsnitten lärde vi känna de logiska operatorerna OCH och ELLER. Nu ska vi komplettera vår lilla samling av logiska operatorer med NEGATIONen vars symbol är utropstecknet (**!**). Som exempel tar vi vårt enkla ”Gissa tal”-spel som vi redan skrivit två varianter av: **SwitchInequ** och **GuessDo** med en **do**-slinga för att kunna fortsätta spelet. Denna senaste variantens svaghet var att körningen avslutades endast om man gissat rätt. För att kunna avsluta, när användaren så önskar, kan man göra som i följande program där negationsoperatorn används:

```
// GuessNEG.cpp
// Gissa tal-spelet med slumptal i dialog
// Kan avslutas även om användaren inte gissar rätt
// Villkor med den logiska operatorn NEGATION (!)
#include <iostream>
using namespace std;

int main()
{
   srand(time(0));
   bool guessOK;
   int guessedNo, secretNo = 1 + rand() % 20;

   do
   {
      cout <<"\nGissa tal mellan 1 och 20 (Avsluta med 0): ";
      cin >> guessedNo;
      cout << "\n\t";
      if (guessedNo == 0) break;     // Bryter do-slingan
      guessOK = (guessedNo == secretNo);
      if (guessOK)
         cout << "\aGrattis, du gissade rätt!\n";
      if (guessedNo < secretNo) cout << "För litet!\n";
      if (guessedNo > secretNo) cout << "För stort!\n";
   } while (!guessOK);

   if (guessedNo == 0)
     cout <<"\nProgr.s hemliga tal var:\t"<< secretNo <<'\n';
}
```

Dessutom har spelet förbättrats i det avseendet att programmets hemliga tal inte längre bestäms redan i koden utan slumpas fram med funktionen **rand**. Om man vill avsluta innan man hunnit gissa rätt, skriver man in **0**. **if**-satsen bryter **do**-loopen med hjälp av **break**. När vi behandlade **switch**-satsen sade vi att **break** är ett reserverat ord som bryter programflödet även i loopar. Och det är precis vad den gör här. **break** bryter **do**-satsen utan att testa **do**-satsens villkor. I detta villkor **(!guessOK)** används NEGATIONEN som logisk operator vilket gör att dialogen fortsätter så länge man gissar fel och avslutas när man gissar rätt.

Till skillnad från OCH/ELLER som alltid har två operander, har NEGATIONEN endast en operand, t.ex. **p**. Negationen sätts framför den: **!p**. Sanningsvärdet vänds om: sant blir falskt och falskt blir sant. Därför har **!** följande enkla sanningstabell:

```
  p        ! p
----------------
true      false
false     true
```

I programexemplet **GuessNEG** är **p** den logiska variabeln **guessOK**:

```
bool guessOK;
...
do
{
   ...
   guessOK = (guessedNo == secretNo);
                  true eller false
   ...
} while (!guessOK);
```

Variabeln **guessOK** deklareras till datatypen **bool**. I **do**-satsen tilldelas den ett sanningsvärdet av det logiska uttrycket **(guessedNo == secretNo)**.

Logiska uttryck

Ett *logiskt uttryck* är en kombination av enkla villkor, logiska variabler (sid 135), de logiska konstanterna **true** och **false**, logiska operatorer och vanliga parenteser som till slut, när det hela evalueras, returnerar ett sanningsvärde Exempel på *logiska uttryck* är sammansatta villkor. Men även **(guessedNo == secretNo)** är ett enkelt logiskt uttryck vars värde är sant om **guessedNo** är lika med **secretNo**, annars falskt. I satsen på bilden ovan får den logiska variabeln **guessOK** detta värde. I denna sats är **=** tilldelningsoperatorn som skickar värdet **true** eller **false** till variabeln **guessOK**, medan == är en jämförelseoperator som returnerar ett sanningsvärde, dvs bestämmer om värdet inom parentesen blir **true** eller **false**.

I villkoret till **do**-satsen (efter **while**) sätts den logiska operatorn **!** framför **guessOK** för att negera den, vilket ger följande innebörd till **do**-satsens villkor:

SÅ LÄNGE	**(!guessOK)** är sant, dvs
SÅ LÄNGE	den logiska variabeln **guessOK**:s värde är falskt, ska **do**-satsen fortsätta.

Detta innebär att dialogen i **do**-satsen ska fortsätta så länge programmets användare *inte* gissat rätt. **do**-satsen kommer alltså avbrytas så fort användaren matat in ett värde för **guessedNo** som är lika med **secretNo**. Samma sak skulle man kunna skriva så här: **while (!(guessedNo == secretNo))**

4.4 Testa lösenord med logiska lagar

Här inleds programserien *Testa lösenord* med ett exempel som tillämpar våra kunskaper om loopar och logik på verifiering av lösenord. Vi börjar först med ett enkelt test av endast ett lösenord (programmet **`Passwd`**) och kommer sedan att utvecklas till att mata in lösenord även om **`Caps Lock`**-tangenten är påslagen (programmet **`PasswdCaps`**). När man tillåter påslagen **`Caps Lock`**-tangent gäller det att verifiera två lösenord. Men först det enkla testet:

```
// Passwd.cpp
// Enkelt test av endast ett lösenord
#include <iostream>
using namespace std;

int main()
{
   string input;
   bool wrongPasswd;

   do
   {
      cout << "\n\tSkriv ditt lösenord:\t";
      cin >> input;
      wrongPasswd = !(input == "hemligt");
      if (wrongPasswd)
         cout << "\n\tFel lösenord. Försök igen!";
  } while (wrongPasswd);

  cout << "\n\tOK, nu är du inloggad!\n";
}
```

En körning av programmet **`Passwd`** ger följande dialog om man vid andra försöket matar in korrekt lösenord och beaktar att man inte har **`Caps Lock`** på, annars kan det bli ännu fler inloggningsförsök.

```
        Skriv ditt lösenord:    HEMLIGT

        Fel lösenord. Försök igen!

        Skriv ditt lösenord:    hemligt

        OK, nu är du inloggad!
```

Jämförelsen mellan strängar är nämligen alltid case sensitive eftersom den görs tecken för tecken varvid tecknens ASCII-koder jämförs med varandra. Och versaler har ju andra ASCII-koder än gemener. Därför kommer inte heller inmatningen av **`Hemligt`** leda till lyckad inloggning.

Logiken i **Passwd** består av den logiska variabeln **wrongPasswd** som initieras till det logiska uttrycket **!(input == "hemligt")** och styr både **do**-loopen och **if**-satsen som ingår i den. **do**-loopen ser till att dialogen mellan program och användare fortsätter så länge **wrongPasswd** är **true** dvs så länge man matar in felaktigt lösenord, någon sträng som är skild från **hemligt**. Strängen läses in och lagras i **String**-variabeln **input**. Jämförelsen mellan den inlästa strängen och lösenordet **hemligt** görs endast en gång i det logiska uttryck vars sanningsvärde tilldelas **wrongPasswd** som används både i **do**-loopens och **if**-satsens villkor. **do**-loopen avslutas om **wrongPasswd** blir **false** dvs om strängen **hemligt** matas in. Då skriver **if**-satsen inte ut något pga **wrongPasswd**:s **false**-värde, utan användaren får ok-meddelandet som står efter **do**-loopen innan programmet avslutas. Man ser fördelen med att använda en och samma logiska variabel med en och samma initiering både i **do**- och **if**-satsen. Den logiska strukturen i båda programmen är den samma: En dialog förs vars avslutning beror på en viss (korrekt) inmatning. Ett meddelande om fortsatt dialog skrivs ut i fall av felaktig inmatning. Detta meddelande måste placeras *inuti* loopen. I fall av korrekt inmatning skrivs ut ett annat meddelande som måste placeras *efter* loopen.

Caps Lock-problematiken

När man loggar in på sitt konto på datorn, måste man se upp att **Caps Lock** inte är aktiverad, annars blir lösenordet felaktigt och man kan inte komma in. Det beror på att i de flesta operativsystem lösenord (till skillnad från användarnamn) är case sensitive. Vill man från systemsidan slippa **Caps Lock**-problematiken och underlätta inloggningen genom att tillåta även lösenord i versaler, måste ett program ingå i operativsystemet som testar lösenordet både med små och stora bokstäver. Ur säkerhetssynpunkt behöver detta inte vara något problem. När en användare känner till sitt lösenord spelar det väl ingen roll om han/hon matar in det med gemener eller versaler. En liknande frågeställning kan förekomma i andra tillämpningar, där både *ja* och *Ja* eller *nej* och *Nej* skall tillåtas som svar på en fråga om fortsatt dialog med programmet. Följande program löser **Caps Lock**-problematiken på två olika, men logiskt likvärdiga sätt:

```
// PasswdCaps.cpp
// Användaren skall kunna mata in lösenord i versal eller
// gemener. Lösning med negerade delvillkor kombinerade med
// OCH. Alternativt: Negation på det hela sammansatta ELLER-
// villkoret
#include <iostream>
using namespace std;

int main()
{
   string input;
   bool wrongPasswd;

   do
   {
```

```
      cout << "\n\tSkriv ditt lösenord:\t";
      cin >> input;
      wrongPasswd =
         !(input == "hemligt") && !(input == "HEMLIGT");
                                          // De Morgan:
      // !(input == "hemligt"  ||   input == "HEMLIGT");
      if (wrongPasswd)
         cout << "\n\tFel lösenord. Försök igen!";
   } while (wrongPasswd);

   cout << "\n\tOK, nu är du inloggad!\n";
}
```

En körning med inmatningen **HEMLIGT** i versaler ger lyckad inloggning:

```
        Skriv ditt lösenord:    HEMLIGT

        OK, nu är du inloggad!
```

Samma resultat skulle förstås ge en körning med inmatningen **hemligt** i gemener. Alla andra inmatningar kommer att misslyckas. Detta beror på **do**-loopens logik, närmare bestämt på dess avslutningsvillkor **wrongPasswd**: Så länge det är sant ska loopen fortsätta. Den logiska variabeln **wrongPasswd** i sin tur har värdet **true** om det logiska uttryck som den är tilldelad till, nämligen:

!(input == "hemligt") && !(input == "HEMLIGT")

har värdet **true**. Detta sammansatta uttryck är i sin tur sant endast om *båda* delut-trycken är sanna dvs om **input** är varken lika med **hemligt** eller **HEMLIGT**.

Är däremot den inmatade strängen **input** lika med **hemligt** eller **HEMLIGT**, ska loopen stoppas. Då kommer efter **do**-satsen ok-meddelandet att skrivas ut och pro-grammet avslutas. Viktigt för att få det hela att fungera korrekt är också att **if**-sat-sen i loopen som skriver ut meddelandet om misslyckat inloggningsförsök har samma logiska variabeln **wrongPasswd** med samma värde som villkor.

I uttrycket ovan har vi: **!p && !q** där **p = (input == "hemligt")** och
q = (input == "HEMLIGT")

Men det finns i logiken en lag som säger att uttrycket ovan dvs det sammansatta OCH-uttrycket bildat av de negerade delutsagorna, är ekvivalent (logiskt likvärdigt) med det negerade sammansatta ELLER-uttrycket:

!p && !q = !(p || q)

Ett vardagligt exempel på denna lag är: *"Jag dricker kaffe utan socker OCH utan mjölk."* betyder samma sak som *"Jag dricker kaffe varken med socker ELLER med mjölk."*. Lagen kallas efter den brittiske matematikern *De Morgan*. Formuleringen ovan är *De Morgans första lag*. Enligt denna lag kan vi alternativt till uttrycket i

programmet **`PasswdCaps`** även tilldela följande uttryck till den logiska variabeln **`wrongPasswd`**:

```
!(input == "hemligt" || input == HEMLIGT")
```

Detta är i den aktuella versionen av programmet **`PasswdCaps`** borkommenterat. Alternativet innebär: Om det *inte* är sant att den inmatade strängen **`input`** är lika med **`hemligt`** eller **`HEMLIGT`**, ska **`do`**-loopen fortsätta dvs inloggningsförsöket är misslyckat och användaren måste göra om försöket. Är däremot **`input`** lika med **`hemligt`** eller **`HEMLIGT`**, ska dialogen stoppas. Då kommer ok-meddelandet att skrivas ut och programmet avslutas. Observera att negationsoperatorn i detta alternativ måste hållas *utanför* det sammansatta ELLER-villkoret. Vart negationen ska sättas, är intuitivt inte självklart, utan framgår av *De Morgans lag*.

Den logiska OCH-operatorn **`&&`** ger en intuitivt bättre förståelig version och är identisk med ELLER-alternativet. Båda versioner tillåter inloggning med lösenord oavsett om det sker med gemener eller versaler.

De Morgans lagar

Så här kan vi sammanfatta *De Morgans lagar*:

```
!p && !q = !(p || q)
!p || !q = !(p && q)
```

För en formellt logisk formulering av dessa lagar och deras framställning med mängder se övn 4.8 på sid 146.

Beviset

Formellt är två logiska uttryck ”lika” med varandra om deras sanningstabeller är identiska. Man säger då att de är *ekvivalenta*, dvs logiskt likvärdiga. Man kan också säga att det handlar om de logiska sanningsvärdenas likhet. I praktiken betyder det att båda ledens uttryck har samma sanningstabell. Den första av De Morgans lagar kan man bevisa genom att manuellt gå igenom de enskilda operatorerna **`&&`**, **`||`** och **`!`**:s respektive sanningstabeller och sätta ihop sedan sanningsvärdena:

```
  p       q            !p && !q       !(p || q)
-----------------------------------------------
true    true              false           false
true    false             false           false
false   true              false           false
false   false             true            true
```

Man ser att båda uttryckens sanningstabeller är identiska. Mönstret som blir tydligt är följande: Appliceras negationen på det sammansatta uttrycket och sätts framför

parentesen istället för att appliceras på varje enskild operand, måste **&&** bytas ut mot **||**. Analogt gäller den andra av De Morgans lagar:

!p || !q = !(p && q)

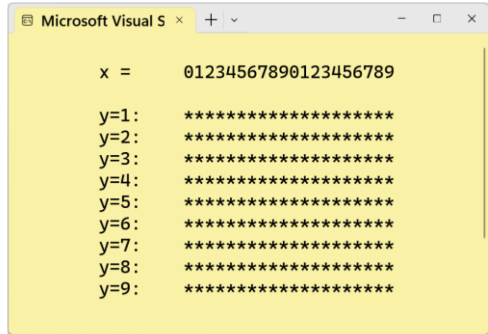
Ävenn denna ekvivalens kan visas på samma sätt som den första: båda uttryckens sanningstabeller är identiska:

```
  p       q           !p || !q       !(p && q)
----------------------------------------------
true    true            false          false
true    false           true           true
false   true            true           true
false   false           true           true
```

Därmed har vi bevisat De Morgans lagar. Gå gärna igenom de enskilda operatorerna **&&**, **||** och **!**:s respektive sanningstabeller. Sätt ihop sedan sanningsvärdena.

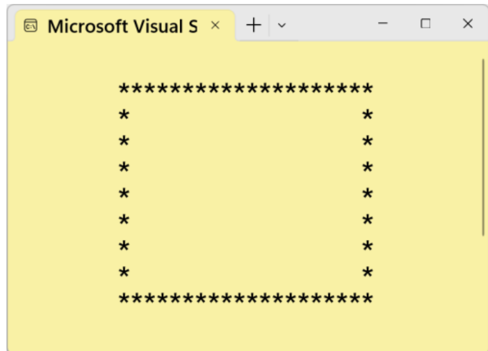
Övningar till kapitel 4

4.1 Skriv ett program som med hjälp av en nästlad **`for`**-sats skriver ut en rektangel fylld med stjärnor (`*`) till konsolen, bestående av 9 rader och 20 kolumner.



Numrera raderna och kolumnerna utan att förstöra helhetsbilden. Uppgiften lägger grunden för nästa övning samt för programmen **`Cross`** (sid 132) och **`NegativeCross`** (sid 134).

4.2 Utgående från övn. 4.1, skriv ett program som skriver ut endast ramen till rektangeln ovan, genom att modifiera det logiska villkor som styr utskriften, se bilden till höger.



Numrera inte längre raderna och kolumnerna.

4.3 Skriv ett program som skriver ut sanningsvärdet till det enkla villkoret **`a < 10`** där **`a`** är en heltalsvariabel vars värde läses in. Testa ditt program genom att mata in t.ex. 9, 10 resp. 11.

4.4 Bestäm sanningsvärden hos de följande logiska uttrycken, först med papper och penna, sedan i ett C++ program:

a) `(8 < 7) && (true || false)`
b) `!(3 < 3.01) || (!(0==0) && true)`
c) `(true || !false) && !(!(4*5==1) && false)`

4.5 Modifiera programmet **`PasswdCaps`** (sid 140) genom att lägga in kod som be gränsar antalet inloggningsförsök till t.ex. 3. Överskrider man denna gräns ska programmet avslutas efter att ha skrivit ut ett meddelande av typ *Du har försökt 3 gånger. Nu avslutas programmet!*

Tips: Använd en **`if`**-sats som avslutar programmet genom att bryta loopen med **`break`**.

4.6 Läs dokumentet ***Mängder***. Diagrammen du ser där kallas för *Venndiagram* efter den brittiske logikern John Venn (1834-1923). Mängdoperationer kan illustreras med Venndiagram.

Även logiska lagar kan illustreras med Venndiagram när de skrivs i mängdnotation, där en *mängd* motsvarar en *utsaga*.

På sid 142 formulerades *De Morgans lagar* så här:

```
!p && !q = !(p || q)
!p || !q = !(p && q)
```

Det var ett sätt som var lämpligt för programmeringen. I formell logik brukar man skriva dem så här:

¬ (p OCH q) ↔ ¬ p ELLER ¬ q

¬ (p ELLER q) ↔ ¬ p OCH ¬ q

där p och q är utsagor, ¬ är symbolen för logisk negation och ↔ symbolen för logisk ekvivalens. Ett annat sätt att formulera De Morgans lagar är att skriva dem som samband mellan mängder (mängdnotation), så här:

Anta att A och B är mängder och ð är symbolen för komplementmängden, ∩ för snittet och ∪ för unionen av två mängder, se ***Mängder***. Då kan De Morgans lagar formuleras så här:

ð (A ∩ B) = (ð A) ∪ (ð B)

ð (A ∪ B) = (ð A) ∩ (ð B)

Illustrera formuleringen av De Morgans lagar i mängdnotation, med Venndiagram.

4.7 Skriv ett program som läser in tre tal, hittar och skriver ut det största av dem. Lös problemet genom att använda tre *enkla* **`if`**-satser med sammansatta villkor och den logiska operatorn **`&&`**. På så sätt kan du i varje **`if`**-sats jämföra ett tal med de två andra. Varför måste variabeln som lagrar det största talet, initieras vid deklarationen?

4.8 Följande enkel version av Gissa tal-spelet tillåter endast *en* spelomgång (utan loop). För att koda ett trevägsval nästlar programmet en **`if-else`** -sats i en annan **`if-else`**-sats:

```
// GuessIfElse.cs
// Flervägsval med nästlad if-else-sats

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

```
int main()
{
   int guessedNo;
   cout << "\n\tGissa ett tal mellan 1 och 20:\t";
   cin >> guessedNo;
   if (guessedNo <= 17)
      if (guessedNo == 17)
         cout << "\n\tGrattis, du har " << "gissat rätt!\n";
      else
         cout << "\n\tFör litet!\n";
   else
      cout << "\n\tFör stort!\n";
}
```

Modifiera programmet ovan genom att använda logiska operatorer och sammansatta villkor i syftet att förenkla nästlingen. Det nya programmet ska göra samma sak som **`GuessIfElse`**. Bedöm i slutet själv om det har blivit mer förståelig kod.

Kapitel 5

Filhantering

Ämne		Sida	Program
5.1	Att skriva till och läsa från filer	148	**`WriteReadFile`**
5.2	Append mode	151	**`AppendFile`**
5.3	Slumplösenord i fil	153	**`RandPasswTest`**
5.4	Kryptering av filer	157	**`EncryptFile`**
Övningar till kapitel 5		161	

5.1 Att skriva till och läsa från filer

Alla våra program hittills har en sak gemensam: Så snart vi avlutat programkörningen försvinner all data från datorn, närmare bestämt från RAM – utom källkoden som ligger på hårddisken. Vi kommer inte längre åt varken programmets in- eller output efter exekveringen. Anledningen är att källkoden laddas från hårddisken till RAM när vi startar körningen och att programmets alla variabler samt in- och utdata allokeras och bearbetas i RAM. När körningen avslutas, ”dör” programmets data i RAM. Ska utdata användas efteråt, måste den under körningen skrivas ut till filer. Samma sak gäller för indata vars mängd kanske är så stor att den praktiskt taget inte kan matas in från tangentbordet, utan måste läsas in från filer. På så sätt kan filhantering i koden bli en nödvändighet.

Innan vi konkret kan inse denna nödvändighet ska vi lära oss grunderna i filhantering. Vi börjar med följande enkelt program:

```
// WriteReadFile.cpp
// Skapar filen WriteRead.txt i projektmappen eller öppnar
// den om den redan finns och skriver över innehållet.
// Skriver text från programmet till filen.
// Läser innehållet från samma fil & skriver det till skärmen
#include <iostream>
#include <fstream>                    // Innehåller ofstream
using namespace std;                  //        och ifstream

int main()
{
   char letter;                       // Objekt av typ ofstream
   ofstream fileForWrite("WriteRead.txt");  // initieras till
                                      // filen för skrivning
   fileForWrite << "\n\tDenna text finns i filen "
               << "WriteRead.txt\n";// Text skrivs till filen
   fileForWrite.close();

   ifstream fileForRead("WriteRead.txt");    // Objekt av typ
            // ifstream initieras till samma fil för läsning
   cout << "\nTexten har skrivits till filen."
               << "\n\nNu läses den från filen: \n";
   while (!fileForRead.eof())        // Så länge filslutstecknet
   {                                 // inte är nått ska tecknen
      fileForRead.get(letter);       // läsas från fileForRead
      cout << letter;                // och skrivas till skärmen
   }
   fileForRead.close();
}
```

Bland allt nytt som finns i programmet ovan låt oss börja med:

```
#include <fstream>
```

Klasserna `ifstream` *och* `ofstream`

Dessa klasser lagras i biblioteksfilen **`fstream`** som står för *file stream*.

`ofstream` står för ***output*** *file stream* och **`ifstream`** för ***input*** *file stream*.

Vad betyder här in- och output? Utgångspunkten för att bestämma ”riktningen” av out- och input är alltid *C++ programmet*. Dvs:

Output = utdata från programmet till en fil, för att *skriva* till filen.

Input = indata från en fil till programmet, för att *läsa* från filen.

Därför:

```
ofstream fileForWrite("WriteRead.txt");
ifstream fileForRead("WriteRead.txt");
```

Att skriva till en fil

I programmet **`WriteReadFile`** definieras i satsen

```
ofstream fileForWrite("WriteRead.txt");
```

objektet **`fileForWrite`** av klassen **`ofstream`**, dvs en filtyp som är avsedd för output, dvs för att skriva till den. Till en sådan fil kan man endast skriva data från programmet, inte omvänt.

Parentesen **`("WriteRead.txt")`** efter objektet **`fileForWrite`** är anropet av konstruktorn: objektet skpas och initieras samtidigt till filen **`"WriteRead.txt"`**. Dvs ett *logiskt* filnamn **`fileForWrite`** kopplas till det *fysiska* filnamnet **`WriteRead.txt`**, en fil som antingen redan finns eller skapas på hårddisken. Observera att filobjektet **`fileForWrite`** tar emot en sträng som värde. Därför måste filnamnet skrivas inom citationstecken. Det som kompilatorn gör är att söka i projektmappen efter en fil med detta namn. Om den finns kommer **`ofstream`**-satsen att radera filens innehåll utan förvarning när programmet **`WriteReadFile`** exekveras. Samtidigt sätts filens markör i början av den tomma filen, redo för att skriva i den. Om filen inte finns kommer satsen att skapa en fil med namnet **`WriteRead.txt`**, sätta markören i början av filen, redo för att skriva i den.

Filobjektet **`fileForWrite`** används sedan för att skriva till filen med:

```
fileForWrite << "\n\tDenna text finns i filen...";
```

För första gången står nu utmatningsoperatorn **`<<`** inte efter **`cout`** utan efter filobjektet **`fileForWrite`**. Om man tolkar **`<<`** som en pil från höger till vänster innebär detta att data strömmar i pilens riktning till filen **`fileForWrite`**. Precis som i satsen **`cout << data;`** där **`data`** strömmar i pilens riktning till datorns standard output-enhet dvs bildskärmen. Nu går dataströmmen till en fil istället.

Slutligen stängs filen efter **`for`**-satsen med **`fileForWrite.close();`** Den explicita stängningen av filen är av betydelse då den sätter *filslutstecknet* som är avgö-

rande för filens korrekta återanvändning. När man t.ex. senare vill läsa från filen används ofta en loop vars avslutningskriterium är just detta filslutstecken som representeras på olika sätt i olika operativsystem, t.ex. *ctrl-z* i Windows och *ctrl-d* i Unix. I C++ tar funktionen **`eof()`** reda på om filslutstecknet är nått eller ej.

Att läsa från en fil

I programmet **`WriteReadFile`** definieras i satsen

```
ifstream fileForRead("WriteRead.txt");
```

objektet **`fileForRead`** av klassen **`ifstream`** – som en filtyp för input – dvs för att läsa från filen. Samtidigt initieras filobjektet till **`"WriteRead.txt"`** – samma fil som i programmets första del. Markören sätts i början av filen, redo för att läsa från den. Lägg märke till att det måste användas en annan klass för inläsning än för skrivning då operationerna för inläsning är definierade i **`ifstream`** medan de för skrivning finns i **`ofstream`**. Vi använder funktionen **`eof()`** som står för *end of file* och är definierad i klassen **`ifstream`** för att avsluta **`while`**-loopen som både läser från filen **`WriteRead.txt`** och samtidigt skriver det lästa till skärmen:

```
while (!fileForRead.eof())
{
   fileForRead.get(letter);
   cout << letter;
}
```

Så länge filslutstecknet *inte* är nått, ska **`while`**-loopen fortsätta. När det är nått ska den avslutas. Den logiska operatorn NEGATION **`!`** kan vi sätta framför anropet därför att **`eof()`** returnerar ett sanningsvärde av typ **`bool`**. Den fördefinierade funktionen **`eof()`** returnerar **`true`** när den påträffar filslutstecknet annars **`false`**. Så länge **`eof()`** returnerar **`false`** ska **`while`**-loopen leda dataströmmen från filen **`fileForRead`** till teckenvariabeln **`letter`**. Detta är innebörden i satsen **`fileForRead >> letter;`** där pilen går från vänster till höger. Precis som i satsen **`cin >> letter;`** där dataströmmen också går i pilens riktning från datorns standard input-enhet, tangenbordet, till variabeln **`letter`**. Nu kommer data från filen **`fileForRead`** istället och lagras i teckenvariabeln **`letter`**. En körning ger:

```
Texten har skrivits till filen.

Nu läses den från filen:

        Denna text finns i filen WriteRead.txt
```

Sedan kan man kolla att utskriftens tredje rad även finns i filen **`WriteRead.txt`**.

5.2 Append mode

Programmet **WriteReadFile** börjar med att skriva till filen, med följande sats:

```
ofstream fileForWrite("WriteRead.txt");
```

Om filen **WriteRead.txt** redan finns i projektmappen raderar satsen ovan filens innehåll utan förvarning varje gång programmet exekveras. Vill man inte ha det så, utan önskar att filens gamla innehåll bibehålls och det nya kommer till som ett tillägg, kan man med följande ändring åstadkomma detta:

```
ofstream fileForWrite("WriteRead.txt", ios::app);
```

Ändringen, dvs tillägget av den 2:a parametern **append:true** i konstruktorns parameterlista gör att filen **WriteRead.txt** öppnas i s.k. *append mode* vilket innebär att man kan lägga till data i filen utan att radera befintlig data. Den syntax som används för konstruktorns 2:a parameter är ny för oss:

```
ios::app
```

Detta ändrar helt och hållet filskrivningens beteende: Markören sätts inte i början utan i slutet av filen. Filens gamla innehåll överskrivs inte utan sparas. Märkören lägger till ny text till den gamla. Följande program testar detta beteende:

```
// AppendFile.cs
// Öppnar filen WriteRead.txt som skapades i programmet
// WriteReadFile utan att radera filens gamla innehåll
// Lägger till text från programmet till filen, läser sedan
// innehållet från samma fil och skriver ut det på skärmen.
#include <iostream>
#include <fstream>                // Innehåller ofstream
using namespace std;              //        och ifstream

int main()
{
   char letter;                  // Objekt av klassen ofstream
   ofstream fileForWrite("WriteRead.txt", ios::app);
                                 // Lägger till ny text till
   fileForWrite << "\n\tDenna text har lagts till filen "
                << "WriteRead.txt.\n"; // Bibehåller filens
   fileForWrite.close();               // gamla innehåll

   ifstream fileForRead("WriteRead.txt");     // Läsning
   cout << "\n\tFöljande text har skrivits från "
        << "programmet till filen.\n\n\t"
        << "Nu läses den från filen:\n";

   while (fileForRead.get(letter))    // Läses från fil och
      cout << letter;                 // skrivs på skärmen
   fileForRead.close();
}
```

Resultatet är följande:

```
Följande text har skrivits från programmet till filen.

Nu läses den från filen:

Denna text finns i filen WriteRead.txt.

Denna text har lagts till filen WriteRead.txt.
```

Den sista raden har kommit till i och med exekveringen av programmet **`Append-File`** medan raden ovan härstammer från programmet **`WriteReadFile`**.

5.3 Slumplösenord i fil

Kalle som är systemadministratör önskar att få en färdig lista över ett antal användarnamn samt lösenord för att dela ut konton till sina användare. Därför skickar han följande uppdrag till oss:

Uppgiften:

> ”Skriv ett program som skriver ut två kolumner. I den första ska stå några användarnamn som t.ex. **`user1`**, **`user2`**, … . I den andra ska till varje användare stå ett slumpvis genererat lösenord med 6 tecken: 4 små bokstäver, 1 siffra och 1 specialtecken. Programmet ska båda skriva till en fil och visa filens innehåll.”

Lösningen:

```
// RandPasswTest.cpp
// Skapar en fil, skriver i den ett antal användarnamn och
// slumpvis genererade lösenord med funktionen randPasswd()
// Läser sedan från samma fil och skriver ut innehållet
#include <iostream>
#include <fstream>                    // Innehåller klasserna
using namespace std;                  // ofstream och ifstream
#include "randPasswd.h"               // Innehåller randPasswd()
int main()
{
   srand(time(0));
   char password[7], letter;          // 6 tecken + nolltecknet
   int antal;
   cout << "\nHur många användarnamn med lösenord "
        << "vill du ha? ";
   cin >> antal;
   ofstream fileForWrite("userPasswd.txt");

   for (int i=1; i<=antal; i++)      // Skriver tabellen
   {
      randPasswd(password);          // Anrop av randPasswd()
      fileForWrite << "\tuser" << i  // Skriver till filen
                   << "\t\t" << password << '\n';
   }
   fileForWrite.close();

   ifstream fileForRead("userPasswd.txt");
   cout << "\nVarsågod, detta står nu"
        << " i filen userPasswd.txt:\n\n";

   while (fileForRead.get(letter))     // Läser från filen och
      cout << letter;                  // skriver på skärmen
   fileForRead.close();                // så länge det finns
}                                      // tecken i filen
```

Skrivning till filen

Programmet **`RandPasswTest`** kopplar filobjektet **`fileForWrite`** till den fysiska filen **`"userPasswd.txt"`**. Sedan skriver det till filen med följande sats:

```
fileForWrite << "\tuser" << i
             << "\t\t" << password << '\n';
```

Satsen är inbyggd i en **`for`**-sats där räknaren **`i`** går från **`1`** till **`antal`** användare man matar in vid körning

Läsning från filen

I **`ifstream`**-satsen definieras objektet **`fileForRead`** till en filtyp för input och initieras till samma fil som vi skrev till. För läsning används här samma metod som i förra programmet, nämligen funktionen **`get()`** som är definierad i datatypen **`ifstream`**. Denna funktion anropas i villkoret till en **`while`**-loop:

```
while (fileForRead.get(letter))
   cout << letter;
```

för att läsa filen **`fileForRead`** tecken för tecken så länge det finns data i den dvs tills funktionen **`get()`** träffar på filslutstecknet. Funktionen **`get()`** gör två saker:

1. Hämtar ett tecken i taget från filen och tilldelar den till sin parameter **`letter`**.

2. Returnerar **`true`** om det hämtade tecknet *inte är* filslutstecknet och **`false`** om det hämtade tecknet *är* filslutstecknet. Loopens avslutningskriterium är alltså implicit inbyggd i funktionen **`get()`** som vid varje anrop läser ett tecken från filen och därmed inbakad i en loop läser hela filen. I **`while`**-loopens kropp skivs sedan de hämtade tecknen ett i taget till skärmen.

Funktionen `randPasswd()`

Följande funktion som anropas i programmet **`RandPasswTest`** i **`for`**-satsen, genererar slumplösenorden.

```
// randPasswd.h
// Skapar slumpvis genererade lösenord best. av av 6 tecken
// genom att anropa funktionen myRand() i olika intervall av
// ASCII-tabellen enligt Kalles lösenordpolicy:
// 4 små bokstäver, 1 siffra, 1 specialtecken
#include "myRand.h"                    // Innehåller myRand()
void randPasswd(char p[])
{
   for (int i=0; i<4; i++)
       p[i] = myRand(97, 122);         // 4 små bokstäver
   p[4] = myRand(48, 57);              // 1 siffra
   p[5] = myRand(33, 47);              // 1 specialtecken
   p[6] = '\0';                        // Gör p till en sträng
}
```

Funktionen **randPasswd()** tar emot som parameter arrayen **p** av **char** och tilldelar i en **for**-sats dess 4 första element tecken som slumpvis tas ur ASCII-intervallet **(97, 122)**. En blick i ASCII-tabellen (på nästa sida) visar att det är tecknen **a, b, c, ..., z** dvs det engelska alfabetet i små bokstäver. Efter anropet av funktionen **randPasswd()** sparas användarnamn och lösenord i filen.

ASCII-tabellen

	0	1	2	3	4	5	6	7	8	9
0	null	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

Det engelska alfabetet finns sammanhängande i ASCII-tabellen. Därför kan den här tilldelningen göras med en **for**-sats. Det 5:e elementet – med index **4** – tilldelas slumpvis ett tecken ur intervallet **(48, 57)**, det är siffrorna **0-9**. Det 6:e elementet – med index **5** – tilldelas något av specialtecknen i funktionen angivna ASCII-intervallet. I alla intervall ingår även gränserna, eftersom funktionen **myRand()** som anropas här flera gånger, även inkluderar intervallgränserna. Slutligen sätts noll-tecknet som strängavslutningstecken på arrayens 7:e element – med index **6** – för att göra **char**-arrayen till en sträng. Funktionen **randPasswd()** anropas i programmet **RandPasswTest** i den **for**-sats som skriver till filen. Därvid skickas parametern **password** som är en **char**-array av längden 7. I funktionen initieras arrayen med hjälp av **p**. Efter anropet är den även initierad i **main()** pga referensanrop. Så hamnar innehållet – ett slumplösenord av 4 små bokstäver, 1 siffra och 1 specialtecken – i filen.

Det ursprungliga målet var ju att skriva en lista över användarnamn och lösenord till filen **userPasswd.txt** för att dela ut konton. Lösenorden kan initialt vara vad som helst, bara de följer en policy med vissa säkerhetskrav. Sedan kan användarna efter den första inloggningen själva bestämma sina individuella lösenord.

Ett körresultat kan se ut så här:

```
Hur många användarnamn med lösenord vill du ha? 12

Varsågod, detta står nu i filen userPasswd.txt:

        user1           wfdx6*
        user2           dgjt9+
        user3           eirb9"
        user4           wyyc5$
        user5           mgro9!
        user6           xlmx1"
        user7           pwtq2&
        user8           wvzl4+
        user9           onew6+
        user10          yxnk1(
        user11          bsnq6#
        user12          qafn9$
```

Samtidigt skapas filen **userPasswd.txt** på hårddisken i samma mapp som **cpp**-filen med ovanstående listan över 20 användarnamn och lösenord som innehåll.

Vill man placera filen på en annan plats på hårddisken, måste i den sats som skapar filen, sökvägen till denna plats anges:

```
ofstream fileForWrite("C:\\ ... \\userPasswd.txt");
```

Sökvägens syntax är plattformsberoende. Den måste börja med diskens enhetsbokstav om man väljer absoluta sökvägar. Men även relativa sökvägar av typ **..\\userPasswd.txt** där **..** betyder en nivå uppåt i mappstrukturen, är möjliga. Då placeras filen t.ex. i mappen strax ovanför den aktuella mappen. Självklart borde samma sökväg anges senare i programmet i den sats som läser filen. Anledningen till användningen av **** i sökvägen är att **** är reserverad för escapesekvensernas inledningssymbol. För själva tecknet **** inom en sträng måste escapesekvensen **** användas.

I funktionen **randPasswd()** (sid 154) anropas samma funktion som användes tidigare, för att skapa slumptal i de önskade intervallen:

```
// myRand.h
// Funktion som returnerar ETT slumptal i
// heltalsintervallet[a, b]

int myRand(int a, int b)
{
    if (a < b)
        return a + rand() % (b - a + 1);
    else
        return b + rand() % (a - b + 1);
}
```

5.4 Kryptering av filer

Tidigare behandlades kryptering av text. De verktyg som utvecklades där kan med fördel användas för att kryptera även filer nu när vi lärt oss filhantering. För avväxlingens skull presenterar vi först körresultatet av ett filkrypteringsprogram och går igenom koden sedan på nästa sida:

```
Okrypterad fil:
Denna text kommer från en fil som heter Okrypterad.txt.
C++ programmet EncryptFile läser den från hårddisken, krypte-
rar den och skriver den krypterade texten i filen Krypterad.-
txt. För att testa krypteringen återställer programmet texten
och skriver den återställda texten i filen Återställd.txt.

Krypterad fil:
Æ│╜╜»n┬│ã┬n╢¢╖╖│└n┤└3╜n│╜n┤À║n┴¢╖nÂ│┬│└nØ╢└Ã¥┬│└»▓|┬ã┬|Xæyyn¥└
¢Á└»╖╖│┬nàäÖ└Ã¥┬öÀ
║n║2┴│└n▓│╜n┤└3╜nÂ3└▓▓À┴╢│╜zn╢└Ã¥┬│└»└X▓│╜n¢░Ân┴╢└À─│└n▓│╜n╢└Ã
¥┬│└»▓│n┬│ã┬│╜nÀn┤
À║│╜nÖ└Ã¥┬│└»▓|┬ã┬|XöD└n»┬┬n┬│┴┬»n╢└Ã¥┬│└À╜Á│╜n3┬│└└┬2║║│└n¥└¢
Á└»╖╖│┬n┬│ã┬│╜n¢░Â
nX┴╢└À─│└n▓│╜n3┬│└└┬2║║▓»n┬│ã┬│╜nÀn┤À║│╜n‼┬│└└┬2║║▓|┬ã┬|X

Återställd fil:
Denna text kommer från en fil som heter Okrypterad.txt.
C++ programmet EncryptFile läser den från hårddisken, krypte-
rar den och skriver den krypterade texten i filen Krypterad.-
txt. För att testa krypteringen återställer programmet texten
och skriver den återställda texten i filen Återställd.txt.

Krypteringsnyckeln:     78
```

Det här är bara ett av flera möjliga körresultat därför att krypteringsnyckeln slumpas fram vid varje körning och är `78` endast just nu.

Programstrukturen

Programmet `EncryptFile` på nästa sida som genererar utskriften ovan, är i högsta grad modulariserat och består av följande filer:

`EncryptFile.cpp`	innehåller	`main()` som anropar alla andra funktioner
`encryptText.h`		funktionen `krypt()` som krypterar text
`readShowFile.h`		funktionen `readShowFile()` som läser en fils innehåll och visar det på skärmen
`writeFile.h`		funktionen `writeFile()` som skriver till en fil

I början av `main()` skapas `char`-arrayen `fileText` för att lagra filens innehåll. Vi har förberett en liten textfil och döpt den till `Okrypterad.txt` som ska krypteras och som ligger i projektmappen.

Satsen:

```
antal = readShowFile(fileText, "Okrypterad.txt");
```

anropar funktionen **readShowFile()** som inkluderas i programmet, läser filen **Okrypterad.txt** tecken för tecken och lagrar innehållet i **char**-arrayen **fileText**. Samtidigt returnerar den ett **int**-värde som när det tilldelas variabeln **antal**, återger antalet tecken i filen. Hur den gör det kommer vi att se lite senare när vi tittar på koden. Sedan låter vi funktionen **myRand()** generera ett slumptal mellan **1** och **1000** som tilldelas variabeln **key**, slumpnyckeln som används vid kryptering. Därför skickas den tillsammans med **fileText** och **antal** till **krypt()**:

```
// EncryptFile.cpp
// Läser text från en fil, krypterar den med en slumpnyckel,
// skriver krypterade texten till en annan fil och visar den
// Slumpnyckeln ger vid varje körning en annan kryptering
// Dekrypterar texten och skriver den till en tredje fil samt
// visar både den återställda filen och slumpnyckeln
#include <iostream>
#include <fstream>
using namespace std;
#include "encryptText.h"       // Innehåller        krypt()
#include "myRand.h"            //                  myRand()
#include "readShowFile.h"      //            readShowFile()
#include "writeFile.h"         //               writeFile()

int main()
{
   srand(time(0));
   char fileText[1000];        // Arrayens storlek behöver
                               // ändras vid större filer
   int antal, key = myRand(1, 1000);        // Slumpnyckeln
   cout << "Okrypterad fil:\n";

   antal = readShowFile(fileText, "Okrypterad.txt"); // Läser
   krypt(fileText, key, antal);  // Krypterar med slumpnyckel

   writeFile(fileText, "Krypterad.txt", antal);    // Skriver
   cout << "Krypterad fil:\n";

   readShowFile(fileText, "Krypterad.txt");          // Läser
   krypt(fileText, -key, antal);                // Dekrypterar

   writeFile(fileText, "Återställd.txt", antal);   // Skriver
   cout << "\n\nÅterställd fil:\n";

   readShowFile(fileText, "Återställd.txt");         // Läser
   cout << "Krypteringsnyckeln:\t" << key << "\n";
}
```

I funktionen **krypt()** vars kod inkluderas i en headerfil i början av programmet och kommer att visas senare, förskjuts varje tecken med slumpnyckeln **key**:s värde i ASCII-tabellen – inte någon avancerad krypteringsmetod – men i och med den är

slumpbaserad får man ett annat resultat vid varje körning. Krypteringsnyckeln **`key`** används senare för att återställa filen genom att anropa funktionen **`krypt()`** med **`key`**:s negativa värde dvs sätta tillbaka alla tecken på sina ursprungliga platser i ASCII-tabellen. Men mellan dessa två anrop av krypteringsfunktionen – en gång för kryptering, en andra gång för dekryptering (framhävda med vit bakgrund i koden) – har vi två andra anrop, först:

```
writeFile(fileText, "Krypterad.txt", antal);
```

som skriver den krypterade texten **`fileText`** till filen **`Krypterad.txt`**. Parametern **`antal`** skickas för att ha ett avslutningskriterium för skrivningen till filen. Sedan:

```
readShowFile(fileText, "Krypterad.txt");
```

som läser den krypterade texten från filen och visar den på skärmen, efter att den med **`writeFile()`** hamnat där. Till skillnad från det första anropet av funktionen **`readShowFile()`** (förra sida) tilldelas här returvärdet inte till någon variabel då det inte behövs. Anropets resultat kan beskådas på sid 157 och visar att filen verkligen är krypterad. Nu återstår beviset på att krypteringen gjorts på ett sätt att vi alltid har möjligheten att återställa filen och att vi verkligen får filens ursprungliga skick. Där-ör anropas krypteringsfunktionen andra gången:

```
krypt(fileText, -key, antal);
```

där tecknet **`-`** inte ska tolkas som bindestreck i texten utan som det matematiska *tecknet minus* till variabeln **`key`**:s talvärde då **`key`** är deklarerad som ett heltal av typ **`int`**. Vi skickar alltså **`key`**:s negativa värde till samma krypteringsfunktion för att sätta tillbaka alla tecken på sina ursprungliga platser i ASCII-tabellen. En blick på funktionen som är externlagrad förklarar saken:

Funktionen krypt()

```
// encryptText.h
// Tar emot en text via arrayen t och krypterar den genom
// att förskjuta alla tecken med n steg i ASCII-tabellen
// Kontrollerar textens slut med 3:e parametern antal

void krypt(char t[], int n, int antal)
{
   for (int i = 0; i < antal; i++)
        t[i] = t[i] + n;
}
```

Den aktuella parametern **`key`** öveförs vid anrop till den formella parametern **`n`**. När **`n`** får ett positivt **`key`**-värde, ökas tecknens ASCII-kod med **`n`**. Ett negativt **`key`**-värde minskar ASCII-koderna med samma belopp dvs sätter tecknen tillbaka på sina ursprungliga platser. Därför kan vi använda samma funktion även för dekryptering. Filinnehållet **`fileText`** som skickas till **`t`** är en array av **`char`**. För att kunna avsluta

for-satsen måste vi använda oss av den tredje parametern **antal**. De andra externlagrade funktioner som anropas i **EncryptFile** är följande:

Funktionen readShowFile()

```
// readShowFile.h
// Funktion som läser innehållet i filen fileName tecken för
// tecken, lagrar det i arrayen t samt visar det på skärmen
// Returnerar dessutom antal tecken som läses och visas

int readShowFile(char t[], string fileName)
{
   int i;              // Antal tecken som läses från filen
   char tecken;
   ifstream fileForRead(fileName);
   for (i=0; fileForRead.get(tecken); i++)
   {
       t[i] = tecken;
       cout << tecken;
   }
   fileForRead.close();
   return i;
}
```

Funktionen **get()** i **for**-satsens villkor läser ett tecken från filen, lagrar det i **char**-variabeln **letter**, flyttar markören till nästa tecken i filen och returnerar **true** om det finns tecken kvar och **false** om det stöter på filslutteckne t. Så läses filen, lagras i **char**-arrayen **t** samt skickas till **cout**. Antalet tecken returneras.

Funktionen writeFile()

```
// writeFile.h
// Funktion som skriver texten t bestående av antal
// tecken till filen fileName

void writeFile(char t[], string fileName, int antal)
{
   ofstream fileForWrite(fileName);
   for(int i = 0; i < antal; i++)
      fileForWrite << t[i];
   fileForWrite.close();
}
```

I **for**-satsen skriver utmatningsoperatorn **<<** arrayen **t** tecken för tecken till filen **fileName**. Parametern **antal** – antal tecken – används för att avsluta **for**-satsen.

Övningar till kapitel 5

5.1 Skriv ett program som skapar en tom fil, skriver i den texten ”Den här texten kommer från mitt första C++ filhanteringsprogram” och sedan läser från den samt skriver ut innehållet på skärmen. Som mall kan du ta programmet **`Write-ReadFile`** och modifiera den (sid 148).

5.2 Modifiera programmet från övn 5.1 ovan: Istället för att hårdkoda texten i programmet, läs in den så att programmet skriver vilken inläst text som helst till filen och läser den sedan därifrån.

5.3 Varje gång man kör programmen från övn 5.1 eller 5.2 efter första gången, rensas och återställs filen och endast den senaste texten hamnar i den. Skriv ett program som gör samma sak som övn 5.2 men bibehåller filens gamla innehåll och lägger till den nyinlästa texten utan att radera gammal data. Du kan åstadkomma det genom att öppna filen i *append mode*.

5.4 Modifiera funktionen **`randPasswd()`** (sid 154) som genererar slumplösenord genom att använda en annan lösenordpolicy: 3 små bokstäver, 2 stora bokstäver (inkl. ? och @) och 2 specialtecken. Testa din funktion i programmet **`Rand-PasswTest`** (sid 153) som skriver ut till en fil dessa slumpvis genererade lösenord med ett antal användarnamn.

5.5 ***Filkryptering (Aktivitet)***

Bilda grupper à två studerande i klassen. Valet av gruppkompis är fritt.

Skriv ett hälsningsmeddelande eller en kort text och spara den i en oformatterad textfil, typ **`*.txt`**. Kryptera filen med funktionen **`krypt()`** på sid 159. Anteckna krypteringsnyckeln vid den aktuella körningen. Skicka den krypterade filen samt krypteringsnyckeln till din gruppkompis med uppmaningen att dekryptera filen och skicka tillbaka den återställda texten till dig.

Byt ut rollerna i gruppen och upprepa experimentet.

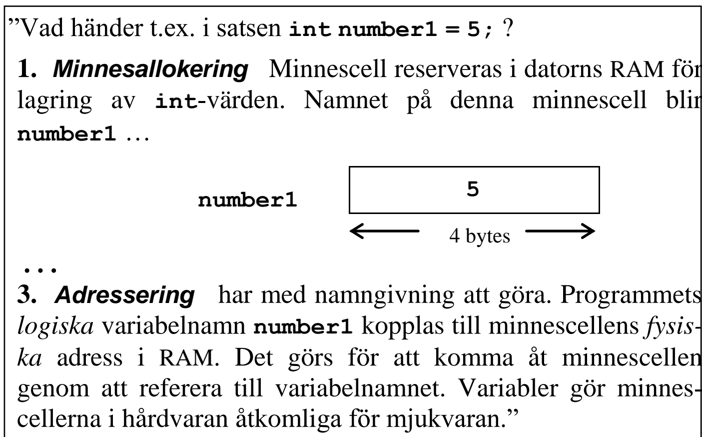
Kapitel 6

Pekare

	Ämne	Sida	Program
6.1	Vad är en pekare?	163	
6.2	Deklaration och initiering av en pekare	165	**`Pointer`**
6.3	Adress- och värdeoperatorn	168	**`Value`**
6.4	Operatorn **`new`**	174	**`New`**
6.5	Pekare och array	178	**`PointArray`**
	- Pekararitmetik	180	**`PointArithm`**
6.6	Stränghantering med pekare	182	**`Initials`**
Övningar till kapitel 6		186	

6.1 Vad är en pekare ?

När vi deklarerade och initierade variabler sa vi så här:



”Vad händer t.ex. i satsen **`int number1 = 5;`** ?

1. ***Minnesallokering*** Minnescell reserveras i datorns RAM för lagring av **`int`**-värden. Namnet på denna minnescell blir **`number1`** …

number1 | 5 |
← 4 bytes →

. . .

3. ***Adressering*** har med namngivning att göra. Programmets *logiska* variabelnamn **`number1`** kopplas till minnescellens *fysiska* adress i RAM. Det görs för att komma åt minnescellen genom att referera till variabelnamnet. Variabler gör minnescellerna i hårdvaran åtkomliga för mjukvaran.”

Vid deklaration av en variabel uppstår alltså en länk mellan en *adress* (hårdvara) och det variabelnamn vi använder i koden (mjukvara). När vi pratar om *adress* menar vi alltid en fysisk plats i datorns RAM-minne (*Random Access Memory*).

Pekare är en ny datatyp för lagring av *adresser* till värden som kan vara av vilken vanlig datatyp som helst.

Till varje vanlig datatyp finns en pekardatatyp: *pekare-till*-datatyp

Variabler deklarerade till denna nya datatyp kallas *pekarvariabler*.

Exempel: Adressen till en **`int`** lagras i en pekarvariabel av typ *pekare-till-***`int`**.

När ett C++ program körs sker all minneshantering – datorns verkliga jobb – med de fysiska adresserna. Varje översättning av ett variabelnamn till en fysisk minnesadress och vice versa tar en viss tid. Därför är det effektivare att kunna arbeta med adresser direkt.

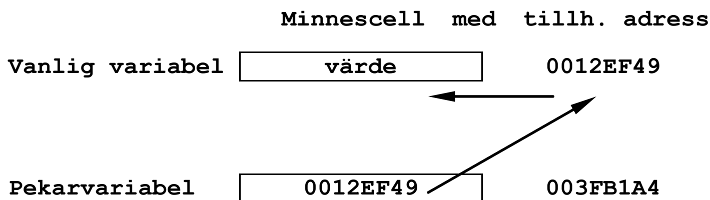
Ju närmare man kan komma hårdvaran desto snabbare kan datorn bearbeta data. C++ är p.g.a. sina rötter i C det programmeringsspråk bland de objektorienterade högnivåspråken som står hårdvaran närmast. Språket har flera verktyg för att kontrollera och styra hårdvaran. Ett av dessa verktyg är *pekare* vars ursprung går tillbaka till C. C++ har ärvt pekarkonceptet från C.

Pekare = ett nytt parallellt system

Alla våra program som vi hittills skrivit, kan skrivas om till nya pekarversioner om man bara byter ut alla vanliga variabler till pekarvariabler. Man refererar till de minnesceller som lagrar programmets variabler med adresser istället för variabelnamn. Det verkar som om vi har tagit steget in i en helt ny värld inom C++ programmering. I själva verket har vi satt på oss nya glasögon som ser på samma värld ur ett mer maskinnära perspektiv. Närmare bestämt är pekare inte *en* datatyp utan täcker C++ språkets alla datatyper. Till varje vanlig datatyp finns en pekardatatyp.

Med pekarvariabler kan man göra allt man kan göra med vanliga variabler, och mer därtill. Det speciella med pekarvariabler är att deras värden är adresser. Bilden nedan illustrerar hur man arbetar med pekare:

En pekarvariabels värde = adressen till en vanlig variabel



Språkbruket bland programmerare:

”Pekarvariabeln *pekar på* den vanliga variabeln”

Man tilldelar en pekarvariabel en vanlig variabels adress för att komma åt den vanliga varibelns minnescell indirekt via adressen istället för via variabelnamnet.

Detta förklarar samtidigt ordet *pekare*. Själva pekarvariabelns adress – adressens adress så att säga – är i detta sammanhang av mindre praktisk betydelse, men den finns där i alla fall.

Pekare tillåter programmeraren att från koden komma åt minnesadresser vilket öppnar helt nya möjligheter att skriva program. I ett program utan pekare görs all minnesallokering vid kompileringen, dvs *statiskt*. Med pekare finns möjligheten att allokera minne under programmets gång, dvs *dynamiskt*. Vid statisk minnesallokering måste man i förväg reservera utrymme som sedan kanske inte används i full utsträckning vid körning vilket innebär slöseri med minnesutrymme. Vid dynamisk minnesallokering reserveras utrymme när det behövs under exekveringen. På så sätt kan man skräddarsy användningen av minnesresurserna.

6.2 Deklaration och initiering av en pekare

Följande program simulerar situationen ”*pekar på*” som beskrevs ovan och introducerar **`*`** som symbolen för pekare och *adressoperatorn* **`&`** (tecknet *ampersand*) som symbol för *adressen* till en variabel. Det ger oss möjligheten att från koden komma åt vanliga variablers adresser utan att behöva använda det speciella format som finns för adresser, nämligen hexadecimala tal. Detta görs genom att deklarera en pekarvariabel och tilldela till den adressen till en vanlig variabel.

```
// Pointer.cpp
// Deklaration och initiering av en pekarvariabel
// Asterisken * är symbolen för pekarvariabel
// Amperand & framför en variabel ger adressen till variabeln
#include <iostream>
using namespace std;

int main()
{
   double vanligVar = 15.6;
   double *pekarVar;       // Deklarerar en pekare-till-double

   pekarVar = &vanligVar;  // Initierar pekarVar med adressen
                           // till vanligVar
                           // "pekarVar pekar på vanligVar"
   cout << "\n\t\t\tVärde\t\t\tAdress\t\t\tStorlek\n\n"
        << "Vanlig variabel:\t" << vanligVar      << "\t\t\t"
        << &vanligVar << '\t'   << sizeof(vanligVar)<< "\n\n"
        << "Pekarvariabel:\t\t" << pekarVar          << "\t"
        << &pekarVar << "\t"    << sizeof(pekarVar)  << "\n";
 }
```

Programmet simulerar minnesbilden på förra sidan när det körs:

```
                  Värde               Adress             Storlek

Vanlig variabel:  15.6                000000E592CFF         8

Pekarvariabel:    000000E592CFFB58    000000E592CFFB78      8
```

Som man ser är pekarvariabeln **`pekarVar`**:s *värde* en adress som är den vanliga variabeln **`vanligVar`**:s *adress*, vilket är ett resultat av tilldelningssatsen **`pekarVar = &vanligVar;`**. Men låt oss börja med deklarationssatsen:

`double *pekarVar;`

Satsen deklarerar variabeln **`pekarVar`** till datatypen pekare-till-**`double`**. Det är asterisken **`*`** som talar om för kompilatorn att den variabel som följer dvs **`pekarVar`** ska vara en pekarvariabel. Observera att själva pekarvariabelns namn är **`pekarVar`**

utan asterisk. Placeringen av **`*`** kan variera så att deklarationen kan även se ut så här:

`double* pekarVar;` eller **`double * pekarVar;`**

Eventuella mellanslag mellan datatyp och variabelnamn saknar betydelse. Avgörande är att asterisken **`*`** står *mellan* dem för att känneteckna variabeln som följer som en pekarvariabel. Fördelen med notationen (sättet att skriva) **`double *pekarVar;`** som vi kommer att använda i fortsättningen, är att man ser att det handlar om en pekarvariabel, fördelen med **`double* pekarVar;`** (OBS! skillnaden) är att man ser att det handlar om datatypen pekare-till-**`double`**. Generellt är:

datatyp **`*`**

C++ syntax för datatypen *pekare-till-datatyp* som t.ex. kan användas vid explicit typkonvertering (ex. på nästa sida). Det finns inga begränsningar för val av *datatyp* som t.o.m. i sin tur kan vara en pekarvariabel vilket leder till datatypen *pekare-till-pekare*. Vill man deklararera *flera* pekarvariabler av samma typ i *en* sats måste asterisken upprepas i den kommaseparerade listan, t.ex.:

```
double *pekarVar1, *pekarVar2, *pekarVar3, ... ;
```

Utelämnas **`*`** på den andra, tredje eller någon efterföljande plats i listan kommer dessa variabler bli deklarerade som vanliga **`double`**-variabler och inte som pekarvariabler.

Precis som hos vanliga variabler reserverar deklarationssatsen **`double *pekarVar;`** minnesutrymme av den storlek som är föreskriven för datatypen pekare-till-**`double`**. Observera att med denna sats reserveras minnesutrymme endast för själva pekarvariabeln, men inte för det den ska peka på, dvs endast för adressen, inte för det **`double`**-värde som ska lagras *vid* denna adress. I programmet **`Pointer`** har vi med hjälp av **`sizeof(pekarVar)`** fått denna information. Körresultatet på förra sidan visar att pekarvariabelns minnesstorlek är **`8`** bytes. Pekarvariabler till *alla* datatyper tar alltid lika mycket i minnesutrymme som datatypen **`double`** gör. I den aktuella C++ installation vi använder är det **`8`** bytes. Det beror på att värdet som lagras i en pekarvariabel, är en adress oavsett vilken typ av data den kommer att peka på. Adresser representeras i C++ i ett visst format nämligen med *hexadecimala tal*. Körresultatet ovan visar att pekarvariabeln **`pekarVar`**:s värde är **`0012FF78`**. Alla hexidecimala tal till en viss övre gräns kan lagras i **`8`** bytes. Det anmärkningsvärda är att pekare alltid tar **`8`** bytes oavsett hur stora datamängder de än pekar på.

Hexadecimala tal

Det är tal som är representerade i talsystemet med basen 16. Om vi hade 8 fingrar på varje hand vore denna anmärkning onödig för då kunde vi räkna med hexadecimala tal. Det avgörande skälet för att människan bestämt sig för att räkna med decimala tal är antalet fingrar – inte för att det decimala systemet i sig är det mest effektiva. Ett bevis på det är datorns binära talsystem. När det gäller minnesadresser har man i C++ som i de flesta andra språk bestämt sig för att *visa* adresser i hexadecimal form.

Anledningen är bl.a. för att lätt känna igen dem som adresser och bättre kunna skilja dem från andra datatyper. Närmare bestämt är det **`cout`** som visar adresser i detta format vid utskrift.

Om du tar fram Kalkylatorn i Windows kan du omvandla tal till och från olika talsystem bl.a. mellan **Dec**imal och **Hex**adecimal. Glöm inte att i menyn Visa välja välja undermenyn Avancerad. Om man väljer Hex (längst till vänster under displayen) aktiveras en knapprad med bokstäverna A-F längst ned till höger. Dessa bokstäver används nämligen som "siffror" i det hexadecimala talsystemet. Precis som man har de 10 siffrorna 0-9 i det decimala talsystem som bygger på basen 10, använder sig det hexadecimala talsystemet av de 16 siffrorna 0-9 och A-F där A simulerar den hexadecimala "siffran" 10, B 11 osv. och F är motsvarigheten till 15. Därför förekommer i hexadecimala tal även bokstäverna A-F som t.ex. i pekarvariabeln **`pekarVar`**:s värde **`0012FF78`** enligt körresultat av programmet **`Pointer`**. I Windows Kalkylator kan man omvandla detta hexadecimala tal till dess decimala motsvarighet: 1 245 048. I äldre C++ installationer kan man få utskriften **`0x0012FF78`** istället, där det inledande prefixet **`0x`** är kännetecknet för hexadecimala tal. Enligt den aktuella C++ standarden skriver **`cout`** ut hexadecimala tal utan prefix. Men vill man i koden tilldela pekarvariabeln **`pekarVar`** adressen ovan direkt utan adressoperator, måste prefixet sättas. De två inledande nollorna kan ignoreras då de inte bidrar med någon information till talet:

```
pekarVar = (double *) 0x12FF78;
```

Observera att en explicit typkonvertering till datatypen pekare-till-**`double`** behövs. Utan den blir det kompileringsfel vilket visar att man måste specificera vilken datatyp pekaren ska peka på. Utan denna specificering är hexadecimala tal i sig varken av typ pekare-till-**`double`** eller kan automatiskt konverteras till den, vilket än en gång understryker att det inte finns någon datatyp *pekare* utan endast *pekare-**till** någon datatyp*. När vi pratar om pekare menar vi alltid pekare-till någon datatyp.

Men vi ska inte fördjupa oss i hexadecimala tal. Det är inte heller nödvändigt då det finns ett bekvämt sätt att komma åt adresser till minnesceller som redan är allokerade t.ex. genom deklaration. I C++ finns en fördefinierad operator för just detta ändamål nämligen adressoperatorn som vi redan lärde känna i programmet **`Pointer`** och som vi ska närmare titta på i nästa avsnitt där vi samtidigt kommer att lära oss hur man kommer åt *värdet* till en vanlig variabel med hjälp av en pekare som pekar på den.

6.3 Adress- och värdeoperatorn

Programmet **`Pointer`**:s ”svaghet” ur pekarsynpunkt är att tilldelningen och utskriften av variabeln **`vanligVar`** sker på konventionellt sätt – med den vanliga variabelns namn – och inte med pekarvariabeln som pekar på den. Kopplingen mellan den vanliga och pekarvariabeln är enkelriktad: Vi får adressen till ett värde, men inte omvänt: värdet lagrat vid adressen.

För att helt och hållet gå över till kod som endast använder pekarvariabler, måste vi konstruera en dubbelriktad koppling. Vi måste ha *två* operatorer som t.ex. i matematiken **`+`** och **`-`** eller **`*`** och **`/`**. Dessa operatorer uppträder parvis, är motsatta till varandra och tar ut varandra. Man kallar dem för *inversa* operatorer. Även i C++ finns faktiskt *två* operatorer varav den ena ger *adressen* till ett värde och den andra *värdet* lagrat vid en adress. Därför kallar vi dem för *adress*- och *värdeoperator*. Den första har vi använt i programmet **`Pointer`** och den andra kommer vi att visa i följande program:

```
// Value.cpp
// Att referera till en variabel indirekt via dess adress
// istället för konventionellt via variabelnamnet.
// * i är icke-deklarationssatser värdeoperatorn som refere-
// rar till variabelns värde som pekarvariabeln pekar på.
// Den är invers till adressoperatorn & .
#include <iostream>
using namespace std;

int main()
{
   double vanligVar;           // Deklaration utan initiering
   double *pekarVar = &vanligVar; // pekarVar pekar på van-
                                  //               ligVar
   *pekarVar = 15.6;           // Här är * värdeoperatorn
                               // VÄRDET i den variabel som
                               // pekarVar pekar på, dvs van-
                               // ligVar, ska bli 15.6
   cout << "\n\t\t\tVärde\t\t\tAdress\t\t\tStorlek\n\n"
        << "Vanlig variabel:\t" << vanligVar << "\t\t\t"
        << &vanligVar << '\t' << sizeof(vanligVar) << "\n\n"
        << "Pekarvariabel:\t\t" << pekarVar << "\t"
        << &pekarVar << "\t" << sizeof(pekarVar) << "\n\n"
        << "Värde- * och adressoperatorn & är inversa"
        << " operatorer:\n\n\t\t"
        <<       " *(&vanligVar) = " << *(&vanligVar)
        << "\n\t\t &(*pekarVar) = " << &(*pekarVar)<< "\n\n";
}
```

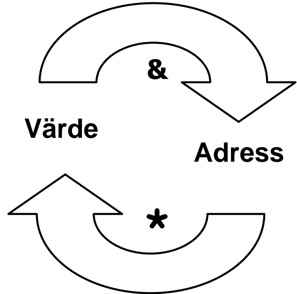
Inittieringen av **`vanligVar`** sker med pekaren: **`*pekarVar = 15.6;`** Asterisken **`*`** har här en annan betydelse än symbolen för pekare, när den deklareras. Här betyder asterisken *värdet* i den minnescell med adressen **`pekarVar`**.

Ett körexempel av programmet **`Value`** ger följande bild:

```
                  Värde                  Adress            Storlek

Vanlig variabel:  15.6                   000000087731FB38    8

Pekarvariabel:    000000087731FB38       000000087731FB58    8

Võrdeoperatorn * och adressoperatorn & õr inversa operatorer:

                 *(&vanligVar) = 15.6
                 &(*pekarVar)  = 000000087731FB38
```

De två sista raderna bekräftar att adress- och värdeoperatorn är inversa till varandra. Adressoperatorn går från värdet **`vanligVar`** till adressen, medan värdeoperatorn går från adressen **`pekarVar`** till värdet. Utför man dem efter varandra återvänder man till utgångspunkten. Cirkeln i bilden till höger sluts. De tar ur varandra. T.ex. betyder:

`*(&vanligVar)`



värdet som lagras vid adressen som anges inom parentes, dvs **`vanligVar`**:s värde. **`*`** och **`&`** neutraliserar varandra och vi får tillbaka **`vanligVar`** vars värde **`15.6`** skrivs ut i körresultatet. Analogt betyder:

`&(*pekarVar)`

adressen till variabeln som anges inom parentes. Där står variabeln som **`pekarVar`** pekar på, dvs **`vanligVar`**. I sin helhet får vi adressen till **`vanligVar`** som är **`pekarVar`**. Igen: **`&`** och **`*`** tar ut varandra och vi får tillbaka den ursprungliga variabeln **`pekarVar`** vars värde **`0012FF78`** skrivs ut i körresultatet.

Anledningen till att sambandet mellan dessa operatorer tas upp här, är att det belyser och underlättar förståelsen för pekaren. Man kan memorera bättre, när man känner igen en underliggande struktur – något som bakom det rent tekniska ger ett koncept och en mening till det hela. Därför sammanfattar vi:

> Värdeoperatorn * är invers (motsatt) till adressoperatorn &

En annan intressant aspekt, som även har praktisk betydelse för kodningen, är att den lilla asterisken **`*`** som ganska ofta förekommer i våra program med pekare, faktiskt har olika betydelser i olika sammanhang. Vi ska titta närmare på denna skillnad.

Asteriskens två olika betydelser

1. **I deklarationssatser** talar asterisken `*` om för kompilatorn att den variabel som följer ska vara en pekarvariabel. Följande sats betyder:

   ```
   double *pekarVar;
   ```

   att variabeln `pekarVar` deklarareras till datatypen pekare-till-`double`. Samtidigt reserveras en minnescell på `8` bytes för lagring av adresser till `double`-värden.
   I explicita typomvandlingar: (*datatyp* `*`) *uttryck* betyder `*` att *uttryck* ska konverteras till datatypen *pekare-till-datatyp*.

2. **I alla andra satser** har asterisken `*` en helt annan betydelse (bortsett från multiplikation). När den i sådana satser skrivs framför en redan deklarerad och tilldelad pekarvariabel, refererar den till den variabelns *värde* som pekarvariabeln pekar på:

   ```
   *pekarVar = 15.6;
   ```

   betyder att *värdet* i den variabeln som `pekarVar` pekar på ska bli `15.6`. I `Value` är variabeln som pekaren `pekarVar` pekar på, pga den andra satsen i `main()`, den vanliga variabeln `vanligVar`. Alltså tilldelar satsen ovan variabeln `vanligVar` värdet `15.6`. Denna tilldelning sker indirekt dvs via adressen istället för konventionellt med variabelnamnet.

Vi ska nu behandla de inversa operatorerna adress- och värdeoperatorn:

Adressoperatorn &

I programmet `Pointer` gör deklarationen av variabeln `vanligVar` som `double` att en minnescell allokeras av storleken 8 bytes för lagring av ett `double`-värde. Därmed är minnescellens adress skapad och i princip känd för programmet. Tack vare *adressoperatorn* är denna adress även tillgänglig för programmet. Adressoperatorn symboliseras med tecknet & och tillämpas på en variabel genom att skriva:

`&variabelnamn` eller `& variabelnamn`

Även här saknar mellanslaget betydelse. Avgörande är att tecknet `&` skrivs *framför* variabelnamnet. Då returnerar adressoperatorn adressen till denna variabel. Variabeln behöver inte ens vara tilldelat ett värde. Det räcker att den är deklarerad. Även om vi i `Pointer` inte hade tilldelat variabeln `vanligVar` värdet `15.6` hade vi kunnat skriva ut dess adress med `cout << &vanligVar;` direkt efter deklarationen. Variabeln som adressoperatorn tillämpas på behöver inte vara en vanlig variabel. Den kan även vara en pekarvariabel. Då får man pekarvariabelns adress. Vi har i `Pointer` tillämpat adressoperatorn både på en vanlig variabel och en pekarvariabel. Körresultatet visar att pekarvariabelns *adress* är en annan adress än pekarvariabelns *värde*. Adressoperatorn kan även tillämpas på namngivna konstanter, däremot inte på icke-namngivna konstanter.

Vi återkommer till den mest intressanta observationen i körresultatet av **`Pointer`**, nämligen att pekarvariabeln **`pekarVar`**:s *värde* som är en adress, är identiskt med den vanliga variabeln **`vanligVar`**:s *adress* vilket beror på tilldelningssatsen:

```
pekarVar = &vanligVar;
```

som gör att **`pekarVar`** pekar på **`vanligVar`**. Vi kan rent formellt göra denna tilldelning då adressoperatorn returnerar adressen till en **`double`** och pekarvariabeln **`pekarVar`** är deklarerad som en pekare-till-**`double`**. Datatyperna på båda sidor överensstämmer alltså. Hade, om detta inte varit fallet, automatisk typkonvertering tillämpats? Svaret är nej. Observera att alla regler för *automatisk typkonvertering* endast gäller för *enkla* datatyper. Pekare är inga enkla datatyper utan sammansatta eller härledda då de är baserade på andra datatyper. Kom ihåg att med pekare allid menas pekare-till någon datatyp. Så, försöker man att tilldela **`pekarVar`** en annan datatyp än pekare-till-**`double`** som t.ex.: **`pekarVar = 2.5;`** blir det kompileringsfel med felmeddelandet att ett **`double`**-värde (enkel datatyp) inte kan konverteras till en pekare-till-**`double`** (sammansatt).

Med tilldelningssatsen ovan har vi gjort något man typiskt brukar göra med pekare. Vi har tagit en befintlig minnesadress, nämligen **`&vanligVar`**, och skrivit den i pekarvariabeln **`pekarVar`**:s minnescell. Vi har kopplat ihop den vanliga variabeln med pekarvariabeln. Först nu förtjänar pekaren sitt namn då den pekar på ett annat allokerat minnesutrymme, den vanliga variabelns minnesutrymme. Gör man inte det är pekaren inte väl definierad. Kom ihåg att först när en variabel är väl definierad får den användas. Först när en pekarvariabel pekar på ett minnesutrymme får den användas. Den typiska "användningen" av en pekarvariabel är att med hjälp av den komma åt det minnesutrymme som den pekar på. Detta har vi inte gjort i **`Pointer`** men kommer att göra i nästa programexempel.

Värdeoperatorn *

Vi kallar den så med tanke på att den returnerar *värdet* analogt till adressoperatorn som returnerar adressen. *Värdeoperatorn* **`*`** (asterisken i en annan betydelse än hittills) ger oss möjligheten att komma åt vanliga variablers *värden* via deras adresser efter att ha kopplat ihop den vanliga variabeln med pekarvariabeln. Andra beteckningar som förekommer i litteraturen är **`*`***-operatorn*, *indirektoperatorn* eller på engelska *dereference operator*.

Nästa programexempel **`Value`** på nästa sidan introducerar värdeoperatorn genom att visa hur man kommer åt *värdet* till en vanlig variabel med hjälp av en pekare som pekar på den. Samtidigt demonstreras sambandet mellan adress- och värdeoperatorn. I programmet **`Value`** har till skillnad från **`Pointer`** variabeln **`vanligVar`** inte tilldelats ett värde direkt via sitt namn utan indirekt via sin adress dvs en pekare som pekar på den. Dessutom har **`pekarVar`**:s deklarations- och tilldelningssats slagits ihop:

```
double *pekarVar = &vanligVar;
```

där i deklarationsdelen (till vänster om tilldelningstecknet) asterisken **`*`** använts som symbol för pekarvariabel. Precis som hos vanliga variabler kan man initiera pekarvariabler redan vid deklarationen. Gör man det till en vana kan man minska risken för oinitierade skräpadresser.

Helt ny här är tilldelningssatsen **`*pekarVar = 15.6;`** där asterisken **`*`** inte längre är kännetecknet för pekarvariabel. Satsen tilldelar istället den variabel som **`pekarVar`** pekar på, nämligen **`vanligVar`**, värdet **`15.6`**. Detta beror på att asterisken **`*`** inte har samma betydelse i programmets alla satser. Vi är redan vana att använda samma tecken för olika saker. Symbolen för multiplikation är samma som för pekare vid deklaration, och nu kommer en användning till för asterisken. Tillgången till tecken är ju begränsad, så det finns helt enkelt inte ett unikt tecken till varje tänkbar operation. Sammanhanget måste avgöra den rätta tolkningen.

Dessutom kan programmet **`Value`** användas – om man vill – för en kraschtest. Tar man bort kopplingen mellan den vanliga och pekarvariabeln, dvs deklarerar bara **`pekarVar`** utan att tilldela den adressen till **`vanligVar`**, så kan man väl kompilera koden. Men exekveringen leder till minneskrasch därför att det uppstår en pekare med en oinitierad skräpadress som pekar på ett minnesutrymme som med största sannolikhet är upptaget av ett annat program i datorns RAM.

Inversa operatorer

Inversa operatorer finns fler än man tror: Addition och subtraktion, multiplikation och division, att potentiera och dra roten, i C++: ökningsoperatorn **`++`** och minskningsoperatorn **`--`** , den logiska negationen **`!`** , **`new`** och **`delete`** (sid 174) osv. Tillämpar man en operator på en operand och sedan den inversa operatorn på resultatet av denna operation får man tillbaka den ursprungliga operanden. Därför säger man: Inversa operatorer tar ut varandra, de neutraliserar varandra. T.ex. tillämpar man operationen +3 på operanden 4 och sedan den inversa operationen –3 på resultatet, får man tillbaka operanden 4. Dvs: (4 + 3) – 3 = 4. Ett annat enkelt exempel är (6 x 5) / 5 = 6. Den logiska negationen är ett exempel på en operator som är sin *egen invers*: **`!(!v) = v`** där **`v`** kan vara ett godtyckligt villkor eller en godtycklig utsaga. Man kan alltså förstå den dubbla negationen även med hjälp av den inversa operatorn. Vad har allt detta att göra med adresser och pekare? Jo, även värdeoperatorn kan man förstå utifrån adressoperatorn som vi redan känner till, plus konceptet om den inversa operatorn.

Vad gör adressoperatorn? Den tar en variabel och returnerar dess adress. Följaktligen gör värdeoperatorn som dess invers det motsatta: den tar en adress och returnerar dess variabel. Men vad är ”dess variabel”? Jo, det är inget annat än den variabel pekaren pekar på. Lite mer noggrant måste vi alltså formulera: Värdeoperatorn tar en pekare som pekar på en variabel och returnerar denna variabel. Själva variabeln som returneras behöver endast vara deklarerad. Antingen är den redan tilldelad ett värde, då returnerar värdeoperatorn detta värde. Eller så är variabeln inte tilldelad än, då kan man med värdeoperatorn tilldela den ett värde. Detta gjorde vi i programmet **`Value`** med satsen **`*pekarVar = 15.6;`** där **`*`** är värdeoperatorn. När vi skriver

denna sats pekar pekarvariabeln **pekarVar** redan på den vanliga variabeln **vanligVar** genom tilldelningen **pekarVar = &vanligVar;** vilket gör att **vanligVar** i resten av programmet blir utbytbar mot ***pekarVar**. De refererar båda till samma minnescell och kan användas vare sig för att läsa eller skriva data. Med ***pekarVar = 15.6;** skrivs värdet **15.6** i minnescellen. Med **cout << vanligVar;** läses det från minnescellen. Beviset är att vi får **15.6** utskrivet med denna **cout**-sats fast vi tilldelat variabeln med ***pekarVar**. Detta visas i utskriften till programmet **Value** som visades tidigare.

Vi återkommer till det tidigare utlovade kraschtestet. Kommenterar man i **Value** bort kopplingen **pekarVar = &vanligVar;** mellan den vanliga variabeln och pekarvariabeln kan man fortfarande kompilera, men det blir exekveringsfel följt av minneskrasch. Testet kan lätt göras genom att ändra den andra satsen i **main()** till:

```
double *pekarVar;          // = &vanligVar;
```

Kopplingen som bryts ovan är ju inget annat än pekarvariabeln **pekarVar**:s tilldelningssats. Tar man bort den blir pekaren oinitierad. Den är deklarerad, men pekar på ingenting, närmare bestämt på inget minnesutrymme allokerat av programmet. Men som vi vet finns det ju rent fysiskt inga tomma minnesceller. I den deklarerade pekarvariabeln **pekarVar**:s minnescell står från tidigare användning något slumpmässigt odefinierat skräpvärde – en skräpadress. Med värdeoperatorn ***pekarVar** tillämpad på pekarvariabeln försöker vi nu komma åt värdet som **pekarVar** pekar på. Pga skräpadressen pekar **pekarVar** på ett minnesutrymme som med största sannolikhet är upptaget av ett annat program i datorn. Försöker man att från ett program komma åt det minnesutrymme, blir det exekveringsfel och programmet kraschar pga minneskonflikt.

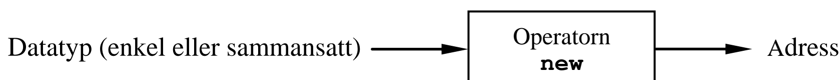
Ett annat lärorikt test är att i **Value** byta ut satsen ***pekarVar = 15.6;** mot:

```
*((double *) 0x12FF78) = 15.6;
```

Programmet kommer att fungera precis som förut, förutsatt att adressen som använts i denna sats är samma adress som **pekarVar**:s värde. Observera att sådana adresser blir olika när man kör programmet på annan dator eller vid olika tillfällen på samma dator. Det bästa är att ta adressen vid en aktuell körning från pekarvariabelns värde (se körresultatet ovan). Observera att den första asterisken är värdeoperatorn medan den andra tillhör den explicita typkonvertering som omvandlar den råa adressen till en adress-till-**double**. Testet visar att värdeoperatorn som vanligen skrivs framför en pekarvariabel, även fungerar när man skriver den framför en konstant adress. I så fall måste den konverteras till rätt datatyp och peka på ett allokerat minnesutrymme, i det här fallet variabeln **vanligVar** som även på det här sättet kan få värdet **15.6**.

6.4 Operatorn `new`

I alla pekarprogram hittills har vi använt både vanliga och pekarvariabler genom att tilldela de vanliga variablers adresser till pekarvariablerna. För att kunna göra det var vi dock tvungna att deklarera vanliga variabler. Nu går vi ett steg vidare: Vi eliminerar de vanliga variablerna och använder endast pekarvariabler. Men hur allokeras minne om vi inte deklarerar vanliga variabler? Svaret är operatorn **`new`** som vi ska behandla i detta avsnitt. **`new`** tar emot en *datatyp* som parameter, allokerar minne av den storlek som datatypen föreskriver och returnerar det allokerade minnesutrymmets *adress*:



En *operator* fungerar som en funktion med returvärde. Den tar emot ett antal parametrar, gör något och returnerar ett värde. Enda skillnaden är syntaxen: operator använder inga parenteser till skillnad från funktion. **`new`** är en fördefinierad minnesallokeringsoperator i C++. Skillnaden till traditionell allokering med vanlig deklaration är att **`new`** allokerar minne under programmets körning – *at run time* medan vanlig deklaration gör det under kompileringen – *at compile time*. Vi har alltså att göra med *dynamisk minnesallokering*. Efterom operatorn **`new`**:s returvärde är en adress måste den tilldelas en pekarvariabel. Man vill ju kunna referera till data som lagras vid denna adress. Data som man inte kan referera till dvs inte kan komma åt, är ju meningslös. Därför ser den generella syntaxen för operatorn **`new`** ut så här:

```
datatyp *pekarvariabel = new datatyp;
```

Ex.:

```
int *pekInt = new int;
```

Själva **`new`**:s syntax står till höger om tilldelningstecknet. Till vänster deklarareras en pekarvariabel som tar hand om **`new`**:s returvärde. Här ser man också att operatorn **`new`** inte behöver parenteser kring sin parameter datatyp, vilket skiljer en operator från en funktion med returvärde. Observera att **`new int`** allokerar minne för lagring av en **`int`** och returnerar en adress till en **`int`**. Därför måste den också tilldelas en variabel av typ pekare-till-**`int`**. Tilldelning till en pekare-till-annan datatyp ger kompileringsfel.

I tidigare program hänvisade vi till minnescellerna direkt med *variabelnamn*. Nu gör vi samma sak indirekt med deras *adresser*. Vilken metod som är ”direkt” och vilken ”indirekt” kan man ha olika åsikter om. När man vant sig vid att använda pekare kan man t.o.m. tycka att hanteringen av data via adresser är det naturliga sättet, vilket inte är någon dum idé med tanke på att variabelnamn ändå är en slags användarvänlig mjukvarulänk till minnescellens adress i hårdvaran. I alla fall är det fullt *möjligt* att eliminera vanliga variabler och jobba endast med pekare. Om det också är

meningsfullt är en annan fråga som måste avvägas från fall till fall. Ofta är en blandning en gångbar kompromiss.

Nästa program är ett exempel på användningen av operatorn **`new`**. Där finns inga vanliga variabler längre eftersom **`new`** allokerar minne och returnerar dess adress till en pekare som används som referens till minnescellen. När värdeoperatorn tillämpas på denna pekare ger den oss åtkomst till innehållet i minnescellen för att både läsa och skriva i den.

```
// New.cpp
// Adderar heltal tills man matar in 0
// Använder enbart pekarvariabler, inga vanliga variabler
// Operatorn new tar en datatyp, allokerar minne passande
// till den och returnerar adressen till minnesutrymmet
// Åtkomst till allokerat minne med hjälp av värdeoperatorn
#include <iostream>
using namespace std;

int main()
{
   int *pekInt = new int;      // Deklarerar och initierar pe-
                               // karvariabeln pekInt
   int *pekSum = new int(0);   // Nollsätter dessutom minnes-
                               // cellen som pekSum pekar på
   do
   {
      cout << "\nGe ett heltal (avsluta med 0): ";
      cin >> *pekInt;
      *pekSum += *pekInt;
   } while (*pekInt != 0);
   cout << "\nSumman av heltalen är:\t" << *pekSum << "\n\n";
}
```

Ett körexempel visar att programmets användare inte märker att koden endast använt pekare:

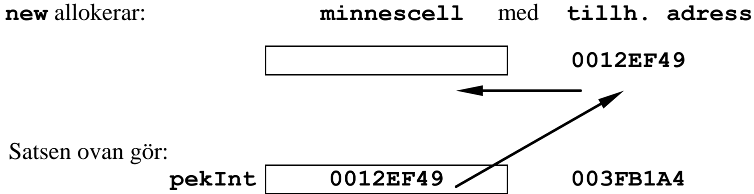
```
Ge ett heltal (avsluta med 0): 359

Ge ett heltal (avsluta med 0): 237

Ge ett heltal (avsluta med 0): 8

Ge ett heltal (avsluta med 0): -12

Ge ett heltal (avsluta med 0): 0

Summan av heltalen är:  592
```

Vad gör new?

Den första satsen i **`main()`**:

```
int *pekInt = new int;
```

allokerar minnesutrymme passande för lagring av ett **`int`**-värde och gör därmed samma sak som en vanlig deklarationssats – med skillnaden att den allokerade minnescellen inte får något namn. Istället returnerar **`new`** adressen. Vi tilldelar adressen i ovanstående sats till pekarvariabeln **`pekInt`** vilket gör att **`pekInt`** nu pekar på den av **`new`** allokerade minnescellen:



Samtidigt deklarerar satsen ovan **`pekInt`** som en variabel av typ pekare-till-**`int`** vilket även reserverar minne för **`pekInt`**. Detta är minnesekonomiskt inte någon fördel just i detta sammanhang, utan kommer att vara det först om pekaren pekar på större datamängder. Just nu har minnescellen inget deklarerat värde. Först i **`do`**-satsen läses in ett värde med:

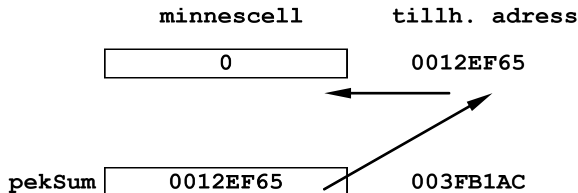
```
cin >> *pekInt;
```

som kommer att hamna i den tomma minnescellen ovan då pekaren **`pekInt`** pekar på den. Här betyder **`*`** värdeoperatorn som används för att komma åt minnescellen och skriva det inlästa värdet i den. Man kan säga att **`*pekInt`** ersätter det konventionella variabelnamnet.

Den andra satsen i **`main()`**:

```
int *pekSum = new int(0);
```

gör samma sak som den första plus att den samtidigt initierar minnescellen som **`pekSum`** pekar på med värdet **`0`**, så att minnesbilden efter satsen ovan ser ut så här:



Observera att tilldelningen av värdet **`0`** till minnescellen görs här på samma sätt som vi en gång kallade objektorienterad initiering – t.ex. **`int sum(0);`** istället för **`int sum = 0;`** – kan man även använda med **`new`**. Skillnaden är bara att det vanliga

variabelnamnet **sum** nu inte längre finns när **new** är med i bilden. Då skrivs initieringen **(0)** direkt efter datatypen. I objektorienterad programmering kommer denna initieringsteknik att få sin fulla förklaring och användas väldigt ofta med **new**.

Att minnescellen som **pekSum** pekar på, initieras till **0**, men inte minnescellen som **pekInt** pekar på, har att göra med programmets ändamål, men också med satsen:

```
*pekSum += *pekInt;
```

som är en kortform för och identisk med ***pekSum = *pekSum + *pekInt;** Ändamålet med programet **new** är att addera heltal tills man avslutar inmatningen med **0**. Det görs i en **do**-loop där de inmatade heltalen läses in och lagras i minnescellen som **pekInt** pekar på. Med satsen ovan adderas dessa tal med värden som finns i minnescellen som **pekSum** pekar på. Referensen ***pekSum** till denna minnescells värde i programmet förekommer även till höger om tilldelningstecknet i den kod som satsen ovan är en kortform för. Därför innehåller den ett odefinierat skräpvärde, när satsen utförs i allra första varvet av **do**-loopen. Då alla tal som matas in, ackumuleras (läggs samman) i denna minnescell, kommer de adderas till detta skräpvärde om vi inte nollsätter den i början. Detta kommer att resultera i ett felaktigt skräpvärde som summa.

Däremot kommer minnescellen som **pekInt** pekar på att initieras vid inmatningen. Därför behöver den inte initieras explicit i programmet även om det inte vore fel att göra det. I följande körexempel initieras ***pekInt** med värdet **359** som skriver över det befintliga skräpvärdet. **do**-loopens villkor ***pekInt != 0** gör att programmet kan avslutas med inmatning av **0** vilket gör villkoret falskt.

Programmet **new** syftade åt att bekanta oss med operatorn **new** och visade möjligheten att i C++ programmera enbart med pekare. Det är fullt möjligt att skriva en pekarversion av alla program – en bra övning för nybörjare som vill lära sig pekare. Däremot visade **new** inte hur dynamisk minnesallokering *egentligen* går till trots att minnet allokerades dynamiskt med **new**. Intressant och praktiskt relevant blir dynamisk minnesallokering först när man frigör eller utökar ett redan allokerat minne senare i programmet.

6.5 Pekare och array

När man fortsätter att programmera med pekare i C++ upptäcker man snart att det finns ett släktskap mellan pekare och array. Arrayen kan beskrivas som en användarvänlig variant av pekare.

Detta är också anledningen, varför andra programmeringsspråk har ”avskaffat” pekare. I C#, Java, Python, ... t.ex. har man utnyttjat detta släktskap för att bli av med pekare. Medan man i dessa språk arbetar med arrays, objekt och referenser görs förstås all minnesallokering med adresser. Ett datorprogram kan inte fungera utan adressering, oavsett programmeringsspråk. Att dessa språk fungerar utan pekare beror på att array är en variant av pekare som kan ersätta den. Man struntar då i vissa möjligheter av minnesmanipulation. I C++ har man som ett arv från C kvar möjligheten att styra adresshanteringen från koden. Hur släktskapet mellan pekare och array ser ut och vilka konsekvenser det har, vill vi studera i detta avsnitt.

Vi börjar med att skriva om programmet **`ArrayDef`** till en pekarversion:

```
// PointArray.cpp
// Pekarversion av programmet ArrayDef
// Skapar med new en array av int, tilldelar den elementvis
// med värdeoperatorn och kopierar värdena till en annan
// array. Pekarstegning ersätter indexering.
#include <iostream>
using namespace std;
int main()
{
   int *no = new int[4];       // Deklarerar och tilldelar
                               // pekarvariabeln no. new
                               // allokerar minne och no pe-
                               // kar på 1:a elementet i det.
   *no     = 64;               // Samma som *(no+0) = 64
                               // Elementvis tilldelning
   *(no+1) = 86;               // Pekarstegning: pekarens
   *(no+2) = 34;               // position flyttas fram
   *(no+3) = -6;
   int *copy = new int[4];     // Ny pekare copy
   for (int i=0; i <= 3; i++)
       *(copy+i) = *(no+i);    // Elementvis tilldelning
   cout << '\n';
   for (int i = 0; i <= 3; i++)
       cout << "\tcopy:s " << i+1 << ":a element *(copy+"
            << i << ") = " << *(copy+i)
            << " med pekarposition " << i << "\n\n";
}
```

Efter tilldelningen av arrayen **`no`** skapas en andra array, även den med **`new`**, som en ny pekare **`copy`** pekar på. Värdena i **`no`** kopieras elementvis över till **`copy`** i en **`for`**-sats. Slutligen skrivs ut värdena i **`copy`** samt deras position, vilket visar att kopieringen har gått bra:

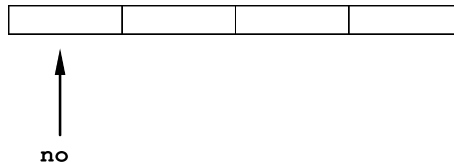
```
copy:s 1:a element *(copy+0) = 64 med pekarposition 0
copy:s 2:a element *(copy+1) = 86 med pekarposition 1
copy:s 3:a element *(copy+2) = 34 med pekarposition 2
copy:s 4:a element *(copy+3) = -6 med pekarposition 3
```

new skapar array

Den första satsen i **main()**:

```
int *no = new int[4];
```

deklarerar en pekare-till-**int** kallad **no**. Samtidigt allokerar **new** minne för en array av **int** med **4** element som **no** kommer att peka på, vilket i praktiken innebär att peka på det första elementet i arrayen. Följande minnesbild uppstår:



Dvs ett sammanhängande minnesområde bestående av **4** minnesceller där varje minnescell kan lagra ett **int**-värde. Till skillnad från den ursprungliga arrayversionen **ArrayDef**, saknar denna array namn eftersom den skapats med **new**. Arraynamnet har ersatts av pekaren **no** som vi kan jobba med istället. Att tilldela den *första* minnescellen ett värde, är inget problem: I den andra satsen av **main()**

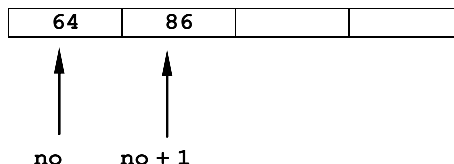
```
*no = 64;
```

gör värdeoperatorn detta: **no** pekar på arrayens första minnescell. Men hur kan vi tilldela den *andra* minnescellen ett värde? Eftersom arrayen saknar namn kan vi ju inte komma åt elementen med *index* vilket vi gjort hittills. Lösningen är:

```
*(no + 1) = 86;
```

Pekarstegning

Pekarens position flyttas fram *ett* steg in i arrayen:



Pekaren pekar nu på den andra minnescellen, ser minnesbilden ovan.

I koden åstadkommer man detta genom att skriva `no+1`. Additionsoperatorn betyder här inte matematisk addition, inte konkatenering utan pekarstegning, dvs flyttning av pekarens position ett steg framåt. Sedan tillämpas värdeoperatorn på den nya positionen: `*(no+1)` så att det refereras till värdet i den minnescell som `no+1` pekar på. Pekarstegning ersätter alltså indexering. Vi kan istället för index som vi använde hos arrays, använda pekaren, för att komma åt arrayens element.

Arraynotation vs. pekarnotation

Vad gäller åtkomsten till arrayelementen kan släktskapet mellan array och pekare bl.a. beskrivas generellt så här:

`no[n]` gör samma sak som `*(no + n)`

där `no` till vänster är en array och `n` arrayens index, medan `no` till höger är en pekare och `n` antalet steg vid pekarstegningen. Dessutom är `*` värdeoperatorn. Koderna ovan är ömsesidigt utbytbara. Till vänster har vi *arraynotation*, till höger *pekarnotation*. I ett och samma program kan arraynotation blandas med pekarnotation efter att allokering av minnesutrymme skett på ett korrekt sätt. För specialfallet `n = 0` följer av den generella regeln ovan:

`no[0]` gör samma sak som `*(no + 0)`
`*(no)`
`* no`

där arraynotationen visar det *första* arrayelementet. Men vad gäller pekarnotationen är `*(no + 0)` identiskt med `*(no)` och därmed med `*no`. Detta känner vi igen som värdet i den minnescell som `no` pekar på och som vi redan använt för att initiera det *första* arrayelementet: `*no = 64;`

Varför börjar arrayindex med 0 ?

Nu kan vi också förklara indexregeln för arrays enligt vilken numreringen av index i arrays alltid börjar med 0. Indexering av en array börjar med 0 därför att, när arrayen skapas vid deklationen, pekaren som pekar på det första elementet, inte stegats dvs antalet steg vid pekarstegningen är 0. Först när vi flyttar pekaren till nästa position så att den pekar på det andra elementet i arrayen, blir antalet steg vid pekarstegningen 1 och därmed blir också indexet 1. På samma sätt går det vidare.

Pekararitmetik

När vi pratar om *pekarstegning* och att *flytta pekarpositionen* menar vi egentligen det som kallas *pekararitmetik*. Med detta begrepp vill man anknyta till vanlig räkning och se på additionen i `no + 1` som vanlig addition av två hexadecimala tal. Vi sade att additionsoperatorn här betyder stegning av pekarens position vilket är en populär tolkning. I själva verket handlar det om att addera den hexadecimala adressen som lagras i pekarvariabeln `no` med 4. Orsaken till att pekarstegning med `1` steg innebär addition med 4, är att `no` är en pekare-till-`int` och `int` tar 4 bytes. Hur

många bytes **1** steg vid pekarstegning motsvarar beror alltså på datatypen som pekaren pekar på. På samma sätt hanteras indexet hos arrays. Mellan **no[0]** och **no[1]** ligger 4 bytes om **no** är en array av **int**. I arrays har man byggt in den användarvänliga tolkningen av pekarstegning som index. I hela detta resonemang är det underförstått att en array alltid är ett sammanhängande minnesområde och att adressering görs bytevis. Följande lilla test illustrerar resonemanget:

```
// PointArithm.cpp
// Skriver ut adresserna till elementen i en int-array.
// Omvandlar dem till decimala tal med explicit typkonverte-
// ring. Differensen är lika med minnesstorleken för en int
#include <iostream>
using namespace std;

int main()
{
   int *no = new int[4];
   cout << '\n';
   for (int i=0; i <= 3; i++)
    cout << "no:s " << i+1 << ":a element lagras vid adress "
            << no+i << " = " << (int) (no+i) << '\n';
                          // Omvandling till decimalt format
   cout << "\nAdressernas differens är "
        << (int) (no+1) - (int) no;
   cout << "\n";
}
```

Pekaren **no** pekar på **int**-arrayen skapad av **new** och därmed på det första elementet i den. **for**-satsen skriver ut adresserna **no**, **no+1**, **no+2** och **no+3** både i hexadecimalt och decimalt format. Omvandling till decimalt format sker i **for**-satsen:

```
(int) (no+i)
```

med explicit typkonvertering. Adresserna som är hexadecimala tal av typ **int *** dvs pekare-till-**int** konverteras till vanliga heltal av typ **int**. Omvandling till decimalt format visar redan att adressernas differens är 4 när vi kör:

```
no:s 1:a element lagras vid adress 00481C10 = 4725776
no:s 2:a element lagras vid adress 00481C14 = 4725780
no:s 3:a element lagras vid adress 00481C18 = 4725784
no:s 4:a element lagras vid adress 00481C1C = 4725788

Adressernas differens är 4
```

Men för att illustrera pekararitmetik beräknar vi denna differens genom att subtrahera de till decimalt format omvandlade första två adresserna från varandra:

```
(int) (no+1) - (int) no;
```

6.6 Stränghantering med pekare

Släktskapet mellan pekare och array som vi observerade i programmet **`PointArray`** (sid 178) blir ännu mer påtagligt när vi går över från pekare-till-**`int`** till pekare-till-**`char`**. Det beror på att strängar i C++ är teckenkedjor som avslutas med nolltecknet. Man kan låta en pekare peka på en **`char`** som då blir början av en sträng och sätta nolltecknet i slutet av strängen. Strängens längd spelar ingen roll. Detta gäller inte för pekare-till-**`int`**. Det som gör stränghanteringen med pekare mer intressant än med array, är möjligheten till dynamisk minnesallokering. Med pekare som verktyg för stränghantering blir man av med begränsningen som arrayens statiska minneshantering medför.

I följande program ska vi använda datatypen *pekare-till-**`char`*** för att låta programmet hitta initialerna i vårt för- och efternamn, vilket vi redan löst med array av **`char`**. Vi börjar även här med att lagra en sträng i en **`char`**-array. Men sedan skickas arrayens adress till en funktion som tar reda på samt skriver ut namnets första och andra initial. Funktionen tar emot arrayens adress via sin parameter av typ pekare-till-**`char`** och bearbetar strängen med denna pekare, vilket kommer att resultera i en enkel men samtidigt elegant kod: **`while`**-loopen som går igenom strängen behöver ingen extra indexvariabel.

```
// Initials.cpp
// Läser in för- och efternamn, skickar strängens pekare till
// funktionen initials() som hittar och skriver ut initia-
// lerna. Stränghantering med datatypen pekare-till-char
// Pekare som parameter i en funktion
#include <iostream>
using namespace std;

void initials(char *namn)     // Funktion med pekaren namn som
{                             // formell parameter
   cout << *namn;             // Ger första initialen
   while (*namn++ != ' ')     // Går igenom endast förnamnet
      if (*namn == ' ')       // Hittar namnets sista bokstav
         cout << *(namn + 1);   // Ger andra initialen
}

int main()
{
   char dittNamn[20];
   cout << "Ge ditt för- och efternamn: ";
   cin.getline(dittNamn, 20);
   cout << "\nHej " << dittNamn << ", dina initialer är: "
                                          << " \n\n\t";
   initials(dittNamn);       // Funktionsanrop med arraynamnet
   cout << "\n";             // som aktuell parameter
}
```

Programmet **`Initials`** ger följande utskrift när jag matar in mitt namn:

```
Ge ditt för- och efternamn: Taifun Alishenas

Hej Taifun Alishenas, dina initialer är:

        TA
```

Första initialen **T** skrivs ut i funktionen **initials()** med satsen **cout << *namn;** då **namn** vid funktionsanropet fått sitt värde överfört från **dittNamn** som är adressen till den första bokstaven i strängen **Taifun Alishenas**. Tecknet som är lagrat vid denna adress dvs ***namn** är strängens första bokstav **T**.

Den andra initialen **A** skrivs ut med satsen **cout << *(namn+1);** som finns i **if**-satsen. I varje varv av loopen testar **if**-satsen om det aktuella tecknet ***namn**, är mellanslaget. Detta är endast fallet när sökningen har nått förnamnets sista bokstav. Då skrivs ut det tecken som kommer efter det aktuella tecknet dvs ***(namn+1)**. Men detta är efternamnets första bokstav dvs den andra initialen. Observera att **while**-satsen endast söker igenom förnamnet. Den använder ingen information om strängens längd. Vi behöver inte skicka arrayens storlek till funktionen.

Array som konstant pekare

Här ska vi titta på hur strängen hamnar i funktionen. Via funktionsanropet förstås:

```
initials(dittNamn);
```

Arrayen **dittNamn** är definierad i **main()**. Vi läser in den med **cin.getline()** för att få in hela namnet inkl. mellanslaget mellan för- och efternamn. Vid anropet överförs arrayen **dittNamn** till pekarvariabeln **namn** i funktionen **initials()**. Men *hur* sker denna parameteröverföring? Vad är det exakt som skickas? Är det hela namnet – inkl. mellanslaget – som skickas eller är det endast adressen? Om det är namnet, som är en array, hur kan den tas emot av pekaren **namn** i funktionen? Detta är bara möjligt om namnet är en adress. Slutsats:

> Ett arraynamn är en konstant pekare.

Med ”konstant pekare” menas ett fast adressvärde. Man kan alltid tilldela ett arraynamn till en pekarvariabel p.g.a. denna egenskap. En gång skapad kan en konstant pekare inte ändras i programmet. En pekarvariabel däremot kan initieras till ett adressvärde, ändra sedan sitt värde och anta ett nytt värde. Att vi hittills ändå pratat om array*variabler* syftar snarare åt arrayens *element*, inte på namnet. Arrayelementen är variabla och deras värden kan ändras. De lagras däremot alltid vid samma adress som i programmet representeras av arraynamnet.

while-loopen

Efter anropet av funktionen **initials()** och parameteröverföringen pekar nu pekaren **namn** på den inlästa sträng som innehåller både för- och efternamn. Att hitta den första initialen är inte svårt: Satsen **cout << *namn;** skriver ut den eftersom ***namn** är värdet som pekaren **namn** pekar på. Eftersom vi i det här läget inte ändrat pekarpositionen pekar **namn** på den *första* bokstaven i den inlästa strängen som är den första initialen. Den andra initialen hittas i loopen:

```
while (*namn++ != ' ')
   if (*namn == ' ')
      cout << *(namn + 1);
```

När vi börjar står pekarpositionen på strängens första bokstav. Loopen stoppas när sökningen stöter på ett mellanslag – tecknet mellan för- och efternamnet. Därför förutsätter programmet **Initials** att användaren matar in sitt för- och efternamn skilda med endast *ett* mellanslag – det vanliga sättet alltså att ange sitt namn. Så länge det aktuella tecknet ***namn** inte är mellanslag fortsätter loopen vilket framgår av loopens villkor ***namn++ != ' '** . Men vad gör ökningsoperatorn **++** direkt efter ***namn**? Utan den vore det som sades, mer begripligt. Jo, ökningsoperatorn **++** har inget med villkoret att göra utan den ska se till att pekarpositionen flyttas fram med ett steg i varje varv av loopen. På så sätt går man igenom strängen. Observera att detta är pekarstegning (sid 180). Dvs ökningsoperatorn **++** opererar på pekaren, därför att den står direkt efter **namn**, medan före pekaren står värdeoperatorn ***** för att hämta värdet (tecknet) som **namn** pekar på. Så uppstår koden ***namn++** där två olika operatorer tillämpas på **namn**. Och det roliga är: De stör inte varandra. Var och en utför sitt jobb oberoende av den andra. Men *när* exakt sker stegningen av pekarpositionen? Är det före, efter eller i **while**-satsen? Är det före eller efter loopens **if**-sats? Eftersom vi använt ökningsoperatorns postfixvariant kan det inte vara för. Faktum är att pekarstegningen görs *efter att* **while***-satsens villkor testats*, vilket görs i början av varje varv strax innan **if**-satsen.

En enklare variant

Man får en enklare variant när man flyttar pekarstegningen från loopens villkor till loopens kropp, närmare bestämt till det ovan beskrivna stället i koden, dvs om man byter ut programmets **while**-sats med:

```
while (*namn != ' ')
{
   namn++;
   if (*namn == ' ')
      cout << *(namn+1);
}
```

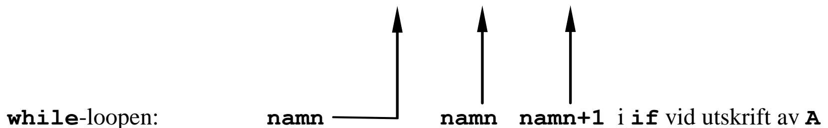
Låt oss i tankarna frysa programkörningen till det kritiska ögonblick då den andra initialen **A** skrivs ut. Vid denna utskrift står pekaren **namn** på en position som pekar på mellanslaget. Då är för första gången **if**-satsens villkor ***namn == ' '** uppfyllt.

Därför utförs `cout`-satsen som står i den. Man kan undra hur det står till med den övergripande `while`-satsens villkor vid denna tidpunkt. Det är ju just motsatsen till `if`-satsens villkor. Faktum är att `while`-satsens villkor inte är uppfyllt vid denna tidpunkt. Men det gör inget eftersom det var uppfyllt när det testades och vi kom in i loopens sista varv. Anledningen till det är att mellan `while`-villkorets test och `if`-villkoret står pekarstegningen `namn++` oavsett om den är inbakad i `while`-villkoret eller separat i början av loopen. Mellan `while`- och `if`-villkoret flyttas pekaren ett steg. Dvs `while` testas när pekaren pekar på bokstaven `n` och är skilt från mellanslag. `if` testas när pekaren pekar på mellanslag. Efter `if` är `while`-satsens villkor falskt och loopen avbryts. Det är loopens sista varv som skriver ut den andra initialen. Det är därför `while`-satsen endast går igenom förnamnet. Vi behöver inte bry oss om när strängen tar slut. Nolltecknet behöver därför inte användas som avslutningsvillkor i `Initials`.

Arrayens minnesbild

I `while`-satsens sista varv har vi följande bild av arrayen. Då pekar nämligen pekaren `namn` på `n`. Men sedan stegas pekaren med 1, så att den pekar på mellanslaget när `if`-satsen testas och `A` skrivs ut som pekaren `namn+1` pekar på:

T	a	i	f	u	n		A	l	i	...



Array vs. pekare

Vad händer om man hela funktionen `initials()` skrivs om till arraynotation?

```
void initials(char namn[])
{
   int i=0;
   cout << namn[0];
   while (namn[i++] != ' ')
      if (namn[i] == ' ')
         cout << namn[i+1];
}
```

Vid denna omskrivning kan resten av programmet `Initials` dvs koden i `main()` bibehållas oförändrad och kommer att fungera utan problem – ett resultat av modularisering. Jämför man båda varianter kan man konstatera att arrayvarianten använder en extra indexvariabel `i` som inte behövs i pekarvarianten. Annars är det i princip smaksak att favorisera den ena eller den andra. Arrayvarianten har fördelen av bättre läslighet. Pekarvarianten kan tänkas föredras p.g.a. sin kompakthet och elegans av folk som har lite längre erfarenhet av programmering.

Övningar till kapitel 6

6.1 Deklarera en vanlig **`int`**-variabel **`a`** och en pekarvariabel **`p`**. Låt **`p`** peka på **`a`**. Läs in ett värde till **`a`**. Skriv ut båda variablernas värden. **`a`**:s värde har du läst in. Men varifrån har **`p`** fått sitt värde? Vad har variabeln **`p`** för datatyp?

6.2 Deklarera en vanlig variabel **`a`** och en pekarvariabel **`p`**, låt **`p`** peka på **`a`** och läs in ett värde till **`a`**. Skriv ut **`a`**:s värde en gång direkt och en annan gång via pekaren.

6.3 Deklarera vanliga variabler **`a`**, **`b`**, **`sum`** och pekarvariabler som pekar på resp. variabel, läs in värden till **`a`** och **`b`** via pekarna. Addera **`a`** och **`b`**:s värden med pekarna, lagra resultatet i **`sum`** och skriv ut det med hjälp av pekaren.

6.4 Lös övn 10.3 ovan utan en enda vanlig variabel, endast med pekare.

6.5 Modularisera lösningen till övn 10.4 ovan genom att flytta additionen till en funktion. Genomför följande experiment: Nollställ summan i **`main()`**, skicka den som pekare via en parameter till funktionen. Ändra summan i funktionen och skriv ut summan från **`main()`**. Ger denna utskrift **`0`** eller får man tillbaka det värdet som ändrades i funktionen, i **`main()`**?

Skriv om hela programmet samt funktionen till ett program med vanliga variabler, utan pekare. Upprepa experimentet ovan. Blir det någon skillnad?

6.6 Skriv en pekarversion av **`ArrayChar`**. Lägg till efter **`sizeof`** som mäter strängens minnesutrymme, kod som med **`strlen()`** även mäter strängens längd. Varför returnerar **`sizeof`** och **`strlen()`** olika värden?

6.7 Skriv en pekarversion av programmet **`NULLcharacter`**, Skapa pekaren med **`const char *text = new char;`** och ersätt överallt *index* med *pekarposition*.

6.8 Skriv ett program som läser in en sträng med pekare och skriver ut den i omvänd ordning.

6.9 Skriv ett program som läser in en sträng, lagrar den i en pekare-till-**`char`**, byter plats på första och sista tecknet samt skriver ut strängen före och efter bytet.

6.10 Modularisera lösningen till övn 10.8 med *arraynotation*, dvs skriv ett program som läser in en sträng med en *array av char* och skickar den till en funktion som vänder om den. Skriv ut den omvända strängen från **`main()`** där funktionen anropas. Programmet ska visa referensanrop.

6.11 Skriv en *pekarversion* av lösningen till övn 10.10, dvs skriv ett program som läser in en sträng med en *pekare till char* och skickar den till en funktion som vänder om den. Skriv ut den omvända strängen från **`main()`** där funktionen anropas. Programmet ska visa referensanrop.

Kapitel 7

Fördjupning i

C++ programmering

	Ämne	Sida	Program
7.1	Array som parameter i funktioner	188	**RefArray**
	- Referensanrop med array	190	
7.2	Sökning och sortering	192	**RandArray**
	- Slumptal i en array	193	**SearchTest**
	- Minimax-problemet	196	**Minimax**
	- Namngivna konstanter och skalbarhet	198	**MinimaxTest**
	- Bubbelsortering	199	**BubbleTest**
7.3	Templates	203	**t_out()**
	- Definition av template funktioner	203	**t_BubbleTest**
	- Templates = Generics	203	
	- Template funktion för bubbelsortering	206	**t_sort()**
7.4	Dynamisk minnesallokering	208	
	- Datorns interna minneshantering	208	
	- Dynamisk array	210	**Dynamic**
	- Operatorerna **new[]** och **delete[]**	212	
7.5	Dynamisk filkryptering	215	**DynEncryptFil**
7.6	2D arrays	220	**2DArray**
7.7	2D array som parameter i funktioner	224	**TableFile**
	- Tabellhantering i filer	226	**setTable**
7.8	Mer om arrays och vektorer		
Övningar till kapitel 7		237	

7.1 Array som parameter i funktioner

Array som bearbetar större datamängder ger upphov till mer komplexa och sofistikerade program. Exempel på det är applikationer som söker, sorterar eller krypterar data. Vi kommer i fortsättningen att behandla enkla varianter av sådana program. Modularisering är lösningen: Helst vill man ha program som består av ett antal enkla, överskådliga funktioner där varje funktion löser ett specifikt problem. Sedan vill man sätta ihop dem dvs anropa dem med ett antal parametrar från **`main()`** och kontrollera hela händelseförloppet från denna funktion som helst ska ha så lite kod som möjligt. Ju mer avancerade datatyper man använder i sitt program desto större blir behovet av modularisering. Det är möjligt att skicka en array som parameter till en funktion. Man kan alltså deklarera en array i parameterlistan, men inte att definiera en funktion med returtypen array. I nästa programexempel (nästa sida) definieras en funktion med en array av **`int`** som parameter.

```
// RefArray.cpp
// Skickar en stor array till en funktion
// Jämför arrayens adress och minnesstorlek i main() med
//                 adress och minnesstorlek i funktionen
// Array som parameter i en funktion behandlas som referens
// Parameteröverföringen är referensanrop: adressen skickas
#include <iostream>
using namespace std;

void funk(int b[])                   // Array som parameter
{
   cout << "\n\nI funktionen\nlagras arrayen vid adressen "
        << b << " och tar " << sizeof(b) << "    bytes\n"
        << "Arrayens sista element är före ändringen "
        << b[999];
   b[999] = 1                        // Sista element ändras
   cout << "\n\t\t     och efter ändringen " << b[999]
        << "\n\n";
}

int main()
{
   int a[1000] = {0};                // Array med 1000 nollor
   cout << "I main()\nlagras arrayen vid adressen " << a
        << " och tar " << sizeof(a) << " bytes\n"
        << "Arrayens sista element är före anropet   "
        << a[999];
   funk(a);                          // Anropet: arrayens adress
                                     // skickas till funktionen
   cout << "I main()\när arrayens sista element "
        << "efter anropet  " << a[999] << "\n\n";
}
```

I programmet ovan har vi i `main()` en array `a` som har 1000 element, alla ifyllda med värdet `0` – till att börja med. Detta har vi åstadkommit med en speciell kod i C++, nämligen `{0}` som kan användas för att initiera en array till `0`. Det går inte att använda denna kortform med ett annat värde än `0`. Vi vet att varje `int` tar 4 bytes i minnesutrymme. Därmed tar hela arrayen 4 000 bytes. När vi i `cout`-satsen med `sizeof(a)` mäter arrayens storlek borde vi få detta värde. En körning av `RefArray` visar detta:

```
I main()
lagras arrayen vid adressen 0012EFE0 och tar 4000 bytes
Arrayens sista element är före anropet   0

I funktionen
lagras arrayen vid adressen 0012EFE0 och tar 4    bytes
Arrayens sista element är före ändringen 0
                     och efter ändringen 1

I main()
är arrayens sista element efter anropet  1
```

Låt oss först gå in på med vilken syntax programmet `RefArray` använder en array som en parameter i en funktion.

1. Att definiera en funktion med array som parameter

har gjorts i funktionen `funk()` genom att deklarera den formella parametern som en array av `int` dvs samma datatyp som den aktuella parametern har i anropet:

```
int b[]
```

Antalet element inom hakparentesen kan, men behöver inte anges. Gör man det ändå kommer kompilatorn i alla fall att ignorera denna uppgift. Utelämnas arrayens storlek – som vi gör – slipper man godtycklighet i koden då kompilatorn tillåter – och ignorerar – alla positiva tal inom hakparentesen. Att antalet element inte behövs här beror på att en formell parameter får sitt initialvärde från den anropande funktionen. Även arraystorleken följer med vid anropet. Detta har i sin tur att göra med att hela definitionen av en funktion endast är en mall, en föreskrift om vad som ska hända om funktionen anropas, en potentiell kod som blir aktuell först när vi anropar funktionen. I funktionen `funk()` står definitionen av parametern `b` till datatypen array av `int` som vanligt i parameterlistan och därmed i funktionshuvudet:

```
void funk(int b[])
```

Här i funktionshuvudet är hakparentesen tom då det är en definition som står i parameterlistan. I funktionskroppen däremot är hakparentesen till arrayen `b` aldrig tom. Det beror på att hakparenteserna i icke-definitionssatser innehåller index, inte arrayens storlek. Index får aldrig utelämnas då det specificerar ett visst element i arrayen. Dessutom exekveras kroppens kod endast när funktionen anropas. I och med anropet följer aktuella värden till arrayens element med parametern så att index till den formella parameterns element inte bara är meningsfulla utan också nödvändiga. Ett icke-specificerat index ger alltid kompileringsfel.

2. Att anropa en funktion med array som parameter
sker genom att skriva den aktuella parametern som array utan hakparenteser i anropet:

`funk(a);`

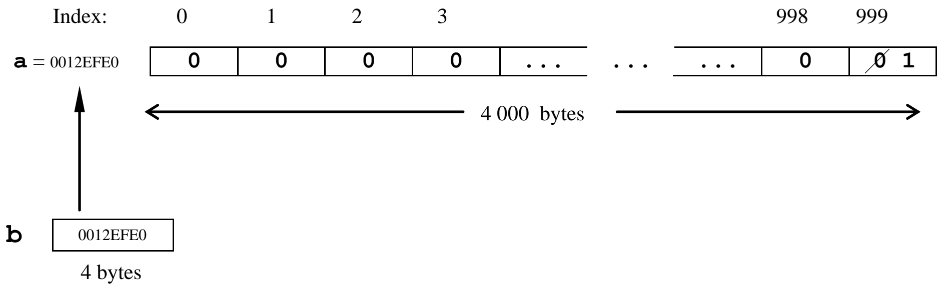
Anmärkningsvärt är att det för första gången dyker upp en array utan hakparenteser. Så, tittar man inte på definitionssatsen några rader ovan kan man inte känna igen **`a`** som array. Anledningen till att hakparentesen inte får stå efter arrayen **`a`** i anropssatsen är skillnaden i tolkningen av hakparentesen mellan definitions- och icke-definitionssatser. Anropssatsen är en icke-definitionssats. Följaktligen skulle hakparentesens innehåll tolkas som index och inte som arrayens storlek. Men index specificerar ett visst element i arrayen. En anropssats av typen **`funk(a[1000]);`** skulle skicka endast *ett element* av arrayen nämligen det med index **`1000`**. Det blir i så fall ett *tal* av typ **`int`** som skickas till funktionen, även om ett odefinierat sådant då det sist korrekt definierade indexet är **`999`** p.g.a. indexering börjar med **`0`**. Men man kommer att få kompileringsfel i alla fall då funktionens formella parameter är deklarerad som en *array* av **`int`** och inte som en vanlig **`int`**. Den enkla datatypen **`int`** kan inte konverteras till den sammansatta datatypen array av **`int`**. De automatiska typkonverteringsreglerna gäller endast för enkla datatyper. Det tänkbara alternativet **`funk(a[]);`** fungerar inte heller av samma anledning: Det handlar om en icke-definitionssats där hakparentesens innehåll tolkas som index. Men index får aldrig utelämnas (se punkt **1**). För att skicka en array som parameter till en funktion måste alltså arrayen i funktionsanropet skrivas endast med arraynamnet *utan hakparentes*. Självklart måste arrayen innan anropet vara definierad i **`main()`** som vanligt med hakparentes och en uppgift om storleken. Arraynamnet används vid anropet som adressen till arrayen.

Referensanrop med array

Körresultatet av **`RefArray`** på förra sidan avslöjar även en del intressanta nyheter för oss: När själva arrayvariabeln **`a`** utan hakparenteser skrivs ut får man det konstiga värdet **`0012EFE0`** som är ett hexadecimalt tal (decimalt: 1241056). Det är arrayen **`a`**:s adress. Adresser visas i datavärlden – det är en de facto-standard – som tal i hexadecimalt format. Med *adress* menas alltid en plats i datorns *RAM*-minne (*Random Access Memory*). När en array definieras lagras den vid en adress och arraynamnet blir en länk mellan programmet och denna fysiska adress. När arrayen **`a`** sedan i funktionsanropet **`funk(a)`** skickas som en aktuell parameter då överförs inte arrayens värden utan arrayens adress till funktionen **`funk()`**. Denna adress tas emot av den formella parametern **`b`** som är deklarerad i funktionens parameterlista som en array av **`int`**. På så sätt hamnar **`a`**:s adress, det hexadecimala talet **`0012EFE0`** i minnescellen **`b`**. Körresultatet ovan visar att **`b`** lagrar **`a`**:s adress och att ett adressvärde tar 4 bytes. Därmed pekar både **`a`** och **`b`** på en och samma array. Någon kopiering av arrayinnehållet på 4 000 bytes till en ny plats förekommer inte. Endast adressen på 4 bytes kopieras till **`b`** vid funktionsanrop. I **`main()`** kommer man åt

arrayen med **`a`** och i **`funk()`** gör man det med **`b`**. När vi sedan i **`funk()`** ändrar arrayens sista element med **`b`** från **`0`** till **`1`**, kan ändringen ses i **`main()`** med **`a`**.

Minnesbild till programmet **`RefArray`**:



Metoden för överföring av arrayparametrar är referensanrop. Dvs inte parametrarnas värden utan deras adresser överförs vid funktionsanropet. Datatypen till de formella parametrarna är referens i det ena och array i det andra fallet. Men det väsentliga är att det i båda fall är parametrarnas *adresser* som överförs och inte deras värden, vilket gör att ingen fördubbling av minnesåtgång förekommer och att ändringar i funktionen återspeglas i **`main()`**. Valet av parameteröverföringsmetod styrs av datatypen:

I C++ väljs automatiskt **referensanrop** (Call by reference) för parameteröverföring vid funktionsanrop, om parametern är av datatypen **array** eller **vector**.

7.2 Sökning och sortering

Ett viktigt – numera självklart – användningsområde för datorer är sökning i och sortering av stora datamängder. Programmeringstekniskt sett kan sådana applikationer inte skrivas utan arrayer (eller högre datatyper). Därför är sökning och sortering klassiska tillämpningar för sammansatta datatyper. Samtidigt ökar behovet av modularisering ju mer avancerade datatyper man använder i sitt program. Nu när vi lärt oss att skicka arrayer som parametrar till funktioner, kan vi modularisera program som använder arrayer. Detta är nödvändigt för att koncentrera sig på den egentliga uppgiften, nämligen sökning, sortering eller andra applikationer som t.ex. kryptering (kommer att tas upp i nästa avsnitt). När man söker eller sorterar data finns redan ett material i form av databaser, tabeller eller listor osv. som man använder. För att skaffa ett liknande underlag för våra testprogram har vi valt att låta slumptalsgeneratorn i C++ producera materialet och lagra det i en array.

Slumptal i en array

Eftersom vi i fortsättningen kommer att jobba med flera program som använder slumptal lagrade i en array vill vi skriva en funktion som kan användas av alla dessa program. Vi har valt formen av en **`void`**-funktion för att generera ett antal slumpvärden och tilldela dem till elementen i en array:

```
// RandArray.h
// Funktion som slumpar fram n styck heltal mellan 1 och 100,
// lagrar dem i en array och skriver ut dem. Anropar
// funktionen myRand() som ger ett slumptal i ett intervall

#include "MyRand.h"                // Innehåller myRand()

void RandArray(int no[], int n)
{
   cout << '\n' << n << " heltal mellan 1 och 100"
        << " slumpas fram:\n\n";
   for (int i=0; i<n; i++)
   {
       no[i] = myRand(1, 100);   // Slumptal mellan 1 och 100
       cout << no[i] << ' ' ;
   }
   cout << "\n";
}
```

För förståelse av funktionen **`myRand()`** och de nödvändiga biblioteksfilerna hänvisas till hantering av slumptal. Det nya i koden ovan är att slumptalen lagras i en array som kommer att användas av fler program vilket demonstrerar modularisering och återanvändning av kod. Filen ovan innehåller inte ett fullständigt program utan endast en funktion vilket även framgår av filnamnet **`RandArray.h`**. Filändelsen är inte **`cpp`**. Istället har vi en **`h`**eaderfil som kan inkluderas i och användas av andra program. I filen definieras **`void`**-funktionen **`RandArray()`** med två parametrar

varav den första är en array av **int**, kallad **no** som lagrar slumptalen och den andra en enkel **int**, kallad **n** som lagrar arrayens storlek. Arrayen deklareras i parameterlistan och tilldelas i kroppen **n** styck slumptal mellan **1** och **100** via satsen:

```
no[i] = myRand(1, 100);
```

som i en **for**-sats anropar den externlagrade funktionen **myRand()** som i sin tur i varje varv av loopen slumpar fram *ett* slumptal mellan **1** och **100**. Då vi använt denna funktion tidigare för ett annat program, står dess kod i filen **MyRand.h** som inkluderas i **RandArray()** med **#include "MyRand.h"**. **for**-satsen som anropar **myRand()** skriver ut de **n** stycken slumptal. **n** behövs som en andra parameter i **RandArray()** för att kunna avsluta **for**-satsen med villoret **i < n**. Sitt värde som är antalet arrayelement, får **n** när funktionen anropas. Det gör vi i början av **main()** i följande program:

```
// SearchTest.cpp
// Definierar en array och skickar den till den externlagrade
// funktionen RandArray() där den tilldelas 20 slumptal
// Den tilldelade arrayen skickas vidare till funktionen
// search() som söker efter ett inläst tal bland slumptalen
#include <iostream>
using namespace std;

#include "RandArray.h"          // Innehåller RandArray()
#include "Search.h"             // Innehåller search()

int main()
{
   srand(time(0));              // Skapar variation i slumpen
   int sokt, integer[20];
   RandArray(integer, 20);      // Anropar slumpfunktionen
   cout << "\tAnge tal som programmet ska söka efter:\t";
   cin >> sokt;                 // sokt = det sökta talet
   search(integer, 20, sokt);   // Anropar sökfunktionen
}
```

En array av **int** har definierats med **20** element. I anropssatsen **RandArray(integer, 20);** skickas både arrayen och storleken till funktionen. Det anmärkningsvärda är följande: När arrayen **integer** som aktuell parameter i anropet överförs till den formella parametern **no** i funktionen **RandArray()**, är den endast deklarerad men inte tilldelad dvs den är inte väl definierad. I den definierade arrayen **integer**:s minnesceller står oinitierade skräpvärden när den i anropssatsen skickas till funktionen **RandArray()**. Faktum är att, när parametern är en array, så används referensanrop där den aktuella parametern, arraynamnet **integer**, och den formella parametern, arraynamnet **no**, endast är två olika *adresser* till ett och samma minnesområde, till en och samma array. Med **integer** definierar vi arrayen i **main()** och anropar **RandArray()**. Med **no** tilldelar vi samma array värden i funktionen **RandArray()**. Efter funktionsanropet är arrayen både definierad och

tilldelad. En sådan ”arbetsdelning” mellan olika funktioner kan endast göras med referensanrop.

I **SearchTest** har modulariseringen drivits ännu vidare i och med programmets två huvuduppgifter – slumptalsgenerering och sökning – flyttats till separata externlagrade funktioner. I **main()** finns bara funktionsanrop och inläsning av det sökta talet kvar. Funktionen **RandArray()** för slumptalsgenerering lagras i filen **RandArray.h** och sökfunktionen **search()** lagras i filen **Search.h**. Dessutom använder sig **RandArray()** av funktionen **myRand()** som också är externlagrad i filen **MyRand.h**.

Efter anropet av slumpfunktionen läses in ett värde till variabeln **sokt** som tillsammans med arrayen **integer** och arrayens storlek **20** skickas till **search()**. När **search()** anropas är arrayen **integer** både definierad och tilldelad. Sökfunktionen får alltså väl definierade slumptalsvärden som överförs till den formella parametern **t**. Vid sidan om **no** är **t** nu en till minnescell som lagrar arrayen **integer**:s adress i detta program. Även den här parameteröverföringen sker med referensanrop. Vid anropet skickas inte värdena i arrayelementen till funktionen utan endast adressen som lagras i **integer**. I själva verket är det arrayens adress som överförs till **search()**, tas emot av **t** och används sedan i sökfunktionen för att hitta det sökta talet i arrayen:

```
// Search.h
// Funktion som tar emot tre parametrar:
// arrayen t, dess storlek n och heltalet s, det sökta ele-
// mentet. Söker efter den första förekomsten av s bland
// arrayelementen.

void search(int t[], int n, int s)
{
   int i;
   for (i = 0; i < n; i++)            // Söker igenom array t
       if (t[i] == s)                 // Sökkriteriet
       {
          cout << "\nDet sökta talet "  // Hittat-meddelande
               << t[i] << " är det " << i + 1
               << ":e elementet bland talen ovan\n";
          break;                        // Bryter for-satsen
       }
   if (i == n)
      cout << "\nDet sökta talet "    // Ej-hittat-meddelande
           << "finns ej bland talen ovan.\n";
}
```

Det sökta talet skickas med parametern **sokt** och tas emot av parametern **s**. Denna parameteröverföring sker däremot enligt värdeanrop eftersom parametrarnas datatyp är **int**. I C++ tillämpas automatiskt värdeanrop när parametern är en enkel datatyp. Respektive parameters datatyp är alltså avgörande för vilken överföringsteknik som

används vid ett funktionsanrop. Även den andra parametern **`n`** tar emot sitt konstanta värde **`20`** enligt värdeanrop.

Efter dessa progammeringstekniska anmärkningar ska vi titta på vad **`void`**-funktionen **`search()`** egentligen gör och hur den hittar eller inte hittar det sökta talet. Arrayen och det sökta talet är givna. Frågan är: finns det sökta talet i arrayen? Om ja, på vilken position? Algoritmen är väldigt rak och enkel varför den kallas för *linjär sökalgoritm*:

1. Gå igenom alla element i arrayen dvs sök igenom arrayen **`t`** från början till slutet (linjär sökning).

2. Jämför varje element med det sökta talet. Finns likhet med något element, skriv ut ett hittat-meddelande samt elementets position som är lika med index + 1. Har du hittat en likhet avbryt sökningen.

3. Har du gått igenom alla arrayelement utan att hitta någon likhet skriv ut ett ej-hittat-meddelande.

Denna algoritm hittar endast den första förekomsten av det sökta talet i arrayen och tar inte hänsyn till att det ev. kan finnas flera exemplar av det sökta talet i arrayen. Progammeringstekniskt har vi översatt algoritmens punkt **1** till C++ kod genom att i funktionen **`search()`** skriva en **`for`**-sats som söker igenom arrayen **`t`** från index **`0`** till **`19`**. I denna **`for`**-sats finns en **`if`**-sats som implementerar (förverkligar, kodar) punkt **2** och i sin tur innehåller två satser: Hittat-meddelandet och **`break`**-satsen. En **`break`**-sats avbryter alltid den loop eller den **`switch`**-sats i vilken den står, här alltså **`for`**-satsen. Det är den som enligt anvisningen i punkt **2** gör att programmet endast hittar den första förekomsten av det sökta talet i arrayen. I punkt **3**:s implementering – den sista **`if`**-satsen i **`search()`** – utnyttjar vi att **`for`**-satsens räknare (styrvariabel) **`i`** är väl definierad även *efter* **`for`**-satsen och att den har kvar det värde den fick där. Om sökningen gått igenom alla arrayelement utan att hitta något element som är lika med det sökta talet, har **`for`**-satsens räknare **`i`** nått värdet **`20`** då detta är första värdet som inte uppfyller **`for`**-satsens villkor **`i < n`**. I detta fall avslutas **`for`**-satsen utan **`break`** med värdet **`20`** för **`i`** så att villkoret till den efterföljande **`if`**-satsen blir uppfyllt och skriver ut ett Ej-hittat-meddelande. En körning av programmet **`SearchTest`** som inkluderar headerfilerna **`Search.h`** och **`RandArray.h`** kan se ut så här:

```
20 heltal mellan 1 och 100 slumpas fram:

36 85 91 42 39 96 86 39 78 45 54 49 3 70 2 73 8 80 50 39

Ange tal som programmet ska söka efter: 50

Det sökta talet 50 är det 19:e elementet bland talen ovan
```

Minimax-problemet

Från att hitta ett specificerat element i en array är det inte långt till att hitta det minsta och det största elementet i en array. Båda bygger på samma princip som kom fram i den linjära sökalgoritmen ovan. Programmeringstekniskt består den kortfattat av en **`if`**-sats i en **`for`**-sats där **`for`**-satsen söker igenom arrayen och **`if`**-satsen testar varje element med sökkriteriet som villkor. Skillnaden är dock att sökkriteriet vid vanlig sökning är en jämförelse av likhet medan sökkriteriet vid minimax-problemet – att hitta det minsta och det största elementet i en array – är en jämförelse av olikhet, mindre eller större än.

Vill man bestämma det minsta elementet i en array initierar man en variabel, säg **`min`** temporärt till arrayens första element och låter sedan **`min`** få andra värden beroende på om sökningen hittar mindre värden. Efter denna kod:

```
min = t[0];
for (int i=1; i < n; i++)
    if (t[i] < min)
       min = t[i];
```

kommer det minsta elementet i arrayen **`t`** att stå i variabeln **`min`**. Vill man hitta **`max`** istället för **`min`** byter man i koden ovan **<** mot **>** i sökkriteriet och **`min`** mot **`max`**:

```
if (t[i] > max)
```

Då kommer det största elementet i arrayen **`t`** att stå i variabeln **`max`**. Kan man inte lösa båda problem i en funktion? Det har vi gjort i funktionen **`extrema()`**:

```
// Minimax.h
// Funktion som hittar minsta eller största talet i en int-
// array. Tar emot tre parametrar: arrayen t, arrayens stor-
// lek n och en boolesk variabel som avgör om min eller max
// ska hittas

void extrema(int t[], int n, bool min)
{
   int minimax = t[0];              // minimax=min eller max
   if (min)                         // initieras till 1:a
      cout << "Det minsta av dem är\t";
   else
      cout << "Det största av dem är\t";
   for (int i=1; i < n; i++)        // Söker igenom array t
       switch (min)
       {                           // Hittar min:
          case true:  if (t[i] < minimax) minimax = t[i];
                      break;       // Hittar max:
          case false: if (t[i] > minimax) minimax = t[i];
                      break;
       }
   cout << minimax << "\n";
}
```

De första två parametrarna i funktionen **`extrema()`**, arrayen **`t`** och antalet element **`n`** är identiska med parametrarna i den vanliga sökfunktionen **`search()`**. Den tredje däremot, den booleska variabeln är ny. Parametern **`min`** är en sorts flagga som, när den är satt till **`true`**, bestämmer funktionen arrayen **`t`**:s minsta element och, när den är satt till **`false`**, hittar arrayens största element. Istället för **`if`**-satsen har vi en **`switch`**-sats som är inbakad i **`for`**-satsen och återspeglar den linjära sökalgoritmens struktur. **`switch`**-satsen avgör om arrayens minimala eller maximala värde ska hittas.

Initialt sätts den lokala variabeln **`minimax`** temporärt till arrayens första element. Sedan påbörjar **`for`**-satsen sin genomgång av arrayen. Då **`minimax`** redan är satt till det första elementet börjar **`for`**-satsen sin genomgång med det andra elementet. Räknaren börjar med **`i=1`**. I **`switch`**-satsens **`case`**s jämförs **`minimax`** med varje element. Vill man bestämma minimum sätts **`minimax`** till det aktuella elementets värde om detta är *mindre* än **`minimax`**. Vill man hitta maximum sätts **`minimax`** till det aktuella elementets värde om detta är *större* än **`minimax`**. Eftersom detta görs för alla element i arrayen, hamnar i variabeln **`minimax`** slutligen det minsta resp. det största arrayelementet. Även om det är ett tvåvägsval inbakat i **`for`**-satsen som hade kunnat formuleras med en **`if`**-**`else`**-sats, har vi istället valt att använda **`switch`**-satsen. Anledningen är att en **`if`**-**`else`**-sats kombinerad med **`case`**-satsernas **`if`**-satser producerar en nästlad struktur som genererar det luriga-**`else`**-fenomenet (sid 54) och tvingar oss att använda blockmarkering. Detta gör koden mindre läslig och överskådlig. **`switch`**-satsen ger en mer lättläst struktur.

Funktionen **`extrema()`** anropas från **`main()`** två gånger, varje gång med ett annat värde till den 3:e parametern, en gång **`true`** en gång **`false`**:

```
// MinimaxTest.cpp
// Definierar en array med namngiven konstant som storlek och
// skickar den till RandArray() för att fylla den med slump-
// tal. Anropar funktionen extrema() första gången för att
// hitta minsta, andra gången för att hitta största talet
// bland slumptalen.
#include <iostream>
using namespace std;

#include "RandArray.h"          // Innehåller RandArray()
#include "Minimax.h"            // Innehåller extrema()

int main()
{
  const int antal = 20;         // antal = namngiven konstant
                                // får inte ändra sitt värde
  int integer[antal];
  RandArray(integer, antal);    // Anrop av RandArray()
  extrema(integer, antal, true);// Anrop av extrema() för min
  extrema(integer, antal, false);                // för max
}
```

Namngivna konstanter och skalbarhet

En konstant är data som inte kan ändras. En *namngiven* konstant är en konstant med ett *namn* som används för att referera till den. I litteraturen finns det även beteckningen *symbolisk konstant* som betyder samma sak. Hittills har vi använt endast icke-namngivna konstanter som t.ex. heltalskonstanten **`9`** av typ **`int`**, decimaltalskonstanten **`3.5`** av typ **`double`**, teckenkonstanten **`'a'`** av typ **`char`**, strängkonstanten **`"Hallå"`** av typ **`string`** eller **`char[]`**, den logiska konstanten **`true`** av typ **`bool`** osv. Självklart behöver sådana konstanter inte definieras, ja de *kan* inte definieras. De skickas som de är – som värden – och man måste komma ihåg deras exakta värden när man vill sätta dem igen. Nackdelen med icke-namngivna konstanter är att man inte kan referera till dem i koden. En annan nackdel är när man i koden vill ändra alla värden på en icke-namngiven konstant. Har man t.ex. skrivit 25 på flera ställen i ett program måste man ändra alla dessa ställen. Är konstanterna namngivna behövs bara en ändring göras. Bl.a. därför finns i C++ alternativet att namnge konstanter. Av detta förstår man att namngivna konstanter måste definieras vilket görs med det reserverade ordet **`const`** som står för det engelska *constant*. Man namnger en konstant genom att i definitionen skriva **`const`** före eller efter datatypen, dock före namnet. I **`MinimaxTest`** har vi definierat den namngivna konstanten **`antal`** t.ex. så här och initierat den till **`20`**:

```
const int antal = 20;
```

Vill vi nu ändra i programmet **`MinimaxTest`** som består av tre funktioner, så att det skapar inte **`20`** utan **`1000`** element i arrayen **`integer`** och därmed **`1000`** slumptal samt hittar det minsta och det största bland *dem*, behöver vi byta ut **`20`** mot **`1000`** på *ett* ställe i hela programmet, nämligen i definitionssatsen ovan. I detta sammanhang pratar man om programmets *skalbarhet* – en viktig egenskap vid hantering av stora datamängder. Namngivna konstanter, speciellt när man använder dem som storlek för arrayer, underlättar att skriva *skalbara* program. Naturligtvis måste även resten av koden vara skriven på sådant sätt att den tål skalning, dvs att programmet fungerar felfritt när man ökar vissa konstanters värden.

Att namnge konstanter innebär förstås inte att vi kan överskriva deras värden med nya värden under *en* programkörning. Kompilatorn kommer att sätta stopp på varje försök att t.ex. efter **`const`**-definitionen av konstanten **`antal`** ovan skriva en sats av typ **`antal = 30;`** Konstanters värde får inte ändras dvs en konstant kan inte tilldelas på nytt. Observera att detta har absolut inget att göra med skalbarhet som vi diskuterade ovan, utan beror på att en **`const`**-definierad minnescell som t.ex.

antal	20

alltid är *read-only* dvs den kan bara läsas. Den är skrivskyddad, spärrad mot överskrivning. En direkt slutsats av det är att man *måste* initiera en namngiven konstant vid definitionen, man måste skriva definition och tilldelning i en och samma sats. Gör man inte det, utan endast definierar, hamnar som vi vet, ett odefinierat skräpvärde i den vilket inte kan överskrivas p.g.a. *read-only*-egenskapen.

I **MinimaxTest** använder vi för första gången ett namn istället för ett tal inom hakparentes vid definitionen:

```
int integer[antal];
```

antal måste vara namnet till en *konstant*. Kompilatorn tillåter inte en variabel för arrayens storlek. Antalet element till en statiskt allokerad array – om en sådan handlar det här – måste vara konstant, antingen en namngiven eller en icke-namngiven konstant.

I **MinimaxTest** skickas de **20** slumptal som genererats i funktionen **RandArray()** till funktionen **extrema()** två gånger. Första gången anropas **extrema()** med värdet **true** till den 3:e parametern så att det minsta talet bland slumptalen hittas. Andra gången anropas **extrema()** med **false** så att det största talet hittas. Även i dessa anrop tillämpas precis som i **70Search** referensanrop på den första parametern p.g.a. dess datatyp array. Alla tre variabler **integer**, **no** och **t** refererar därför till en och samma array som definieras i **main()**, tilldelas i **RandArray()** och bearbetas i **extrema()**.

En körning av programmet **MinimaxTest** som inkluderar headerfilerna **RandArray.h** och **Minimax.h** kan se ut så här:

```
20 heltal mellan 1 och 100 slumpas fram:

84 50 36 56 87 99 33 59 83 52 11 22 43 33 50 11 63 29 100 73

Det minsta av dem är    11

Det största av dem är   100
```

Bubbelsortering

Sökning i och sortering av stora datamängder är klassiska tillämpningar för sammansatta datatyper, speciellt för arrayer. Medan minimax-problemet var en variant av sökning som baserades på i princip på samma linjära sökalgoritm, bygger sortering på en ny algoritm även om den har vissa likheter med sökning. Vi ska fortsätta kapitlet om arrayer med en sorteringsalgoritm som är en vidareutveckling av algoritmen för platsbyte av *två* värden. Vi har i programmet **MiniSort** använt denna algoritm på två tecken:

```
if (letter1 > letter2)
{
   temp = letter1;
   letter1 = letter2;
   letter2 = temp;
}
```

Om tecknen står i fel ordning ska de byta plats. För att göra det läggs **`letter1`**:s värde undan i en tredje, temporär variabel **`temp`**. Sedan tar vi **`letter2`**:s värde och lägger det i **`letter1`**. Till sist läggs värdet i **`temp`** (som ju har mellanlagrat **`letter1`**:s värde) in i **`letter2`**. Illustrationen på sid 59 bör underlätta förståelsen av denna process. I själva verket beskriver den en algoritm för sortering av *två* värden. För att utvidga algoritmen till *flera* värden kopplar vi den till den linjära sökalgoritmen som vi använde för sökning. Principen där var en **`if`**-sats inbakad i en **`for`**-sats. **`for`**-satsen söker igenom värdena i en array och **`if`**-satsen innehåller sökkriteriet. När det gäller sortering måste **`if`**-satsen istället byta plats på två värden om de står i fel ordning. Denna **`if`**-sats har vi ju redan skrivit för två tecken (se ovan). Det gäller bara att formulera den för två arrayelement och stoppa in den i en **`for`**-sats:

```
for (i = 0;  i < n-1;  i++)
     if (t[i] > t[i+1])
     {
        temp = t[i];
        t[i] = t[i+1];
        t[i+1] = temp;
     }
```

där **`t`** är en array som innehåller värdena som ska sorteras och **`n`** antalet element i arrayen. När två på varandra följande arrayelement **`t[i]`** och **`t[i+1]`** står i oönskad ordning ska de byta plats där **`i`** genomlöper alla index. Man skulle kunna tro att problemet vore löst med detta. Men eftersom **`if`**-satsen endast testar om *två grannvärden* står i fel ordning och byter sedan plats på *dem*, räcker koden ovan inte till att sortera arrayen fullständigt, även om **`for`**-satsen söker igenom hela arrayen. Jämförelsen mellan två grannvärden tar inte hänsyn till värden som står längre bort. Ett experiment bekräftar detta: Om man tillämpar koden ovan på en array av 20 heltal som med funktionen **`RandArray()`** är utvalda ur intervallet [1, 100] får man följande resultat:

```
20 heltal mellan 1 och 100 slumpas fram:

75 2 24 94 30 88 10 42 50 54 27 47 45 83 34 86 67 66 14 14

De 20 slumptalen efter koden ovan:

2 24 75 30 88 10 42 50 54 27 47 45 83 34 86 67 66 14 14 94
```

Resultatet visar att sorteringen inte är klar, men att vi är på rätt väg. Arrayen är *delvis* sorterad. Bara om två grannvärden stod i fel ordning har de bytt plats och detta har gjorts löpande genom hela arrayen. Denna delsortering kallas för ett *pass* i en sorteringsalgoritm som är känd under beteckningen *bubbelsortering*. För att uppnå en fullständig sortering måste detta pass upprepas flera gånger vilket innebär att lägga in ovanstående **`for`**-sats i en ny **`for`**-sats som går igenom flera pass. I varje pass kommer en del värden att placera sig i rätt ordning. Metoden kan jämföras med

luftbubblor i vattnet som så småningom stiger upp till vattenytan. Därav namnet *bubbel*sortering.

Vi har implementerat bubbelsorteringsalgoritmen i den separatlagrade **`void`**-funktionen **`bubbleSort()`** som tar emot en array av **`int`** och dess storlek som parametrar:

```
// Bubble.h
// Sorterar n styck heltal lagrade i arrayen t med en algo-
// ritm (bubbelsortering) som baseras på algoritmen för
// platsbyte av två objekt i programmet

void bubbleSort(int t[], int n)
{
   int i, pass, temp;
   for (pass=0; pass<n-1; pass++)    // Sorteringspassen
       for (i=0; i<n-1; i++)         // Ett pass
           if (t[i] > t[i+1])        // Sortering i stigande
           {                         // ordning
              temp = t[i];           // Algoritm för platsbyte
              t[i] = t[i+1];         // av de två elementen
              t[i+1] = temp;         // t[i] och t[i+1]
           }
   cout << "De " << n << " slumptalen efter sortering:\n\n";
   for (i=0; i<n; i++)               // Sorterad utskrift
       cout << t[i] << ' ' ;
   cout << "\n\n";
}
```

Bubbelsorteringsalgoritmen består alltså av en **`if`**-sats inbakad i en nästlad **`for`**-sats där **`if`**-satsen implementerar algoritmen för platsbyte av två värden. Den inre **`for`**-satsen söker igenom arrayelementen, utför *ett* sorteringspass och den yttre **`for`**-satsen upprepar sorteringspassen. Funktionen **`bubbleSort()`** har förutom arrayen **`t`** som ska sorteras även arrayens storlek **`n`** som parameter då den används i den inre **`for`**-satsen. Funktionen som inkluderas i programmet **`BubbleTest`** anropas från **`main()`** efter definitionen av arrayen **`integer`** och dess tilldelning i funktionen **`RandArray()`**:

```
// BubbleTest.cpp
// Definierar en array med namngiven konstant som storlek och
// skickar den till RandArray() för att fylla den med slump-
// tal. Anropar sedan funktionen bubbleSort() som sorterar
slumptalen
#include <iostream>
using namespace std;

#include "RandArray.h"
#include "Bubble.h"
```

```
int main()
{
   const int antal = 20;
   int integer[antal];
   RandArray(integer, antal);
   bubbleSort(integer, antal);

}
```

En körning av programmet **`BubbleTest`** som inkluderar headerfilerna **`Bubble.h`** och **`RandArray.h`** visar att sorteringen nu genomförts fullständigt:

```
20 heltal mellan 1 och 100 slumpas fram:

42 65 69 32 72 13 92 77 22 38 62 62 30 26 24 84 41 85 4 50

De 20 slumptalen efter sortering:

4 13 22 24 26 30 32 38 41 42 50 62 62 65 69 72 77 84 85 92
```

Andra algoritmer

Som en sista anmärkning till kapitlet sökning och sortering bör påpekas att de algoritmer som avhandlats här, är enkla och elementära. De är däremot inte de mest effektiva när det gäller att minimera antalet operationer och maximera snabbheten. Det finns effektivare (och mer komplicerade) algoritmer både när det gäller sökning och sortering som vi inte tar upp här. Vi nämner bara en algoritm som kallas *binärsökning* som heter så för att den i varje steg halverar arrayen man ska söka i. Den behöver ett mindre antal operationer och är därmed snabbare. När det gäller sortering finns den effektiva algoritmen *Quicksort* som bygger på *rekursion*.

7.3 Templates

I vanligt språk betyder ordet *template* (eng.) *mall* eller *schablon* på svenska. Man kan bilda mallar för funktioner eller för klasser. Istället för funktions- eller klassmall använder vi språkbruket *template funktion* eller *template klass*, dvs tolkar substantivet template som ett adjektiv. Lika bra skulle man kunna prata om *generaliserade* funktioner. En annan lämplig beteckning vore *grupper* av funktioner.

I C++ är **`template`** ett reserverat ord (keyword) och står för ett nytt programmeringstekniskt koncept som introducerar en högre abstraktionsnivå i språket, dvs ytterligare en generalisering av koden. Själva konceptet kallas för *Templates*.

Varför Templates?

Vilken praktisk nytta har man av en sådan generalisering? Låt oss säga, vi har ett program för sortering av heltal. Algoritmen – ofta implementerad i en funktion – skiljer sig inte från algoritmer för sortering av decimaltal, bokstäver, strängar, eller andra objekt. Datatypen är för algoritmens funktionalitet oväsentligt. Det vore slöseri med resurser om man skrev flera program för sortering av var och en av dessa olika datatyper. Idealt vore att ha *ett* sorteringsprogram för alla möjliga datatyper oavsett om de är heltal, decimaltal, bokstäver eller strängar. Men då måste funktionen som implementerar sorteringsalgoritmen vara så pass generell att den kan anropas för alla möjliga datatyper, beroende på i vilket syfte den anropas. Lösningen är att *datatyperna* till funktionens parametrar måste vara *variabla*.

Templates = Generics

I de flesta programmeringsspråken kallas motsvarigheten till C++:s Templates för *Generics*. Över lag har man infört det nya konceptet som ett tillägg till standarden först i de senare versioner av språken. I C++ kom Templates först på 90-talet. I Java introducerades Generics 2004. I C# har det funnits stöd för Generics sedan 2005. Genom att använda *Generics (Templates)* kan olika varianter av ett program förenas i ett och samma program som då innehåller *generiska (template)* moduler.

Definition av template funktioner

I programmering är variabler → platshållare för värden.
Med ***templates*** kan variabler användas även som platshållare för ***datatyper***.

Template funktioner är funktioner vars parametrar har variabla datatyper.
Ex.: I funktionen **`void t_out(T a[])`** är parametern **`a`**
en array av typ **`T`**, där **`T`** är en variabel för datatyper som kan ersättas
av vilken datatyp som helst: **`int, double, char, string, … .`**
`T` definieras med koden: **`template <typename T>`**

I template funktioner (eng.: *Function Templates*) är de formella parametrarnas datatyper inte specifierade. De bestäms först när funktionerna anropas, närmare bestämt av de aktuella parametrarnas datatyper. Detta innebär en generalisering som i C++ implementeras med *template* och även kan tillämpas på klasser. Man kan skriva *ett* program för många tillämpningar. Programmet **`t_BubbleTest`** (sid 205) är ett exempel på en generalisering av det traditionella programmet **`BubbleTest`** (sid 201). Medan **`BubbleTest`** med hjälp av den vanliga funktionen **`bubbleSort()`**e endast sorterar heltal, kan dess template (generiska) variant **`t_BubbleTest`** med template funktionerna **`t_out()`** och **`t_sort()`**, sortera både heltal, decimaltal, bokstäver och strängar.

I det här avsnittet behandlas endast template funktioner. Följande exempel implementerar template funktionen **`t_out()`** som skriver ut en arrays element oberoende av deras datatyper. Längre fram kommer vi att presentera även template funktionen **`t_sort()`** som implementerar bubbelsorteringsalgoritmen så att den kan tillämpas på variabla datatyper.

```
// t_Output.h
// Template funktionen t_out() skriver ut arrayen a:s element
// är en array av typen T där T är en variabel för datatyper
// som kan ersättas av int, double, char, string, ...
using namespace std;

template <typename T>        // Definierar typen T som ett
                             // variabelt namn för datatyper
void t_out(T a[])            // Använder T för att deklarera
{                            // arrayen a
   cout << ("\t");
   for (int i = 0; i <= 8; i++)
      cout << a[i] << "  "; // Skriver ut arrayen a:s element
   cout << "\n";
}
```

Det som gör funktionen **`t_out()`** till en template funktion är definitionen av namnet **`T`** till variabeln för datatypen, kallad **`typename`**, och funktionshuvudet:

```
void t_out(T a[])
```

där **`T`** används för att deklarera parametern **`a`** till en array av den variabla datatypen **`T`**. Till skillnad från vanliga funktioner har denna funktion en formell parameter av typ **`T`**, där **`T`** bestäms när funktionen anropas. Detta görs sedan i programmet **`t_BubbleTest`** nedan, med anropet:

```
t_out(hel);
```

`T` får den datatyp som i det anropande programmet har tilldelats arrayvariabeln **`hel`**. Har vi t.ex. deklarerat **`hel`** som en array av **`int`**, så antar den formella parametern **`T`** den aktuella parametern **`int`** och därmed **`T a[]`** array av **`int`**. I template funktioner finns det alltid ett sådant **`typename`**. I programmet **`t_BubbleTest`** nedan där vi testar template funktioner, anropas **`t_out()`** åtta gånger, 2 x 4 gånger med en annan

datatyp, närmare bestämt med **`int`**, **`double`**, **`char`** och **`string`**. Med hjälp av dessa datatyper bildas sedan arrays av **`int`**, **`double`**, **`char`** och **`string`** med koden **`T a[]`**. Den vanliga parametern **`a`** definieras då till sådana arrays.

Här följer nu programmet **`t_BubbleTest`** som anropar de template funktionerna **`t_out()`** och **`t_sort()`**, men i övrigt inte skiljer sig från ett vanligt program:

```
// t_BubbleTest.cpp
// Skapar 4 arrays av olika typer: int, double, char, string
// och skickar dem till t_out() för utskrift och till
//                         t_sort() för sortering
// Template funktionerna anropas som vanliga funktioner
// Utskrift sker före och efter sortering
#include <iostream>
#include "t_Output.h"     // Template funktionen t_out()
#include "t_Bubble.h"     // Template funktionen t_sort()
using namespace std;

int main()
{
  int hel[] = {9, 7, 2, 1, 8, 5, 4, 3, 6 };
  double deci[] = {9.9, 7.7, 2.2, 1.1, 8.8, 5.5, 4.4, 3.3,
                                                          6.6 };
  char boks[] = {'h', 'c', 'f', 'a', 'e', 'i', 'b', 'd', 'g'};
  string text[] = {"zeta", "beta", "gamma", "psi", "alpha",
                   "delta", "omikron", "lambda", "sigma" };
  cout << "\n\tFyra olika datatyper skrivs ut med samma"
       << " funktion\n\tFÖRE SORTERING:\n";
  t_out(hel);             // Osorterad uskrift
  t_out(deci);            // Anrop av template
  t_out(boks);            // funktionen t_out()
  t_out(text);
  cout << "\n\tDe fyra typerna sorteras med samma funktion"
       << " och skrivs ut";
  t_sort(hel);            // Sortering: Anrop av
  t_sort(deci);           // template funktionen
  t_sort(boks);           // t_sort()
  t_sort(text);
  cout << "\n\t\tEFTER SORTERING:\n";
  t_out(hel);             // Sorterad utskrift
  t_out(deci);
  t_out(boks);
  t_out(text);
}
```

Den vitmarkerade koden visar åtta anrop av template funktionen **`t_out()`**. Det anmärkningsvärda är att dessa anrop inte skiljer sig alls från anrop av vanliga funktioner. De aktuella parametrarna **`hel`**, **`deci`**, **`boks`** och **`text`** är definierade som arrays av **`int`**, **`double`**, **`char`** resp. **`string`** och skickar, när de anropas, inte bara sina vanliga värden – heltalen, decimaltalen, bokstäverna och strängarna – till de anropade funktionerna, utan även sina datatyper. Medan de vanliga värdena i resp.

array går till den formella parametern **`a`** i resp. funktions runda parameterlista, går datatyperna arrays av **`int`**, **`double`**, **`char`** och **`string`** till parametern **`T`**. Därmed blir varje datatyp specificerad och insatt på alla ställen där **`T`** står i template funktionen, vare sig i huvudet eller i kroppen.

Så här blir resultatet av en körning av programmet **`t_BubbleTest`**:

```
Fyra olika datatyper skrivs ut med samma funktion
FÖRE SORTERING:
9  7  2  1  8  5  4  3  6
9.9  7.7  2.2  1.1  8.8  5.5  4.4  3.3  6.6
h  c  f  a  e  i  b  d  g
zeta  beta  gamma  psi  alpha  delta  omikron  lambda  sigma

De fyra typerna sorteras med samma funktion och skrivs ut
EFTER SORTERING:
1  2  3  4  5  6  7  8  9
1.1  2.2  3.3  4.4  5.5  6.6  7.7  8.8  9.9
a  b  c  d  e  f  g  h  i
alpha  beta  delta  gamma  lambda  omikron  psi  sigma  zeta
```

Som man ser har heltalen, decimaltalen, bokstäverna och strängarna dvs värdena i de fyra olika arrays skrivits ut som ett resultat av de vitmarkerade anropen i programmet **`t_BubbleTest`** på förra sidan. Alla fyra anrop har gått till en och samma template funktion **`t_out()`** (sid 204) som skriver ut dem. Visserligen behöver man skriva fyra olika anrop i programmet **`t_BubbleTest`**. Men man behöver definiera och koda själva funktionen bara en gång, vilket innebär en stor effektivitet i utvecklingsarbetet.

Template funktion för bubbelsortering

Körresultatet ovan har även andra delar. Efter att värdena skrivits ut, skickas de till template funktionen **`t_sort()`** som sorterar dem med anropen:

```
t_sort(hel);
t_sort(deci);
t_sort(boks);
t_sort(text);
```

Även dessa anrop kan man inte skilja från anrop till vanliga funktioner, fast **`t_sort(T a[])`** är en template funktion. Efter sorteringen skickas arrayvärdena igen till utskrift, så att vi ser dem sorterade i utskriften ovan – och detta sker inte bara för hel- och decimaltalen samt bokstäverna utan även för strängarna. Även här använder vi oss av en enda template funktion som vi nu ska titta närmare på.

I förra avsnitt implementerades bubbelsorteringsalgoritmen med en vanlig funktion. Följande kod implementerar den med en template funktion:

```
// t_Bubble.h
// Template funktionen t_sort() sorterar en array av variabel
// datatyp T som kan vara int, double, char eller string
using namespace std;

template <typename T>       // Definierar typen T som variabel

void t_sort(T a[])          // Använder T för att deklarera
{                           // arrayen a
   T temp;
   for (int pass = 0; pass <= 8; pass++)
      for (int i = 0; i <= 7; i++)
         if (a[i] > a[i + 1])
         {                                // Sortering i sti-
            temp = a[i];                  // gande ordning
            a[i] = a[i + 1];              // Algoritm för
            a[i + 1] = temp;              // platsbyte
         }
}
```

Funktionen **t_sort()** är en template variant av den vanliga funktionen **sort()** i klassen **Bubble** som presenterades när vi behandlade sökning och sortering. Här gäller samma som vi sa om funktionen **t_out()**: Den formella parametern **T** står för datatyper som är kopplade till den aktuella anropsparametern som skickas till den vanliga formella parametern **a**, dvs för datatyperna till de objekt som ska sorteras.

7.4 Dynamisk minnesallokering

Dynamisk minnesallokering är den viktigaste tillämpningen av pekare och innebär en helt ny organisation av datorns interna minneshantering som allokerar minne vid programmets *exekvering*, medan statisk minnesallokering genomförs vid *kompilering*. Anta att vi vill ha ett program som läser data – text, tal, bild eller ljud – från en fil och vi vet inte hur mycket data filen innehåller, när vi skriver kod. Det här problemet kan inte lösas med statisk minnesallokering där man före exekveringen – dvs när man skriver kod – måste fatta beslut om hur mycket utrymme som behövs. Statisk minnesallokering som kodas med vanliga variabler kan inte modifieras vid exekvering. När man läser data från en fil ska minnesallokeringen helst göras samtidigt som filen läses dvs under programmets körning – ett exempel på dynamisk minnesallokering. I det enklaste fallet ska man kunna läsa in text till ett C++ program utan att på förhand behöva ange textens storlek. En vanlig array klarar inte av detta då den automatiskt tillämpar statisk minnesallokering. Lösningen är dynamisk minnesallokering som varken kan åstadkommas av referens eller någon annan datatyp, utan endast av pekare. Programmet **`Dynamic`** (sid 210) använder en *dynamisk array* som läser in och lagrar text utan att i förväg kräva längden på den. För att kunna förstå hur det går till är det bra att veta lite mer om datorns interna minneshantering. Därför öppnar vi här en parentes:

Datorns interna minneshantering

Det finns sex olika platser där datorn lagrar data när ett program kompileras och exekveras. De har olika egenskaper och hanterar minnesallokeringen på olika sätt:

1. **Register** är datorns snabbaste minne då det är lokaliserat i processorn. C++ satsen

   ```
   register int r;
   ```

   talar om för kompilatorn att skapa en **`int`**-variabel **`r`** med snabbast möjliga åtkomst. Då försöker kompilatorn att allokera ett register i processorn. Det finns ingen möjlighet att kontrollera om placeringen lyckats. Antalet register är begränsat. Endast lokala variabler kan definieras som registervariabler.

2. **Stack** är datorns snabbaste minne inom RAM och det näst snabbaste efter register. Det är konstruerat som en stack: Processorns stackpekare åker ned i stacken för att allokera nytt minnesutrymme och upp för att frigöra minnesutrymmet. C++ kompilatorn måste ha exakt information om minnesutrymmets storlek och livslängd för att generera den kod som behövs för att skicka stackpekaren upp och ned. Därför är denna typ av minnesallokering oflexibel och vi har kallat den statisk. Alla vanliga variabel- och arraydefinitioner leder till denna statiska minnesallokering på stacken som sker under kompilering.Variabler lagrade på stacken kallas även *automatiska* variabler och kan explicit definieras som:

   ```
   auto int a;
   ```

där det reserverade ordet **`auto`** kan utelämnas. Vanliga variabler är by default **`auto`**. Vi har i alla våra exempel utnyttjat utelämningsmöjligheten. En automatisk variabel lever antingen så länge livslängden p.g.a. blockstruktur tillåter det eller tills programkörningen är avslutad. Efter livslängden frigörs minnet automatiskt. Det finns ingen annan möjlighet att frigöra minnet för en automatisk variabel.

3. **Heap** är den del av RAM som kompilatorn inte har tillgång till och som är reserverad för att allokera minne med **`new`** under programmets exekvering. Heapen – som även kallas det *fria* minnesområdet – behöver inte ha information om minnesutrymmets storlek och livslängd. Man kan allokera minne så länge det finns utrymme på heapen. När minnet är slut avbryter den nya C++ standarden programmet med felmeddelandet *Abnormal program termination*. Man säger programmet kastar ett undantag. Äldre C++ installationer låter **`new`** returnera nollpekaren när heapen är full. För att undvika detta kan **`delete`** användas som är den till **`new`** motsatta (inversa) operatorn och frigör det minne som allokerats med **`new`**. Denna flexibilitet har förstås sitt pris: Heapen är långsammare än stacken. Både **`new`** och **`delete`** behöver längre tid för att operera på heapen än vanlig definition behöver för minnesallokering på stacken. Då **`new`** alltid returnerar en adress kan denna dynamiska minnesallokering endast göras med pekare:

   ```
   int *p = new int;
   ```

   I Windows läggs data temporärt i en växlingsfil på hårddisken när heapen är full.

4. **Static** har inget att göra med punkt **2** utan är ett fast minnesområde i RAM för variabler som ska leva under hela programmets gång. Står satsen

   ```
   static int s;
   ```

   i en funktion kommer den lokala variabeln **`s`** att gälla inte bara inuti funktionen när den anropas, utan leva vidare även efter funktionsanropet, så länge programmets körs. Utan det reserverade ordet **`static`** är livslängden av **`s`** begränsad till funktionen. Med **`static`** beter sig **`s`** delvis som en global variabel utan att vara en sådan.

5. **Constant** är read-only memory (ROM)-delen av RAM som används för lagring av konstanter. På så sätt är de skyddade mot överskrivning. Syntaxen är känd:

   ```
   const int c;
   ```

6. **Non-RAM** är en lagringsplats utanför programmets kontroll. Vissa jobb i datorn, även kallade processer eller trådar, kan ursprungligen vara initierade av ett C++ program, men körs vidare även efter programmets avslutning. Dataströmmar på väg till en annan dator som förekommer vid filöverföring är exempel på non-RAM lagring vars behandling ligger utanför denna boks ramar.

De reserverade orden **`register`**, **`auto`**, **`static`**, **`const`** som står framför datatypen i definitionen bestämmer lagringssättet och kallas därför *allokeringsmodifierare*. De avgör *hur* allokeringen ska genomföras. Självklart bestämmer även **`new`** att

allokering ska ske på heapen men syntaxen skiljer sig. **new** är ingen modifierare utan en operator. Man ser att **new** är något speciellt, kvalitativt annorlunda än allokeringsmodifierarna. Det nya är att allokeringens storlek kan ändras i programmet och att allokeringens livslängd bestäms av operatorn **delete**. Ett av **new** allokerat minne lever på heapen tills man frigör det genom att explicit skriva **delete**. Utan **delete** är minnet upptaget så länge programmet körs.

Dynamisk array

Efter parentesen om datorns interna minneshantering ska vi nu ägna oss åt dynamisk array som i följande program demonstreras med hjälp av operatorerna **new[]** och **delete[]**, arrayutvidgningar av de enkla operatorerna **new** och **delete**.

```
// Dynamic.cpp
// Läser in text utan att på förhand ange storleken & lagrar
// den i en dynamisk teckenarray vars storlek är variabel
// Förstorar och frigör arrayens minne efter behov
// Dynamisk minneshantering med new[] och delete[]
#include <iostream>
#include <conio.h>                    // Krävs för _getche()
using namespace std;
int main()
{
  int antal = 0;
  char letter;
  char *pekOld = NULL;                // Nollpekarinitiering
  cout << "\nMata in text utan att ange storleken"
       << " och tryck på Enter: \n\n\t\t";
  do
  {
    char *pekNew = new char[antal+1]; // 1. Ny dynamisk array
                                      // 1 byte större än
                                      // förra, men tom
     for (int i=0; i<antal; i++)      // 2. Kopierar gammal
         pekNew[i] = pekOld[i];       // data till ny array
     delete[] pekOld;                 // 3. Frigör gammal array
     pekOld = pekNew;                 // 4. Ompekning till den
                                      // nya arrayen
     letter = _getche();              // Läser 1 tecken
     pekOld[antal] = letter;          // 5. Lägger till tecken
                                      // i array: antal=index
     antal++;
  } while(letter != '\r');

  cout << "\n\nDin text består av " << antal-1 << " tecken "
       << "och har lagrats i en dynamisk array:\n\n\t\t";
  for (int j=0; j<antal-1; j++)
      cout << pekOld[j];
  cout << "\n";
}
```

Vad händer i programmet `Dynamic` *?*

Programmet **`Dynamic`** vinner på att använda dynamisk minnesallokering då man inte bara kan frigöra utan även ”förstora” dynamiskt allokerat minnesutrymme på heapen. På så sätt kan man läsa in text utan att på förhand ange textens storlek genom att hålla på med att mata in tecken för tecken. Ett körexempel ger följande utskrift:

```
Mata in text utan att ange storleken och tryck på Enter:

      abcdefghijklmnopqrstuvwxyz: Detta är alfabetet, men ...

Din text består av 55 tecken och har lagrats i en dynamisk array:

      abcdefghijklmnopqrstuvwxyz: Detta är alfabetet, men ...
```

I princip skulle man kunna mata in hur mycket text som helst. Så länge man inte trycker på Enter lagras texten i en ”växande” array. Vi kommer att se att det inte handlar om samma array. I varje varv av **`do`**-loopen allokeras en annan dynamisk array, medan den gamla tas bort. Men hur görs ”förstoringen” av arrayen? Programmet löser problemet genom att i varje varv av **`do`**-loopen läsa in endast *ett* tecken och lagra det på följande sätt:

1. Skapa en ny tom dynamisk array som är en byte – storleken för en **`char`** – större än arrayen i förra varvet. **`pekNew`** pekar på den.
2. Kopiera de tecknen som hittills dvs i förra varven lästs in från den gamla arrayen som **`pekOld`** pekar på, till den nya.
3. Frigöra den gamla arrayen som **`pekOld`** pekar på.
4. Peka om den gamla arrayens pekare till den nya arrayen. Båda pekar nu på denna array.
5. Lägga till det nyinlästa tecknet på aktuell position i arrayen.

I nästa varv kommer **`pekNew`** att få ett nytt minnesutrymme tilldelad enligt punkt 1 ovan så att endast **`pekOld`** kommer att pekar på den numera gamla arrayen som sedan enligt punkt 2 ovan kommer att kopieras till **`pekNew`**:s nya array osv.

Variabel arraystorlek

En av den statiska arrayens begränsningar är att antalet element måste vara konstant. Konstanten kan vara namngiven eller icke-namngiven. Kompilatorn tillåter inte en variabel för storleken av en statiskt allokerad array, se program **`MinimaxTest`**. Denna begänsning faller bort när vi använder en dynamisk array som i programmet **`Dynamic`** allokeras med satsen:

```
char *pekNew = new char[antal+1];
```

Operatorn **`new[]`** (se nedan) allokerar minne för en array av **`char`** vars storlek är **`antal+1`** där **`antal`** enligt definition är en **`int`**-variabel. Att arrayens storlek är *variabel* beror på att vi har att göra med en dynamisk array. **`new`** returnerar arrayens

adress – närmare bestämt adressen till arrayens första element. I satsen ovan lagrar pekarvariabeln **pekNew** denna adress. Dessutom vet vi från parentesen om *Datorns interna minneshantering* att denna minnesallokering sker på heapen (sid 209). Är det första gången vi använder heapen? Medvetet ja. Omedvetet hade vi gjort det första gången redan i programmet **new** med satsen:

```
int *pekInt = new int;
```

där operatorn **new** allokerar minne för en **int** och returnerar dess adress som vi tilldelar till pekaren **pekInt**. Även här sker allokeringen på heapen då vi använder **new** och inte en vanlig variabeldefinition för allokeringen. Flexibiliteten på heapen gör att vi även kan frigöra minnet på följande sätt:

```
delete pekInt;
```

vilket inte hade varit möjligt om vi hade allokerat på stacken (sid 208) med vanlig variabeldefinition. Operatorn som åstadkommer frigörandet är den till **new** inversa operatorn **delete**. **delete** är invers till **new** därför att den endast kan frigöra minne som allokerats av **new** innan. Med *frigör* menas inget annat än att minnet frigörs för återanvändning. **delete** tar inte bort någon pekarvariabel utan frigör endast den minnescell som pekaren pekar på:

delete *pekarvariabel*;

är den allmänna syntaxen för **delete**. Då **new** returnerar en adress som lagras i en pekarvariabel, skriver man **delete** framför den pekarvariabel som pekar på den minnescell man vill frigöra.

Operatorerna new[] *och* delete[]

Pekar pekarvariabeln på en array och man vill frigöra arrayens minne i sin helhet måste operatorn **delete[]** användas:

delete[] *pekarvariabel*;

Självklart måste en sådan array som man vill tömma, ha varit även allokerad av en operator som allokerar *dynamiska* arrays, nämligen operatorn **new[]**. Båda är arrayutvidgningar av de enkla operatorerna **new** och **delete** och inversa till varandra. I programmet **Dynamic** används **delete[]**-operatorn så här:

```
delete[] pekOld;
```

där pekarvariabeln **pekOld** får sitt värde (en adress) från pekarvariabeln **pekNew**. Hakparentesen måste skrivas med när hela arrayen ska tömmas.

Nollpekaren

I början av programmet **Dynamic** definieras pekarvariabeln **pekOld** i satsen och tilldelas **NULL**, ett värde som vi inte använt hittills:

```
char *pekOld = NULL;
```

NULL kallas för *nollpekaren* och är inte är någon fysisk adress till någon minnescell i datorns RAM utan endast en logisk ”adress” som formellt kan tilldelas pekarvariabler. Man brukar använda nollpekaren för defaultinitiering av pekarvariabler som inte pekar på något värde. Finns det vid tidpunkten av en pekares definition inget minnesutrymme allokerat i programmet som pekaren kan peka på, kan man – och borde helst – initiera pekaren till **NULL**, för att undvika en oinitierad eller s.k. *hängande pekare* (se nedan). Den allmänna syntaxen är:

pekarvariabel **= NULL;**

Istället för **NULL** kan även 0 eller '\0' skrivas. Det sista är inget annat än nolltecknet, vilket visar att nollpekaren är allra första tecknet i ASCII-tabellen med ASCII-koden 0. **NULL** är alltså inte *talet* 0 som har ASCII-koden 48, även om man i koden kan skriva 0 istället för **NULL**. En automatisk typkonvertering av vanliga taltyper till pekardatatyper är i C++ inte möjlig. Därför skulle 0.0 istället för 0 ge kompileringsfel då 0.0 inte längre kan tolkas som nollpekaren. I det här fallet har vi att göra med ett äkta undantag: Koden 0 istället för **NULL** tolkas av kompilatorn som nollpekaren bara därför att den tilldelas en pekarvariabel. I samband med datatypen array av **char** hade vi pratat om noll*tecknet* som används för att terminera en sträng. Att vi här – i samband med pekare – byter terminologi och säger noll*pekare* ska inte leda till missförståndet att det är två olika saker. Båda är identiska tecken som kan kodas med \0, **NULL** eller 0. Det är *funktionen* som är annorlunda – en gång som strängterminator, en gång som pekare som inte pekar på något värde – som motiverar de olika beteckningarna för en och samma sak.

Hängande pekare

Glömmer man tilldela en pekarvariabel adressen till ett allokerat minnesutrymme, händer ingenting så länge man inte försöker att komma åt *värdet* pekaren pekar på. Tar man bort pekarvariabelns tilldelningssats kommer som vanligt ett odefinierat skräpvärde – en skräpadress – att stå i pekarvariabelns minnescell. Men försöker man att komma åt värdet som pekaren pekar på, blir det exekveringsfel och programmet kraschar p.g.a. minneskonflikt. Den skräpadress som råkar stå i en oinitierad pekarvariabels minnescell, pekar på ett minnesutrymme som med största sannolikhet är upptaget av ett annat program i datorn. Fenomenet kallas *hängande pekare* och har konsekvenser som är allvarligare än ett skräpvärde i en vanlig variabel. De flesta abrupta programavbrott följt av en krasch är resultatet av en konflikt i minneshanteringen. Arbetar du med pekare i ett program får du vara noga med att korrekt tilldela dina pekarvariabler så att de pekar på minnesutrymmen som är allokerade i programmet. Man kan ha som en vana att initiera en pekarvariabel med en adress eller med **NULL** i samma sats som den definieras. Står ingen adress till förfogande, därför att det vid tidpunkten av definitionen inte finns något minnesutrymme allokerat, *måste* man initiera med **NULL**, om man sedan använder pekaren för att komma åt ett värde. I programmet **Dynamic** används pekaren **pekOld** redan i **do**-loopens första varv (i kroppen av en **for**-sats) för att kopiera gammal data till den nya arrayen: **pekNew[i] = pekOld[i];** Tar man bort initieringen till **NULL** i

satsen som definierar pekaren **`pekOld`** kan programmet kompileras, men kraschar direkt vid första exekveringen. Testa gärna!

7.5 Dynamisk filkryptering

En av de stora nackdelarna av ett program som kodar vanlig filkryptering är satsen:

```
char filtext[1000];
```

dvs statisk minnesallokering. En array med den fasta storleken `1000` skapas för lagring av filens innehåll fast man varken vet eller kan förutsäga hur stor filen är som ska krypteras. Man var tvungen att ev. justera storleken dvs ändra koden när större filer skulle krypteras – inte precis någon elegant lösning. Nu när vi i förra avsnitt gått igenom dynamisk minnesallokering kan vi lösa problemet. Dynamisk minnesallokering reserverar minne vid exekvering och kan utöka det vid behov när filen läses. Därför lämpar den sig utmärkt för att läsa från vilken fil som helst oavsett storlek, att kryptera och skriva den krypterade texten till en annan fil samt att återställa filens ursprungliga skick i en tredje fil. Det är det vi menar med dynamisk filkryptering.

När vi testade programmet `Dynamic` (sid 210) som introducerar dynamisk minnesallokering matade vi in en text av ganska begränsad storlek som lagrades tecken för tecken i en dynamisk array. Men i princip kan man med den teknik som utvecklades där mata in text av vilken storlek som helst. Och varför inte läsa in text från en fil istället från skärmen? För att göra det med samma teknik ersätts satsen ovan som använde statisk minnesallokering med:

```
char *pekDyn = new char[antal+1];
```

dvs en dynamisk array av `char` vars storlek är `antal+1` där `antal` är en variabel. Operatorn `new[]` allokerar minne för denna array och lagrar arrayens adress i pekaren `pekDyn`. Genom att placera denna sats i en loop och låta `antal` öka med `1` i loopen, skapas i varje varv av loopen en ny array som är `1` byte större än arrayen från förra varvet. Den dynamiska arrayen växer alltså – fast det är en annan ny array varje gång – för att skapa plats för det nyinlästa tecknet. Inläst data frigörs med `delete[]` för att inte i onödan allokera dubbelt så mycket minne som filens storlek.

Programstrukturen

Programmet `DynEncryptFile` är modulariserat och består av följande filer:

`DynEncryptFile.cpp` med	`main()` som anropar alla andra funktioner
`encryptText.h`	funktionen `krypt()` som krypterar text
`readShowFile.h`	funktionen `readShowFile()` som läser en fils innehåll och visar det på skärmen
`writeFile.h`	funktionen `writeFile()` som skriver till en fil
`myRand.h`	funktionen `myRand()` som genererar slumptal

Följande program som kombinerar filkryptering från `EncryptFile` med dynamisk minnesallokering från `Dynamic` implementerar dynamisk filkryptering.

```
// DynEncryptFile.cpp
// Läser in text från en fil utan att ange storleken
// Lagrar texten i en dynamisk array med variabel storlek
// Krypterar texten med en slumpnyckel & dekrypterar med ne-
// gativ slumpnyckel. Skriver till fil och visar den.

#include <iostream>
#include <fstream>
using namespace std;
#include "encryptText.h"             //  Innehåller  krypt()
#include "readShowFile.h"            //        readShowFile()
#include "writeFile.h"               //           writeFile()
#include "myRand.h"                  //              myRand()

int main()
{
   srand(time(0));
   char letter, *filtext = NULL;     // För do:s 1:a varv
   int antal=0, key = myRand(1, 1000);
   ifstream fileForRead("Okrypt.txt");

   do
   {
     char *pekDyn = new char[antal+1];// Dynamisk array (tom)
     for (int i=0; i<antal; i++)     // Kopierar inläst data
         *(pekDyn+i) = *(filtext+i); // till dynamisk array
     delete[] filtext;               // Frigör inläst data
     filtext = pekDyn;               // Ompekning till array
     *(filtext+antal) = letter;      // Lägger nytt tecken i
                                    // array på aktuell pos.
     antal++;
   } while(fileForRead.get(letter)); // Läser tecken från fil

   fileForRead.close();
   cout << "Okrypterad fil:\n";
   for (int i=0; i<antal-1; i++)
   {                                // Tar bort 1:a tecknet:
      *(filtext+i) = *(filtext+i+1); // skräp fr. do 1:a varv
      cout << *(filtext+i);         // Visar filinnehåll
   }

   krypt(filtext, key, antal);      // Krypterar
   writeFile(filtext, "Krypt.txt", antal);
   cout << "\nKrypterad fil:\n";
   readShowFile(filtext, "Krypt.txt");

   krypt(filtext, -key, antal);     // Dekrypterar
   writeFile(filtext, "Återst.txt", antal);
   cout << "\n\nÅterställd fil:\n";
   readShowFile(filtext, "Återst.txt");
   cout << "Krypteringsnyckeln:\t" << key << "\n\n";
}
```

All filhantering görs med de funktioner som redan använts tidigare, vilket är ett exempel på återanvändning av kod i ordets bästa mening – ett resultat av modularisering: För att både läsa från en fil och samtidigt visa filinnehållet på skärmen används funktionen **`readShowFile()`**. Även funktionerna **`writeFile()`** och **`krypt()`** är definierade. Alla dessa funktioners headerfiler har inkluderats i början av programmet. Självklart måste man också kopiera filerna till samma mapp som **`cpp`**-filen eller ange sökvägarna för att kunna kompilera programmet. Inte en enda kod är ändrad i dem. De fungerar som de är fast de skrivits i arraynotation – tack vare släktskapet mellan array och pekare.

`do`-satsen kopierar efter att ha skapat den dynamiska array som **`pekDyn`** pekar på, all hittills inläst data som finns i den array som **`filtext`** pekar på och som är kvar från förra varvet, till den nyskapade tomma dynamiska arrayen med:

```
for (int i=0; i<antal; i++)
    *(pekDyn+i) = *(filtext+i);
```

innan den gamla arrayen frigörs med **`delete[]`**. Den gamla arrayens pekare får den nya arrayens adress med **`filtext = pekDyn;`** för att sedan kunna placera det nyinlästa tecknet på aktuell position i den dynamiska arrayen med:

```
*(filtext+antal) = letter;
```

Samtidigt blir pekaren **`pekDyn`** ledig för att ta emot den nya dynamiska arrayens adress i nästa varv. Själva inläsningen görs i avslutningsvillkoret till **`do`**-satsen med anrop av funktionen **`fileForRead.get(letter)`** som är definierad i klassen **`ifstream`**. Då ingen inläsning görs i det första varvet av **`do`**-satsen hamnar ett odefinierat skräpvärde via den då oinitierade variablen **`letter`** i arrayen. Därför förskjuts hela teckenkedjan ett steg till vänster efter **`do`**-satsen med **`*(filtext + i) = *(filtext + i + 1);`** i en **`for`**-sats.

En av många möjliga slumpkörningar ger följande utskrift:

```
Okrypterad fil:
Denna text kommer från filen Okrypt.txt.
C++ programmet DynEncryptFile läser den från hårddisken utan
kännedom om filens storlek, lagrar texten i en dynamisk array,
krypterar och skriver den krypterade texten till filen Krypt.txt.
För att testa krypteringen återställs texten och skrivs till
filen Återst.txt.

Krypterad fil:
~ƒ¿¿øZ«ƒ▓«ZÑ®°°ƒ¼Zá¼▼¿Záúªƒ¿ZëÑ¼│¬«h«▓«hD}eeZ¬¼®í¼ø°°ƒ«Zrr~│¿à¼│¬«ÇúªZ
ª▲¡ƒ¼Z×ƒ¿Z
á¼▼¿Zó▼¼××ú¡Ñƒ¿Z»«ø¿DÑ▲¿¿ƒ×®°Z®°Záúªƒ¿¡Z¡«®¼ªƒÑfZªøí¼ø¼Z«ƒ▓«ƒ¿ZúZƒ¿Z×│
¿ø°ú¡ÑZø¼¼
ø│fDÑ¼│¬«ƒ¼ø¼Z®ØóZ¡Ñ¼ú░ƒ¼Z×ƒ¿ZÑ¼│¬«ƒ¼ø×ƒZ«ƒ▓«ƒ¿Z«úªªZáúªƒ¿Zà¼│¬«h«▓«hZ
DÇ0¼Zø««Z«
ƒ¡«øZÑ¼│¬«ƒ¼ú¿íƒ¿Z▼«ƒ¼¡«▲ªª¡Z«ƒ▓«ƒ¿Z®ØóZ¡Ñ¼ú░¡Z«úªªDáúªƒ¿Z «ƒ¼¡«h«▓«hD
D
```

```
Återställd fil:
Denna text kommer från filen Okrypt.txt.
C++ programmet DynEncryptFile läser den från hårddisken utan
kännedom om filens storlek, lagrar texten i en dynamisk array,
krypterar och skriver den krypterade texten till filen Krypt.txt.
För att testa krypteringen återställs texten och skrivs till
filen Återst.txt.

Krypteringsnyckeln:     570
```

De externlagrade funktionerna i programmet ovan består av:

Funktionen krypt()

```
// encryptText.h
// Tar emot en text via arrayen t och krypterar den genom
// att förskjuta alla tecken med n steg i ASCII-tabellen
// Kontrollerar textens slut med 3:e parametern antal

void krypt(char t[], int n, int antal)
{
   for (int i = 0; i < antal; i++)
       t[i] = t[i] + n;
}
```

Den aktuella parametern **`key`** öveförs vid anrop till den formella parametern **`n`**. När **`n`** får ett positivt **`key`**-värde, ökas tecknens ASCII-kod med **`n`**. Ett negativt **`key`**-värde minskar ASCII-koderna med samma belopp dvs sätter tecknen tillbaka på sina ursprungliga platser. Därför kan vi använda samma funktion även för dekryptering. Filinnehållet **`fileText`** som skickas till **`t`** är en array av **`char`**. För att kunna avsluta **`for`**-satsen måste vi använda oss av den tredje parametern **`antal`**. De andra externlagrade funktioner som anropas i **`EncryptFile`** är följande:

Funktionen readShowFile()

```
// readShowFile.h
// Funktion som läser innehållet i filen fileName tecken för
// tecken, lagrar det i arrayen t samt visar det på skärmen
// Returnerar dessutom antal tecken som läses och visas

int readShowFile(char t[], string fileName)
{
   int i;              // Antal tecken som läses från filen
   char tecken;
   ifstream fileForRead(fileName);

   for (i=0; fileForRead.get(tecken); i++)
   {
```

```
      t[i] = tecken;
      cout << tecken;
   }
   fileForRead.close();
   return i;
}
```

Funktionen **get()** i **for**-satsens villkor läser ett tecken från filen, lagrar det i **char**-variabeln **letter**, flyttar markören till nästa tecken i filen och returnerar **true** om det finns tecken kvar och **false** om det stöter på filslutteckne t. Så läses filen, lagras i **char**-arrayen **t** samt skickas till **cout**. Antalet tecken returneras.

Funktionen writeFile()

```
// writeFile.h
// Funktion som skriver texten t bestående av antal
// tecken till filen fileName

void writeFile(char t[], string fileName, int antal)
{
   ofstream fileForWrite(fileName);
   for(int i = 0; i < antal; i++)
      fileForWrite << t[i];
   fileForWrite.close();
}
```

I **for**-satsen skriver utmatningsoperatorn **<<** arrayen **t** tecken för tecken till filen **fileName**. Parametern **antal** – antal tecken – används för att avsluta **for**-satsen.

7.6 2D arrays

Med arrays kunde vi bearbeta större mängder av data. Men ibland är inte kvantiteten av data avgörande utan *strukturen*. Följande problem illustrerar detta:

> ”Sex elever i en klass har skrivit fyra olika prov och fått poäng i dem. Skriv ett program som lagrar elevernas poäng i alla prov och skriver ut dem. Sedan ska man kunna ändra ett provresultat till en elev samt skriva ut den uppdaterade elevens poäng.”

Elevernas poäng i olika prov kan lämpligast lagras i en tabell. 2D array som används i följande program, är den naturliga datastrukturen för att lagra tabeller:

```
// 2DArray.cpp
// Elevernas poäng i olika prov lagras i en tabell, skrivs ut
// En elevs poäng uppdateras och visas
// 2D array modellerar tabellen och kommer åt
// arrayens element dvs tabellens värden
#include <iostream>
using namespace std;                 Prov 2

int main()
{
   int poang[6][4] = {  {67, 78, 84, 56},    // (6 x 4)-tabell
                        {49, 37, 59, 74},    // (6 x 4)-array
                        {89, 54, 68, 34},
       Elev 4 -------> {72, 51, 85, 63},
                        {39, 41, 52, 27},
                        {98, 69, 79, 80}  };

   cout << "\n6 elever har fått följande poäng i 4 "
        << "prov:\n\n";

   for (int r=0; r<6; r++)         // Låter skriva ut raderna
   {                               // och gör radbyte
       for (int k=0; k<4; k++)     // Skriver den r:te raden
           cout << poang[r][k] << '\t';
       cout << '\n';
   }

   cout << "\nElev 4 har förbättrat poäng i prov 2 "
        << "och fått: ";
   cin >> poang[3][1];                      // Läser in nytt värde
                                            // till elev 4, prov 2
   cout << "\n Elev 4:s uppdaterade poäng:\n\n";
   for (int k=0; k<4; k++)                  // Skriver ut den 4:e
       cout << poang[3][k] << '\t';         // uppdaterade raden

   cout << "\n\n";
}
```

En tabell bildar en 2D struktur som vi redan stött på i olika sammanhang, t.ex. i nästlade **`for`**-satser med vars hjälp vi skrivit ut tabeller. Men vi har aldrig kunnat lagra tabeller i någon datastruktur i våra program för att sedan komma åt och hantera tabellvärdena. 2D array löser detta problem. I programmet **`2DArray`** definieras den 2Da arrayen **`poang`** i koden

```
int poang[6][4]
```

som är en array vars element i sin tur är arrays, dvs en dubbelarray av **`int`** av storleken 6 x 4, även kallad (6 x 4)-array. Den kan jämföras med en tabell bestående av 6 rader och 4 kolumner. I själva verket är det inget annat än en 6-array av 4-arrays av **`int`**, om vi tillåter att elementen i en array i sin tur kan vara arrays. Eller varför inte prata om *nästlade arrays?* På så sätt kan man – om man vill – tänka vidare och föreställa sig arrays av ännu högre dimension än två. I C++ finns det ingen begränsning mot att bilda flerdimensionella arrays: Man bara ökar antalet nivåer och antalet hakparenteser i definitionen ovan. Vi nöjer oss just nu med två dimensioner där vi har den enkla tabellanalogin.

Definitionen ovan använder definition och initiering i en och samma sats. Vi utnyttjar alltså även här initieringslistan för arrays som tidigare introducerades för endimensionella arrays. Observera att det vid initieringen – närmare bestämt strax efter tilldelningsoperatorn – står *två inledande* klamrar efter varandra **`{ {67, ...`** och även i slutet av initieringssatsen *två avslutande* klamrar **`... , 80} };`** Klamrarna är nästlade i varandra, vilket är det andra kännetecknet för en 2D array. Det första var de två hakparenteserna vid definitionen som innehöll storleken. Vi har alltså att göra med en array på *första nivå* – representerad av de yttre klamrarna. I denna första nivå-array finns det 6 element som i sin tur är arrays. Därför är 6 storleken på denna *första* nivå-array och måste skrivas i den *första* hakparentesen i definitionen ovan. Dess element som i sin tur är arrays, befinner sig på en djupare *andra nivå* – representerade av de inre klamrarna – och har 4 element som är vanliga **`int`**-värden. Därför är 4 storleken på dessa *andra* nivå-arrays och måste skrivas i den *andra* hakparentesen i definitionen ovan. Med andra ord, vi har en *yttre* stor array som innehåller 6 *inre* små arrays med 4 **`int`**-element var, som är nästlade i den stora arrayen. Därför är det hela en 2D (6 x 4)-array. Med hjälp av kodens layout har vi försökt att anknyta till tabellform. Tabellen har 6 rader och 4 kolumner. Varje *rad* representerar en *elev* med sina poäng i olika prov. Varje *kolumn* visar ett *prov* med poäng tillhörande olika elever. Därmed har vi bilden av en (6 x 4)-tabell till en (6 x 4)-array. Generellt kan 2D (m x n)-strukturer kan kodas med (m x n)-arrays där m och n är positiva heltal.

Åtkomst till element i en 2D array

I detta avsnitt hänvisar vi till diskussionen på sid: En arrays hakparenteser **`[]`** har inte samma betydelse i programmets alla satser. I definitionssatser omsluter hakparenteserna *antalet* element i arrayen dvs arrayens *storlek*. I alla andra satser omslutar hakparenteserna *index* till varje element av en array. Detta gäller förstås även för 2D arrays. Hakparenteserna **`[6][4]`** i definitionen ovan innehåller *storleken* till den 2D

arrayen **poang** och talar om att det ska reserveras minnesutrymme för den av storleken 6 x 4 dvs för 24 **int**-värden. Men hakparenteserna i satsen

```
cin >> poang[3][1];
```

längre ned i programmet **2DArray**, innehåller *indexen* till ett element i arrayen **poang**. Självklart är det ett *dubbelindex* som refererar till ett **int**-värde. Man vill komma åt en tabellplats och ändra dess värde genom att läsa in ett nytt värde till den som kommer att skriva över det gamla. Låt oss säga, en elev har gjort omprov i ett ämne, förbättrat sina poäng, och man vill läsa in det nya värdet och föra in det i tabellen. Men vilken elev och vilket prov är det, vilket element i arrayen **poang** är det? Även här måste vi hänvisa till en regel som även gäller för a arrays. Det är indexregeln som säger att numreringen av index i arrays alltid börjar med 0. Det gäller: elementets position = index + 1, där med position menas numret som människan använder för att numrera elementen, medan index är det som skrivs i koden. Därför betyder dubbelindexet **[3][1]** i satsen ovan *inte* elev 3, prov 1, utan enligt indexregeln: elev 4, prov 2. De hårdkodade värdena till arrayen **poang** i programmet **2DArray** visar att det är värdet **51** som står i korset mellan rad 4 och kolumn 2. Alltså refererar koden **poang[3][1]** till värdet **51**. Man tar det första indexet **3** och räknar – genom att börja med 0 – i den stora arrayen **poang** på, så att säga, första nivå. Så kommer man till tabellens rad 4 eller elev 4. Sedan tar man det andra indexet **1** och räknar – genom att börja med 0 – i den redan hittade raden 4. Så kommer man till tabellens kolumn 2 eller prov 2 och hittar där värdet **51**. Det är samma sak som att söka igenom arrayen **poang** på andra, djupare nivå. Dubbelindexets första index refererar till arrayens första och det andra index till arrayens andra nivå. Denna generella regel tillämpas även i den nästlade **for**-satsen som skriver ut hela poängtabellen på skärmen:

```
for (int r = 0; r < 6; r++)      // Låter skriva ut raderna
{                                // och gör radbyte
    for (int k = 0; k < 4; k++) // Skriver den r:te raden
        cout << poang[r][k] << '\t';
    cout << '\n';
}
```

Den inre **for**-slingan skriver ut *en* rad, närmare bestämt den **r**:te raden genom att hålla fast det första indexet **r** och låta det *andra* indexet **k** gå igenom kolumnindexen **0**, **1**, **2**, **3**. Dessutom skickas mellan kolumnerna en tabulator till utskrift. Den yttre **for**-slingan låter den inre slingan att skriva ut raderna 6 gånger genom att låta det *första* indexet **r** gå igenom radindexen **0**, **1**, **2**, **3**, **4**, **5**. Dessutom skickas ett radbyte mellan raderna till utskrift. På liknande sätt hade vi med nästlad **for**-sats skrivit ut ett rektangulärt schema fyllt med stjärnor och en tabell över slumptal.

Efter uppdateringen av elev 4:s poäng i prov 2 vill vi verifiera ändringen genom att skriva ut just denna elevs poäng i alla prov dvs ta ut hela raden 4 ur tabellen med:

```
    for (int k = 0; k < 4; k++)        // Skriver ut den 4:e
```

```
        cout << poang[3][k] << '\t';  // uppdaterade raden
```

Som man ser är detta en kopia av den inre slingan från den nästlade **`for`**-satsen ovan med **`r = 3`**. Raden 4 har enligt indexregeln index **`3`**. Observera att **`poäng`**:s första index hålls fast och det andra indexet räknas upp. Varje enskild rad kan skrivas ut på det här sättet. Att ta ut en enskild kolumn ur tabellen och skriva ut alla elevers poäng från ett prov, t.ex. prov 2, borde gå med följande sats:

```
for (int r = 0; r < 6; r++)
    cout << poang[r][1] << '\n';
```

Här har vi tagit den yttre slingan från den nästlade **`for`**-satsen ovan, eliminerat den inre slingan och ersatt den med utskrift av ett enda värde per rad. Till skillnad från radutskrift hålls **`poäng`**:s andra index fast och det första indexet räknas upp. Dessutom har radbytet lyfts in i **`cout`**-satsen då tabulatorn inte behövs när man skriver ut endast en kolumn. Prova gärna! Slutligen ger ett körresultat av programmet **`2D-Array`**:

```
6 elever har fått följande poäng i 4 prov:

67      78      84      56
49      37      59      74
89      54      68      34
72      51      85      63
39      41      52      27
98      69      79      80

Elev 4 har förbättrat poäng i prov 2 och fått: 99

Elev 4:s uppdaterade poäng:

72      99      85      63
```

7.7 2D array som parameter i funktioner

I förra avsnitt lagrade vi elevernas provresultat i en 2D array och visade samt uppdaterade resultatet i form av en tabell på skärmen (sid 220). En skarp applikation som ska hantera betygsregistrering kan förstås inte nöja sig med detta. Man vill helst lagra resultaten i filer, visa eller uppdatera dem när det behövs och få även de uppdaterade värdena sparade i filer. Dessutom är det i praktiken helt meningslöst att hårdkoda betygen i själva programkoden som vi gjort i **2DArray**, vilket då skedde av pedagogiska skäl. Helst ska man kunna läsa betygen från en fil (ännu bättre från en databas) där administrativ personal lagt in dem. Sedan vill man bearbeta dessa data genom att visa, uppdatera och spara dem i filer. En sådan applikation vore av praktiskt intresse.

Databashantering i C++ ligger utanför denna boks ramar. Inte långt därifrån ligger filhantering som vi lärt oss tidigare. Vi ska nu använda våra kunskaper i filhantering för att programmera betygsregistrering i filer. På köpet kommer vi då även att lära oss att skriva 2D arrays i funktioner. Som vi vet måste koden moduariseras när en applikation vidareutvecklas och växer, för att ge programmet en bra struktur och göra det lätt uppdaterbart. Så, betygsregistrering i filer kommer samtidigt leda till en vidareutveckling av 2D arrays. Låt oss formulera problemet på följande sätt:

”Skriv en funktion **setTable()** som producerar rådata till betygshantering genom att slumpa fram och lagra elevpoäng i olika prov i en 2D array. En annan funktion **writeTable()** ska skriva dessa poäng till en fil. En tredje funktion **readShowTable()** ska läsa dem från samma fil och visa dem på skärmen. En funktion **updateTable()** ska ändra elevers poäng, spara ändringarna i filen genom att anropa **writeTable()** och visa den uppdaterade tabellen genom att anropa **readShowTable()**. Skriv ett program där **main()** anropar dessa funktioner.”

Som man ser består applikationen av fem funktioner inkl. **main()** dvs redan problemställningen är modulariserad. Låt oss börja med att skriva funktionerna – till att börja med – i den ordning de förekommer i uppgiften:

```
// setTable.h
// Tilldelar slumpvärden till 2D arrayen p genom att anropa
// den externlagrade funktionen myRand
// 2D array som parameter i en funktion
#include "myRand.h"
void setTable(int p[][antalProv], int antElever)
{
   srand(time(0));
   for (int r=0; r<antElever; r++)
        for (int k=0; k<antalProv; k++)
             p[r][k] = myRand(10, 100);
}
```

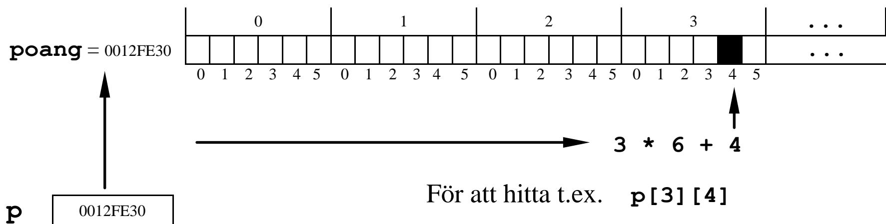
I den ovan beskrivna problemställningen har man redan tänkt igenom strukturen hos uppgiften och formulerat en algoritm så att endast implementeringen i C++ kvarstår.

2D array som parameter

Det nya i funktionen **`setTable()`** ovan är syntaxen, speciellt när det gäller den andra hakparentesen i definitionen av en 2D array i parameterlistan. Samma gäller även i alla följande funktioner som har en 2D array som parameter. En enkel array som parameter hade vi redan behandlat på sid 188. Det finns både likheter och skillnader i behandlingen av 1D och 2D arrays när man skickar dem som parametrar. I funktionen **`setTable()`** är den formella parametern **`p`** en 2D array som definieras i parameterlistan så här:

```
int p[][antalProv]
```

Likheten till enkel array är att den första hakparentesen är tom. Skillnaden är att den andra hakparentesen *inte* är tom – och inte heller *får* vara tom – utan innehåller **`antalProv`** som i hela applikationen är antalet element i 2D arrayens delarrays på andra nivå. Medan arrayens storlek på första nivå är antal elever, är storleken på andra nivå antal prov. Om varje elev gör **`6`** prov har vi följande minesbild:



`poang` är namnet på den array som vid anrop av funktionen **`setTable()`** kommer att överföras till arrayparametern **`p`**. Från enkel array vet vi att p.g.a. referensanrop **`p`** endast lagrar adressen till arrayen **`poang`** (sid 190). För att sedan i funktionen kunna komma åt t.ex. värdet **`p[3][4]`** går kompilatorn från denna adress till arrayen **`poang`**:s index **`3`** på första nivå (elev 4), närmare bestämt till minnescell nummer **`3 * 6 = 18`** och därifrån till index **`4`** på andra nivå (prov 5). Men för denna beräkning behöver kompilatorn utöver indexen **`3`** och **`4`** även informationen om arrayens storlek på andra nivå dvs **`6`** som är antalet prov. Denna information är inte med i funktionen då arrayparametern **`p`** endast lagrar arrayens adress, men inte dess storlek. Därför måste den formella parametern **`p`** i funktionen **`setTable()`**:s parameterlista definieras som **`int p[][antalProv]`**. Den andra hakparentesen får inte vara tom utan måste innehålla just denna information nämligen **`antalProv`**. Tyvärr räcker det inte till att skicka den som parameter, utan den måste redan vara present när man definierar **`p`**. Den första tomma hakparentesen karakteriserar variabeln **`p`** som en arrayparameter som kan lagra adresser. Det är inte förbjudet att fylla den med något värde eller en giltig variabel.

Självklart gör kravet på att antalet kolumner alltid måste anges i 2D arrayparametrar, att dessa funktioner inte blir så generella moduler som man skulle önska. Detta beror på den teknik som internt tillämpas för statisk minnesallokering. Även 2D arrays allokeras statiskt så att de två indexnivåerna måste ändå lineariseras vid allokering i ett sammanhängande minnesområde på ett sätt som visades på bilden på förra sidan.

Tabellhantering i filer

Resten av funktionen **`setTable()`** består av kända koncept som nästlad **`for`**-sats och anropet av funktionen **`myRand()`** som tidigare använts i flera program, här för att fylla arrayen **`poang`** med rådata i form av slumptal mellan **`10`** och **`100`**. Dessa kommer att användas som material för betygshantering i applikationens andra funktioner. Den andra parametern **`antElever`** tar emot elevernas antal och behövs för att avsluta den yttre **`for`**-slingan. Den nästlade **`for`**-satsen initierar 2D arrayen **`poang`** med hjälp av arrayparametern **`p`** vars innehåll sedan skrivs till en fil med följande funktion:

```
// writeTable.h
// Skriver till en fil poäng som lagras i 2D arrayen p
void writeTable(int p[][antalProv], int antElever)
{
   ofstream fileForWrite("Poäng.txt");  // Filen skapas för
                                        // skrivning
   for (int r=0; r<antElever; r++)
   {
      for (int k=0; k<antalProv; k++)
         fileForWrite << p[r][k] << '\t';
      fileForWrite << '\n';
   }
   fileForWrite.close();
}
```

Funktionen **`writeTable()`** skapar filen **`Poäng.txt`** i aktuell mapp på hårddisken, öppnar den för att skriva i den och skickar till den med en nästlad **`for`**-sats hela arrayen som ligger lagrad vid adressen **`p`**. Då **`writeTable()`** kommer att anropas med arrayen **`poang`** som aktuell parameter, är det denna arrays värden som skrivs till filen. Precis som förr används även här parametern **`antElever`** för att avsluta den yttre **`for`**-slingan, medan variabeln **`antalProv`** används inte bara för att avsluta den inre **`for`**-slingan, utan också för att definiera arrayparametern **`p`**. Redan här blir det tydligt att **`antalProv`** som inte är en parameter, måste vara globalt definierad i hela applikationen för att gälla i funktionen **`writeTable()`** vilket är sant även för nästa funktion **`readShowTable()`** som läser från samma fil som **`writeTable()`** skriver till.

I funktionen **`readShowTable()`** på nästa sida förekommer 2D arrayen inte alls i koden. Det är inte heller nödvändigt då filen **`Poäng.txt`** redan innehåller en tabellstruktur som endast behöver läsas och visas på skärmen. Vi behöver här inte

komma åt enskilda arrayvärden. Filen öppnas för läsning med filvariabeln **fileForRead** av typ **ifstream**, men inläsningsmetoden skiljer sig från tidigare. Satsen

```
fileForRead >> buffert;
```

inbakad i en en nästlad **for**-sats gör jobbet. Om man tolkar **>>** som en pil från vänster till höger strömmar data i pilens riktning från filen **fileForRead** dvs **Poäng.txt** till variabeln **buffert**. Precis som **cin >> variabel;** som betyder att data strömmar i pilens riktning från datorns standard input-enhet (**cin** = tangentbordet) till **variabel**. Bara att dataströmmen nu kommer från en fil istället. Att **fileForRead >>** läser endast ett **int**-värde åt gången från filen, beror på att tabulatorn som med funktionen **writeTable()** skickades mellan två **int**-värden till filen, även här tolkas som skiljetecken precis som hos **cin >>**. Variabeln **buffert** har rollen som en temporär lagringsplats mellan inläsning från fil och utskrift till skärmen. Den ersätter arrayparametern. Då den inte kommunicerar med andra funktioner räcker det att definiera den som lokal variabel.

```
// readShowTable.h
// Läser från en fil data som har tabellstruktur och
// visar innehållet på skärmen
void readShowTable(int antElever)
{
   int buffert;
   ifstream fileForRead("Poäng.txt");  // Filen öppnas
   for (int r=0; r<antElever; r++)     // för läsning
   {
       for (int k=0; k<antalProv; k++)
       {
           fileForRead >> buffert;      // Läses från filen
           cout  << buffert << '\t';    // Visas på skärmen
       }
       cout << '\n';
   }
   fileForRead.close();
}
```

För att visa vad som egentligen händer i denna applikation ska vi nu avvika från den ordning funktionerna nämndes i uppgiften och presentera först **main()** innan vi går in på funktionen **updateTable()**.

I **main()** på nästa sida definieras den 2D arrayen **poang** till storleken 14 x 6, initieras med anropet av funktionen **setTable()**, skrivs till fil med anropet av **writeTable()**, läses från samma fil och visas på skärmen med **readShowTable()** för att slutligen ändras, om så önskas, med **updateTable()**. Det enda anmärkningsvärda är att **poang**:s kolumnantal **antalProv** är definierad *globalt*. Anledningen till det är att **antalProv** behövs och därför måste gälla i programmets alla funktioner därför att den representerar antalet element i 2D arrayens delarrays på andra nivå – som förklarades vid genomgången av funktionen **setTable()**. Rekommendationen att vara restriktiv vid användning av globala variabler, gäller förstås fortfarande.

Men just här är alternativet – att parametrisera den globala variabeln – tyvärr inte möjligt, i alla fall inte så länge datatypen är en 2D array. Observera även placeringen av den globala namngivna konstanten **`antalProv`** *före* inkluderingen av alla headerfiler. Det är nödvändigt då **`antalProv`** används i alla våra funktioner som är definierade i dem. Medan arrayen **`poang`**:s kolumnantal är globalt definierad, är radantalet **`antalElever`** en lokal namngiven konstant och skickas som aktuell parameter till alla funktioner där den överförs till de formella parametrarna **`antElever`**. Det är möjligt då antalet rader inte behöver anges när arrayparametern **`p`** definieras i funktionernas parameterlista.

```
// TableFile.cpp
// Anropar funktioner som genererar elevernas poäng i olika
// prov i en tabell, skriver dem till en fil och läser från
// filen. Tabellen kan ändras och uppdateras i filen
#include <iostream>
#include <fstream>
using namespace std;

const int antalProv = 6;         // 2D arrayens kolumnantal

#include "setTable.h"
#include "writeTable.h"
#include "readShowTable.h"
#include "updateTable.h"
int main()
{
  const int antalElever = 14;       // 2D arrayens radantal
  int poang[antalElever][antalProv];// Definition av 2D array
  char startChange;

  setTable(poang, antalElever);     // Arrayen initieras

  writeTable(poang, antalElever);   // Skrivs till fil

  cout << antalElever << " elever har gjort " << antalProv
       << " prov" << " och fått följande poäng:\n\n";

  readShowTable(antalElever);         // Arrayen läses från
                                      // fil och skrivs ut
  cout << "\nVill du göra ändringar i tabellen? (J/N): ";
  cin >> startChange;

  if (startChange == 'j' || startChange == 'J')
     updateTable(poang, antalElever); // Arrayen uppdateras
}
```

Slutligen ska vi även titta på den sista av applikationens funktioner som ger användaren möjligheten att göra ändringar i elevernas poängtabell:

```
// updateTable.h
// Ändrar värden i elevernas poängtabell, skriver den uppdat.
// tabellen till en fil, läser därifrån och visar på skärmen
```

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

void updateTable(int p[][antalProv], int antElever)
{
  int row, column, old;
  char goOnchange;

  do
  {
    cout << "\nVilken elevs poäng vill du ändra på? Elev nr "
         << "(1-" << antElever << "): ";
    cin >> row;
    cout << "Vilket provs poäng vill du ändra på? Prov nr "
         << "(1- " << antalProv << "): ";
    cin >> column;
    old = p[row-1][column-1];          // Gammalt värde spras
    cout << "\nÄndra poängen till: ";
    cin >> p[row-1][column-1];         // Nytt värde läses in
    cout << "\nElev " << row << ":s gamla poäng: " << old
         << ", nya poäng: " << p[row-1][column-1]
         << " i prov " << column << "\n\n";

    writeTable(p, antElever);          // Uppdaterad array
                                       // skrivs till fil
      cout << "Vill du fortsätta ändra? (J/N): ";
      cin >> goOnchange;
   } while (goOnchange  == 'j' || goOnchange  == 'J');

   cout << "\nUppdaterad tabell:\n\n";
   readShowTable(antElever);             // Uppdaterad array
                          // läses från fil och // skrivs ut
}
```

Genom **do**-loopen har användaren t.o.m. möjligheten att göra flera ändringar i elevernas poängtabell. Varje ändring skrivs direkt till filen genom att lägga anropet av **writeTable()** i **do**-loopen. Ändringen av ett arrayelement görs i varje **do**-varv genom inläsning från tangentbordet med en **cin**-sats och skickas vidare till filen via den formella parametern **p** som finns i anropet av **writeTable()**. Den samlade uppdateringen av poängtabellen läses sedan från filen och visas på skärmen efter **do**-loopen med anropet av funktionen **readShowTable()**.

Den här applikationen liknar mycket hanteringen av en databas som administrerar betyg. Filen **Poäng.txt** som lagrar en tabell skulle kunna importeras till en databas av vilket format som helst. Applikationen kan läsa tabellen, skriva i den, ändra den och spara alla uppdateringar i filen – i princip allt som kan göras även med tabeller lagrade i databaser. När programmet **TableFile** körs i sin helhet kan olika dialoger föras. Efter körningen står den ev. uppdaterade slutliga poängtabellen i filen **Poäng.txt**. Dessutom kan t.ex. följande dialog ses på skärmen:

```
14 elever har gjort 6 prov och fått följande poäng:

65      55      85      55      34      88
51      22      39      75      97      21
61      78      55      56      16      44
59      44      63      59      43      66
15      30      30      84      49      65
10      53      45      94      34      25
62      56      90      53      20      71
53      24      92      64      77      65
92      51      91      61      76      18
16      12      28      23      37      20
74      68      14      57      18      82
86      83      38      36      71      46
11      27      96      78      54      88
25      45      88      75      64      30

Vill du göra ändringar i tabellen? (J/N): j

Vilken elevs poäng vill du ändra på? Elev nr (1-14): 7
Vilket provs poäng vill du ändra på? Prov nr (1- 6): 5

Ändra poängen till: 55

Elev 7:s gamla poäng: 20, nya poäng: 55 i prov 5

Vill du fortsätta ändra? (J/N): J

Vilken elevs poäng vill du ändra på? Elev nr (1-14): 14
Vilket provs poäng vill du õndra på? Prov nr (1- 6): 6

Ändra poängen till: 99

Elev 14:s gamla poäng: 30, nya poäng: 99 i prov 6

Vill du fortsätta ändra? (J/N): n
Uppdaterad tabell:

65      55      85      55      34      88
51      22      39      75      97      21
61      78      55      56      16      44
59      44      63      59      43      66
15      30      30      84      49      65
10      53      45      94      34      25
62      56      90      53      55      71
53      24      92      64      77      65
92      51      91      61      76      18
16      12      28      23      37      20
74      68      14      57      18      82
86      83      38      36      71      46
11      27      96      78      54      88
25      45      88      75      64      99
```

7.8 Klasserna `array` och `vector`

Arraybegreppet introducerades tidigt och motiverades med att på ett effektivt sätt lagra och hantera stora datamängder. Även programmeringstekniskt kunde vi skriva kortare och elegantare program med många variabler, när vi i olika sammanhang använde arrays. Programmen blev mer strukturerade och ofta även enklare. I detta avsnitt ska vi går vidare med arrays och lära känna en ny syntax för arrays.

Klassen array

Biblioteksklassen **array** som är en template (generisk) klass, bjuder på möjligheten att skapa objekt av den och på så sätt få tillgång till nya metoder som är definierade i klassen. En av dem är metoden **size()** som mäter arrayens storlek. Följande program demonstrerar detta:

```
// Array_class.cpp
// Array som en klass med metoden size() = storleken
// Ingen automatisk initiering, statisk minnesallokering
#include <iostream>
#include <array>        // Innehåller klassen array
using namespace std;

int main()
{
   array<int, 5> a;     // array-objekt a med 5 int-element
   a = {0};             // Nollinitiering, annars skräp

   array<int, 5> copy = a;    // Skapar en kopia av arrayen a
   cout << "\ncopy's index\tvärde\n";
   for (int i=0; i < copy.size(); i++)
       cout << "\t " << i << "\t  " << copy[i] << "\n";
   cout << "\ncopy's storlek är " << copy.size()
        << ".\tcopy[5] är odefinierat.\n";
// cout << copy[5];        // Överskridning av index
}                          // leder till exekveringsfel
```

En körning av programmet ovan visar att värdena till arrayen **a** via enkel tilldelning överförts till arrayen **copy**:

```
copy's index  värde
        0      0
        1      0
        2      0
        3      0
        4      0
copy's storlek är 5.  copy[5] är odefinierat.
```

Som man ser är följande typiska arrayegenskaper kvar även i klassvarianten **Array_class**:

1. Minnesallokeringen är fortfarande statisk: Storleken måste anges när arrayen skapas:

```
array<int, 5> a;
```

2. Initieringen är inte automatisk och måste göras explicit.

3. Överskridning av den definierade indexgränsen leder till exekveringsfel.

Klassvariantens enda praktiska fördel i exemplet ovan är användningen av metoden **size()**. I vanliga arrays kan **size()** inte användas.

Arrayens initieringslista

Man kan slå ihop definitionen av en array med initieringen till en kortform som kallas för *initieringslista*. Detta har vi använt i följande program som dessutom introducerar (i kommentar) den s.k. *foreach-satsen* i C++:

```
// Array_Init.cpp
// Arrayens initieringslista
// Summerar arraytermerna en gång med vanlig for-sats
// En andra gång med foreach-sats (bortkommenterad)
#include <iostream>
#include <array>
using namespace std;

int main()
{                       // Initieringslista:
   array<int, 5> termer = { 2, 4, 6, 8, 10 };

   cout << "\n\tArrayens index\ttermer\n";
   for (int i = 0; i < termer.size(); i++)
       cout << "\t\t   " << i << "\t  "
            << termer[i] << "\n";

   int sum = 0;
   for (int i = 0; i < termer.size(); ++i)  // Vanlig for-sats
       sum += termer[i];

// for (int element : termer)           // foreach-sats
//     sum += element;

   cout << "\tSumman av termerna: " << sum << "\n";
}
```

Det känns lite märkligt att arrayen **termer** måste definieras *med* uppgift om arrayens storlek:

```
array<int, 5> termer = { 2, 4, 6, 8, 10 };
```

Detta var inte nödvändigt när vi definierade array utan klass. Men här får man kompileringsfel när man utelämnar storleken `5`.

En körning av programexemplet **`ArrayInit`** (sid 232) visar att värdena från arrayen **`termer`** summeras med en vanlig **`for`**-sats som kan ersättas av den bortkommenterade **`foreach`**-satsen:

```
Arrayens index  termer
          0       2
          1       4
          2       6
          3       8
          4       10
Summan av termerna: 30
```

foreach-satsen

Följande **`foreach`**-sats är i programmet **`ArrayInit`** bortkommenterad:

```
for (int element : termer)
    sum += element;
```

Översatt till svenska:

För varje **`element`** av arrayen **`termer`**
addera **`sum`** med **`element`** och tilldela resultatet till **`sum`**

Den gör exakt samma sak som den vanliga **`for`**-satsen två rader över den, nämligen att summera termerna (elementen) i arrayen **`termer`**. Prova gärna själv genom att aktivera den och bortkommentera den vanliga **`for`**-satsen.

Satsen ovan är ett exempel på den s.k. **`foreach`**-satsen i C++, även kallad den *intervallbaserade* (eng.: *range based*) **`for`**-satsen. Ingen av de här beteckningarna är optimal. Satsen inleds med samma reserverade ord som den vanliga **`for`**-satsen, men har en annorlunda syntax och struktur. Nedan följer några viktiga egenskaper hos **`foreach`**-satsen:

Iterationsvariabeln

Variabeln **`element`** – namnet är valt av oss – kallas även för *iterationsvariabel* och ersätter **`for`**-satsens räknare. Men till skillnad från räknaren är **`element`** inget index (nr) utan en variabel som tilldelas arrayens enskilda värden – en i taget. I vårt exempel är den definierad till **`int`**. Den fortskrider från element till element tills alla element är genomgångna och gör att satsens kropp upprepas. Kolonet **`:`** betyder *av* eller *element av* och refererar till arrayen som ska loopas igenom.

`foreach`-satsens enkelhet består i att den till skillnad från **`for`**-satsen varken behöver ett start-, steg- eller slutvärde resp. avslutningsvillkor. Den går helt enkelt igenom arrayens *alla* element, från det första till det sista. Det är själva arrayen som

bestämmer start-, steg- och slutvärdena. Variabeln **element** som i varje varv av loopen pekar på resp. arrayelementets värde, användas sedan i loopens kropp för att göra det man önskar. Därför är **foreach**-satsen idealisk för arrays.

foreach-satsens iterationsvariabel måste ha samma datatyp som arrayelementen eller en sådan datatyp som arrayelementens datatyp automatiskt kan konverteras till. I vårt exempel har vi **int**. Det är t.o.m. möjligt att ha egendefinierade datatyper dvs klasser.

En viktig egenskap av iterationsvariabeln är att den inte kan ändra arrayelementens värden i **foreach**-satsens kropp. Den är så att säga *read only*. I praktiken innebär detta att iterationsvariabeln inte får förekomma till vänster om tilldelningsoperatorn (**=**) i någon sats i **foreach**-satsens kropp. Kopplingen mellan array och **for**-sats som behandlades i slutet av förra avsnitt kan ytterligare förenklas och effektiviseras med **foreach**-satsen.

Iterationsvariabeln är lokal

Följande program demonstrerar **foreach**-satsens ovan nämnda egenskap:

```
// Foreach.cpp
// foreach-satsen ändrar en arrays värden
// Iterationsvariabeln är lokal och är
// odefinierad utanför foreach-satsen
// Därför görs ändringen med referens

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

int main()
{
   array<char, 5> a = {'a', 'b', 'c', 'd', 'e'};

   cout << "\n\tArrayelementen före ändring:  ";
   for (char i : a)      // i = iterationsvariabeln
       cout << i << " ";// i = a[] Kopiering av värdet
                         // Två olika minnesceller
// for (char  j : a)     // OBS! Funkar inte!
                         // j ändras, inte a[]
   for (char& j : a)     // j referens till a[]
       j += 1;           // Samma minnescell:
                         // Däför ändras båda
   cout << "\n\n\tArrayelementen efter ändring: ";
   for (char k : a)      // a[] ändrade ovan
       cout << k << " ";
   cout << "\n";
}
```

En körning av programexemplet **Foreach** visar hur värdena från arrayen **a** endast kan ändras genom att deklarera itertionsvariabeln sm referens. Prova gärna den bortkommenterade koden utan referens.

```
Arrayelementen före ändring:  a b c d e

Arrayelementen efter ändring: b c d e f
```

Klassen vector

Men vi vet också att array har vissa nackdelar som vi nu vill komma över genom att ersätta den med *vektor*. Arrayens nackdelar kan beskrivas i två punkter:

1. ***Statisk minnesallokering***

   Storleken av en array måste alltid anges i koden när arrayen definieras, t.ex.:

   ```
   int no[20];
   ```

   Detta beror på att C++ tillämpar *statisk minnesallokering* för arrays, vilket innebär att kompilatorn allokerar minnesutrymme för **20** minnesceller där varje minnescell lagrar ett **int**-värde. Storleken på detta minnesutrymme kan inte ändras under exekveringen.

2. ***Elementvis hantering***

   En array måste hanteras *elementvis*, vare sig vid tilldelning eller annan behandling, vare sig med eller utan loop.

   Att detta inte alltid behöver vara så kunde vi se i programmet **EmployeeTest** där vi kopierade objektet **anst** efter initiering till ett nytt objekt **copy**:

   ```
   Anstalld copy = anst;
   ```

   Samma sak kan man göra med en vektor:

   ```
   vector<int> copy = no;
   ```

   Om **no** är en vektor som redan är definierad och initierad, kan man direkt överföra dess värden till en ny vektor **copy** som definieras och initieras med satsen ovan.

Vad gäller punkt **1** tillämpar C++ *dynamisk minnesallokering* för vektorer. Storleken på en vektor behöver inte, ja får inte anges på förhand. I den bemärkelsen är alltså en *vektor* en *dynamisk array*.

Vad gäller punkt **2** är denna direkta tilldelning av en vektor inte möjligt med en array. I den bemärkelsen är alltså en *vektor* ett *objekt* vars klass är fördefinierad i ett programbibliotek som måste inkluderas med:

```
#include <vector>
```

Efter denna inledning till vektorer vill vi använda den nya datatypen för att demonstrera vektorns initieringslista.

Vektorns initieringslista

Först inkluderar vi biblioteket `<vector>` i det nya programmet. Sedan skapar vi i **main()** vektorn **no** och initierar den i samma sats med initieringslistan:

```
vector<int> no = {64, 86, -6};
```

Därmed är vi redo för att skapa en andra vektor **copy** och initiera den direkt med den första vektorn **no** utan elementvis tilldelning:

```
vector<int> copy = no;
```

```
// VectorInit.cpp
// Definition & initiering av en vektor med initieringslista
// Ingen storlek behövs i förväg: Dynamisk minnesallokering
// Direkt tilldelning av vektorer, inte elementvis
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   vector<int> no = {64, 86, -6};   // Definition och initiering
                                    // av vektorn no med initie-
                                    // ringslista
   vector<int> copy = no;           // Kopiering av no till copy
   cout << "\n\tcopy:s 1:a element copy[0] = " << copy[0]
        << " med index 0\n"
        << "\n\tcopy:s 2:a element copy[1] = " << copy[1]
        << " med index 1\n"
        << "\n\tcopy:s 3:e element copy[2] = " << copy[2]
        << " med index 2\n";
}
```

En körning av programexemplet **VectorInit** visar att värdena från arrayen **no** verkligen kopierats över till arrayen **copy**:

```
     copy:s 1:a element copy[0] = 64 med index 0

     copy:s 2:a element copy[1] = 86 med index 1

     copy:s 3:e element copy[2] = -6 med index 2
```

Övningar till kapitel 7

7.1 Skriv ett program som läser in tre värden – heltal, decimaltal, tecken eller sträng – och bestämmer samt skriver ut det största av dem. Implementera algoritmen för att bestämma max-värdet, i en template funktion, dvs den ska vara oberoende av inmatningens datatyp. Använd konceptet som beskrivs i avsnitt ***7.3 Templates*** (sid 203).

Kapitel 8

Mer om OOP

	Ämne	Sida	Program
8.1	Komposition	239	**`Date`**
	- Komposition av klasser	239	**`Employ`**
	- Komposition av objekt	242	**`Composition`**
8.2	Arv	244	**`Person`**
	- Arvrelationen	246	**`EmployeeInh`**
		247	**`Inheritance`**
8.3	Polymorfism	249	**`Account`**
	- Överskuggning av metoder	251	**`MinAccount`**
	- Åtkomstmodifieraren protected	252	**`CreateAccount`**
Övningar till kapitel 8		255	

8.1 Komposition

Komposition betyder sammansättning och för tankarna tillbaka till funktioner, modularisering, Lego-principen, återanvändning av kod och hela diskussionen om att bygga program med hjälp av redan skrivna och testade moduler. Då kallades modulerna *funktioner* som satt ihop ett program. Nu är det *klasser* som ska sammansätta (komponera) eller ingå som komponenter i andra klasser. Den övergripande strukturen av ett C++ program är fortfarande en samling av variabler, funktioner och klasser. Objektorienterade program har för det mesta bara **`main()`**-funktionen kvar och resten är klasser i vilka man definierar och anropar sina metoder. Då menar man med komposition sammansättning av en klass med en annan klass. Tänk på en bil som har en motor. Man sätter ihop bilen som ett objekt av klassen *Bil* genom att bygga in i den bl.a. en motor som i sin tur är ett objekt av en annan klass, klassen *Motor*.

Vi har redan använt komposition i klassen **`TimAnstalld`** utan att närmare gå in på begreppet. Där behövde vi arrayen **`arbetstid`** som datamedlem som i sin tur består av ett antal objekt av en annan klass, klassen **`Tid`**, och måste deklareras som sådan. När vi gör det blir **`Tid`**-objekten **`arbetstid`** komponenter i varje **`TimAnstalld`**-objekt. Man har sammansatt klassen **`TimAnstalld`** med hjälp av klassen **`Tid`**. Att detta överhuvudtaget är möjligt ur designsynpunkt, beror på att relationen mellan en timanställd och arbetstider – rent begreppsmässigt utanför all programmering – är att en timanställd *har* arbetstider. En sådan relation mellan två begrepp kallas i objektorienterad design för ”*har*”-relation och är den grundläggande förutsättningen för komposition av klasser. Om två begrepp står i en ”*har*”-relation till varandra kan man bygga det ena (stora) med hjälp av det andra (lilla). Ett hus har en dörr och andra komponenter. En cykel har hjul. En bil har en motor, en motor har i sin tur cylindrar osv. I praktiken bygger man också alla dessa komponenter separat och sammansätter dem sedan.

En annan viktig relation mellan objekt i den reala världen kallas i objektorienterad design för ”*är*”-relation och måste begreppsmässigt noggrant skiljas från ”*har*”-relationen. Båda är relevanta klassificeringsverktyg vid modellering och design av en verklig miljö. ”*Är*”-relationen är den grundläggande förutsättningen för *arv* hos klasser. Klasser kan ärva varandra om de står i en ”*är*”-relation till varandra. En lastbil *är* en bil, därför kan klassen lastbil ärva klassen bil dvs ta över bilens delar och funktioner, modifiera och anpassa dem till lastbilen. *Komposition* är vid modellering ibland ett alternativ och ibland en konkurrent till *arv*. Pedagogiskt sett är det enklare att förklara komposition. Därför kan detta avsnitt anses som en förberedelse för nästa avsnitt som behandlar arv som efter *inkapsling* är den andra hörnstenen i objektorienterad programmering.

Komposition av klasser

För att bättre kunna förstå skillnaden mellan komposition och arv vill vi i båda avsnitt behandla samma exempel, nämligen en anställd som förutom för- och efternamn, också har ett födelse- och ett anställningsdatum. Medan för- och efternamn är

strängar och kan deklareras som sådana, har födelse- och anställningsdatum inte några fördefinierade typer. De är båda av typ *datum*, så vi måste först deklarera en sådan klass. Observera att datum och tid inte är samma sak, så vi kan inte använda klassen **`Tid`** från tidigare. Medan tid är en varaktighet bestående av ett antal tidsenheter, ett intervall med en början och ett slut, är datum en viss tid*punkt*. En tid består av många tidpunkter. I praktiskt sammanhang är det i regel tillräckligt att modellera datum som en klass med datamedlemmarna dag, månad och år. I koden använder vi engelska beteckningar:

```
// Date.h
// Deklarerar klassen Date med 2 konstruktorer (överlagring):
// En vanlig (allmän) konstruktor och en default-konstruktor
// Utskriftsmetoden konkatenerar och skriver ut datum som
// sträng

class Date
{
   int day, month, year;

public:
   Date(int d, int m, int y)           // Allmän konstruktor
   {
      day   = d;
      month = m;
      year  = y;
   }

   Date()                              // Default-konstruktor
   {
      day = month = year = 0;
   }

   void skrivUt()                      // Utskriftsmetod
   {
      cout << year << "-" << month << "-" << day;
   }
};
```

Vi har försökt att formulera klassen så generellt som möjligt så att den kan användas i alla program som behöver objekt av typ **`Date`**. I vårt exempel kommer vi att behöva den för att bygga klassen **`Employ`**, närmare bestämt för att deklarera en anställds födelse- och anställningsdatum som **`Date`**.

I klassen **`Date`** finns två konstruktorer, default-konstruktorn som inte har några parametrar alls och en allmän konstruktor som har tre parametrar. De överlagrar varandra. Kompilatorn kan skilja åt dem p.g.a. det olika antalet parametrar. Skapar vi ett **`Date`**-objekt med **`Date b;`** har vi automatiskt anropat default-konstruktorn. Gör vi det däremot t.ex. med **`Date birthdate(12, 3, 2000);`** har vi anropat den allmänna konstruktorn. Eftersom vi i fortsättningen kommer att använda båda måste

vi ha en *egen* default-konstruktor som har explicit definierats i **`Date`**. Annars får vi kompileringsfelet:

. . . 'Date' : no appropriate default constructor available

Anledningen är att den allmänna konstruktorn slår ut *kompilatorns* default-konstruktor.

Set-metoder har vi inte i **`Date`** därför att vi i vårt exempel inte kommer att ha något behov för ändringar av varken födelse- eller anställningsdatum. För andra ändamål där det behövs kan man lätt komplettera klassen med set-metoder. Vad gäller get-metoder ersätter utskritsmetoden alla sådana när den skriver ut alla tre datamedlemmar och representerar datum som en sträng, dessutom i svenskt datumformat.

Nu, när vi har klassen **`Date`** till förfogande, kan vi använda den i följande klass för att deklarera en anställds födelse- och anställningsdatum med den nya datatypen **`Date`**:

```
// Employ.h
// Klassen Employ sätts ihop bl.a. med klassen Date som kom
// ponent. Mellan klasserna Employ och Date finns en "har"-
// relation: Employ "har" två Dates

class Employ
{
   string firstname, lastname;                  // Komposition
   Date birthdate;
   Date hiredate;

public:
   Employ(string f, string n, Date b, Date h) // Konstruktorn
   {
      firstname = f;
      lastname  = n;
      birthdate = b;
      hiredate  = h;
   }

   string skrivUt()                             // Returnerar
   {                                            // namnet som
      return firstname + " " + lastname;        // konkatenerad
   }                                            // sträng
};
```

I klassen **`Employ`** har en anställd ett för- och efternamn som båda är av typ **`string`**. Faktiskt är även **`string`** en klass, även om en fördefinierad sådan, så att vi redan här har att göra med komposition. När **`string`** introducerades första gången, kallade vi det för datatyp för då kände vi inte till klassbegreppet. Sedan kommer den självgjorda kompositionen med den egendefinierade klassen **`Date`**. En anställd har också ett födelse- och ett anställningsdatum, båda av typ **`Date`**. I koden utgörs denna

”*har*”-relation av deklarationen av datamedlemmarna **birthdate** och **hiredate** som **Date**-objekt (framhävd med vit bakgrund). Metoden **skrivUt()** är lite annorlunda än våra utskriftsmetoder hitttills. Den returnerar en sträng som konkateneras med operatorn **+** .

Komposition av objekt

Nu har vi två klasser till förfogande – **Employ** och **Date** – där den ena är en komponent i den andra. Därmed kommer varje objekt av typ **Employ** att vara ett sammansatt eller komponerat objekt, sammansatt av två objekt av typ **string** och två objekt av typ **Date**. Som en konsekvens har även konstruktorn två **string**-objekt och två **Date**-objekt som parametrar. För **string**-parametrarna går det problemfritt då **string** är en fördefinierad klass:

```
Employ(string f, string n, Date b, Date h)
```

Men för **Date**-parametrarna **b** och **h** i **Employ**-konstruktorns parameterlista krävs en default-konstruktor i klassen **Date** därför att koden **Date b, Date h** innebär anrop av **Date**-klassens default-konstruktor samtidigt den skapar dessa objekt. Men anrop av en metod förutsätter dess existens. Vi tänkte på det när vi i klassen **Date** definierade en egen default-konstruktor då kompilatorns automatiska default-konstruktor var utslagen p.g.a. **Date**-klassens allmänna konstruktor med 3 parametrar (sid 240).

Den berättigade frågan som uppstår nu är följande: Varför förs denna argumentation inte för deklararationen av datamedlemmarna **birthdate** och **hiredate**? Varför anropas inte **Date**-klassens default-konstruktor i följande satser bara någon rad innan?

```
Date birthdate;
Date hiredate;
```

Anledningen är att här *skapas* inte objekten **birthdate** och **hiredate**. De *deklareras* endast här. Som komponenter (delobjekt) skapas de först när ett helt **Employ**-objekt skapas i följande testprogram:

```
// Composition.cpp
// Komposition av objekt:
// Ett Employ-objekt byggs upp med hjälp av 2 Date-objekt
// För att kunna skapa Employ-objektet måste först delobjek-
// ten av typ Date skapas och skickas till konstruktorn
#include <iostream>
using namespace std;

#include "Date.h"                         // Klassen Date
#include "Employ.h"                       // Klassen Employ

int main()
{



   Date birthday(12, 10, 1969);
```

```
   Date hireday (15, 11, 2001);

   Employ anst("Kalle", "Karlsson", birthday, hireday);

   cout << "\n\t" << anst.skrivUt() << " är född ";
   birthday.skrivUt();
   cout <<  "\n\n\toch har jobbat sedan   ";
   hireday.skrivUt();
   cout << "\n";
}
```

Objekten **birthdate** och **hiredate** ingår som komponenter i objektet **Employ**. Därför måste de skapas först. Det gör vi genom att initiera dem med vissa datum och anropa **Date**-klassens allmänna konstruktor med 3 parametrar. Sedan skickas de som parametrar till **Employ**-konstruktorn när objektet **anst** skapas. Observera att **Date**-klassens default-konstruktor inte anropas här. Ändå behövs den för den anropas i klassen **Employ** som inkluderas här, närmare bestämt i parameterlistan till **Employ**-klassens konstruktor. Dit skickas **birthdate** och **hiredate** och kopieras över (värdeanrop) till de formella parametrarna **b** och **h** när objektet **anst** skapas.

Slutligen får vi följande utskrift när vi kör **Composition**:

```
        Kalle Karlsson är född 1969-10-12

        och har jobbat sedan   2001-11-15
```

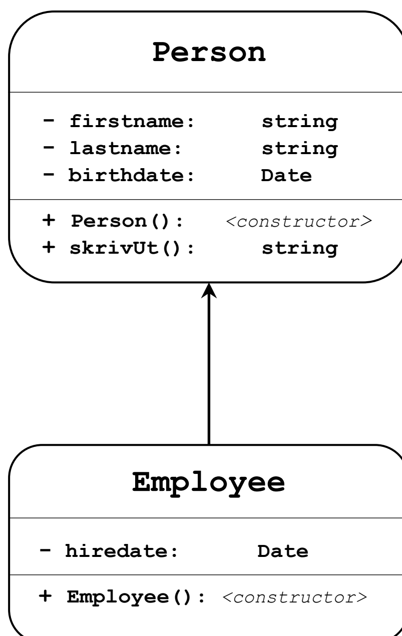
Namnet skrivs ut med anrop av **anst.skrivUt()** som returnerar den konkatenerade strängen med för- och efternamn med mellanslag däremellan. Därför kan den integreras i **cout**-satsen. Datumen däremot skrivs ut med **Date**-klassens utskriftsmetod som är av typ **void** och därför anropas fristående. Det är alltså två olika utskriftsmetoder med sammma namn, definierade i två olika klasser.

I nästa avsnitt kommer vi att vidareutveckla exemplet med anställda genom att flytta en del av koden från klassen **Employ** till en överordnad klass och etablera en arvrelation mellan dem. Komposition med klassen **Date** bibehålls så att resultatet blir en kombination av arv och komposition.

8.2 Arv

Arv är efter inkapsling den andra hörnstenen i objektorienterad programmering. Medan inkapsling har att göra med dataskydd och dataintegritet, är arv ett objektorienterat koncept för att förverkliga programmeringens gamla önskedrömmar om *modularisering*, *återanvändning av kod* och *strukturering av program* – mål som är svårt att uppnå, som i praktiken uppnås endast delvis och som vi eftersträvar ända sedan vi introducerade funktionsbegreppet. Därför är ytterligare verktyg för att komma vidare på den inslagna vägen, bara välkomna. Medan man med funktioner separerade upprepad kod är arv mer sofistikerat: Man skapar en ny klass som en underkategori av en annan befintlig klass och återanvänder den befintliga klassens kod i den nya klassen. Den nya klassen *ärver* den befintliga klassen. I arv är alltså alltid minst två klasser inblandade.

Man kan alltid etablera en arvrelation mellan två begrepp om de står i en en "*är*"-relation till varandra. Exempel: en anställd *är* en person. Därför kan en ny klass **`Employee`** ärva klassen **`Person`** dvs ta över all kod som redan finns där och lägga till ny kod som närmare specificerar en anställd. På så sätt slipper man skriva om kod som redan finns. T.ex. har en person ett för- och efternamn samt ett födelsedatum. Vid modellering av en anställd ärvs dessa datamedlemmar, och man lägger till den nya datamedlemmen anställningsdatum som är speciell för en anställd. Klassen **`Person`** kallas *bas-* eller *superklass*. Klassen **`Employee`** kallas *härledd* eller *subklass*. Subklassen ärver superklassens alla datamedlemmar och metoder. I UML-diagram ritas arvrelationen med en pil riktad mot superklassen:



Det som har hänt med exemplet jämfört med förra avsnitt är en flyttning av en del av koden från klassen **`Employ(ee)`** till klassen **`Person`**. Det som är specifik för en anställd, datamedlemmen anställningsdatum, är kvar i **`Employee`**. Allt som är relevant för alla personer har flyttats till klassen **`Person`**. Arvrelationen garanterar att dessa datamedlemmar och metoder kan nås även från ett **`Employee`**-objekt – självklart upp till åtkomstreglerna. Arv upphäver inte åtkomstmodifierarnas giltighet: En privat medlem är absolut oåtkomlig utifrån klassen, även från en subklass.

Observera att klassen **`Date`** är helt oberörd av denna omplacering av kod (arvkonstruktionen). Fortfarande ”har” en anställd ett anställningsdatum. En person ”har” ett födelsedatum. Detta är oberoende av att en anställd ”är” en person. Båda relationer förekommer parallellt. Därför har vi nu att göra med en kombination av komposition och arv. Komposition är något helt naturligt och ställer inga speciella krav på syntaxen, medan arv introducerar ny kod i C++. Frågan är: Hur ser syntaxen ut för pilen i UML-diagrammet ovan? Och hur påverkar arvrelationen konstruktorns kod speciellt i subklassen? För att få svar implementerar vi modellen ovan genom att börja med klassen **`Person`**:

```
// Person.h
// Person som en superklass till subklassen Employee.
// Kan även användas som superklass till andra subklasser

class Person
{
   string firstname, lastname;
   Date  birthdate;

public:
   Person(string f, string n, Date b)         // Konstruktorn
   {
      firstname = f;
      lastname  = n;
      birthdate = b;
   }

   string skrivUt()                           // Returnerar
   {                                          // namnet som
      return firstname + " " + lastname;      // konkatenerad
   }                                          // sträng
};
```

Som man ser är klassen **`Person`** exakt samma som klassen **`Employ`** minus datamedlemmen **`hiredate`** (sid 241). Bara att den även fungerar nu som en superklass i den ovan beskrivna UML-modellen. Men av denna roll finns det inget spår i koden. Arvrelationen skrivs alltid in i subklassen, inte i superklassen. Klassen **`Person`** måste vara så generell att den även kan användas i andra program som behöver en sådan klass. Det är ju just meningen med *återanvändning av kod*. Självklart kan man tänka sig en ännu mer generell version av klassen **`Person`** med fler medlemmar som

t.ex. personnr, postadress, mailadress, telnr osv. Vi nöjer oss dock för enkelhetens skull med versionen ovan.

Arvrelationen

Som ett resultat av återanvändning av kod blir nu subklassen **`Employee`** mycket kort för det mesta är redan kodad i superklassen **`Person`**:

```
// EmployeeInh.h
// Employee som härledd eller subklass till Person som är
// bas- eller superklass: All kod ärvs från Person och är här
// giltig. En ny datamedlem hiredate tillkommer
// Konstruktorn anropar superklassens konstruktor och lägger
// till initieringen av den nya datamedlemmen hiredate

class Employee : public Person       // Employee ärver Person
{
   Date hiredate;                    // Ny datamedlem

public:                              // Anrop av superklassens
                                     // konstruktor
   Employee(string f, string n, Date b, Date h)
                                        : Person(f, n, b)
   {
      hiredate = h;
   }
};
```

För att koppla ihop klasserna **`Employee`** och **`Person`** och etablera en arvrelation mellan dem måste två saker göras:

1. **I klasshuvudet** måste tilläggas information om att en arvrelation ska etableras utifrån den här klassen (subklassen). Namnet på den klass som relationen ska kopplas till (superklassen) måste anges. Så här ser den allmänna syntaxen ut:

   `class` *`subklass`* **`: public`** *`superklass`*

   Detta innebär att *`subklass`* ärver *`superklass`*. Kolon **`:`** är symbolen för ”ärver”. Vill man sedan t.ex. i **`main()`** från ett **`Employee`**-objekt komma åt **`Person`**-klassens **`public`**-medlemmar – vi vill göra det – måste dessutom åtkomstmodifieraren **`public`** skrivas framför superklassens namn. Superklassens **`private`**-medlemmar kommer man i alla fall inte åt, om man inte ändrar åtkomsten i superklassen.

2. **I konstruktorns huvud** måste så många parametrar tas upp i parameterlistan som det finns privata datamedlemmar *både* i super- *och* subklassen. Det räcker inte bara med subklassens privata datamedlem. Man måste nämligen från ett **`Employee`**-objekt kunna initiera inte bara **`hiredate`**, utan även **`firstname`**, **`lastname`** och **`birthdate`**. Man måste kunna skapa ett *fullständigt* **`Employee`**-objekt med konstruktorn. Därför måste denna ha fyra parametrar av vilka de tre första vidarebefordras till superklassens konstruktor. Den fjärde ska initiera

subklassens datamedlem **hiredate**. Så förklaras i alla fall den första delen av konstruktorhuvudet:

```
Employee(string f, string n, Date b, Date h) : Person(f, n, b)
```

Den andra delen efter arvsymbolen kolon **:** är anropet av superklassen **Person**:s konstruktor med de tre aktuella parametrar som är definierade i parameterlistan strax innan före kolonet. På så sätt vidarebefordrar man deras värden till klassen **Person** och initierar där **firstname**, **lastname** och **birthdate**. Anropet innebär återanvändning av koden till **Person**-konstruktorn. Slutligen initieras datamedlemmen **hiredate** i kroppen till **Employee**-konstruktorn.

Programmet som testar klassen **Employee**:

```
// Inheritance.cpp
// Skapar Employee-objekt & anropar utifrån det superklassen
// Person:s offentliga metod för att skriva ut för- och
efternamn
#include <iostream>
#include <string>
using namespace std;

#include "Date.h"                       // Klassen Date
#include "Person.h"                     // Klassen Person
#include "EmployeeInh.h"                // Klassen Employee

int main()
{
   Date   birthday(12, 10, 1969);
   Date   hireday (15, 11, 2001);
   Employee anst("Kalle", "Karlsson", birthday, hireday);

   cout << "\n\t" << anst.skrivUt()  // Anrop av Person:s
        << " är född ";              // utskriftsmetod med ett
                                     // Employee-objekt
   birthday.skrivUt();
   cout <<  "\n\n\toch har jobbat sedan   ";
   hireday.skrivUt();
   cout << "\n\n";
}
```

är, när det gäller koden, nästan identiskt med **Composition**. Enda skillnaden är att klassen **Employ** har byts ut mot **Employee** och att inkluderingen av klassen **Employ** har ersatts av inkluderingen av klasserna **Person** och **Employee**. Det sista är förstås den avgörande skillnaden: Klassen **Employee** är helt annorlunda och tillämpar arv. T.ex. har den ingen metod **skrivUt**. Ändå kan vi i **cout**-satsen anropa denna metod i objektet **anst** som är av typ **Employee**:

```
anst.skrivUt()
```

Kompilatorn tittar i klassen **`Employee`** och hittar där ingen metod **`skrivUt`**. Men eftersom **`Employee`** tillämpar arv och är subklass till **`Person`**, går kompilatorn ”upp” till superklassen och hittar där metoden **`skrivUt`**. Därav också förklaringen till pilen i UML-diagrammet som är riktad uppåt mot superklassen (sid 244). Programmet producerar samma utskrift som **`Composition`** (sid 243).

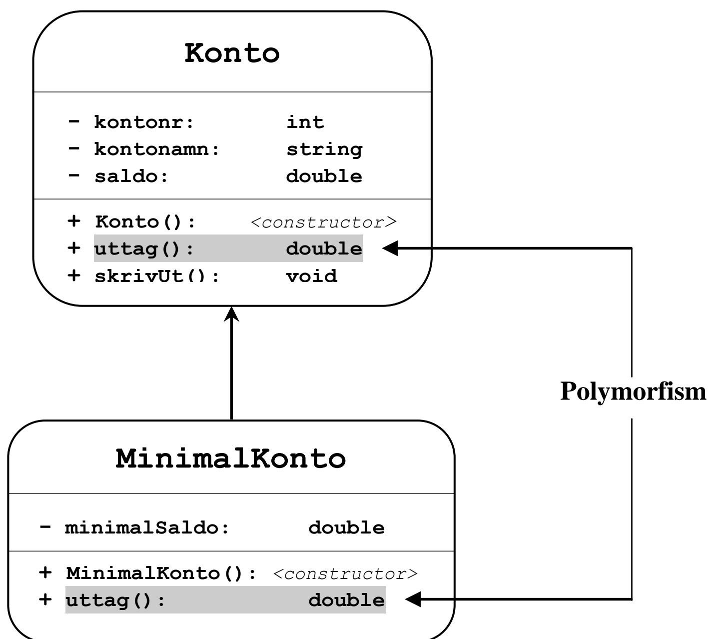
8.3 Polymorfism

”Poly” betyder *många* och ”morf” är *form* eller *gestalt* på latin och gammal grekiska. Polymorfism handlar om en sak som har många olika gestalter, t.ex. ett ord som har många olika betydelser. Den aktuella betydelsen avgörs av sammanhanget. Det vanliga språket är fullt med sådana ord: Ta bara ordet *köra*. Man kan köra bil, köra tåg, köra program osv. Inom programmering pratar man om *överlagring av funktioner* dvs samma namn för olika funktioner. För att kunna skilja åt dem har man olika parameterlistor. Vi har behandlat överlagring av funktioner tidigare (sid 75), och även när vi definierade flera konstruktorer i en och samma klass, t.ex. i klasserna **`Circles`** och **`Date`**.

Polymorfism är en speciell form av överlagring och efter inkapsling och arv den tredje hörnstenen i objektorienterad programmering. Det speciella hos polymorfism som skiljer den från både vanlig överlagring och från flera konstruktorer, är:

- Från vanlig överlagring: De funktioner som har samma namn är inte längre fristående. De är inbundna i klasser och är därmed inte längre funktioner utan metoder. Så är det hos flera konstruktorer i en och samma klass. Kriteriet för att skilja åt konstruktorerna med samma namn från varandra, är fortfarande parameterlistan, närmare bestämt olika antal eller olika typer av parametrar.

- Från flera konstruktorer: De metoder som har samma namn befinner sig inte längre i samma klass. De är placerade i olika klasser som ärver varandra. Låt oss ta följande exempel där klassen **`MinimalKonto`** ärver klassen **`Konto`**:



Polymorfism modifierar helt eller delvis funktionaliteten hos metoder med samma namn som förekommer i en arvhierarki.

Subklassens metod kommer då att överskugga superklassens metod.

En metod beskriver alltid någon funktionalitet. Polymorfism definierar en metod i superklassen och definierar om den (innehållet), men behåller namnet i subklassen.

Metoden **`uttag()`** är definierad både i superklassen **`Konto`** och i subklassen **`MinimalKonto`**. Man har två olika typer av konto i en bank. Operationen ”att ta ut pengar” definieras på olika sätt på dessa två kontotyper, men operationens namn ska vara **`uttag()`**. Utan arvrelation skulle detta vara ett exempel på vanlig överlagring av metoder. Men det tillkommer att **`MinimalKonto`** ”är” ett speciellt **`Konto`** och kan därmed ärva klassen **`Konto`**. Etablerar vi arvrelationen kommer vi att ha att göra med polymorfism. För att se det implementerar vi modellen ovan. Så här kan t.ex. klassen **`Konto`** se ut:

```
// Account.h
// Superklass till subklassen Minimalkonto

class Konto
{
protected:                                    // Datamedlemmar åtkom-
   int    kontonr;                            // liga i subklasser
   string namn;
   double saldo;

public:
   Konto(int nr, string n, double s) // Konstruktorn
   {
      kontonr = nr;
      namn    = n;
      saldo   = s;
   }

   void uttag(double amount)                  // Metod som över-
   {                                          // skuggas i subklassen
     if (saldo - amount < 0)
       cout << "\nIngen täckning på " << namn << "s konto\n";
     else
       saldo = saldo - amount;
   }

   void skrivUt()                             // Utskriftsmetod
   {
      cout << "Kontonr   " << kontonr   << '\n'
           << "Namn      " << namn      << '\n'
           << "Saldo     " << saldo     << "\n\n";
   }
};
```

Överskuggning av metoder

Klassen **`Konto`** beskriver ett vanligt bankkonto med en **`uttag()`**-metod som inte tillåter uttag av pengar om uttagsbeloppet överstiger saldot. En bank har däremot många olika typer av konton. Tänkbart är t.ex. ett konto som alltid behåller ett visst minimalbelopp på kontot och inte tillåter uttag av pengar om saldot efter uttag understiger detta minimalbelopp. Ett sådant specialkonto beskrivs nedan i klassen **`MinimalKonto`** som ett **`Konto`** med en **`uttag()`**-metod som implementerar denna affärslogik.

```
// MinAccount.h
// Sub- eller härledd klass till superklassen Konto

class MinimalKonto : public Konto      // Ärver klassen Konto
{
   double minimalSaldo;                // Ny datamedlem

public:
   MinimalKonto(int nr, string n, double s, double minS)
                                              : Konto(nr, n, s)
   {
      minimalSaldo = minS;
   }

   void uttag(double amount)           // Överskuggar super-
   {                                   // klassens metod
     if (saldo - amount < minimalSaldo)// Inte längre < 0
       cout << "\nIngen täckning på " << namn << "s konto\n";
     else
       saldo = saldo - amount;
   }
};
```

`MinimalKonto` ärver klassen **`Konto`** genom att lägga till den nya datamedlemmen **`minimalSaldo`** och definiera om **`Konto`**-klassens **`uttag()`**-metod. Vi har med två olika metoder **`uttag()`** att göra. I alla objekt av typ **`Konto`** kommer den ena – den ursprungliga – att gälla, i alla objekt av typ **`MinimalKonto`** kommer den andra att gälla. Man säger: Den nya, modifierade metoden **`uttag()`** *överskuggar* den gamla. Dvs en metod i en subklass överskuggar (slår ut temporärt) metoden med samma namn i sin superklass. Överskuggning (eng. *overriding*) är ett koncept som vi redan lärt känna och använt när vi diskuterade lokala och globala variabler. Men då handlade det om *överskuggning av variabler* medan nu har vi att göra med *överskuggning av metoder*.

En konsekvens av att metoderna **`uttag()`** inte längre befinner sig i samma klass, är att de inte längre behöver skiljas åt genom olika parameterlistor. De är redan skilda genom sin placering i olika klasser och kommer därför att anropas i objekt av *olika* klasser. De måste tvärtom ha t.o.m. *samma* parameterlista. För att subklassens metod ska kunna överskugga (slå ut temporärt) superklassens metod, måste metodhuvuden vara exakt identiska. Därför har **`uttag()`** i **`MinimalKonto`** samma huvud som

uttag() i **Konto** (framhävd med vit bakgrund). De skiljer sig endast genom kroppen, närmare bestämt i **if**-satsens villkor: I superklassen implementeras den vanliga policyn för uttag av pengar med **if (saldo - amount < 0)**, medan i subklassen ska den speciella uttagpolicyn gälla: **if (saldo - amount < minimalSaldo)**. Överskuggning av metoder är en direkt konsekvens och en väsentlig ingrediens av polymorfism.

Åtkomstmodifieraren **protected**

När vi diskuterade inkapsling lärde vi känna åtkomstmodifieraren **private**. Innan dess hade vi använt **public**. Det finns ytterligare en åtkomstmodifierare i C++ som heter **protected**. De reglerar åtkomsten till medlemmarna i en klass utifrån klassen. Ställer man upp dem i en rangordning från *restriktiv* till *liberal* får man följande lista:

- **private**
- **protected**
- **public**

private är den mest restriktiva modfieraren och spärrar åtkomsten absolut. Inte ens en subklass har tillgång till superklassens privata medlemmar fast den ärver allt ovanifrån. **public** är den mest liberala modfieraren och friger åtkomsten åt alla utifrån. **protected** är en kompromiss som friger åtkomsten till klassens medlemmar från en subklass och spärrar åtkomsten från alla andra klasser. **private** är den favoriserade default åtkomstmodifieraren i en klass.

I klassen **Konto** är det ganska naturligt att deklarera datamedlemmarna som **protected**. På så sätt skyddas uppgifterna om **kontonr**, **kontonamn** och **saldo** från all kod som inte har att göra med klassen **Konto**. Samtidigt är de tillgängliga från alla klasser som ärver klassen **Konto** dvs är också konton, fast mer specialiserade. Alla dessa specialkonton kommer att ha åtminstone dessa tre grund-datamedlemmar. Med **protected** slipper man skriva set- och get-metoder i subklassen **MinimalKonto**, vilket underlättar programmeringen. Subklassen **MinimalKonto** kan t.ex. i sin **uttag()**-metod komma åt superklassens datamedlemmar **namn** och **saldo** tack vare **protected**. Annars, om **namn** och **saldo** hade varit **private**, hade vi behövt definiera och anropa get-metoder.

Nu när vi testar både **Konto**- och **MinimalKonto**-klassen i en väldigt enkel applikation kan vi konstatera att kompilatorn automatiskt väljer rätt **uttag()**-metod vid anrop – trots samma namn och samma parameterlista:

```
// CreateAccount.cpp
// Demonstrerar anropen av den polymorfa metoden uttag()
// En gång anropas uttag() i ett Konto-objekt (kalle)
// den andra gången i ett MinimalKonto-objekt (pelle)
#include <iostream>
#include <string>
```

```
using namespace std;
#include "Account.h"
#include "100Minimalkonto.h"

int main()
{
   Konto kalle(12345, "Kalle", 100);
   MinimalKonto pelle(67890, "Pelle", 200, 50);
   double ut;

   cout << "Kalles konto före uttag:\n";
   kalle.skrivUt();
   cout << "Pelles konto före uttag:\n";
   pelle.skrivUt();

   cout << "Ta ut ett belopp från Kalles konto: ";
   cin >> ut;

   kalle.uttag(ut);             // Här anropas superklassens
                                // metod uttag()
   cout << "Ta ut ett belopp från Pelles konto: ";
   cin >> ut;

   pelle.uttag(ut);             // Här anropas subklassens
                                // metod uttag()
   cout << "\nKalles konto efter uttag:\n";
   kalle.skrivUt();
   cout << "Pelles konto efter uttag:\n";
   pelle.skrivUt();
}
```

På den första raden i **`main()`** skapas objektet **`kalle`** av typ **`Konto`**, på den andra raden objektet **`pelle`** av typ **`MinimalKonto`**. De initieras med var sin konstruktor. **`kalle`** får **`100`** kr insatt på sitt konto, **`pelle 200`**. Eftersom **`pelle`** har ett **`MinimalKonto`** måste hans konto initieras även med ett värde till den nya datamedlemmen **`minimalSaldo`**. Därför skickas som sista parameter till **`pelle`**-konstruktorn värdet **`50`** som enligt affärslogiken alltid ska vara kvar på ett **`MinimalKonto`**. Så, **`pelle`** får maximalt ta ut 150 kr från sitt konto. Försöker han ta ut t.ex. **`180`** kr – vilket vi gör i körexemplet på nästa sida – godtas inte uttaget och han får meddelandet **`Ingen täckning på Pelles konto`** som har sitt ursprung i anropet **`pelle.uttag(ut)`**. Efter det misslyckade uttagsförsöket är **`pelle`**:s saldo fortfarande **`200`**.

I programmet **`SkapaKonto`** förekommer två anrop av den polymorfa metoden **`uttag()`**, en gång superklassens och en gång subklassens **`uttag()`**-metod:

`kalle.uttag(ut)` och **`pelle.uttag(ut)`**

Att vi kallar metoden för polymorf beror på att det är två *olika* metoder med två olika funktionaliteter (två olika kroppar) med samma namn och samma huvud. Det

är de två olika objekten **kalle** och **pelle** som gör att kompilatorn väljer rätt metod. Men det finns i programmet även två gånger två anrop av metoden **skrivUt()**:

kalle.skrivUt() och **pelle.skrivUt()**

Är det här också två olika metoder? Är även metoden **skrivUt()** polymorf? Svaret är nej, därför att det endast finns en metod **skrivUt()** som är definierad i superklassen **Konto**. Hur kommer det sig då att vi kan anropa den även i **pelle**-objektet som inte är av typ **Konto**? Det kan vi göra därför att **MinimalKonto** som **pelle** är ett objekt av, *ärver* **Konto** och därmed även den publika metoden **skrivUt()**. Därför är metoden **uttag()** polymorf, men inte metoden **skrivUt()**.

Så här kan en körning av **SkapaKonto** se ut:

```
Kalles konto före uttag:
Kontonr   12345
Namn      Kalle
Saldo     100

Pelles konto före uttag:
Kontonr   67890
Namn      Pelle
Saldo     200

Ta ut ett belopp från Kalles konto:  80
Ta ut ett belopp från Pelles konto: 180

Ingen täckning på Pelles konto

Kalles konto efter uttag:
Kontonr   12345
Namn      Kalle
Saldo     20

Pelles konto efter uttag:
Kontonr   67890
Namn      Pelle
Saldo     200
```

Övningar till kapitel 8

8.1 Modellera en arvhierarki över olika typer av anställda och använd på ett polymorft sätt metoden **`lön()`** i alla klasser för att beräkna lönen för de olika anställdtyper. Skriv en superklass **`Anställd`** som ärvs av tre subklasser **`FastAnställd`**, **`Säljare`** och **`TimAnställd`**. Varje subklass ska ha privata datamedlemmar, en konstruktor, set/getmetoder till varje ny datamedlem, en **`toString()`**-metod som skriver ut den anställdas typ, namn och anställningsnummer samt metoden **`lön()`** som i varje subklass definierar om (överskuggar) superklassens metod **`lön()`**. Typiska nya privata datamedlemmar till klassen **`FastAnställd`** kan vara **`månadslön`**, till klassen **`Säljare`** kan tänkas **`bonus`** och **`säljvolym`** och till klassen **`TimAnställd`** t.ex. **`timlön`** och **`antalTimmar`**. Skriv dessutom en subklass **`FastSäljare`** som ärver klassen **`Säljare`** och har den nya privata datamedlemmen **`fastLön`**. Testa dina subklasser i **`main()`** som skrivs i en testklass.

8.2 Modellera en klass **`Cylinder`** som subklass till klassen **`Cirkel`**. Förse superklassen **`Cirkel`** med en privat datamedlem **`radie`**, en konstruktor, en get-metod och med beräkningsmetoderna **`area()`** och **`omkrets()`**. Denna modellering ser **`Cylinder`**n som en **`Cirkel`** som dessutom har en **`höjd`**. Betrakta därför **`Cylinder`**n som en ”utvidgad” **`Cirkel`** som ärver **`Cirkel`**n och lägger till den en privat datamedlem **`höjd`**. Förse även subklassen med en konstruktor och en get-metod. **`Cylinder`**n ska dessutom ha metoderna **`volym()`** och **`yta()`**. Vid beräkning av **`Cylinder`**ns **`volym()`** och **`yta()`** ska koden kunna återanvända **`Cirkel`**ns metoder genom att anropa dem. Testa dina klasser i **`main()`** genom att läsa in en cylinders **`radie`** och **`höjd`** samt skriva ut **`volym()`** och **`yta()`**.

8.3 Skriv en klass **`Rektangel`** med datamedlemmarna **`bredd`**, **`höjd`** och metoderna **`area()`**, **`omkrets()`**. Deklarera datamedlemmarna som **`private`** och metoderna som **`public`**. Förse klassen med en konstruktor och välj andra namn för konstruktorns parametrar än för datamedlemmarna. Testa din klass i en separat fil genom att i **`main()`** skapa ett **`Rektangel`**-objekt vars datamedlemmar initieras till konstanta värden. Skriv ut objektets area och omkrets.

8.4 Modifiera klassen **`Rektangel`** från övn 8.3 genom att lägga till get- och set-metoder i klassen. Testa den nya klassen i **`main()`** genom att *läsa in* värden till datamedlemmarna. Efter utskriften av area och omkrets, fördubbla rektangelns längd och bredd med anrop av get- och set-metoderna. Skriv ut en gång till rektangelns area och omkrets. Med vilken faktor växer arean resp. omkretsen?

8.5 Skriv en klass **`Fisk`** som beskriver en fisk med datamedlemmarna **`sort`**, **`vikt`** och **`längd`**. Förse klassen med en metod **`pris()`** som beräknar priset på fisken som oavsett sort ska kosta 7,25 kr per hekto. Metoden skall returnera priset i kronor. Skriv även en metod **`frakt()`** som beräknar transportkostnaden utifrån fiskens vikt och längd genom att multiplicera kostnadsfaktorn 0,02 med vikten och 0,1 med längden och addera dem. Testa klassen **`Fisk`** i en separat fil genom att i **`main()`** skapa en array av fem **`Fisk`**-objekt vars datamedlemmar tilldelas värden, t.ex. ”laxforell” till **`sort`**, 719 (gram) till **`vikt`** och 38,5 (cm) till **`längd`**. Pris och frakt till varje **`Fisk`**-objekt ska beräknas genom anrop av metoderna **`pris()`** och **`frakt()`**. Skriv sedan ut dessa värden tillsammans med pris och frakt till en tabell som t.ex. kan se ut så här:

C:\WINDOWS\system32\cmd.exe

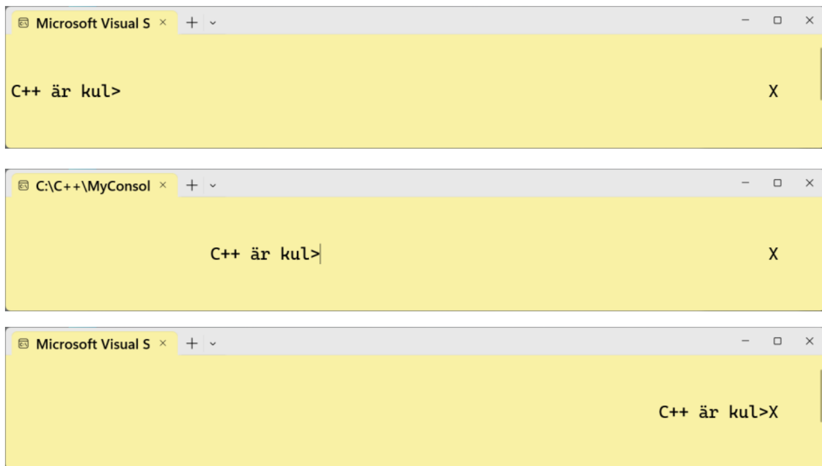
Fisksort	Vikt i g	Längd i cm	Pris	Frakt
Laxforell	719	38.5	52.13	18.23
Torsk	423	28.7	30.67	11.33
Aborre	550	25.5	39.88	13.55
Gõdda	985	58.0	71.41	25.50
G÷s	395	14.0	28.64	9.30

8.6 Deklarera datamedlemmarna i klassen Fisk från övn 8.5 ovan som **`private`** och metoderna som **`public`**. För att initiera de **`private`**-deklarerade datamedlemmarna, förse klassen med en konstruktor. Annars ska programmet göra samma sak som i övn 8.5.

Sex projektuppgifter

1. Löpande texten – en animation i konsolen

Skriv en C++ Console Application som simulerar en löpande text. Ta som exempel texten **`C++ är kul>`** som ska röra sig horisontellt från konsolfönstrets vänstra kant tills den ”träffar” på ett hinder, t.ex. ett kryss **`X`**. Texten ska börja från vänstra kanten. Krysset ska ligga nära den högra kanten (ca. 70-80 tecken borta). Dessa ögonblicksbilder ska illustrera animationen:



Ledning:

Skriv ut först krysset **`X`** i slutet av en tom rad (fylld med mellanslag). Anteckna hur många mellanslag ni har valt för att placera krysset från konsolens vänstra kant. Gå i samma rad tillbaka till radens början genom att använda escapesekvensen **`\r`** (carriage return). **`\r`** skickar tillbaka markören till början av samma rad, utan att byta rad (till skillnad från **`\n`**). Skriv sedan ut **`C++ är kul>`** som då blir textens initialposition – det som visas i den första ögonblicksbilden ovan. Om ni vill bekanta er mer med **`\r`**:s funktion gör experiment med det i ett annat program.

Rörelsen kan sedan simuleras t.ex. i en **`for`**-loop genom att i varje varv av loopen med ett antal **`\b`** ta bort texten som skrevs ut i förra varvet. Escapesekvensen **`\b`** (backspace) tar bort *ett* tecken till vänster om det aktuella tecknet, precis som tangenten backspace (←). Stega sedan med ett (eller flera) mellanslag, vilket kommer att bestämma rörelsens ”hastighet”. Skriv slutligen om texten **`C++ är kul>`**.

Beräkna antalet varv i **`for`**-loopen genom att ta hänsyn till textens längd och avståndet som kryss **`X`** har från vänstra kanten (som antecknats ovan). Har ni räknat rätt, kommer rörelsen att stoppas strax före krysset **`X`**, utan att ta bort det – liknande den tredje ögonblicksbilden ovan.

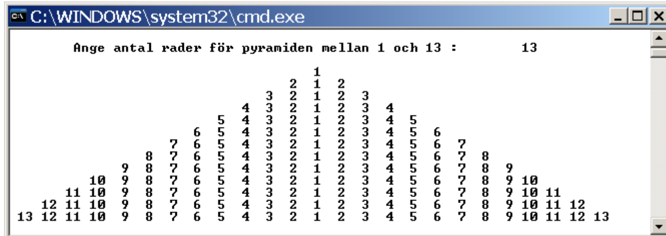
Även om ni gjort allt rätt kommer ni inte ”se” texten att röra sig, eftersom det går så fort, så att ögat inte hinner att se förloppet. Ni måste lägga in en fördröjning, vilket kan göras genom att infoga i loopen t.ex. satsen:

```
Sleep(100);
```

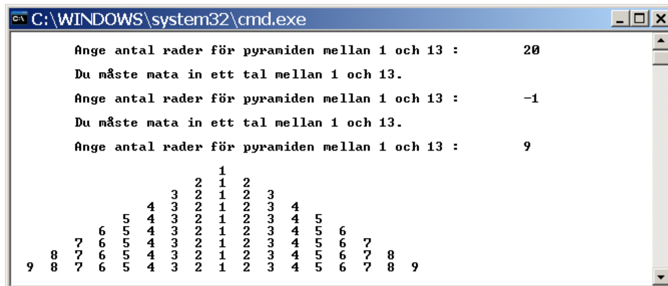
Parameterns enhet är millisekunder. Fördröjningsfunktionen **`Sleep()`** kräver inkluderingen av biblioteket **`windows.h`**.

2. Pyramiden

Slutmålet med detta uppdrag är att utveckla ett program som skriver ut en pyramidliknande figur med tal, som t.ex. ser ut så här:



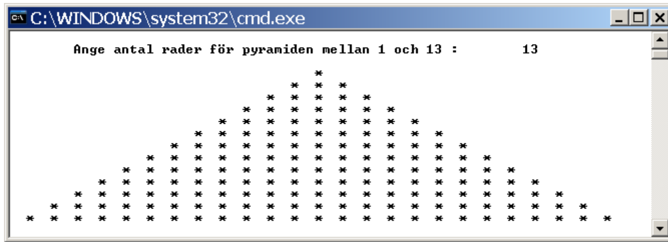
Programmet ska vara så generellt att det skriver ut talpyramider även om man matar in mindre antal rader. Uppmana användaren att hålla sig till talintervallet [1, 13]. Annars ryms talpyramiden inte i konsolen. Så här kan en körning se ut:



Ledning:

Denna ledning är endast en rekommendation och ska inte förhindra att ni använder egna idéer för att lösa problemet. Det finns andra möjliga tillvägagångssätt. Ni kan använda hela eller också delar av denna ledning för att komma igång.

Man kan *börja* med ett program som ritar en pyramid av *stjärnor* istället för tal:



Strunta till att börja med även på hanteringen av felinmatning av antal rader. Jobba med ett fast antal rader. Du kan lägga till det senare.

Använd en nästlad for-sats med en yttre loop och tre inre loopar:

- En för de tomma platserna i pyramiden (mellanslagen),

- En för stjärnorna i pyramidens högra halvan (räknat från den vertikala mittlinjen (symmetriaxeln),

- En för stjärnorna i pyramidens vänstra halvan.

Räkna med att ni måste använda i de inre looparna den yttre loopens räknare och slutvärde. T.ex. kan villkoret i den första inre loop som ritar de tomma platserna, se ut så här:

```
column <= numberOfRows - row;
```

Här är **`column`** den inre loopens, **`row`** den yttre loopens räknare och **`numberOfRows`** hela pyramidens antal rader, t.ex. 13. Då kan den första inre loopen skriva ut tre mellanslag i varje varv. I de två andra inre looparna kan två mellanslag och en **`*`** skrivas ut.

3. Kaffeautomaten

Du får i uppdrag att programmera en kaffeautomat. Uppdragsgivaren förväntar sig ett professionellt program som lätt kan uppdateras, om man skulle byta till en nyare automatmodell om något år. Därför anlitar man en objektorienterad programmerare. Skriv koden så generellt som möjligt så att programmet även kan modifieras för vilken varuautomat som helst, dessutom enkelt kan översättas till vilket programmeringsspråk som helst.

Programmet ska simulera en *aktion* i automaten, dvs det man *gör* med den. I händelsernas centrum

ska finnas en klass som beskriver det som *pågår* i automaten, efter att användaren fått läsa menyn, valt en dryck och stoppat in pengar. Deklarationen till en sådan klass kan – i stora drag – se ut så här:

```
class Coffee_action
{
   string productName;
   double price;
   double payment;
   double change;

   public:
      Coffee_action(char product, double money)
      {
         switch(product)
         {
            . . .
         }

         payment = money;
         change = payment - price;
      }

      void change_in_coins()
      {
            . . .
      }
};
```

Konstruktorn **`Coffee_action()`** ska initiera de privata datamedlemmarna **`productName`** och **`price`** beroende på valet av dryck och skriva ut ett meddelande om inlagt belopp samt drycken som ska levereras. Detta kan med fördel kodas med en **`switch`**-sats (ovan). Efter **`switch`**-satsen initieras även de privata datamedlemmarna **`payment`** och **`change`**.

Skriv ditt huvudprogram i en separat fil. Börja i **`main()`** med att skriva ut en meny över alla varor samt priserna, t.ex.:

K(affe)	12.00 kr
E(spresso)	14.00 kr
C(hoklad)	11.50 kr
L(Kaffe Latte)	13.00 kr
P(Cappuccino)	13.50 kr

Låt sedan användaren välja en dryck genom att läsa in begynnelsebokstaven till varorna ovan med en **`char`**-variabel. Låt användaren sedan lägga in pengar. Läs in beloppet till en **`double`**-variabel. Fortsätt med att skapa ett objekt av klassen **`Coffee_action`** inkl. anrop av konstruktorn. Vid detta anrop skickas till de inlästa värdna, dvs den valda varan samt det inlagda beloppet, som aktuella parametrar till konstruktorn **`Coffee_action()`**.

Ta hand om en ev. felaktig eller otillräcklig betalning från användarens sida genom att ge användaren möjlgheten att komplettera sin betalning.

Efter att objektet skapats och datamedlemmarna initierats via konstruktorn kan metoden **`change_in_coins()`** anropas som ska dela upp växeln i automatens "tillåtna" myntslag (10-kr, 5-kr, 1-kr och 50-öringar) och skriva ut hur många av varje "tillåtet" myntslag som ska ges tillbaka. För att åstadkomma detta kan följande algoritm användas:

Algoritm för omvandling av ett belopp till olika myntslag *

Eftersom denna algoritm endast fungerar för heltal, måste **`change`** som är ett belopp i kronor och ören av typ **`double`**, först räknas om till ett rent örebelopp av typ **`int`**, vilket kan göras genom att multiplicera det först med **`100`** och sedan omvandla till **`int`**:

```
int total = (int) (change * 100);
```

I fortsättningen kommer alltså den givna växeln att stå som ett *öre*belopp i **`int`**-variabeln **`total`**. Gör så här för att få antalen "tillåtna" myntslag:

1. För att få antalet 10-kronor heltalsdivideras **`total`** med **`1000`** eftersom 10-kronor är **`1000`** ören:

```
int ten = total / 1000;
```

Hur många gånger ryms **`1000`** – eller 10-kronor – i **`total`**? Det antalet tilldelas till **`ten`**. Eller med andra ord: **`1000`** dras av från **`total`** så många gånger tills resten blivit mindre än **`total`**. Det antalet som tilldelas till **`ten`** blir antalet 10-kronor. Divisionen ovan är inte vanlig division utan heltalsdivision eftersom både **`total`** och **`1000`** är heltal. Dvs **`total`** divideras med **`1000`**, resultatet tas, resten ignoreras, t.ex. **`6975/1000`** ger **`6`**. Resten **`975`** ignoreras här, men används i fortsättningen.

2. För att få antalet 5-kronor divideras just *resten* som blev kvar från punkt **1** med **`500`** eftersom 5-kronor är **`500`** ören:

```
int five = (total % 1000) / 500;
```

Här används modulooperatorn **`%`**. "Resten som blev kvar från punkt **1**" är just **`(total % 1000)`**. T.ex. **`6975 % 1000`** ger **`975`**. Efter att ha dragit av alla 10-kronor från **`total`** divideras resten med **`500`** för att få reda på hur många 5-kronor som finns i **`total`**. T.ex. **`975/500`** ger **`1`**. Resultatet av denna division ges till **`five`**, resten ignoreras och används i fortsättningen.

---

* Myntbetalningen inkl. behandlingen av 50-öringen beror inte på nostalgi utan på internationalisering. Vi vill hålla möjligheten öppen för en överföring av programmet till andra länder där automater med myntbetalning fortfarande finns. Även ett ev. byte till Euro eller andra valutor där den halva valutaenheten finns kvar, ska vara möjligt. Omvandlingen av växelbeloppet till automatens myntsystem inkluderar en programmeringsteknisk finess som kan vara värd att lära sig. Logiken inkl. användningen av modulooperatorn ligger till grund även för en generell omvandling av det decimala talsystemet till andra system.

I ytterligare tre steg kan de övriga formlerna för beräkning av antalet 1-kronor (**`one`**), 50-öringar (**`half`**) och resten i öre (**`rest`**) skrivas, när mönstret i algoritmen (förhoppningsvis) har trätt fram:

```
int one  =  ((total % 1000) % 500) / 100;
int half = (((total % 1000) % 500) % 100) / 50;
int rest = (((total % 1000) % 500) % 100) % 50;
```

Man tar förra stegets formel, ersätter **`/`** med **`%`** och lägger till en heltalsdivision med den nya enhetens örebelopp. I det allra sista steget däremot, där man är ute efter allra sista resten i öre, måste **`%`** användas hela vägen. Självklart är restörebeloppet inte av praktiskt intresse när automaten inte kan spotta ut det.

4. Frekvenstabell – stämmer sannolikhetsläran?

Följande program simulerar tärningskast genom att generera slumptal mellan **`1`** och **`6`**. Resultatet skrivs ut i tabellform.

```
// Dice.cpp
// Simulerar tärningskast: Slumpar fram tal mellan 1 och 6
// och skriver ut dem i en tabell. Nästlad for-sats
#include <iostream>
using namespace std;

int main()
{
   srand(time(0));              // Skapar variation i slumpen
   int r, k, rader, kolumner;
   cout << "\nAnge antal rader och kolumner (t.ex. 10 15): ";
   cin >> rader >> kolumner;
  cout<<"\nDet blir "<< rader*kolumner<<" tärningskast:\n\n";
   for (r=1; r<=rader; r++)     // Låter en rad att skrivas ut
   {                            // och byter rad.
     for (k=1; k<=kolumner; k++)// Skriver ut den r:te raden.
         cout << 1 + rand() % 6 << "   ";
     cout << '\n';
   }
}
```

Testa programmet **`Dice`** och vidareutveckla det. Det nya programmet ska undersöka sannolikhetslärans sats om att i idealfallet sannolikheten för ett utfall vid tärningskast är **1/6** och att det praktiska resultatet närmar sig idealfallet, ju större antalet slumpförsök blir. Genomför denna undersökning genom att ställa upp en *frekvenstabell*. Nedan beskrivs frekvenstabellen:

Frekvens är *antalet* förekomster av ett resultat (utfall) bland tärningens **`6`** möjliga.

Låt programmet genomföra olika antal simuleringar och räkna vid varje simulering frekvensen för varje resultat **`1`**, ... ,**`6`** av tärningskastet. T.ex. ska man kunna läsa av från tabellen hur många gånger resultatet **`1`** förekommer när man kastar tärningen 50

gånger, 100 gånger, 1 000 gånger, 5 000 gånger, 10 000 gånger, osv. Avgör själv hur långt du går. Samma information ska man kunna läsa av från tabellen om tärningskastets andra resultat **2**, ... ,**6**.

Infoga i tabellen även en kolumn som för varje resultat av tärningskastet visar kvoten:

Frekvens / Antalet tärningskast

Denna kvot är den experimentella sannolikheten för ett visst resultat.

Undersök på vilket sätt den experimentella sannolikheten närmar sig den ideala sannolikheten för varje resultat, som enligt sannolikhetsläran borde vara **1/6** eller **0,16667**.

5. Palindrom – en lek med ord

En *palindrom* är en sträng som inte ändras när den läses baklänges. T.ex. är orden *rar*, *död* och *radar* palindromer, även namnet *Hannah* när det stavas så. Men även en text som *ni talar bra latin* är en palindrom om man ignorerar mellanslagen. Och det ska man göra. Därför: vid behandling av sådana texter i ett program låt koden först ta bort alla mellanslag.

Skriv en funktion **`bool palindrom(char *a)`** som avgör om en sträng är en *palindrom* eller ej. Anropa sedan funktionen i **`main()`** efter inmatning av en sträng som ska testas, i ett C++ program som hanterar strängar med pekare. Låt användaren mata in strängar – med eller utan mellanslag – så länge tills man hittat en palindrom. Bjud på möjligheten att avsluta om ingen palindrom hittas och föreslå användaren ett antal palindromer.

6. Collatz algoritmen – rekursiv

I kursen *Programmering med C++* behandlades *Collatz algoritmen* som alltid slutar med 1 oavsett startvärde – ett empiriskt resultat som matematiskt är obevisat:

> Tänk dig ett positivt heltal (startvärde).
> Är talet udda multiplicera det med 3 och addera 1.
> Är talet jämnt dividera det med 2.
> Gör samma sak med resultatet. Fortsätt **tills** du fått 1.

Då implementerade vi Collatz algoritmen *iterativt* med en **`do`**-loop:

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

int main()
{
   int no;
   cout << "\n\tMata in ett pos.heltal:\t";
   cin >> no;
   cout << "\n\t" << no;

   do
   {
      if (no % 2 == 1)
         no = 3 * no + 1;
      else
         no = no / 2;
         cout << "\n\t" << no;
   } while (no != 1);

   cout << "\n";
}
```

Skriv ett C++ program som implementerar Collatz algoritmen med en rekursiv funktion och anropar den från main().

Ledning: Läs kap **2.6 Rekursion** i kursboken, sid 50-53.

Undersök om fenomenet *beräkningskomplexitet* (sid 53) som observerades i Fibonacci rekursionen, även uppträder här. Förklara orsaken. Resonera om för- och nackdelarna av Collatz algoritmens iterativa och rekursiva implementering.