Programmering

C/C++

Med övningar,
fullständiga lösningar
&
projektuppgifter

TechPages Förlag AB

Titel: Programmering C/C++
ISBN:

Copyright © 2022 TechPages. All rights reserved.
www.taifun.se

Tryckeri: Eprint, Stockholm
September 2022



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.

Om boken

Välkommen till programmeringens spännande värld! När man tröttnat på att bara använda program som andra skrivit, är det dags att börja programmera själv. Visst är det roligare att köra en bil än att bara åka med. Det är kreativiteten och det fria skapandet som lockar. Programmering kan vara en naturlig fortsättning för dig som hittills endast har mailat, surfat, lyssnat på musik eller tittat på film på datorn och nu vill veta mer om vad som händer bakom kulisserna. Man lär sig nämligen på ett helt nytt plan hur datorer fungerar när man programmerar själv. Dessutom kan man testa sina egna, nya idéer.

Programmering är ett av de mest spännande kapitlen i teknologihistorien. Inte bara därför att den har lagt grunden till den moderna IT-industrin. Den har också bidragit till att förverkliga den urgamla mänskliga drömmen att förenkla mödosamma arbeten. Istället för att plåga sig kodar man en maskin med idéer, för att ha mer tid över för annat i livet.

Meningen med boken är att lära ut programmering. Detta kan dock praktiskt åstadkommas endast genom att skriva och testa program, dvs använda ett programmeringsspråk. I denna bok används C++ som medel, verktyg och medium för att presentera programmering. Men medlet är av underordnad betydelse. Målet är att förmedla *tankesättet* och *tekniken* att programmera, oberoende av språk. Har man en gång förstått de grundläggande principer som är gemensamma för alla programmeringsspråk, blir det närmast en teknikalitet att på egen hand lära sig ett nytt språk. Denna bok är en introduktion till programmering och förutsätter inga förkunskaper.

Vi som skrivit boken har många års erfarenhet av undervisning i programmering, databaser, matematik, numerisk analys och andra ämnen både på skol- och högskolenivå i olika länder. I vårt material eftersträvar vi enkelhet och klarhet som resulterar i strukturerade och logiska program så att man lätt kan se *idén* och förstå *tanken* bakom koden.

I början av våra banor som pedagoger antog vi att vissa begrepp, sammanhang och förutsättningar var självklara, men den dagliga undervisningen i klassrum fick oss snart på andra tankar. Våra elevers frågor, kritik och kommentarer fick oss att förstå var de begreppsmässiga luckorna i våra resonemang fanns. De var våra *elever* i programmering, matematik osv. men blev våra *lärare* i pedagogik.

Som experter i ämnet har man för länge sedan glömt vilka svårigheter man själv upplevde som nybörjare. Ska man förklara stundtals ganska komplexa sammanhang krävs det förstås ett intresse och en personlig fallenhet för ämnet, men det räcker inte. Det behövs också ett koncept om hur kunskapen ska *förmedlas* för att verkligen nå läsaren. Boken försöker att ta maximal hänsyn till denna aspekt utan att gå miste om den vetenskapliga noggrannheten och utan att tappa djupet i sak när det behövs.

Röda trådens pedagogik

Böcker i tekniska ämnen är ofta rena faktasamlingar vilket kan vara en konsekvens av ämnenas komplexitet. När de är skrivna för experter behöver det inte heller vara av nackdel. Men när nybörjare ska introduceras till ett ämne blir det problem om boken inte kombinerar kunskap med pedagogik. Då blir läroböcker ofta en ambitiös samling fakta som inte framhäver det väsentliga. Oftast handlar det om elementär kunskap som experten tar för given, men blir den bristande länken i förståelsekedjan hos nybörjaren. Bokens ambition är att förverkliga den röda trådens pedagogik genom att stiga ned till nybörjarens kunskapsnivå och steg för steg bygga upp kunskapens hus av små lösa, logiskt härledbara pusselbitar så att till slut allt faller på plats.

Learning by doing – teaching by example

Programmering är i allra högsta grad ett praktiskt ämne. Därför är *Learning by doing* det enda sättet att lära sig det. I detta avseende liknar programmering bilkörning. Du kommer aldrig att lära dig programmering enbart genom att läsa böcker. Men att bara ”pröva sig fram” räcker inte heller. Ämnet är alltför omfattande. En handledning behövs, inte minst i början, som kombinerar sakkunskap med pedagogik, belyser det väsentliga och tillämpar ett helhetskoncept.

Boken håller inga abstrakta lektioner utan följer principen *teaching by example* dvs exempelorienterad teoriundervisning i kombination med praktiska övningar: All teori, även de mest abstrakta begreppen åskådliggörs med enkla praktiska exempel. Fullständiga, testade program med körexempel – och inte bara korta kodsnuttar – gås igenom i detalj för att förmedla viktiga koncept inom programmering. Ännu mer material presenteras i övningarna inkl. praktiska programmeringsprojekt. Fullständiga lösningar till alla öningsuppgifter finns i slutet av boken.

Gör så här:

- Ladda ned och installera programvaran Visual Studio (sid 10).
- Gå igenom boken avsnitt för avsnitt, program för program. En fullständig programförteckning finns i slutet av boken (sid 270).
- Gör övningarna i slutet av varje kapitel. Kolla lösningarna (sid 231).
- Genomför programmeringsprojekt som finns bland övningarna.
- Pröva dina idéer i egna program och återvänd till teorin.

All form av kritik, korrekturanmärkningar såväl som förslag till förbättringar av både form och innehåll tas tacksamt emot på adressen **`info@taifun.se`**.

Stockholm september 2022 TechPages Förlag AB

Innehåll

Ämne	Sida	Program
Kapitel 1 Programmeringsmiljön	**9**	
1.1 Installation av Visual Studio	10	
1.2 Konfiguration och användning av Visual Studio	11	
- Projekt i Visual Studio	12	
1.3 C++ Console applications	13	**MyFirst**
1.4 ETT projekt för alla konsolapplikationer	18	
- Organisera dina C++ filer	19	
Övningar till kapitel 1	20	**MySecond**
Kapitel 2 Introduktion till programmering	**21**	
2.1 Vad är programmering?	22	
2.2 Kompilering och exekvering	24	
- John von Neumann-modellen	24	
- Olika typer av fel	27	
2.3 Från C till C++	28	
Övningar till kapitel 2	31	
Kapitel 3 Att komma igång med C++	**33**	
3.1 Vårt första C++ program	33	**MyFirst**
- Program i C/C++	34	
- Funktionen **main()**	35	
- Inkludering av bibliotek	36	
- Programmet **MyFirstSwed**	37	**MyFirstSwed**
3.2 God programmeringsstil	39	**MyFirst_bad**
3.3 Utmatning med cout	41	**Cout** (Si-out)
3.4 Konkatenering	44	**Concat**
- Att "rita" i textmiljö	45	**Figure**
Övningar till kapitel 3	46	
Kapitel 4 Grundbegrepp i programmering	**48**	
4.1 Datatyper	49	**Datatype**
4.2 Variabler	53	
- Regler för namngivning	53	
4.3 Deklaration och initiering av variabler	55	**Variable**
- Deklaration och initiering i samma sats	57	**DefInitial**
- Oinitierade och väl definierade variabler	58	**NoInitial**
4.4 Överskrivning eller kan **x = x + 1** vara sant?	60	**OverWrite**
4.5 Inläsning av data	62	**Cin** (Si-in)
4.6 Inmatning – Bearbetning – Utmatning	64	**Hour2Sec**
4.7 Modulooperatorn	66	

Ämne	Sida	Program
4.8 Ökningsoperatorn **++**	67	**Increment**
4.9 Sammansatt tilldelning	69	**CompAssign**
Övningar till kapitel 4	72	
Kapitel 5 Enkla datatyper	**74**	
5.1 Kan datorn lagra hur stora tal som helst?	75	**Primitives**
- Operatorn sizeof	76	
- Overflow	78	**Limits**
5.2 Datatypen **char**	79	**Char**
- unsigned-typerna	80	
5.3 Explicit typkonvertering	81	**Char2int**
	82	**Int2char**
5.4 ASCII-tabellen	83	**Ascii**
5.5 Escapesekvenser	86	**Escape**
Övningar till kapitel 5	88	
Kapitel 6 Kontrollstrukturer	**90**	
6.1 Vad är kontrollstrukturer?	90	
6.2 Enkel selektion: **if**-satsen	92	**SimpleIf**
- Villkor	93	
- Jämförelseoperatorer	94	
- Flera satser i **if**	95	
- Algoritm för platsbyte	95	**MiniSort**
6.3 Tvåvägsval: **if**-else-satsen	98	**IfElse**
6.4 Flervägsval	100	
- **if-else**-stegen	101	**GissaTal_1**
- **switch**-satsen	100	**Switch**
6.5 Efter-testad repetition: **do**-satsen	105	**GissaTal_2**
6.6 För-testad repetition: **while**-satsen	108	**While**
- Evighetsslinga	109	
6.7 Bestämd repetition: **for**-satsen	110	
- Översättning av while till for	111	
- while vs. for	112	**While->For**
- ASCII-tabellen med **for**	113	**AsciiFor**
6.8 Nästlade **for**-satser	116	**Stars**
- Hantering av slumptal	117	**NestedFor**
- Multiplikationstabellen	120	**MultipTab**
Övningar till kapitel 6 (Projekt *Labyrint I*)	123	

Ämne	Sida	Program
Kapitel 7 Funktioner	**126**	
7.1 Exempel på en funktion	127	`totalsek()`
- Vad händer när en funktion anropas?	128	`MyFirstFct`
- Placering av funktioners definition	131	
7.2 Funktionsbegreppet i programmering	132	
7.3 Definition och anrop av funktioner	135	
7.4 Deklaration av funktioner	138	`Random`
7.5 Externlagrade funktioner	141	`ExternFct`
	142	`Netto`
7.6 Funktioner utan returvärde	144	`Compare`
	145	`GuessVoid`
7.7 Lokala och globala variabler	147	`Global`
- Blockstruktur	148	
7.8 Överskuggning av variabler	152	`Scope`
- Räckviddsoperatorn	154	
Övningar till kapitel 7	155	
Kapitel 8 Arrays	**158**	
8.1 Vad är en array?	159	
8.2 Definition och initiering av en array	161	`ArrayDef`
- Initieringslista	163	`ArrayInit`
8.3 Stränghantering med array	166	`ArrayChar`
- Nolltecknet	166	`NullChar`
8.4 Kryptering av text	174	`EncryptTest`
	175	`Encrypt`
Övningar till kapitel 8 (Projekt *Labyrint II*, *Master Mind*)	177	
Kapitel 9 Pekare	**179**	
9.1 Vad är en pekare?	180	
9.2 Deklaration och initiering av en pekare	182	`Pointer`
9.3 Adress- och värdeoperatorn	185	`Value`
9.4 Operatorn **`new`**	191	`New`
9.5 Pekare och array	195	`PointArray`
- Pekararitmetik	197	`PointArithm`
9.6 Stränghantering med pekare	199	`Initials`
Övningar till kapitel 9	204	
Kapitel 10 Klasser	**205**	
10.1 Vägen från C till C++	206	`All_in_main`
	207	`Procedure`
- Vår första klass	208	`Circle`
- Test av klass	211	`Obj_Oriented`
- Objekt och klass	212	

Ämne	Sida	Program
10.2 Klass som egendefinierad datatyp	214	
- Deklaration av en klass	215	**`Employee`**
- Definition av ett objekt	217	**`EmployeeTest`**
- Åtkomst till objektets medlemmar	220	
10.3 Metoder	223	**`TravelTime`**
- Objekt som parameter och returvärde	224	**`Travel_Test`**
Övningar till kapitel 10 (Projekt *Automaten*)	228	
Lösningsförslag till övningar (Facit)	231	
Programförteckning	270	
Sakregister	273	

Kapitel 1

Programmeringsmiljön

	Ämne	Sida	Program
1.1	Installation av Visual Studio	10	
1.2	Konfiguration och användning av Visual Studio	11	
	- Projekt i Visual Studio	12	
1.3	C++ Console applications	13	**`MyFirst`**
1.4	ETT projekt för alla konsolapplikationer	18	
	- Organisera dina C++ filer	19	
Övningar till kapitel 1		20	**`MySecond`**

1.1 Installation av Visual Studio

Programmering är i allra högsta grad ett praktiskt ämne.

1) Gå till webbadressen: https://visualstudio.microsoft.com/vs/

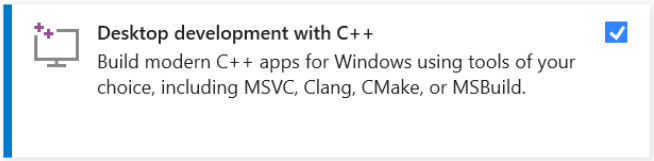
   Gå med musen över knappen: Download Visual Studio

   En dropplista dyker upp. Välj: Community 2022

2) Installationsfilen VisualStudioSetup.exe laddas ner. Dubbelklicka på den. Svara Ja på frågan om du ska tillåta att den här appen får göra ändringar på din dator. Fortsätt med Continue när det dyker upp rutan Visual Studio Installer.

3) Ett stort vitt fönster öppnas. Sikta på rubriken Installing – Visual Studio Community 2022 … och fliken **Workloads**. Leta bland rutor som visas efter följande ruta:

4) Markera rutan Desktop development with C++ genom att bocka den lilla blå rutan i det övre högra hörnet.



5) Klicka sedan på **Install** i det nedre högra hörnet av fönstret Installing – Visual Studio Community 2022 … . Installationsprocessen sätts igång vilket ev. kan ta ganska länge – beroende på din dators prestation och din Internet-uppkoppling.

6) När du får meddelandet Done installing har du lyckats med att installera Visual Studio som startas automatiskt. Annars kan du göra det själv från Start-knappen. Stäng Visual Studio Installer. Följande eventualiteter kan inträffa:

   - Du kan uppmanas att skapa ett Microsoft-konto (Sign in). Gör det. Det är gratis och går fort. Du kommer att behöva kontot senare för uppdatering. Därför: anteckna ditt användarnamn och lösenord.

   - Du kan få upp en ruta med bl.a. dropplistan Development Settings. Välj C++ eller låt General stå där. Följ anvisningarna. Klicka sedan på knappen Start Visual Studio.

7) Fortsätt med att läsa om ***Konfiguration och användning av Visual Studio*** i nästa avsnitt, där några begrepp i Visual Studio förklaras. Följ sedan instruktionerna om ***C++ Console Applications*** på sid 13, för att testa ditt första C++ program.

1.2 Konfiguration och användning av Visual Studio

Efter lyckad installation av Visual Studio enligt anvisningarna ovan kan du läsa i detta avsnitt hur man *använder* miljön för att kunna skriva och testa C++ program. Det vi gör här kallas för *konfiguration* av programvaran, dvs vi definierar vissa inställningar i programvaran som är nödvändiga för att kunna skriva C++ program, även kallad *källkod*, översätta den till *maskinkod*, även kallad *kompilering* och slutligen för att *exekvera* maskinkoden, dvs att utföra programmet.

Visual Studio är en omfattande och komplicerad programvara som är skapad för professionella utvecklare, inte för nybörjare. Vi vill i denna beskrivning hålla oss endast till det absolut minimala som är nödvändigt för att klara av miljön och testa våra koder. Detta för att kunna koncentrera oss på själva *språket C++* som är komplicerat nog. De viktigaste stegen i konfiurationen av Visual Studio är följande:

- Att välja rätt typ av applikation
- Att skapa ett *projekt*
- Att lägga till en C++ källkodsfil till projektet
- Att *kompilera* och *exekvera* C++ koden i projektet

För att kunna genomföra dessa steg och framför allt kunna beskriva dem, behöver vi klarhet över några begrepp i Visual Studio som tas upp nedan, speciellt begreppen *typ av applikation* och *projekt*.

Begreppen *kompilering* och *exekvering* kommer att preciseras senare (sid 25 & 26).

Olika typer av applikation

Det finns olika typer av C++ program, även kallat *applikation.* Begreppen program och applikation är synonym. I Visual Studio finns det en uppsjö av olika typer av applikation som kan köras under Windows. Bland dem nämner vi endast två:

- *Console Application*
- *Windows Desktop Application*

Console Application är ett C++ program vars resultat visas i textform. Utskriften hamnar i ett fönster som heter *Console*. I själva verket är det *Windows Kommandotolk*. Vi kommer i denna bok att behandla endast denna typ av applikation.

Windows Desktop Application är ett program vars körresultat är ett grafiskt användargränssnitt, på eng. *Graphical User Interface (GUI)*. Det är ganska komplicerat att koda grafiska applikationer i C++.

För *Console Applications* går vi igenom alla konfigurationssteg i nästa avsnitt.

Projekt i Visual Studio

Kan man skriva C++ kod i en fil, spara den, ladda den i Visual Studio och köra sin kod? Svaret är nej! För att kunna köra C++ kod i Visual Studio måste koden infogas i ett s.k. *projekt*. Inget annat fungerar i Visual Studio. Så, innan vi kan börja skriva C++ kod måste vi antingen skapa ett nytt eller öppna ett befintligt projekt.

Ett *projekt* är en samling filer som sammanlagt utgör ett C++ program. Denna samling filer bildar både en virtuell arbetsplats i Visual Studio och en fysisk mapp på hårddisken. Dessa två kommunicerar med varandra hela tiden när vi utvecklar och testar våra program. Självklart kan samlingen innehålla endast en källkodsfil.

Den övergripande termen till projekt i Visual Studio är *solution*. Dvs flera projekt kan samlas i en solution. Självklart kan en solution även bestå av ett enda projekt. Vi kommer till att börja med inte att använda flera projekt i en solution utan endast *ett* projekt. Ändå kommer vårt projekt att automatiskt vara paketerat i en solution.

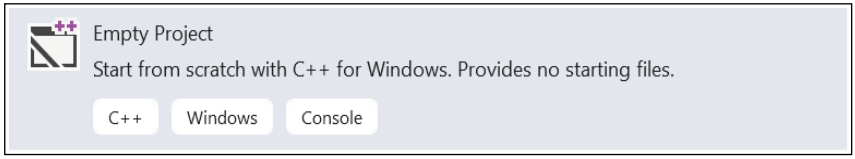
1.3 C++ Console Applications

Starta Visual Studio från Windows *Start*-meny genom att klicka fram dig till:

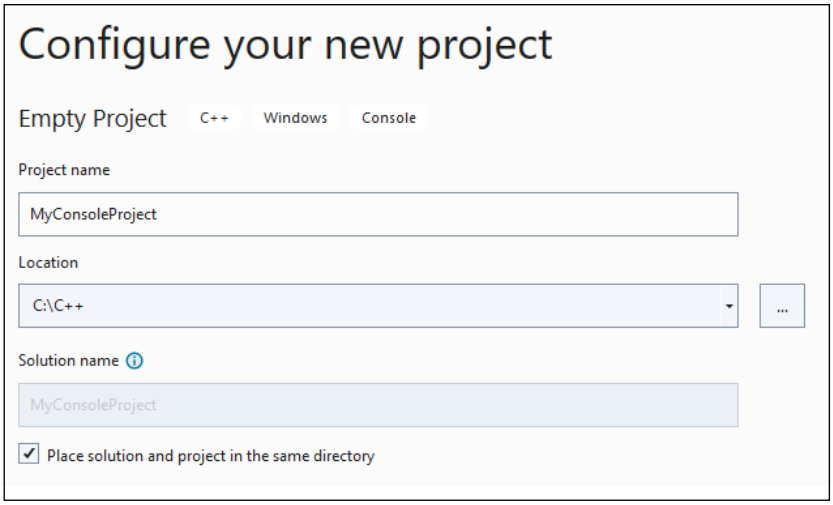
Start → Visual Studio 2022

1. **Att skapa eller öppna ett befintligt projekt:** Beroende på om vi vill skapa ett nytt eller öppna ett befintligt projekt tar vi ett av alternativen **a** eller **b**:

   a. Om vi vill skapa ett nytt projekt – och det vill vi nu – klickar vi i Visual Studio 2022-fönstret till höger under rubriken Get started på rutan Create a new project. En ny dialogruta dyker upp med rubriken Create a new project. Markera rutan med rubriken Empty Project som ser ut så här:



   Klicka sedan på knappen Next. En ny dialogruta dyker upp med rubriken Configure your new project. Fyll i den uppgifterna enligt följande:



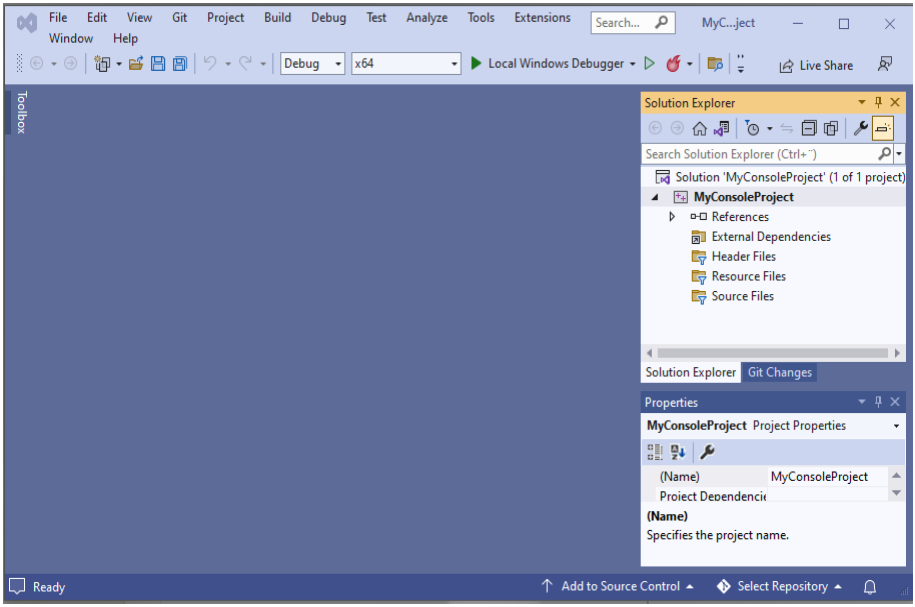
   I den övre delen av dialogrutan ovan döper vi vårt projekt till MyConsoleProject. I textrutan Location anger vi sökvägen till den mapp vi vill placera vårt projekt i. Låt oss säga vi vill samla våra C++ program i en mapp som vi kallar för C++ och placera den i enheten C:\ på vår dator. I så fall anger vi som Location C:\C++. I denna mapp kommer nu projektmappen MyConsoleProject att placeras. Visual Studio skapar automatiskt både den nya mappen. Bocka för den lilla rutan Place solution and project in the same directory. Klicka på knappen Create. Gå till punkt **2**.

b. Om vi vill öppna ett redan befintligt projekt – det gör vi kanske senare – klickar vi i det vita Visual Studio 2022-fönstret på rutan

Open Project …

Vi får upp dialogrutan Open Project/Solution. För att öppna det projekt vi vill jobba med, navigerar vi i datorns filsystem till projektmappen och öppnar där filen med ändelsen **`.vcxproj`**. Gå till punkt **2**.

2. Att lägga till en C++ källkodsfil till projektet: Efter att ha lämnat dialogrutan Configure your new project med Create-knappen enligt **1. a)** eller dialogrutan Open Project/Solution med Open-knappen enligt **1. b)** öppnas projektet. Ett grafiskt gränssnitt kommer upp bestående av en massa menyer, flikar, länkar och fönster osv. som ser ut så här:



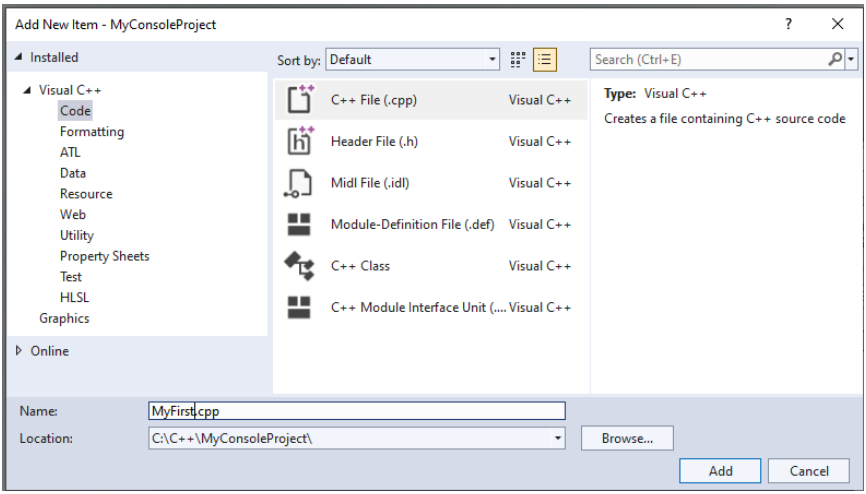
Visual Studio-fönstret har nu i den översta menyraden ett antal menyer, bl.a. menyn Project. Sedan hittar man till höger i det mindre fönstret Solution Explorer, bl.a. rubriken **MyConsoleProject**. Vårt projekt är alltså skapat vars innehåll visas i Solution Explorer-fönstret. Dock saknar projektet just nu en källkodsfil, dvs ett C++ program. Vi valde ju i steg 1 uttryckligen ett Empty Project. Vi måste lägga in en C++ källkodsfil i projektet. Det finns två alternativ att göra det: Antingen vill vi skapa ett helt nytt program, skriva in koden, spara den i en fil och infoga den i projektet eller vi vill lägga till en redan befintlig fil som innehåller ett C++ program, dvs en C++ källkodsfil som vi kanske har från tidigare. Vi börjar med det första:

a) Att skapa en ny fil och infoga den i projektet:

Högerklicka på **MyConsoleProject** i Solution Explorer-fönstret. Välj där:

→ Add → New Item…

Följande dialogruta dyker upp:



Välj först Code i kolumnen Installed till vänster under rubriken Visual C++ alternativet. Markera sedan i den mellersta kolumnen Code File (.cpp). Namnge i textrutan Name filen genom att skriva t.ex. MyFirst.cpp i den. I textrutan Location står redan den fullständiga sökvägen till projektmappen C:\C++\MyConsoleProject. Du kan, om du vill, ange en annan plats, om du vill separera dina C++ källkodsfiler och samla dem i en annan mapp och strukturera mappen efter eget godtycke, kanske efter bokens kapitel. Annars låt den redan angivna projektmappen stå. I så fall kommer filen MyFirst.cpp att skapas där.

Klicka på Add-knappen för att infoga källkodsfilen MyFirst.cpp i projektet.

Så snart du klickat på Add-knappen återvänder du till den ursprungliga miljön där det nu skapats ett stort vitt fönster i en ny flik med namnet MyFirst.cpp, som du kan använda som en editor för att skriva C++ kod i. Skriv in där koden till vårt första C++ program:

Programmet `MyFirst`

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

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

En utförlig förklaring av koden i detta program ges på sid 34. Vi kommer att referera till det i fortsättningen med programnamnet **`MyFirst`**.

b) Att lägga till en befintlig fil till projektet:

Har du redan en C++ källkodsfil bland dina filer på hårddisken, Högerklicka på **MyConsoleProject** i fönstret Solution Explorer. Välj där:

→ Add → Existing Item…

Dialogrutan Add Existing Item – MyConsoleProject dyker upp som tillåter dig att navigera genom datorns fil- och mappsystem för att hitta och ladda en existerande C++ källkodsfil. Gå till den fil du vill ladda, markera den och klicka på knappen Add i dialogrutan Add Existing Item – MyConsoleProject. Filen läggs nu till projektet och du kan i Solution Explorer-fönstret konstatera att den fil du valde har kommit till projektet MyConsoleProject. Dubbelklicka på den, så bildas en ny flik med filens namn. Filens innehåll dvs C++ källkoden kommer att visas i den nya fliken som nu kan användas som en editor.

3. **Att kompilera och exekvera:** Nu när projektet är skapat och innehåller en C++ källkodsfil kan man *kompilera* det vilket betyder att översätta källkoden till maskinkod (sid 25). Detta kan göras från menyraden längst upp med:

Build → Compile

Om kompileringen gått bra får du följande utskrift i Output-fönstret:

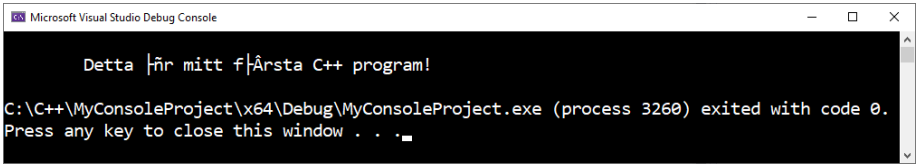
```
Build started...
1>------ Build started: Project: MyConsoleProject, Configuration: Debug x64 -
------
1>MyFirst.cpp
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Meddelandet ovan, närmare bestämt `0 failed`, säger att koden inte innehåller några kompileringsfel. Du kan gå vidare och exekvera den. Annars, om du har syntaxfel i koden kommer du att få felmeddelanden i Output-fönstret. Dessutom öppnas ytterligare ett fönster med rubriken Error List, där du kan spåra dina fel. Åtgärda alltid endast det allra första kompileringsfelet och kompilera om med kommandot ovan, eftersom de andra kan vara följdfel. De kan försvinna (delvis) om du åtgärdar första felet.

Att *exekvera* betyder att utföra den maskinkod som kompilatorn genererat (sid 26). För att exekvera koden klicka i menyraden längst upp på menyn:

Debug → Start Without Debugging

Tyvärr heter denna aktion i Visual Studio inte Execute, vilket hade varit önskvärt. Om allt har gått bra bör det se ut så här på din skärm:



Får du detta på skärmen har du lyckats med att kompilera och exekvera den kod du matade in på förra sidan.

Grattis!

Du har skapat och testat en C++ Console Application. Programmet **`MyFirst`** finns nu lagrad i filen MyFirst.cpp och den i sin tur i projektet MyConsoleProject. Men du borde helst spara den också på en plats som är oberoende av projektet i Visual Studio. Läs mer om detta under nästa avsnitts rubrik ***Organisera dina C++ filer***, sid 19.

Om du undrar varför de svenska tecknen **`ä`** och **`ö`** är förvrängda läs på sid 37.

Om du vill ändra bakgrundsfärgen, storleken osv. på konsolfönstret kan du högerklicka på den lilla ikonen längst ovan till vänster och välja Egenskaper osv.

Stäng Visual Studio. Nästa gång du öppnar Visual Studio, för att testa andra C++ program, gör *inte* om allt du gjorde på sidorna 13-17. Följ istället instruktionerna i nästa avsnitt (eller i övn 1.1, sid 20).

1.4 ETT projekt för alla konsolapplikationer

Vill du skapa nya konsolapplikationer behöver du inte göra om hela proceduren. Det är jobbigt att behöva skapa ett separat projekt varje gång man vill testa ett litet program. Och vi kommer att skriva och testa många små program. Av dessa kommer många – speciellt i början – bestå av endast en fil där skapandet av ett nytt projekt varje gång kan uppfattas som overkill. Du behöver bara ladda projektet MyConsoleProject i Visual Studio, exkludera filen MyFirst.cpp från det och infoga nya filer resp. skriva ny kod, spara och köra enligt instruktioner ovan. *Ett* projekt räcker för alla konsolapplikationer.

Det finns följande möjlighet att slippa detta och ändå uppfylla Visual Studios krav på att utveckla program endast inom ett projekt:

För att underlätta arbetet och inte behöva skapa för *varje* program ett *nytt* projekt, kommer vi att skapa ett enda projekt dvs ta steg **1** (sid 13) bara en gång i början och i fortsättningen endast upprepa stegen **2a**. Dvs vi kommer att skapa *ett* projekt i vilket vi sedan lägger en aktuell C++ källkodsfil, jobbar med den och avlägsnar den från projektet när vi är klara. Nästa gång öppnar vi samma projekt, lägger till en annan källkodsfil i det, jobbar och tar bort filen från projektet osv. För alla C++ program används ett och samma projekt.

Så här kan man realisera detta förfarande:

I fortsättningen, när du vill testa ett annat C++ program, laddar du det redan skapade projektet MyConsoleProject, markerar först den gamla källkodsfil som finns i projektet från tidigare, exkluderar den från projektet genom att högerklicka på filnamnet i Solution Explorer och välja:

Exclude From Project

Filen tas bort och är inte längre med i Visual Studio-projektet, men finns kvar på hårddisken i projektmappen.

Sedan fortsätter du så här: För att ladda och testa nästa program markerar du i Solution Explorer projektnamnet **MyConsoleProject**, högerklicker och väljer:

→ Add → New Item…

Här följer du instruktionerna i stegen **2a**. Så kan du hela tiden använda *samma* projekt för att kompilera *alla* dina C++ program, så länge de är av typ Console Application. På så sätt slipper vi att skapa ett separat projekt för varje C++ program.

Organisera dina C++ filer

Det är upp till dig hur du organiserar dina filer. Men för att underlätta arbetet rekommenderas följande förfarande:

Du kan samla och spara alla dina C++ program tillhörande kapitel 3 *Att komma igång med C++* genom att skapa en undermapp som heter Kap 03 KomIgång i en valfri mapp, t.ex. i C:\C++ och spara filen MyFirst.cpp i mappen C:\C++\Kap 03 KomIgång. Detta kan göras från Visual Studios FILE-huvudmeny med:

→ FILE → Save MyFirst.cpp As…

Anledningen till denna rekommendation är följande: Har du fått körresultatet på förra sidan efter flera försök där du rättat till kompileringsfel och kompilerat om och därmed ändrat C++ kodsfilen, har alla dina ändringar sparats i filen MyFirst.cpp som tillhör projektet MyConsoleProject. Men eftersom vi enligt instruktioner nedan kommer att exkludera filen MyFirst.cpp från projektet för att sedan kunna infoga och köra nästa program i samma projekt är det bra för säkerhets skull att ha alla sina testade program samlade i en egen mapp som ligger utanför projektmappen. På liknande sätt kan du spara dina efterföljande C++ källkodsfiler i mappar du skapar under C:\C++ och betecknar enligt bokens kapitelindelning.

Självklart fungerar bokens alla programexempel även i alla tidigare versioner av Visual Studio än 2022 vars installation beskrevs på sid 10.

Övningar till kapitel 1

1.1 Installera VisualStudio på din dator enligt instruktionerna på sid 10. Konfigurera VisualStudio enligt instruktionerna på sidorna 11-17. Har du fått programmet **`MyFirst`** (sid 16) att fungera, fortsätt med att skapa och testa ditt andra C++ program **`MySecond`** genom att göra så här:

a) Öppna VisualStudio, men skapa *inte* ett nytt projekt. Klicka istället (på vänstersidan) under rubriken Open recent på ditt gamla projekt MyConsoleProject.sln som du skapade när du ville testa vårt första program **`MyFirst`** (sid 13). Ditt första projekt med filen **`MyFirst.cpp`** öppnas.

b) Gå till Solution Explorer-fönstret, högerklicka på filen **`MyFirst.cpp`** och välj:

Exclude From Project

Filen tas bort från Visual Studio-projektet, men finns kvar på hårddisken.

c) Markera projektnamnet MyConsoleProject, högerklicka på det och välj:

→ Add → New Item…

Följ instruktionerna i stegen **2a) Att skapa en ny fil och infoga det i projektet** (sid 15). Men istället för att skriva filnamnet MyFirst.cpp i textrutan Name i dialogrutan Add New Item namnger du den nya filen med MySecond.cpp. Och istället för att mata in koden till programmet **`MyFirst`**, matar du in följande kod i det tomma editfönstret i fliken MySecond.cpp:

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

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

Kompilera och exekvera koden ovan enligt instruktionerna på sid 16-17. Du borde få följande utskrift:

```
Detta är mitt andra C++ program!
Det skriver ut två rader text.
```

Använd alltid detta förfarande när du i fortsättningen vill testa nya program i VisualStudio. Läs mer i avsnitt ***2.4 ETT projekt för alla konsolapplikationer*** (sid 18).

Kapitel 2

Introduktion

till programmering

Ämne	Sida
2.1 Vad är programmering?	22
2.2 Kompilering och exekvering	24
- John von Neumann-modellen	24
- Olika typer av fel	27
2.3 Från C till C++	28
Övningar till kapitel 2	31

2.1 Vad är programmering ?

Var och en har ett intuitivt svar på denna fråga. Ändå är det värt försöket att precisera denna intuitiva uppfattning genom en definition som sätter begreppet även i rätt sammanhang och avgränser ämnet från andra, närbesläktade ämnen. Dessutom kan man enlkare följa bokens röda tråd om man får en beskrivning av vad som väntas. Låt oss börja med att ge negativa svar, dvs att diskutera vad programmering *inte* [*] är för att steg för steg närma oss det positiva svaret. På så sätt avgränser vi ämnet. Vi kommer att se att detta inte är någon ordlek utan att man även kan dra vettiga slutsatser av de negativa svaren som är användbara i praktiken.

Tre negativa svar

1. För det första hävdar jag att programmering *inte är en konstart*. För att syssla med konst som t.ex. måleri, musik, skrivandet osv. behövs en viss begåvning. För programmering däremot behövs ingen speciell talang, vanlig logik räcker. Därför kan i princip alla lära sig programmering. Programmering kan jämföras med bilkörning. Det gäller ju bara att kunna *använda* en teknisk apparat, inte att designa och bygga den. Detta utesluter inte att det i vissa fall kan finnas program som stiliserats till konstverk – undantag som bekräftar regeln.

2. För det andra hävdar jag att programmering *inte är någon vetenskap*. Det kanske förvånar mer än jämförelsen med konst. Jo, det finns ett samband mellan programmering och vetenskap. Det kan jämföras med relationen mellan bilkörning och maskinteknik eller med relationen mellan multiplikationstabellen och matematik. Visst är både maskinteknik och matematik vetenskap. Men ingen skulle därför hävda att även bilkörning och multiplikationstabellen är vetenskap. På liknande sätt är programmering *relaterad* till vetenskapen *datalogi, informatik, datavetenskap, Computer Science*. Programmering är bara en praktisk tillämpning av den:

> ”Programmering är i allra högsta grad ett praktiskt ämne.” (sid 10)

Man kan inte lära sig programmering genom att endast läsa böcker. För att lära sig programmering måste man programmera, jämförbart med bilkörning.

3. För det tredje hävdar jag att programmering *inte är ett självändamål*. Programmering är ett medel, ett verktyg som tjänar ett högre syfte, nämligen att lösa ett *problem* med hjälp av datorn. Lyckas man med att lösa det på datorn, har man datoriserat problemlösningen. Så problemlösning är *målet* och programmering är *medlet* för att uppnå detta mål. Detta innebär att vi i regel inte skriver program för programmeringens skull. Däremot kan detta ske i pedagogiskt eller experimentellt syfte. Som sagt: undantag bekräftar regeln.

---

[*] Filosofen Spinoza anmärkte 1674: *”Determinatio est negatio”* dvs *definition* är *negation*. När man säger vad någonting är, har man samtidigt sagt vad det *inte* är. Därför kan man lika bra definiera någonting genom att säga vad det *inte* är.

Två positiva svar: 1. Programmering som problemlösning

Det var Niklaus Wirth, skaparen av programspråket Pascal, som på 60-talet ställde upp denna definition:

> Program = algoritm + data

En *algoritm* är ett tillvägagångssätt för lösningen av ett problem. Väljer man programkod för att beskriva algoritmen har man ett datorprogram. Sedan måste ibland lite information läggas till. *Data* är information i organiserad form. Wirths definition återspeglar en *algoritmorienterad* syn på programmering som även kallas för *procedural programmering*. En *procedur* är en modul som kodar lösningen (algoritmen) till ett speciellt problem. T.ex. kallas procedurer i C++ för *funktioner*.

> Slutsats: Förstå problemet, hitta och beskriv en lösningsalgoritm *innan* du börjar programmera.

Att förstå problemet, att hitta och beskriva en algoritm är den svårare delen av uppgiften som kräver mer tid och energi än själva programmeringen. Frågan är: Ingår verkligen alla dessa delar i programmerarens uppgift? Svaret är inte så enkelt:

Vad gäller att *förstå problemet*, måste i regel programmeraren vara förtrogen med problemställningen och ha en någorlunda god insikt i problemets viktigaste aspekter utan att därför vara expert i ämnet.

Vad gäller att *hitta en lösning*, beror det på problemets karaktär och komplexitet. Ibland är en lösning känd och behöver bara studeras. I andra fall är problemet så komplext att endast experter i ämnet kan hitta en lösning då det krävs expertkunskaper, uppfinningsrikedom och/eller forskningsinsatser. Men det finns också enklare fall då programmeraren står ensam inför problemet och måste göra allt själv inklusive klara av problemlösningen. Men att *beskriva* en lösningsalgoritm, steget innan kodningen, är nästan alltid programmerarens uppgift.

2. Programmering som modellering

En annan definition som kom på 80-talet och återspeglar den *objektorienterade* synen på programmering är:

> Program = Modell av verkligheten

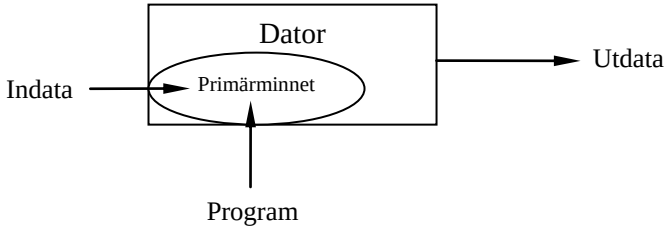
Om man i *Program = algoritm + data* lägger betoningen på data istället för på algoritmen och data inte längre betraktas som ett slags bihang till algoritmen utan som *objekt* eller en modell av verkligheten, kommer man till *objektorienterad programmering* (OOP) som kom upp på 80-talet som en ny programmeringsfilosofi när C vidareutvecklades till C++. Mer om detta tas upp i avsn. ***2.3 Från C till C++*** (sid 28) och ännu senare i ***kap 10 klasser*** som är en djupdykning i OOP (sid 205).

2.2 Kompilering och exekvering

Innan vi ger oss i kast med själva kodningen ska vi på ett övergripande sätt gå in på hur programkoden hamnar i datorn och hur den körs där. För att besvara frågan går vi tillbaka till de första datorerna. Då var den enda möjligheten att skapa ett ”program”, att få in instruktionerna (mjukvaran) i maskinen (hårdvaran). Det handlade om att tekniskt realisera samspelet mellan mjukvaran och hårdvaran. På vilket sätt detta skulle ske var ett svårlöst problem.

John von Neumann-modellen

Den som löste detta problem först var *John von Neumann*. Han lyckades **1944** att konstruera en dator som kunde lagra både *data* och *instruktioner* i datorens primärminne. Allt matades in genom hålkort. Denna tekniska innovation var ett genombrott som inledde programmeringens historia i modern bemärkelse. Än idag fungerar i princip exekvering av kod i datorn enligt denna modell: Startar man körningen av ett program laddas programkoden från en fil på hårddisken, till datorns RAM *(Random Access Memory)*. Om motsvarande instruktion finns i koden, matas in även indata i regel från tangentbordet. John von Neumann-modellen ser ut så här:



Indata består av *tal*. Därför är det enkelt att skriva om det till det binära talsystemet med basen 2 så det det består av ettor och nollor.

Program däremot består av *instruktioner*. En instruktion är ett *kommando* som datorn ska utföra. Detta kan endast ske *efter* att instruktionen översatts till ettor och nollor, så att datorns processor kan utföra den. På John von Neumanns tid bestod programmets instruktioner av långa talkedjor. Man programmerade i en kod som datorn kunde tolka och vidarebefordra i form av ettor och nollor till processorn. Denna kod kallas för *maskinkod*.

Idag programmerar man i något programmeringsspråk, kallat källkod. *Källkod* är kod som människan skriver, medan *maskinkod* är kod som datorn genererar. I dagens datorer *kompileras* (översätts) eller *interpreteras* (tolkas) källkod till maskinkod för att kunna utföras. Processen att översätta källkod till maskinkod kallas för *kompilering* och programvaran som åstadkommer den heter *kompilatorn*. T.ex. är C/C++ kompilerande språk medan Python är ett interpreterande språk.

Editering

För att skriva källkod på datorn behövs en texteditor. Källkod får endast innehålla tecken som ”förstås” av kompilatorn. Om det ska bli ett C++ program får källkodsfilen endast innehålla de reserverade ord som definierar språket samt ev. några biblioteksnamn (sid 29). Därför måste skrivverktyget spara källkodsfilen som *oformaterad textfil*. Arbetar man t.ex. i Windows kan Notepad (Anteckningar) eller Notepad++ vara lämpliga texteditorer, eftersom de sparar alla filer som rena textfiler av typ **`*.txt`** utan några formateringar. Andra bra alternativ finns att gratis ladda ned från Internet som t.ex. TextPad. Ordbehandlare däremot av typ Word och andra formaterar texten och sparar sina filer som dokument av typ **`*.docx`** eller annat. Formatering innebär att det läggs till osynliga tecken i texten som C++ kompilatorn inte känner till. Dätför är sådana ordbehandlingsprogram inte lämpliga för programmering. I andra operativsystem finns andra texteditorer som kan användas, om de inte formaterar. Har du en favoriteditor använd den, annars ta den enklast möjliga. Har du däremot tillgång till en integrerad programutvecklingsmiljö *(IDE = Integrated Development Environment)* som *Visual Studio*, *Bloodshed Dev C++*, *Borland C++ Builder*, *Symantec C++*, *GNU C++ …* är det rekommendabelt att använda den, även om du har en favoriteditor. En stor praktisk fördel med en IDE är att man slipper byta miljö mellan editering och kompilering och att IDEn har olika stödverktyg för språket. Vi kommer att använda IDEn *Visual Studio* vars installation och konfiguration togs upp i detalj tidigare (sid 10 & 11).

Regler för filändelsen

Har du skrivit din programkod i någon editor och sparat filen som **`*.txt`**, kommer du få kompileringsfel även om din kod är helt felfri. Boven i dramat är filändelsen: C++ kompilatorn accepterar inte **`txt`** som filändelse. Kompilatorn måste nämligen kunna identifiera de filer som innehåller C++ kod via filändelsen. Olika plattformar tillämpar olika regler för filändelsen till C++ källkodsfiler. Windows använder ändelsen **`cpp`**. Därför måste du, om du jobbar under Windows, antingen spara din källkodsfil med korrekt filändelse eller ändra ändelsen till **`cpp`** i efterhand. I andra operativsystem används andra ändelser [*]. I *Visual Studio* sparas alla C++ källkodsfiler automatiskt med filändelsen **`cpp`**.

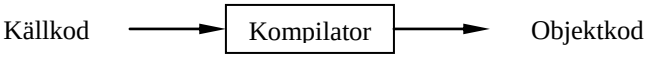
Kompilering av källkod

Kompilering innebär översättning av källkod till maskinkod. Har du skrivit din programkod i en texteditor och sparat den som ren textfil med ändelsen **`cpp`**, måste

[*] I Ubuntu/Linux/Unix t.ex. kan filändelsen vara *C*, *cc*, *cxx* eller *cpp*. Eftersom det finns många Unix-dialekter kan även andra varianter förekomma. I regel är i alla Unix-system en C++ kompilator inbyggd och kan anropas med kommandot *g++ <filnamn>*. I Ubuntu däremot måste kompilatorn först laddas ned och installeras med gratis programvaran *buildessential*. Kompileringen genererar en exekverbar fil med det förvalda namnet *a.out*, en smart metod som gör att hårddisken inte fylls med onödiga maskinkodsfiler: Varje gång man kompilerar, skriver den nya filen *a.out* över den gamla med samma namn. Den kompilerade koden exekveras sedan med kommandot *./a.out*, där den inledande punkten **.** betyder aktuell mapp och **/** är motsvarigheten till \ i Windows. Dvs den fullständiga sökvägen till filen *a.out* måste anges.

du kompilera din källkod innan du kan köra programmet eftersom datorn inte förstår källkod utan endast maskinkod. Därför måste en *kompilator* vara installerad på din dator. Senare beskrivs hur man gör det i Visual Studio (sid 16).

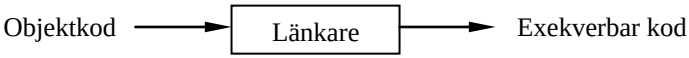
Men det finns olika typer av maskinkod, t.ex. körbar och icke-körbar. Kompilering av C++ källkod (lagrad i en fil av typ **`*.cpp`**) producerar först en icke körbar maskinkod som kallas *objektkod* och lagras i en *objektfil* av typ **`*.obj`**.



Anledningen till att objektkoden inte är körbar, är att programmet inte är fullständigt. Endast den kod som finns i källkodsfilen dvs ditt eget program har kompilerats. Men så gott som alla C++ program inkluderar även biblioteksprogram som t.ex. **`iostream`** eller **`std`** (sid 34). Dessa ingår inte i objektfilen. Där finns endast en sökväg till dem. Biblioteksprogrammen måste först hämtas och länkas ihop med din egen kod för att den ska bli körbar. Kompilatorn kan inte åstadkomma detta. Den är bara en översättare. Hur man kompilerar källkod i Visual Studio beskrivs på sid 16.

Länkning

Det är *länkarens* uppgift att hämta biblioteksprogrammen och länka ihop dem med din egen kod. På så sätt kopplar objektkoden din kod till bibliotekskoden och förbereder programmet för exekvering:

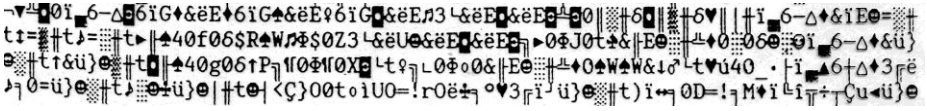


Det behövs ingen särskild åtgärd för att starta länkaren. Den startar automatiskt direkt vid exekveringen utan att man märker det. Endast om det uppstår fel är det relevant för felsökningen att veta att länkning förekommer som ett steg mellan kompilering och exekvering.

Exekvering av maskinkod

Efter länkning har man fått in programmet som komplett körbar maskinkod som lagras i en *exekverbar fil* med ändelsen **`*.exe`**, där filändelsen står för exekverbar dvs körbar. I praktiken innebär det att **`exe`**-filen nu laddas från hårddisken till RAM-minnet och skickas därifrån till datorns processor som nu kan utföra alla instruktioner i **`exe`**-filen. Hur man gör det i Visual Studio kommer att beskrivas senare (sid 17).

Innan vi går vidare är det kanske intressant att titta på hur maskinkod ser ut. Det här är ett exempel på en liten bit av maskinkod:



Maskinkod kan ha olika utseende beroende på vilken kompilator man använt och på vilket sätt man visar den. Den är producerad av kompilatorn och länkaren och består av instruktioner till datorns processor. Som man ser kan maskinkod varken läsas eller förstås av människan. Men processorn ”förstår” koden, omvandlar den till nollor och ettor och utför slutligen instruktionerna. Maskinkod tar alltid mycket mer minne än motsvarande källkod. Redan objektkoden är större än källkoden.

Olika typer av fel

kan uppstå i alla ovan beskrivna steg. Vid felsökning är det avgörande att man först identifierar *typen* av fel dvs skiljer mellan olika sorters fel innan man vidtar någon åtgärd. Det finns i huvudsak följande typer av fel:

- **Kompileringsfel** som kan uppstå pga att vi har brutit mot språkets regler. Dvs i kod som vi själva skrivit finns ”ortografiska” fel, något felstavat nyckelord eller ett utelämnat semikolon osv. Det kan även handla om ”grammatiska” fel, även kallade *syntaxfel* som t.ex. en felanvänd kod eller fel struktur i koden. Kompileringsfel innebär tvärstopp dvs man kan inte gå vidare till nästa steg utan måste först hitta och korrigera felet samt kompilera om.

- **Länkningsfel** kan förekomma om t.ex. filerna till biblioteksprogram som ska länkas inte finns på plats, en ganska sällsynt företeelse. En fullständig av- och nyinstallation av programvaran kan eventuellt åtgärda felet.

- **Exekveringsfel** uppstår endast om processorn inte kan utföra dina instruktioner. Ett typiskt exempel på exekveringsfel i program som involverar beräkningar är division med 0. Ett annat exempel är användningen av minnesutrymme som är redan upptaget av ett annat program i datorn. Ett tredje exempel är skadade eller obefintliga filer som det hänvisas till i den egna programkoden.

- **Logiska fel** kan förekomma i kod som syntaxmässigt är korrekt och kan både kompileras och exekveras. Felet ligger i att den gör något annat än programmeraren hade för avsikt att den skulle göra. Allt verkar fungera korrekt: varken kompilerings- eller exekveringsfel. Ändå blir resultatet fel. Man har tänkt logiskt fel när man kodade. Det föreligger ett ”missförstånd” mellan programmerarens sätt att tänka och hur koden tolkas av datorns processor.

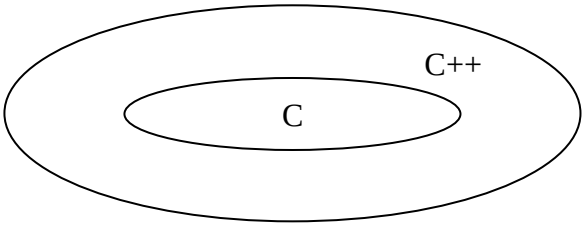
Vanligast är kompileringsfel. Även som nybörjare kan man ofta få *inte ett* – utan en hel samling av felmeddelanden. Bli inte desperat! Det är helt normalt. Glöm alla felmeddelanden utom det allra *första*. De kan nämligen vara *följdfel* orsakade av första felet. Åtgärda endast det första och kompilera om. Om några fel är kvar, upprepa förfarandet. Du kommer att se: efter två eller tre gånger har du blivit av med alla fel.

2.3 Från C till C++

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/C++ och Unix. Idag är inte bara Unix utan även andra operativsystem inkl. Windows skrivna i C/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. 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 genom *pekararitmetik*.

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 tillgodosåg 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*. Läs mer om OOP på sid 205.

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:



C är nämligen en delmängd av C++. Därför gäller all C-kod även i C++, men inte tvärtom. Med andra ord, en C++ kompilator kan kompilera all kod skriven i C, men inte tvärtom (sid 24). Så den som lär sig C++ lär sig automatiskt C.

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.

Reserverade ord

Som alla programmeringsspråk är även C++ definierat av ett antal nyckelord, även kallade *reserverade ord*, på eng. *keywords*. De är reserverade av och för själva språket, dvs bildar språkets ordförråd. De får inte användas som namn (identifierare) för variabler eller programmets andra delar, t.ex. funktioner osv. Några av dem är samlade i följande tabell:

Reserverade ord i C++				
`alignas`	`alignof`	`and`	`and_eq`	`asm`
`auto`	`bitand`	`bitor`	`bool`	`break`
`case`	`catch`	`char`	`char16_t`	`char32_t`
`class`	`compl`	`const`	`constexpr`	`const_cast`
`continue`	`decltype`	`default`	`delete`	`do`
`double`	`dynamic_cast`	`else`	`enum`	`explicit`
`export`	`extern`	`false`	`float`	`for`
`friend`	`goto`	`if`	`inline`	`int`
`long`	`mutable`	`namespace`	`new`	`noexcept`
`not`	`not_eq`	`nullptr`	`operator`	`or`
`or_eq`	`private`	`protected`	`public`	`register`
`reinterpret_cast`	`return`	`short`	`signed`	`sizeof`
`static`	`static_assert`	`static_cast`	`struct`	`switch`
`template`	`this`	`thread_local`	`throw`	`true`
`try`	`typedef`	`typeid`	`typename`	`union`
`unsigned`	`using`	`virtual`	`void`	`volatile`
`wchar_t`	`while`	`xor`	`xor_eq`	`...`

Det finns fler än i tabellen ovan. Observera att de alla skrivs med små bokstäver. Följande allmän regel gäller inte bara för reserverade ord utan för all C++ kod:

C++ är case sensitive (skiftlägeskänslig).

Dvs C++ skiljer på små och stora bokstäver. T.ex. är **`Tal`** och **`tal`** två *olika* variabelnamn. Alla reserverade ord är giltiga endast med små bokstäver. Om vi t.ex.

skriver det reserverade ordet **`new`** som **`New`** kommer C++ kompilatorn generera ett felmeddelande. Fördelen med denna regel är att den utökar möjliheterna för val av nya variabelnamn. Nackdelen är att man lätt kan förväxla små och stora bokstäver.

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.

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. Vi måste därför tala om för kompilatorn vilka fördefinierade program vi tänker att använda och var de finns.

Även C++ biblioteket är numera standardiserat. En av nyheterna som kommittén för ANSI/ISO-standarden har utarbetat, är att man 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.

Om namnet C++

Syftet med att skapa programmeringsspråket C var ursprungligen att skapa ett språk för programmering av operativsystemet Unix. En tidigare version av C i Unix-projektet var bl.a. ett språk som hette B. Konstigt nog fanns däremot ingen föregångare A. Tillägget **++** betyder att öka med ett, vilket syftar på att man lagt till ett utvecklingssteg till C. C++ är C plus ett nytt objektorienterat tillägg.

Om vi sammanfattar allt vi har sagt om språket, kan vi konstatera att C++ är:

- ett kompilerande högnivåspråk
- ett *strukturerat* språk med rötter i C
- ett *universellt* språk för alla möjliga tillämpningar
- ett *objektorienterat* programmeringsspråk efter utvidgningen av C
- lämpligt för att lära nybörjare programmering
- populärt hos professionella programmerare för utveckling av avancerade tillämpningar som operativsystem (Windows, Unix,…), spel osv.
- ett modernt standardiserat programspråk med stor spridning i världen.

Övningar till kapitel 2

Läs texten: http://mathonline.se/Varfor_Programmering.pdf

Besvara frågor om texten:

2.1 Vad tycker du om *Steve Jobs'* påstående att programmering lär oss att *tänka?*

2.2 Hur tolkar du termen *artificiell intelligens?* Tror du att maskiner kan lära sig tänka?

2.3 Försök att med egna ord beskriva *algoritmiskt tänkande*.

2.4 Hur skulle du definiera begreppet *algoritm?*

2.5 Använder du i vardagen algoritmer? Om ja, nämn några exempel.

2.6 Vad har algoritmiskt tänkande med programmering att göra?

2.7 Beskriv med egna ord vad som menas med *modularisering*.

*Besvara följande frågor om kap 2 * :*

2.8 Varför kan man inte lära sig programmering genom att endast läsa böcker?

2.9 Om programmering är *medlet* för att uppnå ett mål vad är då *målet*?

2.10 Hur definieras programmering ur den *algoritmorienterade* synen?

2.11 Vilken programmeringsfilosofi ligger till grund för den algoritmorienterade synen?

2.12 Vilken slutsats kan man dra om man ser programmering som problemlösning?

2.13 Vad borde steget före programmering alltid vara?

2.14 Hur definieras programmering ur den *objektorienterade* synen?

2.15 Varför ändrades synen på programmering under 80-talet?

---

* Svar till alla dessa frågor finns i bokens *kap 2 Introduktion till programmering*, sid 20-29.

2.16 Vad är skillnaden mellan *procedural* och *objektorienterad* programmering?

2.17 Beskriv med egna ord *John von Neumann-modellen*.

2.18 Vad innebär *kompilering* och hur skiljer den sig från *exekvering?*

2.19 Skriver man *källkod* eller *maskinkod* när man programmerar?

2.20 Vilken egenskap borde editorn ha i vilken man skriver programkoden?

2.21 Vad gör *länkningen* som ett steg mellan kompilering och exekvering?

2.22 Redovisa med egna ord de olika typer av fel som kan förekomma vid programmering.

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

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

2.25 Vad innebär det att C++ är ett *universellt* programmeringsspråk?

2.26 Är C++ ett interpreterande eller ett kompilerande språk?

2.27 Är C++ källkod eller maskinkod?

2.28 Vad innebär *reserverade ord* och vilka konskvenser har de för kodningen?

2.29 Vad är fördelen med *case sensitivity* i C++?

2.30 Vad betyder tillägget **++** vid vidareutvecklingen från C till C++?

2.31 Är C++ ett standardiserat programmeringsspråk?

2.32 Varför har man ett stort bibliotek i C++ ?

2.33 Ingår biblioteket i C++ standarden?

2.34 Får man använda reserverad ord som namn till sina egna identifierare?

2.35 Får man använda bibliteksnamn till sina egna identifierare?

Kapitel 3

Att komma igång

med C++

	Ämne	Sida	Program
3.1	Vårt första C++ program	33	**MyFirst**
	- Program i C/C++	34	
	- Funktionen **main()**	35	
	- Programmet **MyFirstSwed**	37	**MyFirstSwed**
3.2	God programmeringsstil	39	**MyFirst_bad**
3.3	Utmatning med cout	41	**Cout** (Si-out)
3.4	Konkatenering	44	**Concat**
	- Att "rita" i textmiljö	45	**Figure**
Övningar till kapitel 3		46	

3.1 Vårt första C++ program

Grattis att du har kommit så långt utan att tappa tålamodet! Nu ska vi försöka att *förstå* koden till det program vi testkörde tidigare. Så, låt oss sätta igång!

```
// MyFirst.cpp                              Filnamnet
// Utskrift av text till konsolen
// cout (Console output) med utmatningsoperatorn <<
// Kommentarer // , radbytet \n och tabulatorn \t

#include <iostream>
using namespace std;

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

En körning av **MyFirst** ger följande utskrift på det lilla (svarta) fönstret, kallat *console* (Windows' kommandotolk):

```
        Detta är mitt första C++ program!
```

I själva verket skrivs även ut bl.a. ett meddelande av typ **Press any key to close this window . . .** eller liknande. Allt detta avbildar vi inte här.

Ett annat dilemma är de svenska tecknen **ä** och **ö**. Orsaken till deras förvrängningar är kodningar som inte följer någon standard. Detta har inte med C++ att göra utan är beroende av miljön. Inte alla miljöer följer en standard för de icke-engelska bokstäverna i världens olika språk. Hur man kan komma runt dilemmat visas i programmet **MyFirstSwed** (sid 37) och löses generellt på sid 87. Nu ska vi koncentrera oss på C++ koden. Varför kallar vi **MyFirst** för *program?* Är inte all kod i C++ ett *program?* Faktiskt inte!

Program i C/C++

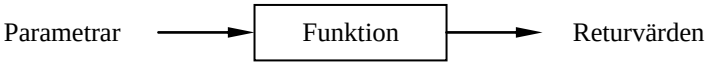
> Ett C/C++ program är en samling funktioner (C) och klasser (C++).
> En och endast en av dessa funktioner måste vara funktionen **main()** som är startpunkten för programexekvering.

Närmare bestämt består alla C program av en samling *funktioner*, medan i C++ kan även *klasser* vara betåndsdelar av ett program. Funktionen **main()** måste definieras i alla C++ program, för att den är programmets exekveringspunkt. Utan **main()** sker ingen exekvering. Namnet, ja t.o.m. hela huvudet **int main()** är fördefinierat och måste stavas korrekt, speciellt med hänsyn till case sensitivity.

Det måste skrivas med små bokstäver. Vad begreppen *funktion* och *klass* exakt betyder kommer att behandlas senare i separata kapitel (sid 126 & 205). Men eftersom funktionen **`main()`** spelar en avgörande roll ska vi titta nämare på den:

Funktionen `main()`

Vårt första C++ program **`MyFirst`** är det enklast tänkbara enligt definitionen ovan, för det består av en enda funktion, nämligen **`main()`**. Just nu räcker det att känna till att en *funktion* fungerar som en låda där man stoppar in indata och får ut utdata. Indata kallas även *parametrar* och utdata *returvärde*:



Koden till en funktion har en rubrik, även kallad *huvud*, och ett innehåll, kallat *kropp*. Den allmänna strukturen hos **`main()`** kan beskrivas så här:

```
int main()
{
    sats(er);
}
```

Funktionens huvud är **`int main()`** bestående av bl.a. namnet **`main`** och parentesen **`()`** som kallas för *parameterlistan*. Just i det här fallet är den tom, dvs **`main()`** har inga parametrar. Notera att det är parentesen **`()`** som gör **`main()`** till en *funktion*. Parameterlistan **`()`** är kännetecknet för en funktion, även om den är tom. T.o.m. när man nämner en funktion i förklarande text är det en allmän konvention att skriva med parentesen **`()`**. Även om namnet strängt taget är **`main`**, skriver vi alltid **`main()`**. Så gör man med *alla* funktioner oavsett antal parametrar.

Huvudet inleds med ordet **`int`**, som talar om vilken typ av data som ska returneras av funktionen. **`int`** är funktionens *returtyp* dvs returvärdets datatyp. Dessa begrepp kommer att förklaras i detalj senare.

Funktionens huvud får inte avslutas med semikolon eftersom huvudet inte är någon sats (se nästa sida) utan *inleder* funktionens definition. Och definitionen är inte avslutad än. Det viktigaste kommer *efter* huvudet, nämligen kroppen, funktionens innehåll, vad den ska göra. Huvudet är endast namnet. Kroppen består av ett antal satser inom **`{ }`** – ett klammerpar – som vi brukar kalla för *måsvingar*. Den inledande måsvingen **`{`** betyder början och den avslutande **`}`** slutet på kroppen. Måsvingarnas uppgift är att gruppera satserna under funktionshuvudet och avgränsa dem från andra delar av programmet. Vi kommer att använda både *klamrar* och *måsvingar* som beteckning för tecknen **`{ }`**.

Intressant är också frågan, när och på vilket sätt funktionen **`main()`** *anropas* medan vi bara definierar den. Svaret är att den automatiskt anropas av ett inbyggt verktyg som heter *C++ Virtual Machine*, när vi exekverar programmet.

Satser

I programmering betecknas termen *instruktion* med *sats*, på eng. *statement*. En sats är en instruktion till datorn att utföra något. En sats avslutas med *semikolon*. Semikolonet `;` är är en *del* av en sats, t.o.m. en *obligatorisk* sådan, det allra *sista tecknet* i satsen. Kod utan semikolon är ingen sats. Semikolonet är C++ språkets *satsavslutningstecken* vars utelämnande leder till kompileringsfel. I vårt första program **`MyFirst`** finns faktiskt endast *två* satser: den första inleds med **`using`**, den andra med **`cout`**. De koder som inte avslutas med semikolon är inga satser.

T.ex. avslutas koden **`#include <iostream>`** inte med semikolon, eftersom den inte är någon sats utan ett s.k. *kompileringsdirektiv* – även kallat *preprocessordirektiv* – dvs ett direktiv till länkaren (sid 26) om att hämta alla filer från biblioteket **`iostream`** och länka ihop dem med vår egen kod. Alla sådana direktiv i C++ inleds med **`#`** och får inte avslutas med semikolon. Däremot måste de avslutas med radbyte, dvs behöver man flera direktiv i ett program måste var och en stå på en separat rad. Om **`int main()`** som inte heller avslutas med semikolon läs nedan:

Kommentarer i C++

De första tre raderna börjar med två snedstreck (eng. *slash*) **`//`**. Detta teckenpar betyder *kommentar*, närmare bestämt *radkommentar*. En radkommentars giltighet börjar med **`//`** och sträcker sig till slutet av raden. **`//`** kan stå i början av en rad, men också någonstans mitt på raden. En *blockkommentar* kan gå över flera rader och ska inledas med **`/*`** och avslutas med **`*/`**. Alla kommentarer kommer att ignoreras av kompilatorn, dvs kompilatorn hoppar över dem när den översätter till maskinkod. De är endast till för att förklara vad programmet gör. I den första kommentarraden brukar jag skriva i vilken fil koden lagras, här **`MyFirst.cpp`**. Därmed får programmet samtidigt ett namn, här **`MyFirst`**. I den andra raden beskrivs vad programmet gör. Sedan följer kommentarer som anger de olika programmeringstekniska koncept som behandlas i programmet.

Inkludering av bibliotek

Efter kommentarerna följer:

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

Dessa två rader kommer att stå i början av alla våra program. De talar om för kompilatorn var den kan hitta koden till **`cout`** som vi använder i programmet för att skriva ut tecken på skärmen. Om vi tittar på tabellen för reserverade ord (sid 29), ser vi att **`cout`** inte finns där. Faktum är att **`cout`** inte är ett reserverat ord utan kod som är förprogrammerad i biblioteksfilen **`iostream`**. För att kunna använda **`cout`** måste vi inkludera denna fil i vårt program vilket görs med den första av de två raderna ovan: Raden inleds med **`#include`** dvs koden i **`iostream`** ska inkluderas, läggas till vår egen kod. Först då blir **`cout`** definierat i vårt program. **`Iostream`** innehåller förprogrammerade rutiner för in- och utmatning där **`io`** (uttal: ”aj-å”) står för **`i`**nput/**`o`**utput.

Men man vill även kunna inkludera andra bibliotek, både egna och sådana som ev. kommer från andra programtillverkare. Samtidigt ska namnkonflikter mellan C++ biblioteket och tredjehandsprogram undvikas. Därför har man enligt den nya C++ standarden definierat hela C++ biblioteket i *namnutrymmet* **std** som ett skydd mot eventuella namnkonflikter. Även biblioteket **iostream** är placerat i detta **namespace** – en slags behållare för namn. För att komma åt **cout** i **iostream** måste vi tala om för kompilatorn att vi vill använda det namnet **cout** som är definierat i namnutrymmet **std**. Detta görs i satsen **using namespace std;** Alternativet vore att, varje gång vi använder **cout**, skriva **std::cout** istället, vilket skulle göra koden onödigt tung. Testa gärna att ta bort **using**-satsen och ersätta överallt **cout** med **std::cout**. Programmet borde fungera ändå. Mer om **cout** läs i avsnitt ***3.3 Utmatning med cout*** på sid 41.

Koderna **\n** *och* **\t**

I datasammanhang är begreppet *sträng* (eng. *string*) synonym till text, närmare bestämt en *följd av tecken* som kan, men behöver inte vara bokstäver. T.ex. betyder **\n** inte bokstaven **n** utan det oskrivbara ”tecknet” som man åstadkommer när man trycker på **Enter**-tangenten. **\n** betyder radbyte där **n** står för **n**ewline. Tecknet **** (eng. *backslash*) är en kod som ändrar innebörden av **n** från *bokstaven* **n** till **n**ewline. Alla tecken som kodas med **** åtföljt av *ett* tecken, kallas för *escapesekvenser* (från eng. *to escape* = att fly), t.ex. **\n**. Man ”flyr” från bokstaven **n**:s egentliga betydelse och tillämpar en annan betydelse, i det här fallet radbyte.

En sträng sätts i C++ alltid inom **" "**. I programmet **MyFirst** (sid 34) ingår **\n** i koden som en del av strängen i **cout**-satsen. Där bakas **\n** in och behandlas som vilket tecken som helst inom **" "**. Samma sak kan göras med alla escapesekvenser som i sin helhet kommer att behandlas senare (sid 86).

\n är vårt första exempel på en escapesekvens. Ett annat exempel som förekommer i programmet **MyFirst**, är **\t** där **t** står för ***t****abulator* där **** gör att **t** inte tolkas som bokstaven **t** utan som tabulator, dvs en horisontell indragning med ett antal mellanslag. Båda escapesekvenserna **\n** och **\t** kan bakas in i strängar, men skulle kunna även kodas som enskilda tecken och skickas separat till utskrift. Samma sak är det med mellanslaget. Skriver man ett mellanslag inom en sträng skickas den med till utskrift, annars inte.

Programmet **MyFirstSwed**

För att åtgärda förvrängningen av de svenska tecknen **ä** , **å** och **ö** tittar vi på följande tabell som avslöjar deras koder i två olika format (sid 87):

Tecknet	Dec. kod	Hex. kod	Escapesekvens
ä	**132**	**84**	**\x84**
å	**134**	**86**	**\x86**
ö	**148**	**94**	**\x94**

De två formaten *decimal kod* och *hexadecimal kod* i tabellen ovan ger upphov till två metoder för att få ut en korrekt utskrift av de svenska tecknen:

- Explicit typkonvertering
- Escapesekvenser

I följnde program använder vi båda metoder (lite överkurs just nu):

```
// MyFirstSwed.cpp
// Skriver ut de svenska specialtecknen korrekt med hjälp av deras
// koder. Läs om explicit typkonvertering på sid 81
// och om        escapesekvenser          på sid 86

#include <iostream>
using namespace std;

int main()
{
   cout << "\n\tExplicit typkonvertering:";
   cout << "\n\t(char) 132 ger " << (char) 132;
   cout << "\n\t(char) 134 ger " << (char) 134;
   cout << "\n\t(char) 148 ger " << (char) 148;
   cout << "\n\tDetta " << (char) 132 << "r mitt f" <<
                           (char)148 << "rsta svenska C++ program!";

   cout << "\n\n\tEscapesekvenser:";
   cout << "\n\t\\\\ ger " << "\\";
   cout << "\n\t\\x84 ger " << '\x84';
   cout << "\n\t\\x86 ger " << '\x86';
   cout << "\n\t\\x94 ger " << '\x94';
   cout << "\n\tDetta \x84r mitt f\x94rsta svenska C++ program!\n";
}
```

En körning av **MyFirstSwed** ger följande utskrift:

```
        Explicit typkonvertering:
        (char) 132 ger ä
        (char) 134 ger å
        (char) 148 ger ö
        Detta är mitt första svenska C++ program!

        Escapesekvenser:
        \\ ger \
        \x84 ger ä
        \x86 ger å
        \x94 ger ö
        Detta är mitt första svenska C++ program!
```

För att förstå alla detaljer i detta program var vi tvungna att gå händelserna i förväg och använda begrepp, koncept och metoder som vi i detalj kommer att behandla senare. Så, det är lite överkurs just nu. Därför behövs lite tålamod. Det skulle dock hjälpa om man läste t.ex. om kodning i avsnitt ***5.3 ASCII-tabellen*** på sid 83 och även lite längre fram.

3.2 God programmeringsstil

Innan vi ger oss i kast med att skriva ytterligare program ska vi här öppna en parentes: Hur gick det när du kompilerade dina första C++ program? Du kanske märkte att felsökning kunde vara jobbig. Det är den också, speciellt när programvolymen växer. Vi ska nu lära oss en teknik som gör felsökning enklare. Frågan är mer av generell karaktär: Hur skriver man bra strukturerade program och hur vänjer man sig vid att göra det *från början*? Att göra det från det allra första programmet är nämligen avgörande för att bibehålla vanan att hålla sig till en viss struktur och därmed för att *utveckla* en god programmeringsstil. Därför denna parentes.

Titta på följande kod. Känner du igen den?

```
#include <iostream>
using namespace std; int main() {cout <<
"\n\tDetta är mitt första C++ program!\n"; }
```

Det är vårt första program **MyFirst** bortsett från kommentarerna (sid 34). Trots skillnaderna i sättet att skriva (layouten) ”fungerar” programmet, dvs koden kan både kompileras och exekveras och producerar samma utskrift som programmet **MyFirst**. Kompilatorn struntar nämligen fullständigt i layouten, den kontrollerar endast kodens syntax. Men det gör inte en människa som ska läsa din kod. Skulle du lämna in den till mig skulle du inte bli godkänd. Varför? Förutom de krav som kompilatorn ställer för att överhuvudtaget kunna få programmet i exekverbar form, finns andra krav på vårt sätt att skriva kod. Det handlar om krav på god programmeringsstil. Koden ovan följer inte kraven på god programmeringsstil. Man skulle kunna kalla den för **MyFirst_bad**.

Så här kan man både läsa och förstå samma kod som ovan mycket bättre:

```
// MyFirst.cpp
// Utskrift av text till konsolen
// cout (Console output) med utmatningsoperatorn <<
// Kommentarer // , radbytet \n och tabulatorn \t

#include <iostream>
using namespace std;

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

God programmeringsstil innebär att man skriver kod så att *andra* kan använda och underhålla den. Alla professionella program som du använder på din dator, operativsystemet, editorer, skriv-, rit-, kalkyl-, spel- och andra applikationer har skrivits med detta i åtanke. Program måste vara användarvänliga. God programmeringsstil

innebär att vi lämnar ifrån oss kod som andra kan modifiera och vidareutveckla. Program måste vara lätt ändringsbara. Vi kommer själva att ha glädje av det, när vi vill vidareutveckla våra program. Följande krav ställs på god programmeringsstil:

- Läslighet
- Förståelighet
- Ändringsbarhet

Och det är därför vi har skrivit vårt första, och kommer att skriva alla våra programexempel, med följande stilelement:

1. **Indragningar** är ett stilelement som används för att uppfylla de ovannämnda kraven på god programmeringsstil. I programmet **`MyFirst`** ser man att vissa rader är indragna, närmare bestämt de rader som utgör **`main()`**-funktionens kropp. Indragningarna ska markera att raderna tillhör **`main()`**. Ett exempel på dålig programmeringsstil ser vi på förra sidan. Och det är därför vi kommer att skriva alla våra program i fortsättningen på det här viset: Den allmänna regeln är att indrag ska återspegla programmets logiska struktur. Man använder indragningar även när man skriver pseudokod. Då gör man indrag för att markera att vissa instruktioner är underordnade andra. Indragningar borde vara tydliga dvs inte alltför små. Tumregeln är: mellan 3 och 5 mellanslag.

2. **Separata rader** tillämpas för att öka kodens läslighet. Varje sats ska som regel stå på en separat rad. Men även klamrarna **`{`** och **`}`** står på egna rader. Detta markerar klamrarnas utomordentligt stora betydelse för att gruppera vissa satser och avgränsa dem från andra delar av programmet. Klamrarna utgör alltså gränser som ska vara mycket tydliga. Dessutom står funktionshuvudet **`int main()`** alltid på separat rad.

3. **Kommentarer** ska förklara koden. Hur mycket och på vilket sätt ska man skriva dem? Rekommendationen är att kommentarerna ska vara korta och inte blandas med koden. Detta gäller speciellt radkommentarerna som annars skulle göra koden mindre lättläst. Vill man skriva längre kommentarer ska man helst skriva en dokumentation till programmet. Denna kan antingen ligga helt separat från koden, t.ex. i en textfil, eller skrivas som *blockkommentar* i början eller på andra ställen av programmet. En blockkommentar i C++ kan bestå av flera rader och ska inledas med **`/*`** och avslutas med **`*/`**.

4. **Ledtext** skrivs ut för att instruera användaren vid inläsning, se sid 62.

Programfel ur stilsynpunkt bör inte bedömas som mindre allvarliga än kompileringsfel. Attityden ”att först skriva rätt så att det fungerar, stilen kan man förbättra sedan” eller ”först ska jag lära mig koda, god programmeringsstil kan jag lära mig senare” är ett allvarligt misstag som nybörjare gör pga oerfarenhet, vilket kan leda till slöseri med tid och energi vid felsökning och till dåligt strukturerade program i längre perspektiv. Man tröttnar på programmering – speciellt vid felsökning – om man inte *från början* lägger stor vikt vid god programmeringsstil.

3.3 Utmatning med cout

I vårt första program **MyFirst** finns en sats som har strukturen:

```
cout  <<  " . . . " ;
```

där . . . är en *sträng*, dvs en följd av tecken (sid 37). Satsen skriver ut strängen till konsolen. En sträng kan bestå av 0, 1, 2, ... tecken. När antalet tecken är 0 talar man om den *tomma strängen*. Den kan kodas med `""`, medan `" "` är en sträng som inte är tom utan består av ett mellanslag. För strängar gäller regeln:

> Strängar omgärdas i C++ kod av citationstecken `"  "`.

I satsen ovan skickas en sträng till konsolen med koden **cout <<** (OBS! två delar).

Ordet **cout** (Uttalet: ”si-out”) står för **c**onsole **out**put, där *console* ursprungligen är en beteckning för *terminal* dvs bildskärm och tangentbord. Men i detta sammanhang menas med *console* fönstret som dyker upp, när man exekverar ett C++ program. **cout** står alltså för utskrift till konsolen. Men för att *utföra* denna utskrift behövs en *operator*, nämligen:

Utmatningsoperatorn <<

De två mindre än-tecknen utan mellanslag **<<** bildar *utmatningsoperatorn*. De kan endast användas tillsammans med **cout**. Man skulle kunna tolka **<<** som en pil från höger till vänster:

Utskrift i konsolen ⟵ **Sträng**

Data strömmar i pilens riktning till datorns *standard output* som är konsolen. Följande programexempel visar detta:

```
// Cout.cpp
// Två cout-satser i koden, men endast EN rad text i utskrift
#include <iostream>
using namespace std;

int main()
{
   cout << "\nDetta är EN rad text produc";
   cout << "erad av två cout-satser.\n\n";
}
```

I vårt första programexempel **MyFirst** fanns en **cout**-sats som resulterade också i en rad på skärmen. Men det finns inget samband mellan antalet **cout**-satser i koden och antalet utskriftsrader på skärmen. Radbyte i utskriften styrs inte av radbyte i koden. Endast **\n** kan åstadkomma radbyte och inget annat. Samma sak är det med mellanslaget. Skriver man ett mellanslag i **cout**-satsen skickas den med till utskrift, annars inte.

Här finns 2 **cout**-satser. Men de producerar en rad utskrift, vilket beror på att det inte finns något **\n** i slutet av den första **cout**-satsen. Det ser man när man kör programmet:

```
Detta är EN utskriftsrad producerad av två cout-satser.
```

Observera också att utskriften blir **... producerad ...** och inte **... produc erad ...** vilket beror på att inget mellanslag skrivs i kodens första **cout**-sats efter **produc** och inte heller i andra **cout**-satsen före **erad**. Man får intrycket att det endast är en utskrift på skärmen som görs här. Och intrycket är rätt: Hur många **cout**-satser man än skriver, det handlar om en enda s.k. **cout**-*ström* som initieras första gången man skriver **cout** i koden. Sedan fortsätter bara strömmen att skriva ut i nästa **cout**-sats. Det som skickas till **cout**-strömmen visas kontinuerligt på skärmen.

Om det nu är så att det hela är en enda kontinuerlig **cout**-ström varför ska man ha flera **cout**-satser? Räcker det inte med en? Ju det gör det. Men vad gör man om satsen är för lång och inte ryms på en rad i koden. Lösningen är att bryta rad på ett lämpligt ställe. Men vad är lämpligt?

Radbrytning i koden

För att förebygga missförstånd borde först påpekas att det som diskuteras här – radbrytning i koden – måste skiljas från radbyte som koden åstadkommer i utskriften när man kör programmet. Det sistnämnda görs med **\n** och diskuterades tidigare. Här pratar vi om att bryta rad när man skriver C++ kod då det t.ex. inte finns tillräckligt med plats på samma rad i editfönstret. Generellt kan **Enter**, mellanslag och tabulator – de kallas *vita tecken* – vara lämpliga ställen för radbrytning. T.ex. kan man bryta rad *i koden* på alla ställen där ett mellanslag förekomme *i koden*. Detta gälller dock inte för mellanslag *i en sträng:*

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

T.ex. ger följande radbrytning i koden kompileringsfel, därför att den sker mitt i en sträng:

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

Vill man ändå bryta rad i koden mitt i en sträng, kan man antingen dela upp den i *två* strängar och skicka mellan dem utmatningsoperatorn:

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

Eller man använder ett speciellt radfortsättningstecken:

Radfortsättning i koden

Radfortsättningstecknet i C++ är ”backslash **** direkt åtföljt av **Enter**” *utan* mellanslag däremellan. Kom ihåg att **** *åtföljt av ett tecken* ger en escapesekvens. Om detta tecken är **Enter** betyder det radfortsättningstecknet. Följande sats ger inget kompileringsfel om **Enter** trycks direkt efter **** :

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

Vi skriver andra kodraden längst till vänster för att inte i onödan få ett tomrum i utskriften av strängen. Radfortsättningstecknet borde användas bara om det inte finns en annan lösning då den ofta försvårar kodens läslighet.

En annan – mycket bättre – lösning är faktiskt konkatenering som vi redan använde när vi delade upp strängen i *två* strängar. Där kan man förenkla ytterligare genom att utelämna utmatningsoperatorn:

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

Detta går, eftersom båda delar i **cout**-satsen är strängar. Konkatenering behandlas i nästa avsnitt.

3.4 Konkatenering

```
// Concat.cpp
// Sammanslagning av flera cout-satser: "concatenation"
// Att bryta en sats över flera rader
#include <iostream>
using namespace std;

int main()
{
   cout << "\n\tNu kommer två rader text\n"
           "\tproducerade av EN cout-sats.\n\n";
}
```

Till skillnad från programmet **Cout** (sid 41) har vi här endast *en* **cout**-sats som skickar en text till utskrift med ett **\n** mitt i. Därför ger programmet **Concat** två rader text när det exekveras:

```
        Nu kommer två rader text
        producerade av EN cout-sats.
```

Radbyte i utskriften görs efter **text** då **\n** finns i koden också efter **text**. En annan sak är radbrytning i koden: Efter att med " ha avslutat den första strängen öppnas en ny sträng på nästa rad. Den första strängen slås ihop i **cout**-strömmen med den andra. Tekniken kallas för *konkatenering*, dvs ihopslagning av strängar.

Concatenation

eller **cat**enation betyder sammanslagning och förekommer i en rad olika sammanhang, inte bara i C++ [*]. I programmet **Cout** (sid 41) hade vi två separata **cout**-satser. I programmet **Concat** ovan har vi konkatenerat dem till en enda. Generellt kan man skriva en enda **cout**-sats och skicka utmatningsoperatorn **<<** mellan de olika delarna. Den allmänna strukturen ser ut så här:

```
cout << ... << ... << ... << ... ;
```

där **...** står för de delar som ska skrivas ut. Satsen kan gå över flera rader, men måste avslutas med ett semikolon då det är en enda sats. Radbrytningar i koden kan göras på ställen där mellanslag förekommer. Denna allmänna struktur kan användas även om de delar **...** som ska skrivas ut, är olika typer av data, t.ex. tal, tecken, text, … . Just den här egenskapen gör konkateneringen till en väldigt använd-

[*] I C++ finns funktionen **strcat()** som gör **str**ing **cat**enation och konkatenerar två strängar. Samma sak gör metoden **concat()** i Java. I Unix, som är skrivet i C, finns kommandot **cat** som konkatenerar data från olika filer och slår ihop dem till en fil. T.ex. kopierar kommandot **cat file1 file2 file3 > nyfil** de tre filerna till **nyfil**.

bar teknik som vi i fortsättningen kommer att utnyttja i nästan alla våra programexempel där olika datatyper är inblandade i utskriften.

Att "rita" i textmiljö

Testa själv att köra följande program som består av en **cout**-sats som konkatenerar bara strängar och därför kan klara sig med en enda utmatningsoperator **<<** :

```
// Figure.cpp
// Ritar en hjärtlig hälsning till alla som lärt sig cout
#include <iostream>
using namespace std;

int main()
{
    cout << "\n              *        *              "
            "\n            *   *   *   *            "
            "\n          *       *       *          "
            "\n         *                 *         "
            "\n         *                 *         "
            "\n         *     Grattis     *         "
            "\n          *                *         "
            "\n            *            *           "
            "\n              *        *             "
            "\n                *    *               "
            "\n                  *                 "
            "\n\n";
}
```

Konkateneringsopratorn +

Det finns i C++ även möjligheten att använda operatorn **+** för att konkatenera strängar. Men detta gäller endast för sträng*variabler*, inte för strängkonstanter. T.ex.:

```
string s = "Slut på";
string t = " kap 3";
cout << s + t;
```

I C++ är **string** datatypen för strängvariabler (sid 53). **s** och **t** är variabler av datatypen **string**. Utskriften blir då **Slut på kap 3**. Operatorn **+** betyder här inte längre addition av tal utan *konkatenering* dvs sammanslagning av strängvariabler. Det är datatypen som avgör tolkningen av **+**. Men man kan inte använda **+** direkt på sträng*konstanter*. **OBS!** `cout << "Slut på" + " kap 3";` ger kompileringsfel. En förklaring ges senare (sid 71). Däremot går det bra att skriva `cout << "Slut på" << " kap 3";` eller t.o.m. `cout << "Slut på" " kap 3";` eller på separata rader:

```
cout << "Slut på"
        " kap 3";
```

Detta har använts i programmen **Concat** och **Figure**.

Övningar till kapitel 3

3.1 Undersök skillnaderna mellan tangenterna *apostrof*, *citationstecken* och *accent*. Mata in koden till programmet **MyFirst** (sid 34), kompilera och kör. Modifiera sedan programmet **MyFirst** genom att ta bort **using**-satsen före **main()** och ersätta istället i **main()** alla **cout** med **std::cout** (läs om detta på sid 37). Kompilera och kör.

3.2 I **MyFirst** (sid 34) finns en **cout**-sats. Skriv om programmet så att det förekommer först tre och sedan två **cout**-satser i det. Använd **\n** i strängarna. Modifiera ytterligare genom att snygga till layouten så att texten hamnar med lite avstånd från konsolens övre och vänstra rand.

3.3 Sätt in följande kod i ett C++ program för att testa vad den ger för utskrift:

```
cout << "****\n";
cout << "*****\n";
cout << "******\n";
cout << "*******\n";
cout << "********\n";
cout << "*******\n";
cout << "******\n";
cout << "*****\n";
cout << "****\n";
```

Konkatenera alla satser till en enda **cout**-sats så att du får samma utskrift.

3.4 Skriv ett program och testa vilken utskrift följande satser ger:

```
cout << "Jag";
cout << "heter";
cout << "K.\n Vad heter du? ";
```

Lägg till och ta bort mellanslag, radbyte och tabulator på lämpliga ställen för att få en snygg utskrift, utan att slå ihop de tre **cout**-satserna till en.

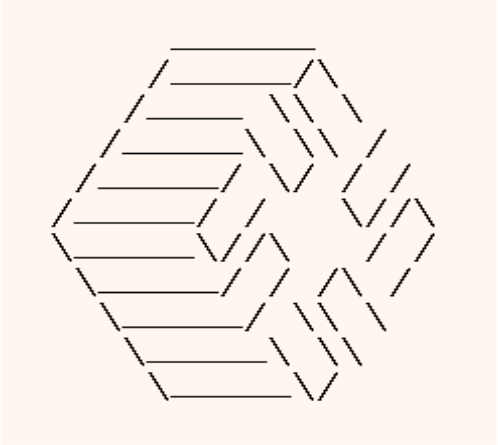
3.5 Vilken utskrift ger följande satser?

```
cout << "*\n**\n***\n****\n*****";

cout << "*****\n****\n***\n**\n*";
```

Sätt in dem i ett C++ program och testa.

3.6 Mata in koden till programmet **Figure** (sid 45), kompilera och kör det. Modifiera det till att skriva ut formen av en oval, en triangel, en rektangel, en pil, en romb, ett kryss, en stjärna eller en blomma, av stjärnor (*****).

3.7 Rita följande figur med en enda utskriftssats genom konkatenering.



Se upp för skillnaden mellan slash `/` och backslash `\`. Använd två backslash `\\` i koden – som en escapesekvens inbakad i den konkatenerade strängen – för att åstadkomma en backslash `\` i utskriften (Läs om escapesekvenser på sid 86).

Kapitel 4

Grundbegrepp i programmering

Ämne	Sida	Program
4.1 Datatyper	49	**Datatype**
4.2 Variabler	53	
- Regler för namngivning	53	
4.3 Deklaration och initiering av variabler	55	**Variable**
- Deklaration och initiering i samma sats	57	**DefInitial**
- Oinitierade och väl definierade variabler	58	**NoInitial**
4.4 Överskrivning eller kan **x = x + 1** vara sant?	60	**OverWrite**
4.5 Inläsning av data	62	**Cin** (Si-in)
4.6 Inmatning – Bearbetning – Utmatning	64	**Hour2Sec**
4.7 Ökningsoperatorn **++**	67	**Increment**
4.8 Sammansatt tilldelning	69	**CompAssign**
4.9 Modulooperatorn	66	
Övningar till kapitel 4	72	

4.1 Datatyper

Hittills har vi i våra program skrivit ut endast strängar och tecken. I koden har vi avgränsat strängar med citationstecken `" "` och tecken med apostrofer `' '`. T.ex. är `'a'` ett tecken. Apostroferna kring `a` talar om att `a` ska tolkas som tecken. Men hur är det med siffror? De kan representera tal, tecken eller sträng. T.ex. `9` är ett tal. När det i koden skrivs *utan* apostrofer tolkas det som tal, men *med* apostrofer `'9'` som tecken, medan `"9"` tolkas som sträng. På skärmen ser man ingen skillnad: Alla dessa tre satser skriver ut `9` på skärmen:

```
cout << 9;
cout << '9';
cout << "9";
```

Hur kan man skilja åt dem? Det behövs, för med tal kan man räkna, inte med tecken eller strängar. Tecken och strängar kan man slå ihop, vilket resulterar i en sträng. Ihopslagning av siffror däremot ger tal som i regel tolkas enligt det decimala talsystemet. Så, *operationer* som man utför med dem kräver att man kan skilja åt dem.

Följande program demonstrerar skillnaderna mellan tal-, tecken- och strängar och vad som kan hända om man inte skiljer åt dessa tre olika typer av data:

```
// Datatype.cpp
// Utskrift av olika typer av data: tal, tecken och text
// ' ' är symbolen för datatypen teckenkonstant
// " " är symbolen för datatypen strängkonstant
// Avsaknaden av dessa är symbolen för datatypen talkonstant

#include <iostream>
using namespace std;

int main()
{
  cout << "\n Detta är talet                    "   <<  9
       << "\n Talet 9 plus talet 9 ger         "    << (9 + 9)

       << "\n\n Detta är tecknet                  " << '9'
       << "\n Tecknet 9 plus tecknet 9 ger   " << ('9' + '9')

       << "\n\n Detta är talet               " << 9
       << "\n Talet 9 plus tecknet 9 ger     " << (9 + '9')

       << "\n\n Detta är strängen             " << "9"
       << "\n Strängkonkatenering ger         " << "9" << "9"

       << "\n";
}
```

En programkörning ger följande utskrift:

```
Detta är talet                    9
Talet 9 plus talet 9 ger         18

Detta är tecknet                  9
Tecknet 9 plus tecknet 9 ger    114

Detta är talet                    9
Talet 9 plus tecknet 9 ger       66

Detta är strängen                 9
Strängkonkatenering ger          99
```

Satsen `cout << (9 + 9);` ger utskriften `18` medan `cout << ('9'+'9');` ger utskriften `114`. Det första resultatet är klart: Parentesen gör att talet `9` adderas med talet `9` och resultatet `18` skrivs ut. Utskriften `114` däremot beror på att `'9'` inte är tal utan tecken. Här omvandlas tecknet `'9'` till sin kod. Varje tecken, vare sig bokstav, siffra eller specialtecken, har sin kod. Dessa koder kallas *ASCII-koder*. *ASCII* är en standard för omvandling mellan tecken och heltalskoder, vilket vi kommer att ta upp senare (sid 83). Tecknet `'9'` har ASCII-koden 57 som adderas med 57, så att `'9'+'9'` blir `114`. Plustecknet i `'9'+'9'` är vanlig addition. Tecknens ASCII-koder adderas *först* som gör att summan `114` bildas. *Sedan* skickas den till `cout` som skriver ut `114`.

Man kan undra: varför gäller inte samma resonemang i satsen `cout << '9';` dvs varför skriver denna sats inte ut 57? Visserligen lagras `'9'` som 57. Men `cout` omvandlar ASCII-koden 57 till *tecknet* `9`, därför att `9` står inom apostrofer som talar om vilken *typ av data* det är, nämligen tecken. Alltså skrivs ut *tecknet* tillhörande koden 57 ut och det är `9`. Det förekommer varken `+` eller någon annan räkneoperation i `cout` -satsen. När det gäller `cout << ('9'+'9')` står *summan* i parentes och summan är ett *tal*. P.g.a. parentesen utförs additionen *före* utskriften. I C++ kod är det en väsentlig skillnad mellan *talet* `9` och *tecknet* `'9'` fast de ser likadana ut när de skrivs ut.

Men varför ger `(9 + '9')` utskriften `66`? Av samma anledning som `('9'+'9')` gav utskriften `114`, nämligen: Talet `9` plus koden 57, som är ASCII-koden till tecknet `'9'`, ger `66`. Även här omvandlas tecknet `'9'` till sin ASCII-kod 57 först, adderas med talet `9` sedan och skickas till `cout` sist. Så skrivs ut `66`.

I `"9" << "9"` har plustecknet ersatts av `<<` . Addition har ersatts av konkatenering (sid 44). Strängarna `9` och `9` konkateneras med varandra (sid 44), de sätts ihop till strängen `99` innan de skrivs ut. Vanlig addition är inte definierad för strängar. Så, koden `"9" + "9"` skulle ge kompileringsfel.

Operatorprioritet

Vad som görs *först* och vad *sedan* i en sats med flera opratorer, t.ex. i satsen `cout << (9 + '9');` bestämdes i programmet `Datatype` (sid 49) med hjälp av paren-

teser. Skriver man inga parenteser – vilket i det här fallet också skulle gå bra och ge samma resultat – avgörs av *operatorprioritet* dvs vilken operator som har högre prioritet, i detta fall additionsoperatorn **+** eller utmatningsoperatorn **<<** . Operatorn med högre prioritet utförs först när båda förekommer utan parentes i en och samma sats. Faktum är att **+** har högre prioritet än **<<** . Därför skulle det bli samma resultat om man skrev utskriftssatserna i **`Datatype`** utan parenteser. Är man osäker på de inblandade operatorernas prioritetsordning kan man alltid sätta parentes för att föreskriva vad som ska göras först. Vi valde göra så för att inte behöva beakta operatorprioritet just nu.

Vad är en datatyp?

Anledningen till att vi måste skilja mellan *talet* **`9`**, *tecknet* **`'9'`** och *strängen* **`"9"`** är att de tillhör olika typer av data. All data representeras i datorn med en sekvens av ettor och nollor. För att kunna identifiera dem, använda de rätta operationerna på dem och åter presentera dem utåt i ursprungligt skick, måste de skiljas åt. Nyckeln till denna identifiering är begreppet *datatyp* – ett nyckelbegrepp inom all programmering oavsett programmeringsspråk.

En datatyp är en föreskrift om

1. hur en viss typ av data ska lagras i datorn,
2. hur mycket minne denna typ av data tar och därmed hur stora värden den kan lagra (det tillåtna värdeområdet),
3. vilka operationer man får utföra med denna typ av data.

Data lagras i datorn på olika sätt. Punkt 1 handlar om *hur* informationen ska lagras, *på vilket sätt* data ska omvandlas till ettor och nollor. Ett tal lagras direkt medan ett tecken måste först kodas. Det är koden som lagras. För att data ska kunna lagras, måste den först omvandlas till ettor och nollor. Omvandlingen sker med olika algoritmer när det gäller t.ex. heltal och decimaltal. Datorn måste ha information om vilken typ av data det handlar om, för att kunna välja rätt algoritm för denna omvandling.

Punkt 2 talar om att varje datatyp får ett bestämt minnesutrymme tilldelad. Av utrymmets storlek följer direkt vilka max- och vilka min-gränser de värden får anta. Man talar om datatypens tillåtna intervall eller värdeområde.

Punkt 3 definierar vad som får *göras* med värden tillhörande en datatyp. Allt som kan göras med tal kan inte göras med tecken och omvänt: Tal kan adderas medan tecken inte kan adderas. Samma sak är det med *strängar* som inte heller kan adderas, de kan däremot konkateneras. De tillhör en tredje typ av data som varken är tal eller tecken, fast de är sammansatta av tecken. Det finns ännu fler typer av data som vi inte än lärt känna.

Olika programmeringsspråk behandlar sina datatyper på lite olika sätt. C++ är ett *strikt typbestämt* språk (eng. *strongly typed language*) vilket innebär att kontrollen över datatyper är väldigt hård. All data som behandlas i ett C++ program måste utan undantag vara typbestämd. Man måste explicit ange datatypen till allt *innan* man använder det. Data utan information om datatypen kan inte bearbetas.

Skriver vi en bokstav med apostrofer i koden, t.ex. `cout << 'a';` tolkas datatypen som *tecken*. Vi kan kompilera och får utskriften `a`. Samma sak är det med `cout << "a";` Men datatypen är *sträng*. Satsen kan kompileras och ger samma utskrift, ty även *en* bokstav kan anses som sträng, den minsta möjliga. Skriver vi däremot en bokstav utan apostrofer, t.ex. `cout << a;` blir det kompileringsfel. Anledningen är att C++ inte kan bearbeta `a` då det inte kan identifiera `a`:s datatyp. `a` utan apostrofer eller citationstecken är varken en tecken- eller en strängkonstant. Talkonstant kan det inte heller vara. Ja, `a` är ingen konstant alls. `a` är en *variabel*. Begreppet tas upp i nästa avsnitt.

4.2 Variabler

Om man skriver `cout << a;` utan att ha skrivit något om `a` innan, får man felmeddelandet *unknown identifier*, dvs okänd identifierare. *Okänd* därför att datatypen är okänd *och* dessutom saknar `a` ett *värde*. Dvs `a` är inte data, kan inte representeras med ettor och nollor. `a` är bara en symbol som kallas för *variabel*.

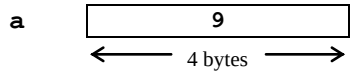
Vad är en variabel?

En variabel är en platshållare (minnescell) för ett värde (data).

I koden får variabeln ett namn som används för att komma åt värdet.

I ett program kan variabelns värde ändras, men inte namnet.

Ex.: Variabeln `a` har värdet `9` och tar 4 bytes i RAM-minnet:



Man kan jämföra en variabel även med en låda och variabelns värde med lådans innehåll. Variabelns namn är då lådans etikett. *Värde* är data i största allmänhet, dvs kan vara – beroende på datatypen (sid 51) – tal, tecken, men även ett sanningsvärde, en sträng, längre text, en fil, ja t.o.m. en bild. Vi kan i fortsättningen komma åt värdet `9` genom att i koden *referera* till variabeln `a`.

Motsatsen till *variabel* är begreppet *konstant*, t.ex. `9`, som inte kan ändra sitt värde under en programkörning. Det kan däremot en variabel göra. För att kunna göra det måste den ha ett namn, t.ex. `a`. Hos en variabel måste man alltid skilja mellan *namnet* och *värdet*, medan konstanter är namnlösa. De har bara sitt värde. Men vilka namn får vi ge till våra variabler? Vi har en ganska stor frihet för detta val. Dock är som vanligt friheten relativ och vi måste följa vissa enkla regler som gäller för all namngivning i C++, inte bara för variabler utan generellt för alla s.k. *identifierare*:

Regler för namngivning

Eftersom dessa regler gäller inte bara för namngivning av variabler utan även för namngivning av andra objekt i ett program formuleras de generellt för att undvika onödig repetition.

En *identifierare* är ett *namn* för en variabel eller andra objekt. Den kan bestå av ett eller flera tecken och får endast innehålla

1. **alla engelska bokstäver,**
2. **alla siffror och**
3. **understreck (underscore _).**

Första tecknet får inte vara en siffra. Reserverade ord i C++ *får inte* användas som identifierare.

Exempel på identifierare är namn på variabler, konstanter, funktioner, klasser, objekt osv. Bland alla specialtecken får endast understreck (underscore _) användas. Självklart får en identifierare inte innehålla mellanslag för då tolkas den inte som *en* utan *flera* identifierare. Mellanslag är *avskiljare* mellan två ord. Så **`tal ett`** är t.ex. inte ett giltigt variabelnamn, inte heller **`1tal`**. Däremot är **`tal1`**, **`tal_ett`**, **`tal_1`**, ja t.o.m. **`_tal`** giltiga variabelnamn.

Svenska specialtecken

Om man får använda de svenska specialtecknen å, ä, ö, Å, Ä, Ö vid namngivning av identifierare beror på programmeringsmiljön. I Visual Studio:s senaste version (2022) t.ex. är det tillåtet. I andra miljöer och versioner måste frågan undersökas. Tecken- och sträng*konstanter* får alltid innehålla svenska specialtecken, eftersom de är icke-namngivna dvs har ingen identifierare, t.ex. **`'ä'`** eller **`"Åsa"`**.

Vi sammanfattar:

> En identifierare består av ETT ord.
>
> Mellanslag är avskiljare mellan två ord.
>
> Användning av de svenska specialtecknen vid namngivning av identifierare är miljöberoende.

Följer man inte *reglerna* ovan får man kompileringsfel och kan inte köra programmet.

Men förutom regler finns det även rekommendationer som är bra att följa, men inte ger kompileringsfel om man inte följer dem. Vi nämner här bara två av dem. För att göra våra program lättare att läsa, förstå, felsöka och göra ändringar i, finns det anledning att följa följande:

Rekommendationer för namngivning

1. **Välj namn som är *beskrivande* dvs beskriver identifierarens *roll* i programmet.**
2. **Bibliotekens klassnamn bör helst inte användas som identifierare.**

Denna rekommendation baseras på de krav som *god programmeringsstil* ställer (sid 39). För att göra våra program lättare att läsa, förstå och göra ändringar i, måste namnen vara *beskrivande*. I programet **`Variable`** (sid 55) har vi valt **`number1`** och **`number2`** som namn för programmets variabler. Namnen kan i princip väljas godtyckliga, dvs skulle lika bra kunna vara t.ex. **`a`**, **`b`**, **`x`**, **`no`**, **`account`** eller vad som helst – upp till reglerna för namngivning. Men vårt val grundas även på rekommendationen ovan: Vi ska lagra tal i variablerna **`number1`** och **`number2`**. Namnet **`account`** vore t.ex. inte beskrivande för variabler som ska lagra vanliga tal.

4.3 Deklaration och initiering av variabler

I alla strikt typbestämda programmeringsspråk – C++ är ett sådant – måste en variabel *deklareras* innan den kan användas i koden. Kod som innehåller variabler utan deklaration ger kompileringsfel. Den allra första tilldelningen av en variabel efter deklarationen kallas för *initiering*. Men först vad exakt betyder *deklaration*?

Deklaration av variabler

Att ange en variabels datatyp i programmet kallas för *deklaration av variabeln*. Det viktigaste skälet för kravet om deklaration är att kompilatorn måste reservera plats för variabelns värde. Vi sa ju att en variabel var en platshållare för ett värde. För att kunna lagra detta värde behövs information om platsens storlek, om sättet att omvandla värdet till ettor och nollor och om vilka operationer man får utföra med värdet. All denna information finns samlad i datatypen (sid 51). Att ge variabeln ett värde kallas för *tilldelning* (eng. *assignment*). Att göra det första gången kallas för *initiering*. Följande program demonstrerar *deklaration* och *initiering av variabler*. Även *tilldelningsoperatorn (=)* introduceras.

```
// Variable.cpp
// Deklarerar och initierar tre int-variabler, lagrar summan
// av de två första i den tredje och skriver ut resultatet
// Initieringen sker med tilldelningsoperatorn =
#include <iostream>
using namespace std;

int main()
{
   int number1;                    // Deklaration av variabler
   int number2;
   int sum;
   number1 = 5;                    // Initiering av variabler
   number2 = 3;
   sum = (number1 + number2);

   cout << "\n\tSumman av " << number1 << " och "
        << number2 << " är " << sum << "\n\n";
}
```

En testkörning av programmet **Variable** ger:

```
          Summan av 5 och 3 är 8
```

Här förekommer tre variabler **number1**, **number2** och **sum**. De behövs för att kunna lagra tre värden i dem. Namnen har vi hittat på. De är helt godtyckliga och skulle lika bra kunna vara t.ex. **a**, **b**, **x**, **kalle**, **...** eller vad som helst. Men enligt rekommendationen på förra sidan ska man för läslighetens skull alltid välja *beskrivande* namn, därför **number1**. I **main()**:s första sats deklareras variabeln **number1**

med **`int number1;`** där **`int`** är ett reserverat ord som står för ***int****eger number*, heltal på engelska. Generellt kan deklaration av en variabel i C++ beskrivas så här:



Man skulle kunna också deklarera alla tre variabler i en och samma sats genom att inleda med datatypen åtföljd av en kommaseparerad lista över variabelnamn:

```
int number1, number2, sum;
```

En sådan konstruktion är endast tillåten om alla variabler är av samma datatyp. Den är mer kompakt, men bidrar inte precis till kodens läslighet. Vi väljer här läsligheten.

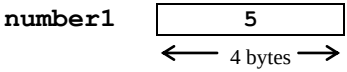
Tilldelningsoperatorn =

Efter deklarationen har vi i programmet **`Variable`** satsen: **`number1 = 5;`**

Här får variabeln **`number1`** värdet **`5`**. Man skulle kunna beskriva bilden så här:

`Variabel` ⟵ **`Värde`**

Dvs tilldelning kan snarare jämföras med en pil som pekar från höger till vänster. I RAM-minnet ser bilden ut så här:



Vi kan i fortsättningen komma åt värdet **`5`** genom att *referera* till **`number1`**. T.ex. om vi nu skriver **`cout << number1;`** får vi *värdet* **`5`** utskrivet. Variabelns *namn* är i koden den mjukvarumässiga motsvarigheten till minnescellens fysiska adress.

Tilldelningen görs med symbolen **`=`** som i matematiken betyder likhet. Men i programmering betyder **`=`** *inte* likhet utan tilldelning och symbolen kallas för *tilldelningsoperatorn*. Den visar ingen likhet utan *uttför* tilldelning vilket betyder att en variabel *får* ett värde. Det är skillnaden mellan *att vara* och *att bli*. Likhet har i C++ en symbolen **`==`** som används i villkor för att testa två värden på likhet.

Samma sak är det förstås med variabeln **`number2`** som i programmet **`Variable`** får värdet **`3`**. Efter tilldelningen av variablerna **`number1`** och **`number2`** utförs additionen **`number1 + number2`**. Här adderas *värdena* lagrade i variablerna **`number1`** och **`number2`**. Resultatet tilldelas variabeln **`sum`**. Vi refererar till värdena med hjälp av variablerna. Att additionen **`+`** görs *först* och tilldelningen **`=`** *sedan* beror på parenteserna i satsen **`sum = (number1 + number2);`** Men även utan parenteser hade vi fått samma resultat då **`+`** binder starkare än **`=`** . Slutligen skrivs alla tre variablers värden ut, konkatenerade med lite text för att göra utskriften användarvänlig.

Deklaration och initiering i samma sats

Ett bra medel mot att glömma variabelinitieringen är att inte avsluta deklarationssatsen förrän man initierat variabeln. Följande program visar att C++ tillåter att deklarera och initiera variabler i en och samma sats:

```
// DefInitial.cpp
// Gör samma sak som programmet Variable, med skillnaden att
// 1) deklaration och initiering skrivs i samma sats och
// 2) summan bildas direkt i cout-satsen: Sparar en variabel
#include <iostream>
using namespace std;

int main()
{
   int number1 = 5;              // Deklaration och initiering
   int number2 = 3;

   cout << "\n\tSumman av " << number1 << " och "
        << number2 << " är " << (number1+number2) << "\n";
}
```

I programmet **Variable** (sid 55) gjordes deklarationen och initieringen av variabler i separata satser. Man kan även slå ihop dessa satser: I programmet **DefInitial** har variabeln **number1** blivit deklarerad och initierad i en och samma sats:

```
int number1 = 5;
```

Samma sak kan man göra med **number2**. Detta är möjligt, för man måste inte deklarera alla variabler i början av programmet. Man kan göra det när det behövs, bara man deklarerar en variabel *innan* man initierar den. Det går t.o.m. att slå ihop de två första satserna i **DefInitial** till en:

```
int number1 = 5, number2 = 3;
```

De två variablers deklaration och initiering kan separeras med komma, vilket dock endast är möjligt om variablerna har samma datatyp. Ska båda variablerna även ha samma värde kan man göra en *dubbelinitiering*:

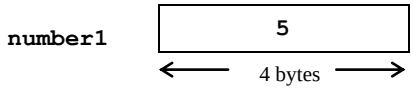
```
int number1, number2;   // Separat deklaration
number1 = number2 = 3;  // dubbelinitiering
```

Men då måste deklarationen stå separat innan. Kom ihåg att tilldelningsoperatorn alltid tilldelar som en pil från höger till vänster. Därför får variabeln **number2** först värdet **3**. Variabeln **number1** får sedan samma värde, dvs variabeln **number2**:s värde som redan är **3**. Programmet **DefInitial** producerar exakt samma utskrift som **Variable** (sid 55).

Vad händer vid deklaration och initiering?

Som exempel tar vi **int number1 = 5;**. Vad händer när variabeln **number1** deklareras till **int** och initieras till **5**?

1. En minnescell reserveras i datorns RAM-minne för lagring av **`int`**-värden. Namnet på minnescellen blir **`number1`**. Storleken på minnescellen bestäms av datatypen **`int`** som i vår C++ installation är föreskriven till 4 bytes dvs 4 x 8 = 32 bitar. (En bit kan lagra *en* 0 eller *en* 1). Detta sker vid kompileringen och upprätthålls tills exekveringen är avslutad. Detta kallas *statisk minnesallokering* i den bemärkelse att den inte kan ändras under exekveringen. *Allokering* betyder reservering av minnesutrymme. Faktiskt är det *initieringen* som allokerar minne. Så här ser det ut när 4 bytes minne reserveras i datorns RAM för variabeln **`number1`**:



2. Specificieringen av datatypen gör att programmet kan tolka innehållet i minnescellen ovan när den fylls med ett värde. Det är 32 ettor och nollor som måste tolkas som ett heltal. Olika datatyper har olika algoritmer för omvandling av data till ettor och nollor och omvänt. Decimaltal omvandlas på ett annat sätt än heltal eller tecken osv. Datatypen är avgörande: *Heltalet* **`1`** t.ex. består av en annan följd av ettor och nollor än *decimaltalet* **`1.0`**. En annan digital sekvens har *tecknet* **`'1'`** för att inte tala om *strängen* **`"1"`**.

3. Namngivning har med *adressering* att göra. Minnescellens *fysiska* adress i RAM kopplas till det *logiska* namn **`number1`**. Det görs för att komma åt minnescellen genom att referera till variabelnamnet. Med andra ord, variabler gör minnescellerna i hårdvaran åtkomliga dvs *adresserbara* för mjukvaran.

Dessa punkter förklarar varför C++ strikt följer regeln som ställdes upp i början av detta avsnitt:

> Variabler som inte deklareras innan de används ger kompileringsfel.

Det finns ingen regel som säger att alla variabeldeklarationer måste stå i början av programmet. Man kan deklarera sina variabler när de behövs, bara man gör det *innan* man ger variabeln ett värde, sätter in den i ett aritmetiskt uttryck eller använder den på ett annat sätt. Att vi ändå placerar variabeldeklarationerna i början av våra program har ofta att göra med strukturering, läslighet och god programmeringsstil.

Oinitierade och väl definierade variabler

Vad händer om man deklarerar en variabel men glömmer initieringen? Svaret är miljö- och versionsberoende. I äldre versioner och i vissa miljöer blir det inget kompileringsfel och man kan exekvera programmet. I nyare versioner av Visual Studio sätter kompilatorn stopp för oinitierade variabler. Men kör man en äldre version och kan kompilera, får man ett helt felaktigt värde, när man försöker skriva

ut den icke-tilldelade variabeln. Felet beror inte på att man brutit mot språkets syntaxregler – då hade man fått kompileringsfel – utan man har använt en variabel ”utan” värde. Mjukvarumässigt, är den ”tom”. Men fysiskt är den inte alls tom. Den har kvar ett värde från tidigare användning. Överskriver vi inte det i vårt program får vi det gamla värdet som råkar finnas där.

Följande program är ett exempel på att oinitierade variabler producerar ”skräp”, t.ex. variabeln **number1** som är deklarerad, men inte initierad:

```
// NoInitial.cpp
// Skriver ut ett skräpvärde till en oinitierad variabel
// En deklarerad men oinitierad variabel ger varken kompile-
// rings- eller exekveringsfel utan ett slump-skräpvärde
#include <iostream>
using namespace std;

int main()
{
   int number1, number2;
   number2 = 2;
                  // number1 skrivs ut här, men är ingen
                  // väl definierad variabel:
   cout << '\n' << number1 << " är skräp, medan "
        << number2 << " är en väl definierad variabels"
                                        << " värde.\n\n";
}
```

Om man kör en gammal C++ kompilator kan koden ovan kompileras felfritt. När man sedan exekverar programmet får man nedanstående utskrift med ett *skräpvärde* för den oinitierade variabeln. Äldre versioner av Visual Studio genererar ett s.k. Debug-felmeddelande vid exekveringen. Klickar man på knappen Ignorera för att komma vidare kan resultatet bli:

```
4301064 är skräp, medan 2 är en väl definierad variabels värde.
```

Variabeln **number1** har alltså skräpvärdet **4301064** vid denna körning. En annan körning på samma eller en annan dator kan ge ett annat eller också samma värde. Det är slumpen som styr. Skriver vi inte ut värdet kommer vi kanske inte ens att märka felet. Därför är det god programmeringsstil att endast använda *väl definierade* variabler dvs variabler som är *både* deklarerade och tilldelade. Att bryta mot god programmeringsstil är inte mindre allvarligt än att få kompileringsfel (jfr. sid 39).

Att äldre versioner ac C++ inte sätter stopp för oinitierade variabler har effektivitetsskäl. Man vill ha så lite kontroller som möjligt vid körningen. Man har valt att prioritera snabbheten på bekostnad av stabilitet och säkerhet.

4.4 Överskrivning eller kan `x = x + 1` vara sant ?

För tilldelning använder C++ samma symbol **=** som för likheten i matematik, vilket kan ge upphov till missförstånd då det handlar om två helt olika typer av operationer. *Tilldelning* är en instruktion som skall utföras, medan *likhet* är en jämförelse som endast kan testas om den är sann eller falsk. Vi ska nu titta lite närmare på den praktiska skillnaden mellan tilldelning och likhet. Vid tilldelningen:

variabel ⟵ värde

förekommer **variabel** endast på vänster sidan. **värde** kan vara antingen en konstant som i satsen **number1 = 9;** eller en annan variabels värde. Men vid en annan tilldelning som t.ex.:

x ⟵ x + 1

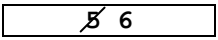
finns *samma* variabel **x** på båda sidor. Översatt till C++ kod blir det:

```
x = x + 1;
```

Har t.ex. **x** värdet **5** *före* denna sats, innebär satsen att **5** ska adderas med **1** och att det nybildade värdet **6** ska tilldelas variabeln **x** dvs

x ⟵ 5 + 1

så att *efter* satsen har **x** värdet **6**. Det nya värdet **6** *skriver över* det gamla värdet **5**:



Detta kallas för *överskrivning* av variabeln **x**, baserad på att variabeln **x** är en platshållare vars värde kan ändras medan namnet bibehålls (sid 53). Följande program visar detta:

```
// OverWrite.cpp
// Demonstrerar överskrivning av variabel
#include <iostream>
using namespace std;

int main()
{
   int x = 5;
   cout << "\nVariabeln har initierats till " << x;

   x = x + 1;                  // Överskrivning av variabel

   // x++;                     // Samma som x = x + 1;
   cout << ", sedan ökats med 1 och är nu " << x << ".\n";
}
```

En körning ger:

```
Variabeln har initierats till 5, sedan ökats med 1 och är nu 6
```

Initieringsvärdet `5` tilldelas variabeln `x`. Progammets centrala sats

```
x = x + 1;
```

ökar värdet till `6` och överskriver det gamla värdet av `x` med detta nya värde. En `cout`-sats skriver ut först initialvärdet, en andra fortsätter utskriften efter satsen ovan med det ökade värdet, båda gånger konkatenerat med en förklarande text.

Vi har i satsen ovan med två olika värden till en och samma variabel `x` att göra, men vid två olika tidpunkter. Det gamla värdet `5` finns i variabeln `x` *före* satsen och det nya värdet `6` finns i variabeln `x` *efter* satsen.

I matematiken betyder tecknet = *likhet*. Därför är det fel att skriva $x = x + 1$ eftersom detta är en ekvation som saknar lösning. Man kan också säga att det är ett falskt påstående som leder till motsägelsen $0 = 1$. Vill man vara matematiskt korrekt måste man använda *två* variabler och skriva så här:

$$x_{\text{nytt}} = x_{\text{gammalt}} + 1$$

I programmeringen däremot betyder tecknet = inte likhet utan *tilldelning*. Därför är det helt OK att skriva `x = x + 1` eftersom det inte handlar om ett påstående som kan vara sant eller falskt utan snarare om en *instruktion* som ska utföras. Samma variabel `x` används på båda sidor av tilldelningstecknet. `x` är *en* platshållare (minnescell) vars innehåll (värde) skall *överskrivas* med satsen `x = x + 1;` Instruktionen lyder att *tilldela* variabeln `x` ett nytt värde, att öka det gamla värdet med `1`.

Filosofiskt handlar det om den klassiska skillnaden mellan *att vara* och *att bli*, mellan *tillstånd* och *handling*, mellan den statiska likheten och den dynamiska tilldelningen. Vid tilldelning relateras sanningen till tiden, dvs frågan är inte *om* utan *när* `x = 5`. Jo, precis när variabeln `x` tilldelas värdet `5`. Inte innan och ev. inte heller efteråt, för redan i nästa programsats kan ju variabeln `x` tilldelas ett annat värde. Med andra ord: Tilldelning är likhet relaterad till tiden dvs vid ett visst ögonblick, medan likheten är tidlös.

Att satsen `x = x + 1;` utför additionen *först* och tilldelnigen *sedan* beror på att operatorn `+` binder starkare dvs har högre prioritet än tilldelningsoperatorn `=` . Därför slipper vi skriva parenteser: `x = (x + 1);` vilket vi hade varit tvungna att göra om `=` hade samma prioritet som eller högre än `+` .

I programmet `OverWrite` kan man ersätta satsen `x = x + 1;` med satsen `x++;` som just nu är bortkommenterad. De gör samma sak: att öka `x` med `1` vilket lätt kan testas genom att aktivera satsen `x++;` och kommentera bort `x = x + 1;` Symbolen `++` (OBS! Utan mellanslag) kallas *ökningsoperatorn* och har en gång gett namnet till språket C++: Tillägget ++ ska antyda att man har lagt till 1 utvecklingssteg till C och därigenom fått fram C++. Ökningsoperatorn kommer att behandlas i detalj senare (sid 67).

4.5 Inläsning av data

Hittills har alla våra programexempel i stort sett bara handlat om att göra utskrifter till skärmen med **cout**. Dessa program har endast utdata och ingen indata. Vill man även läsa in data till programmet, kan man i C++ använda sig av **cin**. Man kan också säga att vi nu lär oss ytterligare ett sätt att tilldela variabler. Hittills tilldelade vi via tilldelningsoperatorn. Nu behandlas tilldelning via inläsning. Följande program läser in två värden och tilldelar två variabler:

```
// Cin.cpp
// Läser in två tal, skriver ut dem och deras produkt
// Inläsning med inmatningsoperatorn >> och cin (uttal:si-in)
// Console Input som är definierad i biblioteket iostream
// Det är god programmeringsstil att låta inläsningen föregås
// av en ledtext som instruerar användaren vid körningen
#include <iostream>
using namespace std;

int main()
{
   int number1, number2;

   cout << "\nGe ett tal och tryck på Enter:  "; // Ledtext
   cin >> number1;                               // Inläsning

   cout << "\nGe ett tal till och tryck på Enter:  ";
   cin >> number2;

   cout << "\n" << number1 << " gånger " << number2
        << " blir " << (number1*number2) << "\n\n";
}
```

Inmatning med `cin` *och inmatningsoperatorn* >>

I programmen **01-10** var det utdata som skickades från programkod till bildskärmen. Data som matas in från tangentbordet eller läses in från filer, är indata. Hur man får indata in i datorn visar bilden på sid 24: Både indata och programkod måste lagras i RAM-minnet. Programkoden laddas från hårddisken till RAM-minnet när maskinkoden i den exekverbara filen körs. Indata däremot måste matas in under programkörning och mellanlagras i en minnescell i RAM-minnet innan den kan vidarebearbetas av programmet. Mjukvarumässigt innebär detta att indata måste tas emot och lagras i en variabel – ytterligare ett skäl till att variabeln måste vara definierad, dvs vara associerad med en minnescell av en viss storlek som är reserverad i datorns RAM-minne. Variabelns namn blir en referens till minnesadressen som sedan kan användas för att komma åt data. Medan allokeringen av minnesutrymme i regel sker under kompilering via variabeldefinition, måste inmatningen göras under exekveringen. Därför avbryts exekveringen när en inmatning ska ske. För en ovan användare kan programavbrott, när markören bara blinkar i det vänstra övra hörnet av en tom skärm, tolkas som program- eller t.o.m. datorkrasch. I själva

verket väntar programmet på att användaren ska mata in värden och trycka på **Enter**. Användaren kan inte hantera situationen. För att undvika den är det god programmeringsstil att koda användarvänligt och ge användaren en anvisning med hjälp av en vanlig **cout**-sats. I programmet **Cin** görs detta med:

```
cout << "\nGe ett tal och tryck på Enter:  ";
```

Satsen skriver ut en *ledtext* som instruerar programmets användare vad denne ska göra. Det är placeringen av satsen som är av betydelse: Den måste stå strax innan programmet når **cin**-satsen som utför själva inläsningen:

```
cin >> number1;
```

cin står för **c**onsole **in**put, uttalat ”si-in”, dvs inläsning från tangentbordet. Detta innebär också att riktningen av utmatningsoperatorn som används tillsammans med **cout**, nu vänds och blir den s.k. *inmatningsoperatorn* **>>**. Man skulle kunna tolka **>>** som en pil från vänster till höger. Data strömmar i pilens riktning från datorns standard inputenhet som är tangentbordet till variabeln **number1**. Detta innebär att det tal som användaren matar in när ledtexten kommer upp, läses in från tangentbordet och lagras i variabeln **number1**. Dessutom ”ekas” (skrivs) det på skärmen. Därmed har variabeln **number1** blivit tilldelad det inmatade värdet. Dess reserverade ”tomma” minnescell har blivit fylld med ett värde. Det är en *tilldelning* som i det här fallet skett via inläsning. När vi senare i programmet – det sker i den avslutande **cout**-satsen – refererar till variabeln **number1**, får vi tillbaka detta värde. Samma resonemang kan tillämpas på variabeln **number2**: Även här skrivs en ledtext ut strax före den **cin**-sats som läser in ett värde till **number2** vars värde efterfrågas i den avslutande **cout**-satsen. Dessutom skriver den ut produkten **number1*number2** där ***** är symbolen för multiplikation. En körning av programmet **Cin** visar dialogen:

```
Ge ett tal och tryck på Enter:  5

Ge ett tal till och tryck på Enter:  6

5 gånger 6 blir 30
```

Precis som **cout** tillåter även **cin** konkatenering dvs vi skulle kunna skriva inläsningen av båda variabler i en enda konkatenerad **cin**-sats:

```
cin >> number1 >> number2;
```

I så fall måste förstås endast en ledtext skrivas ut strax före denna **cin**-sats. Vid inmatningen av två separata värden till de båda variablerna kan mellanslag användas som avskiljare. Ledtexten borde ge bl.a. information om avskiljaren, t.ex. **. . . skilda med mellanslag**. Alla s.k. *vita tecken* dvs **Enter**, mellanslag och tabulator tolkas av **cin** som avskiljare vid inmatning. OBS! Inget komma. I nästa avsnitts programexempel används **cin**-satsen för att läsa in tre värden till tre variabler.’

4.6 Inmatning – Bearbetning – Utmatning

De fyra grundräknesätten **+**, **–** , *****, **/** är exempel på *aritmetiska operatorer*. De objekt som en operator tillämpas på, kallas *operander*. T.ex. i uttrycket **a + b – 4** är **a**, **b** och **4** operander. Ett *uttryck* är en kombination av variabler, konstanter, operatorer och vanliga parenteser som till slut, när uttrycket beräknas, returnerar ett värde. När detta värde är ett *tal*, pratar man om *aritmetiska uttryck* eller räkneuttryck, till skillnad från *logiska utttryck*. Exempel på aritmetiska uttryck är:

```
number1 * number2
a + b – 4
5 * (fahrenheit – 32) / 9
```

Det enklast tänkbara uttrycket – ett specialfall – är *en* konstant eller *en* variabel. Men det typiska är att det förekommer operatorer i ett uttryck. Ett annat exempel på ett aritmetiskt uttryck är:

```
3600*tim + 60*min + sek
```

Om **tim** är antalet timmar, **min** antalet minuter och **sek** antalet sekunder beräknas här det totala antalet sekunder. Att ***** görs först och **+** sedan beror på att i C++ multiplikationsoperatorn ***** – precis som i matematiken – har en högre prioritet än additionsoperatorn **+**. Därför behövs inga parenteser. I följande program används uttrycket ovan för att omvandla all tid som matas in som **tim**, **min** och **sek** till sekunder. Dessutom introduceras programstrukturen *inmatning – bearbetning – utmatning*.

```
// Hour2Sec.cpp
// Läser in tiden i timmar, minuter och sekunder, omvandlar
// allt till sekunder och skriver ut resultatet
// Aritmetiskt uttryck för beräkning av totalsekunder
#include <iostream>
using namespace std;

int main()
{
   int  tim, min, sek, totalsek;

   /* I n m a t n i n g */
   cout << "\nGe timmar, minuter, sekunder "
        << "skilda med mellanslag:  ";
   cin  >> tim >> min >> sek;

   /* B e a r b e t n i n g */
   totalsek = 3600*tim + 60*min + sek; // Aritmetiskt uttryck

   /* U t m a t n i n g */
   cout << '\n' << tim << " timmar, "<< min<< " minuter och "
        << sek << " sekunder är " << totalsek
        << " sekunder totalt.\n\n";
}
```

En körning av programmet **Hour2Sec** ger t.ex. följande dialog:

```
Ge timmar, minuter, sekunder skilda med mellanslag:  4 25 10

4 timmar, 25 minuter och 10 sekunder är 15910 sekunder totalt.
```

Beräkningen av det totala antalet sekunder har i programmet gjorts med uttrycket:

```
3600*tim + 60*min + sek
3600*  4 + 60* 25 +  10
```

Lite svårare är det att vända på problemet och skapa ett program där man matar in det totala antalet sekunder, t.ex. **15910** och får svaret att det är **4** timmar, **25** minuter och **10** sekunder. Att skriva ett sådant program lämnar vi till övningarna (övn 4.9 och 4.10). Algoritmen som utgör problemets egentliga svårighet, finns återgiven där. Läs nästa avsnitt ***4.6 Modulooperatorn %*** för att fötså denna algoritm.

Strukturen Inmatning – Bearbetning – Utmatning

Vid sidan om aritmetiska uttryck, introducerar programmet **Hour2Sec** ett koncept inom programmering som kan bidra till att uppfylla de krav på läslighet, förståelighet och ändringsbarhet som vi ställde upp för god programmeringsstil (sid 40). Det handlar om strukturering av program.

Det enklast tänkbara sättet att strukturera ett program är att dela in det i de tre naturliga stegen *inmatning – bearbetning – utmatning* som man kanske helt spontant tar när man utvecklar ett program. I **Hour2Sec** är dessa tre steg framhävda med vit bakgrund och skrivna i blockkommentar. Där *matas in* först programmets data: timmar, minuter och sekunder. Sedan *bearbetas* dessa data genom att beräkna antalet totalsekunder och lagra resultatet i en ny variabel. Slutligen *matas ut* bearbetningens resultat genom att skriva ut den nya variabelns värde. Man borde hålla sig till denna ordning om man inte har någon speciell anledning att avvika från den. Det finns i regel ingen anledning att t.ex. splittra utmatningen och skriva en del av den före och en annan del efter bearbetningen. Inte minst när koden växer rekommenderas att utnyttja åtminstone denna naturligt givna struktur i sina program.

Indelningen av programkoden i strukturen *inmatning – bearbetning – utmatning* kan få avgörande betydelse när vi behandlar funktioner (sid 134). Då kommer vi nämligen att separera dessa tre delar, skriva dem i var sin funktion – i alla fall några av dem – och sedan anropa dem från **main()**. Även därför kan det vara nyttigt att vänja sig vid denna goda sed redan nu. I fortsättningen håller vi oss i våra programexempel till konventionen att i regel dela in programkoden i dessa tre delar, utan att explicit nämna det.

Det följer en liten parentes om den nya räkneoperationen *modulo* som kommer att behövas för att genomföra övningarna 4.9 och 4.10.

4.7 Modulooperatorn %

Symbolen **%** har i C++ ingenting med procenträkning att göra utan står för ett nytt räknesätt som kallas *modulo*. Man dividerar två heltal utan att gå vidare till decimalerna, tar resten och ignorerar resultatet. T.ex.: **9 % 2** ger **1**, eftersom **9** heltalsdividerat med **2** ger **4**. Sedan blir en rest kvar på **1**. Modulooperationen ignorerar **4** och tar resten 1. Pga att **9 % 2 = 1** är **9** udda. Däremot **8 % 2** ger **0**, eftersom **8** dividerat med **2** ger resultatet **4** och resten **0**. Därför är **8** ett jämnt tal. Alla jämna tal ger rest **0** vid heltalsdivision med **2**. Alla udda tal ger rest **1** vid heltalsdivision med **2**. Resten vid heltalsdivision kallas även *modulo*: **9** *modulo* **2** ger **1**. Man kan uppfatta räknesättet *modulo* även som en upprepad subtraktion: Man drar av **2** från **9** så många gånger det bara går och tar det som blir kvar. Fyra gånger går det att ta bort **2** från **9**, kvar blir **1**. Därför är **9 % 2** = **1**. Generellt innebär att *räkna modulo* **a** att man bortser från (kastar bort) alla multipler av **a** och behåller resten.

Tillämpningar av modulo

Det finns många intressanta tillämpningar av modulooperatorn. T.ex. avgör följande kod *(ett villkor)* om talet **number** är jämnt eller udda:

```
number % 2 == 0
```

Koden betyder: **number** *heltalsdividerad med* **2** *ger resten* **0**.

Det är samma sak som att säga: **number** är jämnt delbart med **2**. Om det är fallet är **number** jämnt annars udda. Ett direkt villkor för att **number** ska vara udda är:

```
number % 2 == 1
```

I andra sammanhang är modulo nyckeloperationen vid omvandling mellan olika talsystem, t.ex. mellan det decimala och binära talsystemet. I matematiken används modulo bl.a. för att bestämma den största gemensamma delaren av två heltal (Euklides algoritm).

En rolig och enkel användning av modulooperatorn är följande exempel:

> Idag är fredag och du vill träffa din kompis om **11** dagar. Vilken veckodag blir det?

Om vi numrerar veckodagarna stigande från 1 med början på måndag så att fredag blir den 5:e veckodagen, får du svaret på frågan ovan genom att räkna modulo **7**:

```
(5 + 11) % 7 = 2
```

Dvs veckodagen i frågan är **tisdag**. Med andra ord man lägger till aktuell veckodag, antalet dagar och räknar modulo **7**. I själva verket handlar det om en omvandling av det decimala talsystemet med basen 10 och siffrorna 0-9 – det system vi är vana vid – till *veckodagarnas system* dvs till *talsystemet med basen* **7** som använder sig av siffrorna 0-6.

4.8 Ökningsoperatorn ++

Denna operator som gett namnet till språket C++, nämndes redan i avsnittet om överskrivning (sid 60). Det finns två varianter av ökningsoperatorn: Man kan skriva den *före*, så här **++a**, eller *efter* operanden, så här **a++**. Sätts den *efter* operanden talar man om ökningsoperatorns *postfix*variant. Skrivs den *före* operanden blir det *prefix*varianten. Följande programexempel demonstrerar skillnaden mellan post- och prefixvarianten:

```
// Increment.cpp
// Jämför a och b:s värden efter b = a++ med värden efter
// b = ++a utgående från samma värde för a
// Skillnaden mellan Ökningsoperatorns post- & prefixvariant
#include <iostream>
using namespace std;

int main()
{
   int a, b;

   a = 0;
   b = a++;
   // Samma som:
   // b = a;
   // a = a + 1;
   cout << "\n a=0: Efter b=a++; blir b = " << b
        << " och a = " << a << '\n';

   a = 0;
   b = ++a;
   // Samma som:
   // a = a + 1;
   // b = a;
   cout << "\n a=0: Efter b=++a; blir b = " << b
        << " och a = " << a << "\n\n";
}
```

En testkörning ger:

```
 a=0: Efter b=a++; blir b = 0 och a = 1

 a=0: Efter b=++a; blir b = 1 och a = 1
```

Postfixvarianten **a++;** betyder:

”Utför satsen med det aktuella värdet på variabeln **a** och öka den därefter med **1**”.

Närmare bestämt ökar **a**:s värde *efter* satsen dvs efter *semikolonet*. Satsen **a++;** är en kompakt kod för ökning med **1** genom överskrivning, dvs:

`a++;` gör samma sak som `a = a + 1;`

Nu ser man att ökningsoperatorn består av två operationer, addition *och* tilldelning. Observera att `a++;` *inte* gör samma sak som `a + 1;` I `a++;` ingår även en tilldelning medan `a + 1;` endast innehåller en addition. Ökningsoperatorn tar hänsyn till att det vid överskrivning endast finns *en* variabel vars värde överskrivs. Därför förekommer i `a++;` variabeln `a` bara en gång. Prefixvarianten `++a;` betyder:

”Öka först variabeln `a`:s värde med `1` och utför därefter satsen med det nya ökade värdet på `a`”.

Även satsen `++a;` gör samma sak som `a = a + 1;` Skillnaden med prefixvarianten blir påtaglig först när det finns något som händer innan och/eller efteråt, dvs när sammanhanget man använder ökningsoperatorn i är lite mer komplex. Programmet **`Increment`** på förra sidan visar ett exempel på ett sådant sammanhang.

För att se skillnaden mellan ökningsoperatorns post- *och* prefixvariant behöver vi en variabel som har samma initieringsvärde. Därför tilldelas i **`Increment`** variabeln `a` värdet `0`. Detta för att en gång använda postfixvarianten och en annan gång prefixvarianten på `a`. I det första fallet görs det i satsen `b = a++;` där tilldelningen utförs *innan* `a`:s värde ökar. Dvs *först* får `b` värdet `0`, *sedan* ökar `a` med `1` och blir `1`. I det andra fallet utförs tilldelningen i satsen `b = ++a;` *efter* att `a`:s värde ökar. Dvs *först* ökar `a` med `1` och blir `1`, *sedan* får `b` detta nyökade värde `1`.

Anmärkningsvärt är att det inte finns någon skillnad mellan ökningsoperatorns post- och prefixvariant när det gäller själva operanden `a` som `++` tillämpas på. I båda fallen ökar operandens värde med `1`. Skillnaden påverkar snarare miljön dvs det som finns kring ökningsoperatorn, i vårt exempel variabeln `b`. Detta beror på att skillnaden mellan post- och prefixvarianten inte ligger i *att* operandens värde ökar med `1` utan *när* detta händer. Skillnaden ligger i saker och tings *ordning*, i vårt fall:

```
b = a++;        gör samma sak som    b = a;
                                     a = a + 1;
```

och

```
b = ++a;        gör samma sak som    a = a + 1;
                                     b = a;
```

Man kan faktiskt i programmet **`Increment`** ersätta satserna till vänster med satserna till höger. Testa gärna! Varför, kan man undra, ska man använda `a++;` eller `++a;` istället för `a = a + 1;` om de åstadkommer samma sak nämligen att öka `a`:s värde med 1? Faktum är att ökningsoperatorn skapar maskinkod som är mycket snabbare och effektivare än maskinkod skapad av tilldelningssatsen.

Det finns i C++ även minskningsoperatorn `--` som fungerar på liknande sätt. Istället för ökning med 1 görs minskning med 1. Även minskningsoperatorn kan sättas antingen efter (postfix) eller före en variabel (prefix):

`a--;` eller `--a;` gör samma sak som `a = a - 1;`

Båda operatorer kan endast öka eller minska med 1, inte med något större värde.

4.9 Sammansatt tilldelning

Ökningss- och minskningsoperatorn ingår i en större grupp av operatorer vars gemensamma egenskap är att de består av *två* operatorer varav en är tilldelning: Ökningsoperatorn **++** t.ex. består av ökning med 1 *och* tilldelning, minskningsoperatorn **--** av minskning med 1 *och* tilldelning. Dessa operatorer används så här: **a++** eller **a--** där **a** är en operand (sid 64). De kallas *unära* operatorer därför att de endast tar *en* operand. Man kan jämföra dem med metoder som har *en* parameter.

En annan grupp är: **+=** **-=** ***=** **/=** **%=**

Vi kallar dem *sammansatt tilldelningar*, eftersom tilldelning förekommer hos alla dessa operatorer. De är sammansatta av tilldelning *och* en operation till. Därför består deras symbol av två tecken. Observera att alla dessa operatorer (precis som **++** och **--**) skrivs *utan* mellanslag. *Med* mellanslag känns de inte igen av kompilatorn och tappar sin betydelse. De sammansatta tilldelningarna är *binära* operatorer: De tar *två* operander, då de används så här: **a += b**. Även de aritmetiska operatorerna **+**, **-**, *****, **/** och **%** är binära.

Operatorn +=

är sammansatt av de två operatorerna addition och tilldelning och betyder:

> Addera först båda sidor av **+=** med varandra och tilldela sedan resultatet till variabeln som står till vänster om **+=** .

Beräkningen utförs helt enkelt i den ordningen man läser: från vänster till höger. T.ex.

sum += a; gör samma sak som **sum = sum + a;**

Dvs addera först **sum** med **a** och tilldela resultatet till **sum**. Operatorn **+=** överskriver variabeln **sum**:s värde direkt. Om a = 1 gör båda satserna ovan samma sak som **sum++;** vilket visar att ökningsoperatorn **++** är ett specialfall av **+=**. På liknande sätt fungerar de andra: **-=**, ***=**, **/=**. De utför *först* en aritmetisk operation och *sedan* en tilldelning.

Programmet **CompAssign** på nästa sida demonstrerar alla binära sammansatta tilldelningsoperatorer. Användaren matar in ett värde till variabeln **a** via tangentbordet som kombineras via tilldelningsoperatorerna **+=**, **-=**, ***=**, **/=** och **%=** med de redan initierade variablerna **sum**, **diff**, **prod**, **div** och **mod**. Dessa namn utför förstås i sig inga aritmetiska operationer, utan är bara valda för att vara beskrivande. Deras initiering är avgörande, annars kommer odefinierade skräpvärden in i beräkningen och förstör resultatet (sid 58).

```
// CompAssign.cpp
// Testar alla binära sammansatta tilldelningsoperatorer
// +=, -=, *=, /= och %= som räknar först och tilldelar sedan
// Konkatenering av strängvariabler
#include <iostream>
#include <string>               // Krävs för cout << s där s
är
using namespace std;            // en variabel av typ string

int main()
{
   int a, sum = 10, diff = 20, prod = 30, div = 42, mod = 42;
   string s = "Slut på", t = " kapitel 4";
   cout << "\nMata in ett heltal:  ";
   cin >> a;
   sum   += a;                  // Samma som sum  = sum  + a;
   diff  -= a;                  //           diff = diff - a;
   prod  *= a;                  //           prod = prod * a;
   div   /= a;                  //           div  = div  / a;
   mod   %= a;                  //           mod  = mod  % a;
   s     += t;                  // + här strängkonkatenering:
                                // Samma som s = s + t; där
   cout <<'\n' <<               // s och t är strängar
   "sum  = 10" << " och a = " << a << ": \tsum  += a "     <<
                          "ger sum  = " << sum << "\n\n"  <<
   "diff = 20" << " och a = " << a << ": \tdiff -= a "     <<
                          "ger diff = " << diff << "\n\n" <<
   "prod = 30" << " och a = " << a << ": \tprod *= a "     <<
                          "ger prod = " << prod << "\n\n" <<
   "div  = 42" << " och a = " << a << ": \tdiv  /= a "     <<
                          "ger div  = " << div << "\n\n"  <<
   "mod  = 42" << " och a = " << a << ": \tmod  %= a "     <<
               "ger mod = " << mod << "\n\n" << s << "\n\n";
}
```

En testkörning av programmet **CompAssign** ger:

```
Mata in ett heltal:  4

sum  = 10 och a = 4:    sum  += a ger sum  = 14

diff = 20 och a = 4:    diff -= a ger diff = 16

prod = 30 och a = 4:    prod *= a ger prod = 120

div  = 42 och a = 4:    div  /= a ger div  = 10

mod  = 42 och a = 4:    mod  %= a ger mod  = 2

Slut på kapitel 4
```

För att förstå den sista raden i utskriften ovan och därmed rollen som variablerna **s** och **t** spelar i programmet **CompAssign**, måste vi få lite information om dessa variablers datatyp **string** och om **+** (konkatenering) som tillämpas på dem:

Datatypen `string`

string är en datatyp som representerar strängar, dvs text som består av ett antal sammanhängande tecken. **string** används för att deklarera *strängvariabler*. Även strängvariabler kan precis som andra variabler anta ”värden” som i så fall är *strängkonstanter*. I programmet **CompAssign** har **s** och **t** definierats till strängvariabler där **s** fått som värde strängkonstanten **"Slut på"** och **t** strängkonstanten **" kapitel 4"**. Som konstanter måste strängar omslutas med citationstecken. Som variabler kan de deklareras med bl.a. datatypen **string**. Använder man **<<** eller **>>** för att med **cout** skriva ut eller med **cin** läsa in **string**-variablers värden dvs text, måste man inkludera biblioteket **string**. Där definieras datatypen **string** inte längre som en enkel datatyp utan som en *klass* (sid 214). Klassen **string** är ett objektorienterat tillägg till språket och har en uppsjö av fördefinierade metoder och operatorer för stränghantering. En av dem är konkateneringsoperatorn **+** som används i programmet **CompAssign** och förklaras nedan.

Konkateneringsoperatorn `+`

Som det redan nämndes tidigare (sid 45) finns i C++ möjligheten att använda operatorn **+** för att konkatenera strängar, vilket dock endast gäller för sträng*variabler*. Man säger att operatorn **+** *överlagrar* den vanliga artitmetiska additionsoperatorn, dvs betyder inte längre addition av tal utan konkatenering av strängar. Det aktuella sammanhanget, närmare bestämt datatypen, avgör tolkningen. T.ex. kan man som i programmet **CompAssign** skriva:

```
string s = "Slut på", t = " kapitel 4";
s += t;          // Gör samma sak som s = s + t;
cout << s;
```

+ tolkas här inte som addition utan konkatenering därför att det till vänster och höger står en variabel av typ **string**. Man får då utskriften **Slut på kapitel 4**. För att **cout** och utmatningsoperatorn **<<** ska fungera med **string**-variabler, måste i början av programmet **CompAssign** stå: **#include <string>**. Kom ihåg att konkateneringsoperatorn **+** inte direkt kan användas på sträng*konstanter*: **cout << "Slut på" + " kapitel 3";** skulle ge kompileringsfel (sid 45). Anledningen är att **+** är definierad som konkateneringsoperator i klassen **string** och därför kan tolkas som sådan endast mellan två *objekt* av denna klass. Strängkonstanter däremot (som omslutas med citationstecken) är inga objekt av klassen **string** utan tolkas som *pekare*-till-**char** (sid 199). För pekare-till-**char** är operatorn **+** inte definierad.

Övningar till kapitel 4

4.1 Satsen **`cout << a;`** ger kompileringsfel till skillnad från **`cout << 'a';`** Sätt in båda i ett C++ program och testa. Ger även **`cout << 6;`** kompileringsfel? Testa vilka utskrifter följande satser ger:

```
cout << 6 << 6;
cout << '6' << '6' ;
cout << (6 + 6);
cout << ('6' + '6');
cout << 6.6 << 6.6 ;
cout << (6.6 + 6.6);
cout << "6.6" << "6.6";
```

Skriv om koden så att du får samma utskrift med en enda utskriftssats i koden. Lägg till och ta bort mellanslag, radbyte och tabulator på lämpliga ställen för att få en snygg utskrift. Förklara resultaten.

4.2 Komplettera programmet **`Variable`** (sid 55) så här: Deklarera ytterligare variabler, säg **`diff`**, **`prod`**, **`div`**, **`mod`**, tilldela till dem uttryck (sid 64) bildade med de andra räknesätten **`-`**, **`*`**, **`/`** och **`%`**. Skriv ut resultaten med meningsfulla utskrifter.

4.3 Vidareutveckla din lösning till övn 4.2 genom att ersätta den hårdkodade initieringen av variablerna **`number1`** och **`number2`** med en initiering genom *inläsning* som t.ex. kan göras med en **`cin`**-sats samt ledtext.

4.4 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.`** Utveckla programmet vidare med ytterligare räkneoperationer, kanske så småningom till en liten kalkylator.

4.5 Ersätt i programmet **`DefInit`** (sid 57) de två satser som deklarerar och initierar variablerna **`number1`**, **`number2`** med **`int number1 = number2 = 2;`** Förklara kompileringsfelet som uppstår då. Åtgärda felet med dubbelinitieringen av **`number1`** och **`number2`** i behåll.

4.6 Modifiera programmet **`OverWrite`** (sid 60) så att variabeln **`x`**:s gamla värde skrivs ut, medan dess nya ökade värde visas senare. Ersätt satsen **`x = x + 1;`** med **`x++;`** Blir det samma resultat om du ersätter den med **`x + 1;`** istället? Förklara!

4.7 Skriv ett program som frågar efter aktuell veckodag (t.ex. idag). Man ska mata in en siffra där veckodagarna numreras stigande från 1 med början på dag så att söndag blir den 7:e veckodagen. Sedan ska programmet fråga, typ ”När vill du träffa din kompis?” och begära som svar ett antal dagar.

Programmet ska beräkna och skriva ut den planerade träffens veckodag, dvs vilken veckodag det blir om man lägger till antalet dagar till aktuell veckodag.

4.8 Skriv ett program som omvandlar tiden i antal år, månader och veckor till antal dagar. Läs in tre heltal till antal år, månader och veckor. Beräkna och skriv ut sedan användarvänligt hur många dagar det blir totalt. Använd strukturen inmatning – bearbetning – utmatning enligt programmet **`Hour2-Sec`** (sid 64).

4.9 Vänd på problemet från övn 4.8: Skriv ett program som läser in ett antal dagar, omvandlar det till antal år, månader, veckor samt resterande dagar och skriver ut resultatet. Använd för denna omvandling följande algoritm och pseudokod.

Algoritmen:

1. Kalla den givna tiden i dagar för totaldagar.
2. Dividera totaldagar med 365 och strunta i resten, så får du det sökta antalet år.
3. Ta resten vid divisionen ovan. Dividera denna rest med 30 och strunta i resten så får du det sökta antalet månader.
4. Ta resten vid divisionen i punkt 3. Dividera denna rest med 7 och strunta i resten så får du det sökta antalet veckor.
5. Resten vid divisionen i punkt 4 är det sökta antalet resterande dagar.

Operationen ”Dividera och strunta i resten” är heltalsdivision och operationen ”Ta resten vid heltalsdivision” är modulo..

Pseudokoden:

```
år                = totaldagar heltalsdividerad med 365
månader           = (totaldagar modulo 365) heltalsdividerad med 30
veckor            = ((totaldagar modulo 365) modulo 30) heltalsdividerad 7
Resterande dagar  = ((totaldagar modulo 365) modulo 30) modulo 7
```

4.10 Tillämpa den logiska strukturen i algoritmen och pseudokoden till övn 4.9 för att lösa följande uppgift:

Efter inköp av en vara i en automat ska växeln ges tillbaka i form av ett antal föreskrivna myntslag: 10-kronor, 5-kronor, 1-kronor, 50-öringar [*] och en rest i ören < 50. Skriv ett program som läser in ett växelbelopp i ören, omvandlar det till ett antal 10-kronor, 5-kronor, 1-kronor och 50-öringar samt skriver ut resultatet. Resten i ören < 50 kan vi försumma (resp. avrunda).

---

[*] 50-öringen finns inte längre i det svenska myntsystemet. Att vi ändå inkluderar den i uppgiften beror inte på nostalgi utan på internationalisering. Vi vill hålla öppen möjligheten för en övergång till andra valutor, t.ex. Euro. Behandlingen av en halv enhet vid omvandling av växelbeloppet till automatens tillåtna mynt inkluderar en programmeringsteknisk finess som kan vara värd att lära sig. Så kan våra program även användas t.ex. för Euron där 50 Cent ersätter 50-öringen.

Kapitel 5

Enkla

datatyper

Ämne	Sida	Program
5.1 Kan datorn lagra hur stora tal som helst?	75	**Primitives**
- Operatorn sizeof	76	
- unsigned-typerna	80	
- Overflow	78	**Limits**
5.2 Datatypen **char**	79	**Char**
5.3 Explicit typkonvertering	81	**Char2int**
	82	**Int2char**
5.4 ASCII-tabellen	83	**Ascii**
5.5 Escapesekvenser	86	**Escape**
Övningar till kapitel 5	88	

5.1 Kan datorn lagra hur stora tal som helst?

Man har ju hört talas om datorernas obegränsade möjligheter. Deras kraft och kapacitet växer med teknikens snabba utveckling. Men hur avancerade de än blir kommer de alltid att ha ett *begränsat* utrymme för lagring av data. Därför har man i alla programmeringsspråk vissa fördefinierade datatyper för att bl.a. ekonomisera och effektivisera minneshanteringen när ett program körs. Datatyperna **`int`** och **`string`** har vi redan stiftat bekantskap med.

Det finns olika kategorier av datatyper: *Enkla*, *sammansatta* datatyper och *objekt*. I detta kapitel ska vi studera kategorin *enkla datatyper* i C++ (eng. *primitive types*).

> En **enkel datatyp** är en datatyp som representerar endast *ett* värde åt gången, dvs *ett* heltal, *ett* decimaltal, *ett* tecken eller *ett* sanningsvärde.

`int` är en enkel datatyp därför att endast *ett* heltal kan lagras i en variabel av typ **`int`**. I C++ finns ett antal fördefinierade datatyper av kategorin *enkel*: **`bool`**, **`char`**, **`short`**, **`int`**, **`long`**, **`float`**, double och **`long double`**. De kallas *enkla* datatyper därför att de representerar endast *ett* värde, t.ex. *ett* heltal, *ett* decimaltal, *ett* tecken eller *ett* sanningsvärde. De kan inte lagra mer invecklade data som *objekt* eller *sammanatta* data med *flera* tecken. Dessa kommer att behandlas senare.

I punkt 2 av definitionen för *datatyp* (sid 51) sa vi bl.a.:

> 2. hur mycket minne denna typ av data tar och därmed med hur stora värden den kan lagra (det tillåtna värdeområdet)

Det tilldelade minnesutrymmet är förbestämt i datatypens definition. De enkla datatypernas fastlagda minnesstorlekar kan man se i följande utskrift som är producerad av programmet **`Primitives`** på nästa sida:

```
Datatypen:
bool          tar 1
char          tar 1
short         tar 2
int           tar 4
long          tar 4
float         tar 4
double        tar 8
long double tar  8 bytes.

Även uttryck kan skickas som operand till sizeof:
Uttrycket a + b tar 4 bytes när a och b är int.
```

`bool` representerar sanningsvärdena sant eller falskt. **`char`** lagrar tecken. **`short`**,

int och **long** är enkla datatyper för representation av heltal. De enkla datatyperna **float**, **double** och **long double** representerar decimaltal. Alla enkla datatyper hittar man i tabellen över reserverade ord. I denna tabell finns också det reserverade ordet **sizeof** som används för att mäta minnesstorleken av varje datatyp i antal bytes. 1 byte består av 8 bitar där 1 bit är den minnesatom som kan lagra endast en nolla eller en etta. Som man ser har vi ordnat de enkla datatyperna efter det minnesutrymme som är tilldelat och förbestämt i deras definition. Det tillåtna värdeområdet ligger inom ett intervall som direkt kan härledas från minnesstorleken som varje datatyp har till förfogande.

I programmet **Primitives** visas hur man mäter minnesstorleken till datatyper:

```
// Primitives.cpp
// De enkla datatypernas minnesstorlek mäts med sizeof
#include <iostream>
using namespace std;

int main()
{
   cout << "\n\tDatatypen:\n\t"
        << "bool         tar " << sizeof(bool)  << "\n\t"
        << "char         tar " << sizeof(char)  << "\n\t"
        << "short        tar " << sizeof(short) << "\n\t"
        << "int          tar " << sizeof(int)   << "\n\t"
        << "long         tar " << sizeof(long)  << "\n\t"
        << "float        tar " << sizeof(float) << "\n\t"
        << "double       tar " << sizeof(double)<< "\n\t"
        << "long double tar  " << sizeof(long double)
        << " bytes.\n\n\t";
   int a, b;
   cout << "Även uttryck kan skickas som operand till"
        << " sizeof:\n\tUttrycket a + b tar " << sizeof(a+b)
        << " bytes när a och b är int.\n";
}
```

Det är anmärkningsvärt att variablerna **a** och **b är** deklarerade, men inte initierade. Vi kan både kompilera och exekvera, eftersom vi inte använder deras värde.

Vi bildar uttrycket **a + b** och skickar det som operand till **sizeof** för att mäta uttryckets minnesstorlek. **a** och **b** är **int**s, därför blir även **a + b** en **int** som tar **4** bytes.

Operatorn `sizeof`

sizeof är en operator i C++ som endast har *en* operand och mäter storleken på operandens minnesutrymme. Begreppet *operator* används här som synonym till funktion, vilket kan igenkännas vid de parenteser som följer när man som i programmet ovan skriver **sizeof(int)**. När *operator* används istället för *funktion* brukar man utelämna parenteserna. **sizeof(int)** mäter storleken av minnesutrymmet som är definierad för datatypen **int**. Begreppet *operand* används som sy-

nonym till parameter, dvs det man skriver i parentesen. T.ex. har **sizeof(int)** operanden **int**. Närmare bestämt returnerar **sizeof** antalet bytes operanden tar i minnesutrymme. Operanden kan vara en datatyp. Generellt:

sizeof(*datatyp***)**

Det finns även möjligheten att skicka konstanter, variabler, ja t.o.m. aritmetiska uttryck som operander till **sizeof** för att mäta deras minnesstorlek

Heltalstypernas gränser

De enkla datatypernas gränser som vi egentligen är ute efter i detta avsnitt, kan nu lätt härledas från deras minnesstorlekar. Ett exempel är heltalsdatatypen **short** som enligt ovan har **2** bytes dvs 2x8 = 16 bitar till förfogande. Därför reserverar varje variabel definierad som **short** 16 bitar i minnesutrymme. Ett värde till en sådan variabel kan alltså inte lagras i datorn om det överstiger det största binära tal som kan lagras i 16 – 1 = 15 bitar. 15 därför att en bit behövs för att lagra själva tecknet + eller – därför att en **short**-variabel kan även anta negativa värden. Det största binära heltal som kan lagras i 15 bitar består av 15 ettor dvs 111 1111 1111 1111. I det decimala talsystemet blir det 32 767. Därför är den positiva gränsen för datatypen **short** 32 767. På samma sätt kan de andra datatypernas gränser härledas från deras resp. minnesutrymme. Ingen panik! Vi kommer inte att göra det. Dessa gränser är lagrade i vissa namngivna konstanter. Här skrivs ut dem för alla enkla datatyper som ett körresultat av programmet **LimitsInt** på nästa sida:

```
char          finns mellan -128         och   127
short                      -32768             32767
int                        -2147483648        2147483647
long                       -2147483648        2147483647
unsigned char                   0             255
unsigned short                  0             65535
unsigned int                    0             4294967295
unsigned long                   0             4294967295
```

Till skillnad från de andra datatyper som kan anta både positiva och negativa värden, kan de *teckenlösa* datatyperna (**u** = **unsigned** dvs utan tecken **+** eller **-**) endast anta positiva värden: De heter så därför att deras värden varken behöver ha plus- eller minustecknet framför talet. Dessa enkla datatyper har precis lika mycket minnesutrymme till förfogande som sina motsvarande vanliga datatyper med tecken. Detta innebär att nödvändigheten att lagra tecknet faller bort hos **unsigned**-typerna. Om vi resonerar vidare i exemplet med **short** skulle datatypen **unsigned-short** ha alla 16 bitar till förfogande för själva positiva heltalet.

Det största binära heltal som kan lagras i 16 bitar består av 16 ettor dvs 1111 1111 1111 1111. I det decimala talsystemet blir detta 65 535. Därför är gränsen för datatypen **unsigned short** dubbelt så stort (fast +1 pga nollan) som för **short**. Och så är det med alla **unsigned**-typer: deras gränser är dubbelt så stora fast de har li-

ka stort minnesutrymme till förfogande, därför att de inte behöver lagra tecknet och därmed har 1 bit mer för att lagra själva positiva heltalet.

```
// LimitsInt.cpp
// Fördefinierade konstanter visar heltalstypernas gränser
#include <iostream>
using namespace std;

int main()
{
   cout << "\n\tchar     finns mellan " << CHAR_MIN
                << "        och    " << CHAR_MAX << "\n\t"
        << "short                  " << SHRT_MIN
                  << "            " << SHRT_MAX << "\n\t"
        << "int                    " << INT_MIN
                       << "       " << INT_MAX  << "\n\t"
        << "long                   " << LONG_MIN
                       << "       " << LONG_MAX << "\n\t"
        << "unsigned char              0           "
                                     << UCHAR_MAX << "\n\t"
        << "unsigned short             0           "
                                     << USHRT_MAX << "\n\t"
        << "unsigned int               0           "
                                     << UINT_MAX  << "\n\t"
        << "unsigned long              0           "
                                     << ULONG_MAX << "\n";
}
```

Programmet **`LimitsInt`** har fördelen att låta datorn göra jobbet genom att lista upp heltalsdatatypernas gränser. Värdena hämtas från lagrade konstanter som är definierade i klasser som är identiska med C++:s enkla datatyper.

I C++ är det en konvention att beteckna konstanter med stora bokstäver. Så, alla namn med stora bokstäver i programmet ovan är fördefinierade konstanter. Som heltalstypernas min- och max-gränser baseras deras beräkning på datatypernas minnesstorlekar som mättes med **`sizeof`** i programmet **`Primitives`**.

Overflow

Vad händer nu om man överskrider de ovan angivna gränser dvs om man tilldelar ett värde till en variabel som överskrider det maximalt tillåtna värdet för datatypen? Fenomenet kallas *overflow*. Ja, vad händer om man försöker att hälla mer vatten i ett glas än det ryms? I vissa miljöer blir det exekveringsfel och programkrasch. I C++ fortsätter programmet: Det överskridna värdet ”slår runt” och hamnar på andra ändan av det tillåtna talområdet.

Overflow innebär förlust eller förfalskning av information. Beroende på datatypen kan det bli felaktigt resultat samt följdfel som är svårt att spåra, om man räknar vidare utan att upptäcka felet. Det enda sättet att undvika overflow är att utveckla en medvetenhet om fenomenet overflow och känna till när det kan inträffa.

5.2 Datatypen `char`

char uttalas ”kar” därför att det står för *character*, dvs tecken på engelska.

> **char** är en enkel datatyp i C++ som representerar ett tecken och används för att deklarera teckenvariabler.

Men hur lagras tecken i datorn? All data, även tecken, måste ju till slut omvandlas till ettor och nollor. Därför omvandlas alla tecken till heltal enligt ett visst kodsystem. Varje tecken har sin heltalskod. Det är dessa koder som lagras som ettor och nollor. Bokstaven **a** t.ex. har koden 97 som är 1100001 binärt. När du t.ex. trycker på tangenten **a** överförs denna sekvens av ettor och nollor till datorn. Datatypen **char** representerar även dessa heltalskoder. Därför kan även dessa heltalskoder direkt tilldelas variabler av typ **char**. Närmare bestämt är alltså **char** en datatyp som representerar *både* tecken *och* sådana heltal som är koder till tecken. Nu ska vi i följande program lära känna datorns s.k. *teckenuppsättning* genom att läsa in ett tecken och skriva ut dess heltalskod utan explicit typkonvertering:

```
// Char.cpp
// Ger koden till ett inmatat tecken
// Representation av tecken med datatypen char
#include <iostream>
using namespace std;

int main()
{
   char letter;           // Deklaration till datatypen char
   int  code;
   cout << "\nMata in ett tecken och tryck på Enter:    ";
   cin >> letter;         // läses in som char, lagras som
                          // heltalskod och omvandlas automa-
   code = letter;         // tiskt till int via tilldelning
   cout << "\nDet inmatade tecknet är " << letter <<
           " och har koden   " << code << "\n\n";
}
```

Vi har en variabel av typ **char** kallad **letter** och en variabel av typ **int** kallad **code**. Vid körningen matar vi in t.ex. bokstaven **a**. **cin**-satsen läser in **a**, omvandlar det till koden **97** och lagrar *denna kod* i variabeln **letter**. Sedan tilldelas detta värde variabeln **code**. Nu finns i båda variablerna värdet **97**. Men när vi kör får vi:

```
Mata in ett tecken och tryck på Enter:     a

Det inmatade tecknet är a och har koden    97
```

cout-satsen skriver ut variabeln **code**:s värde som är **97**, vilket är förväntat. Men

vad som kanske förvånar är att inte variabeln **letter**:s lagrade värde som också är **97**, skrivs ut, utan att det blir *tecknet* **a** istället.

cout gör samma sak som **cin**, fast tvärtom: Det tar värdet **97** från minnescellen **letter**, omvandlar det enligt rådande teckenuppsättning till tecknet **a** och skriver ut det på skärmen. Att inte talet **97** skrivs ut, beror på att variabeln **letter**:s datatyp är **char**. Det är datatypen som styr både **cin**- och **cout**-satsens agerande. De är förprogrammerade så att de behandlar all data enligt definierad datatyp. Även *konstanter* som inte deklareras tolkas enligt sina resp. datatyper (sid 53).

`unsigned` - *typerna*

Till skillnad från de datatyperna ovan som kan anta både positiva och negativa värden kan de *förteckenlösa* datatyperna endast anta positiva (icke-negativa) värden:

```
unsigned char          unsigned int
unsigned short         unsigned long
```

De heter så därför att deras värden varken behöver ha plus- eller minustecknet framför talet. Dessa enkla datatyper har precis lika mycket minnesutrymme till förfogande som sina motsvarande vanliga datatyper med förtecken. Detta innebär att nödvändigheten att lagra förtecknet faller bort hos **unsigned**-typerna. Dvs om vi resonerar vidare i exemplet med **short** skulle datatypen **unsigned short** ha alla 16 bitar till förfogande för själva positiva heltalet. Det största binära heltal som kan lagras i 16 bitar består av 16 ettor dvs 1111 1111 1111 1111. I det decimala talsystemet blir detta 65 535. Därför är gränsen för datatypen **unsigned short** dubbelt så stort som för datatypen **short**. Och så är det med alla **unsigned**-typer: deras gränser är dubbelt så stora fast de har lika stort minnesutrymme till förfogande, beroende på att de inte behöver lagra förtecknet och därmed har 1 bit mer för att lagra själva positiva heltalet.

5.3 Explicit typkonvertering

En fråga inställer sig: Kan vi inte själva göra de omvandlingar mellan tecken och heltal, mellan datatyperna **char** och **int**, som **cin**- och **cout**-satserna automatiskt gjorde i programmet **Char**, när vi vill med egen kod? Jo, det är möjligt med en teknik som tas upp i detta avsnitt och som kallas *explicit typkonvertering*.

Explicit betyder uttrycklig och innebär här att man själv omvandlar datatypen. Programmet nedan använder explicit typkonvertering för att omvandla tecken till heltal och kan på så sätt få fram koden tillhörande ett tecken som matas in:

```
// Char2int.cpp
// Ger koden till ett tecken: Explicit typkonvertering
#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';
           // I raden ovan görs explicit typkonvertering:
           // vi omvandlar variabeln letters värde till int
}
```

Explicit typkonvertering kodas genom att sätta datatypen **int** inom parentes och placera den framför variabelnamnet **letter**: **(int) letter**

Det som omvandlas är variabelns *värde* till **int** när själva variabeln är deklarerad till datatypen **unsigned char**. Valet av **unsigned** ska garantera att vi alltid får positiva koder, aldrig negativa. Om man vid körning matar in **a**, lagras tecknet **a** som sin tillhörande kod **97**. Variabeln **letter** fortsätter att vara av typ **unsigned char** i programmet. Explicit typkonvertering kan alltså inte förändra variabelns deklaration, utan endast dess värde vid den aktuella konverteringen.

Som resultat får man samma utskrift som programmet **Char**, men vi har förenklat koden genom att arbeta med endast *en* variabel. Dessutom har vi lärt oss en metod för omvandling av datatyper som vi kommer att använda i fortsättningen. Generellt kan programkoden för explicit typkonvertering i C++ beskrivas så här:

(datatyp) *uttryck*

där *uttryck* är en kombination av variabler, konstanter, operatorer och vanliga parenteser som i specialfall även kan bestå av en enda variabel eller en enda konstant (sid 64). Dessutom finns – som en kvarleva från C – möjligheten att sätta paren-

teserna kring variabeln eller konstanten istället för kring datatypen. Man kan alltså skriva **`int (letter)`** istället för **`(int) letter`**. Vi använder inte denna variant.

Programmet **`Char`** gav oss koden när vi matade in ett tecken. Nästa programexempel löser det omvända problemet att ge oss tecknet när vi matar in en kod.

```
// Int2char.cpp
// Ger tecknet till en inmatad kod
// Representation av tecken med heltalskoder
#include <iostream>
using namespace std;

int main()
{
   int code;
   cout << "\nMata in heltal (33-126) och tryck på Enter : ";
   cin >> code;
   cout << "\nDet inmatade heltalet är " << code <<
        " och är koden till tecknet " << (char) code << "\n";
}
```

Även här använder vi explicit typkonvertering: **`(char) code`**

som omvandlar variabeln **`code`**:s värde vars datatyp enligt deklaration är **`int`** till datatypen **`char`**. Därför får vi i utskriften bokstaven **`a`**, om vi vid en körning matar in **`97`**. Anledningen till att vi i ledtexten skriver **`(33-126)`** – vilket innebär att man ska mata in heltal mellan 33 och 126 – är att man utanför detta intervall antingen får oinitierade tecken eller inga tecken alls utskrivna. Orsaken är att det finns tecken som är icke-skriv- och icke-läsbara. Det finns även styr- och kontrolltecken som är kombinationer av två eller fler tangenttryckningar vilka inte heller kan visas på skärmen. I följande körexempel har vi hållit oss till det rekommenderade kodintervallet:

```
Mata in heltal (33-126) och tryck på Enter :    76

Det inmatade heltalet är 76 och är koden till tecknet L
```

En fördjupad insikt i datorns hantering av tecken får vi först när vi konkretiserar vad vi hittills lite mystiskt har kallat koder. Vilka koder är det egentligen? Det finns olika kodsystem. Man talar också om en dators eller om ett programs *teckenuppsättning*. Den vanligaste teckenuppsättningen är *ASCII-tabellen* som vi tar upp i nästa avsnitt.

5.4 ASCII - tabellen

ASCII (uttalas ”aski”) står för **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange och är en standard för kodning av tecken skapad av det amerikanska standardiseringsorganet ANSI. Innan man hann sätta igång något standardiseringsarbete på internationell nivå, hade ASCII redan erövrat världen. Så idag är ASCII en de facto-standard i hela världen och används på alla persondatorer, både PC och Mac.

Teckenstandarden ASCII omfattar alla engelska bokstäver, siffrorna 0-9, de vanligaste specialtecknen och en del styr- och kontrolltecken. ASCII använder sig av s.k. 7-bitars kodning vilket innebär att heltalskoden till ett tecken placeras som ettor och nollor i 7 bitar av en byte. Den lediga åttonde biten används för felkontroll och kan därför inte utnyttjas för representation av data. Exempel: Tecknet **a**:s ASCII-kod är **97** vilket omvandlat till ettor och nollor blir 1100001*, dvs ett binärt tal som är 7 bitar långt. För att få reda på vilka tal man kan uttrycka med 7 bitar, kan man titta på det minst möjliga talet – det är 7 nollor 0000000 som är **0** – och det störst möjliga – det är 7 ettor 1111111 som är **127** (kontrollera med kalkylatorn). Därför består ASCII-koderna av heltalen mellan **0** och **127** och är definierade enligt följande:

	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		

Siffrorna till vänster i tabellens 1:a kolumn anger de två första siffrorna i ASCII-koden och siffran högt upp i tabellens 1:a rad anger den sista siffran i ASCII-koden. Söker man ASCII-koden till t.ex. **t** går man på samma rad längst till vänster och hittar **11**, går sedan från **t** i samma kolumn högst upp och hittar **6**. Alltså har **t** ASCII-koden **116**. Nu förstår du varför vi i **Int2char** (sid 82) i ledtexten gav uppma-

* För att verifiera omvandlingen till binärt tal kan du t.ex. starta Kalkylatorn med graffunktion i Windows. Välj inställningen Programmerare för att kunna omvandla tal till och från olika talsystem: **Hex**adecimal med basen 16, siffrorna 0-9 och A-F, **Dec**imal med basen 10, siffrorna 0-9, **Oct**al med basen 8, siffrorna 0-7, **Bin**är med basen 2, siffrorna 0 och 1.

ningen att mata in heltal mellan **33** och **126**. ASCII-tabellen visar att endast dessa är skriv- och läsbara. De andra består av icke-skriv- och icke-läsbara styr- och kontrolltecken eller ”vita” tecken. Dessa specialtecken har symboliska namn. T.ex. kallas mellanslaget för sp = space med ASCII-koden 32 och radbyte för nl = new-line med ASCII-koden 10 (en äldre beteckning är lf = line feed).

ASCII-tabellen visar också denna teckenstandards begränsningar. Det största binära heltal man kan placera i 8 bitar är 8 ettor 11111111 som ger **255**. Det har man gjort i andra teckenuppsättningar och skapat koder mellan **128** och **255**. Denna utvidgning är däremot inte en standard, inte ens ”de facto”. Ändå pratar man ofta lite slött om ”ASCII”-koder. I själva verket tillämpas i kodintervallet **128–255** olika teckenuppsättningar inte bara i olika datorer, utan t.o.m. i olika program på samma dator, t.ex. Windows och Kommandotolken. Idag används i de flesta moderna miljöer teckenstandarden *Unicode* för utvidgningen av ASCII-tabellen. C++ däremot ”förvränger” de svenska tecknen å, ä, ö, Å, Ä, Ö när de skrivs ut till konsolen. Förklaringen är att de skriver till konsolen med en annan teckenuppsättning än Unicode. Däremot visar t.ex. C++ de svenska tecknen korrekt även i konsolen.

Vi avslutar detta avsnitt med ett programexempel som visar sambandet mellan tecken och deras ASCII-koder:

```
// Ascii.cpp
// Tilldelar tal till en char-variabel och räknar med den
// Det är ASCII-koder som lagras i variabler av typ char
// Demonstrerar datatypen chars dubbla roll som
// 1) tecken i cout-satsen och som
// 2) tal i aritmetiska uttryck: Teckenaritmetik
#include <iostream>
using namespace std;

int main()
{
   char letter = 89;          // 89 lagras i minnescellen let-
                              // ter, men deklarerat som char
   // char letter = 'Y';      // Ger samma resultat

   cout << "\n Tecknet " << letter << " har ASCII-koden "
        << (int) letter;
   letter++;                  // Ökar till nästa tecken: 90
   cout << "\n Tecknet " << letter << " har ASCII-koden "
        << (int) letter;

   int no = letter/2 + 36*10/letter - 1;   // Teckenaritmetik
             // 45   +      4       - 1   ger 48
   cout << "\n\n " << no << " är ASCII-koden till tecknet "
        << (char) no << "\n\n";
}
```

Man kan konstatera: När **char**-variabler skrivs ut tolkas de som tecken. När beräkning utförs tolkas **char**-variabler som tal dvs deras ASCII-koder. Datatypen **char**:s

dubbla roll är att representera *både* tecken och små heltal där med ”små” menas talområdet **0-255**, heltal som kan lagras i 1 byte dvs 8 bitar, den utvidgade ASCII-tabellens koder. I programexemplet ovan kommer denna dubbla roll fram när vi å ena sidan ökar **char**-variabeln **letter**:s värde med 1:

```
letter++;
```

Här behandlas teckenvariabeln som ett tal man kan räkna med. Samma sak är det när vi använder den i aritmetiska uttryck som

```
letter/2 + 36*10/letter - 1
```

Det är möjligt eftersom i minnescellen **letter** lagras ett tal. Å andra sidan behandlas **char**-variablerna i programmets **cout**-satser som tecken. Den förprogrammerade kodkomponenten **cout** utgår från variabeln **letter**:s deklaration och utför automatiskt de nödvändiga omvandlingarna mellan tecken och heltal. **cout** omvandlar till tecken vid utskrift *efter* att ha hämtat talen från minnescellerna. Om vi hade en **cin**-sats i programmet skulle **cin** omvandla tecken till heltal *innan* det lagrades i en minnescell. Som språkets gränssnitt mot programmeraren är **cout** och **cin** designade att agera användarvänligt.

En körning av **Ascii** ger följande utskrift som bekräftar det ovanstående:

```
Tecknet Y har ASCII-koden 89
Tecknet Z har ASCII-koden 90

48 är ASCII-koden till tecknet 0
```

Att vi får **48** beror på att uttrycket **letter/2 + 36*10/letter - 1** ger **48** när det beräknas. Då är nämligen värdet på **letter** efter ökningen med **letter++;** just **90** dvs **90/2 + 36*10/90 - 1** ger **48**.

5.5 Escapesekvenser

Följande frågor är fortfarande obesvarade:

1. Hur kommer vi åt de icke-skriv- och icke-läsbara styr- och kontrolltecknen dvs ASCII-koderna **0-32**?
2. Hur skriver vi i C++ kod de tecknen som har en symbolisk betydelse (t.ex. " eller ') inom en strängkonstant "..."?
3. Hur kan vi få de svenska specialtecknen å, å, ö, ... korrekt utskrivna i en DOS-miljö som t.ex. Kommandotolken?

Svaret på ovanstående frågor är *escapesekvenser*.

På svenska betyder *to escape* att fly. Escapesekvenser inleds med tecknet backslash **** åtföljt av *ett* tecken eller tecknets kod i ett visst format. Med **** vill man *fly* från tecknets vanliga betydelse och ge det en *annan* innebörd. Med **\n** t.ex. vill man undvika bokstaven **n** och åstadkomma radbyte istället. På samma sätt fungerar andra escapesekvenser vars innebörd demonstreras i följande program:

```
// Escape.cpp
// Visar vad escapesekvenserna \n, \r, \t, \b, \a, ... gör
// och ger deras ASCII-koder. Här är \n aktiverad.
// För att testa de andra escapesekvenserna kommentera bort
// raden med \n och aktivera EN av de andra raderna.
#include <iostream>
using namespace std;

int main()
{
   char letter;
   letter = '\n';       // newline, radbyte, markören flyttar
                        // till början av NÄSTA rad
   // letter = '\r';    // markören till början av SAMMA rad
   // letter = '\t';    // tabulator
   // letter = '\"';    // tecknet "
   // letter = '\a';    // datorljud
   // letter = '\\';    // tecknet backslash (\)
   // letter = '\x5C';  // tecknet backslash (\)
   // letter = '\x84';  // tecknet ä
   // letter = '\x86';  // tecknet å
   // letter = '\x94';  // tecknet ö
   cout << "\nTecknet \x84r " << letter <<
           "och har koden \"" << (int) letter << "\"\n\n";
}
```

När man kör programmet ser man skillnaden mellan **\n** som i ASCII-tabellen har beteckningen nl (newline) och **\r** som heter där cr (carriage return dvs vagnretur).

Det finns även andra escapesekvenser vars innebörd kan testas genom att sätta in dem i programmet ovan som t.ex.: **\b** , **\?** , **\'** , **\0**, **\f** , **\v** , ...

Escapesekvensen **\"** kan användas för att skriva ut ett citationstecken. Det är ju inte möjligt på vanligt sätt eftersom själva tecknet **"** redan är reserverat för att avgränsa en sträng, se sid 37. Kompilatorn tolkar alltid **"** som en inledande eller avslutande strängavgränsare. Räddaren i nöden är escapesekvensen **\"** som vi använde i programmet **Escape**:s **cout**-sats för att skriva ut koden (variabeln **letter**:s värde omvandlat till heltal) inom citationstecken. Observera att **\"** måste bäddas in i en strängkonstant – som i **Escape** – eller behandlas som enskilt tecken och får dessutom inte blandas med variabler. Escapesekvenser är nämligen symboler för teckenkonstanter. De skiljer sig från teckenkonstanters vanliga skrivsätt i och med att de inleds med symbolen **** .

Nu kommer vi till den lite mystiska raden

```
letter = '\x5C';      // tecknet \
```

som så småningom kommer att besvara fråga 3 som vi ställde på förra sidan. Vad är **'\x5C'** ? Eftersom det är omgärdat av apostrofer måste det vara en teckenkonstant och eftersom det inleds med **** måste det vara en escapesekvens. Kvarstår att tolka **x5C**. Använd t.ex. Kalkylatorn i Windows. Välj i menyn Visa undermenyn Avancerad. Markera Hex och mata in **5C**. Markera sedan Dec. Displayen kommer då att visa **92**. Du har just omvandlat det *hexadecimala* talet **5C** (tal representerat i ett talsystem med basen 16) till det *decimala* talet **92** (tal representerat i det vanliga talsystemet med basen 10). För att skilja de båda talsystemen åt inleds alla hexadecimala tal med **x**. Med **\x5C** menas alltså tecknet med ASCII-koden **5C** i hexadecimalt format och **92** i decimalt format. Enligt ASCII-tabellen är tecknet till koden **92** backslash **** . Man kan faktiskt skriva alla tecken som escapesekvenser, bara man efter **** anger tecknets kod i hexadecimalt format. Testa gärna andra exempel med programmet **Escape**.

Detta öppnar vägen till att bli av med de konstiga tecken som skrivs ut istället för ä, å, ö, vilket händer när vi kör C++ program som *console application* i Kommandotolken, och programkoden är editerad i Windowsmiljö, t.ex. i Visual Studio. Kommandotolken (DOS) har en annan teckenuppsättning än Windows utanför ASCII-standarden. Med programmet **Char2int** (sid 81) kan vi få reda på de svenska specialtecknens DOS-koder när vi exekverar i Kommandotolken. Sedan kan vi med hjälp av en kalkylator ta reda på kodernas *hexadecimala* format. Det blir:

Tecknet	Dec. kod	Hex. kod	Escapesekvens
ä	**132**	**84**	**\x84**
å	**134**	**86**	**\x86**
ö	**148**	**94**	**\x94**

Slutligen bäddas escapesekvensen in i vanlig text, t.ex. **\x84r** för **är**, för att få korrekta svenska tecken. Detta har vi gjort i **cout**-satsen i programmet **Escape**.

Övningar till kapitel 5

5.1 Skriv ett program som läser in tre tecken och skriver ut dem i omvänd ordning.

5.2 Skriv ett program som läser in en gemen (liten bokstav) och skriver ut dess versal (stor bokstav) och sedan läser in en versal och skriver ut dess gemen.

5.3 Experimentera med programmet **`Int2char`** (sid 82) för att få reda på ASCII-koden till datorns ljudsignal. Skriv ett program som genererar ljudsignal.

5.4 Kryptering av tecken: Skriv ett program som läser in ett tecken och förskjuter det i teckentabellen med ett visst antal steg (krypteringsnyckeln). Skriv ut både det inlästa och det förskjutna tecknet på ett användarvänligt sätt. Börja med att hårdkoda krypteringsnyckeln och fortsätt med att läsa in den.

5.5 Kryptering av ord: Skriv ett program som läser in fem tecken och skriver ut dem förskjutna med *ett* steg i ASCII-tabellen så att t.ex. inmatningen **`Kalle`** ger utskriften **`Lbmmf`**. Återställ och skriv ut det krypterade ordet. Vidareutveckla programmet genom att utöka och läsa in antalet steg som en krypteringsnyckel.

5.6 Ta fram din lösning till övn 5.4 (annars se facit **`Ovn_5_4`** sid 240), ändra datatypen till variabeln **`step`** (krypteringsnyckeln) från **`int`** till **`char`**. Undersök vad som händer när du testkör programmet med denna ändring. Förklara resultatet.

5.7 Skriv ett program som skriver ut en tabell med fyra kolumner över alla escapesekvenser du känner till. Tabellens första kolumn ska visa själva escapesekvensen i C++ kod. Den andra kolumnen ska visa tecknets ASCII-kod. I den tredje kolumnen ska stå tecknet själv eller, om det är oskrivbart, tecknets betydelse. Den fjärde kolumnen ska visa hur de skrivbara tecknen skrivs i konsolen direkt, inte med escapesekvens. Om de är oskrivbara ska stå där oskrivbart. Även de svenska specialtecknen **`ä`**, **`å`**, **`ö`**, **`Ä`**, **`Å`** och **`Ö`** ska vara med i tabellen samt någon vanlig bokstav, säg **`a`**. Med hjälp av programmen **`Char2int`** (sid 81) och **`Escape`** (sid 86) kan du ta reda på de svenska specialtecknens ASCII-koder och escapesekvenser.

5.8 *Inlämningsuppgift LGT*

Skriv ett program som producerar följande utskrift:

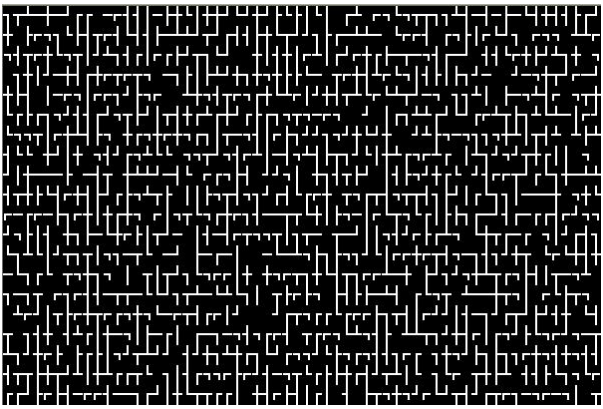


Dessa tecken finns i den utvidgade delen av teckentabellen och används för att rita raka linjer, ramar, tabeller, skisser osv i en textbaserad miljö *(text mode)*. De kallas för *Linjegrafiktecken* ***(LGT)***. Ta reda på koderna till tecknen ovan genom att experimentera med programmet **`Char2int`**. Även ett studium av programmen **`Char`**, **`Int2char`** och **`Ascii`** samt lösning av övningarna på förra sidan kan bidra till att lösa detta detektivarbete.

Denna övning är tänkt för att förbereda dig på projektetuppgiften ***Labyrint*** som kommer att introduceras i nästa kapitel. De koder du hittar här, för att ”rita” ***LGT*** (figuren ovan), kommer du att behöva i projektuppgiften.

I projektuppgiften ***Labyrint*** ska du rita en labyrintartad figur som med hjälp av en loop är slumpmässigt sammansatt av just de ***LGT*** deras koder du hittat här, plus mellanslaget. En algoritm följer med som handledning (övn 6.11 sid 124). Loopar och hantering av slumptal tas upp i nästa kapitel.

Är du nyfiken på ***Labyrinten*** kan du se här ett exempel på en köning. Exempel, därför att varje körning ger p.g.a. slumpen en lite annorlunda figur. Kolla gärna om figuren är sammansatt av ***LGT*** plus mellanslaget:



Kapitel 6

Kontrollstrukturer

Ämne	Sida	Program
6.1 Vad är kontrollstrukturer?	90	
6.2 Enkel selektion: **`if`**-satsen	92	**`SimpleIf`**
- Flera satser i **`if`**-satsen	95	
- Algoritm för platsbyte	95	**`MiniSort`**
6.3 Tvåvägsval: **`if`**-else-satsen	98	**`IfElse`**
6.4 Flervägsval	100	
- **`if-else`**-stegen	101	**`GissaTal_1`**
- **`switch`**-satsen	100	**`Switch`**
6.5 Efter-testad repetition: **`do`**-satsen	105	**`GissaTal_2`**
6.6 För-testad repetition: **`while`**-satsen	108	**`While`**
- Evighetsslinga	109	
6.7 Bestämd repetition: **`for`**-satsen	110	
- for vs. while	111	**`While->For`**
- ASCII-tabellen med **`for`**	113	**`AsciiFor`**
6.8 Nästlade **`for`**-satser	116	**`Stars`**
- Hantering av slumptal	117	**`NestedFor`**
- Multiplikationstabellen	120	**`MultipTab`**
Övningar till kapitel 6 (Projekt *Labyrint I*)	123	

6.1 Vad är kontrollstrukturer?

Kontrollstrukturer är algoritmers byggstenar och programmeringens mest grundläggande verktyg. Det finns generella strukturer i alla algoritmer som är oberoende av det aktuella problemet. Därför kan de användas som byggstenar vid beskrivning av *alla* algoritmer som i sin tur ligger till grund för alla datorprogram, oberoende av programmeringsspråk.

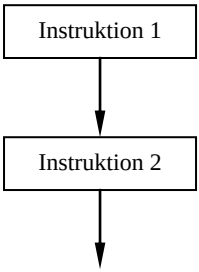
Kontrollstrukturer består av tre grundläggande typer:

- **Sekvens (följd)**
- **Selektion (val)**
  - Enkel selektion
  - Tvåvägsval
  - Flervägsval
- **Repetition (upprepning)**
  - Förtestad repetition
  - Eftertestad repetition
  - Bestämd repetition

Alla datorprogram är kombinationer av dessa tre typer av kontrollstrukturer. I detta kapitel ska vi gå igenom alla tre och lära oss hur de kodas i C++. Kontrollstrukturer används och är i princip uppbyggda enligt samma logik i alla programmeringsspråk. Både Javas och C#:s kontrollstrukturer har – när det gäller syntaxen – tagits över från och är i princip identiska med C/C++ bortsett från några detaljer. Ännu längre tillbaka i historien kan man hitta deras spår i de första strukturerade språken som *Algol*, *Simula* och *Pascal*.

Sekvens (följd)

En *sekvens* är en följd av instruktioner (bilden till höger) – den enklast möjliga strukturen som tänkas kan. Alla våra program hittills består endast av sekvenser. Varje instruktion kan i sin tur innehålla andra kontrollstrukturer. Så även om sekvensen är en enkel struktur, kan nästlade sammansättningar av den med sig själv (underinstruktioner) och andra kontrollstrukturer ändå ge en ganska invecklad bild.



Selektion (val)

Kontrollstrukturen *selektion* är mer komplex än sekvens. Beroende på antalet alternativ man kan välja mellan tre olika varianter: *Enkel selektion, två- eller flervägsval*. Vi börjar med den första.

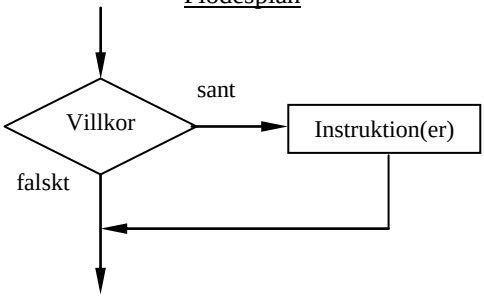
6.2 Enkel selektion: `if`-satsen

Enkel selektion är ett val utan alternativ. Valet görs pga ett villkor. Är villkoret sant, utförs en eller flera instruktioner. Är villkoret falskt, görs ingenting. I termer av pseudokod kan man kalla det för en **OM**-sats.

Pseudokod

OM *villkor* uppfyllt
instruktion(er)

Flödesplan



I C++ kallas den enkla selektionen för **if**-sats och kodas generellt på följande sätt:

```
if (villkor)
{
    sats(er);
}
```

Första raden är **if**-satsens *huvud* och får inte avslutas med semikolon. Resten är **if**-satsens *kropp* som omsluts av klammerparenteserna **{** och **}** som vi i fortsättningen kommer att kalla kort *klamrar*. Kroppens avslutande klammer kan ersätta det semikolon som skulle avsluta hela **if**-satsen. Om kroppen består endast av en sats kan klamrarna utelämnas vilket vi utnyttjar i följande program:

```
// SimpleIf.cpp
// Dividerar endast om det som ska divideras med inte är 0
// Enkel selektion: if-satsen med EN sats: utan klamrar
#include <iostream>
using namespace std;

int main()
{
   float number1, number2;
   cout << "\nGe två positiva tal skilda med mellanslag:  ";
   cin >> number1 >> number2;
   if (number2 != 0)
      cout << "\n\t" << number1 << " dividerad med " <<
          number2 << " blir " << number1 / number2 << "\n\n";
   if (number2 == 0)
      cout << "\n\tOBS! Du har matat in 0 för det andra " <<
            "talet.\n\tDet går inte att dividera med 0.\n\n";

}
```

Programmet läser in två heltal och dividerar dem med varandra. **if**-satserna gör att division endast sker om det andra talet **number2** (det som ska divideras med) *inte* är **0**, för att förhindra den matematiskt odefinierade divisionen med **0**. Följande dialog får man när man matar in ett värde skilt ifrån **0** till det andra talet:

```
Ge två positiva tal skilda med mellanslag:  9 2

        9 dividerad med 2 blir 4.5
```

P.g.a. datatypen **float** till **number1** och **number2** blir det vanlig division: **9/2** vilket ger **4.5** . Matas in däremot **0** till det andra talet uppstår följande dialog:

```
Ge två positiva tal skilda med mellanslag:  9 0

        OBS! Du har matat in 0 för det andra talet.
        Det går inte att dividera med 0.
```

Inmatning av **0** till det andra talet genererar ett ”felmeddelande”, annars får man ut från programmet **SimpleIf** resultatet av divisionen för vilka heltal som helst. Låt oss nu titta närmare på **if**-satserna som åstadkommer distinktionen mellan dessa två alternativ: Det finns två **if**-satser i programmet **SimpleIf**. Den första **if**-satsens huvud

```
if (number2 != 0)
```

betyder i termer av pseudokod: **OM** *number2 är skilt ifrån 0*

Huvudet inleds med det reserverade ordet **if** utan att avsluta raden med semikolon. Utan semikolon, därför att **if**-satsen inte är avslutad än i slutet av denna rad. Sedan ska ju kroppen följa. Efter **if** skrivs ett *villkor (condition)* inom parentes. Observera att parenteserna tillhör syntaxen och inte får under några omständigheter utelämnas. Men hur skriver man *villkor* i C++? Vi blir påminda om algoritmer när vi hör begreppet villkor.

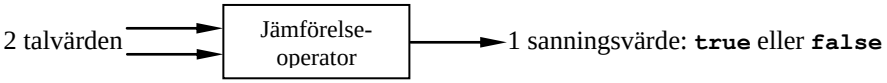
Villkor

Det är viktigt att skilja mellan begreppen *villkor* och *instruktion*. Enklast kan ett villkor förklaras som en *fråga* som endast kan besvaras jakande eller nekande: är **number2** skilt ifrån **0**, ja eller nej? Närmare bestämt är ett villkor en *utsaga* som endast kan vara sann eller falsk. Medan en instruktion är en handling som ska *utföras* kan ett villkor endast *testas* för att få ut svaret ja eller nej, sant eller falskt. **if (number2 != 0)** testar om **number2** är skilt ifrån **0** eller ej. Variabeln **number2**:s värde jämförs med **0**. Finns icke-likhet mellan dessa värden är villkoret sant, annars är villkoret falskt. Dubbeltecknet **!=** (utan mellanslag) är en s.k. *jämförelseoperator*. Det är vanligt att formulera villkor med jämförelseoperatorer. *Icke lika med* med symbolen **!=** är en av dem. Det finns fler som används i **if**-satsers villkor. Därför ska vi titta närmare på sådana operatorer.

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 och returnerar jämförelsens resultat som ett s.k. *sanningsvärde* dvs sant eller falskt, **`true`** eller **`false`** som är reserverade ord.



Sanningsvärdena **`true`** och **`false`** är de enda värden som villkor kan anta varför jämförelseoperatorer används för att skriva villkor. Exempel på villkor formulerade med jämförelseoperatorer är:

```
number == 0
number != 0
7 > 5
guessedNo <= 17
```

Observera att de jämförelseoperatorer som är dubbeltecken, inte får innehålla mellanslag, annars tolkas de som respektive tecken och inte som jämförelseoperatorer. T.ex. är == symbolen för *lika med*. Redan på sid 61 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, t.ex. i villkoret till den andra **`if`**-satsen.

Så långt om **`if`**-satsens *huvud*. Sedan kommer **`if`**-satsens kropp som i programmet **`SimpleIf`** består av *en* enda utskriftssats. Därför kan klamrarna **{ }** kring kroppen utelämnas. Men det vore inte heller fel att skriva dem. Villkorets sanningsvärde avgör nu om kroppen dvs utskriftssatsen utförs eller ej. Är variabeln **`number2`**:s värde icke lika med **`0`**, utförs kroppen. Observera också att hela utskriftssatsen är indragen för att markera att denna tillhör **`if`**-satsen och att den bildar **`if`**-satsens kropp – en kodstil som hör till god programmeringssed och höjer kodens läslighet.

Den andra **`if`**-satsens huvud i programmet **`SimpleIf`**:

```
if (number2 == 0)
```

betyder i termer av pseudokod: **`OM`** **`number2`** *är lika med* **`0`**

Precis som **`!=`** är även dubbeltecknet == (utan mellanslag) en jämförelseoperator, men står för *lika med*. Dvs värdet i variabeln **`number2`** jämförs med **`0`**. Finns likhet mellan dem är **`if`**-satsens villkor sant, annars är villkoret falskt. Observera skillnaden mellan likhet som kodas med *två* likhetstecken == och tilldelning vars kod är

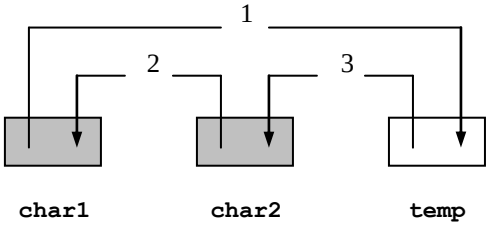
ett likhetstecken =. Även den andra **if**-satsens kropp är en utskriftssats som skriver ut ett felmeddelande om värdet **0** matas in som andra tal. På så sätt utförs inte division med **0**, för divisionen förekommer endast i den första **if**-sats som inte utförs eftersom dess villkor blir falskt, när man matar in **0** som andra tal.

Flera satser i `if`

I programmet **SimpleIf** (sid 92) består båda **if**-satsers kroppar av en enda sats. Därför räcker det med satsens semikolon för att avskilja kroppen från programmets efterföljande satser. Men om **if**-satsens kropp består av flera satser *måste* klamrarna **{** och **}** markera kroppen. Hur ska annars kompilatorn skilja mellan **if**-kroppens och de efterföljande satserna? I programmet **MiniSort** nedan finns ett exempel på detta. Men först ska vi titta på programmets algoritm som handlar om sortering:

Algoritm för platsbyte

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 ovan är en grafisk beskrivning av algoritmen där 1, 2 och 3 anger ordningen i den. Den tredje platsen **temp**, behövs, för att temporärt lägga undan det felplacerade tecknet. I följande program inplementerar vi algoritmen ovan:

```
// MiniSort.cpp
// Läser in 2 tecken, sorterar dem i teckentabellens ordning.
// Enkel selektion: if-satsen med FLERA satser i kroppen.
#include <iostream>
using namespace std;

int main()
{
   char char1, char2, temp;

   /* I n m a t n i n g */
   cout << "\nGe 2 olika tecken skilda med tabulator:   ";
   cin >> char1 >> char2;
```

```
   /* B e a r b e t n i n g */
   if (char1 > char2)
   {
      temp = char1;              // Algoritm för platsbyte
      char1 = char2;             // av de två teckenvärdena
      char2 = temp;              // char1, char2
   }

   /* U t m a t n i n g */
   cout << "\nDe inmatade tecknen förekommer\n "
        << "i teckentabellen i ordningen:\t\t  " << char1
        << "\t" << char2 << "\n\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:

```
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.

Block

I C++ kallas ett antal satser som omsluts av klamrarna **{** och **}** för ett *block*. Blockets uppgift är att gruppera satserna inom klamrarna och *avgränsa* dem från andra delar av programmet. Klamrarna är *gränser* mellan programmets olika delar. De sätter gräns för variablers räckvidd. För att överskrida dem måste vissa regler om *blockstruktur* beaktas. Ibland kan blockets avslutande klammer t.o.m. ersätta ett ev. efterföljande semikolon. Exempel på block fanns redan i vårt allra första program. **Main()**-metodens kropp bildar ett block, det s.k. **Main()**-blocket. För läslighetens skull brukar blockets satser skrivas indragna. Dessutom placerar man blockets klamrar på separata rader. Alternativt kan man placera den inledande klammern i slutet av huvudets rad och den avslutande i blockets slut på separat rad.

Jag väljer dock att använda den förstnämnda stilkonventionen då denna ger en mer lättläst kod: Man kan lättare para ihop de inledande och avslutande klamrarna.

I programexemplet **`MiniSort`** kodas sorteringsalgoritmen i de tre tilldelningssatser som finns i **`if`**-satsens kropp. Avgränsningen innebär här att alla tre satser hör till **`if`**-satsen och att alla tre ska utföras i fall att **`if`**-satsens villkor är sant. Om block-markeringen med klamrarna fattades, skulle endast den första av de tre satserna utföras, vilket skulle innebära att sorteringsalgoritmen inte utförs i sin helhet, dvs ingen sortering sker.

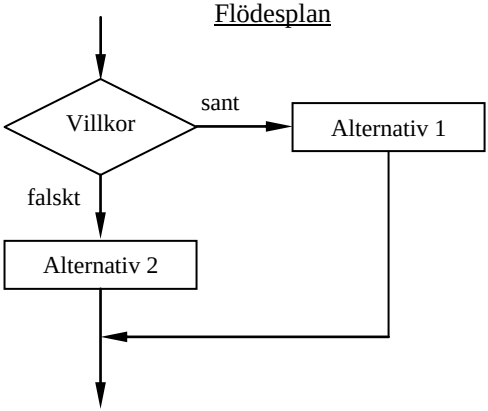
6.3 Tvåvägsval: if-else-satsen

Tvåvägsval är ett val mellan två alternativ. Precis som i **`if`**-satsen görs valet pga ett enda villkor. Är villkoret sant, utförs en eller flera instruktioner. Låt oss kalla dessa *alternativ 1*. Är villkoret falskt, utförs en annan uppsättning instruktioner som vi kallar *alternativ 2*. **`OM-ANNARS`**-satsen är ett exempel på tvåvägsval. Allmänt kan tvåvägsvalet beskrivas så här:

Pseudokod

`OM` *villkor* uppfyllt
    alternativ 1
`ANNARS`
    alternativ 2

Flödesplan



Alternativ 1 och
Alternativ 2 är två olika instruktioner eller uppsättningar av instruktioner.

Endast ett av de två alternativen kommer att utföras, beroende på villkorets sanningsvärde. Då sanningsvärdena sant och falskt utesluter varandra, utesluter även de båda alternativen varandra. Därför går flödet (pilen i flödesplanen) efter alternativ 1 till flödet *efter* alternativ 2. Det vore logiskt fel att leda pilen till ett ställe *före* alternativ 2.

I C++ kallas tvåvägsvalet för **`if-else`**-sats och kodas generellt på följande sätt:

```
if (villkor)
{
    sats(er)1;
}
else
{
    sats(er)2;
}
```

Om **`if`**- eller **`else`**-blocket består endast av en sats kan klamrarna **`{`** och **`}`** utelämnas. Anta att båda block består bara av en sats, då förenklas formen:

```
if (villkor)
    sats1;
else
    sats2;
```

Observera att varje sats i **if-else**-satsen måste avslutas med semikolon enligt semikolonets roll i C++ som satsavslutningstecknet. Detta gäller även för den allra sista satsen i ett block och för sats1 ovan strax före **else**. I andra programmeringsspråk, där semikolonet inte är satsavslutningstecken utan skiljetecknet *mellan* satser, kan semikolonet utelämnas efter sista satsen i ett block. I Pascal t.ex. får inget semikolon sättas före **else**. I C++ däremot måste semikolon skrivas både i blockets sista sats och före **else**. Därför förekommer flera semikolon – minst två – även om vi pratar om *en* **if-else**-sats, vilket beror på att **if-else**-satsen är en huvudsats som innehåller flera delsatser, minst två. Jämför detta med huvud- och underinstruktioner i algoritmer. Följande exempel behandlar **if-else**-satsen med endast en sats i resp. del:

```
// IfElse.cpp
// Läser in ett heltal och avgör om det är jämnt eller udda
// Tvåvägsval: if-else-satsen med EN sats i resp. if-else-del
#include <iostream>
using namespace std;

int main()
{
   int no;
   cout << "\nMata in ett heltal och tryck på Enter:  ";
   cin >> no;
   if (no % 2 == 0)
      cout << "\n\t" << "Det inmatade talet är jämnt.\n\n";
   else
      cout << "\n\t" << "Det inmatade talet är udda. \n\n";
}
```

Körexempel av programmet **IfElse** med ett udda tal som inmatning ger:

```
Mata in ett heltal och tryck på Enter:  7

        Det inmatade talet är udda.
```

Med ett jämnt tal som inmatning får vi:

```
Mata in ett heltal och tryck på Enter:  8

        Det inmatade talet är jämnt.
```

Det egentliga jobbet – nämligen att avgöra mellan *jämnt* och *udda* – har gjorts med hjälp av modulooperatorn som behandlats tidigare (sid 66).

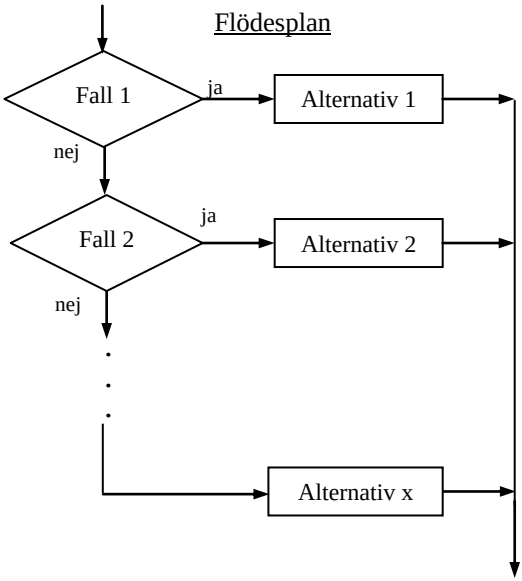
6.4 Flervägsval

Flervägsval är ett val mellan fler än två alternativ. Strukturen och logiken kan beskrivas så här:

Pseudokod

VÄLJ *fall* ur
  fall 1: alternativ 1
  fall 2: alternativ 2
    .
    .
    .
  Annars: alternativ x

Flödesplan



Alternativ 1, 2, … innebär olika *instruktioner* eller olika uppsättningar instruktioner och Fall 1, 2, … motsvarar olika *villkor*.

Observera att det logiska flödet går efter varje fall till ett alternativ och därefter *lämnar* flödesplanen. Dvs flödet går inte efter ett fall till nästa fall. I slutet behöver inget nytt villkor formuleras därför att Alternativ x utförs när det varken föreligger Fall 1, Fall 2, … .

Det finns olika sätt att implementera (realisera) flödesplanen ovan i kod. I praktiken har det visat sig att följande två koncept är mest effektiva och användbara i programmeringen oavsett programmeringsspråk:

- **`if-else`-stegen**
- **`switch`-satsen**

`if-else`-*stegen*

Låt oss ta följande exempel på ett trevägsval: Vi vill skriva ett program som lägger grunden till ett *Gissa tal-spel* som kommer att vidareutvecklas senare. Användaren ska gissa fram ett hemligt tal som är hårdkodat i programmet och får som hjälp inom vilket intervall talet ska ligga `(1-20)` samt om det gissade talet var mindre än, större än eller lika med det hemliga talet. Här kommer version 1 av Gissa tal-

spelet som innehåller ett val mellan tre alternativ: en gissning som är lika med (fall 1), mindre (fall 2) eller större än (fall 3) programmets hemliga tal **17**.

```
// GissaTal_1.cpp
// Låter användaren gissa programmets hemliga tal secret
// Version 1 av Gissa tal-spelet: Kan köras endast en omgång
// Trevägsval med en "if-else stege"
#include <iostream>
using namespace std;

int main()
{
   int guess, secret = 17;
   cout << "\n\tGissa ett heltal mellan 1 och 20:\t";
   cin >> guess;
   if (guess == secret)
      cout << "\n\tGrattis, du har gissat rätt!\n";
   else if (guess < secret)   // Ger hjälp för nästa körning:
      cout << "\n\tFel:\t" << guess << " < hemliga talet\n\n"
           << "\t\tGissa högre nästa gång.\n";
   else
      cout << "\n\tFel:\t" << guess << " > hemliga talet\n\n"
           << "\t\tGissa lägre nästa gång.\n";
}
```

De tre relevanta testen av programmet **GissaTal_1** med gissningar mindre än, större än och lika med 17 ger:

```
     Gissa ett heltal mellan 1 och 20:          12

     Fel:    12 < hemliga talet

             Gissa högre nästa gång.
```

```
     Gissa ett heltal mellan 1 och 20:          19

     Fel:    19 > hemliga talet

             Gissa lägre nästa gång.
```

```
     Gissa ett heltal mellan 1 och 20:          17

     Grattis, du har gissat rätt!
```

`switch`-*satsen*

I C++ kan ett flervägsval vars flödesplan visades på sid 100 även kodas med **`switch`**-satsen. Den generella strukturen till **`switch`**-satsen kan beskrivas så här:

```
switch (uttryck)
{
    case konstant1 :
       sats(er)1;
       break;
    case konstant2 :
       sats(er)2;
       break;
      .
      .
      .
    default:
       sats(er)x;
}
```

Första raden är **`switch`**-satsens *huvud* och får inte avslutas med semikolon. Resten är **`switch`**-satsens *kropp* som består av ett block. Kroppens avslutande klammer ersätter här det semikolon som skulle avsluta hela **`switch`**-satsen.

Med *uttryck* i huvudet menas ett aritmetiskt uttryck (sid 64) vars värde får bara vara av typ **`int`** eller **`char`**. När **`switch`**-satsen exekveras, jämförs uttrycket i huvudet en i taget med alla konstanter som står efter **`case`**. Jämförelsen görs på likhet och innebär följande när man översätter alla **`case`** till **`if`**:

```
if (uttryck == konstant1)
if (uttryck == konstant2)
          .
          .
          .
```

Då blir villkoren som är dolda i **`switch`**-satsen avslöjade: Man ser att de är hårdkodade med operatorn `==` och inte kan ersättas med andra jämförelseoperatorer. Två enskilda värden kan jämföras med varandra endast på likhet.

Om likhet föreligger mellan uttrycket och en konstant, så kommer man in i **`switch`**-satsens kropp och utför alla satser som följer **`case`** tills **`break`** (sid 104) kommer eller kroppen slutar. Programmet utför alltså inte bara de satser som omedelbart följer det **`case`** där likheten inträffar, utan *alla* satser som följer ända tills en **`break`**-sats kommer eller **`switch`**-satsen avslutas. Har man en gång kommit in i **`switch`**-satsen via något **`case`**, stannar man i den utan att likhet mellan uttrycket och konstanten som finns i de efterföljande **`case`**-satserna testas. Om **`switch`**-satsen ska välja endast ett enskilt värde bland flera, borde varje **`case`** avslutas med **`break`**.

Följande programexempel demonstrerar **switch**-satsen: Vi läser in begynnelsebokstaven till en veckodag och det fullständiga veckodagsnamnet skrivs ut. I **switch**-satsen väljs ett alternativ av sex. Tisdag och torsdag behandlas i ett fall.

```
// Switch.cpp
// Demonstrerar flervägsval med switch-satsen.
// Skriver ut veckodagen fullständigt efter inmatning av 1:a
// bokstaven. Vid t(isdag/torsdag) krävs den 2:a bokstaven.
#include <iostream>
#include <conio.h>       // Krävs för funktionen _getch()
using namespace std;

int main()
{
   char letter1, letter2;
   cout << "\n\tMata in begynnelsebokstaven till en"
        << " veckodag:\n\t";
   letter1 = _getch();   // Läser in ETT tecken som tilldelas
                         // letter1 utan att eka på skärmen
   switch (letter1)      // switch-satsen börjar
   {
      case 's':
         cout << "\n\tsöndag";
         break;
      case 'm':
         cout << "\n\tmåndag";
         break;
      case 't':
         cout << "\n\tMata in andra bokstaven:\n\t";
         letter2 = _getch();
         if (letter2 == 'i')
            cout << "\n\ttisdag";
         else
            cout << "\n\ttorsdag";
         break;
      case 'o':
         cout << "\n\tonsdag";
         break;
      case 'f':
         cout << "\n\tfredag";
         break;
      case 'l':
         cout << "\n\tlördag";
         break;
      default:
         cout << "\n\tDet är ingen veckodag!";
   }                     // switch-satsen slutar
   cout << "\n";
}
```

break

break är ett reserverat ord i C++ som bryter programflödet i **switch**-satsen och i loopar (repetitionssatser sid 108). **break** skickar programflödet till den första satsen efter det block i vilket **break** skrivs. Alla satser mellan **break** och blockets avslutande klammer **}** hoppas över. I det här fallet gör alltså **break** att programflödet lämnar **switch**-satsen. Detta garanterar ett entydigt val mellan flera alternativ. Användningen av **break** i **switch**-satsen är frivillig dvs kompilatorn protesterar inte om man utelämnar den. I vissa fall där man inte önskar ett entydigt val mellan enstaka värden, t.ex. när ett val mellan olika intervall simuleras, finns möjligheten att utelämna **break**.

default är motsvarigheten till **else** i **switch**-satsen. Om ingen likhet påträffats i någon **case**-sats, utförs istället de satser som följer efter **default**. På så sätt har man möjligheten att skriva kod som dokumenterar det just inträffade. Ofta väljer man att skriva ut någon form av felmeddelande. Användningen av **default**-satsen är frivillig. Den kan utelämnas i **switch**-satsen, men rekommendationen är att utnyttja möjligheten.

Biblioteksfunktionen _getch()

Inläsningen av ett tecken görs med funktionen **_getch()** som finns förprogrammerad i bibliteksfilen **conio.h** (**con**sole **i**nput/**o**utput). Varje **#include**-direktiv måste stå på en separat rad.

Funktionen **_getch()** läser in endast ett tecken från tangentbordet utan att eka den på bildskärmen. Vi har inte använt **cin** här för inläsning då vi vill åstadkomma den roliga effekten att det fullständiga veckodagsnamnet skrivs ut direkt när man matar in den första bokstaven utan att trycka på Enter. Den inlästa bokstaven tilldelas i samma sats variabeln **letter1**. Sedan jämför **switch**-satsen denna variabels värde med de sex teckenkonstanterna **'s'**, **'m'**, **'t'**, **'o'**, **'f'** och **'l'**. Hittar den likhet med någon av dem, utförs de satser som följer efter resp. **case** tills **break** bryter **switch**-satsen. På så sätt träffas ett entydigt val mellan de sex alternativen. Hittas ingen likhet, har användaren tryckt på en bokstav som inte motsvarar en veckodag. Då skriver **default**-satsen ut ett felmeddelande.

Väljer man alternativ **'t'** som står för både tisdag och torsdag, uppmanas man att mata in den andra bokstaven. Denna läses in med ett andra anrop av funktionen **_getch()** och tilldelas variabeln **letter2**. En vanlig **if-else**-sats skiljer sedan mellan tisdag och torsdag. För enkelhets skull har vi testat den andra bokstaven endast på **'i'** och låtit bli att testa även på **'o'** och andra eventuellt felaktiga inmatningar.

Biblioteksfunktionen **_getch()** är en nyare version av den gamla funktionen **getch()** som är föråldrad (eng.: *depreciated*) och inte längre stöds. En besläktad variant är funktionen **getchar()** som gör samma sak, men dessutom ekar det inlästa tecknet på skärmen.

6.5 Efter-testad repetition: `do`-satsen

Datorn har några egenskaper som är helt överlägsna motsvarande egenskaper hos människan: snabbheten, noggrannheten och förmågan att effektivt lagra och hantera stora datamängder samt förmågan att inte bli trött. Datorn kan upprepa en sak miljardtals gånger utan att tappa i noggrannhet. Denna förmåga utnyttjas i stor skala av alla möjliga datorprogram. Och därför har man en speciell kontrollstruktur i algoritmer som beskriver den: *repetitionen* [*]. ”Att låta datorn göra jobbet” innebär som regel att datorn utför en repetition. Beroende på hur repetitionen, speciellt hur avslutningsvillkoret formuleras och var det placeras i loopen skiljer man mellan tre typer av repetition:

- **Efter-testad repetition**
- **För-testad repetition**
- **Bestämd repetition**

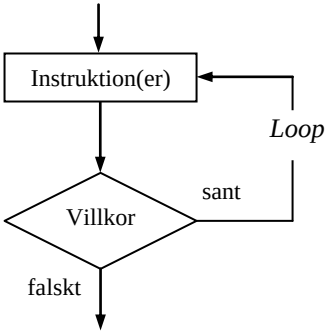
Efter-testad repetition

Det är en upprepningsslinga där avslutningsvillkoret testas *efter* slingans instruktioner dvs *efter* det som egentligen ska upprepas. Så här kan den formuleras i pseudokod och som flödesplan:

Pseudokod

REPETERA
instruktion(er)
SÅ LÄNGE *villkor* uppfyllt

Flödesplan



I C++ inleds den efter-testade repetitionen med **`do`** och skrivs generellt så här:

```
do
{
    sats(er);
} while (villkor);
```

Första raden är **`do`**-satsens *huvud* och får inte avslutas med semikolon. Resten är **`do`**-satsens *kropp* som består av ett block. Om kroppen består endast av en sats kan

[*] I några böcker kallas det för *iteration*. Vi undviker denna term eftersom den används som fackterm i andra sammanhang, t.ex. i numerisk analys. En besläktad term är *rekursion* som behandlas i avsnitt 8.11, sid 222. Allt man kan göra med rekursion kan man även göra med repetition.

klamrarna **{** och **}** utelämnas. Till skillnad från **if**-satsen kan här kroppens avslutande klammer inte ersätta **do**-satsens avslutande semikolon, då **do**-satsen inte är komplett utan fortsätter med villkoret. Och villkoret kan bara testas när det som vanligt skrivs inom vanliga parenteser som följer det reserverade ordet **while**. Först efter villkoret är **do**-satsen komplett vilket bekräftas med det avslutande semikolonet.

Med hjälp av den nya kontrollstrukturen *efter-testad repetition* ska vi nu vidareutveckla Gissa tal-spelets första version **GissaTal_1** genom att lägga till en loop, för att kunna spela flera omgångar. Följande program använder en **do**-sats för att kunna köra programmet så länge tills man gissat rätt:

```
// GissaTal_2.cpp
// Gissa tal, ver 2 med loop av typ efter-testad repetition
#include <iostream>
using namespace std;

int main()
{
   int guessedNo;

   do
   {
      cout << "Gissa ett tal mellan 1 och 20: ";
      cin >> guessedNo;
      cout << "\n\t";
      switch (guessedNo)
      {
         case 17:
            cout << "\aGrattis, du gissade rätt!\n\n";
            break;
         default:
            if (guessedNo < 17)
               cout << "För LITET, försök igen!\n\n";
            else
               cout << "För STORT, försök igen!\n\n";
      }
   } while (guessedNo != 17);

}
```

do-satsen är en lämplig variant av repetition när det gäller att åstadkomma en dialog mellan datorn och användaren. I **GissaTal_2** inleds dialogen med en ledtext för inläsning av **guessedNo**. Sedan tar **switch**-satsen hand om valet mellan tre alternativ, nämligen om det gissade talet är lika med, mindre än eller större än spelets hemliga tal **17**. I slutet testas om **guessedNo** är skilt från **17**. Om så är fallet, återvänder programflödet till början av **do**-blocket och allt upprepas tills **guessedNo** någon gång blir lika med **17**.

I **do**-satsen utförs satserna första gången oavsett om villkoret är sant eller falskt. Sedan testas villkoret: är det sant upprepas satserna. Sedan testas villkoret igen: är

det fortfarande sant, fortsätts repetitionen osv. Är villkoret falskt, stoppas repetitionen. Man kan alltså säga: dörrvakten (villkoret) står vid utgången till lokalen (slingan). Konsekvensen blir att, när villkoret är falskt från början, kommer satserna i alla fall att utföras åtminstone *en* gång. I nästa avsnitt behandlas en annan variant av repetition, den för-testade repetitionen där dörrvakten så att säga står vid ingången till lokalen och inte tillåter att någon sats exekveras när villkoret är falskt från början. Är villkoret sant hela tiden, kommer slingan att snurra i all evighet. En körning av **`GissaTal_2`** kan t.ex. ge följande dialog med datorn:

```
Gissa ett tal mellan 1 och 20: 5
        För LITET, försök igen!
Gissa ett tal mellan 1 och 20: 15
        För LITET, försök igen!
Gissa ett tal mellan 1 och 20: 19
        För STORT, försök igen!
Gissa ett tal mellan 1 och 20: 17
        Grattis, du gissade rätt!
```

Det är avgörande att skilja mellan *repetition* och *selektion*. I selektionens pseudokod har vi nyckelordet **`OM`** och i C++ det reserverade ordet **`if`**, vilket innebär ett val *en enda* gång, dvs ingen upprepning alls. I repetitionens pseudokod har vi **`SÅ LÄNGE`** och i C++ det reserverade ordet **`while`**, vilket innebär att villkoret testas *upprepade gånger*. I selektionens flödesplan går allt flöde endast framåt dvs alla pilar nedåt, se sid 92, 98 och 100. I repetitionens flödesplan ovan går pilen efter instruktionerna *tillbaka* till villkoret för att testa det igen. Orsaken till att programflödet går tillbaka till en tidigare punkt är att det finns en hoppsats av typ **`goto`** inbyggd i alla repetitioner där labeln är hårdkodad och placerad på en fast föreskriven plats.

Ett annat sätt att se på efter-testad repetition är att i pseudokoden (sid 105) använda nyckelordet **`TILLS`** istället för **`SÅ LÄNGE`**. Så kan logiken ibland uppfattas enklare:

`REPETERA`
    instruktion(er)
`TILLS` *villkor* **inte** uppfyllt

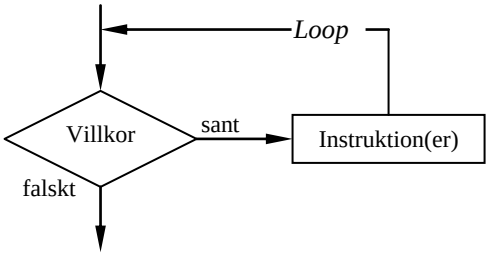
Om man väljer samma villkor som i formuleringen med **`SÅ LÄNGE`**, dvs bibehåller villkorets formulering, måste man negera villkoret när man går över till **`TILLS`**. Det beror på skillnaden i den logiska innebörden av **`SÅ LÄNGE`** och **`TILLS`**. I flödesplanen av den efter-testade repetitionen blir det ingen strukturell ändring, bara man sätter sant och falskt på de logiskt korrekta utgångarna av villkoret. I några programmeringsspråk finns **`TILLS`**, men i C++ används **`while`**.

6.6 För-testad repetition: `while`-satsen

`while-satsen` är en upprepningsslinga där avslutningsvillkoret testas *före* slingans instruktioner dvs *innan* det som ska upprepas. Enda skillnaden gentemot den efter-testade repetitionen är ordningen mellan villkor och instruktioner. Denna ordning blir omvänd:

Pseudokod och ritas som Flödesplan

SÅ LÄNGE *villkor* uppfyllt
instruktion(er)



I C++ inleds den för-testade repetitionen med det reserverade ordet **`while`** och skrivs generellt på följande sätt:

```
while (villkor)
{
    sats(er);
}
```

Första raden är **`while`**-satsens *huvud* och får inte avslutas med semikolon. Resten är **`while`**-satsens *kropp* som omsluts av klamrarna **{** och **}**. Kroppens avslutande klammer kan ersätta det semikolon som skulle avsluta hela **`while`**-satsen. Om kroppen består endast av en sats kan klamrarna utelämnas. Här följer ett exempel med två satser i kroppen och därför med klamrar kring dem:

```
// While.cpp
// Skriver ut alla heltal mellan 0 och 99
// För-testad repetition: while-satsen
#include <iostream>
using namespace std;

int main()
{
   int i = 0;

   while (i < 100)
   {
      cout << i << '\t';
      i++;
   }

   cout << "\n\t\t\tEfter loopen är i = " << i << "\n\n";
}
```

Först testas villkoret **i < 100**: är det sant utförs kroppens satser. Sedan testas villkoret igen: är det fortfarande sant, utförs satserna igen osv. Detta upprepas gång på gång. Här består **while**-satsens kropp av två satser, en **cout**-sats – den första – som i varje varv skriver ut variabeln **i**:s värde följt av en tabulator. Den andra satsen **i++;** gör att **i**:s värde ökar med **1**. Båda satser utförs så länge **i** är strikt mindre än **100** dvs från **0** till **99**. Från **0** därför att **i** har initierats till **0** *före* **while**-loopen. Och till **99** därför att **99** är det sista värdet som uppfyller loopens avslutningsvillkor **i < 100**. Dvs med **99** som **i**:s värde kommer vi fortfarande in i loopen, värdet skrivs ut och ökas sedan. Men med **100** som **i**:s värde kommer vi inte längre in i loopen. Programflödet lämnar **while**-satsen och fortsätter med den andra **cout**-satsen som står utanför och skriver ut **i**:s aktuella värde **100**. En körning visar detta:

```
0     1     2     3     4     5     6     7     8     9
10    11    12    13    14    15    16    17    18    19
20    21    22    23    24    25    26    27    28    29
30    31    32    33    34    35    36    37    38    39
40    41    42    43    44    45    46    47    48    49
50    51    52    53    54    55    56    57    58    59
60    61    62    63    64    65    66    67    68    69
70    71    72    73    74    75    76    77    78    79
80    81    82    83    84    85    86    87    88    89
90    91    92    93    94    95    96    97    98    99

                    Efter loopen är i = 100
```

Hade vi använt prefixvarianten av ökningsoperatorn **++i** hade talen från **1** till **100** skrivits ut därför att utskriften av **i** hade då skett *efter* ökningen. Testa gärna!

while-satsen är den enklaste varianten av loop i C++. Vi vill använda den för att illustrera en företeelse som man brukar råka ut för när man jobbar med loopar:

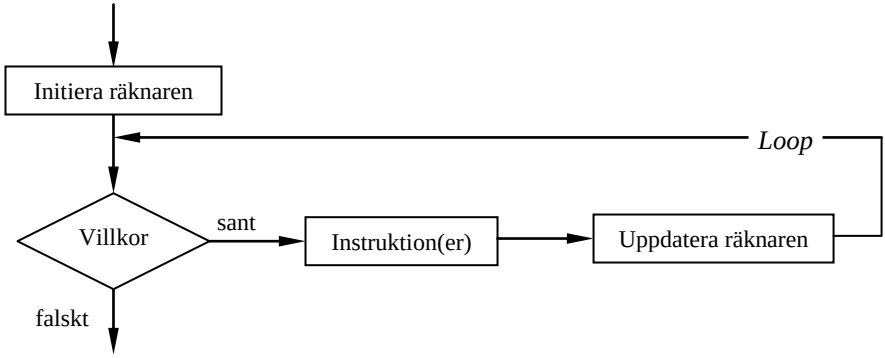
Evighetsslinga

I exemplet **While** är villkoret **i < 100**. Är villkoret sant från början och förblir sant hela tiden, kommer satserna att utföras i all evighet vilket resulterar i en s.k. *evighetsslinga*. För att undvika den, måste villkoret och satserna formuleras på ett sådant sätt att slingans kropp *ändrar* villkorets sanningsvärde, så att villkoret *blir* falskt efter några varv. I programmet **While** har vi åstadkommit detta genom att ha **i++** i kroppens **cout**-sats samtidigt som villkoret är formulerat som **i < 100**. Dvs, har man med en lämplig initiering av **i** kommit in i **while**-satsen, kommer **i** att öka med 1 i varje varv så att det någon gång når **100**. Då stoppas slingan. Glömmer man ökningen **++** och initierar **i** med ett värde mindre än **100** blir **while**-satsen en evighetsslinga. Omvänt: Är **while**-villkoret falskt från början, görs ingenting. Initieras **i** till ett värde större än eller lika med **100**, blir villkoret falskt från början och man kommer aldrig in i kroppen (”aldrigslinga”). Programflödet fortsätter vid första satsen *efter* **while**-slingan.

6.7 Bestämd repetition: `for`-satsen

`for`-satsen är en upprepningsslinga där antalet repetitioner är känt i förväg. I de hittills behandlade varianterna – för- och efter-testad repetition – styr endast villkoret antalet repetitioner och man kan få reda på antalet repetitioner efter att ha kört programmet dvs i efterhand. I den bestämda repetitionen kan programmeraren redan vid kodningen *bestämma* antalet repetitioner. Det kan vara användbart i de fall där man vet hur många gånger en sak ska upprepas. Visserligen finns även i den bestämda repetitionen ett villkor som testas i varje varv, men det finns även en inbyggd möjlighet att styra villkoret och därmed antalet repetitioner med hjälp av en *räknare*, även kallad *styrvariabel*.

Räknaren sätts före repetitionen till ett önskat startvärde, för det mesta något heltal, ofta 1. Detta kallas *initiering* av räknaren dvs den allra första tilldelningen av ett värde till räknaren. Sedan testas ett villkor där man brukar lägga in ett önskat *slutvärde* på räknaren. Därmed är antalet repetitionerna fastlagt, t.ex. till slutvärde minus startvärde om räknaren ökats med 1. Om villkoret är uppfyllt, t.ex. om räknaren är mindre än slutvärdet, utförs ett antal instruktioner. Sedan görs en *uppdatering* av räknaren, oftast en ökning med 1, men det är även möjligt att räkna nedåt eller välja ett annat steg än 1. Allt detta händer i varje varv. Den bestämda repetitionens flödesplan ser ut så här:

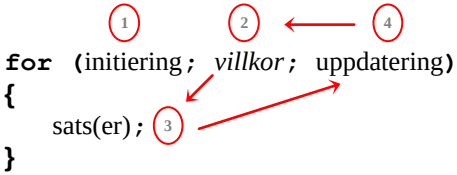


Flödesplanen åskådliggör den *logiska strukturen* av problemet. Den bestämda repetitionens pseudokod blir enligt flödesplanen ovan:

```
Initiera räknaren
SÅ LÄNGE villkor är uppfyllt
    utför instruktion(er)
    uppdatera räknaren
```

Nyckelordet **`SÅ LÄNGE`** i denna pseudokod visar att den bestämda repetitionen alltid kan översättas till en **`while`**-sats om man själv tar hand om räknaren. I programexemplet på nästa sida har vi översatt **`while`**-satsen i **`While`** till **`for`**-sats. Precis som i **`while`**-satsen har man i princip friheten att formulera villkoret hur som helst. Men då räknaren är inbyggd i den bestämda repetitionen, kan man i

villkoret jämföra räknaren med slutvärdet, t.ex. så här: *”räknare är mindre än eller lika med slutvärde”*. Detta ger ett specialfall av den bestämda repetitionen*. **for**-satsen inleds med det reserverade ordet **for** och skrivs generellt så här:



Första raden är **for**-satsens *huvud* och får inte avslutas med semikolon. Resten är **for**-satsens *kropp* som omsluts av klamrarna **{** och **}**. Kroppens avslutande klammer kan ersätta det semikolon som skulle avsluta hela **for**-satsen. Om kroppen endast består av en sats kan klamrarna utelämnas. Jämför man C++ koden med flödesplanen på förra sidan kan man konstatera att C++ koden är lite kryptisk i den bemärkelsen att den inte följer flödesplanens ordning initiering – villkor – instruktion(er) – uppdatering. Därför har vi i koden ovan numrerat **for**-satsens olika delar för att visa i vilken ordning de utförs. Pilarna markerar loopens förlopp. Initieringen görs endast en gång och ingår ej i loopen. Både initieringen och uppdateringen, avser räknaren som är en vanlig variabel och därför måste deklareras precis som vilken variabel som helst.

Översättning av while till for

För att se hur **for**-satsen fungerar har vi i följande program översatt **while**-satsen i programmet **While** (sid 108) till en **for**-sats:

```
// While->For.cpp
// Gör samma sak som While: skriver ut talen mellan 0 och 99
// Översättning av while-satsen (sid 108) till for-satsen
// Bestämd repetition: for-satsen
#include <iostream>
using namespace std;

int main()
{
   int i;

   for (i = 0; i < 100; i++)
       cout << i << '\t';

   cout << "\n\t\t\tEfter loopen är i = " << i << "\n\n";
}
```

* I några äldre programspråk som t.ex. Basic, Fortran och Pascal, finns endast detta specialfall, där villkoret implicit (dvs underförstått) är inbyggt och räknarens uppdatering sker automatiskt. Det här specialfallet kan beskrivas med följande pseudokod:

STEGA räknaren **FRÅN** startvärde **TILL** slutvärde (med **STEG**)
instruktion(er)

Programmet ovan ger exakt samma utskrift som **While** (sid 108) eftersom **for**-satsen i programmet är en ren översättning av **while**-satsen i **While**.

while vs. for

Vi ska här föra en kort diskussion om dessa två varianter av repetition. I **While** ser **while**-satsen ut så här:

```
int i = 0;
while (i < 100)
{
   cout << i << '\t';
   i++;
}
```

I **While->For** har den direkt översatts till följande **for**-sats:

```
int i;
for (i = 0; i < 100; i++)
    cout << i << '\t';
```

Dessa två varianter av repetition gör precis samma sak. Beviset är att de producerar samma utskrift, se sid 109. Låt oss jämföra deras koder. Variabeln **i** spelar en nyckelroll i båda, räknarens roll. I **while**-satsen initieras den innan. I **for**-satsen utnyttjar vi möjligheten att initiera räknaren i satsens huvud. Denna möjlighet saknas i **while**-satsen. Villkoret **i < 100** tas över oförändrat. Uppdateringen av räknaren **i++** som står i **while**-satsens kropp flyttas till **for**-satsens huvud. Andra sätt att skriva **for**-satsen med samma funktionalitet är:

```
int i = 0;
for (i; i < 100; i++)
    cout << i << '\t';
```

Eller:

```
int i = 0;
for (; i < 100; i++)
    cout << i << '\t';
```

Dvs även initieringen av räknaren kan skrivas precis som i **while** före **for**-satsen vilket bättre överensstämmer med flödesplanen på sidan 110: Initieringen står utanför loopen. I alla dessa tre varianter deklareras räknaren **i** *före* **for**-satsen och är därför giltig även *efter* **for**-satsen när vi skriver ut värdet i **cout**-satsen:

```
cout << "\n\t\t\tEfter loopen är i = " << i << "\n\n";
```

Och får utskriften med det korrekta **i**-värdet:

```
Efter loopen är i = 100
```

Det kan vi göra eftersom **i** är en variabel som gäller i hela programmet då den är definierad utanför **for**-satsen (globalt). En helt annan situation uppstår om vi – vilket är också tillåtet – deklarerar räknaren **i** inuti **for**-satsen (lokalt):

```
for (int i = 0; i < 100; i++)
    cout << i << '\t';
```

Här är variabeln **i** inte längre definierad i hela programmet och gäller därför inte i hela programmet utan endast i **for**-satsen, så att säga lokalt. Efter **for**-satsen ”dör” variabeln **i**. Varje försök att referera till den *efter* **for**-satsen leder till kompileringsfel. **for**-satsen fungerar som ett inre block nästlat i det yttre **main()**-blocket. Det inre blockets variabler är inte synliga utåt i enlighet med de generella regler i C++ om globala och lokala variabler. Detta gäller i den nya C++ standarden och kan formuleras som en regel:

> **for**-satsens räknare är odefinierad efter **for**-satsen om den deklareras inuti **for**-satsen.

Det kan hända att i några äldre kompilatorer som inte 100% följer ANSI/ISO-standarden, den här regeln inte är implementerad.

ASCII-tabellen med `for`

När vi i kapitel 5 tog upp ASCII-tabellen kunde vi i programmen **Char**, **Char2int** och **Ascii** få reda på *enskilda* teckens ASCII-kod och omvänt med hjälp av explicit typkonvertering. Nu när vi kan hantera loopar kan vi skriva ut hela ASCII-tabellen genom att mata in **start** och **slut** på ett kodintervall. En **for**-loop skriver sedan ut både tecken och tillhörande ASCII-kod genom att använda en **int**-variabel **code** som räknare och explicit typkonvertera resp. tecken från **int** till **char**:

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

int main()
{
   int start, slut;
   cout << "\nAnge början                        : ";
   cin >> start;
   cout << "och slutet på ett ASCII-intervall: ";
   cin>> slut;
   cout << '\n';
   for (int code=start; code<=slut; code++)
   {
       cout << (char) code << " = " << code << "    ";
       if (code % 6 == 0) cout << "\n";
   }
   cout << "\n\n";
}
```

Efter att ha läst in två heltal till variablerna **start** och **slut**, initieras **for**-satsens räknare till **start**. Så länge **code** är mindre än eller lika med **slut**, kommer vi in i **for**-satsens kropp där två satser, en **cout**- och en **if**-sats utförs. Ett körexempel med programmet **AsciiFor** ger följande utskrift som visar alla läs- och skrivbara tecken i ASCII-tabellen, både standard ASCII (till **127**) och den utvidgade teckenuppsättningen – de icke-standardkoderna mellan **128** och **255**:

```
Ange början                      : 33
och slutet på ett ASCII-intervall: 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
```

Det första inmatade värdet **33** lagras i **start** och det andra, **255** i **slut**. **for**-satsens räknare **code** initieras till **start** dvs **33**. Då **code** (**33**) är mindre än **slut** (**255**), kommer vi in i **for**-satsen: Den första satsen där, **cout**-satsen skriver ut **! = 33** följd av fyra mellanslag. Att det blir **!** och inte **33** beror på att vi i **cout**-satsen omvandlat **code**:s datatyp som är **int** till **char**. Sedan kommer **if**-satsen vars villkor är **code % 6 == 0**. Det här villkoret är falskt då **code**:s värde **33**, modulo **6** ger **3**, se modulooperatorn sid 66, som inte är **== 0**. Därför utförs inte **if**-satsens kropp dvs inget radbyte skrivs ut. Efter **if**-satsen utförs enligt flödesplanen på sid 110 uppdateringen **code++** som ökar **code**:s värde med 1 så att **code** blir **34**.

Efter **for**-satsens första varv går programflödet tillbaka till villkoret **code<=slut**. Nu är **code 34** som jämförs med **slut**:s värde **255**. **34** är mindre än **255**, så vi kommer in i **for**-satsen för andra gången. Då skrivs ut **" = 34** följt av fyra mellanslag. Då **if**-satsens villkor fortfarande är falskt – **34** modulo **6** ger **4** som inte är **== 0** – utförs kroppen inte heller den här gången: inget radbyte. Sedan uppdateras **code**:s värde till **35**. Där slutade loopens andra varv.

Allt detta upprepas även i **for**-satsens 3:e, och 4:e varv. Vi kommer så långt: **35** och **36** är mindre än **255**. I det 4:e varvet har **code** hunnit bli **36**. Då skrivs ut **$ = 36** följt av fyra mellanslag. Men nu är för första gången **if**-satsens villkor **code % 6 == 0** sant, eftersom **code**:s värde, **36** modulo **6** ger **0**. Därför utförs **if**-satsens kropp dvs radbyte. Sedan uppdateras **code**:s värde till **37**. Då slutar 4:e varvet.

På den nya raden – i loopens 5:e varv – skrivs ut **% = 37** följt av fyra mellanslag. Därefter inget radbyte då pga **37** modulo **6** är **1**. Utskriften fortsätter på den nya raden utan radbyte tills **code**:s värde nått **42** för alla följande **code**-värden **38**, **39**, **40** och **41** modulo **6** ger ett värde skilt ifrån **0**. För andra gången blir **if**-satsens villkor sant, när **code** är **42** för **42** modulo **6** ger **0**. Mönstret har klarnat: **if**-satsens roll är att producera radbyte när **code**:s värde är jämnt delbart med **6** dvs var sjätte utskrift. Detta är just innebörden i villkoret **code % 6 == 0**. Vi har gjort så för att få en tabellartad utskrift.

for-satsen avslutas när villkoret **code<=slut** blivit falskt dvs när **code** nått **256** som är varken mindre eller lika med utan större än **slut (255)**. Därför stoppas repetitionen. Med programmet **AsciiFor** kan man, genom att mata in olika värden för **start** och **slut**, avsöka hela ASCII- och den utvidgade teckentabellen och få reda på den aktuella teckenuppsättningen på sin dator. Medan **do**-satsen är en lämplig variant av repetition för dialoger mellan dator och användare, används **for**-satsen främst i repetitioner vars antal är känt i förväg. I **AsciiFor** är intervallgränserna givna i förväg och därför vet man hur många gånger utskriften ska upprepas. Av den anledningen är **for**-satsen den lämpliga varianten av repetition för denna typ av problem. Frågan om vilken variant av repetition man ska välja, kan inte besvaras generellt, eftersom det är det konkreta problemet som avgör valet.

6.8 Nästlade `for`-satser

Nästlade **for**-satser är ett viktigt verktyg i alla programmeringsspråk för att bearbeta flerdimensionella strukturer, t.ex. att åstadkomma en ordnad tvådimensionell utskrift. En tabell, ett rektangulärt schema av tal eller den labyrintartade figuren i övn 6.9 (sid 124) är exempel på sådana strukturer. Följande program nästlar två **for**-satser i varandra (därför 2-dim.) för att skriva ut ett rektangulärt schema fyllt med stjärnor, s.k. asterisker:

```
// Stars.cpp
// Ritar en rektangel fylld med stjärnor med nästlad for-sats
#include <iostream>
using namespace std;

int main()
{
   cout << '\n';
   for (int y=1; y<=9; y++)           // Yttre slinga ordnar
   {                                  // 9 rader med radbyte
       cout << "y=" << y << ":   ";
       for (int x=1; x<=20; x++)      // Inre slinga ritar en
           cout << "*";               // rad av 20 stjärnor
       cout << '\n';
   }
   cout << '\n';
   return 0;
}
```

När vi nästlade **if-else**-satser i varandra (sid 101) pratade vi om en inre **if-else**-sats som nästlas i en yttre. Samma sak är det här: Vi har en inre **for**-slinga som nästlas i en yttre. Den yttre **for**-slingan omfattar tre satser, för det första en **cout**-sats som skriver ut den yttre varvräknaren **y**:s värde, för det andra den inre **for**-slingan och för det tredje en **cout**-sats som gör radbyte. Därför är dessa satser omslutna av klamrar. Medan den inre slingan omfattar endast en sats, **cout**-satsen som skriver ut en stjärna (=asterisk). Programmet producerar en rektangulär utskrift av stjärnor bestående av 9 rader och 20 kolumner samt en kolumn, en slags vertikal ”rubrik”, med radnumrering:

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

Den yttre slingan med räknaren **y** skriver först ut **y**:s värde. Sedan låter den utföra den inre slingan med räknaren **x** som går 20 varv och skriver ut en stjärna i varje varv. Programmet **Stars** demonstrerar följande regel som gäller generellt för alla nästlade **for**-satser:

> Regel för nästlade **for**-satser:
>
> Har programflödet kommit in i en nästlad **for**-sats, måste den inre **for**-slingan slutföras innan den yttre kan varva vidare.

Enligt denna regel skriver den inre slingan ut en rad av 20 stjärnor varje gång den utförs. Den yttre slingan ser till att detta sker 9 gånger i och med den går 9 varv. Sist utför den yttre slingan ett radbyte. Så uppstår utskriften ovan. Sammanfattningsvis kan vi säga att den yttre slingan låter den inre slingan skriva ut raderna och göra radbyte, medan den inre slingan skriver ut stjärnorna i varje rad.

Hantering av slumptal

Utskrift av en tabell är ett annat typiskt exempel som man kan programmera med en nästlad **for**-sats. Vi vill skriva ut en tabell av slumptal. Därför öppnar vi här en parentes om hantering av slumptal. Med datorn som en deterministisk maskin kan man inte producera ”äkta” slumptal utan endast simulera dvs på något sätt beräkna slumptal, s.k. *pseudo*slumptal. I C++ kan man göra det med den fördefinierade funktionen **rand()**. Metoden som används i **rand()** är ganska enkel och går ut på att beräkna ett slumptal baserat på förra slumptal – en serie av slumptal utgående från ett startvärde. Samma startvärde producerar alltid samma följd av slumptal så att det inte blir någon variation vid olika körningar om man inte varierar startvärdena. Ett startvärde kan bestämmas genom anrop av en annan funktion som heter **srand()**. För att få variation vid olika körningar måste **srand()** vid varje körning anropas med en annan parameter (det som skrivs inom parenteserna) för att initiera **rand()** med ett nytt startvärde. I programmet **NestedFor** har vi åstadkommit detta genom att skicka **time(0)** som parameter till **srand()**. Funktionen **time()** returnerar datorns aktuella datum och tid, omräknat till antal sekunder – ett stort heltal. Då tiden varierar från körning till körning initieras **rand()** vid varje körning med detta heltal till ett nytt startvärde. Så garanterar den varierande tiden variation i slumpen. Funktionen **time()** är definierad i biblioteksfilen **ctime** som måste inkluderas i alla program som anropar **time()**. Alla biblioteksnamn som börjar på **c** kommer ursprungligen från C:s standardbibliotek. På sid 30 nämnde vi att man i C++ även kunde använda C:s standardbibliotek som ingår i ANSI/-ISO-standarden för C och därmed i den nya standarden för C++. Där har man gett alla C:s biblioteksfiler ”filnamn**.h**” det nya namnet ”**c**filnamn”.

rand() slumpar fram heltal mellan **0** och den fördefinierade konstanten **RAND_MAX** som är lika stor som **SHRT_MAX**, max.-gränsen 32 767 till datatypen **short** (sid 78). För att skräddarsy **rand()** för vårt ändamål, att få slumptal mellan 1 och 6, utför vi en *skalning* med **% 6** och en *skiftning* med **1**:

```
1 + rand() % 6
```

Skalningen innebär en minskning av de slumpvärden som **rand()** ger. Modulooperatorn drar ju av alla multiplar av 6 från slumpvärdena så att endast en rest som är mindre än 6 blir kvar. Dvs **rand() % 6** ger slumptal mellan 0 och 5 (om modulooperatorn se sid 66). Skiftningen dvs förskjutningen av intervallet [0, 5] med + 1 ger slumptal mellan 1 och 6. Detta är en transformation som kan generaliseras: Vill man ha slumptal mellan **a** och **b** och **a < b**, kan man transformera slumptal mellan **0** och **RAND_MAX** till slumptal mellan **a** och **b**, genom att skriva:

```
a + rand() % (b – a + 1)
```

Är **a > b** måste formeln ovan ersättas med:

```
b + rand() % (a - b + 1)
```

Vi kommer att använda dessa formler (sid 138) när vi modulariserar programmet nedan som skriver ut slumptal mellan 1 och 6 i en tabell:

```
// NestedFor.cpp
// Simulerar tärningskast: Slumpar fram tal mellan 1 och 6
// och skriver ut dem i en tabell. Nästlad for-sats
#include <iostream>
#include <ctime>                // Krävs för funktionen time()
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';
   }
   cout << '\n';
   return 0;
}
```

Den nästlade **for**-satsen, även kallad *dubbel* **for**-sats, består av två slingor: den *inre* och den *yttre* slingan. För att förstå deras samverkan räcker det att tillämpa **for**-satsens logiska struktur – åskådliggjord i flödesplanen på sid 110 – på de enskilda slingorna. Därför är den nästlade **for**-satsen ingen ny kontrollstruktur utan en nästling av den redan kända **for**-satsen. Den yttre **for**-slingans huvud är:

```
for (r=1; r<=rader; r++)
```

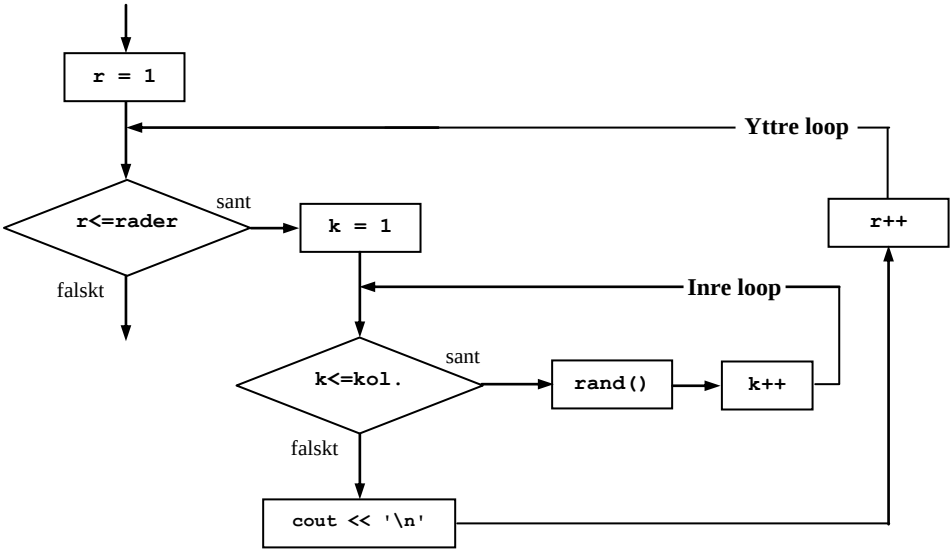
Detta huvud initierar en räknare **r** till **1**, testar villkoret, men skickar sedan programflödet enligt flödesplan till kroppen. Den består av två satser, därför klam-

rarna, varav den första är den inre **for**-slingan och den andra en **cout**-sats som gör radbrytning. När vi kommer in i den inre **for**-slingan, ser dess huvud ut så här:

```
for (k=1; k<=kolumner; k++)
```

Detta initierar en ny räknare **k** till **1**, testar villkoret och skickar programflödet till sin egen kropp som består av utskriften av ett slumptal följd av tre mellanslag. När detta första slumptal skrivs ut, är värdet till både den yttre och den inre **for**-slingans räknare **1** och vi befinner oss i koden i den inre **for**-slingans kropp. Enligt flödesplan kommer vi nu till den inre **for**-slingans uppdatering **k++**. Dvs **k** blir **2**, villkoret testas och om det är sant, återvänder vi till den inre **for**-slingans kropp och nästa slumptal följd av tre mellanslag skrivs ut på samma rad som det första. Radbrytning utförs först i **cout**-satsen. Detta sker, när den inre **for**-slingans villkor blir falskt första gången, dvs när **k** blir större än det inmatade värdet för **kolumner**. Då har tabellens första rad skrivits ut. Vi ser alltså att programmet skriver ut tabellen radvis.

Efter den 1:a raden är vi klara med den yttre **for**-slingans kropp. Enligt flödesplan går programflödet vidare till den yttre **for**-slingans uppdatering **r++**, dvs **r** blir **2**. Nu har programflödet gått ett fullständigt varv i den yttre **for**-slingan. Därför återvänder det till villkoret **r <= rader**. Om det är sant, kommer vi in i den yttre **for**-slingans andra varv och allt upprepar sig: den inre slingan skriver ut tabellens andra rad. Sedan görs radbyte. Radutskrifterna avslutas när den yttre **for**-slingans räknare **r** blivit större än det inmatade värdet för **rader**. Så uppstår tabellen.



Illustrationen ovan som är flödesplanen till den nästlade **for**-satsen i **NestedFor**, visar förloppet.

Jämför denna flödesplan med den enkla **for**-satsens flödesplan på sid 110. Pga platsbrist i flödesplanen står i den inre loopens första ruta, **rand()** som en slags förkortning för satsen **cout << 1 + rand() % 6 << " ";** som förekommer i programmet **NestedFor**.

Ett körexempel av **NestedFor** ger följande utskrift. Pga variationen i slumpningen får man vid varje körning andra slumptalsvärden:

```
Ange antal rader och kolumner (t.ex. 10 15): 10 15

Det blir 150 tärningskast:

5   6   6   6   3   6   3   3   3   6   3   5   6   1   2
4   4   6   1   6   4   3   2   4   3   1   3   1   1   6
1   2   2   5   2   5   5   3   1   2   4   4   3   1   3
2   2   1   4   4   3   6   2   5   5   3   6   5   3   2
1   3   3   5   6   1   6   3   5   1   4   3   6   2   2
4   2   2   3   5   5   4   6   5   3   1   3   4   5   4
3   5   5   4   6   3   6   1   6   1   6   4   6   6   5
3   6   6   3   4   4   4   3   6   4   3   6   5   6   4
2   2   4   6   5   6   6   6   1   5   4   2   6   1   3
2   6   6   4   5   3   6   6   2   3   6   6   6   4   6
```

Multiplikationstabellen

```
// MultipTab.cpp
// Skriver ut multiplikationstabellen med nästlad for-sats
#include <iostream>
using namespace std;

int main()
{
   cout << "\nMultiplikationstabellen\n"
        << "-----------------------\n\n";
   for (int a = 1; a <= 9; a++)
   {
       for (int b = 1; b <= 9; b++)
       {
           cout.width(6);   // Reserverar 6 platser för varje
                            // utskrift: Värdena placeras
           cout << a*b;     // by default högerjusterade
       }
       cout << "\n\n";
   }
   return 0;
}
```

Nu när vi någorlunda behärskar den nästlade **for**-satsen kan vi använda den för att skriva ut multiplikationstabellen och samtidigt vidareutveckla tekniken att få rätt mellanrum mellan utskrifterna. I utskriften ovan t.ex. hade vi fått mellanrum mellan slumptalen genom att i koden helt enkelt skicka ett antal mellanslag `" "` till **cout** vilket inte precis är någon professionell teknik. I programmet ovan används istället en funktion för formatering av utskrift som är fördefinierad i **cout** just för sådana ändamål:

Funktionen **width()** är en s.k. *metod* * eller *medlemsfunktion* som är fördefinierad i **cout** och anropas med s.k. *punktnotation* (sid 220) :

```
cout.width(6);
```

Denna funktion reserverar ett antal platser – här 6 – för den näst följande utskriften i **cout**. Dess giltighet sträcker sig inte över hela **cout**-strömmen, utan endast till *en* utskrift efter anropet. Därför måste anropet stå i den inre **for**-loopen före **cout**-satsen om *alla* utskrifter i tabellen ska ha 6 platser till förfogande. Om man flyttar anropet utanför den inre, men forfarande i den yttre loopen, får varje rads första utskrift 6 platser, de andra inte. Och om man flyttar anropet utanför den inre och den yttre loopen, får endast den allra första utskriften 6 platser, alla andra inte. Testa gärna dessa alternativ. Men så som vi har placerat anropet får vi följande utskrift när vi kör **MultipTab**:

```
Multiplikationstabellen
-----------------------

     1     2     3     4     5     6     7     8     9
     2     4     6     8    10    12    14    16    18
     3     6     9    12    15    18    21    24    27
     4     8    12    16    20    24    28    32    36
     5    10    15    20    25    30    35    40    45
     6    12    18    24    30    36    42    48    54
     7    14    21    28    35    42    49    56    63
     8    16    24    32    40    48    56    64    72
     9    18    27    36    45    54    63    72    81
```

* *Metoder* är funktioner inkapslade i ett *objekt* som för våra syften kan liknas vid en *påse*. Först måste påsen öppnas, sedan kan innehållet nås. Metoder kan endast anropas genom att först kalla på objektet och sedan anropa metoden. Objektet står *före* punkten och metoden står *efter* punkten.

Självklart kan man minska eller höja avståndet mellan utskrifterna genom att skicka större eller mindre värden än 6 som parameter till **`width()`**. Det finns även möjligheten att justera utskrifterna till vänster istället för till höger:

```
cout << left << a*b;
```

`left` är en s.k. *manipulator* som inte skriver ut något utan ändrar **`cout`**-satsens inställningar, i det här fallet från att placera högerjusterat till vänsterjusterat. *By default* placerar **`cout`** alla utskrifter högerjusterat. Även **`right`** är en **`cout`**-manipulator. *By default* betyder förinställd, automatiskt inställd eller fördefinierad. Den automatiska inställningen används, om man inte uttryckligen väljer någon annan inställning.

Övningar till kapitel 6

6.1 Skriv ett program som läser in två *tal* och skriver ut **OK** om de matats in i rätt ordning, dvs om det första är mindre än det andra. Vad händer om de är lika stora?

6.2 Skriv ett program som läser in tre tal, hittar och skriver ut det största av dem. Vilken ändring i koden leder till det minsta talet?

6.3 Följande pseudokod beskriver hur man tar på sig sjal, mössa och handskar beroende på hur kallt det är ute:

```
Start Vinterklädsel
Läs av temperaturen
OM  temperatur <  0
    ta sjal, mössa och handskar
ANNARS OM  temperatur <  5
    ta sjal och mössa
ANNARS OM  temperatur < 10
    ta sjal
ANNARS
    slipper du vinterklädsel
Slut Vinterklädsel
```

Översätt pseudokoden *Vinterklädsel* till ett C++ program med hjälp av en **if-else**-stege. Läs in ett värde för *temperatur* och låt programmet avgöra val av klädsel genom att skriva ut ”Ta …”.

6.4 Skriv ett program som läser in två *tecken* och skriver ut **OK** om de matats in i rätt ordning, dvs det första förekommer före det andra i ASCII-tabellen. Annars ska programmet skriva ut ett meddelande om att inmatning skedde i fel ordning.

6.5 Vidareutveckla övn 6.4 så att användaren får flera chanser att mata in två tecken i rätt ordning så länge han/hon matar in dem i fel ordning. Du kan göra det genom att bygga in inmatningen, bearbetningen och utmatningen i en **do**-loop.

6.6 a) Använd en loop med **while**-satsen för att skriva ut de första 10 positiva heltalen.

b) Vilken ändring ändring i koden till a) måste göras för att få fram de första 20 positiva heltalen?

6.7 a) Skriv ett program som skriver ut de första 10 *jämna* talen.

b) Modifiera a) så att endast de första 10 *udda* talen skrivs ut. (1502)

6.8 a) Skriv ett program som summerar de första 10 positiva heltalen.

b) Generalisera a) så att programmet beräknar summan av de första n positiva heltalen där n kan matas in. Testa för n = 100 och 1 000.

c) Skriv ett program som summerar de första n pos. heltalen med formeln:

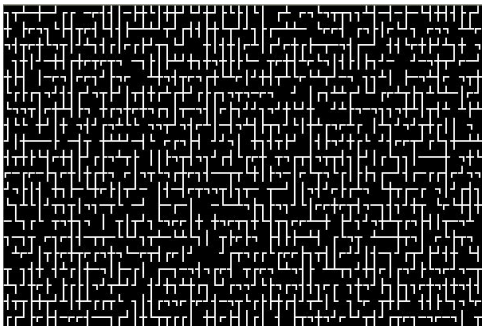
$$summa = n\,(n+1)\,/\,2$$

Testa om du får samma svar i b) och c) för n = 1 000, 5 000 och 1 000 000.

6.9 Skriv ett program som läser in ett heltal som stegvariabel för att skriva ut tal från 1 till 5000. Om steget är t.ex. 5 skrivs var femte tal ut.

6.10 Låt en **`do`**-loop producera ljudsignal tills man trycker ned någon tangent. Så länge ljudet hörs, ska ett meddelande instruera användaren för att stoppa ljudet. För att avsluta **`do`**-loopen använd funktionen **`kbhit()`** som är **`false`** från början och blir **`true`** när man trycker på en tangent. Skriv den logiska negationen **`!`** framför den för att avsluta **`do`**-loopen (sid 105). **`kbhit()`** är definierad i biblioteket **`conio.h`**.

6.11 ***Labyrint I (projekt)*** Visst är det roligt att med de verktyg vi lärt oss, låta datorn rita en labyrintartad figur på skärmen som kan se ut så här:



Och detta med hjälp av ett enkelt program som bara använder sig av den utvidgade ASCII-tabellen och C++:s slumptalsgenerator. Figuren ovan är nämligen en slumpmässigt sammansatt utskrift av mellanslaget och de 13 linjegrafiktecken (LGT) som är avbildade i inlämningsuppgiften (sid 89). Du kan gärna försöka med egna algoritmer. Här följer ett *förslag* till en algoritm som fungerar:

Förberedelser:

Gör inlämningsuppgiften (sid 89) för att få fram LGTs teckenkoder. I algoritmen nedan används en numrering av LGT som radvis följer ordningen i den på sid 89 avbildade sammanställningen av LGT. Skälet till att de avbildas i fyra olika rader är att varje rad visar deras gruppering i den utvidgade halvan av ASCII-tabellen. Bekanta dig med hantering av slumptal i C++ (sid 117).

Algoritmen:

Deklarera en teckenvariabel **`letter`** och en heltalsvariabel **`randNo`**. Låt **`randNo`** anta värden mellan 0 och 13 med satsen:

```
randNo = rand() % 14;
```

Om **`randNo`** är	0	ska	**`letter`** tilldelas	1:a LGTs teckenkod.
Om **`randNo`** är	1	ska	**`letter`** tilldelas	2:a LGTs teckenkod.
Om **`randNo`** är	2	ska	**`letter`** tilldelas	3:e LGTs teckenkod.
	.			.
	.			.
	.			.
Om **`randNo`** är	11	ska	**`letter`** tilldelas	12:e LGTs teckenkod.
Om **`randNo`** är	12	ska	**`letter`** tilldelas	13:e LGTs teckenkod.
Om **`randNo`** är	13	ska	**`letter`** tilldelas	mellanslaget.

Skriv ut **`letter`**. Det blir då antingen *ett* LGT eller mellanslaget. Testa nu om du vid varje körning får *olika* LGT eller mellanslaget.

Först när allt detta fungerar låt tilldelningen av variabeln **`randNo`** samt alla 14 **`if`** -satserna ingå i en *loop* , t.ex. en **`for`**-sats, för att rita labyrinten.

Extrauppgifter (frivilligt):

Ersätt loopen ovan med en nästlad **`for`**-sats och lägg in ett radbyte mellan den inre och yttre slingan för att kunna styra labyrintens storlek (höjd och bredd) vid varje körning. Låt användaren ange höjd och bredd.

`switch`-*variant:*

Skriv om **`if`**-satserna ovan till en **`switch`**-sats.

Fortsättning:

Labyrintprojektet kommer att vidareutvecklas som *Labyrint II* (övn 8.7, sid 177).

Kapitel 7

Funktioner

	Ämne	Sida	Program
7.1	Exempel på en funktion	127	**totalsek()**
	- Vad händer när en funktion anropas?	128	**MyFirstFct**
	- Placering av funktioner	131	
7.2	Funktionsbegreppet i programmering	132	
7.3	Definition och anrop av funktioner	135	
7.4	Deklaration av funktioner	138	**Random**
7.5	Externlagrade funktioner	141	**ExternFct**
		142	**Netto**
7.6	Funktioner utan returvärde	144	**Compare**
		145	**GuessVoid**
7.7	Lokala och globala variabler	147	**Global**
	- Blockstruktur	148	
7.8	Överskuggning av variabler	152	**Scope**
	- Räckviddsoperatorn	154	
Övningar till kapitel 7		155	

7.1 Exempel på en funktion

Det finns två typer av funktioner inom programmering:

- **Funktioner med returvärde**
- **Funktioner utan returvärde, s.k. void-funktioner**

I detta avsnitt behandlar vi den första typen. Funktioner utan returvärde, s.k. **void**-funktioner kommer att tas upp senare (sid 144). Låt oss gå tillbaka till sid 64:

```
// Hour2Sec.cpp
// Läser in tiden i timmar, minuter och sekunder, omvandlar
// allt till sekunder och skriver ut resultatet
#include <iostream>
using namespace std;

int main()
{
   int  tim, min, sek, totalsek;

   /* I n m a t n i n g */
   cout << "\nGe timmar, minuter, sekunder "
        << "skilda med mellanslag:  ";
   cin  >> tim >> min >> sek;

   /* B e a r b e t n i n g */
   totalsek = 3600*tim + 60*min + sek;

   /* U t m a t n i n g */
   cout << '\n' << tim << " timmar, "<< min<< " minuter och "
        << sek << " sekunder är " << totalsek
        << " sekunder totalt.\n\n";
}
```

Vi hade då av en viss anledning delat in koden ovan i strukturen *inmatning – bearbetning – utmatning*, vilket vi ska återkomma till. Vi vill separera *bearbetningen* från **main()** och skriva den som en funktion. Så här skulle funktionen se ut:

```
// Totalsek.h, en headerfil
// Definierar en funktion med returvärde
// Namnet: totalsek()
// Tre parametrar: t, m och s, alla av typ int
// Returtypen: int, dvs returnerar en int

int totalsek(int t, int m, int s)    // Huvudet
{                                    // Kroppen börjar
   /* B e a r b e t n i n g */
   return 3600*t + 60*m + s;
}                                    // Kroppen slutar
```

Första raden kallas för funktionens *huvud*, innehållande funktionens namn **totalsek()**, funktionens *parameterlista* **(int t, int m, int s)** och returvärdets data-

typ **int**, kallad *returtypen*. Inom måsvingarna som avgränsar funktionen som ett block står funktionens *kropp*. Den i sin tur består av en enda sats som inleds med det reserverade ordet **return**, kallad **return**-sats, som returnerar *returvärdet*. Uttrycket efter **return** beräknar totalsekunderna med hjälp av *parametrarna* t, m och s. Att vår funktion **totalsek()** innehåller en **return**-sats är anledningen till varför den är en *funktion med returvärde*, dvs den första av de två typer som nämndes inledningsvis. Alla dessa begrepp kommer att preciseras allt eftersom.

Funktionen **totalsek()** kan kompileras när den infogas i ett projekt i Visual Studio. Men den kan inte exekveras därför att den inte är ett *program:* **main()**, exekveringens startpunkt finns inte. Vi måste lägga in den i ett program med **main()**.

Så här gör vi för att bygga in funktionen **totalsek()** i ett program med **main()**:

```
// MyFirstFct.cpp
// Bearbetningen har flyttats till funktionen totalsek() som
// placeras före main() och anropas från main().
// In- och utmatning görs fortfarande i main().
#include <iostream>
using namespace std;
/*************************************************************/
int totalsek(int t, int m, int s) // Definition av funktion
{
   /* B e a r b e t n i n g */
   return 3600*t + 60*m + s;
}
/*************************************************************/
int main()                          // Här startar exekveringen
{
   int tim, min, sek;

   /* I n m a t n i n g */
   cout << "\n\tGe timmar, minuter, sekunder "
        << "skilda med mellanslag:  ";
   cin  >> tim >> min >> sek;

   /* U t m a t n i n g */
   cout << "\n\t" << tim << " timmar, " << min << " minuter "
        << "och " << sek << " sekunder är "
        << totalsek(tim, min, sek) << " sekunder totalt.\n";
}          // Anrop av funktion
```

Nu kan vi både kompilera och exekvera programmet ovan. Vid exekveringen anropas funktionen **totalsek()** i den sista raden av **main()**. Vad betyder det?

Vad händer när en funktion anropas?

Tre saker händer när funktionen **totalsek()** anropas från **main()**:

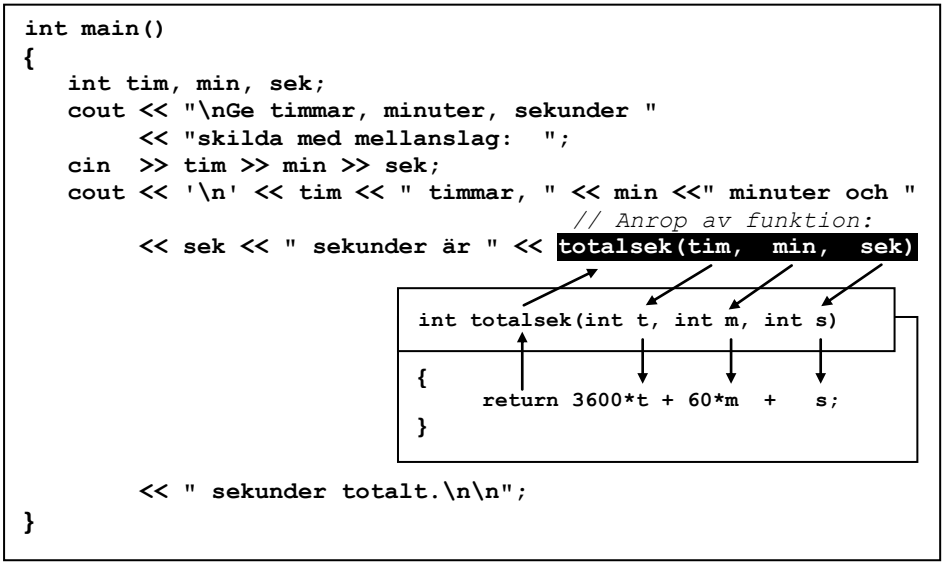
1. **Parameteröverföring** Då överförs de aktuella parametrarna **tim, min, sek**:s värden som lästs in före anropet, till de formella parametrarna **t, m, s**. Det finns olika mekanismer för parameteröverföring beroende på parameterns

datatyp vilket tas upp senare. I vårt exempel kopieras värdena från **tim, min, sek** till **t, m, s**. För de aktuella parametrarna har genom definitionssatsen **int tim, min, sek;** tre minnesceller reserverats i början av **main()**. För de formella parametrarna har tre andra minnesceller reserverats i **totalsek()**:s parameterlista via definitionerna **int t, int m, int s**. Observera att korresponderande parametrar borde vara av samma datatyp. Om vi t.ex. matar in **5 35 49** när programmet körs, tilldelas dessa värden variablerna **tim, min, sek**. Funktionsanropet vidarebefordrar värdena till funktionens formella parametrar **t, m, s**. Så hamnar de inmatade värdena i kroppen till **totalsek()**.

2. **Exekvering av funktionskroppens kod** vilket innebär att **return**-satsen utförs dvs i vårt fall uttrycket **3600*t + 60*m + s** beräknas. Med inmatningen från punkt **1** blir det **3600*5 + 60*35 + 49**. Resultatet **20149** tilldelas funktionsnamnet **totalsek**. Mer finns inte just i vårt exempel.

3. **Överföring av returvärdet** sker i omvänd riktning jämfört med parameteröverföringen, nämligen från den anropade funktionen **totalsek()** till den anropande funktionen **main()**. Vi får tillbaka returvärdet från funktionen, i exemplet är det variabeln **totalsek**:s värde dvs strängen **20149** med inmatningen från punkt **1**. Att returvärdet hamnar i **main()** beror på att anropet görs med funktionsnamnet **totalsek** som är identiskt med returvärdet.

Det som händer vid anrop av en funktion är alltså att data byts ut mellan den anropande och den anropade funktionen, i vårt exempel mellan **main()** och **totalsek()**, att koden i den anropade funktionen utförs när anropet inträffar, samt att returvärdet till sist hamnar i den anropande funktionen. En översikt över dataflödet mellan dessa två funktioner (se pilarna nedan) och framför allt i *vilken ordning* deras koder utförs ges i följande bild som illustrerar punkterna **1-3** ovan:



Bilden ovan visar vad som *händer* när programmet **`MyFirstFct`** (sid 128) exekveras: Funktionen **`totalsek()`**:s kod hämtas och utförs (på bilden: klistras in) på det stället där funktionens anrop står (på bilden: framhävt med svart bakgrund). Det är skillnad mellan kod och handling. Bättre sagt: ordningen i koden – låt oss kalla det *dokumentationen* – är annorlunda än det som händer när koden körs – låt oss kalla det *aktionen*. Aktionen börjar som vanligt med exekveringen i **`main()`** och fortsätter tills den kommer till anropet av funktionen. Punkterna **1**-**3** ovan utförs vilket visas bl.a. med pilarna. Sedan fortsätter aktionen med den kod i **`main()`** som står efter anropet. **`main()`**:s och **`totalsek()`**:s aktioner är nästlade i varandra. Dokumentationen har en helt annan struktur. De båda funktionernas koder är inte alls nästlade i varandra, snarare tvärtom, de är isolerade från varandra. Och så måste det vara: Koden till funktionen **`totalsek()`** får inte under några omständigheter skrivas *i* **`main()`**, den måste stå *utanfö*r **`main()`**, antingen *före* eller *efter* **`main()`**, ja den kan t.o.m. ligga i en *separat* fil. Men när vi kör programmet i sin helhelt händer saker och ting i den ordning som visas på bilden.

Bilden på förra sidan visar också dataflödet som går från de aktuella parametrarna **`tim`**, **`min`**, **`sek`** till de formella parametrarna **`t`**, **`m`**, **`s`**. Dessa bearbetas i funktionen **`totalsek()`** dvs **`return`**-uttrycket beräknas. Sedan skickas resultatet av beräkningen som returvärde via funktionsnamnet till **`main()`**. Vid exekvering utförs funktionskroppens kod, precis på det stället där funktionsanropet i **`main()`** förekommer. Vid denna process spelar funktionshuvudet **`int totalsek(int t, int m, int s)`** rollen av ett *gränssnitt* mellan **`main()`** och **`totalsek()`**. Det är där kommunikationen mellan dessa separata moduler äger rum. Därför har vi satt den i en extra ruta för att understryka denna roll. Huvudet är nämligen åtkomligt både från **`main()`** och **`totalsek()`**. De skulle annars inte kunna kommunicera med varandra då de ligger i olika block. Bilden ska nämligen även visa att funktionsanropet resulterar i en *blockstruktur* som återges av ramarna i bilden. Motsvarande kod till denna blockstruktur utgörs av klamrarna **`{ }`** till både **`main()`** och **`totalsek()`**. Klamrarna bildar dessa modulers fasta gränser för kodens giltighet eller räckvidd. För att överskrida dem måste vissa regler beaktas vilket vi kommer att precisera senare. Blockstrukturen är den egentliga orsaken till att funktion **`totalsek()`** inte kan kommunicera med **`main()`** annat än via funktionshuvudet som gränssnitt. Det motiverar dessutom varför funktionerna inte får nästlas i varandra utan måste kodas separat. Mer om blockstruktur kommer att tas upp på sid 148. Inledningsvis hade vi nämnt block på sid 96.

Ett körexempel av programmet **`MyFirstFct`** kan se ut så här:

```
Ge timmar, minuter, sekunder skilda med mellanslag:  5 35 49

5 timmar, 35 minuter och 49 sekunder är 20149 sekunder totalt.
```

Placering av funktioners definition

När vi pratar om funktioners placering menar vi placeringen av deras *definition*. Kan man definiera funktioner var som helst i ett C++ program? I vårt fösta exempel (sid 127) placerade vi funktionen **`totalsek`** före **`main()`**. Det är också den naturliga platsen för en funktion så länge man håller på med att utveckla och testa den. Men nu, när vi är klara med utvecklingsstadiet, vill vi titta på alternativ. I större program brukar man skriva och testa sina funktioner en i taget. När man är klar med dem, vill man helst lägga dem ”åt sidan” och få upp, när man öppnar programfilen, högst på sin skärm endast den funktion som man aktuellt håller på med att utveckla och testa. ”Åt sidan” innebär antingen efter **`main()`** eller i en separat fil, vilket behandlas i de efterföljande avsnitten.

Men innan vi går vidare ska vi sammanfatta det vi sagt i följande regel om funktioners placering:

> En funktions definition får inte placeras inuti en annan funktion och därmed inte heller i **`main()`**.

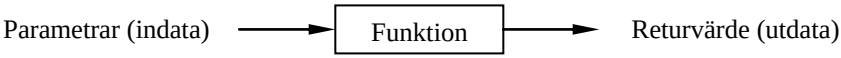
Denna regel borde vara en självklarhet med tanke på att syftet med funktioner är *modularisering*. För att en modul (funktion) ska kunna användas (anropas) i alla program får den inte nästlas i en annan modul. Därför en Lego-bit alldrig konstruerad i en annan Lego-bit (sid 133). För att fullt förstå detta läs nästa avsnitt.

7.2 Funktionsbegreppet i programmering

Begreppet *funktion* härstammar från matematiken. Man har en formel som beräknar ett värde, säg y, utgående från ett annat, säg x, och skriver y = f(x). y är en funktion av x. Denna matematiska syn på funktion har tagits över till programmering som ett underliggande koncept och historisk utgångspunkt. Men under tiden har begreppet vidareutvecklats och fått en bredare tolkning då den tillämpats på all datoriserad problemlösning. Inom programmering inkluderar funktioner matematiska problem utan att vara begränsade till sådana.

> I ett datorprogram är en funktion ett antal satser som definieras som en separerat modul utanför **`main()`**. De utförs när funktionen anropas. Vid anropet kan funktionen ta emot indata, s.k. parametrar, bearbeta dem och returnera utdata, s.k. returvärde.

En funktion kan även jämföras med en (”svart”) låda i vilken man stoppar in indata och får ut utdata: Indata kallas även *parametrar* och utdata *returvärde*:



En funktion kan ha 0, 1 eller flera parametrar. Den kan ha 0 eller 1 returvärde. En funktion kan inte ha flera returvärden. Både parametrarna och returvärdet kan vara tal, tecken, strängar, sanningsvärden eller andra datatyper. Funktionen bearbetar de inkommande parametrarna och returnerar endast *ett* värde eller inget alls. En funktion utan returvärde kallas för **`void`**-funktion. Som separat och namngiven modul kan en funktion anropas i **`main()`**, men även även i andra program. I denna bemärkelse är en funktion ett *underprogram* (eng. *subroutine*).

”Svart” är lådan så länge vi inte själva definierat funktionen och därför inte vet hur den fungerar ”inuti”. Vi använder den endastför att lösa ett visst problem. Det gäller t.ex. för de biblioteksfunktioner som vi hittills använt i våra programexempel: **`sizeof()`**, **`_getch()`**, **`rand()`**, **`srand()`** och **`time()`**. De är förprogrammerade och lagras i biblioteksfiler. Vi behöver bara inkludera dem i våra program för att dra nytta av deras funktionalitet utan att behöva veta hur de i detalj är konstruerade. Biblioteken i alla programmeringsspråk består av sådana ”svarta” lådor.

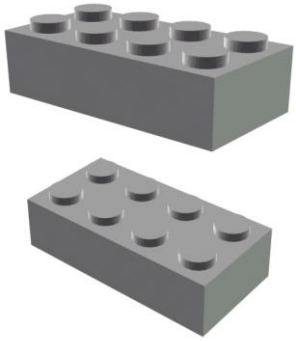
Den enda funktion som vi hittills definierat själva – och det har vi gjort i *alla* våra programexempel – är **`main()`**. Den är unik därför att utan **`main()`** som programmets exekveringspunkt händer ingenting (sid 34). I alla procedurala programspråk bildar funktioner programmets byggstenar (moduler). Att C++ som är objektorienterat även bygger på funktioner och inte bara på klasser, beror på språkets rötter i C som är ett proceduralt, men inte objektorienterat språk. C# och Java t.ex. bygger endast på klasser som inkapslar sina funktioner och döper dem, när de definieras i klasser, till *metoder*.

Varför funktioner?

Frågan är berättigad för nybörjare: Kan man inte helt enkelt skriva kod rakt ned i `main()`? Är detta med funktioner inte att krångla till det hela?

Föreställ dig en verksamhet som växer med tiden, ett expanderande företag eller en organisation med stigande antal medlemmar. Hur organiserar man jobbet? Man gör arbetsdelning. Man delegerar uppgifterna. Var och en får en väl definierad arbetsuppgift. Annars skulle man inte kunna klara av jobbet. Samma sak gör man med program vars kod växer. Det händer när man utvecklar programmet efter behov och behoven blir större och större med tiden. Lösningen är: man delar upp det stora programmet i mindre, logiskt meningsfulla delproblem, för att kunna klara av komplexiteten. Hur det görs ska vi nu diskutera under rubrikerna: *Modularisering eller Lego-principen*, *återanvändning av kod* och *strukturering av program*.

Modularisering eller Lego-principen

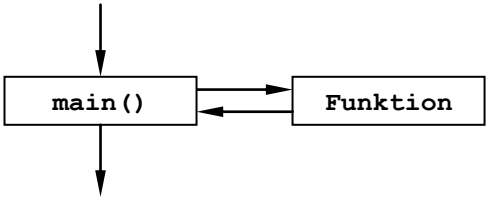


De flesta har väl någon gång som barn, eller tillsammans med sina barn, byggt ett hus, en bil eller liknande med Lego-bitar. Efter ett tag har huset kanske rasat och nya tekniska underverk har konstruerats. Men även de har någon gång plockats isär. Det enda som blivit kvar är själva Lego-bitarna som man så småningom samlat i en kartong för att kunna återanvända dem senare.

Vill man lösa ett komplext problem, t.ex. bygga ett hus eller en bil, bryter man ned det i ett antal mindre problem som är enklare att lösa. Sedan sätter man ihop de små enkla lösningarna till den stora komplexa lösningen. Principen heter *modularisering* och kan användas vid nästan all problemlösning. Ett stort komplext problem bryts ned i mindre *moduler* – motsvarande Lego-bitarna – och bearbetas en i taget. I programmering är dessa moduler ingenting annat än *funktioner*. Stora program bryts ned i ett antal funktioner. Varje funktion löser ett visst delproblem som är oberoende av andra och antagligen enklare att koda än det stora programmet. Sedan gäller det att sätta ihop modulerna till det stora programmet.

För att kunna göra det måste varje modul kommunicera med sin omgivning. Även här kan man lära av Lego: Varje Lego-bit är konstruerad så att den passar in i en annan Lego-bit. De delar av Lego-biten som tillåter denna passning, kan anses som Lego-bitens *gränssnitt* mot andra Lego-bitar. På samma sätt har en funktion ett gränssnitt mot andra funktioner för att kunna kommunicera med dem. Även detta gränssnitt har två delar: För det första funktionens *parametrar* som importerar värden från omgivningen och för det andra funktionens *returvärde* som exporterar ett värde till omgivningen. Men sedan måste Lego-bitarna ”sättas ihop” vilket i programmeringstermer innebär att *anropa* den ena från den andra. Ett *anrop* av en funktion innebär att *aktivera* funktionen. Detta sker genom att ev. skicka till den parametrar, utföra koden som står i funktionen och ev. få tillbaka returvär-

det. Generellt finns det i ett program flera funktioner som anropar varandra. Det enklast tänkbara exemplet är att **`main()`** anropar en **`Funktion`** dvs **`main()`** är den *anropande* och **`Funktion`** den *anropade* funktionen. Då kan programflödet mellan dem se ut så här:



Återanvändning av kod

är det andra svaret på frågan varför man i programmering sysslar med funktioner. Samma idé finns bakom Lego-biten som minsta återanvändbara modul för att bygga i princip vad som helst. Har man i ett program löst ett litet delproblem som även dyker upp i andra sammanhang och vars kod kan vara relevant i andra program, så vill man ju helst inte satsa tid och resurser för att koda det en gång till. Man vill undvika att återuppfinna hjulet. Detta är inte bara av teoretiskt-estetiskt intresse utan även av stort ekonomiskt intresse. Det man gör är att lösa koden för det lilla delproblemet från det aktuella programmet och skriva den som en funktion för att kunna återanvända koden i vilket annat program som helst. Man behöver då endast anropa den från andra program. Det kräver förstås att den ursprungliga koden som kanske var skräddarsydd för just det speciella programmet då, nu som funktion måste formuleras på ett mer generellt sätt och förses med möjligheten att kunna kommunicera med andra program. Därför måste koden kompletteras med parametrar och returvärdet. Hela tanken bakom standardbibliotek – inte bara i C++ utan i alla programspråk – bygger på idén om återanvändning av kod. Även om man väljer att inte skriva egna funktioner kan man i alla fall inte komma ifrån att använda redan fördefinierade funktioner från standardbiblioteket.

Strukturering av program

är det tredje svaret på frågan varför funktioner, närmare bestämt egendefinierade funktioner, används i programmering. Genom att modularisera ett komplext problem som ska lösas med hjälp av datorn underlättar man inte bara själva lösningen (innehållet) utan kan även lättare få en strukturering av programkoden (formen). Det enklast tänkbara sättet att strukturera vilket program som helst är t.ex. att dela in det i ***inmatning – bearbetning – utmatning*** (sid 64). Dessa tre delar kan skrivas i var sin funktion vilka sedan anropas av **`main()`**. Denna huvudfunktion kan då bestå av ett få antal satser som endast anropar programmets olika funktioner. På så sätt har man från **`main()`** en övergripande kontroll över hela programflödet. Dessutom kan funktionerna lagras i separata filer och inkluderas med **`#include`**-direktiv i den fil som innehåller **`main()`**. Så kan man så småningom bygga upp sitt eget bibliotek av egendefinierade funktioner.

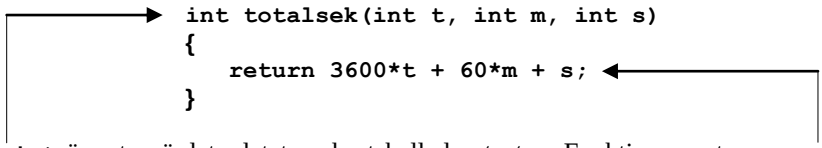
7.3 Definition och anrop av funktioner

Vi ska nu i detalj gå igenom alla steg hos funktioner *med* returvärde. I avsnitt ***7.6 Funktioner utan returvärde*** kommer den andra typen av funktioner, s.k. **`void`**-funktioner, att behandlas (sid 144).

Definition av en funktion med returvärde

```
returtyp Funktionsnamn(datatyp fpar1, datatyp fpar2, ...)
{
    sats(er);
    return uttryck;
}
```

Med returtyp menas datatypen till *returvärdet* och fpar står för *formell parameter*. Så kallas parametrar som förekommer i funktionens definition. Första raden är *funktionshuvudet* inklusive parentesen **`(`** ... **`)`** som innehåller listan över alla parametrar, därför kallad *parameterlistan*. Den kan innehålla en eller flera parametrar, men kan även vara tom. Oavsett antalet parametrar inkluderar man alltid, när man i beskrivande text nämner en funktion, parentesen **`()`** och lägger den till funktionsnamnet. Det gör man generellt för att skilja mellan funktioner och variabler. Parentesen är alltså kännetecknet för en funktion. Observera att funktionshuvudet inte avslutas med semikolon. Det är inte en sats utan bara huvudet till en funktion vars kropp följer. I programexemplet **`MyFirstFct`** är funktionen **`totalsek()`** definierad på följande sätt:



Detta **`int`** är returvärdets datatyp, kort kallad *returtyp*. Funktionen returnerar ett heltal. Men varför står returtypen framför funktionsnamnet? Det verkar – om man för ett ögonblick bortser från parameterlistan – som om **`int totalsek`** vore en deklarqtion av ”variabeln” **`totalsek`** till datatypen **`int`**. Denna tolkning är korrekt om man tar hänsyn till att **`totalsek`** är både funktionsnamn och returvärdets minnescell. Denna minnescell kan få sitt värde endast från **`return`**-satsen, när funktionen anropas. Medan returvärdet är funktionens output (utdata) är parametrarna funktionens input (indata). **`totalsek()`** har tre parametrar av typ **`int`** som vi döpt till **`t, m, s`**. De är definierade i parameterlistan: **`(int t, int m, int s)`**. Observera att det inte går att skriva **`(int t, m, s)`** som man kan göra vid definition av vanliga variabler. Visserligen är parametrar också variabler, men när de definieras i parameterlistan, måste de upprepa sina datatyper även om de är av samma typ. Det är den enda syntaktiska skillnaden mellan parametrar och vanliga variabler. I definitionen heter de *formella* parametrar därför att de definieras i parameterlistan som ”tomma” minnesceller i väntan på att bli fyllda med värden när funktionen an-

ropas. Deras namn saknar betydelse – bara man använder konsekvent samma namn i funktionskroppen. De *initieras* dvs tilldelas värden *första gången* när funktionen anropas. Vid anropet *importerar* **t, m, s** de erhållna värdena till funktionen dvs vidarebefordrar dem endast från **main()** till funktionen där de bearbetas. Returvärdet *exporterar* sedan resultatet ur funktionen till **main()**.

I *kroppen* till en funktion kan ett antal satser stå avgränsade med klammerparet **{ … }**. Klamrarnas uppgift är att gruppera satserna under funktionshuvudet till ett block. I funktionen **totalsek()** består detta block av en enda sats som kallas **return**-*satsen*. Denna sats returnerar uttryckets värde till funktionsnamnet. Men **return**-satsen gör en sak till: Den avslutar även funktionen. Eventuell kod efter den kommer att inte utföras. Därför ska **return**-satsen alltid vara funktionens sista sats. Den logiska slutsatsen är att det får finnas endast en **return**-sats i en funktion. Då **return**-satsen returnerar uttryckets värde till funktionsnamnet, måste namnet ha beredskapen at ta emot det. Detta innebär att **totalsek** samtidigt som det är funktionsnamnet, också är en variabel av typ **int** – med den begränsningen att den endast kan få sitt värde från **return**-satsen. **Totalsek** kan inte lagra returvärdet som är ett haltal, om det inte är via returtypen definierat till datatypen **int**.

Anrop av en funktion med returvärde

Funktionens definition är endast är en *mall*, en *föreskrift* om vad som *skulle* hända om funktionen anropas, jämförbar med ett matrecept som man skriver ned och stoppar i kökskåpets låda i väntan på att någon gång ta fram det och laga mat. Först när beredskapen till matlagning finns – alla ingredienser är handlade och finns på plats – kan matreceptet komma till användning som en algoritm. Samma sak är det med funktionens definition: den är endast en potentiell eller formell kod. Aktuell blir den först när vi anropar funktionen. Då börjar saker och ting att hända. I programmet **MyFirstFct** anropas funktionen **totalsek()** från funktionen **main()** i följande sats:

```
cout << … << totalsek(tim, min, sek) << … ;
```

Själva anropet består av den gråmarkerade koden dvs av att kalla funktionen vid namn och i parameterlistan skriva *lika många* parametrar som definitionen föreskriver – i det här fallet tre. Parametrar som förekommer i funktionens anrop – här **tim**, **min**, **sek** – kallas *aktuella*. Antalet aktuella parametrar *måste* vara lika med antalet formella parametrar – de som förekommer i funktionens definition. Annars blir det kompileringsfel. Samtidigt *borde* de aktuella parametrarna ha samma datatyp som de formella. Annars försöker kompilatorn att göra automatisk typkonvertering till måldatatypen. Dessutom måste vi se till att parametrarnas *ordning* stämmer dvs vi måste kontrollera om vi verkligen vill skicka **tim** till **t**, **min** till **m** och **sek** till **s** för exakt i den ordning vi skriver dem i anropet, kommer deras värden att överföras till de formella parametrarna i funktionens definition. Vi har döpt dem till **tim**, **min**, **sek**, definierat dem i **main()** som **int**-variabler och tilldelat dem värden via inläsning med **cin**-satsen strax före anropet.

Att anropet kan läggas i **cout**-satsen ovan beror på att **totalsek()** är en funktion *med returvärde* – dvs har en **return**-sats – och funktionsnamnet därmed samtidigt bär returvärdet i sig. I exemplet bakar vi in anropet i **cout**-satsen för att skriva ut returvärdet direkt på skärmen. Andra alternativ är att lägga anropet i en tilldelningssats till höger om tilldelningstecknet och låta en variabel ta hand om returvärdet eller anropa funktionen genom att kalla på namnet i en sats utan att ta hand om returvärdet:

```
cout <<    funktionsnamn(apar1, apar2, ...);
```

eller

```
variabel = funktionsnamn(apar1, apar2, ...);
```

eller

```
funktionsnamn(apar1, apar2, ...);
```

där **apar** står för aktuell parameter och **funktionsnamn** är den anropade funktionen. Självklart måste **variabel** och även alla aktuella parametrar vara definierade *före* anropet på vanligt sätt i den anropande funktionen – i vårt exempel i **main()**.

7.4 Deklaration av funktioner

I vårt första program om funktioner i förra avsnitt placerade vi den funktionen **totalsek** *före* **main()** dvs funktionens definition kom *före* anropet i **main()**. Detta behöver inte alltid vara så med tanke på att en funktion ska helst vara en separat modul som kan användas i andra program. Man kan i C++ placera funktionen även *efter* **main()** som en led i processen att isolera funktionen helt och hållet och placera den t.o.m. i en separat fil som en externlagrad funktion. För att kunna göra det behöver vi begreppet *deklaration av en funktion* som introduceras i följande exempel. Observera den lilla, men avgörande skillnaden mellan *definition* och *deklaration* av en funktion!

```
// Random.cpp
// Gör samma sak som programmet NestedFor
// In- och utmatning i tabellform görs i main()
// Slumptalsgenerering har flyttats till en funktion
// Funktionen deklareras i main() och definieras efter main()
#include <iostream>
#include <ctime>                // Krävs för funktionen time()
using namespace std;
/*************************************************************/
int main()
{
   int myRand(int, int);        // Deklaration av funktion

   srand(time(0));              // Skapar variation i slumpen
   int r, k, rader, kolumner;
   cout << "\nAnge antal rader och kolumner (t.ex. 20 15): ";
   cin >> rader >> kolumner;
   cout << "\nDet blir "<< rader*kolumner <<
                                      " tärningskast:\n\n";
   for (r=1; r<=rader; r++)
   {
       for (k=1; k<=kolumner; k++)
           cout << myRand(1, 6) << "   ";// Anrop av funktion
       cout << '\n';
   }
   cout << '\n';
}
/*************************************************************/
int myRand(int a, int b)         // Definition av funktion som
{                                // slumpar fram heltal i in-
   if (a < b)                    // tervall mellan a och b
      return a + rand() % (b - a + 1);
   else
      return b + rand() % (a - b + 1);
}
/*************************************************************/
```

En förklaring till formlerna i funktionen **myRand()** hittar man på sid 118.

Strax i början av **main()** står deklarationen till funktionen **myRand()**:

```
int myRand(int, int);
```

myRand() anropas i **main()** i varje varv av **for**-satsen medan definitionen av **myRand()** står efter **main()**. Om vi inte definierar funktionen i början stöter kompilatorn som går igenom koden rad för rad, *först* på funktionsanropet i **main()** *innan* den hittar funktionsdefinitionen. I så fall kan programmet inte kompileras för att kompilatorn inte kan tolka funktionsanropet. Deklarationen är i själva verket en förhandsinformation till kompilatorn som talar om att det kommer att anropas en funktion som:

1. heter **myRand()**
2. returnerar ett värde av typen **int**
3. har 2 parametrar, båda är av typen **int**

Denna information är både tillräcklig och nödvändig för att kunna kompilera funktionsanropen i programmet **Random**. Vad den talar om är att så här ser funktionen ut som kommer att anropas. Vad anropet innebär kommer att definieras senare. Pga deklarationens roll som en förhandsinformation pratar man ibland på engelska om *forward declaration*. I litteraturen finns även beteckningen *prototyp* av en funktion.

Signaturen

Deklarationens ovannämnda egenskaper 1 och 3 – funktionens namn, antalet parametrar och parametrarnas datatyper – kallas för funktionens *signatur* dvs dess igenkänningstecken. Det är anmärkningsvärt är att returtypen inte ingår i signaturen. Dvs i exemplet ovan är funktionens signatur:

```
myRand(int a, int b)
```

Signaturen är alltså en del av deklarationen. Funktioner med samma signatur anses vara *identiska*. Funktioner som skiljer sig på någon av punkterna 1 eller 3 ovan, dvs namnet eller parameterlistan anses vara *olika*. Denna distinktion blir avgörande när man har att göra med *överlagrade funktioner*. Det är funktioner som har samma namn, men olika parameterlistor, antingen olika antal parametrar eller olika datatyper till sina parametrar.

Hur deklarerar man funktioner?

Att deklarera en funktion – eller att skriva en prototyp – är väldigt enkelt: Man kopierar funktionshuvudet från definitionen och avslutar den med semikolon så att det uppstår en sats. Semikolonet är den avgörande skillnaden till funktionshuvudet, visar att deklarationen är en sats som måste avslutas med semikolon, medan funktionshuvudet är en rubrik som inleder funktionens definition. I så fall borde i exemplet ovan deklarationen skrivas så här: **int myRand(int a, int b);** Om man vill kan man göra det. Men man kan också utelämna parametrarnas namn **b** och **m** vilket vi har gjort i **Random**. Detta för att öka programmets läslighet genom att bättre kunna skilja en deklaration från ett funktionshuvud. Semikolonet kan lätt förbises vid hastig läsning. Däremot är det mer iögonfallande att parameternamnen

fattas vilket indikerar att man har att göra med en deklaration. För kompilatorn gör det ingen skillnad, den ignorerar alla parameternamn i deklarationen. Det är anmärkningsvärt att det inte ens blir kompileringsfel om man väljer helt andra namn på parametrarna i deklarationen än senare i funktionens anrop. Redan för att undvika godtyckligheten vid val av namn borde det vara rimligt att utelämna parametrarnas namn i deklarationen av en funktion. Huvudskälet för oss är dock bättre läslighet. Därför skriver vi i exemplet ovan deklarationen så här:

```
int myRand(int, int);
```

Deklarationen kan placeras på två ställen: antingen *före* **`main()`** eller *i* **`main()`** som vi gjort. En deklarationssats *före* **`main()`** alltså på samma plats som **`include`**-direktiven, skulle gälla inte bara i **`main()`** utan även i alla funktioner som ev. följer. Däremot en deklarationsats *i* **`main()`** gäller endast i **`main()`**. Med deklarationens giltighet syftar vi åt den plats i koden därifrån man anropar funktionen. I **`Random`** är det **`main()`**. Därför har deklarationssatsen placerats *i* **`main()`**.

Vad gäller själva slumptalsgenereringen i funktionen **`myRand()`** hänvisas till sid 117 där hantering av slumptal behandlas.

Ett körexempel av programmet **`Random`** ger ”samma” resultat som programmet **`NestedFor`** (sid 118), det första *med* och det andra *utan* funktioner. ”Samma” förstås om man bortser från att varje körning ger andra slumptalsvärden:

```
Ange antal rader och kolumner (t.ex. 20 15): 20 15

Det blir 300 tärningskast:

5   6   6   6   3   6   3   3   3   6   3   5   6   1   2
4   4   6   1   6   4   3   2   4   3   1   3   1   1   6
1   2   2   5   2   5   5   3   1   2   4   4   3   1   3
2   2   1   4   4   3   6   2   5   5   3   6   5   3   2
1   3   3   5   6   1   6   3   5   1   4   3   6   2   2
4   2   2   3   5   5   4   6   5   3   1   3   4   5   4
3   5   5   4   6   3   6   1   6   1   6   4   6   6   5
3   6   6   3   4   4   4   3   6   4   3   6   5   6   4
2   2   4   6   5   6   6   6   1   5   4   2   6   1   3
2   6   6   4   5   3   6   6   2   3   6   6   6   4   6
5   1   5   2   5   3   4   1   1   3   2   4   1   5   1
1   4   4   6   6   6   2   5   1   1   2   3   5   4   3
6   3   1   1   5   3   4   5   2   6   3   2   4   2   5
1   4   2   5   6   6   3   5   5   1   3   1   2   4   6
3   4   4   3   6   3   4   4   6   6   3   4   2   5   4
6   4   3   6   3   1   4   5   1   2   6   6   6   3   5
4   3   4   5   3   4   1   2   5   1   3   3   4   3   4
1   5   1   6   4   5   2   2   6   4   2   3   2   4   5
4   3   2   3   1   5   6   5   6   6   6   5   1   2   4
4   1   1   3   3   5   5   3   2   3   2   4   6   2   3
```

7.5 Externlagrade funktioner

Prototypen som behandlades i förra avsnitt var en led i processen att isolera en funktion och placera den i en separat fil som en *externlagrad* funktion. Man vill bygga separata moduler som kan användas även av andra program. Så länge en funktion är inbunden i samma fil som det program som anropar funktionen kan den inte användas av andra program. Man vill bryta ned program i oberoende moduler för att kunna dra nytta av återanvändning av kod (Modularisering sid 133). Detta leder till att man löser den anropade funktionen helt och hållet från den fil som innehåller den anropande funktionen, här **`main()`**, och lagrar den i en annan fil. Sedan måste modulerna länkas ihop. Ett exempel på detta finns i följande program:

```
// ExternFct.cpp
// Skriver ut en tabell över nettobelopp och ingående moms
// Programmet består av två funktioner:
// 1) main() matar in och ut samt anropar funktionen netto()
// 2) netto() som är externlagrad beräknar nettobeloppet
#include <iostream>
#include <iomanip>           // Krävs för setprecision()
using namespace std;

#include "Netto.h"           // Innehåller funktionen netto()

int main()
{
   float first, last, step, vatRate, brutto;
   cout << "\nTabell över brutto- och nettobelopp samt "
        << "ingående moms:\n";
   cout << "\n\tBruttobelopp för början av tabellen:\t";
   cin >> first;
   cout << "\toch för slutet av tabellen:\t\t";
   cin >> last;
   cout << "Ange steget för momstabellen    : ";
   cin >> step;
   cout << "Aktuell momssats i % (t.ex.25)  : ";
   cin >> vatRate;
   cout << fixed << setprecision(2);  // Efter detta kommer
                    // alla cout-satser skriva ut tal med
                    // 2 fasta decimaler
   cout << "\nBrutto\t\tNetto\t\tIngående moms\n"
        << "-----------------------------------------\n";
   for (brutto=first; brutto<=last; brutto=brutto+step)
                                    // 2 anrop av netto():
       cout << brutto << "\t\t" << netto(brutto, vatRate)
            << "\t\t" <<  brutto - netto(brutto, vatRate)
            << "  kr\n";
   cout << "\n";
}
```

`include`-direktivet **`#include "Netto.h"`**

som står strax före **main()** gör att kompilatorn hittar filen **Netto.h**, även kallad **h***eaderfil*. Den innehåller definitionen av funktionen **netto()**. Filens ändelse **h** i de externlagrade funktionerna är inte obligatorisk utan endast en konvention som ska påminna om att det i dessa filer lagras funktioner som inte utgör kompletta program utan *delar* av fullständiga program. Syntaxen för inkludering av **h**-filer skiljer sig från syntaxen för biblioteksfilerna: Filens namn skrivs inom citationstecken för att skilja den från standardbiblioteksfiler. Kompilatorn ska inte söka den i den mapp som är avsedd för biblioteksfilerna utan i samma mapp som programfilen **ExternFct.cpp**. I detta fall måste vi placerat headerfilen i samma mapp som programfilen. Men det är även möjligt att placera den i en annan mapp och ange en korrekt sökväg i **include**-direktivet. Så här ser headerfilen ut:

```
// Netto.h
// Beräknar nettobeloppet utgående från ett bruttobelopp
// Definierar funktionen netto() som tar in ett bruttobelopp
// b samt momssatsen m och returnerar nettobeloppet
// netto() anropas från main() lagrad i filen ExternFct.cpp
// Inget fullständigt program, därför ändelsen h (headerfil)

float netto(float b, float m)        // Funktionsdefinition
{
   return 100*b/(100+m);             // Formeln för beräkning
}                                    // av nettobelopp
```

De formella parametrarna **b** och **m** kommer att ta emot sina värden från de aktuella parametrarna **brutto** och **vatRate** när **main()** anropar funktionen **netto()**. Funktionen beräknar sedan uttrycket i **return**-satsen och returnerar resultatet till **main()** som är placerad i programfilen **ExternFct.cpp**. Att funktionerna hittar varandra trots att de är placerade i olika filer, beror på att **h**-filen är kopplad till **cpp**-filen via **include**-direktivet. Programmet ovan består av *två* funktioner som anropar varandra: **netto()** och **main()** lagrade i två separata filer.

Vad programmet **ExternFct** i sin helhet gör är att läsa in önskade bruttobelopp för början och slutet på en tabell. Dessutom ska även ett steg anges som tillämpas i tabellen. Tabellen skriver ut givna bruttobelopp i en första kolumn och till varje bruttobelopp det beräknade nettobeloppet enligt en viss momssats som också läses in i början. Nettobeloppen hamnar i den andra kolumnen. Dessutom beräknas och skrivs ut i en tredje kolumn den moms som ingår i varje bruttobelopp. Huvudjobbet görs i en **for**-sats som i varje varv anropar funktionen **netto()** två gånger, första gång för att skriva ut nettobeloppet i den andra kolumnen, andra gång för att skriva ut den ingående momsen i den tredje kolumnen genom att subtrahera den från bruttobeloppet. Med tabulatorn **\t** justeras avstånden mellan kolumnerna. **\n** i slutet av **for**-satsens **cout**-sats åstadkommer radbyte i tabellen. **cout**-satsen direkt innan **for**-satsen skriver ut tabellhuvudet. Kvarstår hur vi får att alla värden i tabellen skrivs ut med två decimaler. Lösningen är:

```
cout << fixed << setprecision(2);
```

fixed är en manipulator som ändrar **cout**-strömmens inställningar så att alla decimaltal skrivs ut med ett fast antal decimaler, by default (automatiskt) med 6. **Setprecision(2)** upphäver denna defaultinställning och sätter antalet decimaler till 2. **setprecision()** som kräver inkludering av **iomanip**, sätter antal *siffror* utan **fixed** och antal *decimaler* med **fixed**. I körexemplet nedan har vi valt en momssats som är vanlig i Sverige. Men den generella formel som används i funktionen **netto()** tillåter även andra momssatser.

```
Tabell över brutto- och nettobelopp samt ingående moms:

        Bruttobelopp för början av tabellen:     125
        och för slutet av tabellen:              150
        Ange steget för momstabellen:            5
        Aktuell momssats i % (t.ex.25):          25

Brutto          Netto           Ingående moms
-----------------------------------------------
125.00          100.00          25.00  kr
130.00          104.00          26.00  kr
135.00          108.00          27.00  kr
140.00          112.00          28.00  kr
145.00          116.00          29.00  kr
150.00          120.00          30.00  kr
```

Nedan ges en matematisk härledning av formeln för nettobeloppet:

$$\text{netto} + \frac{\text{netto} \cdot \text{momssats}}{100} = \text{brutto}$$

$$\frac{100 \cdot \text{netto} + \text{netto} \cdot \text{momssats}}{100} = \text{brutto}$$

$$\text{netto} \cdot (100 + \text{momssats}) = 100 \cdot \text{brutto}$$

$$\text{netto} = \frac{100 \cdot \text{brutto}}{100 + \text{momssats}}$$

Denna formel används i funktionen **netto()** i satsen **return** `100*b/(100+m);` Skulle man vilja skriva en funktion för ingående moms kan man sätta in resultatet ovan i sambandet *ingående moms = brutto – netto* och använda följande direkt formel (lämpligt som övningsuppgift):

$$\text{ingående moms} = \frac{\text{brutto} \cdot \text{momssats}}{100 + \text{momssats}}$$

7.6 Funktioner utan returvärde

Hittills hade alla våra funktioner returvärden. Det var *en* typ av funktion, en ganska viktig sådan. Men funktioner med returvärde har en begränsning: De kan returnera endast *ett* värde, *ett* tal, *ett* tecken, *ett* sanningsvärde, *en* sträng eller *ett* objekt. Med *returvärde* menar vi alltid det som returneras av **return**-satsen via funktionsnamnet. En annan typ av funktioner är sådana som inte har något returvärde alls. I denna bemärkelse pratar vi nu om *funktioner utan returvärde* som även kallas **void**-funktioner. I några andra programmeringsspråk heter de *procedurer*.

> En **void**-funktion är en funktion som inte returnerar något värde. I funktionshuvudet ersätter **void** returtypen.
> En **void**-funktion har i regel ingen **return**-sats eller en tom **return**-sats: **return;** vilket är samma sak.

void är i C++ ett reserverat ord som kan tolkas som ”ingenting”. I vissa fall kan **void** ersätta datatypen. När den i funktionshuvudet ersätter datatypen till returvärdet och skrivs framför funktionsnamnet, eliminerar den returvärdet och definierar en **void**-funktion. T.ex. är **void main()** ett helt korrekt funktionshuvud som medför att **return**-satsen i **main()** antingen måste strykas eller ersättas av en tom **return**-sats. Även en tom **return**-sats avslutar funktionen var i kroppen den än skrivs vilket innebär att all kod efter den inte utförs. **void** kan även dyka upp i parameterlistan till en funktion och betyder då att funktionen inte har några parametrar. T.ex. är även **main(void)** ett korrekt funktionshuvud som är identiskt med **main()**. Däremot får **void** inte ersätta datatypen i deklarationer av vanliga variabler. Det strikt typbestämda språket C++ tillåter inte att en variabel definieras till ”ingen datatyp”. Följande är ett exempel på en **void**-funktion som definieras och lagras i en separat fil:

```
// Compare.h
// Funktion som jämför två tal på likhet, mindre än, större
// än och skriver ut resp. meddelande. Tom return-sats avslu-
// tar funktionen om talen är lika stora.
void compare(int g, int h)
{
   if (g == h)
   {
      cout << "\aGrattis, du har gissat rätt!\n\n";
      return;         // Tom return-sats avslutar funktionen
   }                  // utan att returnera något värde
   if (g < h)
      cout << "För LITET, försök igen !\n\n";
   else
      cout << "För STORT, försök igen !\n\n";
}
```

Meddelandetexterna tyder på att denna funktion används i samband med ett program som gissar tal. Det är faktiskt Gissa tal- spelet som vi tidigare behandlat i ett antal varianter: **GissaTal_1** (sid 101) och **GissaTal_2** (sid 106). Låt dem passera revy. Nu ska vi modularisera **GissaTal_2**. Vilka moduler – logiskt meningsfulla delproblem – finns i den? En av dem är den ovan definierade **void**-funktionen **compare()**. Där löses följande delproblem: Jämför det gissade talet **g** med programmets hemliga slumptal **h** och skriv ut ett grattismeddelande om de är lika. Dessutom meddela, om de inte är lika, mindre eller större. För att kunna lösa denna uppgift måste **compare()** ta in värdena till det gissade och hemliga talet via parametrarna **g** och **h**, jämföra dem med varandra och avsluta i fall att **g = h** dvs om användaren gissat rätt. Just i detta fall avslutar satsen **return;** som står i den första **if**-satsen, funktionen: Koden efter den utförs inte. En **return**-sats avslutar alltid en funktion. Utan den skulle efter grattismeddelandet även skrivas ut: **För STORT, försök igen!** vilket inte vore önskvärt. Den tomma **return**-satsen returnerar dock inget värde. Därför är **compare()** en **void**-funktion.

Den andra modulen i applikationen är:

```
// GuessVoid.cpp
// Gissa tal-spelet med slumptal i upprepad dialog
// Kan avslutas även om användaren inte gissar rätt
// Slumptal genereras i intervall med funktionen myRand()
// Gissningen testas i void-funktionen compare()
// Båda är externlagrade och inkluderas här
#include <iostream>
#include <ctime>
using namespace std;

#include "Compare.h"                    // Innehåller compare()
#include "..\MyRand.h"                  // Innehåller myRand()

int main()
{
   srand(time(0));
   int guessedNo, secretNo = myRand(1, 20); // Anrop myRand()

   do
   {
     cout << "\nGissa tal mellan 1 och 20 (Avsluta med 0): ";
     cin >> guessedNo;
     cout << "\n\t";
     if (guessedNo == 0)
     {
        cout <<"\nProgr.s hemliga tal var:\t" <<
                                          secretNo <<'\n';
        break;                          // Bryter do-loopen
     }
     compare(guessedNo, secretNo);      // Anrop av compare()
   } while (guessedNo != secretNo);
}
```

I denna modul finns **main()** som anropar **compare()** i en **do**-loop. Loopen håller igång dialogen för att användaren ska kunna fortsätta att gissa ett nytt tal i fall han/hon gissat fel. Loopen avslutas när avslutningsvillkoret **(guessedNo != secretNo)** är falskt dvs när **guessedNo** är lika med **secretNo**, när användaren gissat rätt. **if**-satsen i **do**-loopen ger användaren möjligheten att kunna avsluta och få reda på programmets hemliga tal genom att mata in **0**. Men det är inte loopens avslutningsvillkor som bryter loopen utan **break**-satsen. Den hoppar över resten av koden på ett liknande sätt som den tomma **return**-satsen i **compare()**.

En viktig praktisk konsekvens av **void**-funktioner är att anropet till skillnad från funktioner med returvärde inte behöver – ja inte får – inbäddas i en **cout**- eller tilldelningssats. **void**-funktioner anropas helt enkelt med namn och väl definierad parameterlista, om sådan finns. I vårt exempel finns två parametrar:

```
compare(guessedNo, secretNo);
```

När själva funktionsnamnet inte längre bär något värde, kan det varken skrivas ut eller tilldelas någon variabel. Därför behöver vi inte längre någon variabel som tar hand om returvärdet. Det enda anropet gör är att exekvera den kod som står i funktionens definition, närmare bestämt i kroppen, efter att ha överfört parametrarnas värde till funktionen. Från funktionen kommer inget tillbaka. Den tredje modul som ingår i programmet **GissaVoid** är slumptalsgenereringen:

```
// MyRand.h

int myRand(int a, int b)          // Funktion som slumpar fram
{                                 // heltal i intervallet mel-
   if (a < b)                     // lan a och b, inkl. a, b
      return a + rand() % (b-a+1);
   else
      return b + rand() % (a-b+1);
}
```

Det är ett delproblem som är logiskt helt oberoende av resten av programmet och lämpar sig som en separat modul. Vi har i ett tidigare program (sid 138) redan använt den som funktion, fast i samma fil. Nu tar vi över den direkt därifrån och skriver den i filen **MyRand.h** som inkluderas i **GuessVoid** med **#include "..\MyRand.h"**. Anledningen till **..** är att vi placerat filen i mappen strax över den aktuella mappen för att använda den även till andra program: **..** betyder *mappen ovan*. I **main()** anropas funktionen **myRand()** före **do**-loopen då det vid varje körning bara behövs ett hemligt tal, till skillnad från det gissade talet. I anropet har vi valt det lilla intervallet **(1, 20)** för att förenkla gissningen. **else**-delen i funktionen **myRand()** ska göra användningen säkrare: Även om man skulle anropa funktionen med **myRand(20, 1)** skulle slumptal genereras mellan **1** och **20**. Till skillnad från **compare()** är **myRand()** ingen **void**-funktion utan returnerar det genererade slumptalet till **main()** där det tilldelas variabeln **secretNo**. Hela Gissa talspelet består nu av dessa tre funktioner: **main()**, **compare()** och **myRand()**. Ett körexempel skulle likna det på sid 107. För förklaring av formlerna se sid 118.

7.7 Lokala och globala variabler

```
// Global.cpp
// Skriver ut en tabell över tillkommande eller ingående moms
#include <iostream>
#include <iomanip>
using namespace std;

int vatType;                              // Global variabel
/*********************************************************************/
float vat(float a, float v)               // Definition av funk-
{                                         // tionen vat()
   if (vatType == 1)
      return a*v / 100;                   // Tillkommande moms
   else
      return a*v / (100 + v);             // Ingående moms
}
/*********************************************************************/
int main()
{
   float first, last, step, vatRate, amount;
   cout << "Vill du lägga till(1) eller dra av(0) moms? " <<
                                                  " (1/0): ";
   cin >> vatType;
   cout << "Ange ett belopp för början                  : ";
   cin >> first;
   cout << "och för slutet på en momstabell            : ";
   cin >> last;
   cout << "Ange steget för momstabellen               : ";
   cin >> step;
   cout << "Ange aktuell momssats i procent (t.ex. 25)  : ";
   cin >> vatRate;
   cout << fixed << setprecision(2) << '\n';
   if (vatType == 1)
      cout << "Netto\t\tBrutto\t\tTillkom. moms\n";
   else
      cout << "Brutto\t\tNetto\t\tIngående moms\n";
   cout << "-----------------------------------------\n";
   for (amount=first; amount<=last; amount=amount+step)
   {
       cout << amount << "\t\t";
       if (vatType == 1)                          // Anrop av
          cout << amount  + vat(amount, vatRate)  // vat():
               << "\t\t" << vat(amount, vatRate) << " kr\n";
       else
          cout << amount  - vat(amount, vatRate)
               << "\t\t" << vat(amount, vatRate) << "  kr\n";
   }
   cout << "\n";
}
```

Programmet ovan består av de två funktionerna **vat()** och **main()**. Dessutom finns nu för första gången en variabel definierad utanför **main()** och även utanför funktionen **vat()**: Variabeln **vatType** är en s.k. *global* variabel, just därför att den är definierad både utanför **vat()** och **main()**. Anledningen för denna placering är att den används i båda funktionerna och måsta gälla i båda. För att kunna göra det måste den definieras globalt. Man kan även säga att variabeln **vatType** är *global* därför att den är definierad ovanför alla *block* i programmet.

Blockstruktur

Om *block* sa vi på sid 96:

> ” I C++ kallas ett antal satser som omsluts av klamrarna **{** och **}** för *block*. … Klamrarna är *gränser* mellan programmets olika delar. De sätter gräns för variablers räckvidd. För att överskrida dem måste vissa regler om *blockstruktur* beaktas.”

Dessa regler är inget annat än reglerna för lokala och globala variabler. Ett mycket typiskt exempel på block är funktioner. Deras kroppar bildar block när de anropas. Därmed ger ett program som innehåller funktionsanrop, upphov till blockstruktur. Programmet **Global** innehåller flera anrop av funktionen **vat()**. Ett av dem – det första – ger upphov till följande blockstruktur:

```
int vatType;                    // Global variabel

int main()
{
                                // Lokala variabler i main():
   float first, last, step, vatRate, amount;
   .
   .
   .
   if (vatType == 1)            // Funktionsanrop:
      cout <<  . . .  << vat(amount, vatRate) << . . .

                    float vat(float a, float v)
                    {     // Lokala variabler i vat()

                       if (vatType == 1)
                          return a*v / 100;
                       else
                          return a*v / (100 + v);
                    }
   .
   .
   .
}
```

Att blockstrukturen ritas så här, beror på att funktionen **vat()** anropas i **main()** vilket innebär att koden till **vat()** exekveras där anropet inträffar. Detta skapar blockstrukturen som följer händelseförloppet vid exekvering och inte ordningen funktionerna skrivits i koden. Se även blockstrukturen på sid 148.

Frågan som dyker upp är: Vad händer med en variabel när man överskrider blockgränserna? Dvs hur långt räcker en variabeldefinition? Man pratar om variablers giltighetsområde eller *räckvidd*, på engelska *scope*. Generellt gäller:

Regeln för räckvidden av variabler:

En variabels räckvidd börjar vid deklarationssatsen och slutar vid gränsen till det block där den deklarerats.

Globala variabler

Om vi tillämpar regeln ovan på t.ex. variabeln **momstyp** i programmet **Global**, visar blockstrukturen på förra sida att det inte finns något block som variabeln är definierad i och därmed inte heller någon gräns. **vatType** är ju definierad utanför alla block. Därav följer att variabelns räckvidd börjar vid definitionen och slutar först när programet tar slut vilket innebär att **vatType** är en global variabel. Med andra ord, regeln kan även användas för att skilja mellan globala och lokala variabler. Avgörande för frågan om en variabel är global eller lokal, är alltså *platsen* där den definieras. Definieras en variabel utanför alla block blir den global. Definieras en variabel i ett block, blir den lokal i det blocket. En global variabels räckvidd är obegränsad. Därför gäller den globala variabeln **vatType** i hela programmet och därmed i programmets alla funktioner. Globala variabler tränger genom alla blockgränser och gäller i alla inre block, så länge de inte omdefinieras. Därför gäller **vatType** både i **main()** och **vat()**. En annan egenskap av globala variabler är att de initieras automatiskt till 0 om de är tal, till nolltecknet om de är tecken och till tom sträng om de är strängar. I vårt programexempel utnyttjas inte denna egenskap. Dock är den av betydelse då lokala variabler beter sig annorlunda: De måste initieras explicit i programmet.

Lokala variabler

I **main()** finns det fem variabler som är lokala: **first**, **last**, **step**, **vatRate**, **amount** och **vat**. De gäller endast i **main()**, men inte i **vat()**.

Hur är det med lokala variabler i funktionen **vat()**, finns några där? I kroppen är inga variabler definierade. Hade sådana funnits hade de varit lokala. De enda variabler som används i **vat()** är parametrarna **a** och **v** som är definierade i parameterlistan. Angående parametrar i en funktion gäller generellt följande:

Parametrarna i en funktion behandlas som lokala variabler i funktionen.

Därför gäller variablerna **b** och **m** endast i **vat()**, men inte i **main()**. Ett försök att använda dem i **main()** skulle ge kompileringsfel. Gör gärna testet! Kriteriet för om en variabel är global eller lokal, är *platsen* där den definieras. En parameter som definieras i funktionens parameterlista tillhör funktionens huvud och därmed funktionen. Trots detta är parametrarna inte i alla avseenden likställda de lokala variabler som definieras i funktionens kropp. Därför säger vi inte att de *är* lokala variabler utan att de *behandlas som* sådana. Deras särställning i parameterlistan innebär att de till skillnad från kroppens lokala variabler, kan *kommunicera* med omgivningen dvs med andra funktioner. Kom ihåg att vi har kallat dem *formella parametrar* därför att de initieras (får sina värden första gången) när funktionen anropas. Dvs de tar emot sina värden från de aktuella parametrarna **amount** och **vatRate** vilka är definierade i **main()**. Värdena kopieras från **amount** till **a** och från **vatRate** till **v**. Nu kan vi se att den här kopieringen går över blockgränsen från lokala variabler i ett block till variabler som behandlas som lokala (parametrar), i ett annat block. Detta ger upphov till formuleringen: parametrarna bildar en del av funktionens *gränssnitt* mot omgivningen dvs mot andra funktioner. I själva verket är det funktionshuvudet i sin helhet som utgör gränssnittet.

Ett körexempel av programmet **Global** kan se ut så här:

```
Vill du lägga till(1) eller dra av(0) moms? (1/0): 1
Ange ett belopp för början                       : 66.50
och för slutet på en momstabell                  : 70
Ange steget för momstabellen                     : 0.50
Ange aktuell momssats i procent (t.ex. 25)       : 25

Netto           Brutto          Tillkom. moms
-------------------------------------------
66.50           83.12           16.62 kr
67.00           83.75           16.75 kr
67.50           84.38           16.88 kr
68.00           85.00           17.00 kr
68.50           85.62           17.12 kr
69.00           86.25           17.25 kr
69.50           86.88           17.38 kr
70.00           87.50           17.50 kr
```

Körexemplets indata har valts på så ett sätt som visar att programmet kan användas på väldigt olika och även udda sätt. Tabellsteget kan vara decimalt pga datatypen **float** till variabeln **step**. Man skulle kunna förfina tabellsteget ytterligare. Tabelländarna kan också vara decimala. Andra momssatser fungerar lika bra. Väljer man i början momstypen **0** dvs ingående moms och matar in annars samma indata som på sid 143 till programmet **ExternFct** får man exakt samma resultat.

Varför globala variabler?

Av vilken anledning har **vatType** definierats som global variabel i **Global**? För att förstå det måste vi gå tillbaka till programmets upplägg. Vilket problem ska

programmet lösa? Man ska kunna beräkna moms som tillkommer till ett nettobelopp eller moms som ingår i ett bruttobelopp. För att kunna behandla båda alternativ efterfrågas först användarens önskemål i början av **`main()`**:

```
Cout<< "Vill du lägga till(1) eller dra av(0) moms? (1/0): ";
cin >> vatType;
```

Svaret läses in och tilldelas **`int`**-variabeln **`vatType`**. Denna inläsning som enligt ledtexten görs med **`1`** eller **`0`** används i **`main()`** för att skilja mellan moms på nettobeloppet och momsdelen av bruttobeloppet för att skriva ut korrekt huvud till tabellen och beräkna rätt netto- resp. bruttobelopp. Men samma distinktion måste även göras i funktionen **`vat()`** vilket ger upphov till en variabel som är giltig i både **`main()`** och **`vat()`**, dvs en global variabel. Även i funktionen **`vat()`** måste **`vatType`** med en **`if`**-sats avgöra valet av rätt momsformel. Så, **`vatType`** är nödvändig i båda funktioner. Framför allt ska variabeln behålla sitt värde när man vid anropet går över från den ena till den andra dvs när man överskrider blockgränserna. Därför måste värdet vara lagrat i en och samma minnescell. Men generellt sett medför globala variabler nackdelar även om dessa inte är påtagliga i exemplet.

Användningen av globala variabler strider mot modularisering och återanvändning av kod. T.ex. kan funktionen **`vat()`** inte isoleras som en separat modul och användas i något annat program då den alltid är beroende av programmet **`Global`**. Orsaken är att den globala variabeln **`vatType`** svetsar samman dem. Vid kompilering av **`vat()`** som del av att annat program måste ju variabeln **`vatType`** vara definierad. Men den är inte definierad i **`vat()`**. Då skulle den inte gälla i **`main()`**. Ett annat skäl är att felsökning i större program blir svårare när man försöker spåra en global variabel genom många funktioner. Även kontrollen av programflödet och struktureringen av koden kan kompliceras. Därför:

Rekommendation för användning av globala variabler:

Var restriktiv i bruket av globala variabler och använd dem endast då det är absolut nödvändigt.

Egentligen borde bra programmering kunna undvika globala variabler. Därav följer frågan: Var det absolut nödvändigt att ha en global variabel i programmet **`Global`**? Vid närmare betraktelse (och lite mer kunskap) ser man nämligen att detta inte alls var absolut nödvändigt. Det finns följande alternativ till globala variabler:

Alternativet: Parametrisering av globala variabler

Detta innebär att förvandla den globala variabeln till en lokal variabel i **`main()`** och att ersätta den i funktionen med en ny parameter. Dvs man inför en ny, tredje parameter i **`vat()`** som får sitt värde överförd från **`main()`** vid funktionsanropet. Denna uppgift överlämnas till övningarna i slutet av detta kapitel, se övn 8.7.

7.8 Överskuggning av variabler

När vi pratade om blockstruktur ställde vi upp regeln för *räckvidden* av variabler (sid 149) som i sin tur gav upphov till *lokala* och *globala variabler*. Nu ska vi komplettera våra kunskaper ytterligare med ett koncept som löser namnkonflikter vilka kan uppstå när lokala och globala variabler har samma namn: *överskuggning* kallas det. På köpet kommer vi att lära känna den s.k. *räckviddsoperatorn*. Titta på följande program:

```
// Scope.cpp
// Globala och lokala variablers räckvidd (giltighetsområde)
// Överskuggning av variabler och räckviddsoperatorn ::
#include <iostream>
using namespace std;

int x;                          // Global variabel initieras
                                // automatiskt till 0
void local();
void update(int);               // Deklaration av funktioner

int main()
{
   int x = 5;                   // Lokal variabel i main()
   cout<<"Lokalt x i main() före fkt.anropen är "<< x<< '\n';
   local();
   update(10);
   local();
   update(10);
   cout<<"\nLokalt x i main() efter anropen är "<< x << '\n';
   cout<< "\nGlobalt x (hämtat med ::) är " << ::x << "\n\n";
   return 0;
}

void local()                    // Funktion som använder egen
{                               // lokal variabel
   int x = 55;                  // Lokalt x i local()
   cout << "\nLokalt x i local() är " << x
        << " i början av local()" << '\n';
   x++;
   cout << "\nLokalt x i local() är " << x
        << " vid slutet av local()" << '\n';
}

void update(int dx)             // Funktion som uppdaterar
{                               // global variabel
   cout << "\nGlobalt x är " << x << " i början av update()"
                                                  << '\n';
   x += dx;
   cout <<"\nGlobalt x är " << x << " vid slutet av update()"
                                                   << '\n';
}
```

Hur många variabler har vi i programmet **Scope**? Man ser bara variabeln **x**. Men är det hela vägen ett och samma **x** eller är det olika variabler med samma namn? Tillämpar vi våra kunskaper om lokala och globala variabler som vi lärt oss i förra avsnittet, kan vi konstatera följande: Det finns tre *olika* variabler som har samma namn **x**. De refererar till tre *olika* minnesceller dvs tre olika fysiska adresser med samma logiska namn i koden. Det kan de göra eftersom de definieras i olika block:

1. Först har vi där strax i början utanför **main()** den globala variabeln **x**. Platsen där den definieras, kallas *globalt namnutrymme*, en slags globalt ”block”. Därför gäller den i hela programmet och i programmets alla underblock dvs funktionerna **main()**, **local()** och **update()**. Det finns dock undantag, t.ex. vid överskuggning som kan förekomma i vissa underblock (se **2** och **3** nedan). Den globala variabeln **x** får i programmet följande värden:

Globalt x	~~0~~ ~~10~~ 20

Som global variabel initieras den automatiskt till **0** när den definieras. I det första anropet av funktionen **update()** i **main()** dvs i satsen **update(10);** ändras värdet till **10** och i det andra anropet till **20**, båda gånger genom satsen **x += dx;** Att det är den globala variabeln **x** som gäller i funktionen **update()**, beror på att den inte omdeklareras där. Till skillnad från funktionen **local()** skapas i **update()** ingen ny lokal variabel med samma namn. Därför är det **x** som används där, det globala **x**.

2. Sedan har vi en lokal variabel **x** i **main()** som endast gäller där. Nu uppstår uppenbarligen en konflikt: Denna lokala variabel har samma namn som den globala variabeln från punkt **1** som gäller i alla underblock och därmed även i **main()**. Båda kan inte gälla samtidigt, en måste slå ut den andra. Regeln säger följande:

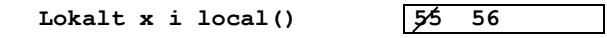
> En lokal variabel i ett block överskuggar (slår ut temporärt) en global variabel med samma namn.

Överskuggning (eng. *overriding*) bör inte förväxlas med överskrivning (eng. *overwriting*). Överskriva kan man bara variabelns *värde* t.ex. med en ny tilldelning. Då blir det gamla värdet överskrivet för gott, kan aldrig återskapas och det nya värdet gäller i fortsättning. Överskuggning har inget att göra med variabelns värde utan med variabelns giltighet. I ett lokalt block kastar den lokala variabeln med samma namn temporärt en skugga över den globala variabeln. Bilden av skuggan har vi valt för att betona fenomenets temporära karaktär. Före och även efter det lokala blocket har den globala variabeln sin fulla giltighet. Så, i det lokala **main()**-blocket gäller den ”egna” lokala variabeln **x**. Den globala variabeln **x** gäller inte längre i **main()** utan är överskuggad av den lokala.

Den globala gäller endast före och efter **main()**. Hade vi valt ett annat namn för den lokala variabeln i **main()** hade förstås den globala haft full giltighet i **main()**. Men då hade vi inte lärt oss överskuggning, ett viktigt koncept inom programmering som tillämpas inte bara på variabler utan även på funktioner (se bokens sista kapitel om polymorfism). Den lokala variabeln **x** i **main()** initieras till **5** och ändras aldrig:



3. Slutligen används namnet **x** även för att definiera en lokal variabel **x** i funktionen **local()** på vilken samma överskuggningsregel (se förra sidan) tillämpas som på det lokala **x** i **main()**. Dvs även i funktionen **local()** överskuggar det lokala **x** som definieras där och initieras till **55**, programmets globala variabel **x** pga samma namn. Sedan ökas **x**-värdet med **1** genom **x++;** så att det blir **56** innan variabeln ”dör” då funktionen **local()** upphör:



Denna minnesbild uppstår i programmet **Scope** två gånger pga att funktionen **local()** anropas två gånger i **main()**.

Räckviddsoperatorn **::**

En annan nyhet i programmet **Scope** är räckviddsoperatorn som består av två kolon **::** utan mellanslag och kan sättas framför en variabel **::x** för att referera till det **x** som definierats i det globala namnutrymmet dvs här till den globala variabeln **x**. I **main()** används den i den sista utskriftssatsen för att hämta den globala variabelns aktuella värde. Ett körexempel av programmet **Scope** bekräftar detta:

```
Lokalt x i main() före fkt.anropen är 5
Lokalt x i local() är 55 i början av local()
Lokalt x i local() är 56 vid slutet av local()
Globalt x är 0 i början av update()
Globalt x är 10 vid slutet av update()
Lokalt x i local() är 55 i början av local()
Lokalt x i local() är 56 vid slutet av local()
Globalt x är 10 i början av update()
Globalt x är 20 vid slutet av update()
Lokalt x i main() efter anropen är 5
Globalt x (hämtat med ::) är 20
```

Övningar till kapitel 7

7.1 Modularisera lösningen till övn 4.4 (sid 72) som läser in två heltal, gör beräkningar med dem och skriver ut resultaten. Separera beräkningarna *(bearbetningen)* från kodens andra delar *inmatning* och *utmatning*.

a) Flytta först multiplikationen till en funktion med returvärde med huvudet **`int mult(int a, int b)`** i samma fil som **`main()`**. Anropa funktionen **`mult()`** från **`main()`**. Bibehåll alla andra beräkningar. Se upp med att placera den nya funktionen inte i, utan *före* **`main()`**.

b) Fortsätt med att flytta funktionen **`mult()`** till en separat fil och inkludera den som headerfil vid kompilering. Anropet ska fortfarande göras från **`main()`**.

c) Gör samma sak med alla andra beräkningssätt. Lagra alla funktioner externt i en separat fil och anropa dem från **`main()`**.

7.2 a) Skriv först ett program med endast **`main()`**-funktionen som läser in radien **`r`** till en cirkel samt beräknar och skriver ut cirkelns area π **`r`**^{2} och dess omkrets 2π **`r`**, där $\pi = 3.14159$.

b) Flytta sedan bearbetningsdelen dvs beräkningen av area och omkrets ur **`main()`** till separata funktioner **`area()`** och **`omkrets()`**, men stanna i samma fil. I **`main()`** ska finnas kvar variabeln för radien, inmatning, utmatning och anropet av **`area()`** och **`omkrets()`**. Förse de nya funktionerna med en parameter som överför radiens värde från **`main()`** till dem. Välj olika namn för den aktuella än för den formella parametern. Definiera π lokalt i funktionerna. Dessutom ska **`area()`** och **`omkrets()`** returnera ett **`double`**-värde. För att testa mata in radien **`1`**. Då ska arean bli π pga π **`r`**$^2 = \pi$ och omkretsen bli 2π pga 2π **`r`** $= 2\pi$.

7.3 Skriv ett program som läser in **`sida`** till en kub samt beräknar och skriver ut kubens volym **`sida`**^{3} och dess yta 6 **`sida`**^{2}. Dessa beräkningar ska göras i två funktioner, en för volymen, en för ytan, båda i en separat headerfil. Avgör själv om funktionerna **`volym()`** och **`yta()`** ska returnera ett värde eller vara av **`void`**-typ. Anropa dem från **`main()`**.

7.4 Modularisera programmet **`Hour2Sec`** (sid 64) genom att skriva dess bearbetningsdel som en ny funktion. Bibehåll in- och utmatnigsdelen i **`main()`** och anropa den nya funktionen från **`main()`**. Avgör själv om den nya funktionen ska returnera ett värde. Ge den ett beskrivande namn.

7.5 Programmet **Hour2Sec** (sid 64) omvandlar timmar, minuter och sekunder till totalsekunder. Programmet **Sec2Hour** nedan löser det omvända problemet, nämligen att omvandla ett givet antal totalsekunder till timmar, minuter och sekunder:

```
// Sec2Hour.cpp
// Omvandlar antal sekunder till timmar, minuter & sekunder

#include <iostream>
using namespace std;

int main()
{
   int tim, min, sek, totalsek;

   /* I n m a t n i n g */
   cout << "\nAnge tid i sekunder: ";
   cin  >> totalsek;

   /* B e a r b e t n i n g */
   tim =   totalsek / 3600;
   min =  (totalsek % 3600) / 60;
   sek = ((totalsek % 3600) % 60) % 60;

   /* U t m a t n i n g */
   cout << '\n' << totalsek << " totalsekunder = " << tim
        << " timmar, " << min << " minuter, " << sek
        << " sekunder.\n\n";
}
```

En körning av programmet **Sec2Hour** ger t.ex. följande dialog: :

```
Ange tid i sekunder: 15910

15910 totalsekunder = 4 timmar, 25 minuter, 10 sekunder.
```

Modularisera programmet **Sec2Hour** genom att flytta bearbetnings- och utmatnigsdelen till en **void**-funktion. Dvs skriv ett program som läser in tiden i ett antal sekunder, anropar **void**-funktionen som omvandlar tiden till antal timmar, minuter samt resterande sekunder och skriver ut resultaten. Använd för omvandlingen den algoritm som är implementerad i programmet **Sec2Hour**. Varför kan man inte här använda en funktion med returvärde?

7.6 Programmet **ExternFct** (sid 141) är delvis modulariserad i och med att beräkningen sköts av funktionen **netto()**. Modularisera det ytterligare genom att flytta tabellutskriften till en **void**-funktion **table()**. Därmed flyttas även anropet av funktionen **netto()** till den nya funktionen och du får ett program med två externlagrade funktioner, där **main()** anropar **table()** och

table() anropar **netto()**. Fundera noga på vilka parametrar den nya funktionen **table()** ska ha.

7.7 Varför ger följande program kompileringsfel? Åtgärda felet genom att flytta på kod, utan att ta bort någon klammer och utan att ha tomma klamrar:

```
#include <iostream>
using namespace std;
int main()
{
   {
      int t = 30;
   }
   cout << "t = " << t;
}
```

Kapitel 8

Arrays

Ämne		Sida	Program
8.1	Vad är en array?	159	
8.2	Definition och initiering av en array	161	**ArrayDef**
	- Initieringslista	163	**ArrayInit**
8.3	Stränghantering med array	166	**ArrayChar**
	- Nolltecknet	166	**NullChar**
8.4	Kryptering av text	174	**EncryptTest**
		175	**Encrypt**
Övningar till kapitel 8 (Projekt *Labyrint II, Master Mind*)		177	

8.1 Vad är en array ?

När vi pratade om loopar sade vi så här (sid 105):

> ”Datorn har några egenskaper som är helt överlägsna motsvarande egenskaper hos människan: snabbheten, noggrannheten och förmågan att effektivt lagra och hantera stora datamängder samt förmågan att inte bli trött.”

Kontrollstrukturen *repetition* i sina olika varianter: **`while`**-, **`for`**- och **`do`**-satser utnyttjar både snabbheten och förmågan att inte bli trött. Detta ska nu kompletteras med ett verktyg som även utnyttjar datorns andra överlägsna egenskap: Att kunna effektivt lagra och hantera *stora datamängder*. Detta verktyg är datatypen *array* som i engelskan ordagrant betyder *ordnad uppställning* (*battle array* = stridsordning), en ordnad skara av värden i datorns minne. Andra beteckningar som används i litteraturen är *fält*, *vektor*, *matris*, *lista*. Vi kommer att använda *array*.

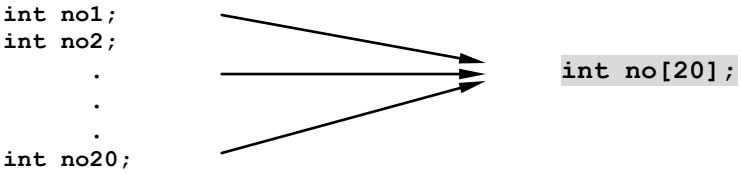
En array är en ordnad mängd av variabler av *samma* datatyp grupperade under *samma* namn och lagrade i ett *sammanhängande* minnesområde. En array består av ett antal element. Elementens position i arrayen kallas för index. Indexnumreringen börjar med 0, inte med 1.

Den sammanhängande lagringen underlättar adresseringen av elementen via indexet och är möjligt därför att alla element är av samma datatyp. Kända datatyper har hittills varit de enkla datatyperna (sid 75). En array är inte längre en enkel utan en s.k. *sammansatt* datatyp, den enklast tänkbara sammansatta datatypen.

En *sammansatt datatyp* representerar fler än ett värde åt gången, t.ex. flera heltal, flera flyttal, flera tecken, flera sanningsvärden osv. Det finns i litteraturen även det synonyma begreppet *härledd datatyp* dvs baserad på en annan datatyp. Man kan gruppera *enkla* datatyper till den nya härledda datatypen array.

Om vi t.ex. grupperar variabler av den enkla datatypen **`int`** får vi den sammansatta datayp som i kod skrivs **`int[]`** och läses *”array av* **`int`***”*. Varje element i en sådan array kan betraktas som en indexerad dvs numrerad variabel av typ **`int`**. Anta att vi vill deklarera 20 heltalsvariabler, då bjuder array på följande förenkling:

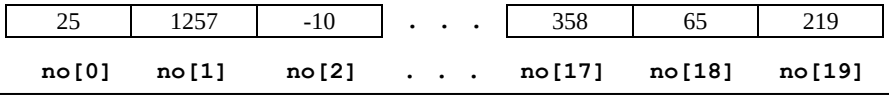
Hittills: enkel datatyp **`int`**: Nu: sammansatt datatyp ”*array av* **`int`**”:



Hittills har vi skrivit *20* satser (koden till vänster) för att deklarera 20 **int**-variabler. Men nu med array som sammansatt datatyp har vi möjligheten att göra samma sak med endast *en* sats genom att deklarera *en enda* variabel – visserligen inte längre en vanlig variabel utan en *arrayvariabel* – och lägga till informationen om antalet element i den. På så sätt har vi använt den nya sammansatta datatypen *array av* **int** för att definiera vår arrayvariabel **no**. Typiskt för den nya datatypen *array av* **int** är hakparenteserna **[ ]** i deklarationssatsen ovan som innehåller *antalet* element i arrayen. Arrayvariabeln **no** ersätter de 20 vanliga variablerna **no1**, **no2**, …, **no20** och består nu i sin tur av 20 *element*. Varje element är en heltalsvariabel som kan lagra ett heltalsvärde. Enda skillnaden är *sättet* dvs *koden* att komma åt dessa värden. Det är namnet på arrayvariabeln samt index till resp. element som måste anges för att komma åt värdet i elementet. Indexet är ett nummer som specificerar varje elements position i arrayen.

Låt oss anknyta till exemplet ovan där vi hade definitionssatsen: `int no[20];` som reserverar 20 minnesceller för lagring av 20 värden av typ **int**. Låt oss ytterligare anta att värdena i bilden nedan tilldelats alla element i arrayen **no**. De lagras i ett sammanhängande minnesområde. Det uppstår följande:

Minnesbild av arrayen no:



Under minnescellerna står hur C++ adresserar minnescellerna i en array och på vilket sätt man i koden kommer åt varje element. Exemplet visar hur indexeringen av element i en array görs. Anmärkningsvärt är att indexnumreringen börjar med **0**. Medan vi människor är vana vid att påbörja numreringen av ett antal objekt med 1, gör C++ det inte när det gäller numreringen av element i en array. Varför kommer vi att förstå när vi behandlar sambandet mellan pekare och array (sid 197). Tills vi når dit måste vi acceptera följande indexregel där vi med position menar numret som människan använder för att numrera elementen.

> Indexregeln: I arrays börjar numreringen av index alltid med 0.
> Därför gäller: elementets position = index + 1

Tillämpad på exemplet: Det 1:a elementet i arrayen **no** ovan (värdet 25) har index 0: Positionen är 1 medan indexet är 0. C++ kodar det med **no[0]**. Det 2:a elementet (värdet 1257) har index 1 och koden **no[1]**, det 3:e elementet (värdet –10) har index 2 och koden **no[2]** osv. Det n:e elementet har alltid index n-1. Därför har också det 20:e elementet (värdet 219) index 19. Det gäller att hålla isär det mänskliga sättet att numrera som börjar med 1 från C++ kodens sätt som börjar med 0. Vi har definierat 20 heltalsvariabler **no[0], ..., no[19]**. Antalet element är 20. Indexen går från 0 till 19.

8.2 Definition och initiering av en array

Följande program testar allt vi hittills sagt om arrays speciellt indexregeln. Det visar skillnader mellan vanliga variabler och arrays samtidigt som det avslöjar att de för arrays så typiska hakparenteserna **[ ]** inte alltid har samma betydelse.

```
// ArrayDef.cpp
// Definierar en array av int och initierar den elementvis
// For-sats skriver ut värdena med resp. index
// For-satsens räknare används samtidigt som arrayens index
// Ingen kontroll av indexgränserna i array via kompilatorn
#include <iostream>
using namespace std;

int main()
{
   int no[4];                          // Definition av en array
   no[0] = 64;                         // Elementvis initiering
   no[1] = 86;                         // av arrayen
   no[2] = 34;
   no[3] = -6;
   for (int i=0; i <= 3; i++)          // i = index = räknare
      cout << "\n\tArrayens "<< i + 1 << ":a element no["<< i
           << "] har värdet "<< no[i] << " och index " << i
           << "\n";
   cout << "\tIndex -1 och 4 ligger utanför och är "
        << "ej definierade.\n\n\tResultat:\n";
   cout << "\tno[-1] = " << no[-1] << " med index -1 och\n"
        << "\tno[ 4] = "  << no[4]  << " med index  4 "
        << "är odefinierade skräpvärden!" <<"\n";
}
```

Det blir enklare att prata om programmet om man även tittar på en körning:

```
Arrayens 1:a element no[0] har värdet 64 och index 0

Arrayens 2:a element no[1] har värdet 86 och index 1

Arrayens 3:a element no[2] har värdet 34 och index 2

Arrayens 4:a element no[3] har värdet -6 och index 3

Index -1 och 4 ligger utanför och är ej definierade.

Resultat:
no[-1] = -858993460 med index -1 och
no[ 4] = -858993460 med index  4 är odefinierade skräpvärden!
```

Körningen ovan visar att icke-definierade arrayelement varken leder till kompilerings- eller exekveringsfel. Index **-1** och **4** överskrider de definierade indexgränserna **0** och **3**, den ena till vänster, den andra till höger. Arrayelementen **no[-1]**

och **no[4]** är varken definierade eller tilldelade några värden. Ändå kan man kompilera och köra programmet utan något felmeddelande. Inte ens en varning påpekar att man använt kod som är odefinierad. Anledningen är följande:

> I en array kontrollerar C++ kompilatorn inte indexgränserna utan endast arraynamnet.

Ett annat namn än det definierade arraynamnet **no** leder till kompileringsfel. Däremot kan vi använda vilket index som helst, även om det överskrider de definierade gränserna, utan att det blir kompileringsfel. Ansvaret för kontroll av indexgränserna ligger helt och hållet hos programmeraren. Av indexregeln (sid 160) följer att negativa index generellt inte är tillåtna, även om kompilatorn inte protesterar. Skälet för denna liberala attityd är bl.a. strävan efter snabb kompilering, vilket förstås är på bekostnad av säkerheten.

Man kan ju undra varför **no[4]** inte är definierat – som vi hävdar ovan – fast det ”förekommer” i definitionssatsen **int no[4];** Det ser bara ut *som om* det förekommer. Detta beror på att hakparenteserna **[]** inte har samma betydelse i programmets alla satser. Den korrekta tolkningen av **[]** beror på sammanhanget:

Hakparentesernas två olika betydelser

1. **I definitionssatser** omsluter hakparenteserna *antalet* element i arrayen dvs arrayens *storlek*. T.ex. innebär raden

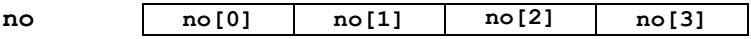
   ```
   int no[4];          // Definition av en array
   ```

   i programmet **ArrayDef** att variabeln **no** definieras till datatypen *array av* **int** med **4** element dvs att **4** minnesceller reserveras för lagring av **int**-värden. Det gemensamma för alla dessa element är arraynamnet **no**:

   Här är frågan om ”Hur många element?”. Detta kallas för *kardinaltal*.

2. **I alla andra satser** omslutar hakparenteserna *index* till varje element av en array. Här handlar det om ett elements *position* i arrayen. Man anger index inom hakparenteser för att referera till elementet när man vill hämta eller tilldela det ett värde. Indexregeln (sid 160) tillämpas enligt vilken indexeringen börjar med **0**. Därför är t.ex. **no[4]** i arrayen **no** inte definierat:

   Här är frågan om ”Vilket element?”. I matematiken kallas detta för *ordinaltal*.

Den ovan beskrivna skillnaden i tolkningen av **[]** har viktiga praktiska konsekvenser som vi kommer att se senare.

Initieringslista

Precis som det finns skillnader i definitionen av arrayvariabler jämfört med vanliga variabler, finns även skillnader vid initieringen dvs första tilldelningen. T.ex. är initieringen av arrayen **no** i programexemplet **ArrayDef** (sid 161) – en sats för varje element – inte särskilt lämplig för arrays, speciellt om man skulle tillämpa samma teknik på större arrays. Men just hanteringen av stora datamängder var ju motiveringen för att syssla med array. Kan man inte effektivisera initieringen? Jo, till en viss gräns. Det finns i huvudsak två möjligheter: antingen att använda **for**-satser eller att slå ihop definitionen med tilldelningen till en kortform som använder sig av en s.k. *initieringslista*. Båda har vi använt i följande program:

```
// ArrayInit.cpp
// Kortform för definition och initiering av en array i EN
// och samma sats med hjälp av en initieringslista
// Elementvis tilldelning av en array kan med fördel göras
// med en for-sats: Arrayens index = for-satsens räknare
#include <iostream>
using namespace std;

int main()
{
   int no[] = { 64, 86, 34, -6 }; // Kortform för definition
                                  // och initiering i EN sats
                                  // med initieringslista
   int copy[4];                   // Endast definition
   for (int i=0; i <= 3; i++)     // Elementvis initiering med
       copy[i] = no[i];           // for-sats. Direkt (array-
                                  // vis) initiering copy=no;
                                  // skulle ge kompileringsfel
   cout << '\n';
   for (int i=0; i <= 3; i++)
       cout << "\tcopy:s "<< i + 1 << ":a element copy[" << i
            << "] = " << copy[i] << " med index " << i
                                                  << "\n";
}
```

En körning av programexemplet **ArrayInit** 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:a element copy[2] = 34 med index 2

     copy:s 4:a element copy[3] = -6 med index 3
```

Både definitionssatsen och initieringssatserna i **ArrayDef** – det är de 5 första satserna i **main()** – kan slås ihop till den enda satsen:

```
int no[] = { 64, 86, 34, -6 };    // Kortform för definition
                                  // och initiering i EN sats
```

som gör två saker: Först, fram till tilldelningstecknet definieras arrayen **no** *utan* någon uppgift om arrayens storlek. Sedan, från och med tilldelningstecknet tilldelas arrayen **no**:s element fyra värden som står i en kommaseparerad lista grupperad inom klamrarna **{ }** som kallas arrayens *initieringslista*. Satsen ovan är endast en kortform för de fem första satserna i **main()**-funktionen till **ArrayDef** och gör precis samma sak som de. Hakparenteserna i kortformen (före **=**) tillhör definitionsdelen. Därför måste deras innehåll tolkas som arrayens storlek som kan och bör utelämnas eftersom satsen inte är avslutad än efter den avlutande hakparentesen utan kompilatorn kompilerar satsen fram till semikolonet. På så sätt får kompilatorn informationen om arrayens storlek i initieringslistan, räknar alltså antalet element inom klamrarna **{ }**. Det är också tillåtet att explicit ange det korrekta antalet element inom hakparenteserna **[ ]**. Kompilatorn ignorerar denna uppgift om man anger ett tal som är lika med eller större än det faktiska antalet element som finns i initieringslistan. Däremot blir det kompileringsfel om man anger ett tal som är mindre än det faktiska antalet element. För att undvika eventuella komplikationer beroende på felräkning rekommenderas att låta kompilatorn räkna och utelämna uppgiften om arrayens storlek när man definierar och tilldelar en array i en och samma sats. I praktiken är det bäst att i kortformen skriva ett tomt hakparentespar **[ ]** precis som vi gjort i programexemplet **ArrayInit**. Observera att man får använda kortformen ovan endast i samma sats som definitionen.

Däremot är det inte möjligt att utelämna uppgiften om arrayens storlek när man i en sats endast definierar en array som t.ex. i raden:

```
int copy[4];                      // Endast definition
```

eftersom här finns inte någon annan information om arrayens storlek än uppgiften inom hakparenteserna. Utan den informationen kan inget minnesutrymme för arrayen reserveras. Denna array har definierats i **ArrayInit** för att skapa en kopia av arrayen **no** genom att tilldela **no**:s värden till **copy** dvs tilldela värdena av alla element i arrayen **no** till resp. element i arrayen **copy**. Om **no** och **copy** varit vanliga variabler hade man kunnat skriva tilldelningssatsen **copy = no;** då variabeln **no** är både definierad och tilldelad. Även **copy** är definierad. Men med arrayvariabler går det inte. Det ger kompileringsfel om man försöker att tilldela värdena i en array direkt (arrayvis) till en annan array. Följande gäller:

> Tilldelning av värden i en array kan endast göras elementvis.

Den elementvisa tilldelningen kan antingen göras ”manuellt” dvs varje elementtilldelning i en separat sats – exempel på det hade vi i **Array** – eller i en **for**-sats.

Array i en `for`*-sats*

Man kan också effektivisera denna primitiva metod genom att använda en loop, med fördel en **`for`**-sats. Exempel på det har vi i **`ArrayInit`**:

```
for (int i=0; i <= 3; i++)    // Elementvis tilldelning med
    copy[i] = no[i];          // for-sats
```

Här görs en elementtilldelning per varv i **`for`**-satsen. Därför sker tilldelningen också elementvis vilket kroppens kod på den andra raden visar: vanliga variablers tilldelningssats **`copy = no;`** ersätts av arrayvariablers elementvisa tilldelningssats **`copy[i] = no[i];`** där **`i`** som både **`for`**-satsens räknare och arrayernas index går igenom alla element av båda arrayerna.

Att **`for`**-satsen är den mest använda loopen vid hantering av arrays är ingen tillfällighet. Det är naturligt att **`for`**-satsen lämpar sig bäst för arrays då det i förväg kända antalet element i en array ger upphov till det i förväg kända antalet repetitioner i **`for`**-satsen. Den logiska slutsatsen blir att sätta **`for`**-satsens räknare till arrayens index för att gå igenom alla element i arrayen. Därför kan, när en array bearbetas med for-sats, följande teknik användas med fördel:

> Arrayens index = **`for`**-satsens räknare

Denna teknik har även använts vid utskrift av arrayen **`copy`**:s värden i slutet av programmet **`ArrayInit`**.

8.3 Stränghantering med array

I början av detta kapitel motiverade vi behandlingen av arrays med att kunna lagra och bearbeta stora datamängder. Ett exempel på det är *textbehandling* där stora textmängder snarare är regel än undantag. Textbehandling är ett klassiskt område för datortillämpning just pga datorns förmåga att effektivt och snabbt kunna hantera stora datamängder.

I de två föregående avsnitten använde vi endast datatypen ”array av **`int`**” som är sammansatt av den enkla datatypen **`int`**. När array kan sätta samman enkla datatyper till en ny, större enhet, kan man använda den även för att behandla text som en sammansättning av tecken. För att representera tecken har vi den enkla datatypen **`char`**. Vi behöver bara bilda en array av **`char`** för att representera text. Men vad exakt är text? Det enda vi kan säga är att den består av ett antal tecken. Men var går den övre gränsen? Ett ord, en mening, en rad, en sida, en hel bok, ... – allt detta kan vara text. I det vanliga språket har man *ordet* som den minsta enheten i text och *mellanslaget* som avskiljare mellan två ord. Punkt och komma samt andra specialtecken är ytterligare verktyg för att strukturera text i vanligt språk. Däremot det vanliga språkets definitioner och verktyg lämpar sig inte alltid när man vill representera text i datorn. Man använder begrepp som är mer anpassade till datorns arbetssätt. Ett av dem är begreppet *sträng*.

> Sträng = samling av tecken som avslutas med *nolltecknet.*
> Nolltecknet har ASCII-koden 0 och kan i C++ kodas med escapesekvensen `\0`, med **`NULL`** eller `0`.

Nolltecknet

Självklart har nolltecknet inget att göra med siffran 0 som har ASCII-koden 48. Nolltecknet är det första av de icke-skrivbara, icke-läsbara styr- eller kontrolltecknen i början av ASCII-tabellen (sid 83). Dessa tecken har olika funktioner i olika språk. I C++ har nolltecknet bl.a. funktionen att avsluta strängar. *Sträng* är den minsta enheten i en text och *nolltecknet* avskiljaren mellan två strängar. Kvarstår frågan: När och hur läggs nolltecknet till en sträng? Skriva kan man inte, det är inte skrivbart. Läsa kan man inte heller, det är inte läsbart. Faktum är att nolltecknet automatiskt läggs till i slutet när man sätter strängen inom citationstecken `" "` vilket kännetecknar datatypen sträng. Sedan kan man spåra det med ASCII-koden 0 eller med escapesekvensen **`\0`**. Självklart ska man hantera strängar med datatypen **`string`** som introducerades på sid 71. Generellt är **`string`** stabilare och borde föredras i professionellt sammanhang. Men då kan man inte längre på en elementär nivå manipulera tecknen, vilket vi just vill göra. Dessutom vill vi lära oss mer om arrays och nolltecknet som vi kommer att koda med **`\0`** även om det lika bra går att göra det med **`NULL`** eller **`0`**. Därför definieras här strängar som **`char`**-arrays.

För att definiera och tilldela en **char**-array kan vi använda en initieringslista (sid 163):

```
char letter[] = {'a', 'b', 'c', 'd'};
```

precis som i: `int no [] = {64 , 86 , 34 , -6 };`

Den första satsen skapar variabeln **letter** som en array av **char** och tilldelar till den en sammanhängande följd av tecken. Men **letter** blir inte automatiskt en strängvariabel då det avslutande nolltecknet saknas. I koden finns det ingen information om att denna teckenföljd ska vara en sträng. Därför läggs i det här fallet nolltecknet inte till automatiskt. Men arrayvariabeln **letter** kan ersättas med en strängvariabel om vi i koden lägger till nolltecknet som sista tecknet i arrayen:

```
char text[] = {'a', 'b', 'c', 'd', '\0'};
```

Ett enklare alternativ är:

```
char text[] = "abcd";
```

Citationstecknen som kännetecknar datatypen sträng, visar att **abcd** *skall* vara en sträng och då läggs nolltecknet *automatiskt* till teckenföljden. Vi behöver inte längre göra det explicit. Därför blir **text** en *strängvariabel* som tilldelas *strängkonstanten* **abcd**. Observera att vi har åstadkommit detta med en array av **char** i definitionsdelen kombinerad med citationstecken i tilldelningsdelen.

I alla satser ovan skrivs definition och tilldelning av arrayvariabeln **text** i en och samma sats. Vi gör det även i följande program där lär vi oss stränghantering med array, manipulering med nolltecknet samt strängfunktionen **strcpy_s()** som kopierar strängar:

```
// ArrayChar.cpp
// Skriver ut en sträng och mäter dess längd med sizeof
// Kopierar (tilldelar) strängen till en ny variabel
// Kapar strängen med nolltecknet
// Definition av strängvariabler till datatypen array av char
#include <iostream>
using namespace std;
int main()
{
   char text[] = "C++";         // Definition och tilldelning
                                // i EN och samma sats
   // char text[] = {'C', '+', '+', '\0'};  // Sträng som ovan
   // char text[] = {'C', '+', '+'};        // Ingen sträng
   for (int i=0; i<=3; i++)
       cout << i+1 <<":a tecknet i \"C++\" är " << text[i] <<
                                                      '\n';
   cout << "\nSträngen \"C++\" tar " << sizeof(text)
        << " bytes: 3 tecken + nolltecknet\n\n"
        << "4:e tecknets ASCII-kod är " << (int) text[3] <<
                                                      '\n'
        << "Nolltecknets ASCII-kod är " << (int) '\0' << '\n'
```

```
        << "Alltså är 4:e tecknet i \"C++\" nolltecknet\n\n";
   char copy[4];
   strcpy_s(copy, text);     // text="C++" kopieras till copy
   copy[1] = '\0';           // Avkortning av "C++" till "C"
   cout << copy << " är en del av " << text << "\n\n";
}
```

Vi kommer att använda även separata definitionssatser. Men än så länge utnyttjar vi kortformen **`char text[] = "C++";`** eftersom den är så bekväm. Har man skrivit dem separat, kan initieringslistan inte användas efteråt för att tilldela arrayen.

Ett körresultat av programmet ovan visar att nolltecknet är osynligt dvs icke-läsbart och icke-skrivbart. Man kan identifiera det endast via ASCII-koden:

```
1:a tecknet i "C++" är C
2:a tecknet i "C++" är +
3:a tecknet i "C++" är +
4:a tecknet i "C++" är

Strängen "C++" tar 4 bytes: 3 tecken + nolltecknet

4:e tecknets ASCII-kod är 0
Nolltecknets ASCII-kod är 0
Alltså är 4:e tecknet i "C++" nolltecknet

C är en del av C++
```

Utskriften visar att ASCII-koden är 0: Det är det första tecknet i ASCII-tabellen och har inte det minsta att göra med siffran 0. En första indikation för nolltecknets existens i arrayen **`text`** är returvärdet **`4`** till **`sizeof(text)`** där strängvariabeln **`text`** har värdet **`C++`** som innehåller 3 tecken. Varje tecken är av typ **`char`** som tar 1 byte i minnesutrymme. Om nolltecknet inte fanns i arrayen **`text`**, skulle **`sizeof`** returnera 3 bytes. Men nolltecknet tar också 1 byte så **`sizeof`** returnerar **`4`** bytes när den tillämpas på **`text`** som parameter. Beviset för nolltecknets existens i arrayen **`text`** är att ASCII-koden till både arrayens 4:e tecken och nolltecknet är 0.

Slutligen använder vi i **`ArrayChar`** nolltecknet för att ändra i strängen **`C++`**. När man har ett tecken som fungerar som strängslutstecken kan man använda det för att manipulera strängar på alla möjliga sätt, t.ex. för att kapa en sträng vilket vi gjorde ovan. Men före detta ingrepp vill vi göra en kopia av strängen, ändra i kopian och på så sätt spara originalet för att sedan kunna använda båda. Kopiering innebär tilldelning av strängen till en ny strängvariabel som definieras i satsen **`char copy[4];`** Men tilldelning av arrayvärdena kan endast göras elementvis vilket i regel görs med en **`for`**-sats. Kommer man tänka på att överföra med **`for`** även nolltecknet när det gäller array av **`char`**? Annars blir kopian ingen sträng. Därför finns den fördefinierade funktionen **`strcpy_s()`** som står för ***str**ing **copy*** som kopierar sin

andra parameter till sin första och hanterar nolltecknet automatiskt. Båda parametrar måste vara strängar. Så här fungerar **strcpy_s()**:

Ersätter raden nedan: **strcpy_s(copy, text);**

Får ej skrivas i C++ kod: **copy = text**

copy ⟵ text

Den mellersta raden går förstås inte att skriva i C++ pga regeln om elementvis tilldelning av arrayvärden (sid 164). Efter kopieringen av strängen **C++** från **text** till **copy** tar vi kopian som nu också innehåller strängen **C++** och gör om den med:

```
copy[1] = '\0';
```

Här använder vi själva nolltecknet i koden för att avsluta strängen **copy** efter eget önskemål. Vi sätter nolltecknet på elementet med index 1 så att strängen **C++** kapas efter bokstaven **C** därför att tecknet **+** i elementet med index 1 (det första **+** tecknet i strängen) skrivs över av nolltecknet. Om vi jämför med originalet har vi:

text	**C**	**+**	**+**	**\0**
copy	**C**	**\0**	**+**	**\0**

Därför får vi **C** när vi skriver ut **copy** och **C++** när vi skriver ut **text**. Vad som står i **copy**:s element efter det första nolltecknet är irrelevant. Visserligen får man återanvända dem pga definitionssatsen **char copy[4];** men de är i princip oinitierade så länge det första nolltecknet finns där och strängvariabeln **copy** har värdet **C**.

När läggs nolltecknet till automatiskt? Sker det vid definitionen eller tilldelningen? Följande program visar att det är vid tilldelningen nolltecknet tillkommer:

```
// NullChar.cpp
// Läser in en sträng som kan även innehålla mellanslag och
// skriver ut strängens alla tecken med tillhörande ASCII-kod
// Använder nolltecknet för att avsluta utskriften i for-
// loopen precis när den inmatade strängen är slut
#include <iostream>
using namespace std;

int main()
{
   char namn[20];            // Reserverar plats för en sträng
                             // med max 20 tecken inkl. \0
   cout << "Ge ditt ditt för- och efternamn: ";
   cin.getline(namn, 20);    // Läser in en sträng med max 20
                             // tecken till strängvariabeln
                             // namn där mellanslag kan ingå
   cout << "\nIndex\t\tTecken\t\tASCII-kod\n"
        << "----------------------------------------\n";
```

```
   int i;                              // for-satsens räknare
   for (i=0; (namn[i] != '\0'); i++) // Skriver ut tecken så
       cout << i << "\t\t" << namn[i]// länge tecknen inte \0
            << "\t\t" << (int) namn[i] << '\n';
   cout << i << "\t\t" << namn[i] << "\t\t"
        << (int) namn[i] << "\n\n";
}
```

Här har vi separerat definitionen av **char**-arrayen från tilldelningen för att kunna mata in vilken sträng som helst. Kortformen i **ArrayChar** hade sina fördelar – enkelheten och kompaktheten – men nackdelen var att tilldelningen var hårdkodad och inte kunde göras interaktivt. Vill vi göra tilldelningen via inmatning kan vi tyvärr inte använda kortformen. Detta leder i sin tur till att vi redan vid definitionen måste bestämma oss hur stor arrayen ska vara. Det går inte att i en separat definition av en array utelämna uppgiften inom hakparenteserna om antalet element. Eftersom vi sedan vill tilldela ett namn – för- och efternamn – till strängvariabeln **namn**, tar vi en storlek som verkar rimlig för namn i allmänhet. Vi antar att de längsta namnen inte kan ha fler än 19 bokstäver. Vi reserverar 1 element också för nolltecknet och definierar därför strängvariabeln **namn** med

```
char namn[20];
```

som en array av **char** med **20** element. En sådan definition av en array kallas *statisk minnesallokering*. Allokera betyder i datasammanhang att reservera minnesutrymme. Definitionen ovan är en statisk minnesallokering därför att den redan vid kompilering reserverar ett sammanhängande minnesområde bestående av **20** minnesceller där varje minnescell kan lagra ett **char**-värde i arrayvariabeln **namn**. Storleken på detta minnesområde är oföränderlig under hela programmet. Arrays som definieras på detta sätt är alltid statiska i den bemärkelsen. Man måste bestämma sig redan vid definitionen, alltså *innan* man använder arrayen, hur stor den ska vara. Detta kan uppfattas som en stor inskränkning och är också i många fall ett verkligt hinder för vissa applikationer. Alternativet – *dynamisk minnesallokering* – är förstås möjligt i C++, men kräver att man använder *pekare* vilket introduceras i kapitel 10. I programmet **NullChar** vill vi läsa in användarens namn och lagra det i arrayen **namn**. Då vi i förväg inte vet hur långt det inmatade namnet kommer att vara, måste vi för säkerhets skull reservera lite mer plats än vad som kommer att behövas i de flesta körningar av programmet. Ett begränsat slöseri med minnesutrymme är oundvikligt när man använder statisk minnesallokering.

Stränginmatning med `cin.getline()`

Första frågan som dyker upp i **NullChar** är: Varför har vi inte använt en vanlig **cin**-sats för att läsa in användarens namn? Svaret är: därför att **cin** tolkar mellanslaget som avskiljare mellan två värden som ska läsas in. Alla s.k. *vita tecken* dvs **Enter**, mellanslag och tabulator tolkas av **cin** som avskiljare vid inmatning (sid 63). Därför skulle en **cin**-sats endast läsa in förnamnet och ignorera efternamnet när användaren matar in båda namnen skilda med mellanslag. Men vi vill ju läsa in

både för- och efternamn och lagra det hela som en sträng i arrayen **namn**. Därför använder vi en s.k. *metod* [*] eller *medlemsfunktion* som är fördefinierad i **cin** som heter **getline()** och anropas med s.k. *punktnotation* (sid 220):

```
cin.getline(namn, 20);
```

Denna funktion tolkar inte de vita tecknen som avskiljare vid inmatning så att strängar som läses in med **cin.getline()** även kan inkludera mellanslag och tabulatorer. Man kan alltså läsa in längre texter med **cin.getline()** – som det engelska namnet antyder en hel rad – där ”rad” betyder fram till *radslutstecknet* dvs **Enter**-tangenten. Funktionen **cin.getline()** sätter nolltecknet till den inlästa strängen och lagrar den inkl. nolltecknet i sin första parameter, i exemplet i arrayen **namn**. Observera att **cin.getline()** har två obligatoriska parametrar. Den första är strängvariabeln som man vill lagra den inmatade strängen i. Den andra är det maximala antalet tecken inkl. nolltecknet som strängen kan innehålla.

Efter att ha läst in både för- och efternamn till strängvariabeln **namn** skrivs **namn**:s alla tecken med tillhörande index och ASCII-koder ut i en **for**-sats vars villkor

```
namn[i] != '\0'
```

formuleras med hjälp av nolltecknet för att kunna avsluta loopen precis när strängen tar slut. Vilkoret innebär: Så länge tecknet med index **i** inte är nolltecknet ska loopen fortsätta. Så går loopen igenom strängen tecken för tecken då index **i** som samtidigt är **for**-satsens räknare börjar med **0** och ökar med **1** för varje varv. När denna genomgång påträffar nolltecknet avslutas loopen – utan att någon som helst information om den inmatade strängens längd har behövt användas. Så här kan en körning se ut:

```
Ge ditt ditt för- och efternamn: Kalle Karlsson

Index           Tecken          ASCII-kod
-----------------------------------------
0               K               75
1               a               97
2               l               108
3               l               108
4               e               101
5                               32
6               K               75
7               a               97
8               r               114
9               l               108
10              s               115
```

[*] Funktioner definierade i en klass heter *metoder*. De kan endast anropas med *punktnotation* dvs deras anrop står efter en punkt. Före punkten står i regel ett *objekt* – ett exemplar av den klass där metoden är definierad.

```
11              s               115
12              o               111
13              n               110
14                              0
```

Med hjälp av detta resultat ska vi nu förklara följande frågor:

1. När exakt läggs nolltecknet till arrayen?
2. När exakt slutar **for**-satsen?
3. Vad händer med de arrayelement som definierats men inte utnyttjas?

1. När man matar in för- och efternamn och trycker på **Enter**, läggs nolltecknet till arrayen direkt efter den sista bokstaven av efternamn, i körexemplet ovan efter bokstaven **n** med index **13**. Allra sista raden i utskriften ovan visar det osynliga nolltecknet med index **14** och ASCII-koden **0**. Definitionen reserverar alltid – vid varje körning – **20** tecken, men nolltecknet läggs till i slutet av varje inmatad sträng – dvs på olika ställen vid olika inmatningar. Närmare bestämt är det den funktion som läser in strängen dvs **cin.getline()** som lägger till nolltecknet så snart man trycker på **Enter** vid inmatning. Nolltecknet läggs till alltså vid inläsningen dvs tilldelningen och inte vid definitionen. Detta gäller generellt.

2. Det sista varvet i **for**-satsen skriver ut:

```
13          n               110
```

dvs indexet **i** är **13**, tecknet **n** är skilt från **\0** – därför skrivs det ut – samt tillhörande ASCII-kod som är **110**. Sedan blir **i 14** pga **i++**. Tecknet med detta index dvs **namn[14]** är lika med **\0**. Därför blir **for**-satsens villkor falskt och loopen avslutas med värdet **14** för **i**. Observera att det sista **i** som skrivs ut är **13**, men att **i** är **14** när vi lämnar **for**-satsen beroende på att **i++** utförs *efter* **cout**-satsen. Se **for**-satsens flödesplan på sid 110.

3. Efter **for**-satsen använder vi i en **cout**-sats räknaren **i** utan att ge den ett nytt värde. Dvs **i**:s värde efter **for**-satsen är samma som vi lämnade den med, nämligen **14**. Från punkt **2** vet vi också att tecknet med detta index dvs **namn[14]** är lika med **\0**. Därför skrivs ut:

```
14                              0
```

dvs det osynliga nolltecknet med ASCII-koden 0. Att **for**-satsens räknare **i** är väl definierad även efter **for**-satsen beror på att den är definierad som variabel *före* **for**-satsen, som en slags global variabel – global relativ till **for**-satsen. Det har vi gjort just för att kunna använda den efteråt för att visa att nolltecknet finns på index **14**, den aktuellt tilldelade strängens sista plats och inte på index **19**, den definierade strängvariabelns sista plats. Vid varje kör-

ning kommer denna plats vara olika pga olika inmatningar. Att nolltecknet finns på index `14` och inte på index `19` (arrayens sista reserverade element) visar att nolltecknet lagts till vid tilldelningen och inte vid definitionen.

Det inmatade namnet var kortare än `20` tecken. Vilka värden står i de resterande 5 minnescellerna efter nolltecknet? Även de har definitionen reserverat plats för utan att den aktuella körningen utnyttjat dessa platser. Med dem händer samma sak som alltid med variabler som är definierade, men inte tilldelade: De förblir oinitierade och innehåller skräp, vilket lätt kan testas genom att låta programmet skriva ut deras skräpvärden.

8.4 Kryptering av text

Vi ska nu dra lite praktisk nytta av våra samlade kunskaper om bl.a. slumptal, ASCII-koder, array, stränghantering, funktioner och referensanrop, för att med ganska enkla medel skriva en liten applikation om kryptering av text. Följande program läser in en text, skickar den till en externlagrad funktion som kan både kryptera och återställa texten med ett slumptal som krypteringsnyckel. Metoden som används för kryptering är väldigt enkel, nästan primitiv, men kan lätt ersättas av mer sofistikerade algoritmer.

```
// EncryptTest.cpp
// Skickar en text som array samt en krypteringsnyckel till
// funktionen krypt() där den krypteras. Krypteringsnyckeln
// slumpas fram vid varje körning tidsberoende. Referens-
// anrop gör den krypterade texten tillgänglig i main()
// krypt() anropas en 2:a gång med den krypterade texten och
// inverterad (neg.) krypteringsnyckel för att återställa den
#include <iostream>
#include <ctime>
using namespace std;
#include "Encrypt.h"                // Innehåller krypt()
#include "..\MyRand.h"              // Innehåller myRand()
                                    // se sid 146
int main()
{
   srand(time(0));                 // Gör slumpen tidsberoende
   char text[80];
   cout << "\nGe en text:\t\t";
   cin.getline(text, 80);
   int key = myRand(1, 1000);      // Slump-krypteringsnyckel

   krypt(text, key);               // 1:a anropet krypterar

   cout << "\nKrypterad  text:\t" << text << "\n\n";

   krypt(text, -key);              // 2:a anropet återställer
                                   // med negativ nyckel
   cout << "Återställd text:\t" << text << "\n\n"
        << "Krypteringsnyckeln:\t" << key << "\n\n";
}
```

Med en array av **char** allokeras minne för texten med en maximal längd på **80** tecken. För att kunna inkludra mellanslag i texten, läses den in med **cin.getline()** och lagras i arrayvariabeln **text**. Med ett första anrop av funktionen **krypt()**:

```
krypt(text, key);
```

som är externlagrad i filen **Encrypt.h**, överförs parametern **text** med referensanrop då den är definierad som array. Därför är ändringarna som görs med **text** i

funktionen **krypt()**, tillgängliga efter anropet. Texten är okrypterad före och krypterad efter detta anrop både i fukntionen och i **main()**. Den andra parametern **key** däremot överförs med värdeanrop då den är definierad till den enkla datatypen **int**. Efter **krypt()**:s första anrop skrivs den krypterade texten ut. Sedan anropas **krypt()** med **-key**, det negativa värdet av **key**, för att återställa texten vilken sedan skrivs ut för kontroll. Hur krypteringsnyckeln **key** och därmed hela krypteringsmetoden fungerar, förstår man bäst om man samtidigt tittar på funktionen **krypt()**:

```
// Encrypt.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 nolltecknet

void krypt(char t[], int n)
{
   for (int i=0; t[i] != '\0'; i++)
       t[i] = t[i] + n;
}
```

Krypteringsmetoden är väldigt enkel: tecknens ASCII-värden ökas med **n** i satsen **t[i] = t[i] + n;** genom vanlig addition. Att det verkligen adderas **n** till *ASCII-koden* till **t[i]** beror på att **t[i]** är av typ **char** och att en teckenvariabel i aritmetiska uttryck tolkas som sin ASCII-kod – ett tal man kan räkna med (sid 85). **for**-satsen som går igenom hela strängen genom att koppla räknaren till arrayens index, gör att hela texten förskjuts med **n** steg i ASCII-tabellen. **n** får sitt värde genom kopiering (värdeanrop) från **key** vid första och **–key** vid andra anropet. **key**:s värde i sin tur slumpas fram i **main()** med hjälp av funktionen **myRand()** vars kod finns på sid 146. Detta värde som är något heltal mellan **1** och **1000** skickas som krypteringsnyckel till **krypt()** via den andra parametern. Vid andra anropet av **krypt()** skickas **–key** för att återställa texten. Genom att ersätta **t[i] + n** med mer sofistikerade formler kan man utveckla mer avancerade krypteringsalgoritmer. Filen **MyRand.h** som innehåller funktionen **myRand()** inkluderas i **EncryptTest** med **#include "..\MyRand.h"**. Anledningen till **..** är att filen är placerad i mappen strax över den aktuella mappen för att använda den till andra program.

Programmet **EncryptTest** kan köras på olika sätt. Varje körning ger en annan slumpmässig krypteringsnyckel. Här ett exempel på en körning:

```
Ge en text:              abcd

Krypterad  text:         ÁÂÀ©

Återställd text:         abcd

Krypteringsnyckeln:      340
```

Man kan kontrollera krypteringen för hand: Man ser att bokstaven **a** förskjutits till **Á**. Krypteringsnyckeln har vid denna körning varit **340**. ASCII-koden till **a** som är 97, har förskjutits **340** steg vidare till 97 + 340 = 437 som överskrider datatypen **char**:s övre gräns. Det blir *overflow:* Värdet måste räknas *modulo* 2^m där *m* är antalet bitar i minnesutrymme som står till förfogande för den aktuella datatypen. Pga datatypen **char** är *m* = 1 byte dvs 8 bitar. Overflow-värdet 437 måste alltså räknas *modulo* 2^8 där 2^8 = 256. Men 437 *modulo* 256 är 118 dvs resten vid heltalsdivision av 437 med 256 är 118 eller: 437 % 256 = 118. På sid 114 kan man se att 118 är ASCII-koden till tecknet **Á**. Därför har **a** förskjutits till **Á** med krypteringsnyckeln **340**.

Självklart borde i en skarp applikation krypteringsnyckeln inte skrivas ut utan endast sparas i variabeln **key** för att använda den vid återställningen. Vi gör det här endast för experimentens skul.

Då krypteringsnyckeln är ett slumptal mellan **1** och **1000** som tack vare programsatsen **srand(time(0));** varierar från körning till körning, får vi en annan kryptering vid en annan körning även om vi matar in samma sträng. Prova gärna! En körning med en längre text som även innehåller mellanslag, ger:

```
Ge en text:          Längre text som skulle kunna stå i en fil

Krypterad  text:
É╚▓½Â®d©®╝©dÀ|▒dÀ»┤▒▒®d»┤▓▓ÑdÀ©╩d¡d®▓d¬¡▒

Återställd text:     Längre text som skulle kunna stå i en fil

Krypteringsnyckeln: 580
```

Lägger man till filhantering i programmet **EncryptTest** kan samma funktion **krypt()** användas för kryptering av filer. Vi kommer att göra det i nästa kapitel som tar upp filhantering.

Men först ska vi avsluta detta kapitel med ett sista avnsnitt som behandlar en utvidgning av arraybegreppet: en array vars element i sin tur är arrays.

Övningar till kapitel 8

8.1 Skriv ett program som läser in 20 heltal, lagrar dem i en array och skriver ut dem i omvänd ordning.

8.2 Skriv ett program som läser in en text, lagrar den i en array av **`char`** och skriver ut den baklänges. Utveckla en teknik för att ta reda på den aktuellt inmatade **`char`**-arrayens exakta längd.

8.3 Skriv ett program som läser in text i gemener, lagrar den i en array av **`char`** och skriver ut den framhävd i versaler och med mellanslag mellan varje tecken. Gör som i övn 8.2 angående arrayens längd.

8.4 Skriv ett program som frågar efter användarens för- *och* efternamn, hälsar sedan användaren i en utskrift med fullständiga namnet, förnamnets längd samt efternamnets första och sista bokstav. Lös uppgiften generellt utan att använda information om något speciellt för- och efternamn.

8.5 Skriv ett program där **`main()`** läser in en persons fullständiga namn och hälsar tillbaka med namnets initialer. Dessa ska bestämmas och skrivas ut i en annan funktion – med huvudet **`void initialer(char[] namn)`** – som anropas i **`main()`**.

8.6 Skriv ett program som slumpar fram 1000 heltal mellan 60 och 140 (tänkbara hastigheter på en motorväg), lagrar dem i en array **`hastighet`**, beräknar och skriver ut deras medelvärde med förklarande text och lämplig rubrik.

8.7 ***Labyrint II (forts.):*** Vidareutveckla din lösning av labyrintprojektet från *Labyrint I* (övn 6.8, sid 124). Förslagen till algoritmen där gav inte precis den mest eleganta lösningen. Nu kan vi utnyttja våra kunskaper om array för att hitta problemets programmeringstekniskt mest eleganta lösningen:

Börja med att skapa en array av **`char`** med 14 element som består av koderna till de 12 linjegrafiktecknen (LGT) som är avbildade i inlämningsuppgiften (sid 89) samt mellanslaget. Skriv sedan ut tecknen i en nästlad loop genom att låta slumpen bestämma arrayens index, t.ex. så här:

```
ch[rand() % 14]
```

där **`ch`** är namnet på den skapade **`char`**-arrayen. På så sätt löser en nästlad loop med en enda utskriftssats samt en för radbyten hela labyrintuppgiften med några få rader kod och ger den programmeringstekniskt mest eleganta lösningen.

8.8 ***Master Mind (projekt)*** är ett litet spel som låter användaren gissa ett slumpmässigt genererat fyrsiffrigt heltal genom att leda spelaren med en

inbyggd hjälpprocedur vars regler är beskrivna nedan. Även här gäller det att försöka hitta egna lösningar. Följande ska anses som ett förslag till lösning:

Börja med att behandla *fyr*siffriga heltal som en serie av *fyra en*siffriga tal dvs som en array av heltal med fyra element.

Skriv först en funktion med huvudet **`void skapa(int[] secretNo)`** som ska generera det hemliga fyrsiffriga talet och lagra det i en **`int`**-array, säg **`secretNo`**, med 4 element. Varje element i arrayen **`secretNo`** kan genereras som ett slumptal mellan 0 och 9. Dessutom ska funktionen **`skapa()`** kontrollera spelets regel enligt vilken alla fyra siffror måste vara olika.

Skriv sedan en funktion med huvudet **`void help(int[] guessedNo, int[] secretNo)`** som ska bearbeta spelarens gissning enligt följande regler:

För varje rätt siffra på rätt plats från vänster till höger skrivs ut ett	**`R`**
För varje rätt siffra på fel plats från vänster till höger skrivs ut ett	**`S`**
För varje fel siffra från vänster till höger skrivs ut ett frågetecken	**`?`**

Om t.ex. det hemliga talet är 4693 och spelaren gissar 7498, så erhålls hjälpen:

`? S R ?`

När hjälpen skriver ut **`RRRR`** har spelaren lyckats och programmet avslutas med att skriva ut ett lämpligt meddelande. Skriv ett program som tillåter flera spelomgångar.

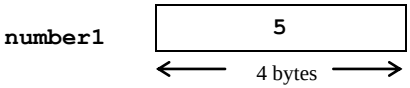
Kapitel 9

Pekare

Ämne	Sida	Program
9.1 Vad är en pekare?	180	
9.2 Deklaration och initiering av en pekare	182	**Pointer**
9.3 Adress- och värdeoperatorn	185	**Value**
9.4 Operatorn **new**	191	**New**
9.5 Pekare och array	195	**PointArray**
- Pekararitmetik	197	**PointArithm**
9.6 Stränghantering med pekare	199	**Initials**
Övningar till kapitel 9	204	

9.1 Vad är en pekare ?

Om deklaration och namngivning av variabler sa vi tidigare så här:

> ” Vad händer när variabeln **`number1`** deklareras till **`int`**?
>
> **1.** En minnescell reserveras i datorns RAM-minne för lagring av **`int`**-värden. Namnet på denna minnescell blir **`number1`** …
>
> 
>
>
> **. . .**
>
> **3.** Namngivning har med adressering att göra: Minnescellens *fysiska* adress i RAM kopplas till det *logiska* namnet **`number1`**. Det görs för att komma åt minnescellen genom att referera till variabelnamnet. Med andra ord, variabler gör minnescellerna i hårdvaran åtkomliga dvs *adresserbara* för mjukvaran.” (sid 58)

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*).

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. Det vore effektivare att kunna arbeta med adresser direkt. Ju närmare man kan komma hårdvaran från programmet desto snabbare kan datorn bearbeta data. Bland de objektorienterade högnivåspråken är C++ pga sina rötter i C, det programmeringsspråk som kommer hårdvaran närmast. C++ har flera verktyg för att kontrollera och styra hårdvaran vars ursprung går tillbaka till C. Ett av dessa verktyg är *pekare* som är ett arv som C++ tagit över av C.

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`**.

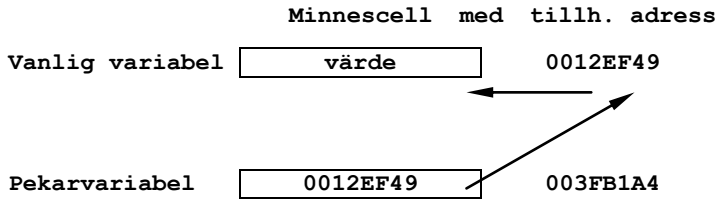
Pekare = ett nytt parallellt system

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:

Att tilldela en pekarvariabel en vanlig variabels adress



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.

9.2 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 **pe-**

karVar *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 kännetecknа 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 deklarera *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 **4** bytes. Pekarvariabler till *alla* datatyper tar alltid lika mycket i minnesutrymme som datatypen **int** gör. I den aktuella C++ installation vi använder är det **4** 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 **4** bytes. Det anmärkningsvärda är att pekare alltid tar **4** 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 (fotnot sid 83) 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.

9.3 Adress- och värdeoperatorn

Programmet **Pointer**:s ”svaghet” ur pekarsynpunkt är att tilldelningen av den vanliga variabeln till **15.6** och även utskriften fortfarande sker på konventionellt sätt – med den vanliga variabelns namn – och inte med den pekarvariabel som pekar på den. Programmet bearbetar både en vanlig och en pekarvariabel. Men kopplingen mellan dem är enkelriktad: Vi får adressen till ett värde, men inte omvänt: värdet lagrat vid en adress. För att konstruera en dubbelriktad koppling – förutsättningen för att helt och hållet gå över till kod som endast använder pekarvariabler, måste vi lära känna två operatorer som, så ofta i operatorsammanhang, t.ex. **+** och **–** eller ***** och **/**, uppträder parvis. De är motsatta till varandra i den bemärkelsen att de tar ut varandra. I C++ finns två fördefinierade operatorer varav den ena ger *adressen* till ett värde och den andra *värdet* lagrat vid en adress. Därför kallar vi dem *adress*- och *värdeoperator* och tar upp dem i följande. Den första har vi använt i sista och den andra kommer vi att demonstrera i nästa program.

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 (sammanfattning sid 59). 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.

```
// Value.cpp
// Att referera till en variabel indirekt via dess adress
// istället för konventionellt via variabelnamnet
// * i icke-deklarationssatser är värdeoperatorn som refere-
// rar till variabelns VÄRDE som pekarvariabeln pekar på
// Värde- * och adressoperatorn & är inversa operatorer
#include <iostream>
using namespace std;

int main()
{
   double vanligVar;
   double *pekarVar = &vanligVar; // * symbol för pekare
                                  // pekarVar pekar på vanligVar
   *pekarVar = 15.6;              // Här är * värdeoperatorn:
                                  // VÄRDET i den variabeln 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";
}
```

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 **4** 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:

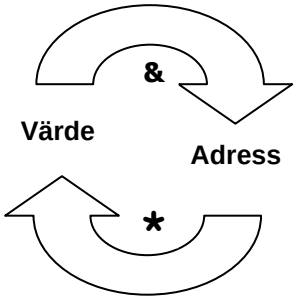
```
*pekarVar = 15.6;
```

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.

Det finns faktiskt ett intressant samband mellan dessa operatorer vars belysning bl.a. underlättar förståelsen för pekaren. Man kan memorera bättre, när man känner igen en underliggande struktur – något som bakom det rent tekniska ger ett koncept till det hela.

Värdeoperatorn * är invers (motsatt) till adressoperatorn &

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 191) 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 när vi kör programmet **Value**:

```
                  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 sluts (i bilden ovan). 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.

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.

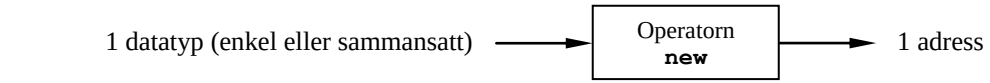
Ett annat lärorikt test är att i **Value** byta ut satsen ***pekarVar = 15.6;** mot:

```
*((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**.

9.4 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 **`new`** som 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*. Då operatorn **`new`**:s returvärde är en adress måste den tilldelas en pekarvariabel om man sedan vill referera till data som lagras vid denna adress – och det vill man ju alltid. 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;
```

T.ex.:

```
int *pekInt = new int;
```

Själva **`new`**:s syntax står höger om tilldelningstecknet. Till vänster deklareras 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.

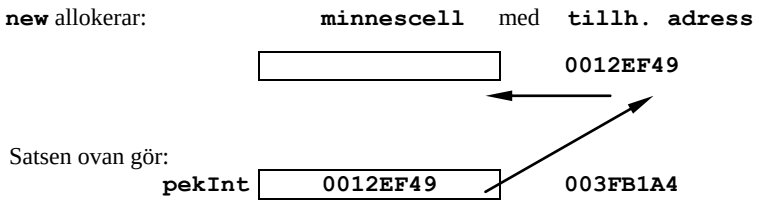
I tidigare program hänvisade vi till minnescellerna direkt med *variabelnamn*. Nu gör vi samma sak indirekt med deras *adresser*. Vilken metod som är ”direkt” och vilken ”indirekt” kan man ha olika åsikter om. När man vant sig vid att använda pekare kan man t.o.m. tycka att hanteringen av data via adresser 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 då new allokerar minne och returnerar dess adress till en pekare som används som referens till minnescellen. Värdeoperatorn tillämpad på denna pekare ger 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 datatypen 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";
}
```

Den första satsen i **main()** 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:

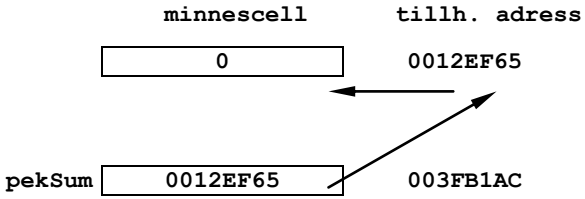
```
cin >> *pekInt;
```

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 (kapitel 12) 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.

```
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
```

Körexemplet ovan visar förresten att programmets användare inte märker att koden endast använt pekare.

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, vilket kommer att tas upp i avsnitt *Dynamisk minnesallokering* i slutet av detta kapitel.

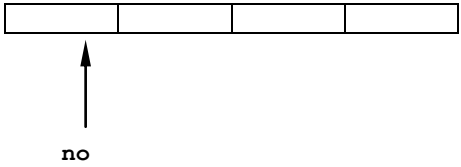
9.5 Pekare och array

Om man fortsätter att programmera med pekare i C++ upptäcker man snart att det finns en koppling mellan pekare och array. Arrayen kan beskrivas som en användarvänlig variant av pekare. I Java t.ex. har man utnyttjat detta släktskap för att avskaffa pekare. Medan javakod arbetar med arrays, objekt och referenser görs förstås all minnesallokering med adresser. Ett datorprogram kan ju inte fungera utan adressering, oavsett programmeringsspråk. I C++ har man möjligheten att till en viss grad styra adresshanteringen från koden. Att javaprogram fungerar utan att använda pekare beror på att array som en variant av pekare kan ersätta den. Man struntar då helt enkelt i vissa möjligheter av minnesmanipulation. Hur omfattande släktskapet mellan pekare och array är och vilka konsekvenser det har, ska vi studera i detta avsnitt. Samtidigt kommer vi att förstå vissa aspekter av array som vi inte kunde förklara tidigare, t.ex. att all indexering börjar med 0. Men trots stora likheter mellan pekare och array bör man hela tiden ha klart för sig att det handlar om *två olika datatyper*. Vi börjar med att skriva om programmet **`ArrayInit`** (sid 163) till en pekarversion:

```
// PointArray.cpp
// Pekarversion av programmet ArrayInit (sid 163)
// 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. Allo-
                                // kerar minne för en array
                                // av int med 4 element. no
                                // pekar på 1:a el. i arrayen.
   *no     = 64;                // Elementvis tilldelning av
                                // arrayen no.
   *(no+1) = 86;                // Pekarstegning:
   *(no+2) = 34;                // pekarens position flyttas
   *(no+3) = -6;                // fram i arrayen.
   int *copy = new int[4];      // Ny pekare copy
   for (int i=0; i <= 3; i++)   // Elementvis tilldelning av
       *(copy+i) = *(no+i);     // arrayen copy med for-sats.
   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";
}
```

Denna pekarversion följer noggrant koden i den ursprungliga arrayversionen som behandlade tilldelning av en array – en ren översättning.

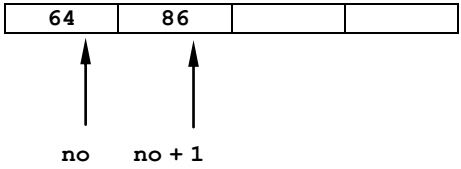
Den första satsen i **main()** deklarerar en pekare-till-**int** kallad **no**. Samtidigt allokerar **new** minne för en array av **int** med **4** element som **no** kommer att peka på. Att peka på en array innebär att peka på det första elementet i arrayen. Detta känner vi till från **Dynamic** med skillnaden att vi där hade en pekare-till-**char**. Det uppstår följande minnesbild av denna array:



som enligt deklation (sid 159) är ett sammanhängande minnesområde bestående av **4** minnesceller där varje minnescell kan lagra ett **int**-värde. Till skillnad från den ursprungliga arrayversionen **ArrayInit**, saknar denna array namn då 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. Värdeoperatorn gör det: ***no = 64;** i den andra satsen av **main()** där **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;
```

och heter *pekarstegning*. Pekarens position flyttas fram *ett* steg in i arrayen så att den pekar på den *andra* minnescellen. I koden åstadkommer man det genom att skriva **no+1**. Additionsoperatorn betyder här stegning av pekarens position. Efter den här stegningen tillämpas värdeoperatorn på den nya positionen: ***(no+1)** så att det nya uttrycket refererar till innehållet av den minnescell som **no+1** pekar på. Slutligen läggs värdet **86** i denna minnescell. Vid det här stadiet ser minnesbilden ut så här:



Pekarstegning ersätter alltså indexering. Vi kan istället för index hos arrays, använda oss av stegning med pekare för att komma åt alla element i en array. De övriga elementen tilldelas på samma sätt.

Arraynotation vs. pekarnotation

Vad gäller åtkomsten till arrayelementen kan generellt en aspekt av släktskapet mellan array och pekare beskrivas 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 och ***** är värdeoperatorn. Koderna ovan – man pratar om *arraynotation* resp. *pekarnotation* – är ömsesidigt utbytbara. 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)**

där arraynotationen visar det *första* arrayelementet. Men vad gäller pekarnotationen är ***(no + 0)** identiskt med ***(no)** och därmed med ***no**. Detta känner vi till som åtkomst till den minnescell som **no** pekar på och som vi redan i satsen ***no = 64;** använt för att tilldela det *första* arrayelementet.

Varför börjar arrayindex med 0 ?

Nu kan vi också förklara indexregeln för arrays (sid 160) 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.

Efter tilldelningen av arrayen **no** skapas i **PointArray** på samma sätt en annan array som pekaren **copy** pekar på. Värdena i **no** kopieras till **copy** med en **for**-sats. Slutligen skrivs värdena i **copy** ut – med pekarnotation som man kan byta ut mot arraynotation (står bekrivet redan i **ArrayInit**):

```
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
```

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 an-

vändarvänliga 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;
```

9.6 Stränghantering med pekare

Släktskapet mellan pekare och array som vi började att studera i förra avsnittet (sid 195) blir ännu mer påtagligt när man vill hantera strängar med datatypen pekare-till-**char**. Att man kan göra så beror på att strängar i C++ är teckenkedjor som avslutas med nolltecknet. Man låter en pekare peka på en **char** som blir början på en sträng och sätter nolltecknet till slutet på strängen. Stränghantering är av generellt intresse bl.a. därför att den bildar grunden för filhantering. Men det som gör stränghanteringen med pekare mer intressant än med array, är kombinationen med pekararitmetik och möjligheten till dynamisk minnesallokering. Båda dessa koncept kan endast realiseras med pekare.

I följande program ska vi använda våra kunskaper om pekare-till-**int** och pekararitmetik för att lära känna den nya datatypen pekare-till-**char** och låta programmet hitta initialerna till vårt för- och efternamn – en sträng som initialt lagras i en **char**-array. 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 resulterar i en någorlunda enkel men samtidigt elegant kod: **while**-loopen som går igenom strängen behöver ingen extra indexvariabel. Ökningsoperatorn **++** stegar pekaren och går så igenom strängen.

```
// Initials.cpp
// Läser in för- och efternamn, skickar strängens pekare till
// funktionen initialer() som hittar och skriver ut initia-
// lerna. Stränghantering med datatypen pekare-till-char
// Dessutom: Pekare som parameter i en funktion
#include <iostream>
using namespace std;

void initialer(char *namn)  // Funktionsdefinition med pekare
{                           // 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;
   initialer(dittNamn);      // Funktionsanrop med array som
   cout << "\n";             // skickas som aktuell parameter
}
```

Men innan vi kommer till **while**-loopen som är programmets hjärta, ska vi titta på hur strängen hamnar i denna loop: Självklart via funktionsanropet

```
initialer(dittNamn);
```

från **main()** där vi definierar arrayen **dittNamn** och läser in till den för- och efternamn med **cin.getline()** för att få in även mellanslaget mellan för- och efternamn. Vid anropet överförs arraynamnet **dittNamn** till pekarvariabeln **namn** dvs en kopiering av den aktuella parametern **dittNamn**:s värde till den formella parametern **namn** utförs som man kan jämföra med satsen:

```
char *namn = dittNamn;
```

där pekarens deklaration redan står i parameterlistan till funktionen **initialer()** och tilldelningen görs vid parameteröverföringen. Men hur är det möjligt att ett arraynamn kan tilldelas som värde till en pekarvariabel? Endast om arraynamnet är en adress! Satsen ovan som endast är ett förtydligande av parameteröverföringen vid anrop av funktionen **initialer()**, avslöjar än en gång släktskapet mellan pekare och array:

En array är en konstant pekare.

Man kan alltid tilldela ett arraynamn till en pekarvariabel då arrayen är ett fast adressvärde. En gång skapat kan det inte ändras i ett program. En pekare däremot är en variabel som kan initieras till ett adressvärde, ändra sedan värdet och ta ett nytt värde. Att vi hittills – speciellt i arraykapitlet – ändå pratat om array*variabler* syftar snarare på arrayens *element*, inte på namnet. Arrayelementen är variabla och kan ändra sina värden. De lagras däremot alltid vid samma adress som i programmet representeras av arraynamnet.

Men hur hittar programmet initialerna till för- och efternamnet? Efter anropet av funktionen **initialer()** 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 då ***namn** är värdet som **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 – mellanslaget 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 öknings-

operatorn **++** där 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 och på så sätt går igenom strängen. Observera att detta är pekararitmetik, närmare bestämt pekaraddition (sid 197). 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? Om det är i loopen, är det före eller efter loopens **if**-sats? Eftersom vi använt ökningsoperatorns postfixvariant kan det inte vara före … . Faktum är att pekarstegningen görs *efter att* **while**-*satsens villkor testats* dvs i början av varje varv av **while**-loopen strax innan **if**-satsen. Ett bevis på detta påstående får man 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);
}
```

och får exakt samma resultat som med programmet **Initials**, nämligen:

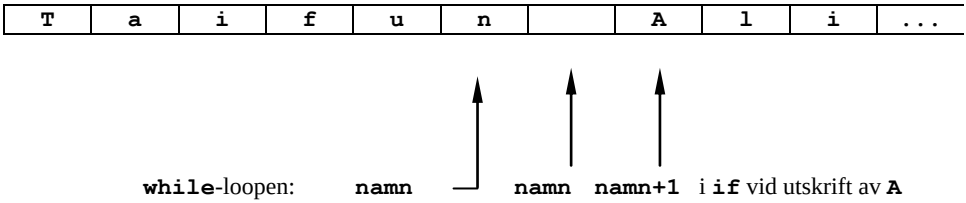
```
Ge ditt för- och efternamn: Taifun Alishenas

Hej Taifun Alishenas, dina initialer är:

        TA
```

Första initialen **T** skrivs ut i funktionen **initialer()** 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 till **while**-satsen. I varje varv av loopen testar **if**-satsen om det aktuella tecknet ***namn**, är mellanslag. 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 och därmed den andra initialen. Observera att **while**-satsen endast söker igenom förnamnet. Den använder ingen information om strängens längd. Därför behöver vi inte heller skicka arrayens storlek till funktionen.

Låt oss i tankarna frysa programkörningen till det kritiska ögonblick där den andra initialen **A** skrivs ut. Vid denna utskrift står pekaren **namn** på en position som pekar på mellanslaget mellan för- och efternamn. Då är nämligen 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 detta sista varv av loopen. Anledningen till det är att mellan **while**-villkorets test och **if**-villkoret står pekarstegningen **namn++** oavsett om den är inbakad i **while**-villkoret som i programmets originalversion eller separat i början av loopen som i bevisversionen av **while**-satsen på förra sidan. Mellan **while**- och **if**-villkoret flyttas pekaren ett steg. Dvs **while** testas när pekaren pekar på bokstaven **n** och därmed skilt från mellanslag. **if** testas när pekaren pekar på mellanslag. Efter **if**-satsen är **while**-satsens villkor falskt när det testas 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 **85initialer**.

I **while**-satsens sista varv har vi följande minnesbild av arrayen när det aktuella tecknet vid genomsökningen av arrayen är förnamnets sista bokstav. Då testas nämligen **while**-loopen när pekaren **namn** pekar 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:



Släktskapet mellan pekare och array som vi började det här avsnittet med, kan även beskådas i programexemplet om man man skriver om hela funktionen **initialer()** till arraynotation:

```
void initialer(char namn[])
{
   int i=0;
   cout << namn[0];
   while (namn[i++] != ' ')
      if (namn[i] == ' ')
         cout << namn[i+1];
}
```

Resten av programmet **`Initials`** dvs koden i **`main()`** kan bibehållas oförändrat och kommer att fungera med arrayvarianten av funktionen **`initialer()`** ovan utan problem – ett resultat av modularisering. Jämför man den med pekarvarianten 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 pga sin kompakthet och elegans av folk som har lite längre erfarenhet av programmering.

Övningar till kapitel 9

9.1 Deklarera en vanlig variabel **a** och en pekarvariabel **p**, låt **p** peka på **a**, läs in ett värde till **a**, men skriv ut värdet via pekaren.

9.2 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.

9.3 Lös övn 9.2 ovan utan en enda vanlig variabel, endast med pekare.

9.4 Skriv en pekarversion av programmet **ArrayDef** (sid 161). Skapa arrayen med **int *no = new int[4];** och ersätt överallt *index* med *pekarposition*.

9.5 Skriv en pekarversion av **ArrayChar** (sid 167). Lägg till efter **sizeof** som mäter strängens minnesutrymme, kod som med **strlen()** även mäter strängens längd. Varför returnerar **sizeof** och **strlen()** olika värden?

9.6 Skriv ett program som läser in en sträng med pekare och skriver ut den i omvänd ordning.

9.7 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.

9.8 Modularisera programmet från övn 9.3 ovan genom att låta summan beräknas i en funktion. För att illustrera referensanrop med pekare nollställ summans värde i **main()**, skicka det som pekare (parameter) till funktionen där värdet ändras och skriv ut det ändrade värdet från **main()**.

9.9 ***Palindrom (projekt)*** Skriv en funktion **bool palindrom(char*)** som avgör om en sträng är en *palindrom* eller ej. T.ex. är *rar*, *död* och *radar* palindromer då de inte ändras när de läses baklänges. Men även en text som *ni talar bra latin* är en palindrom om man ignorerar mellanslagen. Vid behandling av sådana texter låt programmet först ta bort alla mellanslag.

9.10 ***Master Mind (forts.)*** Skriv en pekarversion av projektet *Master Mind* (sid 177).

Kapitel 10

Klasser

Ämne	Sida	Program
10.1 Vägen från C till C++	206	`All_in_main`
	207	`Procedure`
- Vår första klass	208	`Circle`
- Test av klass	211	`Obj_Oriented`
- Objekt och klass	212	
10.2 Klass som egendefinierad datatyp	214	
- Deklaration av en klass	215	`Employee`
- Definition av ett objekt	217	`EmployeeTest`
- Åtkomst till objektets medlemmar	220	
10.3 Metoder	223	`TravelTime`
- Objekt som parameter och returvärde	224	`Travel_Test`
Övningar till kapitel 10 (Projekt *Automaten*)	228	

10.1 Vägen från C till C++

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Vår första klass

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

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

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

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

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

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

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

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

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

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

Klasskonceptet har förutom modelleringsaspekten följande fördelar i koden:

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

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

Test av klass

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

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

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

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

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

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

```
Circle myCircle;
```

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

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

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

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

```
Ange radien till en cirkel: 1

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

Objekt och klass

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

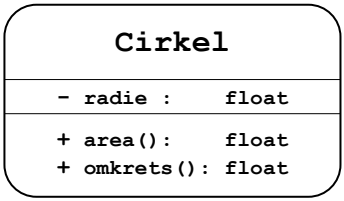
Ett av syftena med objektorienterad programmering är att efterlikna verkligheten så mycket som möjligt. Man vill kunna avbilda den reala världen, konstruera en modell av den i sina datorprogram för att kunna simulera verkligheten så noggrant som det går. Den reala världen, åtminstone den del som tillåter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

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

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

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

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



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

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

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

10.2 Klass som egendefinierad datatyp

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

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

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

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

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

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

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

1. Deklaration av en klass

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

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

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

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

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

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

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

   double lon();
};
```

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

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

#include "Employee.h"                    // Inkluderar klassen

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

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

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

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

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

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

2. Definition av ett objekt

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

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

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

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

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

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

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

```
Anstalld anst;
```

som kan jämföras med:

```
int a;
```

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

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

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

```
Anstalld copy = anst;
```

precis som hos vanliga variabler:

```
int b = a;
```

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

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

Datatyptest med `sizeof`

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

```
sizeof(Anstalld)
```

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

```
        Anders Larsson, personnr 590714-2493

        Gammal lön:     21450

        Ny lön:         24667.5

        Våra lönekostnader kommer att öka med 3217.5 kr

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

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

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

3. Åtkomst till objektets medlemmar

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

Punktnotation

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

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

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

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

Då ser punktnotation ut så här:

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

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

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

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

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

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

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

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

Anrop av metoder

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

```
anst.lon()
```

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

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

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

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

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

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

10.3 Metoder

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

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

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

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

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

Objekt som parameter och returvärde

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

Två programmeringstekniskt nya moment kan observeras här:

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

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

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

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

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

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

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

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

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

```
// 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\n";
}
```

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

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

analogt till:

```
tisdag "+" onsdag
```

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

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

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

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

1. Upprepat eller kedjeanrop.
2. Nästlat anrop.

1. Den första möjligheten används i programmet **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 **tvadagsresa**ns och **torsdag**ens restider och lägger resultatet i **tredagsresa**.

2. Den andra, alternativa möjligheten för att addera flera restider med **sum()** är nästlat anrop som i programmet **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:

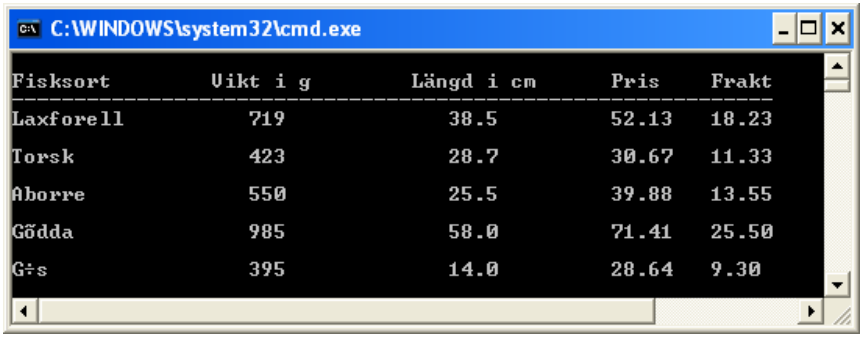
```
Ange timmar och minuter till tisdagsresan: 3 45
Ange timmar och minuter till onsdagsresan: 5 25

        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 10

10.1 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.

10.2 Modifiera klassen **`Rektangel`** från övn 10.1 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?

10.3 Skriv en klass **`Fish`** 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 **`Fish`** i en separat fil genom att i **`main()`** skapa en array av fem **`Fish`**-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 **`Fish`**-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:



10.4 Deklarera datamedlemmarna i klassen Fish från övn 10.3 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 10.3.

10.5 Modellera en klass **Cylinder** som subklass till klassen **Circle**. Förse superklassen **Circle** med en privat datamedlem **radie**, en konstruktor, en get-metod och med beräkningsmetoderna **area()** och **omkrets()**. Denna modellering ser **Cylinder**n som en **Circle** som dessutom har en **höjd**. Betrakta därför **Cylinder**n som en ”utvidgad” **Circle** som ärver **Circle**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 **Circle**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()**.

10.6 ***Anställd (projekt)*** 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.

10.7 ***Automaten (projekt)*** Skriv ett program som simulerar interaktionen med en automat. Följande klass beskriver det som pågår i automaten efter att användaren lagt in pengar och valt en vara:

```
class Automataktion
{
   string varunamn;
   float  pris;
   float  betalning;
   float  vaxel;
```

```
public:
   Automataktion(float pengar, char vara)
   {


      switch(vara)
      {
         . . .
      }
      betalning = pengar;
      vaxel = betalning - pris;
   }

   void vaxel_i_mynt()
   {
         . . .
   }
};
```

switch-satsen i konstruktorn ska tilldela datamedlemmarna **varunamn** och **pris** värden beroende på valet av vara och skriva ut ett meddelande om inlagt belopp samt varan som ska levereras. Metoden **vaxel_i_mynt()** är till för att dela upp växeln i ”tillåtna” myntslag (endast 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 det sista, använd en algoritm.

Börja i **main()** med att skriva en meny över alla varor samt priserna. Låt sedan användaren lägga in pengar. Läs in beloppet till en **float**-variabel. Låt användaren även välja en vara. Sedan kan ett objekt av den ovan deklarerade klassen **Automataktion** skapas och samtidigt konstruktorn **Automataktion()** anropas. Vid detta anrop skickas de inlästa värdena till det inlagda beloppet och den valda varan som aktuella parametrar till **Automataktion()**. Efter att objektet skapats och datamedlemmarna initierats via konstruktorn, kan metoden **vaxel_i_mynt()** anropas.

Komplettera programmet med att ta hand om en eventuellt felaktig eller otillräcklig betalning från användarens sida.

Lösningsförslag till övningar (Facit) *

Övningar till kap 1 Programmeringsmiljön, sid 20

Övning 1.1 är en praktisk uppgift som löses enligt instruktioner i övningen.

Övningar till kap 2 Introduktion till programmering, sid 31

Övningarna till kap 2 består av frågor. Svar till alla frågor finns i boken, sid 20-28.

Övningar till kap 3 Att komma igång med C++, sid 46

Ovn_3_1
Undersök skillnaderna mellan tangenterna apostrof, citationstecken och accent. Mata in koden till programmet MyFirst (sid 34), kompilera och kör. Modifiera sedan programmet MyFirst genom att ta bort using-satsen före main() och ersätta istället i main() alla cout med std::cout (läs om detta på sid 34). Kompilera och kör.

```
#include <iostream>

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

```
Skillnaderna mellan tangenterna apostrof, citationstecken och accent:
Apostrof är                '            (Tillsammans med * tangenten)
Citationstecken är         "            (Tillsammans med 2:ans tangent)
Två typer av accent är     ´ och `      (Till höger om   + tangenten)
**********************************************************************
```

Ovn_3_2
I MyFirst finns fyra cout-satser. Skriv om programmet så att det förekommer först tre och sedan två cout-satser i det. Tips: använd \n i strängarna. Modifiera ytterligare genom att snygga till layouten så att texten hamnar med lite avstånd från konsolens övre och vänstra rand.

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

int main()
{
   cout << "\n\tDetta är mitt första C++ program!\n";
   cout << "\tDet skriver ut två rader text på skärmen.\n\n";
}
**********************************************************************
```

Ovn_3_3 *Sätt in följande kod i ett C++ program för att testa vad den ger för utskrift:*

* För att uppmuntra till egna lösningar ges inga lösningsförslag till projektuppgifterna (t.ex. *Labyrinten, Automaten*, *Mater Mind* osv.), inte heller till övningar som är relaterade till ett projekt eller delar av det. Som kompensation finns det i projektenas lydelse alltid en ganska utförlig ledning till lösningen.

```
cout << "****\n";
cout << "*****\n";
cout << "******\n";
cout << "*******\n";
cout << "********\n";
cout << "*******\n";
cout << "******\n";
cout << "*****\n";
cout << "****\n";
```

Konkatenera alla satser till en enda cout-sats så att du får samma utskrift.

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

int main()
{
   cout << "****\n"
           "*****\n"
           "******\n"
           "*******\n"
           "********\n"
           "*******\n"
           "******\n"
           "*****\n"
           "****\n";
}
```

*****

Ovn_3_4

Skriv ett program och testa vilken utskrift följande satser ger:

```
cout << "Jag";
cout << "heter";
cout << "K.\n Vad heter du? ";
```

Lägg till och ta bort mellanslag, radbyte och tabulator på lämpliga ställen för att få en snygg utskrift, utan att slå ihop de tre cout-satserna till en.

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

int main()
{
   cout << "\n\tJag ";
   cout << "heter ";
   cout << "K.\n\tVad heter du?\n\n";
}
```

*****

Ovn_3_5

Vilken utskrift ger följande satser? Sätt in dem i ett program.

```
cout << "*\n**\n***\n****\n*****";
cout << "*****\n****\n***\n**\n*";
```

Skriv om koden så att du får samma utskrift med en enda utskrifts-

sats i koden.

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

int main()
{
   cout << "*\n**\n***\n****\n*****"
           "\n*****\n****\n***\n**\n*\n";
}
```

****

Ovn_3_6_Oval

Mata in koden till programmet Figure (sid 45), kompilera och kör det. Modifiera det till att skriva ut formen av en oval, en triangel, en rektangel, en pil, en romb, ett kryss, en stjärna eller en blomma, av stjärnor (*).

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

int main()
{
   cout << "\n        * * * * * * * * * *        "
           "\n     *                      *     "
           "\n    *                        *    "
           "\n   *                          *   "
           "\n   *                          *   "
           "\n   *                          *   "
           "\n    *                        *    "
           "\n     *                      *     "
           "\n       * * * * * * * * * *        "
           "\n\n");
}
```

****

```
// Ovn_3_6_Triangel
#include <iostream>
using namespace std;

int main()
{
   cout << "\n                *          "
           "\n               * *         "
           "\n              *   *        "
           "\n             *     *       "
           "\n            *       *      "
           "\n           *         *     "
           "\n          *           *    "
           "\n         *             *   "
           "\n        *               *  "
           "\n       * * * * * * * * * * "
           "\n\n";
}
```

****

Ovn_3_7

Rita följande figur med en enda utskriftssats med konkatenering. Se

upp för skillnaden mellan slash / och backslash \. Använd två backslash i koden – som en escapesekvens inbakad i den konkatenerade strängen – för att åstadkomma en backslash \ i utskriften (Läs om escapesekvenser på sid 86).

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

int main()
{
   cout << "\n          ______            "
           "\n         /_____/\\          "
           "\n        /____ \\\\ \\       "
           "\n       /_____\\ \\\\ /      "
           "\n      /_____/ \\/ / /       "
           "\n     /_____/ /   \\//\\     "
           "\n     \\_____\\//\\   / /    "
           "\n      \\_____/ / /\\ /      "
           "\n       \\_____/ \\\\ \\     "
           "\n        \\_____\\ \\\\      "
           "\n         \\_____\\/         "
           "\n\n\n";
}
```

****

Övningar till kap 4 Grundbegrepp i programmering, sid 72

Ovn_4_1

Satsen cout << a; ger kompileringsfel till skillnad från cout << 'a'; Sätt in båda i ett C++ program och testa. Ger även cout << 6; kompileringsfel? Testa vilka utskrifter följande satser ger:

```
cout << 6 << 6;
cout << '6' << '6' ;
cout << (6 + 6);
cout << ('6' + '6');
cout << 6.6 << 6.6 ;
cout << (6.6 + 6.6);
cout << "6.6" << "6.6";
```

Skriv om koden så att du får samma utskrift med en enda utskriftssats i koden. Lägg till och ta bort mellanslag, radbyte och tabulator på lämpliga ställen för att få en snygg utskrift. Förklara!

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

int main()
{
// cout << a;                                  // Ger kompileringsfel
   cout << "\n\t" << 6 << 6                    // Heltalkonstanter
        << "\n\n\t" << '6' << '6'              // Teckenkonstanter
        << "\n\n\t" << (6 + 6)                 // Addition av 6 med 6
        << "\n\n\t" << ('6' + '6')             // Addition av 54 med 54
        << "\n\n\t" << 6.6 << 6.6              // Decimaltalkonstanter
        << "\n\n\t" << (6.6 + 6.6)             // Addition av 6.6 med 6.6
        << "\n\n\t" << "6.6" << "6.6"          // Konkatenering
        << "\n\n";
}
```

Ovn_4_2
Komplettera programmet Variable (sid 55) så här:
*Definiera ytterligare variabler, säg diff, prod, div, mod, tilldela till dem uttryck (sid 64) bildade med de andra räknesätten -, *, / och %. Skriv ut resultaten med meningsfulla utskrifter.*

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

int main()
{
   int no1, no2, sum, diff,                  // Deklaration
       prod, div, mod;                       // av variabler
   no1 = 5;                                  // Initiering av variabler
   no2 = 3;
   sum  = no1 + no2;                         // Addition
   diff = no1 - no2;                         // Subtraktion
   prod = no1 * no2;                         // Multiplikation
   div  = no1 / no2;                         // Heltalsdivision
   mod  = no1 % no2;                         // Modulo: resten vid div.
   cout << "\n\t"<< no1 << " + " << no2 << " ger " << sum  << "\n\t"
                << no1 << " - " << no2 << " ger " << diff << "\n\t"
                << no1 << " * " << no2 << " ger " << prod << "\n\t"
                << no1 << " / " << no2 << " ger " << div  << "\n\t"
                << no1 << " % " << no2 << " ger " << mod  << "\n\n";
}
```

****

Ovn_4_3
Vidareutveckla din lösning till övn 4.2 genom att ersätta den hårdkodade initieringen av variablerna no1 och no2 med en initiering genom inläsning som t.ex. kan göras med en cin-sats samt ledtext.

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

int main()
{
   int no1, no2, sum, diff, prod, div, mod;              // Ledtext:
   cout << "Mata in två heltal (skilda med mellanslag):\t";
   cin >> no1 >> no2;                                    // Inläsning
   sum  = no1 + no2;
   diff = no1 - no2;
   prod = no1 * no2;
   div  = no1 / no2;
   mod  = no1 % no2;
   cout << "\n\t"<< no1 << " + " << no2 << " ger " << sum  << "\n\t"
                << no1 << " - " << no2 << " ger " << diff << "\n\t"
                << no1 << " * " << no2 << " ger " << prod << "\n\t"
                << no1 << " / " << no2 << " ger " << div  << "\n\t"
                << no1 << " % " << no2 << " ger " << mod  << "\n\n";
}
```

****

Ovn_4_4
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. Utveckla programmet vidare med ytterligare räkneoperationer, kanske så småningom till en liten kalkylator.

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

int main()
{
   int no1, no2;                                          // Ledtext:
   cout << "\n\tMata in två heltal (skilda med mellanslag):\t";
   cin >> no1 >> no2;                                     // Inläsning

   cout << "\n\t"
        << no1 << " plus "   << no2 << " är " << (no1+no2) << "\n\t"
        << no1 << " minus "  << no2 << " är " << (no1-no2) << "\n\t"
        << no1 << " gånger " << no2 << " är " << (no1*no2) << "\n\t"
        << no1 << " heltalsdividerad med "   <<
                                 no2 << " är " << (no1/no2) << "\n\t"
        << no1 << " modulo " << no2 << " är " << (no1%no2) << "\n\n";
}
```

***\

```
                                        // fel: Tilldelning saknas
   cout << "har ökats med 1 i programmet och är nu " << x << ".\n\n";
}
```

*****

Ovn_4_7

Skriv ett program som frågar efter aktuell veckodag (t.ex. idag). Man ska mata in en siffra där veckodagarna numreras stigande från 1 med början på måndag så att söndag blir den 7:e veckodagen. Sedan ska programmet fråga, typ "När vill du träffa din kompis?" och begära som svar ett antal dagar. Programmet ska beräkna och skriva ut den planerade träffens veckodag, dvs vilken veckodag det blir om man lägger till antalet dagar till aktuell veckodag.

```
#include <iostream>
using namespace std;
int main()
{
   int  idag, antalDagar, veckodag;

   cout << "\nVilken veckodag har vi idag?\n"
           "Svara med en siffra (mån=1, ..., sön=7):\t";
   cin  >> idag;
   cout << "\nOm hur många dagar vill du träffa din kompis?   ";
   cin  >> antalDagar;

   veckodag = (idag + antalDagar) % 7;

   cout << "\nDu kommer att träffa din kompis veckodag\t"
        << veckodag << "\n\n";
}
```

*****

Ovn_4_8

Skriv ett program som omvandlar tiden i antal år, månader och veckor till antal dagar. Läs in tre heltal till antal år, månader och veckor. Beräkna och skriv ut sedan användarvänligt hur många dagar det blir totalt. Använd strukturen inmatning – bearbetning – utmatning enligt programmet Hour2Sec (sid 64).

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

int main()
{
   int  år, månader, veckor, dagar, totaldagar;

   /* I n m a t n i n g */
   cout << "\n\tAnge antal år:\t\t";
   cin >> år;
   cout << "\n\tAnge antal månader:\t";
   cin >> månader;
   cout << "\n\tAnge antal veckor:\t";
   cin >> veckor;
   cout << "\n\tAnge antal dagar:\t";
   cin >> dagar;
```

```
   /* B e a r b e t n i n g */
   totaldagar = 365*år + 30*månader + 7*veckor + dagar;

   /* U t m a t n i n g */
   cout << '\n' << år << " år, "        << månader << " månader, "
        << veckor     << " veckor och " << dagar   << " dagar är "
        << totaldagar << " dagar totalt.\n\n";
}
```

```
*********************************************************************
```

Ovn_4_9

Vänd på problemet från övn 4.8: Skriv ett program som läser in ett antal dagar, omvandlar det till antal år, månader, veckor samt resterande dagar och skriver ut resultatet.

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

int main()
{
   int år, månader, veckor, restdagar, totaldagar;

   /* I n m a t n i n g */
   cout << "\n\tAnge antal dagar:\t\t";
   cin >> totaldagar;

   /* B e a r b e t n i n g */
   år        =   totaldagar / 365;
   månader   =  (totaldagar % 365) / 30;
   veckor    = ((totaldagar % 365) % 30) / 7;
   restdagar = ((totaldagar % 365) % 30) % 7;

   /* U t m a t n i n g */
   cout << "\n\t"    << totaldagar   << " dagar är " << år << " år, "
        << månader   << " månader, " << veckor << " veckor och "
        << restdagar << " dagar.\n\n";
}
```

```
*********************************************************************
```

Ovn_4_10

Tillämpa den logiska strukturen i algoritmen och pseudokoden till övn 4.9 för att lösa följande uppgift:

Efter inköp av en vara i en automat ska växeln ges tillbaka i form av ett antal föreskrivna myntslag: 10-kronor, 5-kronor, 1-kronor, 50-öringar (och en rest i ören). Skriv ett program som läser in ett växelbelopp i ören, omvandlar växeln till ett antal 10-kronor, 5-kronor, 1-kronor och 50-öringar samt skriver ut resultatet.

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

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

```
   cin >> amount;

   vaxel = (int) (amount * 100.001);                 // växel som int
   tio   =    vaxel / 1000;                          // 10-kronor
   fem   =   (vaxel % 1000) / 500;                   // 5-kronor
   en    =  ((vaxel % 1000) % 500) / 100;           // 1-kronor
   halv  = (((vaxel % 1000) % 500) % 100) / 50;     // 50-öringar
   ore   = (((vaxel % 1000) % 500) % 100) % 50;     // rest i öre

   cout << '\n' << amount << " kr =\t"
        << 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\n";
}

*********************************************************************
```

Övningar till kap 5 Enkla datatyper, sid 88

Ovn_5_1

Skriv ett program som läser in tre tecken och skriver ut dem i omvänd ordning.

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

int main()
{
   char letter1, letter2, letter3;
   cout << "\n\tMata in tre letter:\t\t";
   cin >> letter1 >> letter2 >> letter3;

   cout << "\nDe inmatade tecknen i omvänd ordning:\t" << letter3
        << ' ' << letter2 << ' ' << letter1 << "\n\n" ;
}

*********************************************************************
```

Ovn_5_2

Skriv ett program som läser in en gemen (liten bokstav) och skriver ut dess versal (liten bokstav) och sedan läser in en versal och skriver ut dess gemen.

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

int main()
{
   char letter;
   cout << "\nMata in en gemen:\t";
   cin >> letter;
   cout << "\nDess versal är:\t\t" << (char)(letter - 32) << '\n';

   cout << "\nMata in en versal:\t";
   cin >> letter;
```

```
   cout << "\nDess gemen är:\t\t" << (char)(letter + 32) << "\n\n";
}
```

****

Ovn_5_3

Experimentera med programmet Int2char (sid 82) för att få reda på ASCII-koden till datorns ljudsignal. Skriv ett program som genererar ljudsignal.

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

int main()
{
   cout << '\007';                // ASCII-koden till ljudsignal är 7
   cout << "\n\tHörde du två ljudsignaler?\n\n";
   cout << '\a';                  // Symboliskt alternativ till \007
}
```

****

Ovn_5_4

Kryptering av tecken: Skriv ett program som läser in ett tecken och förskjuter det i teckentabellen med ett visst antal steg (krypteringsnyckeln). Skriv ut både det inlästa och det förskjutna tecknet på ett användarvänligt sätt. Börja med att hårdkoda krypteringsnyckeln och fortsätt med att läsa in den.

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

int main()
{
   char letter;
   int step;

   cout << "\n\tMata in ett tecken:\t\t\t\t\t";
   cin >> letter;
   cout<<"\n\tMata in ett heltal (används som krypteringsnyckel):\t";
   cin >> step;
   cout << "\nDet inlästa tecknet " << letter;
   letter += step;
   cout << " har krypterats (med " << step
        << " steg) och är nu:\t" << letter << "\n\n";
}
```

****

Ovn_5_5

Kryptering av ord: Skriv ett program som läser in fem tecken och skriver ut dem förskjutna med ett steg i ASCII-tabellen så att t.ex. inmatningen Kalle ger utskriften Lbmmf. Återställ och skriv ut det krypterade ordet. Vidareutveckla programmet genom att utöka och läsa in antalet steg som en krypteringsnyckel.

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

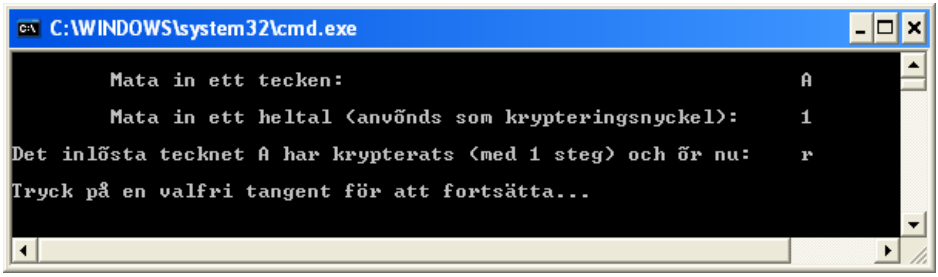
```
int main()
{
   char letter1, letter2, letter3, letter4, letter5;
   int key;

   cout << "\nMata in 5 tecken utan mellanslag:\t";
   cin >> letter1 >> letter2 >> letter3 >> letter4 >> letter5;
   cout << "\nMata in ett heltal (krypteringsnyckel):\t";
   cin >> key;

   letter1 += key;
   letter2 += key;
   letter3 += key;
   letter4 += key;
   letter5 += key;
   cout << "\nDet krypterade ordet:\t\t\t"
        << letter1 << letter2 << letter3 << letter4 << letter5;

   letter1 -= key;
   letter2 -= key;
   letter3 -= key;
   letter4 -= key;
   letter5 -= key;
   cout << "\n\nDet återställda ordet:\t\t\t"
    << letter1 << letter2 << letter3 << letter4 << letter5 << "\n\n";
}
```

****

Ovn_5_6

Ta fram din lösning till övn 5.4, ändra datatypen till variabeln step (krypteringsnyckeln) från int till char. Undersök vad som händer när du testkör programmet med denna ändring. Förklara resultatet.

*När man i lösningen till övn 5.4 på förra sidan (**Ovn_5_4**) definierar variabeln **step** som **char** istället för **int**, kan man kompilera utan problem. Men när man exekverar får man t.ex. följande felaktig utskrift:*



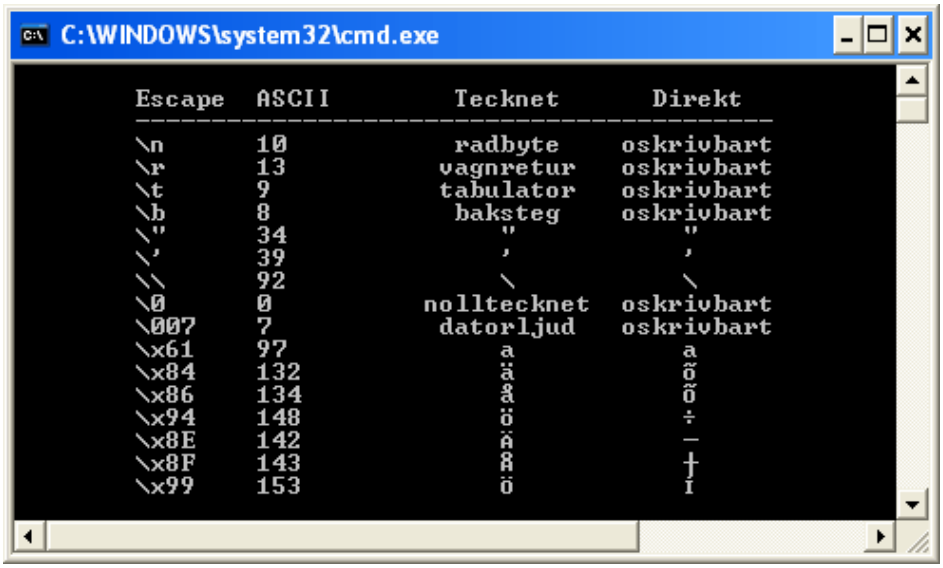
*Det felaktiga i utskriften ovan är att **r** inte är tecknet som i ASCII-tabellen följer **A** dvs när man förskjuter med steget **1**. Rätt svar borde vara **B** vilket också ges av programmet **Ovn_5_4** om man kör det i ursprungligt skick, dvs definierar variabeln **step** som **int**. Anledningen till den felaktiga utskriften är följande: Definitionen av variabeln **step** som **char** gör att i körexemplet ovan värdet 49 (ASCII-koden till tecknet **1**) adderas till 65 (ASCII-koden till tecknet **A**)*

vilket resulterar i 114 (ASCII-koden till tecknet **r***). Detta pga satsen* **letter += step;** *som är identisk med* **letter = letter + step;** *Att ASCII-koden till* **1** *tas här och inte talet* **1** *beror på att* **char***-variabeln* **step** *lagrar tecknet* **'1'** *- och därmed dess ASCII-kod 49 - och inte talet* **1***. Det i sin tur beror på att satsen* **cin >> step;** *som läser in värdet för* **step***, tolkar inmatningen pga vår definition som en* **char** *och därför lagrar ASCII-koden 49 i minnescellen för* **step** *och inte heltalet 1. Intelligensen för denna tolkning finns i den förprogrammerade koden för* **cin** *i biblioteket* **iostream.**

****

Ovn_5_7

Skriv ett program som skriver ut en tabell med fyra kolumner över alla escapesekvenser du känner till. Tabellens första kolumn ska visa själva escapesekvensen i C++ kod. Den andra kolumnen ska visa tecknets ASCII-kod. I den tredje kolumnen ska stå tecknet själv eller, om det är oskrivbart, tecknets betydelse. Den fjärde kolumnen ska visa hur de skrivbara tecknen skrivs i konsolen direkt, inte med escapesekvens. Om de är oskrivbara ska stå där oskrivbart. Även de svenska specialtecknen ä, å, ö, Ä, Å och Ö ska vara med i tabellen samt någon vanlig bokstav, säg a. Med hjälp av programmen Char2int (sid 81) och Escape (sid 86) kan du ta reda på de svenska specialtecknens ASCII-koder och escapesekvenser. Så här kan utskriften av ett sådant program se ut:




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

int main()
{
 cout << "\n\tEscape  ASCII        Tecknet      Direkt \n"
         "\t-----------------------------------------\n"
 "\t\\n     \t" << (int) '\n'   << "\t     radbyte\toskrivbart\n"
 "\t\\r     \t" << (int) '\r'   << "\t    vagnretur\toskrivbart\n"
```

```
 "\t\\t    \t" << (int) '\t'   << "\t    tabulator\toskrivbart\n"
 "\t\\b    \t" << (int) '\b'   << "\t     baksteg\toskrivbart\n"
 "\t\\\"   \t" << (int) '\"'   << "\t\t\"    \t    \"\n"
 "\t\\\'   \t" << (int) '\''   << "\t\t\'    \t    \'\n"
 "\t\\\\   \t" << (int) '\\'   << "\t\t\\    \t    \\\n"
 "\t\\0    \t" << (int) '\0'   << "\t   nolltecknet\toskrivbart\n"
 "\t\\007  \t" << (int) '\007' << "\t\007    datorljud\toskrivbart\n"
 "\t\\x61  \t" << (int) '\x61' << "\t\t\x61\t    a \n"
 "\t\\x84  \t" << (int) ('\x84'+256) << "\t\t\x84\t    ä \n"
 "\t\\x86  \t" << (int) ('\x86'+256) << "\t\t\x86\t    å \n"
 "\t\\x94  \t" << (int) ('\x94'+256) << "\t\t\x94\t    ö \n"
 "\t\\x8E  \t" << (int) ('\x8E'+256) << "\t\t\x8E\t    Ä \n"
 "\t\\x8F  \t" << (int) ('\x8F'+256) << "\t\t\x8F\t    Å \n"
 "\t\\x99  \t" << (int) ('\x99'+256) << "\t\t\x99\t    Ö \n\n";
}
```

****

Övning **5.8** är del av en projektuppgift och har därför inget facit, se fotnoten på sidan 231.

****

Övningar till kap 6 Kontrollstrukturer, sid 123

Ovn_6_1

Skriv ett program som läser in två tal och skriver ut OK om de matats in i rätt ordning, dvs om det första är mindre än det andra. Vad händer om de är lika stora?

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

int main()
{
   int no1, no2;
   cout << "\nMata in två heltal skilda med mellanslag:\t";
   cin >> no1 >> no2;
   if (no1 < no2)
      cout << "\n\t\t\tOK\n\n\tTalen matades in i rätt ordning.\n\n";
   cout << "\n\n";
}
```

Om talen är lika stora eller om det första talet är större än det andra, avslutas programmet utan att skriva ut något, eftersom alternativ (else) saknas.

****

Ovn_6_2

Skriv ett program som läser in tre tal, hittar och skriver ut det största av dem. Vilken ändring i koden leder till det minsta talet?

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

int main()
{
   int no1, no2, no3, max;
```

```
   cout << "\n\tMata in tre tal skilda med mellanslag:\t";
   cin >> no1 >> no2 >> no3;

   max = no1
   if (no2 > max)
      max = no2;
   if (no3 > max)
      max = no3;

   cout << "\n\tDet största talet är " <<  max << "\n\n";
}
```

Genom att i hela den nästlade if-else-satsen byta ut alla > mot < får man det minsta talet.

****

Ovn_6_3

Följande pseudokod beskriver hur man tar på sig sjal, mössa och handskar beroende på hur kallt det är ute:

Start *Vinterklädsel*
Läs av temperaturen
OM *temperatur < 0*
    ta sjal, mössa och handskar
ANNARS OM *temperatur < 5*
    ta sjal och mössa
ANNARS OM *temperatur < 10*
    ta sjal
ANNARS
    slipper du vinterklädsel
Slut *Vinterklädsel*

Översätt pseudokoden Vinterklädsel till ett C++ pro¬gram med hjälp av en if-else-stege. Läs in ett värde för temperatur och låt programmet avgöra val av klädsel genom att skriva ut ”Ta …”.

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

int main()
{
  int temperatur;
  cout << "\n\tMata in temperatur:\t";
  cin >> temperatur;
  if (temperatur < 0)
     cout << "\n\tTa sjal, mössa och handskar!\n\n";
  else if (temperatur < 5)
     cout << "\n\tTa sjal och mössa!\n\n";
  else if (temperatur < 10)
     cout << "\n\tTa sjal!\n\n";
  else
     cout << "\n\tDu slipper sjal, mössa och handskar.\n\n";
}
```

****

Ovn_6_4

Skriv ett program som läser in två tecken och skriver ut OK om de matats in i rätt ordning, dvs det första förekommer före det andra i ASCII-tabellen. Annars ska programmet skriva ut ett meddelande om att inmatning skedde i fel ordning.

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

int main()
{
   char letter1, letter2;
   cout << "\nMata in två tecken skilda med mellanslag:\t";
   cin >> letter1 >> letter2;

   if (letter1 < letter2)
      cout << "\n\t\t\tOK\n\n\tDu matade in i rätt ordning.\n\n";
   else
      cout << "\n\tDu matade in i fel ordning.\n\n";
}
```

*****

Ovn_6_5

Vidareutveckla övn 6.4 så att användaren får flera chanser att mata in två tecken i rätt ordning så länge han/hon matar in dem i fel ordning. Du kan göra det genom att bygga in inmatningen, bearbetningen och utmatningen i en do-loop.

```
#include <iostream>
using namespace std;
int main()
{
   char letter1, letter2;

   do
   {
      cout << "\nMata in två tecken skilda med mellanslag:\t";
      cin >> letter1 >> letter2;

      if (letter1 < letter2)
         cout << "\n\t\t\tOK\n\n\tDu matade in i rätt ordning.\n\n";
      else
         cout << "\n\tDu matade in i fel ordning.\n";
   } while (letter1 >= letter2);
   cout << '\n';

}
```

*****

Ovn_6_6

a) Använd en loop med while-satsen för att skriva ut de första 10 positiva heltalen.
b) Vilken ändring ändring i koden till a) måste göras för att få fram de första 20 positiva heltalen?

a)

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

```
int main()
{
   int number = 0;
   cout << '\n';
   while (number < 10)
   {
      number++;
      cout << '\t' << number;
   }
   cout << '\n';
}
```

b) *Endast raden* **while (number < 10)** *måste bytas ut mot:*

```
                while (number < 20)
```

****

Ovn_6_7

a) Skriv ett program som skriver ut de första 10 jämna talen.
b) Modifiera a) så att endast de första 10 udda talen skrivs ut

a)

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

int main()
{
   int number = 0;
   cout << '\n';
   while (number < 20)
   {
      number++;
      if (number % 2 == 0)
         cout << '\t' << number;
   }
   cout << '\n';
}
```

b) *Endast raden* **if (number % 2 == 0)** *måste bytas ut mot:*

```
                if (number % 2 == 1)
```

****

Ovn_6_8

a) Skriv ett program som beräknar summan av de första 10 positiva heltalen.
b) Generalisera a) så att programmet beräknar summan av de första n positiva heltalen där n kan matas in. Testa för n = 100 och 1 000.
c) Skriv ett program som beräknar summan s av de första n positiva heltalen med hjälp av formeln summa = n(n+1)/2. Testa om du får samma svar i b) och c) för n = 1 000, 5 000 och 1 000 000.

a)

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

int main()
```

```
{
   int sum = 1, term = 1;
   while (term < 10)
   {
      term++;
      sum += term;
   }
   cout << "\n\tSumman 1 + 2 + ... + 10 = " << sum << '\n';
}
```

---

b)

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

int main()
{
   int n;
   cout << "\n\tHur långt vill du summera de positiva heltalen?\t";
   cin >> n;
   int sum = 1, term = 1;
   while (term < n)
   {
      term++;
      sum += term;
   }
   cout << "\n\tSumman 1 + 2 + ... + 10 = " << sum << '\n';
}
```

---

c)

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

int main()
{
   int n;
   cout << "\n\tHur långt vill du summera de positiva heltalen?\t";
   cin >> n;
   int sum = n * (n + 1) / 2;
   cout << "\n\tSumman 1 + 2 + ... + 10 = " << sum << '\n';
}
```

****

Ovn_6_9

Skriv ett program som läser in ett heltal och använder det som stegvariabel för att skriva ut tal från 1 till 5000. Om steget är t.ex. 5 skrivs var femte tal ut.

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

int main()
{
   int step;
   cout << "\nMata in ett heltal för steget:\t";
   cin >> step;

   for (int i = 0; i <= 5000; i += step)
       cout << i << '\t';
```

```
   cout << "\n\n";
}

*********************************************************************

Ovn_6_10
Låt en do-loop producera ljudsignal tills man trycker ned någon tangent. Så länge ljudet hörs, ska ett meddelande instruera användaren för att stoppa ljudet. För att avsluta do-loopen använd funktionen kbhit() som är false från början och blir true när man trycker på en tangent. Skriv den logiska negationen ! framför den för att avsluta do-loopen. kbhit() är definierad i biblioteket conio.h.
#include <iostream>
#include <conio.h>                                  // Krävs för kbhit()
using namespace std;

int main()
{
   cout << "\n\tHör du ljudet?\n"
           "\n\tFör att stoppa det, tryck på någon tangent.\n\n";
   do
      cout << '\007';
   while (!kbhit());
}

*********************************************************************
```

Övning **6.11** är en projektuppgift och har därför inget facit, se fotnoten på sidan 231.

```
*********************************************************************
```

Övningar till kap 7 Funktioner, sid 155

```
Ovn_7_1_a
Modularisera lösningen (sid 235) till övn 4.4 som läser in två heltal, gör beräkningar med dem och skriver ut resultaten. Separera beräkningarna från kodens andra delar (inmatning och utmatning).
a) Flytta först multiplikationen till en funktion med returvärde med huvudet int mult(int a, int b) i samma fil som main(). Anropa funktionen mult() från main(). Bibehåll alla andra beräkningar. Se upp med att placera den nya funktionen inte i, utan före main().

#include <iostream>
using namespace std;
---------------------------------------------------------------------
Funktionen mult() som placeras i samma fil före main() och anropas från main(). Tar in två heltal via sina parametrar a och b och returnerar ett heltal som är a * b.

int mult(int a, int b)      // Funktionen mult() med parametrarna a, b
{
   return a * b;            // Returvärdet: a * b
}
---------------------------------------------------------------------
int main()
{
   int no1, no2;
   cout << "\n\tMata in två heltal (skilda med mellanslag):\t";
```

```
   cin >> no1 >> no2;
   cout << "\n\t"
        << no1 << " plus "   << no2 << " är " << (no1+no2) << "\n\t"
        << no1 << " minus "  << no2 << " är " << (no1-no2) << "\n\t"
        << no1 << " gånger " << no2 << " är "
                                    << mult(no1, no2) // Anropet
        << "\n\t" << no1 << " heltalsdividerad med "   <<
                                 no2 << " är " << (no1/no2) << "\n\t"
        << no1 << " modulo " << no2 << " är " << (no1%no2) << "\n\n";
}
```

***\

```
#include <iostream>
#include "Arithmetics.h"                          // Headerfil
using namespace std;

int main()
{
  int no1, no2;
  cout << "\n\tMata in två heltal (skilda med mellanslag):\t";
  cin >> no1 >> no2;

  cout << "\n\t"                                   // Anropen:
     << no1 << " plus "   << no2 << " är " << add(no1, no2) << "\n\t"
     << no1 << " minus "  << no2 << " är " << sub(no1, no2) << "\n\t"
     << no1 << " gånger " << no2 << " är " << mult(no1, no2)<< "\n\t"
     << no1 << " heltalsdividerad med " <<
                          no2 << " är " << intDiv(no1, no2) << "\n\t"
     << no1 << " modulo "<< no2 << " är " << mod(no1, no2) << "\n\n";
}
```

Arithmetics.h *(Ovn_7_1_c forts.)*
Separat fil med ett antal funktioner som definierar räkneoperationer. Inkluderas som headerfil i programmet Ovn_7_1_c där räkneoperationerna anropas från main().

```
int mult(int a, int b)           // Funktionen mult() med parametrarna
a, b
{
   return a * b;                 // Returvärdet: a * b
}
```

```
int add(int a, int b)            // Funktionen add()
{
   return a + b;
}
```

```
int sub(int a, int b)            // Funktionen sub()
{
   return a - b;
}
```

```
int intDiv(int a, int b)         // Funktionen intDiv()
{
   return a / b;
}
```

```
int mod(int a, int b)            // Funktionen mod()
{
   return a % b;
```

```
}
```

****

Ovn_7_2_a

a) Skriv först ett program med endast main()-funktionen som läser in radien r till en cirkel & beräknar och skriver ut cirkelns area πr^2 *och dess omkrets* $2\pi r$*, där* $\pi = 3.14159$.

```
#include <iostream>
using namespace std;
int main()
{
   double radie, area, omkrets, PI = 3.14159;

   cout << "\n\tAnge radien till en cirkel:\t";
   cin >> radie;                                          // Input

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

   cout << "\n\tEn cirkel med radien\t\t" << radie     // Output
        << "\n\thar arean           \t\t" << area
        << "\n\toch omkretsen       \t\t" << omkrets << "\n\n";
}
```

Ovn_7_2_b

b) Flytta sedan bearbetningsdelen dvs beräkningen av area och omkrets ur main() till separata funktioner area() och omkrets(), men stanna i samma fil. I main() ska finnas kvar variabeln för radien, inmatning, utmatning och anropet av area() och omkrets(). Förse de nya funktionerna med en parameter som överför radiens värde från main() till dem. Välj olika namn för den aktuella än för den formella parametern. Definiera π *lokalt i funktionerna. Dessutom ska area() och omkrets() returnera ett double-värde. För att testa mata in radien 1. Då ska arean bli* π *pga* $\pi r^2 = \pi$ *och omkretsen bli* 2π *pga* $2\pi r = 2\pi$.

```
#include <iostream>
using namespace std;
-------------------------------------------------------------
double area(double r)                  // Funktionen area() med formell
{                                      // parameter r som tar emot
   return 3.14159 * r * r;             / aktuell parameter radie
}
-------------------------------------------------------------
double omkrets(double r)               // Funktionen omkrets()
{
   return 2 * 3.14159 * r;
}
-------------------------------------------------------------

int main()
{
   double radie;

   cout << "\n\tAnge radien till en cirkel:\t";
```

```
   cin >> radie;                                                  // Input

   cout << "\n\tEn cirkel med radien\t\t" << radie          // Output
        << "\n\thar arean           \t\t" << area(radie)    // Anrop
     << "\n\toch omkretsen       \t\t" << omkrets(radie) << "\n\n";
}
```

*****

Ovn_7_3
*Skriv ett program som läser in sida till en kub samt beräknar och skriver ut kubens volym sida*sida*sida och dess yta 6*sida*sida. Dessa beräkningar ska göras i två funktioner, en för volymen, en för ytan, båda i en separat headerfil. Avgör själv om funktionerna volym() och yta() ska returnera ett värde eller vara av void-typ. Anropa dem från main().*

```
#include <iostream>
#include "Kub.h"                          // Headerfil
using namespace std;

int main()
{
   double sida;

   cout << "\n\tAnge sidan till en kub:\t\t";
   cin >> sida;

   cout << "\n\tEn kub med sidan\t\t" << sida
        << "\n\thar volymen      \t\t" << volym(sida)       // Anrop
        << "\n\toch ytan         \t\t" << yta(sida) << "\n\n";
}
```

Kub.h *(Ovn_7_3, forts.)*
Separat fil med ett två funktioner som definierar kubens volym och yta. Inkluderas som headerfil i programmet Ovn_7_3 där funktionerna anropas från main().

```
double volym(double s)
{
   return s * s * s;
}

double yta(double s)
{
   return 6 * s * s;
}
```

*****

Ovn_7_4
Modularisera programmet Hour2Sec (sid 64) genom att skriva dess bearbetningsdel som en ny funktion. Bibehåll in- och utmatnigsdelen i main() och anropa den nya funktionen från main(). Avgör själv om den nya funktionen ska returnera ett värde. Ge den ett beskrivande namn.

```
#include <iostream>
```

```
#include "Hour2sec.h"
using namespace std;

int main()
{
   int tim, min, sek;

   /* I n m a t n i n g */
   cout << "\nGe timmar, minuter, sekunder "
        << "skilda med mellanslag:  ";
   cin  >> tim >> min >> sek;

   /* U t m a t n i n g */
   cout << '\n' << tim << " timmar, " << min << " minuter och "
        << sek << " sekunder är " << hour2sec(tim, min, sek)
        << " sekunder totalt.\n\n";
}
```

Hour2sec.h *(Ovn_7_4, forts.)*
Separat fil med en funktion som omvandlar tiden från timmar, minuter, sekunder till totalsekunder. Inkluderas som headerfil i programmet Ovn_7_4 där funktionen anropas från main().

```
int hour2sec(int t, int m, int s)
{
   /* B e a r b e t n i n g */
   return 3600*t + 60*m + s;
}
```

****

Ovn_7_5
*Modularisera programmet **Sec2Hour** (sid 156) genom att flytta bearbetnings- och utmatnigsdelen till en **void**-funktion. Dvs skriv ett program som läser in tiden i ett antal sekunder, anropar **void**-funktionen som omvandlar tiden till antal timmar, minuter samt resterande sekunder och skriver ut resultaten. Använd för omvandlingen den algoritm som är implementerad i programmet **Sec2Hour**. Varför kan man inte här använda en funktion med returvärde?*

```
#include <iostream>
#include "Sec2hourOutput.h"
using namespace std;

int main()
{
   int totalsek;

   /* I n m a t n i n g */
   cout << "\nAnge tid i sekunder: ";
   cin  >> totalsek;

   sec2hourOutput(totalsek);
}
```

Sec2hourOutput.h *(Ovn_7_5, forts.)*
Separat fil med en funktion som omvandlar tiden från sekunder till ett antal timmar, minuter samt resterande sekunder och skriver ut resultaten. Inkluderas som headerfil i programmet Ovn_7_5 där funktionen anropas från main().

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

void sec2hourOutput(int total)                    // void-funktion
{
   int tim, min, sek;
   /* B e a r b e t n i n g */
   tim =   total / 3600;
   min =  (total % 3600) / 60;
   sek = ((total % 3600) % 60) % 60;

   /* U t m a t n i n g */
   cout << '\n' << total << " totalsekunder = " << tim
        << " timmar, " << min << " minuter, " << sek
        << " sekunder.\n\n";
}
```

Man kan här inte använda en funktion med returvärde därför att en sådan funktion endast kan returnera ETT värde. Här har vi att göra med tre värden, timmar, minuter och sekunder, som ska returneras. Det var därför vi var tvungna att skriva ut resultatet från funktionen sec2hour(), vilket ur modulariseringssynpunkt inte är optimalt. En bättre lösning finns i följande övning.

***\

```
}
```

Table.h *(Ovn_7_7, forts.)*
Separat fil med en funktion som skriver ut en tabell. Inkluderas som headerfil i programmet Ovn_7_7 där funktionen anropas från main().

```
#include <iostream>
#include <iomanip>                    // Krävs för setprecision()
#include "Netto.h"
using namespace std;

void table(float first, float last, float step, float tax, float
brutt)
{
   cout << fixed << setprecision(2); // Efter detta kommer alla cout-
                                     // satser skriva ut tal med två
                                     // fasta decimaler
   cout << "\nBrutto\t\tNetto\t\tIngående moms\n"
        << "-----------------------------------------\n";
   for (brutt = first; brutt <= last; brutt += step)
       cout << brutt << "\t\t" << netto(brutt, tax)      // Två anrop
            << "\t\t" <<  brutt - netto(brutt, tax)     // av netto()
            << "  kr\n";
   cout << '\n';
}
```

Netto.h *(Ovn_7_7, forts.)*
Beräknar nettobeloppet utgående från ett bruttobelopp. Definierar funktionen netto() som tar in ett bruttobelopp b samt momssatsen m och returnerar nettobeloppet. Anropas i funktionen table().

```
float netto(float b, float m)             // Funktionsdefinition
{
   return 100*b/(100+m);                  // Formeln för beräkning
}                                         // av nettobelopp
```

****

Ovn_7_7
Varför ger följande program kompileringsfel? Åtgärda felet genom att flytta på kod, utan att ta bort någon klammer och utan att ha tomma klamrar:

```
#include <iostream>
using namespace std;
int main()
{
   {
      int t = 30;
   }
   cout << "\n\tt = " << t << "\n\n";
}
```

Lösningen:

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

```
int main()
{
   int t;
   {
      t = 30;
   }
   cout << "\n\tt = " << t << "\n\n";
}
```

Kompileringsfelet i programmets första variant berodde på att variabeln t var definierad i ett inre block och att programmet refererade till den utanför det inre blocket där t inte längre var giltig.

****

Övningar till kap 8 Arrays, sid 177

Ovn_8_1
Skriv ett program som läser in 20 heltal, lagrar dem i en array och skriver ut dem i omvänd ordning.

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

int main()
{
   int no[20];

   cout << "\n\tSkriv in 20 heltal:\n\n";
   for (int i = 0; i <= 19; i++)
   {
       cout << "\tTal nr " << (i+1) << ":\t";
       cin >> no[i];
   }

   cout << "\nDina tal i omvänd ordning:\n\n";
   for (int i = 19; i >= 0; i--)
       cout << no[i] << "\t";

   cout << "\n\n";
}
```

****

Ovn_8_2
Skriv ett program som läser in en text, lagrar den i en array av char och skriver ut den baklänges. Utveckla en teknik för att ta reda på den aktuellt inmatade char-arrayens exakta längd.

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

int main()
{
   char text[80];
   cout << "\n\tSkriv in text:\t\t";
   cin.getline(text, 80);              // Läser in texten: getline()
                                       // sätter nolltecknet
```

```
   int length = 0;                      // Textens längd initieras
   while(text[length] != '\0')          // Räknar upp längden tills
      length++;                         // nolltecknet hittas

   cout << "\n\tTexten baklänges:\t";
   for (length; length > 0; length--)  // Skriver ut texten baklänges
       cout << text[length-1];
   cout << "\n\n";
}
```

*****

Ovn_8_3

Skriv ett program som läser in text i gemener, lagrar den i en array av char och skriver ut den framhävd i versaler och med mellanslag mellan varje tecken. Gör som i övn 9.2 angående arrayens längd.

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

int main()
{
   char text[80];
   cout << "\n\tSkriv in text:\t\t";
   cin.getline(text, 80);               // Läser in texten: getline()
                                        // sätter nolltecknet
   int length = 0;                      // Textens längd initieras
   while(text[length] != '\0')          // Räknar upp längden tills
      length++;                         // nolltecknet hittas

   cout << "\n\tTexten framhävd:\t";
   for(int i = 0; i < length; i++)      // Skriver ut texten i versa-
      cout << (char)(text[i] - 32) << ' '; // ler och med mellanslag
   cout << "\n\n";                          // mellan varje tecken
}
```

*****

Ovn_8_4

Skriv ett program som frågar efter användarens för- och efternamn, hälsar sedan användaren i en utskrift med fullständiga namnet, förnamnets längd samt efternamnets första och sista bokstav. Lös uppgiften generellt utan att använda information om något speciellt för- och efternamn.

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

int main()
{
   char efternamn0;
   char namn[80];
   cout << "\n\t Skriv in ditt för- och efternamn: ";
   cin.getline(namn, 80);               // Läser in namn: getline()
                                        // sätter nolltecknet
   int length = 0;                      // Hela namnets längd
   while(namn[length] != '\0')          // Räknar upp längden tills
      length++;                         // nolltecknet hittas
```

```
   int i = 0;                              // i = endast förnamnets längd
   while (namn[i] != ' ')                  // Går igenom endast förnamnet
   {                                       // Räknar upp i tills för- och
      i++;                                 // efternamnets avskiljare dvs
      if (namn[i] == ' ')                  // mellanslaget hittas
         efternamn0 = namn[i+1];           // Hittar efternamnets första
   }                                       // bokstav

   cout << "\n\tHej, "                                  << namn
        << "\n\tDitt förnamns längd är "                << i
        << "\n\tDitt efternamns första bokstav är " << efternamn0
        << "\n\tDitt efternamns sista bokstav är "  << namn[length-1]
        << "\n\n";
}
```

****

Ovn_8_5

Skriv ett program där main() läser in en persons fullständiga namn och hälsar tillbaka med namnets initialer. Dessa ska bestämmas och skrivas ut i en annan funktion – med huvudet void initialer(char[] namn) – som anropas i main(). Allt skrivs i en fil.

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

void initialer(char namn[])             // Funktionen
{
   int i = 0;
   cout << namn[i];                     // Första initialen
   while (namn[i] != ' ')               // Går igenom endast förnamnet
   {
      i++;
      if (namn[i] == ' ')               // Hittar för- och efternamnets
                                        //                   avskiljare
         cout << namn[i+1];             // Andra initialen
   }
}

int main()                              // main()
{
   char dittNamn[80];
   cout << "\n\tSkriv in ditt för- och efternamn: ";
   cin.getline(dittNamn, 80);           // Läser in dittNamn: getline()
                                        // sätter nolltecknet
   int length = 0;                      // Hela namnets längd
   while(dittNamn[length] != '\0')      // Räknar upp längden tills
      length++;                         // nolltecknet hittas

  cout << "\n\tHej, " << dittNamn << "\n\n\tDina initialer är\t\t  ";

   initialer(dittNamn);                 // Anropet

   cout << "\n\n";
}
```

****

Ovn_8_6

Skriv ett program som slumpar fram 1000 heltal mellan 60 och 140 (tänkbara hastigheter på en motorväg), lagrar dem i en array hastighet, beräknar och skriver ut deras medelvärde med förklarande text och lämplig rubrik. Använd funktionen RandArray() som en extern modul.

```
#include <iostream>
#include "SlumpArray.h"                    // Innehåller myRand()
using namespace std;

int main()
{
   const int n = 1000;                     // Arrayens storlek
   int hastighet[n];
   RandArray(hastighet, n, 60, 140);       // Anrop av RandArray()
   int sum = 0;
   for (int i = 0; i < n-1; i++)
       sum += hastighet[i];
   cout << "Medelvärdet av " << n << " möjliga hastigheter "
        << "mellan 60 och 140 är:    "  <<  sum/n << "\n\n";
}
```

SlumpArray.h *(Ovn_8_6, forts.)*
Funktion som slumpar fram en array av heltal mellan a och b, lagrar dem i arrayen no och skriver ut dem.
Inkluderas i Ovn_8_6: Där anropar main() funktionen RandArray() som i sin tur anropar funktionen myRand() som ger ETT slumptal i [a, b]

```
#include <iostream>
#include <ctime>                           // Krävs för time()
#include "MyRand.h"                        // Innehåller myRand()
using namespace std;

void RandArray(int no[], int n, int a, int b)
{
   srand(time(0));                      // Skapar variation i slumpen
   cout << '\n' << n << " heltal mellan " << a << " och " << b
        << " slumpas fram:\n\n";
   for (int i = 0; i < n; i++)
   {
       no[i] = myRand(a, b);            // Anrop av myRand()
       cout << no[i] << ' ' ;
   }
   cout << "\n\n";
}
```

MyRand.h *(Ovn_8_6, forts.)*
Funktion som returnerar ETT slumptal i heltalsintervallet [a, b]
Inkluderas och anropas i RandArray()

```
int myRand(int a, int b)
{
   if (a < b)
      return a + rand() % (b-a+1);
   else
```

```
      return b + rand() % (a-b+1);
}

*********************************************************************
```

Övningar till kap 9 Pekare, sid 204

Ovn_9_1

Definiera en vanlig variabel a och en pekarvariabel p, låt p peka på a, läs in ett värde till a, men skriv ut värdet via pekaren.

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

int main()
{
  int  a;                                  // Vanlig variabel
  int* p = &a;                             // Pekarvariabel som pekar på
                                           // vanlig variabel
  cout << "\n\tAnge ett heltal som lagras i en vanlig variabel a:\t";
  cin >> a;                                // Inläsning till vanlig
                                           //                variabel
  cout << "\n\tVärdet av vanlig variabel a återges med en \n\tpekar"
       "variabel p som pekar på den (p = &a):\t\t" << *p << "\n\n";
}

*********************************************************************
```

Ovn_9_2

Definiera tre vanliga variabler a, b, sum och tre pekarvariabler som pekar på resp. vanlig 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.

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

int main()
{
   int a, b, sum;                          // Vanliga variabler
   int *pa = &a, *pb = &b, *psum = &sum; // Pekarvariabler som pekar
                                           // på de vanliga variablerna
   cout << "\n\tAnge två heltal skilda med mellanslag:\t";
   cin >> *pa >> *pb;                      // Inläsning via pekare

   sum = *pa + *pb;                        // Addition via pekare
   cout << "\n\tSumman av " << *pa << " och "
        << *pb << " är\t" << *psum << "\n\n";
}

*********************************************************************
```

Ovn_9_3

Lös Ovn_9_2 ovan utan en enda vanlig variabel, endast med pekare.

```
#include <iostream>
using namespace std;
int main()
{
```

```
   int *pa, *pb, *psum;                          // Tre pekarvariabler
   pa   = new int;                               // new allokerar minnen:
   pb   = new int;                               // Pekarna pekar på dem
   psum = new int;
   cout << "\n\tAnge två heltal skilda med mellanslag:\t";
   cin >> *pa >> *pb;                             // Inläsning via pekare

   *psum = *pa + *pb;                             // Addition via pekare
   cout << "\n\tSumman av " << *pa << " och " // Utskrift via pekare
        << *pb << " är\t" << *psum << "\n\n";
}

*********************************************************************
```

Ovn_9_4

*Skriv en pekarversion av programmet ArrayDef (sid 161). Skapa arrayen med int *no = new int[4]; och ersätt överallt index med pekarposition.*

```
#include <iostream>
using namespace std;
int main()
{
   int *no = new int[4];                 // Definition av en array
   *no     = 64;                         // Elementvis initiering
   *(no+1) = 86;                         // av arrayen
   *(no+2) = 34;
   *(no+3) = -6;
   for (int i=0; i <= 3; i++)            // i = pekarposition = räknare
       cout << "Arrayens " << i + 1 << ":a element no[" << i
            << "] har värdet "<< *(no+i) << " och index " << i
            << "\n\n";
   cout << "Index -1 och 4 ligger utanför och är "
        << "ej definierade. Resultat:\n\n";
   cout << "no[-1] = " << *(no-1) << " med index -1 och\n"
        << "no[ 4] = " << *(no+4) << " med index  4 "
        << "är odefinierade skräpvärden!" <<"\n\n";
}

*********************************************************************
```

Ovn_9_5

Skriv en pekarversion av programmet ArrayChar. Lägg till efter sizeof som mäter strängens minnesutrymme, kod som med strlen() även mäter strängens längd. Varför returnerar sizeof och strlen() olika värden?

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

int main()
{
   const char *text = new char;          // Pekare till en const char
   text = "C++";                         // Initiering av pekare till
                                         // en sträng (= konst. pekare)
   for (int i=0; i<=3; i++)
       cout << i+1 <<":a tecknet i \"C++\" är "<< *(text+i) <<'\n';
   cout << "\nSträngen \"C++\" tar\t   " << sizeof(text)
        << " bytes: Storleken på pekaren text med sizeof\n"
        << "Strängen \"C++\" har längden " << strlen(text)
        << " bytes: Storleken på strängen C++ med strlen()\n"
        << "4:e tecknets ASCII-kod är " << (int) *(text+3) << '\n'
```

```
        << "Nolltecknets ASCII-kod är " << (int) '\0' << '\n'
        << "Alltså är 4:e tecknet i \"C++\" nolltecknet\n\n";

   char copy [4];                    // Ny array av char

   strcpy_s(copy, text);             // text="C++" kopieras till copy
   *(copy+1) = '\0';                 // Avkortning av "C++" till "C"
   cout << copy << " är en del av " << text << "\n";
}
```

sizeof och strlen() returnerar olika värden därför att sizeof mäter antal bytes strängen tar i minnesutrymme – 1 byte för varje char – inklusive nolltecknet, medan strlen() mäter strängens antal synliga tecken utan nolltecknet. Därför returnerar sizeof("C++") 4 medan strlen("C++") ger 3.

****

Ovn_9_6

Skriv ett program som läser in en sträng med pekare och skriver ut den i omvänd ordning.

```
#include <iostream>
using namespace std;
int main()
{
   char *text = new char[80];        // Sträng med pekare
   cout << "\n\tSkriv in text:\t\t";
   cin.getline(text, 80);            // Läser in strängen med get-
                                     // line() som sätter nolltecknet
   int length = strlen(text);        // Strängens längd

   cout << "\n\tSträngen baklänges:\t";
   for (length; length > 0; length--)// Skriver ut strängen baklänges
       cout << *(text+length-1);     // utan nolltecknet
   cout << "\n\n";
}
```

****

Ovn_9_7

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.

```
#include <iostream>
using namespace std;
int main()
{
   char *text = new char[80];            // Sträng med pekare
   cout << "\n\tSkriv in text:\t\t";
   cin.getline(text, 80);                // Läser in strängen
   int length = strlen(text);            // Strängens längd
   cout << "\n\tTextens första och sista tecknet "
           "byts ut mot varandra:\n"
           "\n\tTexten före  platsbytet:\t" << text;

   char temp = *text;                    // Algoritm för platsbyte på
   *text = *( text+length-1);            // det 1:a och sista tecknet
   *( text+length-1) = temp;
```

```
   cout << "\n\tTexten efter platsbytet:\t" << text << "\n\n";
}
```

*****

Ovn_9_8
Modularisera programmet från Ovn_9_3 ovan genom att låta summan beräknas i en funktion. För att illustrera referensanrop med pekare nollställ summans värde i main(), skicka det som pekare (parameter) till funktionen där värdet ändras och skriv ut det ändrade värdet från main().

```
#include <iostream>
#include "AddPekare.h"
using namespace std;
int main()
{
   int *pa, *pb, *psum;
   pa   = new int;
   pb   = new int;
   psum = new int(0);                    // Summans värde nollställs
   cout << "\n\tAnge två heltal skilda med mellanslag:\t";
   cin >> *pa >> *pb;

   add(pa, pb, psum);                    // Referensanrop: Pekarna
                                         // skickas till funktionen
   cout << "\n\tSumman av " << *pa << " och "
        << *pb << " är\t" << *psum << "\n\n";// Summans ändrade värde
                                            // återfinns i main()
}
```

AddPekare.h *(Ovn_9_8 forts.)*
Extern funktion som beräknar summan med parametrar som är pekare. Den tredje parametern pekar på värdet 0 när den kommer in och pekas om till summans värde i funktionen. Pga referensanrop återförs summans värde till main() där funktionen anropas, eftersom båda pekarna psum och ps pekar på samma minnescell.

```
void add(int *p1, int *p2, int *ps)
{
   *ps = *p1 + *p2;                      // Summans värde ändras
}
```

*****

Övningar till kap 10 Klasser, sid 228

Rektangel.h (Ovn_10_1)
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.

```
class Rektangel
{
   int bredd, hojd;                      // Privata datamedlemmar
public:
   Rektangel(int b, int h)               // Klassens konstruktor
```

```
   {
      bredd = b;
      hojd  = h;
   }

   int area()                              // Publika Metoder
   {
      return bredd * hojd;
   }

   int omkrets()
   {
      return 2 * (bredd + hojd);
   }
};
```

RektangelTest *(Ovn_10_1, forts.)*
Testa klassen Rektangel i en separat fil genom att i main() skapa ett Rektangel-objekt vars datamedlemmar initieras till konstanta värden. Skriv ut objektets area och omkrets.

```
#include <iostream>
using namespace std;
#include "Rektangel.h"                 // Innehåller klassen Rektangel

int main()

   Rektangel rekt(8, 4);               // Objekt rekt av typ Rektangel
                                       // skapas och klassens kon-
                                       // struktor anropas
   cout << "\n\tArea = " << rekt.area() << '\n' <<
                        "\n\tOmkrets = " << rekt.omkrets() << "\n\n";
}
```

****

Rektangel_ny.h (Ovn_10_2)
Modifiera klassen Rektangel från övn 10.1 genom att lägga till get- och set-metoder i klassen.

```
class Rektangel_ny
{
   int bredd, hojd;                        // Privata datamedlemmar

public:
   Rektangel_ny(int b, int h)              // Konstruktorn
   {
      bredd = b;
      hojd  = h;
   }

   int getLängd()                          // get-metod för bredd
   {
      return bredd;
   }

   void setLängd(int nyLängd)              // set-metod för bredd
   {
      bredd = nyLängd;
   }
```

```
   int getHojd()                          // get-metod för hojd
   {
      return hojd;
   }

   void setHojd(int nyHojd)               // set-metod för hojd
   {
      hojd = nyHojd;
   }

   int area()
   {
      return bredd * hojd;
   }

   int omkrets()
   {
      return 2 * (bredd + hojd);
   }
};
```

Rektangel_nyTest *(Ovn_10_2, forts.)*
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?

```
#include <iostream>
using namespace std;
#include "Rektangel_ny.h"          // Innehåller klassen Rektangel_ny

int main()
{
   int no1, no2;
   cout << "\n\tMata in bredd och höjd till en rektangel:\t";
   cin >> no1 >> no2;

   Rektangel_ny rekt(no1, no2);   // Objekt rekt skapas och initieras

   cout << "\nFöre fördubblingen:\n"
           "\n\tArea    = " << rekt.area()    << '\n'
           "\n\tOmkrets = " << rekt.omkrets() << '\n';

   rekt.setLängd(2 * rekt.getLängd());
   rekt.setHojd (2 * rekt.getHojd() );

   cout << "\nEfter fördubblingen:\n"
           "\n\tArea    = " << rekt.area()    << '\n'
           "\n\tOmkrets = " << rekt.omkrets() << "\n\n";
}
```

Arean växer med faktor 4 när rektangelns sidor fördubblas, medan omkretsen växer med faktor 2, eftersom arean är en kvadratisk funktion av sidorna, medan omkretsen är en linjär funktion av dem.

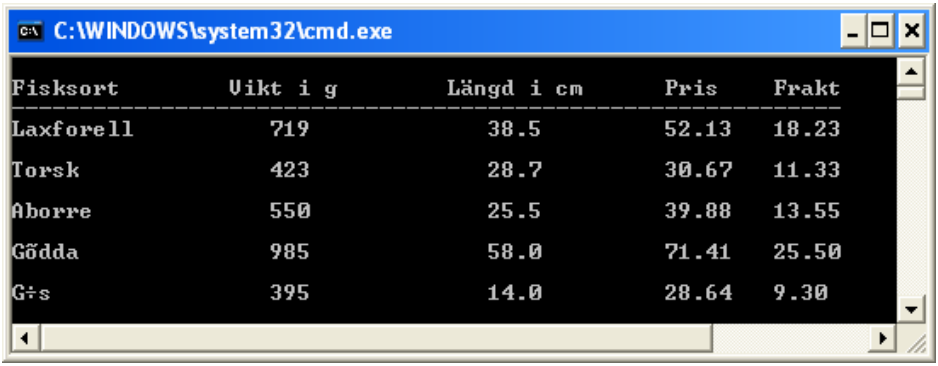
****

Fish.h (Ovn_10_3)
Skriv en klass Fish som beskriver en fisk med datamedlemmarna sort, vikt och längd. Förse klassen med en metod pris() som beräknar pri-

set 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.

```
class Fish
{
public:
   string sort;
   double vikt, langd;

   double pris()
   {
      return vikt * 7.25 / 100;
   }

   double frakt()
   {
      return vikt * 0.02 + langd * 0.1;
   }
};
```

FishTest *(Ovn_10_3, forts.)*
Testa klassen Fish i en separat fil genom att i main() skapa en array av fem Fish-objekt vars datamedlemmar tilldelas värden, t.ex. "laxforell" till sort, 719 (gram) till vikt och 38,5 (cm) till längd. Pris & frakt till varje Fish-objekt ska beräknas genom anrop av metoderna pris() och frakt(). Skriv sedan ut dessa värden tillsammans med pris & frakt till en tabell som t.ex. kan se ut så här:




```
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;
#include "Fish.h"                          // Innehåller klassen Fish
int main()
{
   Fish f[5];                              // Array av 5 Fish-objekt

   f[0].sort = "Laxforell";    f[0].vikt = 719;    f[0].langd = 38.5;
   f[1].sort = "Torsk\t";      f[1].vikt = 423;    f[1].langd = 28.7;
   f[2].sort = "Aborre\t";     f[2].vikt = 550;    f[2].langd = 25.5;
```

```
   f[3].sort = "Gädda\t";      f[3].vikt = 985;    f[3].langd = 58;
   f[4].sort = "Gös\t";        f[4].vikt = 395;    f[4].langd = 14;
   cout << "\nFisksort\tVikt i g\tL\u201Engd i cm\tPris\tFrakt\n"
   "-------------------------------------------------------------\n";
   cout << fixed;

   for (int i = 0; i < 5; i++)
   {
       cout << f[i].sort << "\t   " << setprecision(0) << f[i].vikt
            << "\t\t   " << setprecision(1)   << f[i].langd << "\t\t"
            << setprecision(2) << f[i].pris() << "\t"
            << f[i].frakt()    << "\n\n";
   }
}
*********************************************************************
```

Fish_priv.h (Ovn_10_4)

Deklarera datamedlemmarna i klassen Fish från övn 12.3 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 10.3.

```
class Fish_priv
{
   string sort;
   double vikt, langd;

public:
   Fish_priv(string s, double v, double L)
   {
      sort  = s;
      vikt  = v;
      langd = L;
   }

   double pris()
   {
      return vikt * 7.25 / 100;
   }

   double frakt()
   {
      return vikt * 0.02 + langd * 0.1;
   }

   void output()
   {
      cout << sort << "\t   " << fixed << setprecision(0) << vikt
           << "\t\t   " << setprecision(1)   << langd << "\t\t"
           << setprecision(2) << pris() << "\t"
           << frakt() << "\n\n";
   }
};
```

Fish_privTest *(Ovn_10_4, forts.)*

```
#include <iostream>
#include <iomanip>
#include <string>
```

```
using namespace std;
#include "Fish_priv.h"
int main()
{
   Fish_priv *f[5];        // Array av 5 pekare till Fish_priv-objekt

   f[0] = new Fish_priv("Laxforell", 719, 38.5);// 5 Fish_priv-objekt
   f[1] = new Fish_priv("Torsk\t",   423, 28.7);
   f[2] = new Fish_priv("Aborre\t",  550, 25.5);
   f[3] = new Fish_priv("Gädda\t",   985, 58  );
   f[4] = new Fish_priv("Gös\t",     395, 14  );

   cout << "\nFisksort\tVikt i g\tL\u201Engd i cm\tPris\tFrakt\n"
   "-----------------------------------------------------------------\n";
   for (int i = 0; i < 5; i++)
       f[i]->output();
}
*********************************************************************
```

Cylinder.h (Ovn_10_5)

Modellera en klass Cylinder som subklass till klassen Circle. Denna modellering ser Cylindern som en Circle som dessutom har en höjd. Betrakta därför Cylindern som en "utvidgad" Circle som ärver Circlen och lägger till den en privat datamedlem höjd. Förse även subklassen med en konstruktor och en get-metod. Cylindern ska dessutom ha metoderna volym() & yta(). Vid beräkning av Cylinderns volym() och yta() ska koden kunna återanvända cirkelns metoder genom att anropa dem.

```
#include "Circle.h"                  // Innehåller klassen Circle
class Cylinder : Circle              // Cylinder ärver klassen Circle
{
   double hojd;                      // Den nya privata datamedlemmen

public:
   Cylinder(double r, double h) : Circle(r) // Cylinderns konstruktor
   {                                        // med anrop av Circlens
      hojd = h;                             // konstruktor
   }

   double getHojd()
   {
      return hojd;
   }

   double cylinderVolym()            // Cylinderns metod för volym
   {                                 // återanvänder Circlens metod
      return cirkelArea() * hojd;    // för area genom att anropa den
   }

   double cylinderYta()              // Cylinderns metod för yta
   {                                 // återanvänder Circlens metod
      return cirkelOmkrets() * (getRadie() + hojd);    // för omkrets
   }
};
```

Circle.h *(Ovn_10_5, forts.)*

Förse superklassen Circle med en privat datamedlem radie, en konstruktor, en getmetod och med beräkningsmetoderna area() och omkrets().

```
class Circle
{
   double radie;

public:
   Circle(double r)
   {
      radie = r;
   }

   double getRadie()               // getmetod för att exportera radie
   {                               // bl.a. till klassen Cylinder som
      return radie;                // behöver den för yta()
   }

   double cirkelArea()
   {
      return 3.14159 * radie * radie;
   }

   double cirkelOmkrets()
   {
      return 2 * 3.14159 * radie;
   }
};
```

CylinderTest *(Ovn_10_5, forts.)*
Testa dina klasser i main() genom att läsa in en cylinders radie och höjd samt skriva ut volym() och yta().

```
#include <iostream>
using namespace std;
#include "Cylinder.h"                  // Innehåller klassen Cylinder
int main()
{
   int no1, no2;
   cout << "\n\tMata in radie och höjd till en cylinder:\t";
   cin >> no1 >> no2;

   Cylinder c(no1, no2);               // Ett Cylinder-objekt skapas
                                       // Konstruktorn anropas som
                                       // initierar radie och höjd
   cout << "\nEn cylinder med radien " << no1
        << " och höjden "    << c.getHojd() << " har volymen "
        << c.cylinderVolym() << "\n\t\t\t\t\toch ytan "
        << c.cylinderYta()   << "\n\n";

}
```

****

Övningarna **10.6** och **10.7** är projektuppgifter och har inget facit (fotnoten på sid 231).

****

Programförteckning

Program	Ämne	Sida
Kapitel 3	**Att komma igång med C++**	
MyFirst	Utskrift till skärmen med **cout** och **<<**, kommentarer	33
MyFirstSwed	Utskrift med korrekta svenska specialtecken	37
Cout	Radbyte med **\n**, flera **cout**-satser, **\t**	38
Concat	Konkatenering: Sammanslagning av flera **cout**-satser	44
Figure	Tillämpning av **cout**, **<<**, **\n**, **\t** och konkatenering	45
Kapitel 4	**Grundbegrepp i programmering**	
Datatype	Olika datatyper hos konstanter: tal, tecken och sträng	49
Variable	Variabler: Definition och initiering. Datatyper	55
NoInitial	Definierade, men oinitierade variabler	58
DefInitial	Definition och initiering av variabler i samma sats	57
OverWrite	Överskrivning av variabelvärdet Skillnaden mellan likhet och tilldelning	60
Cin	Inläsning av värden med **cin** och inmatningsoperatorn **>>**	62
Hour2Sec	Inmatning – Bearbetning – Utmatning	64
Increment	Ökningsoperatorn **++** med post- och prefixvarianten	67
CompAssign	Sammansatt tilldelning med **+=**, **-=**, ***=**, **/=**	70
Kapitel 5	**Enkla datatyper**	
Primitives	De enkla datatypernas minnesstorlek	76
Limits	Heltalstypernas gränser	78
Char	Representation av tecken med datatypen **char**	79
Char2int	Kod till inmatat tecken: Explicit typkonvertering	81
Int2char	Tecken till inmatad kod: Kodrepresentation av tecken	82
Ascii	ASCII-tabellen, teckenaritmetik	84
Escape	Escapesekvenser	86
Kapitel 6	**Kontrollstrukturer**	
If	Enkel selektion: **if**-satsen	92
MiniSort	Flera satser i **if**-satsens kropp: Algoritm för platsbyte	95
IfElse	Tvåvägsval: **if-else**-satsen	98
GissaTal_1	Flervägsval: **if-else**-stegen	101
Switch	Flervägsval: **switch**-satsen	103
GissaTal_2	Efter-testad repetition: **do**-satsen	106
While	För-testad repetition: **while**-satsen	108
While->For	Bestämd repetition **for**: Översätning av **while** till **for**	111

AsciiFor	ASCII-tabellen med **for**-satsen	113
Stars	Ritar rektangel fylld med stjärnor med nästlad **for**-sats	116
NestedFor	Simulerar tärningskast med nästlad **for**-sats	118

Kapitel 7 Funktioner

MyFirstFct	Definition och anrop av egna funktioner med returvärde	127
Random	Deklaration av en funktion: Prototypen	138
ExternFct	Program i två filer: **main()** *anropar* funktionen **netto()**	141
Netto	Headerfil som *definierar* funktionen **netto()**	142
Compare	Funktioner utan returvärde: **void**-funktioner	144
GuessVoid	Anrop av **void**-funktionen **compare()**	145
RandNo	Funktion som slumpar fram heltal i intervallet [a, b] Används av flera program i boken (har därför inget nr.)	146
Global	Globala och lokala variabler, blockstruktur	147
Scope	Överskuggning av variabler	152

Kapitel 8 Arrays

ArrayDef	Definition och elementvis initiering av en array Datatypen array av **int**	161
ArrayInit	Initieringslista: Kortform för definition och initiering i en och samma sats, användning av **for**-sats med array	163
ArrayChar	Stränghantering med array: Datatypen array av **char**	167
NullChar	Användning av nolltecknet vid stränghantering	169
EncryptTest	Anropar funktionen **krypt()** för att (de-)kryptera text	174
Encrypt	Funktion krypterar text med en slumpkrypteringsnyckel	175

Kapitel 9 Pekare

Pointer	Definition och initiering av en pekare	182
Value	Adress- och värdeoperatorn	187
new	Operatorn new: Att koda endast med pekarvariabler	192
PointArray	Sambandet mellan pekare och array	195
PointArithm	Pekararitmetik	198
Initials	Stränghantering med pekare Pekare som parameter i en funktion	199

Kapitel 10 Klasser, objekt och metoder

All_in_main	Vägen till objektorienterad programmering	207
Procedure	Funktioner som anropas från **main()** Modularisering på funktionsnivå	207 208

Program	Ämne	Sida
`Circle`	Modularisering på klassnivå. Vår första klass.	209
`Obj_Oriented`	Första objektorienterade program: Testar klassen **`Circle`**	211
`Employee`	Deklaration av klass som egendefinierad datatyp	216
`EmployeeTest`	Definition av objekt, punktnotation Åtkomst till objektets medlemmar	216
`TravelTime`	Klass med metod	223
`Travel_Test`	Objekt som parameter och returvärde	224

Sakregister

A

Abstraktion 213
Adressoperatorn 185
Array
  Definition 161
  for-sats 165
  Hakparenteser 162
  Indexgränser 162
  Initiering 161
  Tilldelning 164
ASCII 83
ASCII-tabellen med **for** 113

B

Bibliotek 30
Biblioteksprogram 36
Block 96, 148
Blockstruktur 148

C

cin 62
cin.getline() 171
Console 41
Console application 87
Console Application 11
cout 41

D

Datatyp 51
  Enkel 75
  Gränser 80
  Sammansatt 159
Dereference operator 186
do-sats 105, 159
  Programexempel 106

E

Escapesekvens 86
Evighetsslinga 109
Exekvering 26

F

Formell parameter 135
for-sats 110
  Nästlad 118, 121
    Flödesplan 120
    Inre-, yttre slinga 117
  Programexempel 111, 113
Funktion 132
  Anrop 136
  Begreppet 132
  Deklaration 138
  Externlagrad
    Programexempel 141
  Med returvärde 135
  Placering 131
  Prototyp 138
  Signatur 139
  Utan returvärde 144

G

Gissa tal-spel 100, 106, 145
Global variabel 149

H

Hexadecimala tal 87, 183

I

if-else-sats 98
  Programexempel 99
if-sats 92
  Programexempel 92, 95, 106, 108
Indata 62
Indexregeln 160
Indirektoperatorn 186
Indragningar 40
Initieringslista 163
Inmatning 62

J

Jämförelseoperator 94

K

Klass
- Datatyp 214
- Deklaration 215
- Sammansatt datatyp 213
- Skillnaden till objekt 212
- Test av 211
- UML-diagram 213
- Vår första 209

Kommentar 36
Kompilering 25
Kompileringsdirektiv 36
Konkatenering 44
Kontrollstruktur 91
Kryptering 174
- Text 174

L

Labyrint I 124
Lokal variabel 149
Loopar 105, 108, 110, 159
Länkning 26

M

`main()` 35
Manipulator 122
Maskinkod 26
Master Mind 204
Medlemsfunktioner 223
Metod 223
- Anrop 221
  - Nästlat 226

Modul 207
Modularisering 133
- Funktioner 208
- Klasser 209

Modularisering på funktionsnivå 208
Modularisering på klassnivå 209
Moduler 207
Modulo 66

N

Namngivning 53
- Regler 53

Namnutrymme (**`namespace`**) 37
`new` 191
- Programexempel 192

Nolltecknet 166
- Programexempel 169

O

Objekt 213, 218
- Definition 217
- Parameter 224
- Returvärde 224

Objektorienterad design 206
Objektorienterad programmering 206
Operand 64
Operator 64
- Aritmetisk 64

Overflow 78
Overriding 153

P

Paradigmskifte 206
Parameter 132
Parameterlista 135
Parameteröverföring 128
Parametrisering av globala variabler 151
Pekararitmetik 197
Pekare 180
- Asterisken * 187
- Programexempel 182

Pekare och array 195
Procedural programmering 208
Programmeringsstil 39
Punktnotation 220

R

Radbrytning i koden 42
Radbyte i utskrift 41
Radfortsättning i koden 43
Repetition 105, 159
- Bestämd 110
- Efter-testad 105, 159

För-testad 108
Reserverat ord 29
return-sats 135
Returtyp 135
Returvärde 132
Räckviddsoperatorn 154

S

Scope 152
Sekvens 91
Selektion 91, 92
Enkel 92
Tvåvägsval 98
sizeof 219
Skräp 55
Slumptal 117
slumpTal-funktionen 146
Sortering
Platsbyte 95
Standardbibliotek 30
Statisk minnesallokering 170
string 71
Sträng 166
Stränghantering med array 166
Stränghantering med pekare 199
switch-sats
break 104
Programexempel 103
SÅ LÄNGE 108

T

Textbehandling 166
TILLS 107
Typkonvertering
Explicit 81

U

UML 206
unsigned 80
Utmatning 41
Uttryck 64
Aritmetiskt 64

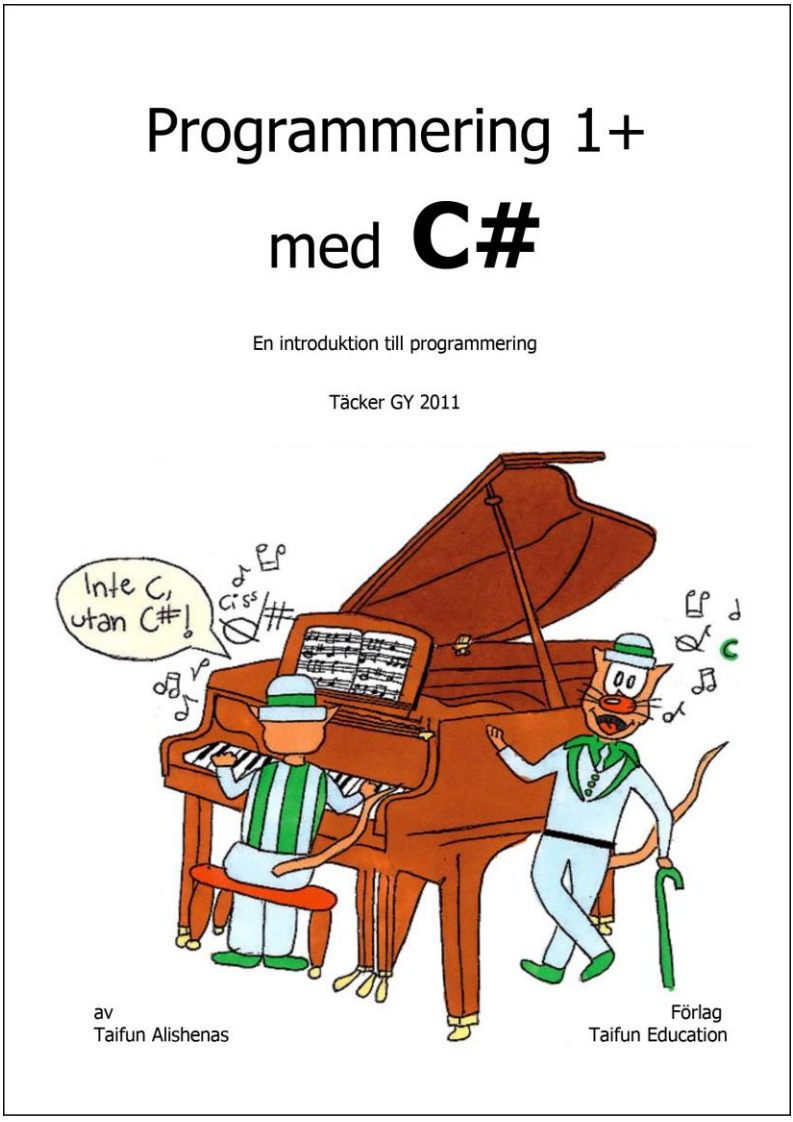
V,W

Variabel 53
Variablers räckvidd 149
Vita tecken 42
void-funktioner 144
Värdeoperatorn 186
Programexempel 187
while-sats 108
Programexempel 108

Ö

Ökningsoperator 67
Postfixvariant 67
Prefixvariant 67
Ökningsoperator 67
Överskrivning 60
Överskuggning 152
Variabler 152

Ladda ner smakprov från www.taifun.se:



*Programmering **1+** med C# (376 sidor)* innehåller allt som finns i *Programmering **1** med C# (286 sidor)* och dessutom mer om algoritmer, flödesschema, pseudokod, god programmeringsstil, logik, automatisk typkonvertering, Unicode, 2D arrays mm. För mer info ladda ner innehållsförteckningen från **www.taifun.se**.