Programmering 1

med **C++**

Med övningar,
projekt- och
inlämningsuppgifter

Titel: Programmering 1 med C++

Författare: Taifun Alishenas
info@taifun.se

Augusti 2023

Om boken

Välkommen till programmeringens spännande värld! När man tröttnat på att bara surfa, maila, lyssna på musik eller titta på film på datorn och nu vill veta mer om vad som händer bakom kulisserna, är det dags att börja programmera själv. Man lär sig nämligen på ett helt nytt plan hur datorer fungerar när man programmerar 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 använt program som andra skrivit och nu vill äntligen 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 koncept krävs det förstås ett intresse och en personlig fallenhet för ämnet, men det räcker inte. Det behövs också ett pedagogiskt koncept om hur kunskapen ska *förmedlas* för att verkligen nå läsaren. Boken försöker att ta maximal hänsyn till denna aspekt genom att eftersträva maximal enkelhet, utan att gå miste om den vetenskapliga noggrannheten och utan att tappa djupet i sak när det behövs – en balansgång som gäller att bemästras med sunt förnuft och erfarenhet.

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, projektuppgifterna samt inämningsuppgifterna. De sista två har mer tillämpad karaktär.

Gör så här:

- Ladda ned och installera programvaran Visual Studio (sid 41).
- Gå igenom bokens programexempel och testa dem i Visual Studio.
- Läs igenom teorin samt kodernas förklaring.
- Gör *övningarna* i slutet av varje kapitel.
- Genomför *projekt- & inlämningsuppgifterna* i slutet av övningarna.

  Projektuppgifterna är större än *övningarna*,
  inlämningsuppgifterna är större än *projektuppgifterna*.
- 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 augusti 2023

Innehåll

Kapitel 1

Introduktion

till programmering

1.1 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 reda på vad som väntas. Låt oss börja med att ge *negativa* svar, dvs att diskutera vad programmering *inte* är, för att sedan närma oss steg för steg 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 t.o.m. 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, i regel i alla fall – undantagen bekräftar regeln. Programmering kan jämföras med bilkörning. Det gäller ju bara att kunna *använda* en teknisk apparat, inte att bygga eller designa den.

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*, närmare bestämt en praktisk tillämpning av den. Som vi sa tidigare (sid 41):

> Programmering är i allra högsta grad ett praktiskt

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*. Gör man det med hjälp av datorn, har man att göra med datoriserad problemlösning. Så problemlösning är *målet* och programmering är *medlet* för att uppnå detta mål. Detta innebär att vi i regel inte programmerar för programmeringens skull utan för att lösa ett problem. Däremot kan man ibland göra det i pedagogiskt eller experimentellt syfte. Som sagt: undantagen 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

Här ska vi precisera det tredje negativa svaret. Det var Niklaus Wirth, skaparen av programspråket Pascal, som på 60-talet ställde upp definitionen:

> 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++.

1.2 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. Vi hänvisar här till *John von Neumann-modellen*, se avsn. ***1.5 Programmeringens historia*** (sid 27).

Editorer & IDE

En *editor* är ett skrivverktyg på datorn, dvs ett program som kan hantera text. *Ordbehandlingsprogram* är en annan beteckning på editorer. På de flesta datorerna finns ofta minst en editor förinstallerad. För att skriva källkod och spara den i en fil behövs en editor. Men källkod får endast innehålla tecken som kan tolkas av interpretatorn resp. av kompilatorn. Därför måste editorn spara filen som *oformaterad textfil*, dvs utan styr- och formateringskoder. 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. Även andra bra alternativ som TextPad finns att ladda ned från Internet. Ordbehandlare däremot av typ Word formaterar texten och sparar sina filer som dokument. Formatering innebär att det läggs till osynliga tecken i texten som interpretatorn resp. kompilatorn inte känner till. Därför är sådana program inte lämpliga för att skriva kod.

Generellt är nackdelen med en vanlig editor att man efter editering måste byta miljö, för att kunna kompilera och exekvera sin kod. Detta slipper man med en IDE.

En ***IDE*** står för ***I****ntegrated* ***D****evelopment* ***E****nvironment*, är alltså en *integrerad programutvecklingsmiljö* som samlar flera verktyg i en och samma miljö, så att man inte bara kan editera utan även kompilera, exekvera, felsöka, få online hjälp osv. Under kodens utvecklingsperiod är det en fördel att slippa byta miljö, speciellt mellan editering och kompilering. Därför är en IDE det ideala verktyget för en programmerare. Några exempel på IDEs för C++ är *Visual Studio*, *Visual Studio Code*, *Bloodshed Dev C++*, *Borland C++ Builder*, *Symantec C++*, *GNU C++*, *CLion …*. Vi kommer att använda *Visual Studio* vars hantering tas upp i detalj senare (sid 41 & 42).

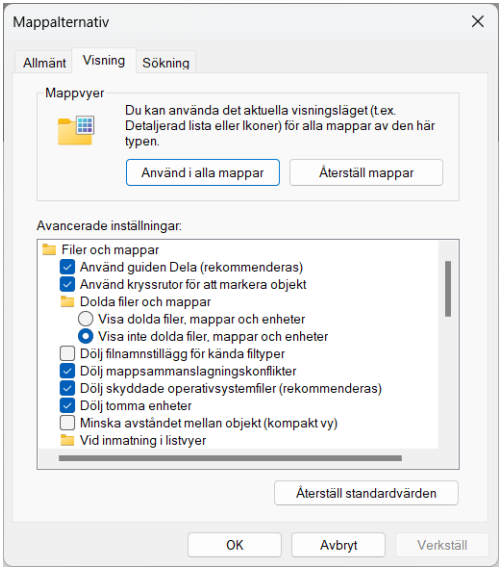
Regler för filändelsen

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 *Visual Studio* sparas alla C++ källkodsfiler automatiskt med filändelsen **cpp**.

Att hantera filändelser

För att kunna följa reglerna för filändelsen som beskrevs ovan, förutsätts att man kan *se* filändelserna när man öppnar en mapp. Men i praktiken är detta ofta inte fallet. Orsaken är på operativsystemets inställningar. I Windows t.ex. är default inställningen att man i regel *inte* kan se dem. Ta själv reda på hur det är på din dator. Så här kan man göra för att synliggöra filändelserna i Windows:

- Öppna en mapp i Windows.
- Gå i mappens menyrad till Mappalternativ. Om du inte hittar denna meny klicka på de tre små punkterna till höger (Visa mer) och välj Alternativ.
- Du borde få upp dialogrutan Mappalternativ. Välj fliken Visning. *Bocka av* rutan Dölj filnamnstillägg för ända filtyper. Så här borde nu dialogrutan se ut:
- Klicka på knappen Använd i alla mappar, sedan på Ja och OK.



Nu borde du kunna se dina filers ändelser och kunna följa reglerna på förra sidan. Generellt rekommenderas att ha synliga filändelser på sin dator, när man programmerar.

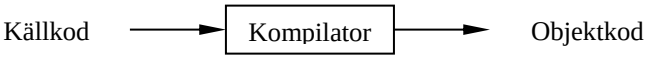
I andra operativsystem används andra ändelser [*].

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 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 47).

[*] 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.

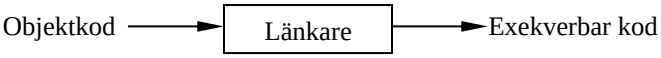
Men det finns olika typer av maskinkod, t.ex. körbar och icke-körbar. Kompilering av C++ källkod (lagrad i en fil av typ **`*.cpp`**) producerar först en icke körbar maskinkod som kallas *objektkod* och lagras i en *objektfil* av typ **`*.obj`**.



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 54). 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 47.

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:

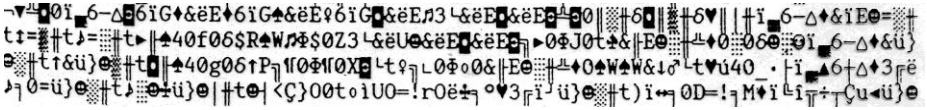


Det behövs ingen särskild åtgärd för att starta länkaren. Den startar automatiskt direkt vid exekveringen utan att man märker det. Endast om det uppstår fel är det relevant för felsökningen att veta att länkning förekommer som ett steg mellan kompilering och exekvering.

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 47).

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 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.

1.3 Algoritmer och deras beskrivning

Många tror att algoritmer endast har med matematik att göra. Även om algoritmer historiskt har introducerats av matematiker kan de användas på all problemlösning. Man kan t.o.m. tillämpa algoritmer på vardagliga problem. Samtidigt ligger de till grund för all programmering. Ett datorprogram är ingenting annat än en algoritm beskriven i datorns språk. Men även följande vägbeskrivning är ett fullgott exempel på en algoritm:

> *” ... gå ut från ditt hus till vänster, fortsätt rakt fram, sväng till höger vid trafikljuset, fortsätt sedan andra korsningen till vänster, där finns ett gult hus, på 2:a våningen bor jag ...”*

En *algoritm* är alltså ett tillvägagångssätt att lösa ett problem – vilket som helst. Och det behöver inte heller vara datorn som löser det. Vi kommer att precisera denna definition lite senare (sid 19). Problemet som ska lösas kan sakna lösning – då kan det inte heller finnas någon algoritm. Om däremot problemet är lösbart, kan det ha ingen, en eller flera algoritmer. Vi sysslar här endast med sådana problem som har minst en algoritm.

Historiens första algoritm

Det är alltid lärorikt att blicka tillbaka till historien. Själva ordet *algoritm* härstammar från ett namn på en person: namnet på den framstående persiska matematikern *Al-Kharazmi*[*]. Namnet har sedan latiniserats och blivit *algoritm.* Han levde på 800-talet. I sin berömda bok om *Algebra* ställde han upp historiens första algoritm som beskriver addition och multiplikation av heltal. Den används även idag. Men kunde man inte addera eller multiplicera heltal på 800-talet? Jo, redan långt tidigare kunde man räkna med tal i Egypten, Indien, Persien och Grekland. Vad var i så fall Al-Kharazmis historiska prestation? Ja, det var inte att komma på hur man *adderar* eller *multiplicerar* heltal – för det var ju redan känt, utan hur man i allmänna ordalag *beskriver* tillvägagångssättet, dvs formulerar en algoritm för dessa operationer.

1000 år mellan praktisk lösning och formell beskrivning

Det är anmärkningsvärt att *beskrivningen* av hur man räknar med heltal kom till mer än 1000 år efter den praktiska lösningen. Orsaken är att den korrekta, allmänna beskrivningen som ska hålla i *alla* tänkbara situationer, är mycket svårare att åstadkomma än den faktiska lösningen av ett eller en klass av problem. Att själv gå en väg som man känner till är enklare än att formulera en korrekt vägbeskrivning som alla förstår och kan följa. Anledningen är att algoritmer är *generella* till sin natur, och just det är tjusningen: Att försöka beskriva dem så att de håller i *alla* situationer, det är konsten. Detta gäller även idag: Program – det moderna sättet att beskri-

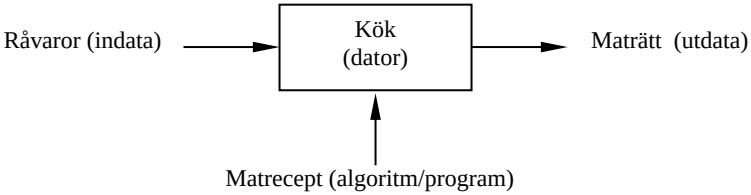
[*] Så uttalas hans namn på persiska idag (utan prefixet *Al-* som är arabiska). Han är född i *Kharazm*, en antik region som fanns i nuvarande nordöstra delen av Iran (Khorasan) mot Turkmenistan och Uzbekistan. På den tiden var Iran ockuperat av araberna.

va algoritmer – måste fungera under alla omständigheter och ska helst aldrig krascha. Dessvärre vet vi ju att så inte är fallet. En av utmaningarna inom programmering ligger just i att skriva program som fungerar i *alla* situationer. Det vi kan lära oss av det 1000-åriga glappet mellan praktisk lösning och formell beskrivning är: Satsa tid och energi på att först *analysera* det problem du vill lösa med ett program. Fokusera på att *beskriva* lösningen av problemet så generellt som möjligt.

Exempel på algoritmer

I vardagen använder vi algoritmer hela tiden, om än omedvetet. Här några exempel:

- **Matrecept** vars användning kan jämföras med programkörning på datorn:



  Matrecept skrivs fortfarande med vanligt språk men man kan konstatera att det finns en viss stil som är typisk för alla matrecept.

- **IKEA:s monteringsanvisningar** för att sätta ihop delarna till en möbel. Här används en kombination av text och grafik som är mycket effektiv. Grafiken förenklar algoritmen avsevärt. ”En bild säger mer än tusen ord.” På köpet får man en slags internationalisering, ett oberoende av det lokala språket, vilket gör att algoritmen förstås över hela världen.

- **Bruksanvisningar** av alla slag är exempel på algoritmer, även om många av dem i praktiken är värdelösa. Men det finns dåliga algoritmer på andra områden också.

- **Manualer** för datorprogram som visar hur ett program ska användas.

- **Konstruktionsritningar** som ingenjörer gör för att en viss produkt ska kunna tillverkas i fabrik. En arkitektritning av ett hus är ett specialfall av det. Här har grafiken tagit över helt och hållet.

- **Partiturer:** Noter i musik som används för att spela ett musikstycke och som omfattar noggranna anvisningar om hur en hel orkester ska spela. Ett speciellt ”språk” används som varken består av text eller grafik, utan snarare av symboler längs en tidslinje.

- **Spelregler** är snarare ett negativt exempel: De talar mest om vad man *inte* får göra och lämnar ett stort utrymme för hur man får spela inom reglernas ram. Därför finns två skilda problemställningar. Den ena är: ”Hur *får* jag spela?”

Spelregler ger delvis (negativa) svar på det. En helt annan problemställning är: ”Hur *vinner* jag spelet?” *Spelteori* som involverar sannolikhetslära behandlar denna fråga. I spelteori brukar man tala om *strategier* snarare än algoritmer. Här befinner vi oss i ett gränsområde där problem inte alltid har en entydig lösning eller saknar algoritm. I fortsättningen kommer vi att undvika sådana frågeställningar. Vi betraktar endast problem som är lösbara och har minst en algoritm. Exemplet belyser dock en viktig aspekt: Inte bara vägen till lösning måste beskrivas. Först måste *problemställningen* vara klart och exakt formulerad så att man kan avgöra om det finns en entydig lösning och minst en algoritm.

Definition av algoritm

Låt oss titta på vad som är *gemensamt* för exemplen ovan (utom spelreglerna), för att kunna fornulera en generell definition. Vilka typiska faktorer förekommer i alla exempel?

För det första består de alla av en rad anvisningar om vad som ska göras för att lösa det givna problemet. Frågan är: Ska man tillåta alla slags anvisningar? Om de leder till problemets lösning, varför inte? Men leder alla slags anvisningar till lösningen? T.ex. anvisningen ”Bygg ett hus!” är helt värdelös. Ingen av oss kan bygga ett hus med bara denna anvisning. Problemet är ju just *hur* man bygger huset. Anvisningarna måste vara mycket enklare och mer detaljerade. Vem som helst ska kunna utföra dem. Sådana anvisningar kallas *elementära instruktioner*. Bara sådana kan tillåtas i en algoritm om de ska leda till problemets lösning.

För det andra. Undersöker man de ovannämnda exemplens innehåll kan man konstatera att anvisningarna måste utföras i en viss *ordning.* Det går inte att kasta om ordningen. Man inser redan vid receptexemplet att man *först* måste knåda degen och *sedan* ställa in den i ugnen, inte vice versa. Vid partiturexemplet är ju ordningen helt avgörande. Och så är det i alla algoritmer. Ordningsföljden för de elementära instruktionerna måste finnas med i algoritmen. Självklart måste en algoritm också ange när instruktionerna ska upphöra. Om vi sammanfattar kan vi formulera följande definition:

> En *algoritm* är en följd av precisa anvisningar, s.k. *elementära instruktioner*, som löser ett givet problem, inklusive anvisningar om i vilken *ordning* instruktionerna ska utföras och när de ska avslutas. Dvs en algoritm måste ha ett exakt *avslutningskriterium*.

Av stor betydelse, speciellt för datoriseringen, är att algoritmen måste vara tolkningsbar *på ett enda sätt*. Det får inte finnas tvetydigheter i formuleringen. Datorn kan ju bara tolka våra anvisningar på ett enda sätt. Svårigheten ligger alltså i algoritmens *beskrivning*, vilket är en god illustration till det 1000-åriga glappet mellan praktisk lösning och formell beskrivning som nämndes på sid 17. Det är i regel svårare att *beskriva* en algoritm än att lösa ett specifikt problem i en specifik situation.

Anledningen är att algoritmer måste vara *generella* till sin natur: De måste hålla i *alla* situationer. Följande dilemma uppstår:

Hur beskriver man en algoritm bäst, så att den kan tolkas *endast på ett sätt*, men samtidigt behålla sin *generella* karaktär? Vi ska nu diskutera några hjälpmedel som kan användas för att formulera sådana algoritmer:

Olika sätt att beskriva algoritmer

- **Vanligt språk** är ett sätt att beskriva algoritmer, t.ex. vägbeskrivningen till en kompis. Största fördelen med det är att alla som kan språket direkt förstår algoritmen utan att behöva lära sig något nytt. Nackdelen är att det ofta kan tolkas på olika sätt. Och tur är det! Annars skulle man ju t.ex. inte kunna skriva en dikt eller njuta av den. Men just i samband med algoritmer då man eftersträvar entydighet, är möjligheten till olika tolkningar en nackdel.

- **Pseudokod** är en hybrid (blandning) mellan vanligt språk och formaliserad kod, ett försök att minska det vanliga språkets tvetydighet genom att införa vissa strukturer och t.o.m. grafiska stilmedel i layouten. Allt som på ett entydigt sätt beskriver en algoritm, även en matematisk formel, kan användas som pseudokod. I nästa avsnitt tar vi upp ett exempel på pseudokod med vanligt språk kombinerad med generella *kontrollstrukturer* (sid 24) som förekommer i alla algoritmer. På så sätt uppnår det vanliga språket en högre grad av entydighet, noggrannhet och struktur.

- **Flödesschema** eller flödesschema är en variant av IKEA:s monteringsanvisningar som kombinerar text och grafik med en klar dominans mot det senare. Man använder sig av geometriska figurer som symboliserar algoritmens byggstenar och av pilar som visar flödet i algoritmen och definierar instruktionernas ordning. Med dessa få stilmedel uppnår man en hög noggrannhet i beskrivningen, eliminerar tvetydigheter och åskådliggör algoritmens logiska struktur. Det tänkta händelseförloppet syns tydligt. I det avseendet är flödesschema överlägset både vanligt språk och pseudokod. Flödesschemassymbolik är ett utmärkt medel som lämpar sig inte bara för beskrivning av fullständiga algoritmer, utan också för att åskådliggöra logiken hos mindre, men kritiska delar av ett program. Vi kommer att använda oss av detta medel i hela boken.

- **Programkod** är den variant av algoritmbeskrivning som används för att låta en dator utföra algoritmen. Därför måste den kunna tolkas av datorn. Programkoden översätts till ett språk, kallat *maskinkod* som datorns processor förstår. Programkoden däremot – även kallad *källkod* – är skriven i något programmeringsspråk som man måste lära sig. Medan källkod förstås av människan, men inte av datorn, förstås maskinkod av datorn, men inte av människan.

- **Andra sätt** att beskriva algoritmer finns också. Inget av dem har lyckats etablera sig som standard. Anledningen är att det är oförutsägbart vilka metoder som i allmänhet kan lösa problem. Många av de traditionella sätten kan betecknas med det samlande namnet *pattern designs*. Andra använder begrepp

som *strukturdiagram*, *Mind Maps* eller *beslutstabeller*. Mest känt är dock *UML = Unified Modeling Language* som är ett språk för objektorienterad design och modellering. Man använder UML för att att planera, utveckla och visa strukturen hos avancerade objektorienterade system. UML används för att lägga upp och modellera stora programmeringsprojekt, vilket förutsätter bekantskap med den objektorienterade programmeringens terminologi. I nästa avsnitt ska vi börja utveckla de traditionella struktureringsverktygen *pseudokod* och *flödesschema*.

1.4 Pseudokod och flödesschema

Låt oss som exempel ta följande beskrivning på ren svenska av en vardaglig syssla:

> ”K. går upp kl. 6 och duschar tills kroppen känns fräsch. Sedan torkar K. sig, tar på sig kläderna och äter frukost. Vid frukosten lyssnar K. på radions trafikinformation. Om det är mycket biltrafik, går K. ut, väntar tills ingen bil kommer, går över gatan och tar bussen till jobbet. Annars tar K. bilen till jobbet.”

Det är en beskrivning av en algoritm, låt oss kalla den för *Morgonsyssla*, som använder sig av det vanliga språket. Egentligen kan den knappast misstolkas när den används med lite sunt förnuft. Ändå vill vi skriva om den, först som *pseudokod* och sedan som *flödesschema* för att lära känna de nya begreppen. Som vi ska se kommer detta att leda till en precisering av algoritmen.

Pseudokod till algoritmen Morgonsyssla

```
Gå upp kl. 6
Duscha TILLS kroppen känns fräsch
Torka och ta på dig kläderna
Ät frukost och lyssna på radio
OM det är mycket biltrafik
    gå ut
    vänta TILLS ingen bil kommer
    gå över gatan och ta bussen till jobbet
ANNARS
    ta bilen till jobbet
```

Låt oss analysera denna pseudokod lite närmare. Vad skiljer den från vanligt språk? Vi har gett texten en ny *form* utan att ändra *innehållet*. Nya ”regler” för formen har införts: För det första finns det varken punkter eller kommatecken mellan satserna. För att skilja dem åt, börjar istället varje sats på en ny rad. För det andra innehåller varje sats endast *en* elementär instruktion. För det tredje är vissa rader indragna vilket visar att instruktionerna på dessa rader, är underordnade andra instruktioner dvs är delar av dem. Så kan vi skilja mellan huvud- och underinstruktioner. Algoritmen har 5 huvudinstruktioner:

```
I.   Gå upp kl. 6
II.  Duscha TILLS kroppen känns fräsch
III. Torka och ta på kläderna
IV.  Ät frukost och lyssna på radio
V.   OM ...
     ANNARS ...
```

Att vi räknar **OM-ANNARS**-satsen som *en* instruktion, beror på att de hör ihop och bildar ett par: **ANNARS** skulle förlora sin mening om det skiljdes från **OM**. Dessutom är **OM-ANNARS** logiskt *uteslutande*, dvs alternativen under **OM** resp. **ANNARS** utesluter varandra och kan inte inträffa bådå. Sedan har algoritmen 4 underinstruktioner som är indragna, 3 under **OM** och 1 under **ANNARS**. Underinstruktionen ”gå ut” skulle kunna betecknas med V.a eftersom den tillhör huvudinstruktion V. Underinstruktionen ”vänta **TILLS** *ingen bil kommer*” skulle i så fall få beteckningen V.b. Undersinstruktionen ”gå över gatan och ta bussen till jobbet” blir V.c och ”ta bilen till jobbet” V.d. Hela algoritmen består av 5 huvud- och 4 underinstruktioner.

Villkor

Låt oss nu fördjupa analysen av pseudokoden och ta itu med de lite mer invecklade instruktionerna, t.ex. med II:an:

Duscha **TILLS** *kroppen känns fräsch*

Hur länge står K. under duschen? Innebörden av **TILLS** säger att detta avgörs av hur länge *kroppen känns ofräsch.* Dvs K. frågar sig ständigt, självfallet omedvetet: *känns kroppen fräsch, ja eller nej?* Om nej, fortsätt duscha! Om ja, sluta! Detta händer kontinuerligt under duschandet. Hur många gånger, är inte bestämt, utan avgörs av K.:s subjektiva svar på frågan. Menar K. att kroppen förblir ofräsch trots duschandet, då ska K. enligt algoritmen fortsätta att duscha i all evighet – rent hypotetiskt! I pseudokoden formuleras *känns kroppen fräsch* däremot inte som fråga, utan som ett *villkor* som ingår i **TILLS**-satsen, ett villkor för att fortsätta eller avsluta duschandet. Villkoret testas gång på gång: är det sant, ska K. avsluta duschen. Är villkoret falskt, ska K. duscha vidare. Valet avgörs av villkorets s.k. *sanningsvärde*, dvs om det är sant eller falskt. Ett villkor kan antingen vara sant eller falskt. På så sätt skiljer sig ett villkor från en instruktion. En instruktion utförs, medan ett villkor *testas*. Testet avgörs av villkorets sanningsvärde. Därmed avgörs även om den instruktion som knyts till villkoret, ska utföras eller ej.

Det finns flera villkor i pseudokoden, utmärkta i kursiv stil. Nästa villkor förekommer i huvudinstruktion V:

OM *det är mycket biltrafik*
. . .
ANNARS ta bilen till jobbet

Den kursiva texten är ett villkor som avgör om K. ska gå över gatan och ta bussen eller ta bilen till jobbet. Är villkoret sant (mycket trafik), då ska K. gå över gatan och ta bussen. Är villkoret falskt (inte mycket trafik), ska K. ta bilen till jobbet. Men till skillnad från **TILLS**-satsen testas villkoret här endast en gång, beroende på den annorlunda logiska innebörden av **OM**.

Ett tredje villkor finns i underinstruktionen V.b:

vänta **TILLS** *ingen bil kommer*

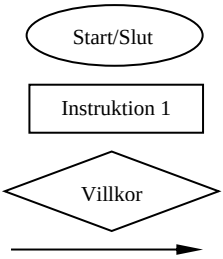
Logiken avgörs igen av **TILLS** dvs K. ska vänta så länge det kommer någon bil. När det inte längre kommer någon bil, ska K. sluta vänta. K. ställer sig gång på gång frågan: *kommer någon bil, ja eller nej?* Om ja, fortsätt vänta! Om nej, sluta vänta! Kommer det bilar hela tiden, då ska K. enligt algoritmen vänta i all evighet!

Kontrollstrukturer i algoritmer

Har vi därmed kartlagt pseudokoden till algoritmen Morgonsyssla? Nästan! Vi har identifierat *instruktioner* (normal stil) och *villkor* (kursiv stil). Vi nämnde även orden **TILLS** och **OM-ANNARS** (fet, versal stil), men vi har ännu inte identifierat dessa ord. De är ju varken instruktioner eller villkor, så vad är de? Låt oss för ett ögonblick glömma algoritmen Morgonsyssla och tänka oss en helt annan algoritm som ska lösa ett helt annat problem. Vilka ord skulle även förekomma i den nya algoritmen? Säkert ingen K.*, inget jobb, ingen dusch, ingen bil, ingen ... . Men just det! Orden **TILLS** och **OM-ANNARS** kan finnas i den nya algoritmen också. Och de kan förekomma inte bara i denna algoritm utan i alla algoritmer. De är nyckelord och fungerar som algoritmens byggstenar. I programmering kallas de för *kontrollstrukturer* eftersom de är generella strukturer som styr och kontrollerar hela algoritmen. Ja, alla algoritmer är uppbyggda av dessa kontrollstrukturer. Behärskar man dem, har man tagit ett stort steg mot förståelse av algoritmer och därmed förståelse för programmering. Det finns tre grundläggande kontrollstrukturer i alla procedurala programmeringsspråk:

- **Sekvens (följd)**
- **Selektion (val)**
- **Repetition (upprepning, loop)**

För att rita flödesschema används följande symboler:



Algoritmens start och slut ritas med en oval.

En **instruktion** ritas som rektangel. Ett **villkor** ritas som romb.

Villkoret skrivs in i romben och kan även formuleras som fråga.

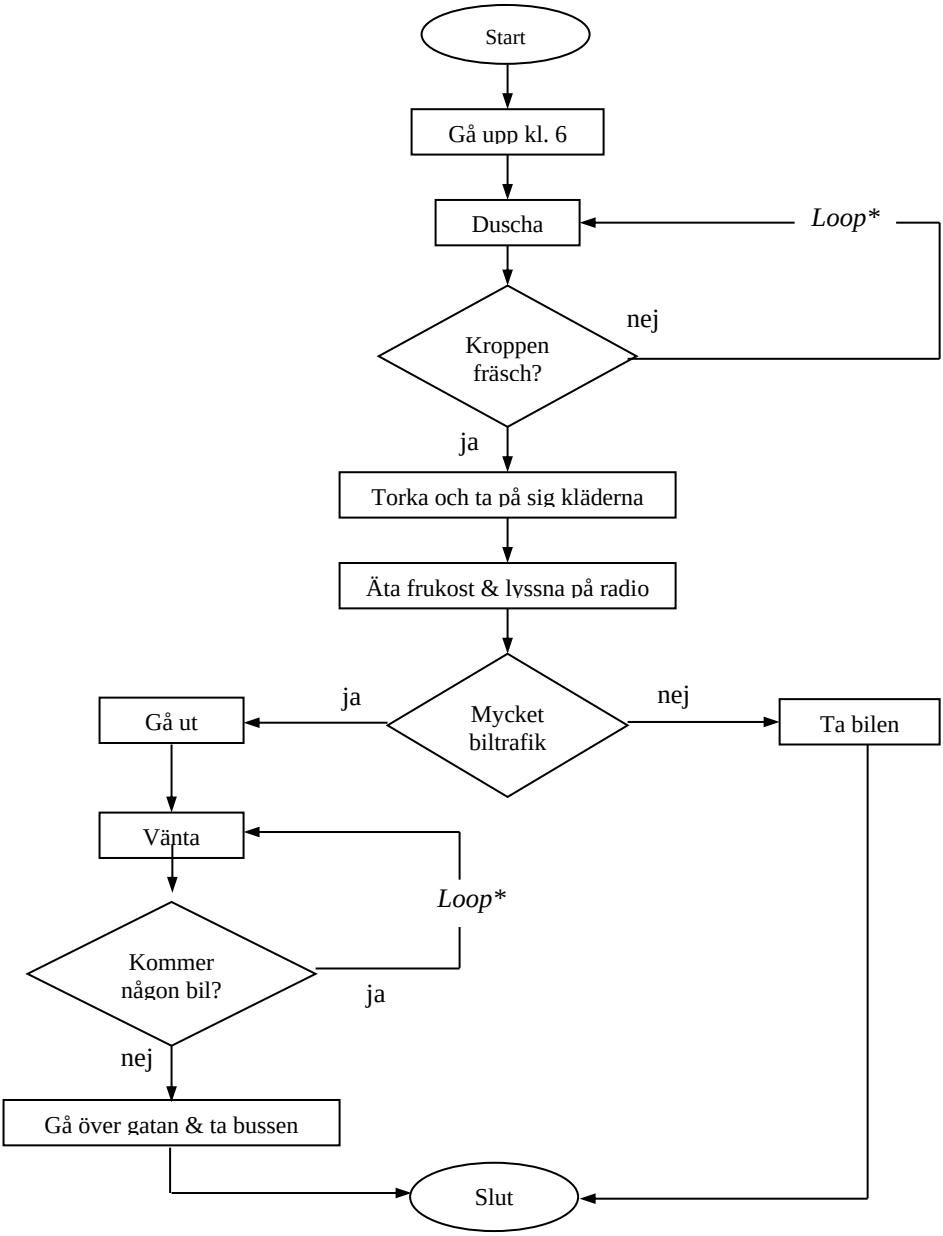
Ordningen i algoritmen (flödet) visas med pilar.

Det finns fler symboler än de som använts i flödesschemat till algoritmen Morgonsyssla som ska vara en exakt översättning av den algoritm som vi ursprungligen formulerade först på vanligt språk och sedan som pseudokod. Precis som vi gav texten i vanligt språk en ny *form* utan att ändra *innehållet* när vi skrev om den till pseudokod, ska även vid översättning till flödesschema ytterligare en ny form ges till algoritmen utan att ändra innehållet, framför allt inte den logiska innebörden. Flödesschemats fördel kan beskrivas med ordspråket *En bild säger mer än tusen ord*. Nu ska vi rita algoritmen Morgonsysslas flödesschema.

---

* Precis som i litterära verk protagonisten kan vara vem som helst (t.ex. Kafkas romanfigur ”Herr K.”) kan även algoritmens K. stå för *vem som helst*. I pseudokoden och även i flödesschemat på nästa sida förekommer inte ens K., vilket visar att det inte handlar om personen utan om *problemet ”Att ta sig till jobbet”*. Vi har att göra med problemlösning (procedural), inte med modellering av verkligheten (objektorientering).

Flödesschema till algoritmen Morgonsyssla



* *Loop* = upprepningsslinga med inbyggt villkor som testas gång på gång.

När vi säger att Morgonsyssla-algoritmens flödesschema ska bli en exakt översättning av den algoritm som vi ursprungligen formulerade på sid 22 menade vi förstås den *logiska* likheten, inte den språkliga. T.ex. står i pseudokoden ”vänta **TILLS** *ingen bil kommer*” medan i flödesschemat står ”Kommer någon bil?” och flödesschemat svarar på denna fråga: ”om ja, vänta” vilket innebär ”vänta **SÅ LÄNGE** *det kommer någon bil*”. Formuleringen är logiskt likvärdig med ”vänta **TILLS** *ingen bil kommer*”. Hade vi formulerat frågan negativt ”Kommer *ingen* bil?” hade det lett till dubbel negation vid svaret nej, vilket försvårar förståelsen. För att förenkla har frågan i flödesschemat formulerats positivt. Undersök själv om det finns flera exempel på språklig olikhet men logisk likhet mellan den ursprungliga texten och flödesschemat. Det är en utmärkt övning att kontrollera om vi på vägen från vanligt språk till flödesschema verkligen inte ändrat algoritmens innehåll.

Om man jämför pseudokoden med flödesschemat till Morgonsyssla kan man konstatera att det är avsevärt enklare att få en snabb överblick över algoritmen när man tittar på flödesschemat. Frågan uppstår varför man i så fall överhuvudtaget ska syssla med pseudokod. Svaret är att det är programkod som vi slutligen ska skriva, och programkod liknar pseudokod mer än flödesscheman. Vi kan inte mata datorn med grafik som är huvudingrediensen i flödesscheman. Pseudokodens värde ligger i närheten till programkod. Dessutom är den oberoende av programmeringsspråk. Flödesschema däremot är ett utmärkt hjälpmedel som kan användas *innan* man skriver programkod för att strukturera sina tankar om ett problems lösning som ska tas fram med ett datorprogram. Även detta verktyg är helt oberoende av programmeringsspråk. Är problemet enkelt eller om en klar struktur för lösningen redan finns, behövs ingen flödesplan. Växer problemets komplexitet rekommenderas en flödesschema kombinerad med pseudokod.

1.5 Programmeringens historia

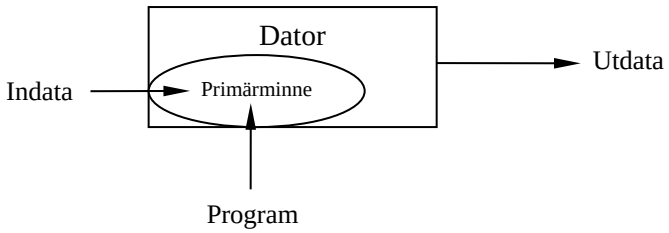
Programmeringens historia skulle kunna fylla en hel bok. Vi får nöja oss med ett urval, de mest kända programspråken. Denna framställning gör alltså inte något anspråk på fullständighet. Men samtidigt ska den förklara varför det finns flera hundra olika programspråk. Det är nämligen *funktionaliteten* som är avgörande.

Från vävstolarna till John von Neumann

Redan på 1800-talet programmerade man vävstolarna med jättelika slags trähålkort – en form av manuell programmering. Speldosor av olika slag vars melodier är förprogrammerade och stansade i cylinderformiga metalltrummor som rullar över en spik (1800-talets iPhones!), är ett annat exempel på manuell programmering. Även när de första datorerna konstruerades på 1930/40-talet, skedde all programmering manuellt. Man matade de stora maskinerna med både information (data) och instruktion (program) för att åstadkomma en liten beräkning. Dessa jätteapparater med en bråkdel av datorkraften hos en modern PC – en av dem: 35 ton och 16 meter lång – kunde lagra endast *data*. Men att även kunna lagra *instruktioner*, var inte löst än.

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.

Från maskinkod till Assembler

Så småningom kom man på idén att använda sig av kortkommandon på engelska som motsvarade instruktionerna i talform. Ett program tolkade sedan kommandona till maskinkod. Programmet kallades *assembler* eller *assemblator*. Kortkommandona var de första nyckelorden av programmeringsspråket *Assembler*.

50-talet **Assembler** betecknas som *lågnivåspråk* eftersom det är nära *datorns* språk utan att vara maskinkod. Fördelen med Assembler är att det är snabbt. Än idag finns det ingen kod skriven av människan som kan köras på datorn snabbare. Nackdelen med Assembler är att det inte finns *ett* språk som heter så, utan varje processor har sitt *eget* assemblerspråk. Dvs program skrivet för en datortyp kan inte köras på en annan. På 40-talet var datorerna tekniska underverk, byggda för hand. Varje dator hade sin egen programmerare, oftast tillverkaren själv som var specialiserad på just sin maskins assemblerspråk. I längden var detta ohållbart. Lösningen var att komma bort från maskinberoende språk.

De första högnivåspråken

1957 **FORTRAN** = **FOR**mula **TRAN**slator är historiens första *högnivåspråk* i den bemärkelsen att det ligger nära *människans* språk. Avståndet till maskinkod är större än hos Assembler. Därför måste en källkod i Fortran först översättas till maskinkod. Denna översättning kallas *kompilering* och är mer invecklad än assemblering. Den nya maskinkod som direkt kan köras, är mycket större än källkoden och lagras separat på hårddisken. Fortran är till skillnad från Assembler ett kompilerande språk. Dessutom är det som namnet antyder, i första hand inriktat på beräkning av matematiska formler. Än idag används fortranprogram av ingenjörer och vetenskapsmän som behöver snabba beräkningar. Men det finns även administrativa tillämpningar av Fortran. Språket har utvecklats och marknadsförts av företaget IBM.

1959 **COBOL** = **CO**mmon **B**usiness **O**riented **L**anguage är, som namnet säger, specialiserat på administrativa och ekonomiska tillämpningar. Det kräver hantering av stora datamängder vilket Cobol är bra på. Många stora banker och försäkringsbolag har kvar sina program som en gång var skrivna i Cobol. Även om det numera finns modernare språk, håller man ofta fast vid det gamla pga de stora kostnader som ett byte skulle innebära. Även

Cobol är ett högnivåspråk och därmed kompilerande. Cobol är utvecklat av USAs försvarsdepartement i samarbete med landets datorindustri.

1960 **ALGOL** = **ALGO**rithmic **L**anguage är det första språk som utvecklades i Europa. Det hade akademisk bakgrund: Initiativet låg hos det tyska *Gesellschaft für Angewandte Mathematik und Mechanik (GAMM)*. Man var ute efter ett verktyg för att utnyttja datorkraften för teknisk-vetenskapliga beräkningar på ett mer strukturerat sätt än Fortran. Beräkningarna skulle baseras på numeriska algoritmer snarare än matematiska formler. Algol som var ett kompilerande högnivåspråk, berikade programmeringen med många nya idéer och introducerade bl.a. *kontrollstrukturer* som används i algoritmer. Dessa har tagits över och vidareutvecklats i de moderna programspråken.

1963 **BASIC** = **B**eginners **A**ll-purpose **S**ymbolic **I**nstruction **C**ode är ett av de få högnivåspråk som inte är kompilerande utan *interpreterande*. Dvs källkoden tolkas rad för rad av datorns processor, utförs direkt och glöms bort sedan. Det uppstår ingen ny kod som lagras på hårddisken. Interpretering av källkod är alltid långsammare än exekveringen av redan kompilerad maskinkod. Däremot är interpretering snabbare än kompilering av källkod. I Basic finns inget kompileringssteg. Basic är, som namnet berättar, inriktat på att lära ut programmering för nybörjare. Därför har man hållit språket så enkelt som möjligt, så enkelt att man struntat i kontrollstrukturer som redan fanns i Algol och därmed lagt grunden för hoppsatser. Basic utvecklades ursprungligen av Dartmouth College i USA, men har sedan tagits över av Microsoft och integrerats som *QuickBasic* i DOS och Windows. På 90-talet har Microsoft lanserat vidareutvecklingen *Visual Basic* som blivit ett modernt och populärt utvecklingsverktyg. Den nyaste versionen heter *Visual Basic.NET* och är objektorienterad. I Visual Basic kan man även generera exekverbar kod i efterhand.

1971 **Pascal** är ingen förkortning för något utan har uppkallats efter *Blaise Pascal* som konstruerade räknemaskinen 1652. Pascal utvecklades av Niklaus Wirth på ETH (Eidgenössische Technische Hochschule) i Zürich. Tanken var att skapa ett kompilerande språk för att lära ut programmering för nybörjare genom att kombinera Basics enkelhet med Algols logiska strukturer och dess algoritmiska upplägg. På 80-talet utvecklade mjukvaruföretaget Borland *Turbo-Pascal* som blev en stor succé pga kompilatorns snabbhet och den *integrerade programutvecklingsmiljön (IDE)* som möjliggjorde kompilering, felsökning, editering och online hjälp i en och samma miljö. Idag marknadsför Borland Pascals objektorienterade vidareutveckling *Delphi*. Borland själv har också bytt ägare.

Från procedural till objektorienterad programmering

70-talet **C** utvecklades 1972 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 är hantering av bl.a. datorns primärminne med hjälp av *pekare*. Det är kod som ger åtkomst till den fysiska adressen av data och på gott och ont tillåter manipulationer av minnesadresser, vilket kallas för *pekararitmetik*.

80-talet **C++** är en direkt utvidgning och vidareutveckling av **C**. Denna övergång är en milstolpe i programmeringshistorien och markerar gränsen mellan procedural till objektorienterad programmering.

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*. Mer om C++ se nästa avsnitt.

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

90-talet **Java**s uppkomst motiverades av en annan utveckling inom IT som man skulle kunna kalla den grafiska eller Webbrevolutionen. Urspungligen har Java utvecklats av *Sun Microsystems* som ett projekt för att skapa ett språk för programmering av hushållsmaskiner. Men detta projekt visade sig vara en bubbla som sprack som mycket annat inom IT. Webben, som revolutionerade IT, blev räddaren i nöden för Java. Men Java är inte bara grafik och webb. Sun satsade på att utveckla Java till ett universellt ob-

jektorienterat språk som var plattformsoberoende. Idag används Java bl.a. för webbapplikationer, t.ex. *Java Server Pages (JSP)*.

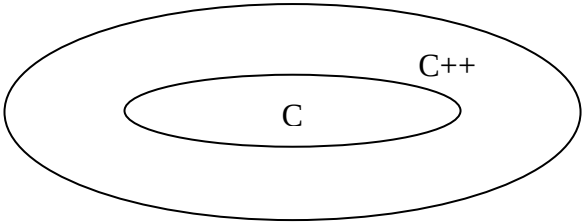
Sedan *Sun Microsystems* köpts upp av *Oracle*, är Java en Oracle-produkt. Oracle är en av världens ledande utvecklare av databashanterare. Java står inte i fokus av deras affärsverksamhet. Senaste tiden har Java tappat på popularitet inte minst pga sin lite krångliga kod jämfört med nyare utvecklingar som Python och C#.

90-talet **Python** skapades år 1989 av Guido van Rossum, en forskare på *National Research Institute for Mathematics and Computer Science* i Amsterdam och är en av de ovannämnda nyare utvecklingarna. Språket är *interpreterande* – liknande goda gamla BASIC – dessutom universellt. Python kan enkelt och gratis installeras på alla plattformar utan att man behöver bry sig om licenser. Koden är nästan självbeskrivande, ligger nära pseudokod och återspeglar algoritmen. I vissa avseenden är Python t.o.m. revolutionerande. Med små tekniska detaljer har man underlättat kodningen avsevärt. T.ex. har man avskaffat de obligatoriska symbolerna **{ }** för ett block. Det är inte längre nödvändigt att avsluta en sats med semikolon. De logiska indragningar som gör koden läsligare, har man lyft till obligatorisk syntax. Man är tvungen att följa god programmeringsstil. Variabler behöver inte explicit deklareras. Löpande kod och funktioner behöver inte nödvändigtvis skrivas i klasser. Språkets interpreterande karaktär gör det möjligt att på ett lekfullt sätt experimentera med kod. Pga dessa fördelar och sin enkla, smidiga och kloka kodningsteknik har Python mer eller mindre konkurrerat bort Java och kan idag anses som ett av världens mest populära programmeringsspråk, inte minst inom utbildning.

2000 **C#** har sina rötter i programspråken C, C++ och Java och är därmed byggt på det gamla, beprövade och välkända. Den allra första versionen av C# släpptes år 2000 av *Microsoft*. Man tog över allt som var bra och skrotade allt som var lite krångligt hos de andra språken. Men den viktigaste förnyelsen var att det nya språket integrerades i Microsofts .NET-miljö för att göra det utbytbart mot de andra språken inom .NET. En stor del av världens mjukvara utvecklas idag i C#.

1.6 Om programmeringsspråken C och C++

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” filosofins tilläggskaraktär:



C är nämligen en delmängd av C++. Därför är all C kod även C++ kod, men inte tvärtom. Eller: C++ inkluderar C. Därför lär sig den som lär sig C++ automatiskt även C. På datornivå: en C++ kompilator kan kompilera all C kod, men inte tvärtom.

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.

Reserverade ord (keywords)

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	...

Alla programmeringsspråk är definierade av ett antal nyckelord, på eng. *keywords*. De bildar språkets ordförråd. De kallas även för *reserverade ord*, därför att de inte får användas som namn (identifierare) för variabler eller programmets andra delar. Tabellen ovan visar en del av de resererade orden i C++.

Case sensitivity

Observera att alla reseverade ord skrivs med små bokstäver. Följande allmän regel gäller inte bara för reserverade ord utan för all C++ kod:

> C/C++ är case sensitive (skiftlägeskänslig).

Dvs i koden skiljer man mellan små och stora bokstäver. T.ex. är **`Tal`** och **`tal`** två *olika* variabler. Alla reserverade ord är giltiga endast med små bokstäver. Om vi t.ex. skriver det reserverade ordet **`new`** som **`New`** kommer C++ kompilatorn inte känna igen **`New`**. 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.

Standardisering

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

Bibliotek

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

Dessa små program ligger som ett skal kring den inre kärnan av reserverade ord och använder förstås i sin tur bara nyckelord eller andra fördefinierade program.

Om vi vill använda dem i våra egna program måste vi referera till dem med deras namn. Men dessa finns inte bland de reserverade orden. Vi måste därför tala om för kompilatorn vilka fördefinierade program vi tänker att använda och var de finns. Hur man gör det i praktiken visas i vårt första C++ program **`MyFirst`** (sid 54). Förklaringar om inkludering av bibliotek följer sedan på sid 56.

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

Sammanfattning

Om vi ska sammanfatta C++ språkets egenskaper, kan vi konstatera att C++ är:

- ett *kompilerande* högnivåspråk,
- ett *strukturerat* språk med rötter i C,
- ett *objektorienterat* programmeringsspråk efter utvidgningen av C
- ett *standardiserat* språk med ett stort bibliotek,
- ett *case sensitive* språk,
- ett *strikt typbestämt* språk, eng. *strongly typed language* (sid 74),
- ett *universellt* språk för alla möjliga tillämpningar,
- populärt hos professionella programmerare för utveckling av avancerade tillämpningar som operativsystem (Windows, Unix,…), spel, inbyggda system osv., men även:
- lämpligt för att lära nybörjare programmering,
- ett programspråk med stor spridning i världen.

Övningar till kapitel 1

Besvara följande frågor:

1.1 Vad är programmering?, sid 13-14

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

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

1.3 Vilken programmeringsfilosofi ligger till grund för den algoritmorienterade synen?

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

1.5 Vilken vetenskap är programmering en praktisk tillämpning av?

1.6 Vad borde steget före programmering alltid vara?

1.7 Hur definieras programmering ur den *procedurala* synen?

1.8 Hur definieras programmering ur den *objektorienterade* synen?

1.9 Varför ändrades synen på programmering under 80-talet?

1.10 Vad är skillnaden mellan procedural och objektorienterad programmering?

1.2 Kompilering & exekvering, sid 15-18

1.11 Vad innebär kompilering och hur skiljer den sig från exekvering?

1.12 Skriver man källkod eller maskinkod när man programmerar?

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

1.14 Vad gör länkningen som ett steg mellan kompilering och exekvering?

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

1.16 Vad bestod den tekniska innovationen av i den datormodell John von Neumann utvecklade 1944?

1.17 Informera dig på nätet om John von Neumann och undersök frågan om det var han som byggde världens första dator.

1.3 Algoritmer & deras beskrivning, sid 19-23

1.18 Följande pseudokod beskriver en algoritm för hårtvätt:

```
Start hårtvätt
Blöt håret
SÅ LÄNGE  håret känns smutsigt
      massera in shampo
      skölj
OM  solen skiner
      låt håret självtorka
ANNARS
      använd hårtorken
Slut hårtvätt
```

a) Vilka delar av pseudokoden är *instruktioner*, vilka är *villkor* och vilka är *kontrollstrukturer*? Förklara ditt svar.
b) Dela in instruktionerna i huvud- och underinstruktioner.
c) Rita ett flödesschema till pseudokoden ovan.

1.4 Pseudokod och flödesschema, sid 24-28

1.19 Följande algoritm – *Kalle-algoritmen* – är formulerad på vanligt språk:

> På vardagar går Kalle upp. Han tvättar sig, om mamman tittar på. På söndagar sover Kalle vidare tills mamman ropar honom till frukost, i så fall gör han som på vardagar.

a) Rita ett flödesschema till Kalle-algoritmen. Anta att lördag är en vardag.

b) Översätt Kalle-flödesschemat till pseudokod.

c) Finns det i Kalle-algoritmen möjligheten till en evighetsloop? När skulle den kunna inträffa? Hur kan den förhindras?

1.20 Rita flödesschemat till följande pseudokod:

```
Sätt på radion
Välj en kanal och  lyssna
SÅ LÄNGE  du inte har hittat ett bra program
      byt kanal
      lyssna
Fortsätt att lyssna på det valda programmet
Stäng av radion
```

1.21 Rita ett flödesschema till följande pseudokod:

```
Start Vinterklädsel_1
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_1*

1.22 Algoritmen i övn 1.21 ovan kan formuleras med följande pseudokod:

Start *Vinterklädsel_2*
Läs av temperaturen
VÄLJ *fall* ur
temperatur < 0: ta sjal, mössa och handskar
temperatur < 5: ta sjal och mössa
temperatur < 10: ta sjal
Annars: slipper du vinterklädsel
Slut *Vinterklädsel_2*

Rita flödesschemat till pseudokoden ovan och undersök den logiska likheten mellan flödesscheman i övn 1.21 och övn 1.22.

1.23 *Lothar Collatz* (1910-1990) var professor för tillämpad matematik vid Hamburgs Universitet på 60-talet. Förresten var han Taifuns lärare i matematik. Som ung student ställde Collatz upp följande algoritm:

> Tänk dig ett positivt heltal (startvärde).
> Är talet udda multiplicera det med 3 och addera 1.
> Är talet jämnt dividera det med 2.
> Gör samma sak med resultatet. Fortsätt **tills** du fått 1.

Det visar sig att talföljderna i denna algoritm alltid slutar med 1 oavsett startvärde. Algoritmen är även känd som *Collatz förmodan*, på nätet: *Collatz conjecture*. Förmodan heter den eftersom påståendet att den alltid slutar med 1, hittills är matematiskt obevisat. För att kunna testa påståendet i praktiken behöver vi ett datorprogram (inget bevis!). För att förbereda implementationen:

Rita ett flödesschema och skriv en pseudokod för denna algoritm.

1.24 Är följande pseudokod logiskt identisk med *Kalle-algoritmen* i 1.19, sid 36?

Start Kanske_Kalle?
OM *det är söndag*
sover Kalle vidare
TILLS *mamma ropar till frukost*
ANNARS
går han upp
OM *mamma tittar på*
tvättar han sig
Slut Kanske_Kalle?

1.5 Programmeringens historia, sid 29-33

1.25 Vilket var det första programmeringsspråk som utvecklades för de första datorerna? Vilka egenskaper hade det? Vad är dess största skillnad till dagens programmeringsspråk?

1.26 Vad karaktäriserar de programmeringsspråk som kallades för lågnivåspråk? Varför ”lågnivå”?

1.27 Vilket var det första högnivåspråket? Varför ”högnivå”?

1.28 Redogör för skillnaderna mellan begreppen assemblering, kompilering och interpretering.

1.29 Nämn ett exempel på programmeringsspråk som använde en av metoderna i frågan ovan.

1.30 Vad var det första användningsområdet för programmering?

1.31 Finns det fortfarande kod som används som är skriven i något av de första programmeringsspråken? Nämn några sådana samt deras användningsområde.

1.32 Vilket var det första programmeringsspråk som introducerade kontrollstrukturer i programmeringen?

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

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

1.6 Om programmeringsspråken C och C++, sid 34-36

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

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

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

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

1.39 Är C++ källkod eller maskinkod?

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

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

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

1.43 Är C++ ett standardiserat programmeringsspråk?

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

1.45 Ingår biblioteket i C++ standarden?

1.46 Får man använda reserverade ord som namn för identifierare i sina program?

1.47 Får man använda bibliteksnamn till sina egna identifierare?

1.48 Vilka regler gäller för namngivning av identifierare?

1.49 Vilka rekommendationer kan man anföra för namngivning av identifierare?

*

[*] Man kan testa Collatz algoritmen i appen *Mattekollen* där den är kodad i Python. Ladda ned appen eller kör den som Webbapp: **app.mattekollen.se → En mobil pythonmiljö**. Eller kör den direkt som webbapp: beta.mattekollen.se/#/app/coding. Prova koden med olika startvärden för att kolla om algoritmens talföljder alltid slutar med 1.

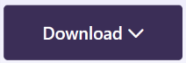
Kapitel 2

Programmeringsmiljön

2.1 Installation av Visual Studio

Programmering är i allra högsta grad ett praktiskt ämne.

- Gå till webbadressen: https://visualstudio.microsoft.com/vs/

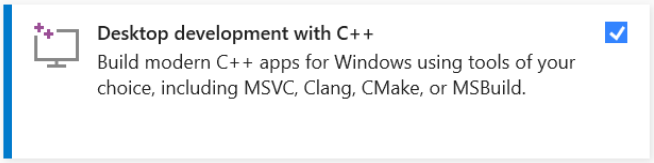
  Gå med musen över knappen Download: 

  En dropplista dyker upp. Välj: Community 2022

- 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.

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

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



- 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.

- 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.

- 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 44, för att testa ditt första C++ program.

2.2 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 14 & 15).

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.

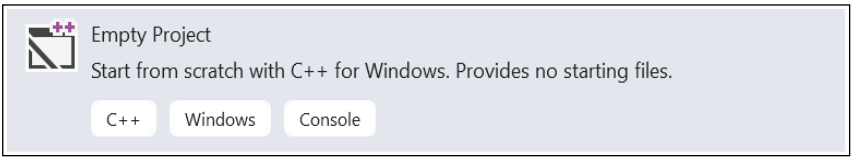
2.3 C++ Console Applications

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

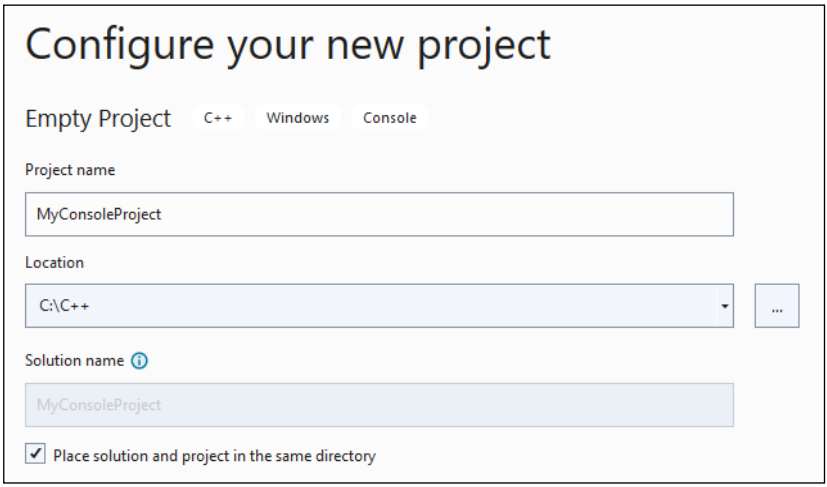
Start → Visual Studio 2022

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. Välj i dropplistan på rutans andra rad språket C++. Markera bland rutorna som dyker upp, rutan Empty Project som ser ut så här:



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



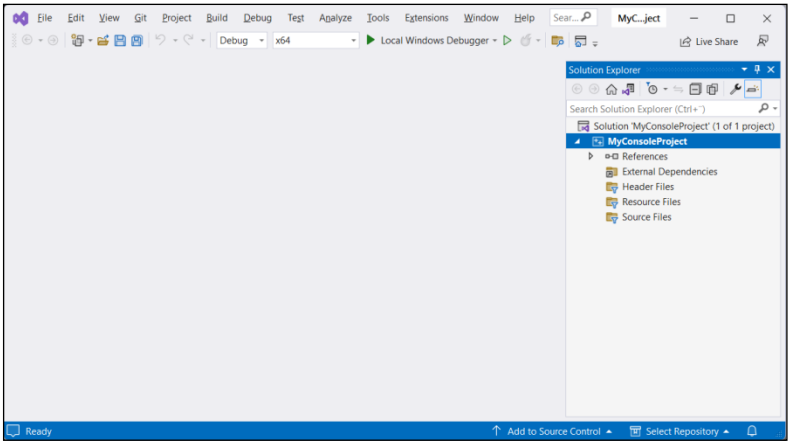
   I den övre delen av dialogrutan ovan döper vi vårt projekt till MyConsoleProject. I textrutan Location anger vi sökvägen till den mapp vi vill placera vårt projekt i. Låt oss säga vi vill placera projektet i en mapp som vi kallar för C++ och placera mappen i sin tur i enheten C:\. I så fall anger vi som Location C:\C++. Här kommer nu det nya projektet MyConsoleProject att placeras. 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.

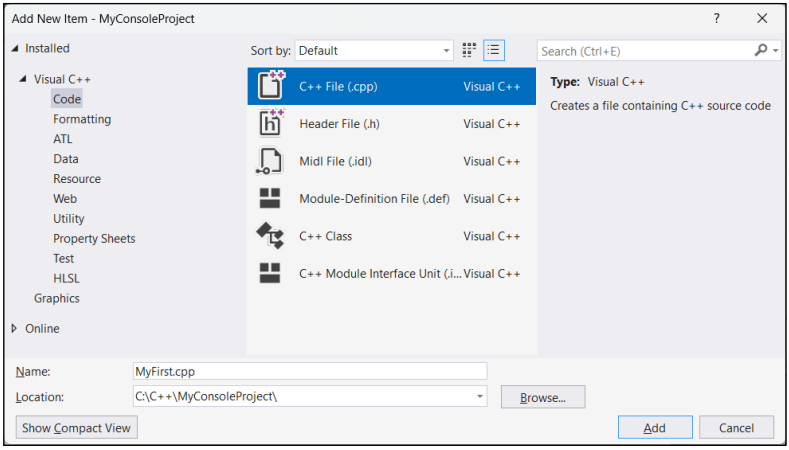
Vi valde ju i steg **1a** uttryckligen ett Empty Project (sid 44). Vi måste lägga in en C++ källkodsfil i projektet. Det finns två alternativ att göra det: Antingen vill vi skapa ett helt nytt program, skriva in koden, spara den i en fil och infoga den i projektet eller vi vill lägga till en redan befintlig 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…

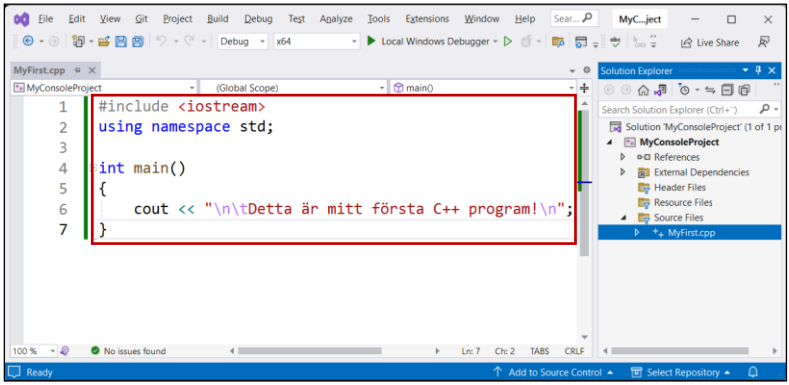
Följande dialogruta dyker upp:



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. Låt den redan angivna projektmappen stå. I så fall kommer filen MyFirst.cpp att skapas där. Klicka på Add-knappen.

Du återvänder du till Visual Studios ursprungliga miljö, där det skapats ett stort vitt editfönster under fliknamnet MyFirst.cpp. I Solution Explorer-fönstret ser du att filen MyFirst.cpp har lagts till projektet. Markera filnamnet i Solution Explorer och skriv i editfönstret koden nedan (rött inramad):

Programmet `MyFirst`



En utförlig förklaring av koden ges på sid 54. Radnumreringen till vänster och kodens textstorlek kan ställas in i Visual Studio.

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. 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 (sid 14), vilket kan göras från menyraden längst upp med:

Build → Compile

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

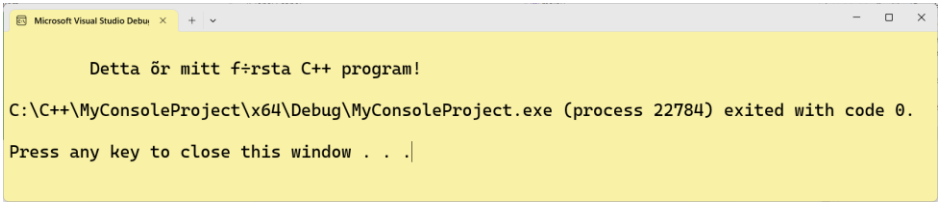
```
Build started...
1>--- Build started: Project: MyConsoleProject, Configuration: Debug x64 ---
1>MyFirst.cpp
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

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.

För att *exekvera* koden klicka i menyraden längst upp på menyn:

Debug → Start Without Debugging

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



Som man ser är de svenska tecknen **ä** och **ö** förvrängda. Vi kommer att åtgärda detta på sid 51 och förklara mer på sid 58. Men i fortsättningen kommer vi att förbise detta estetiska fel i våra utskrifter och koncentrera oss på programmeringen.

Du kanske har svart bakgrundsfärg i konsolen. Om du vill ändra bakgrundsfärg, storlek eller annat på konsolfönstret, kan du klicka på den lilla pilen till höger i menyraden, välja Settings och gå vidare.

Du har nu skapat och testat din första C++ Console Application. Programmet **`MyFirst`** finns i filen MyFirst.cpp och den i sin tur i projektet MyConsoleProject. Men, för att vara oberoende av Visual Studio borde du helst spara filen dessutom på en annan plats än projektetmappen. Läs mer om detta under ***Organisera dina C++ filer*** i nästa avsnitt (sid 50).

Stäng Visual Studio. Nästa gång du öppnar Visual Studio, för att testa andra C++ program, gör *inte* om allt du gjorde nu. Följ istället instruktionerna i nästa avsnitt (eller i övn 1.1, sid 51).

2.4 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 44) 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 41.

Övningar till kapitel 2

2.1 Installera Visual Studio på din dator enligt instruktionerna på sid 41.

Skapa ett *projekt* i Visual Studio av typ *C++ Console Application* enligt instruktionerna på sid 44. Infoga programmet **MyFirst** (sid 46) i projektet.

Kompilera och exekvera projektet. Får du utskriften på sid 47?

2.2 Programmet **MyFirst** förvränger de svenska tecknen **ä** och **ö** i utskriften (sid 47). För att få korrekta svenska tecken i utskriften, ändra koden enligt följande och kör:

```
// MyFirstSwed_1.cpp
// Skriver ut de svenska specialtecknen korrekt med hjälp av deras
// hexadecimala koder.

#include <iostream>
using namespace std;

int main()
{
   cout << "\n\tDetta \x84r mitt f\x94rsta svenska C++ program!\n";
}
```

Det vi här och i fortsättnigen – lite slarvigt – kallar för *svenska specialtecken* är inget annat än de svenska tecknen **ä**, **å** och **ö** samt deras versaler. Här används deras hexadecimala koder. Bry dig inte just nu om dessa koder (**\x84** och **\x94**) utan baka in dem i utskriftssträngen, för att få korrekt utskrift. De kommer att förklaras senare (sid 124). I fortsättningen kommer vi dock för enkelhetens skull att bortse från förvrängningen av de svenska tecknen.

2.3 Ta bort i dina program satsen **using namespace std;** Ersätt istället **cout** med **std::cout**. Kompilera och kör. Vilken slutsats kan man dra?

2.4 Skapa och testa ditt andra C++ program **MySecond** genom att göra så här:

a) Stäng och öppna igen VisualStudio. Skapa *inte* ett nytt projekt utan klicka istället (på vänstersidan) under rubriken Open recent på ditt gamla projekt MyConsoleProject.sln som skpades för att testa vårt första program **MyFirst**. Ditt första projekt öppnas.

b) Gå till Solution Explorer-fönstret. Markera projektnamnet MyConsoleProject, högerklicka på den **cpp**-fil som detta projekt innehåller och välj:

Exclude From Project

Filen tas bort från projektet.

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 45). 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 47. Du borde få följande utskrift (bortsett från de svenska tecknens förvrängning):

```
    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.

Förfarandet som beskrivs här, förklaras även i avsnitt ***2.4 ETT projekt för alla konsolapplikationer*** (sid 49). Läs mer där.

Kapitel 3

Att komma igång

med C++

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 ("si-out" 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 den s.k. *konsolen* (Windows' kommandotolk), vars bakgrundsfärg man kan ändra, om man vill:

```
            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 avbildas inte här.

Ett annat dilemma är de svenska tecknen **ä** och **ö**. Orsaken till deras förvrängningar är icke-standard kodningar. Detta har både med C++ och miljön att göra. 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 58) och löses generellt på sid 125. Vi ska dock koncentrera oss på C++ koden: Varför kallas **MyFirst** för ett *program?* Är inte all kod i C++ ett *program?* Faktiskt inte!

Program i C/C++

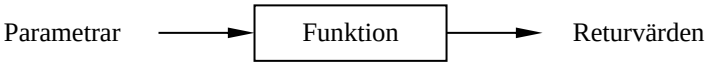
> Ett C/C++ program är en samling funktioner (C) och klasser (C++).
> Endast en av dessa funktioner måste vara funktionen **main()**.
> Funktionen **main()** är startpunkten för programexekvering.

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. Man kan förstås invända: Vad är en *funktion*? Vad begreppet *funktion* exakt betyder kommer att behandlas senare i ett separat kapitel.

Men eftersom funktionen **`main()`** spelar en avgörande roll ska vi titta nämare på den nu:

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 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 klamrar **`{ }`** 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 **`{ }`**. På engelska heter de *braces*.

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`** (uttal: ”si-out”). 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 15) 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 på skärmen. Om vi tittar på tabellen för reserverade ord (sid 32), ser vi att **`cout`** inte finns där. Faktum är att **`cout`** inte är ett reserverat ord utan kod som är förprogrammerad i *biblioteket* **`<iostream>`** (sid 33). För att kunna använda **`cout`** måste vi inkludera detta bibliotek i vårt program vilket görs med den första av de två raderna ovan: Raden inleds med **`#include`** dvs koden i biblioteksfilen **`<iostream>`** ska inkluderas, läggas till vår egen kod. Först då blir **`cout`** definierat i vårt program. Biblioteket **`<iostream>`** innehåller förprogrammerade rutiner för in- och utmatning där **`io`** (uttal: ”aj-å”) står för **`i`**nput/**`o`**utput. En mer exakt förklaring ges i avsnitt ***3.3 Utmatning med cout och <<*** på sid 61.

Namespace `std`

Man vill tillåta att inkludera även andra bibliotek, både egna och sådana som ev. kommer från andra programtillverkare. Samtidigt vill man undvika namnkonflikter mellan C++ biblioteket och tredjehandsprogram. 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 just 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. Läs mer om **`cout`** på sid 61.

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 54) 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 124).

`\n` är vårt första exempel på en escapesekvens. Ett annat exempel som förekommer i programmet **`MyFirst`**, är **`\t`** där **`t`** står för ***`t`****`abulator`* och **`\`** 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 125):

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

```
// MyFirstSwed.cpp
// Skriver ut de svenska tecknen korrekt med hjälp av deras koder
// Två metoder: 1. Explicit typkonvertering
//              2. Escapesekvenser

#include <iostream>
using namespace std;

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

   cout << "\n\n\tEscapesekvenser:";
   cout << "\n\tKoden \\\\   ger tecknet "<< '\\';
   cout << "\n\tKoden \\x84 ger tecknet " << '\x84';
   cout << "\n\tKoden \\x86 ger tecknet " << '\x86';
   cout << "\n\tKoden \\x94 ger tecknet " << '\x94';
   cout << "\n\tDetta \x84r mitt f\x94rsta svenska C++ program!\n";
}
```

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

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

        Escapesekvenser:
        Koden \\   ger tecknet \
        Koden \x84 ger tecknet ä
        Koden \x86 ger tecknet å
        Koden \x94 ger tecknet 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 121 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 54). 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 ("si-out" Console output) med utmatningsoperatorn <<
// Kommentarer // , radbytet \n och tabulatorn \t

#include <iostream>
using namespace std;

int main()              // Startpunkt 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 84.

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 och <<

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

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

där . . . är en *sträng*, dvs en följd av tecken (sid 57). 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 " " .

Vad är cout *egentligen?*

Hittills har vi talat om **cout** som ”kod” som som är förprogrammerad i ett bibliotek (sid 56). Närmare bestämt är **cout** ett objekt av klassen **iostream** som är definierad i biblioteksfilen **<iostream>**. **cout** (Uttalet: ”si-out”) står för **c**onsole **out**put, där *console* ursprungligen är en beteckning för *terminal* som i sin tur betyder bildskärm och tangentbord. Men i vårt sammanhang menas med *console* fönstret som dyker upp, när man exekverar ett C++ program. **cout** genomför utskrift till konsolen. Men **cout** kan inte ensam åstadkomma utskrift. För att *utföra* utskrift behövs en *operator*, nämligen:

Utmatningsoperatorn <<

Två mindre än-tecken **<** och **<** skrivna *utan* mellanslag bildar *utmatningsoperatorn* **<<** som används tillsammans med **cout**. **<<** är som en pil från höger till vänster:

Utskrift i konsolen ⟵ **Sträng (eller annat)**

Dvs **Sträng** skickas till **utskrift**: Data strömmar i pilens riktning till datorns *standard output*-enhet som är konsolen (**c**onsole **out**put). **cout** och utmatningsoperatorn **<<** är ett par som hör ihop. De får inte användas ensam.

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 << "\n\tDetta 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 det blir **producerad** i utskriften och inte **produc erad**, vilket beror på att inget mellanslag skrivs i koden *efter* **produc** och inte heller *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 skrivas ut i nästa **cout**-sats. Det som skickas till **cout**-strömmen visas kontinuerligt i konsolen. **cout**-strömmen är indelad i två **cout**-satser.

Frågan är nu: Om det 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, vilket visas i nästa avsnitt.

3.4 Konkatenering

```
// Concat.cpp
// Sammanslagning av flera cout-satser
// Att bryta en cout-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";
}
```

Till skillnad från programmet **Cout** (sid 61) ger programmet **Concat** två rader text:

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

Samma teknik används i nästa program för att ”rita” i konsolens textmiljö.

Att "rita" i textmiljö

Programmet **Figure** klarar sig med en enda **cout**-sats genom att slå ihop delsträngar, utan att upprepa utmatningsoperatorn **<<** :

```
// 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";
}
```

Tekniken att – med en enda **cout**-sats samt radbrytningar i koden på rätt ställe –

skriva ut resp. ”rita” figurer i konsolen, kallas för *konkatenering*, som betyder ihopslagning av strängar. I själva verket slås de delsträngar ihop som står på en separat rad utan man skriver någon kod mellan dem, bara man bryter rad på rätt ställe i koden. Konkatenering kan användas bl.a. för att lösa uppgifterna 3.3-3.7 i slutet av detta kapitel (sid 68).

Concatenation *

eller ***cat****enation* betyder sammanslagning och förekommer i en rad olika sammanhang, inte bara i C++. I programmet **Concat** har vi konkatenerat de strängar som i programmet **Cout** (sid 61) stod i två separata **cout**-satser, till en enda. Generellt kan man skriva en enda **cout**-sats och skicka utmatningsoperatorn **<<** mellan de olika delsträngarna. Den allmänna strukturen ser ut så här:

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

där `...` står för de delsträngar 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. Programmen **Concat** och **Figure** visar att man kan klara sig med en enda **cout** och en enda utmatningsoperator **<<** om man bryter rad i koden på rätt ställe. Men av dessa program framgår inte vilken kod det egentligen är som åstadkommer konkatenering.

Konkateneringsopratorn +

Här vill vi introducera en operator som konkatenerar strängar i C++, men konstigt nog inte *strängkonstanter* utan *strängvariabler*. Följande program som producerar samma utskrift som programmet **Concat** (sid 63) visar detta:

```
// Concat_var.cpp
// Konkatenering av strängvariabler med
// konkateneringsoperatorn +
#include <iostream>
using namespace std;

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

Begreppet *variabel* kommer att behadlas i nästa kapitel. Kort sagt, är en variabel en platshållare för ett värde som kan vara tal, tecken, sanningsvärde, men även sträng.

* 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**.

Förklaringen

Vad är i så fall skillnaden mellan programmen **Concat** och **Concat_var**? Och varför kan man inte använda konkateneringsoperatorn **+** mellan sträng*konstanter* (**Concat**), däremot mellan sträng*variabler* (**Concat_var**)?

I koden upptäcker vi en hel del nytt. Det viktigaste är **+** i sista satsen:

```
cout << s + t;
```

Operatorn **+** betyder här inte längre addition av tal utan *konkatenering* av strängvariablerna **s** och **t**. Det är datatypen **string** som avgör den aktuella tolkningen av **+**. Variablerna **s** och **t** är nämligen av datatypen **string**. Begreppen *variabler* och *datatyp* kommer att tas upp i nästa kapitel.

I C++ kan man inte använda **+** på sträng*konstanter*. T.ex. skulle följande kod ge kompileringsfel. Testa gärna själv detta exempel som ersätter variablerna **s** och **t** med de motsvarande konstanterna:

```
cout << "Nu kommer två rader text\n " + "producerade av EN sats.";
```

Däremot kan man sätta konkateneringsoperatorn **+** mellan sträng*variabler*.

Anledningen till denna svårbegripliga regel är att **+** är definierad som operator i klassen **string** och som sådan endast kan tolkas mellan två *objekt* av denna klass. Strängkonstanter däremot (som omslutas med citationstecken) är inga objekt av klassen **string** utan tolkas i C++ som *pekare*-till-**char**, minnesadresser till en samling av tecken. För pekare-till-**char** är operatorn **+** inte definierad.

3.5 Radfortsättning

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

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

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

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

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

Lösningen är att dela upp strängen i *två* delsträngar och – som en ytterligare förenkling – att utelämna utmatningsoperatorn:

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

I koden ovan har vi *två* delsträngar som **cout**-satsen konkatenerar automatiskt. Ett annat alternativ är *radfortsättning* som bibehåller strängen:

Radfortsättningstecknet ****

Tekniken att åstadkomma radfortsättning i C/C++ kod är:

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

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

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

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

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

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

#include <iostream>
using namespace std;

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

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

```
    Detta är endast en sträng kodad med radfortsättningstecknet.

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

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

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

Övningar till kapitel 3

3.1 Har du en favorit editor (sid 13)? Om ja, öppna den. Om inte, ladda ned open-source editorn Notepad++ och installera den. Undersök i editorn skillnaderna – vad gäller formen och utseendet – mellan tecknen *apostrof* (**'**), *citationstecken* (**"**), *accent* (´) och *backslash* (****). Ta reda på och kom ihåg deras tangenter på ditt tangentbord.

3.2 Visar din dator filändelserna när du öppnar en mapp? Om inte, genomför instruktionerna ***Att hantera filändelser*** på sid 14 för att synliggöra filändelserna.

3.3 Mata in koden till programmet **`MyFirst`** (sid 54), 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`**. Kompilera och kör. Läs om detta på sid 57.

3.4 Modifiera programmet **`MyFirst`** (sid 54) så att du får följande utskrift:

3.5 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.

3.6 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.

3.7 Vilken utskrift ger följande satser? Sätt in dem i ett program och testa.

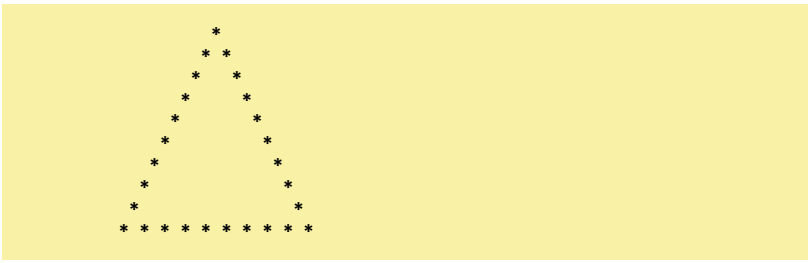
```
cout << "*\n**\n***\n****\n*****";

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

3.8 Varför ger följande program kompileringsfel? Åtgärda felet:

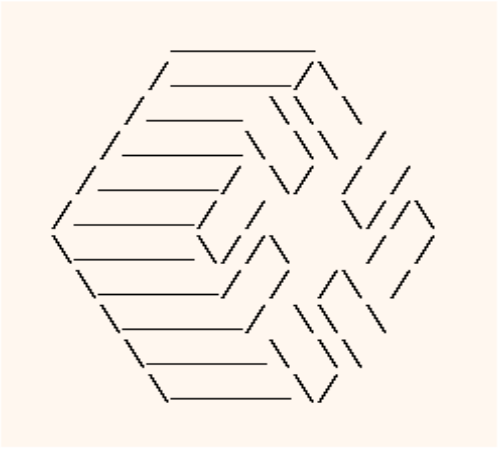
```
#include <iostream>
using namespace std;
int main()
{
   cout << "\n\tNu kommer två rader text" +
           "\n\tproducerade av EN cout-sats.\n";
}
```

3.9 Modifiera programmet **`Figure`** (sid 63) till att ”rita”, dvs skriva ut formen av en **`oval`**, en **`triangel`**, en **`rektangel`**, en **`pil`**, en **`romb`**, ett **`kryss`**, en **`stjärna`** och en **`blomma`**, bestående av stjärnor (**`*`**).

Utskriften av programmet som ritar t.ex. en **`triangel`** kan se ut sä hår:



3.10 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 124).

Kapitel 4

Grundbegrepp i Programmering

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 121). 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 64). Strängarna `9` och `9` konkateneras med varandra (sid 64), 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 71) 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 uskriften **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*.

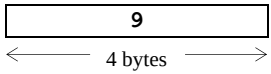
Vad är en variabel?

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 73) – 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`. Man måste alltid skilja mellan *namnet* och *värdet*, Konstanter kan vara namnlösa eller namngivna.

Regler för namngivning av identifierare

Identifierare är beteckningar eller namn för saker och ting i ett program, bl.a. för variabler, namngivna konstanter, funktioner, metoder, klasser, objekt osv. Följande *strikta* regler gäller för namngivning av identifierare. Brott mot dem ger kompileringsfel:

Namn för identifierare måste bestå av endast ETT ord, dvs ett eller flera tecken utan mellanslag, och får endast innehålla

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

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

Observera att bland alla specialtecken endast underscore (_) får användas för att beteckna identifierare. Ett namn med mellanslag tolkas 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 t.ex. är det tillåtet. I andra miljöer och versioner måste frågan undersökas. Namnlösa tecken- och sträng*konstanter* får alltid innehålla svenska specialtecken, eftersom de har ingen identifierare, t.ex. **`"Åsa"`** eller 'ä'.

Vi sammanfattar:

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

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 59). För att göra våra program lättare att läsa, förstå och göra ändringar i, måste namnen vara *beskrivande*. I programet **`Variable`** (sid 77) 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 73). 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.

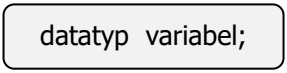
```
// Variable.cpp
// Deklarerar och initierar tre int-variabler, lagrar summan
// av de två första i den tredje och skriver ut resultatet
// Initieringen sker med tilldelningsoperatorn =
#include <iostream>
using namespace std;

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. 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.

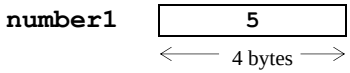
Tilldelningsoperatorn =

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 77) 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. Tilldelningsoperatorn tilldelar som en pil från höger till vänster. Därför får variabeln **`number2`** först värdet **`3`**. Sedan får **`number1`** samma värde, dvs variabeln **`number2`**:s värde som redan är **`3`**. Programmet **`DefInitial`** producerar samma utskrift som **`Variable`** (sid 77).

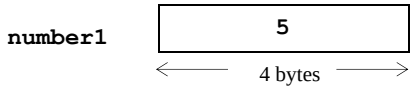
Sammanfattningsvis kan vi säga att C++ strikt följer regeln:

> Variabler som inte *deklareras* innan de används ger kompileringsfel.

Vad händer vid deklaration och initiering?

Vad händer t.ex. i satsen `int number1 = 5;` ?

1. ***Minnesallokering*** Minne reserveras i datorns RAM 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). Så här ser det ut när 4 bytes minne reserveras för variabeln `number1`:



2. ***Tolkning av data*** Datatypen gör att programmet kan tolka innehållet i minnescellen ovan när den fylls med ett värde. Det är 4 x 8 = 32 ettor och nollor som måste tolkas som ett heltal. Olika datatyper har olika algoritmer för omvandling av ettor och nollor till data och omvänt. *Heltalet* `1` t.ex. består av en annan följd av ettor och nollor än *decimaltalet* `1.0`. *Tecknet* `'1'` har en annan digital sekvens, för att inte tala om *strängen* `"1"`. Det är datatypen som möjliggör den korrekta tolkningen.

3. ***Adressering*** har med namngivning att göra. Programmets *logiska* variabelnamn `number1` kopplas till minnescellens *fysiska* adress i RAM. Det görs för att komma åt minnescellen genom att referera till variabelnamnet. Variabler gör minnescellerna i hårdvaran åtkomliga för mjukvaran.

Oinitierade variabler

Vad händer om man deklarerar en variabel men glömmer initieringen? Svaret är miljö- och versionsberoende. I nyare versioner av Visual Studio sätter kompilatorn stopp för oinitierade variabler. Så, i Visual Studio 2022 producerar följande program kompileringsfel. Variabeln `number1` är deklarerad, men inte initierad:

```
// NoInitial.cpp
// En deklarerad men oinitierad variabel ger kompileringfel
#include <iostream>
using namespace std;

int main()
{
   int number1, number2;          // number1 är deklarerad,
   number2 = 2;                   // men inte initierad
   cout << number1 << number2;    // number1 används här
}
```

Felmeddelandet lyder: *uninitialized local variable 'number1' used*. Variabler anses vara väl definierade, först när de är både deklarerade och initierade. Slutsatsen:

> Variabler som inte *initieras* innan deras värde används ger kompileringsfel.

Observera formuleringen *deras värde* och titta på operatorn **`sizeof`** i programmet **`Primitives`** (sid 114) där oinitierade variabler används utan värde.

Äldre versioner av C++

I äldre versioner av C++ och i vissa miljöer kan man både kompilera och exekvera programmet **`NoInitial`**, men får ett ”felaktigt” värde för **`number1`**, beroende på att man använt en oinitierd variabel. Mjukvarumässigt har variabeln **`number1`** inget värde dvs är tom. Men fysiskt, dvs i datorns RAM, är den inte alls tom. Den har kvar något slumpvärde eller ett värde från tidigare användning. Överskriver vi inte det i vårt program får vi det gamla värdet som råkar finnas där. Det är slumpen som styr. Ett sådant värde kallar vi för ”skräp”.

Följande program är ett exempel på att oinitierade variabler producerar skräp, vilket dock endast kan testas i äldre versioner av C++ resp. i vissa miljöer:

```
// NoInitial_Old.cpp
// Skriver ut ett skräpvärde till en oinitierad variabel
#include <iostream>
using namespace std;

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

Körresutatet kan i äldre versioner av C++ resp. i vissa miljöer se ut så här:

```
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.

OBS! Resultatet ovan får du endast i äldre versioner av C++, inte i Visual Studio 2022. Där blir det kompileringsfel: *uninitialized local variable 'number1' used*.

Att äldre versioner av 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

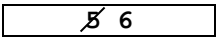
finns *samma* variabel **x** på båda sidor. Översatt till C++ kod blir det:

```
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 75). 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 103).

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 13: 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";
}
```

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 ***4.8 Modulooperatorn %*** (sid 96) 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 60). 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 172). 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.

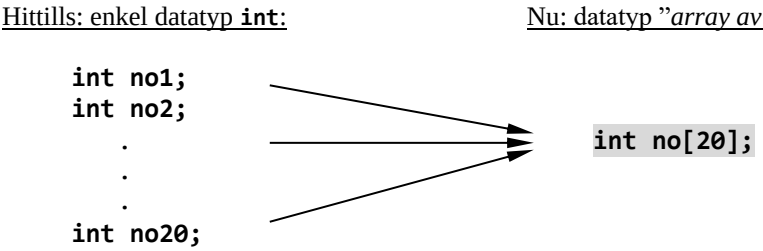
Det följer en liten parentes om den nya räkneoperationen *modulo* som kommer att behövas för att genomföra övningarna 4.9 och 4.10.

4.7 Arrays

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 aldrig bli trött.

Vi ska i detta avsnitt introducera ett verktyg som utnyttjar en av dessa överlägsna egenskaper, nämligen att kunna lagra och hantera *stora datamängder*. Detta verktyg heter *array* och betyder *ordnad uppställning*, en ordnad skara av data. Ibland används i litteraturen begreppet *fält* som är identiskt med *array*.

Man kan t.ex. gruppera 20 variabler av den enkla datatypen **`int`** i en array med 20 element som kan anses som en ny datatyp ”array av **`int`**”:



En *array* är en ordnad mängd av variabler grupperade under ETT namn.

Arrayens delar kallas för *element*. Elementens position kallas för *index*.

Hittills har vi skrivit *20* satser (koden till vänster) för att deklarera 20 **`int`**-variabler. Men nu med array 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*. För att definiera arrayvariabeln **`no`** används den nya datatypen ”*array av* **`int`**” som i kod skrivs så här: **`int[]`** .

Typiskt för arrays är *hakparenteserna* **`[ ]`** (eng.: *brackets*). som används för att lägga till informationen om arrayens *storlek*. Definitionssatsen **`int no[20];`** anger *antalet* element i arrayen i hakparenteserna: **`[20]`**. I andra sammanhang har hakparentesen en annan betydelse. Mer om detta senare.

Arrayvariabeln **`no`** ersätter de 20 vanliga variablerna **`no1`**, **`no2`**, …, **`no20`** och består nu i sin tur av 20 *element*. Varje element är en variabel som kan lagra ett värde. Enda skillnaden är *sättet* dvs *koden* att komma åt dessa värden. Indexet är ett nummer som specificerar varje elements position i arrayen. Varje element i en array kan betraktas som en *indexerad* dvs *numrerad variabel*.

En array är inte längre en enkel utan en s.k. *sammansatt* datatyp. En *enkel datatyp* representerar ETT värde åt gången, t.ex. ett heltal, ett decimaltal, ett tecken, ett san-

ningsvärde osv. En *sammansatt datatyp* representerar fler än ett värde åt gången, t.ex. flera heltal, flera flyttal, flera tecken, flera sanningsvärden osv. Om vi t.ex. grupperar variabler av den enkla datatypen **int** får vi den sammansatta dataypen **int[]** som läses *"array av* **int***"*. Varje element i en sådan array kan betraktas som en indexerad dvs numrerad variabel av typ **int**. Men till skillnad från enkla datatyper initieras alla element i en sammansatta datayp automatiskt till ett s.k. *default*-värde. Hos **int** är default-värdet **0**. Det är anledningen varför vi använder begreppet *definition* istället för deklaration, när vi skriver `int no[20];` .

Åtkomst till arrayens element

Definitionssatsen `int no[20];` reserverar 20 minnesceller för lagring av 20 värden av typ **int**. Låt oss anta att alla element i arrayen **no** tilldelats några värden. Exemplet visar hur indexeringen av element i en array är organiserad. En array lagras alltid i ett sammanhängande minnesområde så att vi får följande bild:

Minnesbild av arrayen no:

25	1257	-10	. . .	358	65	219
Kod: no[0]	no[1]	no[2]	. . .	no[17]	no[18]	no[19]

Den sista raden visar koden som används i C++ för att komma åt arrayens element. Anmärkningsvärt är att hakparenteserna **[ ]** här inte används för att ange antalet utan *indexet*, dvs hakparentesen har här en annan betydelse än i definitionssatsen. Läs mer om detta på sid 91. Dessutom börjar indexeringen med **0**, inte med **1**. Medan vi människor är vana vid att påbörja numreringen av ett antal objekt med **1**, börjar C++ numreringen av elementen i en array med **0**. Följande regel gäller:

> Indexregeln: I arrays börjar numreringen av index alltid med 0.
> Därför gäller: elementets position = index + 1

Med *position* menar vi numret som människan använder för att numrera elementen, medan *index* betecknar numret som C++ använder.

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 kodas med **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. Så, indexen går från **0** till **19**.

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
// Skriver ut värdena med resp. index
// Ingen kontroll av indexgränserna
#include <iostream>
using namespace std;

int main()
{
   int no[3];                        // Definition av en array
   no[0] = 64;                       // Elementvis initiering
   no[1] = 86;
   no[2] = -6;
   cout << "\n\tArrayens 1:a element no[0] har värdet "
        << no[0] << " och index 0\n"
        << "\n\tArrayens 2:a element no[1] har värdet "
        << no[1] << " och index 1\n"
        << "\n\tArrayens 3:e element no[2] har värdet "
        << no[2] << " och index 2\n"
        << "\nIndexen -1 och 3 ligger utanför de definierade "
        << "indexgränserna.\n\n"
        << "no[-1] har värdet " << no[-1] << " med index -1\n"
        << "no[ 3] har värdet " << no[3]  << " med index  4:"
        << " Odefinierade värden!\n";
 }
```

En körning ger:

```
        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:e element no[2] har värdet -6 och index 2

Indexen -1 och 3 ligger utanför de definierade. indexgränserna.

no[-1] har värdet -8589934 med index -1
no[ 3] har värdet -8589934 med index  4: Odefinierade värden!
```

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 kom-

pilera 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 89) 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.

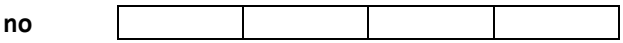
Man kan ju undra varför **no[4]** inte är definierat – som vi hävdar ovan – fast det ”förekommer” i definitionssatsen **int no[4];** Det ser bara ut *som om* det förekommer. Detta beror på att hakparenteserna **[]** inte har samma betydelse i programmets alla satser. Den korrekta tolkningen av **[]** beror på sammanhanget:

Hakparentesernas två olika betydelser

1. **I definitionssatser** omsluter hakparenteserna *antalet* element i arrayen dvs arrayens *storlek*. T.ex. innebär raden

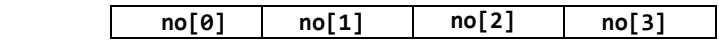
   ```
   int no[4];          // Definition av en array
   ```

   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 89) 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.

Arrayens 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 90) – 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: Att slå ihop definitionen med tilldelningen till en kortform som använder sig av en s.k. *initieringslista*. Detta har vi använt i följande program:

```
// ArrayInit.cpp
// Kortform för definition och initiering av en array i en
// och samma sats med initieringslista
// Elementvis tilldelning av en array
#include <iostream>
using namespace std;

int main()
{
   int no[] = {64, 86, -6};        // Definition och initiering
                                   // med initieringslista
   int copy[3];                    // Endast definition
   copy[0] = no[0];                // Elementvis initiering
   copy[1] = no[1];
   copy[2] = no[2];
// copy = no;                      // Ger kompileringsfel!
   cout << "\n\tcopy:s 1:a element copy[0] = " << copy[0]
        << " med index 0\n"
        << "\n\tcopy:s 2:a element copy[1] = " << copy[1]
        << " med index 1\n"
        << "\n\tcopy:s 3:e element copy[2] = " << copy[2]
        << " med index 2\n";
 }
```

En körning av programexemplet **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:e element copy[2] = -6 med index 2
```

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, -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 göras ”manuellt” dvs varje elementtilldelning i en separat sats – exempel på det hade vi i **ArrayInit**.

4.8 Hantering av slumptal

Datorn kan som en deterministisk maskin inte producera ”äkta” slumptal utan endast *simulera* slumptal, dvs enligt en viss algoritm *beräkna* s.k. *pseudoslumptal*. I fortsättningen menas med slumptal alltid pseudoslumptal. Tillspetsat kan man undra om det överhuvudtaget finns ”äkta” slumptal, när man inte kan få fram dem. I C++ kan man simulera slumptal med den fördefinierade funktionen **rand()**. Programmet **Random** demonstrerar några exempel på olika typer av slumptal i C++:

```
// Random.cpp
// Slumpar tal mellan 0 och 1 med funktionen rand()
// 1 + rand() % 6 slumpar heltal mellan 1 och 6
#include <iostream>
using namespace std;

int main()
{
   srand(time(0));                // Skapar variation i slumpen
   cout << "\n\trand() slumpar heltal mellan 0 och " << RAND_MAX
        << ":\n\n\t" << rand() << "\n\t" << rand() << "\n\t"
        << rand() << "\n";

   cout << "\n\t1 + rand() % 6" << " slumpar heltal\n\t"
        << "mellan 1 och 6 (Tärningskast):\n\n\t"
        << 1 + rand() % 6 << "\n\t"
        << 1 + rand() % 6 << "\n\t" << 1 + rand() % 6 << "\n";
}
```

Funktionen **rand()** slumpar heltal mellan **0** och **32 767** som är den fördefinierade max.-gränsen till datatypen **short** och lagras i konstanterna **SHRT_MAX** (sid 116) och **RAND_MAX**. Mer exakt slumpar **rand()** heltal inom intervallet **[0, 32 767)**, dvs från och med **0** till, men inte med, **32 767**. Matematiskt uttryckt:

$$0 \leq \texttt{rand()} < 32\,767$$

En körning av programmet **Random** ger:

```
        rand() slumpar heltal mellan 0 och 32767:

        19532
        19504
        15319

        1 + rand() % 6 slumpar heltal
        mellan 1 och 6 (Tärningskast):

        6
        4
        5
```

Variation i slumpen

Vad gör anropet **srand(time(0))** i programmet **Random** strax i början av **main()**? Algoritmen som används i **rand()** är *rekursiv*, dvs beräknar ett slumptal baserat på förra slumptal, vilket dock kräver ett startvärde. Samma startvärde producerar alltid samma sekvens av slumptal. Därför kommer det inte bli någon variation vid olika körningar om man inte varierar startvärdena. För att garantera denna variation bestäms ett startvärde genom anrop av en annan funktion som heter **srand()**. Men hur åstadkommer **srand()** variation? Det enda i universum som är absolut varierande på ett förutsägbart sätt, är tiden. Därför tar funktionen **srand()** datorns tid via en tredje funktion som heter **time()** och returnerar datorns aktuella datum och tid, omräknat till antal sekunder. Eftersom datorn vid varje körning har en annan tid garanterar anropet **srand(time(0))** att varje sekvens av slumptal som genereras av funktionen **rand()**, har ett annat startvärde.

Slumptal inom ett intervall

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**. 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)
```

Om modulooperatorn **%** läs i nästa avsnitt.

4.9 Modulooperatorn %

Symbolen **%** har i C++ ingenting med procenträkning att göra utan står för ett nytt räknesätt som kallas för *modulo*, besläktat med heltalsdivision. Modulo är en heltalsoperation. Man dividerar två heltal, tar resten och ignorerar resultatet. T.ex.:

9 dividerat med **2** ger **4**, rest **1**. Därför: **9** *modulo* **2** ger **1**.
Modulooperationen ignorerar **4** och tar resten **1**.
Man skriver: **9 % 2 = 1**.

Ett annat exempel på modulo är:

9 % 2 = 1, alltså är **9** ett udda tal. Däremot är **8** ett jämnt tal därför att **8 % 2 = 0**, eftersom **8** dividerat med **2** ger resultatet **4** och resten **0**. Vid heltalsdivision med **2** ger alla jämna tal resten **0**, medan alla udda tal ger resten **1**.

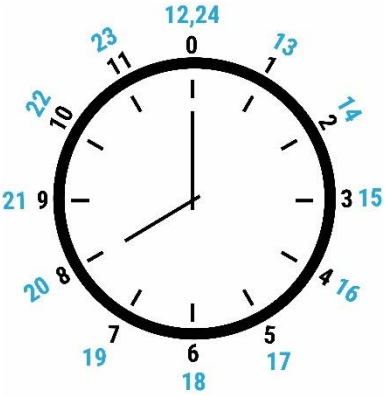
Tillämpningar av modulo

1. ***Klockan*** ”räknar” modulo 12.

   Klockan räknar i talsystemet med basen 12, dvs med siffrorna 0-11.

   T.ex.:

   ```
   15 % 12 = 3
   18 % 12 = 6
   20 % 12 = 8
   24 % 12 = 12 % 12 = 0
   ```



2. ***Veckodag***

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

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

   (5 + 11) % 7 = 2

   Dvs veckodagen i frågan är **2**:a veckodagen, nämligen **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 att räkna med – till *veckodagarnas system* dvs till *talsystemet med basen* **7** som använder sig av siffrorna **0-6**.

I övn 4.7 (sid 108) kan du skriva ett C++ program för detta problem.

3. ***Tärningskast***

Den i C++ fördefinierde funktionen **`rand()`** slumpar heltal mellan **`0`** och **`32 767`**. För att skräddarsy **`rand()`** för att få slumptal mellan **`1`** och **`6`**, t.ex. för simulering av tärningskast, utförs en *skalning* med **`% 6`** och en *skiftning* med **`1`**:

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

Modulooperatorn drar av alla multiplar av **`6`** från **`rand()`** så att endast en rest mindre än **`6`** blir kvar. På så sätt ger *skalningen* **`rand() % 6`** slumptal mellan **`0`** och **`5`**. Sedan får man med *skiftningen* av intervallet **`[0, 5]`** med **`1 +`** slumptal mellan **`1`** och **`6`** (tärningskast), se programet **`Random`** (sid 94).

4. ***Jämnt eller udda***

Med följande villkor kan man avgöra att talet **`no`** är jämnt:

```
no % 2 == 0
```

Delar man **`no`** med **`2`** och resten är **`0`**, så är **`no`** jämnt delbart med **`2`** och därmed jämnt. Om villkoret är falskt är talet **`no`** udda, se programet **`SimleIf`** (sid 130).

5. ***Talsystem***

En annan tillämpning av modulo är omvandling mellan olika talsystem, t.ex. mellan det decimala och binära talsystemet. Generellt är modulo nyckeloperationen vid omvandling mellan olika system, se ***Klockan*** och ***Veckodag***.

6. ***Euklid***

I matematiken används modulo bl.a. för att bestämma den största gemensamma delaren av två heltal (Euklides algoritm).

Att modulo ger resten vid heltalsdivision kan även uppfattas 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 drar av alla multipler av **`a`** och behåller resten: **`33`** *modulo* **`6`** ger **`3`**, därför att man får **`3`**, när man drar av **`5`** gånger **`6`**, dvs **`30`**, från **`33`**.

4.10 Bestämning av max/min

Nu ska vi lösa problemet att bestämma det största (minsta) värdet bland tre givna tal. Vi kommer att ha användning av lösningen bl.a. i *inlämningsuppgiften 1 (sid 111)*. På köpet lär vi känna programmeringens enkaste kontrollstruktur, den s.k. **if**-satsen, som vi kommer att fördjupa oss i senare (kap 6). Även funktionsbegreppet kommer att behandlas i detalj senare (sid 171), och lära oss att själva definiera *funktioner* i C++. Programmet **Max** tar första steget i lösningen av detta problem:

```
// Max.cpp
// Läser in tre tal och bestämmer det största bland dem
#include <iostream>
using namespace std;

int main()
{
   int no1, no2, no3, max;
   cout << "\n\tMata in ett heltal:\t\t";
   cin >> no1;
   cout << "\n\tMata in ett heltal till:\t";
   cin >> no2;
   cout << "\n\tMata in ett tredje heltal:\t";
   cin >> no3;

   max = no1;            // Vi antar att no1 är störst
   if (no2 > max)
      max = no2;         // Byter till no2 om no2 är större
   if (no3 > max)
      max = no3;         // Byter till no3 om no3 är större

   cout << "\n\t" << max << " är det största talet bland talen "
        << no1 << ", " << no2 << " och " << no3 << ".\n";
 }
```

En testkörning ger: Testa gärna för flera inmatningar.

```
       Mata in ett heltal:            12

       Mata in ett heltal till:       45

       Mata in ett tredje heltal:     23

       45 är det största talet bland talen 12, 45 och 23.
```

Själva algoritmen att hitta det största bland tre tal, är inramad i programmet ovan och innehåller två enkla **if**-satser: I första satsen av denna algoritm antar vi att **no1** är det största talet. **max**-”rollen” tillelas varaiabeln **no1**. Det behöver inte stämma. Den första **if**-satsen testar detta antagande genom att kolla om **no2** är större än

max och därmed även större än **no1**. Om det är fallet byts **max**-”rollen” från **no1** till **no2**. Samma sak görs i den andra **if**-satsen med **no3**. Slutligen kommer **max**-”rollen” vara hos det tal som är störst av alla tre.

Avgörande för enkelheten av denna algoritm är att endast *två* av tre värden samtidigt jämförs med varandra, dvs två i taget och inte alla tre på en gång. Detta görs i de två **if**-satserna, dvs i två steg efter varandra. **if**-satsen kommer att behandlas i detalj senare (sid 130).

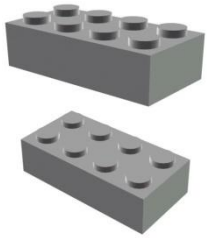
Resten av programmet **Max** är inläsning och utskrift. Vi kommer i fortsättningen att separera dessa delar från den inramade algoritmen för att lära oss om funktioner, se programmet **MaxFct** på nästa sida.

Min

För att få det *minsta* talet bland tre inmatade behövs en mycket liten modifiering av programmet **Max**. Man behöver bara byta ut i **if**-satsernas villkor den s.k. *jämförelseoperatorn* **>** mot **<**. Självklart borde man, för att följa god programmeringsstil, även byta ut variabelnamnet **max** mot **min** och ändra texten i utskriftssatsen. Om jämförelseoperatorer kan man läsa mer på sid 132.

Anledningen till att algoritmen för bestämning av max/min är inramad i programmet **Max**, är att vi nu vill isolera denna algoritm för att skriva den som en funktion. Processen kallas för *modularisering* (tas upp utförligt på sid 171).

Modularisering i två steg



Modularisering innebär att bryta ner ett problem i mindre, återanvändbara delar, s.k. *moduler*, jämförbart med Legobitar. I C++ kallas modulerna för *funktioner*.

Programmet **Max** (sid 98) löser problemet att bestämma det största talet bland tre givna tal. Men detta problem kan även förekomma i andra sammanhang. Och då vill man helst använda den redan befintliga algoritmens kod, utan att behöva återuppfinna hjulet. På så sätt höjs effektiviteten i programutvecklingsarbetet. Genom att isolera den inramade koden och skriva den som en separat funktion, kan vilket program som helst använda den.

Ett exempel på ett sådant behov är vår inlämningsuppgift 1 (sid 111). Där ska man bestämma den största bland tre tävlandes totalpoäng. Som ledning anges i uppgiften att man ska använda funktionen **max()**. Denna funktion ska vi skriva nu genom att definiera den inraade koden som en funktion, binda in funktionen i ett program och anropa den därifrån. Det nya programmet **MaxFct** ska åstadkomma samma sak som programmet **Max**. På så sätt modulariserar vi programmet **Max**. Samtidigt blir funktionen **max()** vår första *egendefinierade* funktion i C++. Hittills hade vi endast *anropat* redan fördefinierade funktioner.

Vi genomför modulariseringprocessen i två steg: först skriver vi den inramade koden som en funktion, kallad **max()**, men behåller den i samma fil. Sedan bryter vi ut funktionen **max()** från filen och placerar den i en extern fil. Nedan följer:

Steg 1: En fil

```
// MaxFct.cpp
// Definierar och anropar funktionen max() som bestämmer
// det största bland tre tal
#include <iostream>
using namespace std;

int max(int a, int b, int c)  // Funktionen max() definieras
{
   int tmp = a;               // Antar att a är störst
   if (b > tmp)
      tmp = b;                // Byter till b om b är större
   if (c > tmp)
      tmp = c;                // Byter till c om c är större
   return tmp;                // Returnerar tmp till max()
}

int main()                    // Huvudprogrammet
{
   int no1, no2, no3, noMax;
   cout << "\n\tMata in ett heltal:\t\t";
   cin >> no1;
   cout << "\n\tMata in ett heltal till:\t";
   cin >> no2;
   cout << "\n\tMata in ett tredje heltal:\t";
   cin >> no3;
   noMax = max(no1, no2, no3);   // Funktionen max() anropas
   cout << "\n\t" << noMax << " är det största talet bland "
        << no1 << ", " << no2 << " och " << no3 << ".\n";
}
```

Prorammet **MaxFct** producerar samma utskrift som programmet **Max** (sid 98).

Steg 2: Två filer

Nu klipper vi ut funktionen **max()** från filen **MaxFct.cpp** och placerar den i en extern fil. Vi öppnar en ny textfil, klistrar in funktionen **max()** i den nya filen och döper den till **Max.h**. I ytterligare en ny fil placerar vi resten av koden i **MaxFct.cpp** (med en rad tillägg) och döper den nya filen till **MaxExt.cpp**. Vi ser till att filerna **Max.h** och **MaxExt.cpp** ligger i samma mapp. Nu har vi *två* filer som utgör *ett* C++ program. I Visual Studio laddar vi båda i samma projekt. Se till att du vid denna omvandling inte skriver över eller tappar den gamla filen **MaxFct.cpp**. Nedan ser du de två nya filerna, först filen **Max.h** som innehåller funktionen, sedan filen **MaxExt.cpp** som innehåller huvudprogrammet.

Funktionen max()

```
// Max.h
// Definierar funktionen max() som extern modul
// Filen lagras separat som headerfil (*.h)
// Kan användas av vilket program som helst

int max(int a, int b, int c)  // Funktionen max() definieras
{
   int tmp = a;               // Antar att a är störst
   if (b > tmp)
      tmp = b;                // Byter till b om b är större
   if (c > tmp)
      tmp = c;                // Byter till c om c är större
   return tmp;                // Returnerar tmp till max()
}
```

För första gången har vi en fil som inte har filändelsen **cpp**. Orsaken är att denna fil inte utgör ett program, utan är en *funktion*, närmare bestämt endast definitionen till en sådan. Den kan kompileras, men inte exekveras, i alla fall inte ensam. Det finns ju ingen exekveringsunkt i den, nämligen funktinen **main()**. Den nya filtypen som vi skapat här har filändelsen **h** och kallas för *headerfil*.

Huvudprogrammet

```
// MaxExt.cpp
// Anropar funktionen max() som extern modul
// Inkluderar headerfilen Max.h som lagras separat och
// innehåller definitionen till funktionen max()
#include <iostream>
using namespace std;

#include "Max.h"       // Inkluderar filen Max.h som lagras i
                       // samma mapp som programfilen MaxExt.cpp
int main()
{
   int no1, no2, no3, noMax;
   cout << "\n\tMata in ett heltal:\t\t";
   cin >> no1;
   cout << "\n\tMata in ett heltal till:\t";
   cin >> no2;
   cout << "\n\tMata in ett tredje heltal:\t";
   cin >> no3;
   noMax = max(no1, no2, no3);    // Funktionen max() anropas
   cout << "\n\t" << noMax << " är det största talet bland "
        << no1 << ", " << no2 << " och " << no3 << ".\n";}
}
```

Prorammet **MaxExt** producerar samma utskrift som programmet **Max** (sid 98).

Efter att ha avlägsnat funktionen **max()** från filen **MaxFct.cpp**, placerar vi resten i en ny fil som vi döper till **MaxExt.cpp**. Men, för att koppla ihop det nya programmet med funktionen **max()**, inkluderar vi headerfilen **Max.h** som innehåller denna funktions definition, i den nya filen, med följande sats, för att kunna anropa den:

```
#include "Max.h"
```

Observera hur syntaxen i denna sats skiljer sig från den vid inkluderingen av biblioteksfiler: För det fösta skrivs filnamnet *inte* inom **< >** utan som en sträng inom citationstecken **" "**.

Förutsättningen för att denna inkludering ska fungera är att båda filerna **Max.h** och **MaxExt.cpp** ligger i samma mapp. Nu har vi *två filer* som utgör *ett C++ program*:

I Visual Studio räcker det att infoga **MaxExt.cpp** i ett projekt, kompilera och köra, om båda filerna **Max.h** och **MaxExt.cpp** finns i samma mapp på hårddisken.

Headerfiler i C++

Det år inte obligatoriskt att förse filen **Max.h** med filändelsen **h**. Däremot är det en konvention bland C++ utvecklare att göra så, när man vill lagra en eller flera funktioner i en extern fil och inkludera filen i sitt huvudprogram. En sådan fil kallas för *headerfil* eftersom den innehåller kod som i regel placeras i *huvudet* på programfilerna med filändelsen **cpp**. Ett exempel på det är programmet **MaxFct** (sid 98) där funktionen **max()** är placerad i början av filen, före **main()**. Dessutom kommer vi att se att t.ex. i nästa program headerfilen **Max.h** inkluderats i *huvudet* på programfilen **MaxExt.cpp**. Så brukar man göra med alla headerfiler.

Om funktioner

Funktionen **max()** ligger i filen **Max.h** (sid 101). Parentesen **(int a, int b, int c)** kallas för *parameterlistan*. **a, b** och **c** är funktionens *formella parametrar*, medan **no1, no2** och **no3** som står i funktionsanropet istället för dem, kallas för *aktuella parametrar*. Vid anropet kopieras de inlästa värdena från de aktuella till de formella parametrarna. På så sätt hamnar värdena i funktionen, där deras **max** bestäms. Första raden inkl. parameterlistan, dvs **int max(int a, int b, int c)** utgör funktionens *huvud*.

Efter huvudet står funktionens *kropp* inom måsvingar. Kroppen avslutas med en s.k. **return**-sats som med hjälp av variabeln **tmp** returnerar det största värdet till namnet **max()**. På så sätt hamnar funktionens *returvärde* i programmet, när funktionen anropas. Eftersom namnet **max()** bär returvärdet måste anropet inbakas i en tilldelningssats, så att variabeln **noMax** i **main()** kan ta emot detta värde som slutligen skrivs ut. Att funktionen **max()** innehåller en **return**-sats ger upphov till att kalla **max()** för en *funktion med returvärde*. Det finns i C++ även *funktioner utan returvärde*. Dessa saknar **return**-sats. Funktioner tas upp utförligt i kap 7 (sid 169).

4.11 Ökningsoperatorn ++

Denna operator som gett namnet till språket C++, nämndes redan i avsnittet om överskrivning (sid 82). Det finns två varianter av ökningsoperatorn, eng. *increment operator*: Man kan skriva den *efter*, så här **a++**, eller *före* operanden, så här **++a**. Sätts den *efter* operanden talar man om ökningsoperatorns *postfix*variant. Skrivs den *före* operanden blir det *prefix*varianten. Följande program demonstrerar skillnaden mellan båda dessa varianter:

```
// 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;  och sedan  a = a + 1;
   cout << "\n\t a=0: Efter b=a++; blir b = " << b
        << " och a = " << a << '\n';

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

Prefixvarianten **++a;** betyder:

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

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.

Även satsen `++a;` gör samma sak som `a = a + 1;` Skillnaden med postfixvarianten 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!

Minskningsoperatorn `--` 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.

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 snabbare och effektivare än maskinkod skapad av tilldelningsoperatorn.

4.12 Sammansatt tilldelning

Ökningsoperatorn som behandlades i förra avsnitt, består egentligen av *två* operatorer varav en är tilldelning, fast osynlig. Ökningsoperatorn **++** består av ökning med **1** *och* tilldelning, minskningsoperatorn **--** av minskning med 1 *och* tilldelning som dock inte visas i symbolen. De används så här: **a++** eller **a--**, där **a** är en operand.

Är man inte nöjd med ökning (minskning) med **1** utan vill göra ändringen med vilket värde som helst, alltså variabel, kommer man till en mer generell grupp operatorer som kallas för *sammansatta tilldelningar*, eng. *compound assignments*:

Operatorerna **+=** **–=** ***=** **/=** **%=**

De heter så därför att de är sammansatta av *två* operatorer, varav en är tilldelning som förekommer i alla, är alltså till skillnad från ökningsoperatorn *synlig*. De sammansatta tilldelningarna har precis som de vanliga räkneoperatorerna **+**, **-**, *****, **/**, **%** *två operander* och används t.ex. så här: **sum += a**, där **sum** och **a** är operander.

Om vi tar den första av de sammansatta tilldelningarna **+=** så är den sammansatt av addition och tilldelning. T.ex.:

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

Dvs: Addera först **sum** med **a** och tilldela sedan resultatet till **sum**.

Beräkningen utförs helt enkelt i den ordningen man läser: från vänster till höger. När vi adderar först **sum** med **a** och tilldelar resultatet sedan till **sum**, överskrivs variabeln **sum**:s värde direkt.

I specialfallet **a = 1** gör **sum += a;** samma sak som **sum++;** vilket visar att ökningsoperatorn **++** är ett specialfall av **+=**. På liknande sätt fungerar de andra sammansatta tilldelningsoperatorerna: **-=**, ***=**, **/=** och **%=**. De utför *först* en räkneoperation och *sedan* en tilldelning.

Observera att alla sammansatta operatorer (precis som **++** och **--**) skrivs *utan mellanslag*. Med mellanslag känns de inte igen av kompilatorn och tappar sin betydelse.

Programmet **CompAssign** (nedan) demonstrerar alla sammansatta tilldelningsoperatorer. Man matar in ett värde till variabeln **a** som kombineras via **+=**, **-=**, ***=**, **/=** och **%=** med de redan initierade variablerna **sum**, **diff**, **prod**, **div** och **mod**. Dessa namn utför förstås inga räkneoperationer, utan är bara valda för att vara beskrivande. Deras initiering är avgörande, annars kommer odefinierade skräpvärden in i beräkningen och förstör resultatet (sid 80).

```
// CompAssign.cpp
// Testar sammansatt tilldelning (Compound assignment)
// +=, -=, *=, /= och %= räknar först och tilldelar sedan
// Konkatenering av strängvariabler
#include <iostream>
using namespace std;

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 s
   cout <<'\n' <<               // och t är string-variabler
   "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*. 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.

Konkatenering av strängvariabler med `+`

Som det redan nämndes tidigare (sid 64) finns i C++ möjligheten att använda operatorn **+** för att konkatenera strängar, vilket dock endast gäller för sträng*variabler*. Operatorn **+** *överlagrar* den vanliga artitmetiska additionsoperatorn och 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;
```

Man får då utskriften **Slut på kapitel 4**, därför att **+** tolkas här inte som addition utan konkatenering. Det i sin tur, därför att det till vänster och höger står en variabel av typ **string**.

Dessvärre kan – vilket vi redan nämnde på sid 64 – konkateneringsoperatorn **+** inte direkt användas på sträng*konstanter*. T.ex. skulle **cout << "Slut på" + " kapitel 3";** ge kompileringsfel. 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**. 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 77) så här: Deklarera ytterligare variabler, säg **`diff`** och **`prod`**, tilldela till dem uttryck bildade med räknesätten **`-`** och **`*`**. Skriv ut resultaten med meningsfulla utskrifter.

4.3 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.

4.4 Ersätt i programmet **`DefInitial`** (sid 79) 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.

4.5 Modifiera programmet **`OverWrite`** (sid 82) 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?

4.6 Vidareutveckla din lösning till övn 4.4 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.

4.7 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 – kalla det **`Year2Days`** – 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 som visas i programmet **`Hour2Sec`** (sid 86).

4.9 Ersätt i programmet **`Increment`** (sid 103) de satser som använder ökningsoperatorn med satser som inte använder ökningsoperatorn. I övrigt ska programmet producera samma utskrift som på sid 103. Kompilera och kör.

4.10 Ersätt i programmet **`CompAssign`** (sid 106) de satser som använder sammansatt tilldelning med satser som använder vanlig tilldelning. I övrigt ska programmet producera samma utskrift som på sid 106. Kompilera och kör.

4.11 ***Days2Year (Projekt)*** Programmet **`Year2Days`** i övn 4.8 (sid 109) omvandlar tiden i antal år, månader och veckor till antal dagar.

Vi ska nu i ett nytt program **`Days2Year`** vända om problemet: Vi vill omvandla ett antal dagar, som vi läser in, till antal år, månader, veckor och resterande dagar. 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.
Operationen *Ta resten vid heltalsdivision* är modulo.
Läs om modulo på sid 96.

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

Utveckla ett C++ program för ovanstående algoritm och pseudokod.

4.12 Tillämpa den logiska strukturen i algoritmen och pseudokoden till övn 4.11 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.

Projektuppgift

Gymnastiktävling Skriv ett C++ program som avgör en tävling i gymnastik. Tre tävlande deltar i tävlingen. De får sina poäng av tre olika domare. Poängen ska ligga mellan **1** och **100**. Varje tävlandes poäng ska summeras till en totalpoäng. Sedan ska programmet bestämma den största totalpoängen samt ska skriva ut både varje tävlandes totalpoäng, den största totalpoängen och tävlingens vinnare.

Använd tre arrays. Varje array ska lagra poängen för varje tävlande. Varje element i arrayen ska tilldelas en domares poäng. Simulera domarnas poänggivning med slumptal inom intervallet **[1, 100]**. Bestäm den största totalpoängen bland de tre tävlandes totalpoäng.

Frivilligt: I fall att två tävlande får samma antal totalpoäng, utropa båda som tävlingens vinnare.

Ledning:

Steg 1 Läs i kursboken, avsn. *4.7 Arrays* (sid 88).

Steg 2 Läs om slumptal i kursboken, avsn. *4.9 Hantering av slumptal* (sid 94), speciellt om *Slumptal inom ett intervall* (sid 95).

Steg 3 Skapa tre arrays, en för varje tävlandes poäng. Varje array ska innehålla 3 element.

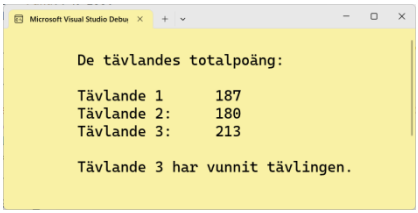
Steg 4 Fyll varje array med slumpvärden mellan 1 och 100 motvarande de tre domarnas poäng.

Steg 5 Skapa tre variabler för totalpoängen, en för varje tävlande, och tilldela dem summan av varje tävlandes poäng.

Steg 6 Bestäm den största bland de tre tävlandes totalpoäng. Använd funktionen **`max()`** som behandlas i kursboken (sid 101).

OBS! För att bestämma tävlingens vinnare, borde du lägga till lite kod till funktionen **`max()`**, t.ex. i form av en **`string`**-variabel **`winner`**, så att den inte bara hittar den största totalpoängen (**`max`**) utan även den tävlande (**`winner`**) som fått denna totalpoäng.

Steg 7 Skriv ut varje tävlandes totalpoäng och gymnastiktävlingens vinnare. Ex.:



Kapitel 5

Enkla

datatyper

5.1 De enkla datatyperna och deras gränser

Olika kategorier av datatyper är: *enkla*, *sammansatta* och *objekt*. Array är en underkategori av sammansatta datatyper. **`int`** är ett exempel på en *enkel*, **`string`** på en *sammansatt* datatyp. Här ska vi ta upp de *enkla datatyperna* (eng. *primitive types*).

Vad är en enkel datatyp?

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

I C++ har vi följande enkla datatyper: **`bool`**, **`char`**, **`short`**, **`int`**, **`long`**, **`float`**, **`double`** och **`long double`**. Alla de är *enkla* därför att de representerar endast *ett* värde. Mer invecklade datatyper som *objekt* eller *sammansatta datatyper* som t.ex. arrays, kan lagra *flera* värden. I definitionen på *datatyp* (sid 73) sa vi bl.a.:

> En datatyp är en föreskrift om
> ...
> hur mycket minne denna typ av data tar och därmed
> hur stora värden den kan lagra (det tillåtna värdeområdet)
> ...

I detta avsnitt ska vi härleda värdeområdet för de enkla datatyperna. Oavsett datorernas prestanda och teknikens snabba utveckling finns det alltid ett *begränsat* utrymme för lagring av data. Därför har man i alla programmeringsspråk fasta värdeområden för de fördefinierade datatyperna, för att effektivisera minneshanteringen. Nedan listas upp alla enkla datatyper i C++ med sina resp. minnesstorlekar:

```
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.
```

I utskriften ovan anges de enkla datatypernas fastlagda minnesstorlekar i antal bytes. 1 *byte* består av 8 bitar där 1 *bit* är den minnesatom som kan lagra endast en nolla eller en etta. Dessa minnesstorlekar är förbestämda i datatypernas definition.

Programmet **Primitives** som producerar utskriften ovan, visar hur minnesstorlekarna mäts med operatorn **sizeof**.

```
// 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";
}
```

Dataypen **bool** representerar sanningsvärdena **true** och **false**. **char** lagrar tecken. **short**, **int** och **long** är enkla datatyper för representation av heltal. Datatyperna **float**, **double** och **long double** representerar decimaltal. Alla enkla datatyper är reserverade ord (sid 32). Som man ser i utskriften ovan har de enkla datatyperna ordnats efter det minnesutrymme som är förbestämt i deras definition.

Det är anmärkningsvärt att variablerna **a** och **b** är deklarerade, men inte initierade i programmet **Primitives**. Ändå kan vi kompilera och exekvera, eftersom vi inte använder deras *värde*. Vi bildar t.o.m. uttrycket **a + b** och skickar det som operand till **sizeof()** för att mäta uttryckets minnesstorlek. Variablerna **a** och **b** är båda **int**, därför blir även **a + b** en **int** som tar **4** bytes. Jämför med regeln på sid 81.

Operatorn `sizeof`

sizeof är ett reserverat ord i C++ och samtidigt en operator. T.ex. mäter **sizeof(int)** storleken av minnesutrymmet i antal bytes som är definierad för datatypen **int**. *Operand* är det som skrivs i parentesen: **sizeof(int)** har operanden **int**. Men **sizeof()** kan även ha variabler eller uttryck som operander. Generellt:

sizeof(*datatyp***)**
sizeof(*variabel***)**
sizeof(*uttryck***)**

Det finns även möjligheten att skicka konstanter som operander till **sizeof()** för att mäta deras minnesstorlek.

Begreppet *operator* används här som synonym till funktion (eller metod), vilket kan igenkännas vid de parenteser som följer när man skriver t.ex. **sizeof(int)**, se programmet **Primitives**. När *operator* används istället för *funktion* brukar man utelämna parenteserna.

Gränserna

De enkla datatypernas gränser, det tillåtna värdeområdet, 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 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 lika stort minnesutrymme till förfogande, därför att de inte behöver lagra tecknet och därmed har 1 bit mer för att lagra själva positiva heltalet.

I programmet **LimitsInt** hämtas de enkla datatypernas gränser från ett antal konstanter som är lagrade i klasser som definierar de enkla datatyperna i C++. Vi låter helt enkelt datorn göra jobbet genom att använda dessa konstanter. De är förstås härledda från de minnesstorlekar som tidigare visades som utskrift av programmet **Primitives** (sid 113).

```
// 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";
}
```

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, vilket kan förorsaka teckenbyte.

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*, 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 75).

`unsigned` - *typerna*

Till skillnad från de datatyperna ovan som kan anta både positiva och negativa värden kan de *teckenlösa* 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 tecken. Detta innebär att nödvändigheten att lagra tecknet 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 tecknet 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 86). 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 tabell:

	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 120) 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 = newline 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 57. 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 119) 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.

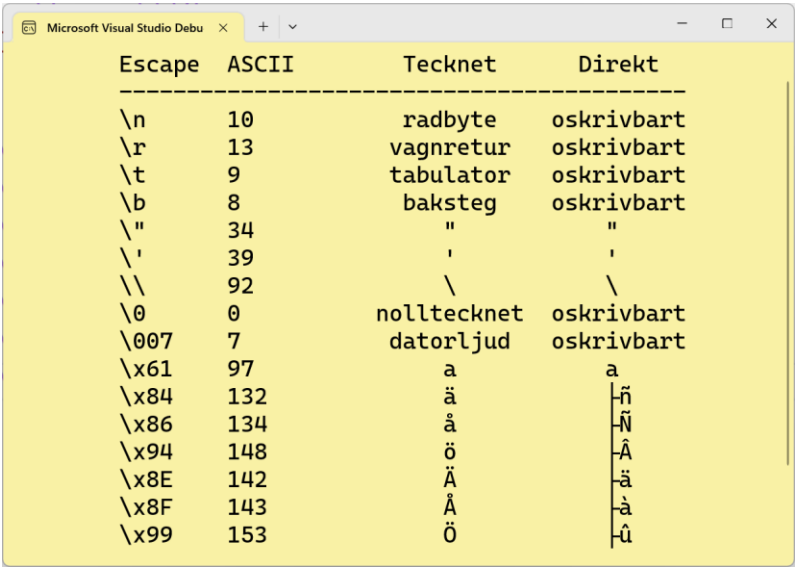
5.3 Experimentera med programmet **`Int2char`** (sid 120) för att få reda på ASCII-koden till datorns ljudsignal. Skriv ett program som genererar ljudsignal.

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, 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 ***EscapeTab (Projekt)*** Skriv ett program som skriver ut följande tabell:




```
Escape  ASCII        Tecknet      Direkt
-----------------------------------------
\n      10            radbyte    oskrivbart
\r      13           vagnretur   oskrivbart
\t      9            tabulator   oskrivbart
\b      8             baksteg    oskrivbart
\"      34               "           "
\'      39               '           '
\\      92               \           \
\0      0           nolltecknet  oskrivbart
\007    7            datorljud   oskrivbart
\x61    97               a           a
\x84    132              ä           ├ñ
\x86    134              å           ├Ñ
\x94    148              ö           ├Â
\x8E    142              Ä           ├ä
\x8F    143              Å           ├à
\x99    153              Ö           ├û
```

Tabellen har fyra kolumner över några utvalda escapesekvenser.

- Tabellens första kolumn visar escapesekvensen i C++ kod.
- Den andra kolumnen visar ASCII-koden till det tecken som representeras av resp. escapesekvens.
- I den tredje kolumnen står själva tecknet, om det är skrivbart.
  Om tecknet är oskrivbart står tecknets beteckning (betydelse) där.
- Den fjärde kolumnen visar hur de skrivbara tecknen skrivs ut i konsolen, om man skriver ut dem *direkt*, dvs utan escapesekvens.

  Om de är skrivbara, men skrivs ut felaktigt, står där det felaktiga tecknet.

  Om de är oskrivbara står där **`oskrivbart`**.

Även de svenska tecknen **`ä`**, **`å`**, **`ö`**, **`Ä`**, **`Å`** och **`Ö`** är med i tabellen samt någon vanlig bokstav, säg **`a`**.

Med hjälp av programmen **`Char2int`** (sid 119), **`Int2char`** (sid 120) och **`Escape`** (sid 124) kan du kolla tecknens ASCII-koder resp. escapesekvenser.

Anmärkningar

`vagnretur` (i tabellens tredje kolumn) betyder **`Return`** eller **`Enter`** *utan* radbyte, dvs förflyttning av skrivmarkören till början av *samma* rad.

`nolltecknet` har i C++ olika betydelser, t.ex. som strängavslutningstecknet i arrays av **`char`**. I andra sammanhang har det andra funktionaliteter (sid 209).

Kapitel 6

Kontrollstrukturer

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.

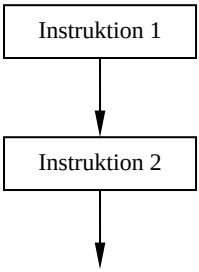
Kontrollstrukturer består av tre grundläggande typer:

- **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 programexempel hittills består endast av sekvenser. Däremot kan varje instruktion i sin tur innehålla andra kontrollstrukturer. Så även om sekvensen är en enkel struktur, kan *nästlade* sammansättningar av den, dvs instruktioner som i sin tur består av underinstruktioner, 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.

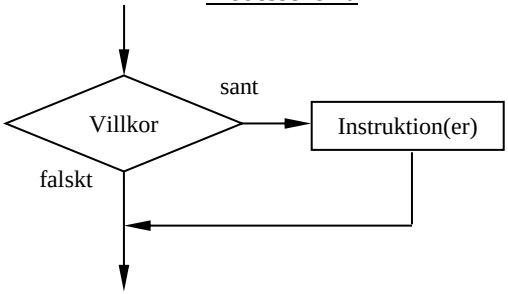
6.2 Enkel selektion: `if`-satsen

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)

Flödesschema



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.

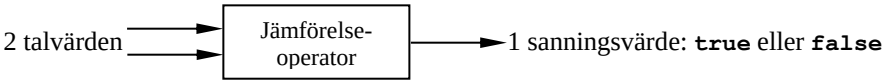
Villkor

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 83 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**

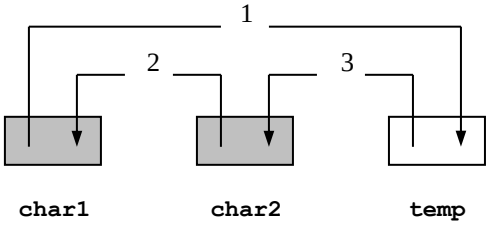
Precis som **!=** är även dubbeltecknet == (utan mellanslag) en jämförelseoperator, men står för *lika med*. Dvs värdet i variabeln **number2** jämförs med **0**. Finns likhet mellan dem är **if**-satsens villkor sant, annars är villkoret falskt. Observera skillnaden mellan likhet som kodas med *två* 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 130) 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.

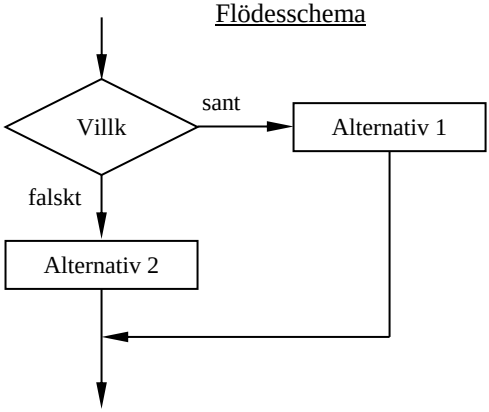
I körexemplet ovan jämför **if**-satsens villkor **char1 > char2** värdena **Z** och **A** med varandra. Men tecken kan inte sättas i en relation av typ ”större än” till varandra. I själva verket är det Unicode-koderna till **Z** och **A** som jämförs med varandra. Det är endast tal som kan jämföras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

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



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

Endast ett av de två alternativen kommer att utföras. När villkorets sanningsvärdena sant och falskt utesluter varandra, utesluter även de båda alternativen varandra. Därför går flödet (pilen) efter alternativ 1 till flödet *efter* alternativ 2. Det vore logiskt fel att leda pilen till ett ställe *före* alternativ 2. Generellt är tvåvägsvalet **`OM-ANNARS`** logiskt uteslutande, dvs alternativen under **`OM`** resp. **`ANNARS`** utesluter varandra och kan inte inträffa båda.

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 96).

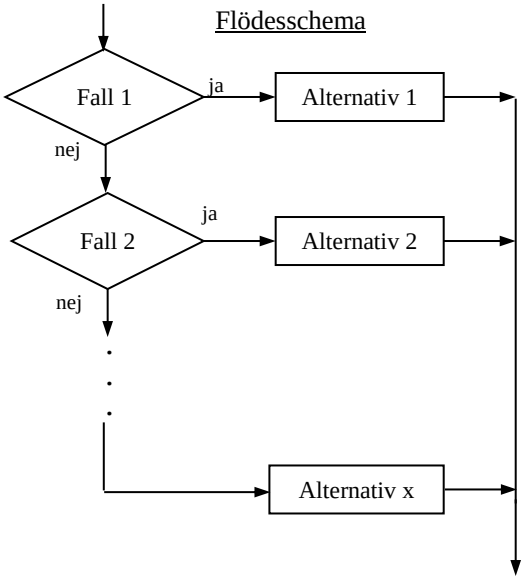
6.4 Flervägsval

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

Flödesschema



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* flödesschemat. 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) flödesschemat 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 flödesschema visades på sid 137 även kodas med **`switch`**-satsen. Den generella strukturen till **`switch`**-satsen kan beskrivas så här:

```
switch (expression)
{
    case constant1 :
       sats(er)1;
       break;
    case constant2 :
       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 *expression* i huvudet menas ett aritmetiskt uttryck (sid 86) 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 (expression == constant1)
if (expression == constant2)
                .
                .
                .
```

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 141) 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 148). **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**

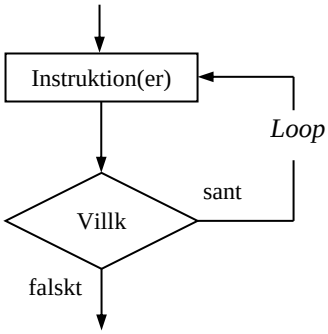
Efter-testad repetition

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 flödesschema:

Pseudokod

REPETERA
instruktion(er)
SÅ LÄNGE *villkor* uppfyllt

Flödesschema



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 klamrarna **`{`** och **`}`** utelämnas. Till skillnad från **`if`**-satsen kan här kroppens avslutande klammer inte ersätta **`do`**-satsens avslutande semikolon, då **`do`**-satsen inte

[*] I några böcker kallas det för *iteration*. Vi undviker denna term eftersom den används som fackterm i andra sammanhang, t.ex. i numerisk analys.

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. **do**-satsen utan avslutande semikolon ger kompileringsfel.

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 (slin-

gan). 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!
```

TILLS vs. SÅ LÄNGE

I pseudokoden till **do**-satsen (sid 142) är *villkoret* inbyggt i en **SÅ LÄNGE**-sats:

REPETERA
    instruktion(er)
SÅ LÄNGE *villkor* uppfyllt

Ett annat sätt är att använda nyckelordet **TILLS** istället för **SÅ LÄNGE**. Så här skulle det i så fall se ut:

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

I vissa pseudokoder används faktiskt denna formulering, eftersom man antar att logiken uppfattas enklare med **TILLS** istället för **SÅ LÄNGE**.

Observera ordet **inte** i **TILLS**-satsen: man väljer samma villkor som i **SÅ LÄNGE**-satsen, men negerar, dvs vänder om villkoret i **TILLS**-satsen. Anledningen är skillnaden i den logiska innebörden av **SÅ LÄNGE** och **TILLS**. I flödesschemat av **do**-satsen blir det ingen strukturell ändring, bara man sätter sant och falskt på de logiskt korrekta utgångarna av villkoret. I några programmeringsspråk finns **TILLS** vid sidan av **while**, eller en av dem, men i C++ finns endast **while**. I pseudokoder däremot kan man inte undvika att stötta på **TILLS** istället för **SÅ LÄNGE**.

Ett exempel på **TILLS**-satsen är följande algoritm som är formulerad med **tills** i vanligt språk. Vill vi koda den i C++ måste vi formulera om den med **while**. Vi tittar först på den ursprungliga formuleringen och skriver om den sedan i pseudokod.

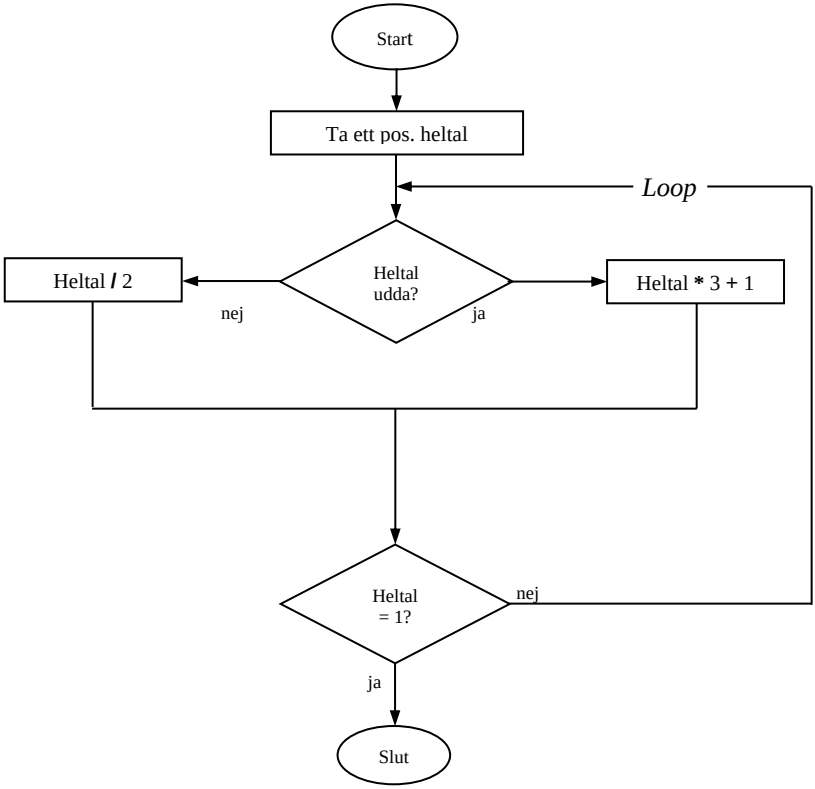
Collatz algoritmen

Lothar Collatz (1910-1990) var professor för tillämpad matematik vid Hamburgs Universitet på 60-talet. Som ung student ställde han upp följande uppgift:

> Tänk dig ett positivt heltal (startvärde).
> Är talet udda multiplicera det med 3 och addera 1.
> Är talet jämnt dividera det med 2.
> Gör samma sak med resultatet. Fortsätt **tills** du fått 1.

Det visar sig att talföljderna i denna algoritm, även känd som *Collatz förmodan* alltid slutar med 1 oavsett startvärde. Förmodan heter det eftersom påståendet är matematiskt hittills obevisat. Så här kan flödesschemat för denna algoritm se ut:

Flödesschema



Flödesschemat visualiserar algoritmens logiska struktur som är grundläggande för en korrekt implementering. Men för att slutligen koda kan det vara fördelaktigt att formulera algoritmen även som pseudokod som ligger närmare programkoden än flödesschemat. I pseudokoden använder vi formuleringen med **`SÅ LÄNGE`**:

Pseudokoden till Collatz algoritmen *

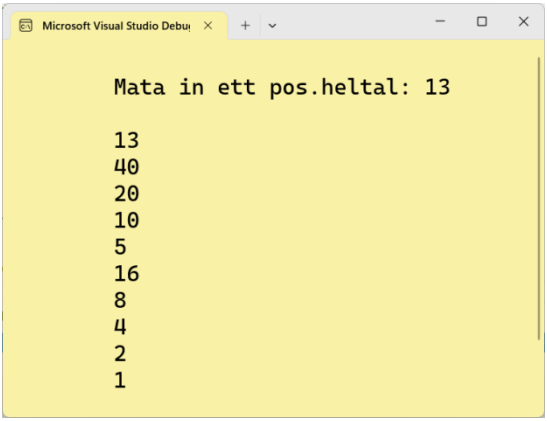
```
Läs in ett positivt heltal
REPETERA
   OM talet är udda
      multiplicera med 3, addera 1
   ANNARS
      dividera talet med 2
   Skriv ut talet
SÅ LÄNGE talet ≠ 1
```

Pseudokoden har anpassats till C++ programmering, t.ex. med formuleringar som `Läs in...`, **`REPETERA`** och `Skriv ut...` . Även **tills** i textformuleringen har bytts ut mot **`SÅ LÄNGE`**. Som en konsekvens av detta har även logiken vänts om från ”Fortsätt **tills** du fått 1” till ”**`SÅ LÄNGE`** `talet ≠ 1`”. Detta av anledningen att det i C++ inte finns **`TILLS`** utan bara **`while`**. I följande program implementeras Collatz algoritmen i C++. För **`REPETERA`** väljs **`do`**-satsen:

```
1  // Collatz.cpp
2  // Läser in ett pos. heltal, tar det gånger 3 och adderar 1,
3  // om det är udda. Delar det med 2 om talet är jämnt.
4  // Upprepar samma sak med resultatet, tills det blir 1.
5  // Använder do-sats för repetitionen -->
6  #include <iostream>
7  using namespace std;
8
9  int main()
10 {
11    int no;
12    cout << "\n\tMata in ett pos.heltal:\t";
13    cin >> no;                          // Startvärde
14    cout << "\n\t" << no;
15    do                                  // do loop börjar
16    {
17       if (no % 2 == 1)                 // Om no är udda
18          no = 3 * no + 1;
19       else                             // Om no är jämnt
20          no = no / 2;
21       cout << "\n\t" << no;
22    } while (no != 1);                  // do loop slutar
23    cout << "\n";
24 }
```

* Man kan testa Collatz algoritmen i appen *Mattekollen* där den är kodad i Python. Ladda ned appen eller kör den som Webbapp: **app.mattekollen.se → En mobil pythonmiljö**. Eller kör den direkt som webbapp: beta.mattekollen.se/#/app/coding. Prova koden med olika startvärden för att kolla om algoritmens talföljder alltid slutar med 1.

Här har vi ett körresultat med startvärdet **`13`**:



Prova gärna själv med andra startvärden, för att se: Talföljden som produceras av programmet **`Collatz`**, kommer att alltid avslutas med **`1`**, oberoende av startvärdet. Detta är ett rent empiriskt påstående som varken motbevisats hittills eller bevisats teoretiskt.

`do`-satsen i programmet **`Collatz`** på förra sidan är framhävd med vit bakgrund. Dess arbetssätt, dvs *repetitionen (loopen)* skiljer sig grundläggande från kontrollstrukturen *selektion* (sid 130) som är ett val. Det är avgörande att skilja mellan *repetition* och *selektion*. Medan en selektions alltid går *framåt*, efter den har avgjort valet p.g.a. det styrande villkoret, återvänder en repetition alltid till kontrollstrukturens början, dvs går *tillbaka* och utför koden som står i kroppen en gång till, även detta p.g.a. sitt avslutningsvillkor. Tydligast ser man detta i Collatz algoritmens flödesschema (sid 145) där programflödet (pilen) går från avslutningsvillkoret tillbaka, för att utföra det hela en gång till. I själva koden är de olika arbetssätten inprogrammerade i de reserverade ord som inleder dessa kontollstrukturer. Utan kännedom av deras logiska struktur som visas i flödesscheman är det inte möjligt att använda dem på korrekt sätt.

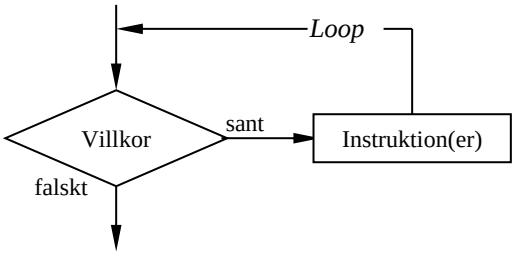
Ytterligare varianter av kontrollstrukturen *repetition* följer i de kommande avsnitten.

6.6 För-testad repetition: `while`-satsen

`while-satsen` är en loop där avslutningsvillkoret testas *före* loopens instruktioner, dvs *innan* det som ska upprepas. Enda skillnaden gentemot den efter-testade repetitionen är att ordningen mellan villkor och instruktioner vänds om:

Pseudokod och ritas som Flödesschema

SÅ LÄNGE *villkor* uppfyllt
instruktion(er)



I C++ inleds den för-testade repetitionen med **`while`**. Generellt:

```
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 1 och 104
// För-testad repetition: while-satsen
#include <iostream>
using namespace std;

int main()
{
   int i = 1;
   cout << '\n';
   while (i < 105)
   {
      cout << i << '\t';
      if (i % 8 == 0) cout << '\n';
      i++;
   }
   cout << "\n\t\t\tEfter loopen är i = " << i << "\n\n";
}
```

Först testas villkoret **`i < 105`**: 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 tre satser, en **`cout`**-sats – den första – som i varje varv skriver ut variabeln **`i`**:s värde följt av en tabulator. Den tredje satsen **`i++;`** gör att **`i`**:s värde ökar med **`1`**. Båda satser utförs så länge **`i`** är strikt mindre än **`105`** dvs från **`1`** till **`104`**. Dvs med **`104`** som **`i`**:s värde kommer vi fortfarande in i loopen, värdet skrivs ut och ökas sedan. Men med **`105`** 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 **`105`**:

```
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      100     101     102     103     104
                Efter loopen õr i = 105
```

Hade vi använt prefixvarianten av ökningsoperatorn **`++i`** hade talen från **`1`** till **`105`** 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:

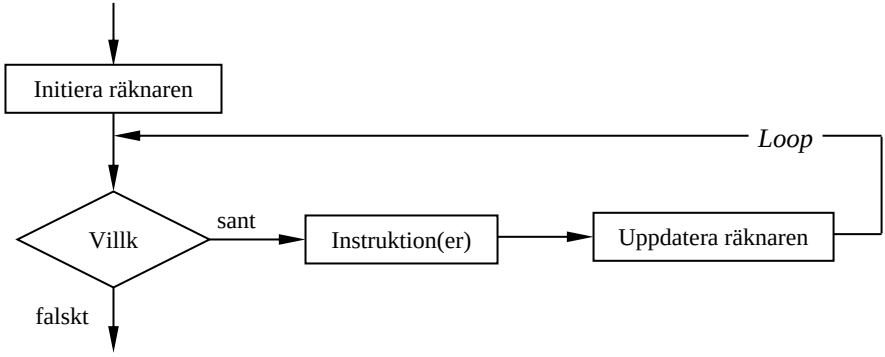
Evighetsslinga

I exemplet **`While`** är villkoret **`i < 105`**. 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 < 105`**. 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 **`105`**. Då stoppas slingan. Glömmer man ökningen **`++`** och initierar **`i`** med ett värde mindre än **`105`** blir **`while`**-satsen en evighetsslinga. Omvänt: Är **`while`**-villkoret falskt från början, görs ingenting. Initieras **`i`** till ett värde större än eller lika med **`105`**, 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 variabel som kallas *räknare* eller *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. Flödesschema ser ut så här:

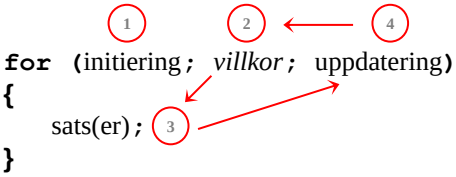


Flödesschemat åskådliggör den *logiska strukturen* av problemet. Den bestämda repetitionens pseudokod blir enligt flödesschemat ovan:

```
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 el-*

ler 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 flödesschemat på förra sidan kan man konstatera att C++ koden är lite kryptisk i den bemärkelsen att den inte följer flödesschemats ordning initiering – villkor – instruktion(er) – uppdatering. Pilarna markerar loopens förlopp. Initieringen görs endast en gång och ingår ej i loopen.

Översättning av while till for

För att se hur **for**-satsen fungerar har vi i följande program översatt **while**-satsen i programmet **While** (sid 148) till en **for**-sats:

```
// While->For.cpp
// Gör samma sak som While: skriver ut talen mellan 1 och 104
// Översättning av while-satsen (sid 148) till for-satsen
#include <iostream>
using namespace std;

int main()
{
   int i;
   cout << '\n';
   for (i = 1; i < 105; i++)
   {
       cout << i << '\t';
       if (i % 8 == 0) cout << '\n';
   }
   cout << "\n\t\tEfter loopen är i = " << i << '\n';
}
```

Programmet ovan ger exakt samma utskrift som **While** (sid 148) eftersom **for**-satsen i programmet är en ren översättning av **while**-satsen i **While**.

* 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)

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++;
}
```

Översatt till **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 149. 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.

for-satsens räknare

Andra sätt att skriva **for**-satsen med samma funktionalitet som ovan 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 flödesschemat på sidan 150: Där står initieringen utanför loopen. I alla dessa varianter deklareras räknaren **i** som en vanlig variabel *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 96, som inte är **== 0**. Därför utförs inte **if**-satsens kropp dvs inget radbyte skrivs ut. Efter **if**-satsen utförs enligt flödesschemat på sid 150 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 datastrukturer, t.ex. att åstadkomma en ordnad 2D utskrift, t.ex. en tabell, ett rektangulärt schema av tal eller den labyrintartade figuren i inlämningsupgift 2 (sid 166). Följande program nästlar två **`for`**-satser i varandra 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';
   }
}
```

När vi nästlade **`if-else`**-satser i varandra (sid 138) 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 utförs 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. 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. Nu när vi någorlunda behärskar den nästlade **for**-satsen kan vi använda den för att skriva ut t.ex.:

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";
   }
}
```

Samtidigt vill vi vidareutveckla tekniken att få rätt mellanrum mellan utskrifterna. Självklart kan man helt enkelt skicka ett antal mellanslag `"  "` till **cout**. I programmet ovan används istället en funktion för formatering av utskrift som är fördefinierad i **cout** just för sådana ändamål: **width()** är en s.k. *metod* eller *medlemsfunktion* som anropas så här strax före **cout**-satsen:

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

Denna metod 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.

Metoden **width()** * används här som ett alterativ till den manuella manipulationen av **cout**-satsen.

Testa gärna de olika alternativen för att få lite uft mellan utskrifterna. Men så som vi har placerat anropet i programmet **MultipTab** får vi följande utskrift när vi kör:

```
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
```

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.

* *Metoder* är funktioner som är definierade i en *klass* och anropas i ett *objekt* av klassen. Metoder kan endast anropas genom att först kalla på objektet och sedan med hjälp av s.k. *punktnotation* (sid 250) anropa metoden, t.ex. **cout.width()**.

Simulering av tärningskast

För att simulera tärningskast ska vi generera slumptal mellan **1** och **6**. Resultatet ska skrivas ut i tabellform, vilket är ett typiskt exempel på användning av nästlade **for**-satser. Repetera avsn. 4.9 (sid 94) för hantering av slumptal.

```
// Dice.cpp
// Simulerar tärningskast: Slumpar fram tal mellan 1 och 6
// och skriver ut dem i en tabell. Nästlad for-sats
#include <iostream>
using namespace std;

int main()
{
   srand(time(0));              // Skapar variation i slumpen
   int r, k, rader, kolumner;
   cout << "\nAnge antal rader och kolumner (t.ex. 10 15): ";
   cin >> rader >> kolumner;
  cout<<"\nDet blir "<< rader*kolumner<<" tärningskast:\n\n";
   for (r=1; r<=rader; r++)     // Låter en rad att skrivas ut
   {                            // och byter rad.
     for (k=1; k<=kolumner; k++)// Skriver ut den r:te raden.
         cout << 1 + rand() % 6 << "   ";
     cout << '\n';
   }
}
```

Ett körexempel av **Dice** 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
```

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 flödesschemat på sid 150 – 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:

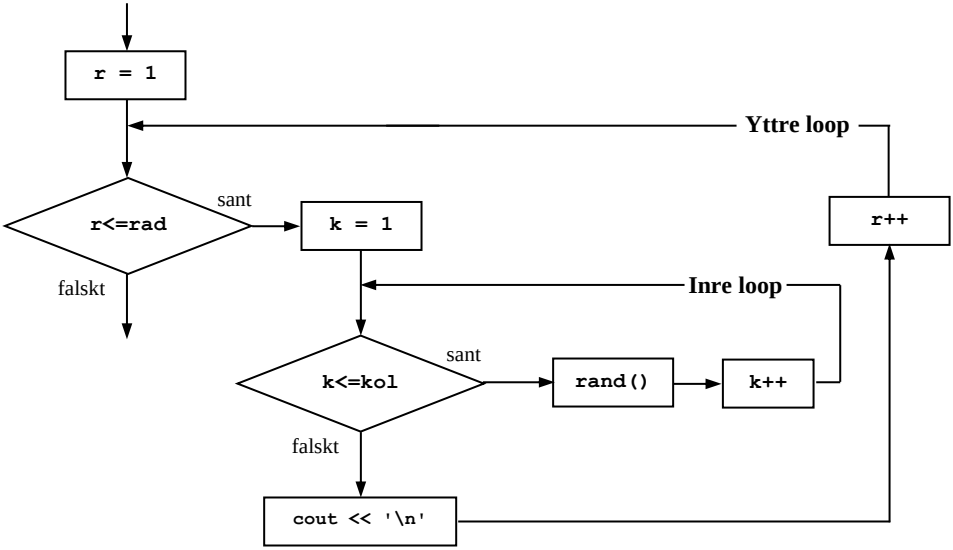
```
for (r=1; r<=rader; r++)
```

Detta huvud initierar en räknare **r** till **1**, testar villkoret, men skickar sedan programflödet enligt flödesschema till kroppen. Den består av två satser, därför klamrarna, 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 flödesschema 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. Tabellen skrivs ut radvis.

Efter den 1:a raden är vi klara med den yttre **for**-slingans kropp. Enligt flödesschema 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 2:a 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.



Detta är flödesschemat till den nästlade **for**-satsen. **rand()** är en slags förkortning för satsen **cout << 1 + rand() % 6 << " ";** i programmet **Dice**, sid 159.

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 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 142). **kbhit()** är definierad i biblioteket **conio.h**.

6.7 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 för sta 20 positiva heltalen?

6.8 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.

6.9 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.

6.10 Skriv ett program som läser in ett heltal som stegvariabel för att skriva ut tal från 1 till 5 000. Om steget är t.ex. 5 skrivs var femte tal ut.

6.11 ***Bergvärme (Projekt)*** En borrutrustning för bergvärme kan borra 25 m under den 1:a timmen i en viss tomtmark.

Under de följande timmarna minskar borrens prestation med uppskattningsvis 10 - 20% per timme. Den exakta minskningen är inte känd. Borren ska gå oavbrutet i 8 timmar.

Skriv ett program som uppskattar det totala borrdjupet.

Börja med att simulera minskningen av borrens prestation efter den 1:a timmen med slumptal mellan 10 och 20. Summera borrhålets djup efter den 1:a timmen baserad på denna simulation.

Skriv ut slutligen ett närmevärde för borrhålets totala djup efter 8 timmar.

Skriv ut även borrprestationens procentuella minskning per timme vid aktuell körning, t.ex.:

”Detta närmevärde baseras på en 14%-ig minskning av borrens prestationen per timme efter den första timmen.”

P.g.a. simuleringen med slumptal borde man få vid olika körningar olika procentsatser för minskningen av borrens prestation och därmed även andra uppskattnningar av det totala borrdjupet. I praktiken duger dock ofta en sådan uppsattning och är en värdefull information vid planering av arbetet.

Frivilligt! När programmet fungerar fundera på en vidareutveckling av uppgiften som kan lyda så här: Lös det omvända problemet: Skriv ett program som läser in ett önskat totalt borrdjup och beräknar samt skriver ut antalet arbetstimmar som borren skulle behöva för att nå detta borrdjup.

6.12 ***Frekvenstabell (Projekt)*** Skriv utgående från programmet **`Dice`** (sid 159) som simulerar tärningskast, ett program som genererar slumptal mellan **`1`** och **`6`** och ställer upp en frekvenstabell.

Frekvens är antalet förekomster av ett resultat bland tärningens 6 möjliga. Tabellen ska visa frekvensen för varje resultat vid olika antal simuleringar.

Låt programmet genomföra olika antal simuleringar och räkna vid varje simulering frekvensen för varje resultat **`1`**, ...,**`6`** av tärningskastet.

T.ex. ska man kunna läsa av från tabellen hur många gånger resultatet **`1`** förekommer när man kastar tärningen 50 gånger, 100 gånger, 1 000 gånger, 5 000 gånger, 10 000 gånger, osv. Avgör själv hur långt du går. Samma information ska man kunna få om tärningskastets andra resultat **`2`**, ... ,**`6`**.

Infoga i tabellen även en kolumn som för varje resultat visar kvoten:

Frekvens / Antalet tärningskast

Denna kvot är den experimentella sannolikheten för ett visst resultat.

Undersök på vilket sätt den experimentella sannolikheten närmar sig den ideala sannolikheten för varje resultat, som enligt sannolikhetsläran borde vara 1/6.

6.13 ***Löpande texten – en animation i konsolen (Projekt)***

Skriv en C++ Console Application som simulerar en löpande text. Ta som exempel texten **`C++ är kul>`** som ska röra sig horisontellt från konsolfönstrets vänstra kant tills den ”träffar” på ett hinder, t.ex. ett kryss **`X`**. Texten ska börja från vänstra kanten. Krysset ska ligga nära den högra kanten (ca. 70-80 tecken borta). Dessa ögonblicksbilder ska illustrera animationen:

Ledning:

Skriv ut först krysset **`X`** i slutet av en tom rad (fylld med mellanslag). Anteckna hur många mellanslag ni har valt för att placera krysset från konsolens vänstra kant. Gå i samma rad tillbaka till radens början genom att använda escapesekvensen **`\r`** (carriage return). **`\r`** skickar tillbaka markören till början av samma rad, utan att byta rad (till skillnad från **`\n`**). Skriv sedan ut **`C++ är kul>`** som då blir textens initialposition – det som visas i den första ögonblicksbilden ovan. Om ni vill bekanta er mer med **`\r`**:s funktion experimentera med det i ett annat program.

Rörelsen kan sedan simuleras t.ex. i en **`for`**-loop genom att i varje varv av loopen med ett antal **`\b`** ta bort texten som skrevs ut i förra varvet. Escapesekvensen **`\b`** (backspace) tar bort *ett* tecken till vänster om det aktuella tecknet, precis som tangenten backspace (←). Stega sedan med ett (eller flera) mellanslag, vilket kommer att bestämma rörelsens ”hastighet”. Skriv slutligen om texten **`C++ är kul>`**.

Beräkna antalet varv i **`for`**-loopen genom att ta hänsyn till textens längd och avståndet som kryss **`X`** har från vänstra kanten (som antecknats ovan). Har ni räknat rätt, kommer rörelsen att stoppas strax före krysset **`X`**, utan att ta bort det – liknande den tredje ögonblicksbilden ovan.

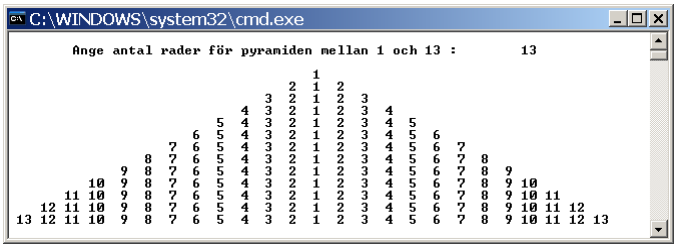
Även om ni gjort allt rätt kommer ni inte ”se” texten att röra sig, eftersom det går så fort, så att ögat inte hinner att se förloppet. Ni måste lägga in en fördröjning, vilket kan göras genom att infoga i loopen t.ex. satsen:

```
Sleep(100);
```

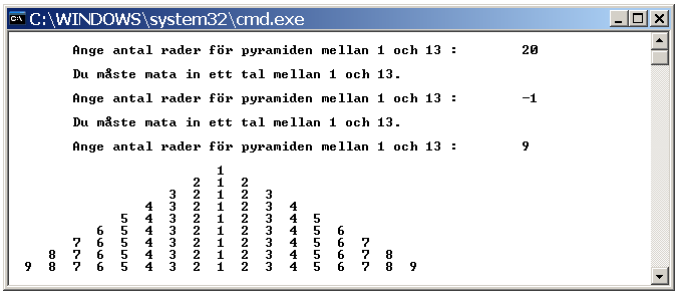
Parameterns enhet är millisekunder. Fördröjningsfunktionen **`Sleep()`** kräver inkluderingen av biblioteket **`windows.h`**.

6.14 *Pyramiden (Projekt)*

Slutmålet med detta projekt är att utveckla ett program som skriver ut en pyramidliknande figur med tal, som t.ex. ser ut så här:



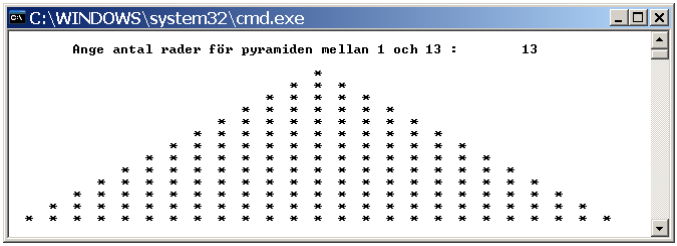
Programmet ska vara så generellt att det skriver ut talpyramider även om man matar in mindre antal rader. Uppmana användaren att hålla sig till talintervallet [1, 13]. Annars ryms talpyramiden inte i konsolen. Så här kan en körning se ut:



Ledning:

Denna ledning är endast en rekommendation och ska inte förhindra att ni använder egna idéer för att lösa problemet. Det finns andra möjliga tillvägagångssätt. Ni kan använda hela eller också delar av denna ledning för att komma igång.

Man kan *börja* med ett program som ritar en pyramid av *stjärnor* istället för tal:



Strunta till att börja med även på hanteringen av felinmatning av antal rader. Jobba med ett fast antal rader. Du kan lägga till det senare.

Använd en nästlad for-sats med en yttre loop och tre inre loopar:

- En för de tomma platserna i pyramiden (mellanslagen),
- En för stjärnorna i pyramidens högra halvan (räknat från den vertikala mittlinjen (symmetriaxeln),
- En för stjärnorna i pyramidens vänstra halvan.

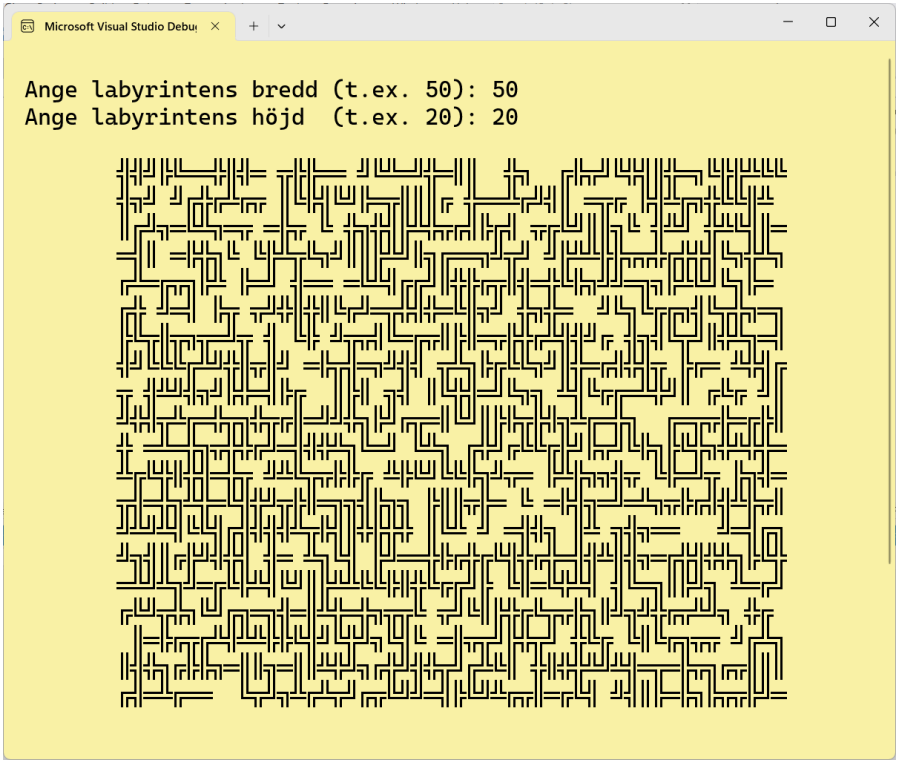
Räkna med att ni måste använda i de inre looparna den yttre loopens räknare och slutvärde. T.ex. kan villkoret i den första inre loop som ritar de tomma platserna, se ut så här:

```
column <= numberOfRows - row;
```

Här är **`column`** den inre loopens, **`row`** den yttre loopens räknare och **`numberOfRows`** hela pyramidens antal rader, t.ex. 13. Då kan den första inre loopen skriva ut tre mellanslag i varje varv. I de två andra inre looparna kan två mellanslag och en **`*`** skrivas ut.

Projektuppgift

Labyrinten Visst är det roligt att med våra C++ kunskaper hittills kunna skriva ett program som ritar en labyrintartad figur i konsolen som kan se ut så här:



Detta är förstås inte någon riktig labyrint. För en sådan skulle kräva mycket mer. En riktig labyrint med in- och utgång osv. skulle kunna ingå i ett spelprojekt med grafiska finesser, färger osv. En sådan avancerad figur kan inte ritas i konsolen.

Vår uppgift går snarare ut på att i *text mode* ”rita” en *labyrintartad figur* som är slumpmässigt ihopsatt av mellanslaget och ett antal tecken som vi kallar för *dubbla linjegrafiktecken (LGT)*. I figuren ovan är dessa tecken slumpade i en 2D utskrift, en slags tabell eller matris med 50 rader och 20 kolumner. I koden åstadkommer man detta med en nästlad **`for`**-loop, där den yttre loopen skriver ut raderna och den inre loopen kolumnerna, se avsn. ***6.8 Nästlade for-satser***, sid 156. Alla tecken i figuren ovan är slumpvist valda bland de dubbla LGT och mellanslaget. Därför borde varje körning av programmet generera en lite annorlunda labyrintartad figur.

Du kan gärna försöka med en egen algoritm att skriva ett C++ program som ritar en sådan figur. Men i instruktionerna som följer nedan har du i alla fall ett förslag till en enkel algoritm (***Steg 1-5***) som fungerar.

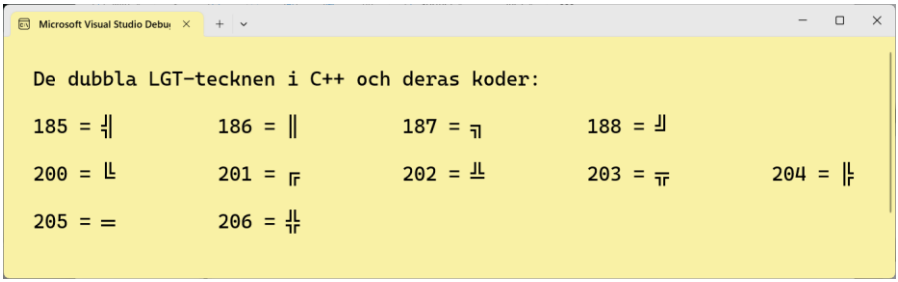
Algoritmen

Steg 1 Bekanta dig med hantering av tecken i C++ inkl. explicit typkonvertering (sid 119), genom att experimentera med programmet **`Int2char`** som behandlades på lektion 11 (sid 120):

```
// 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 ett heltal: ";
   cin >> code;
   cout << "\nDet inmatade heltalet är " << code <<
        " och är koden till tecknet " << (char) code << "\n";
}
```

Experimentera med **`Int2char`** genom att mata in koderna **`185-188`** och **`200-206`**.

Steg 2 För få en översikt över alla elva dubbla LGT samt deras koder i C++ skriv ett program som producerar följande utskrift:



Dessa tecken finns i den utvidgade delen av teckentabellen (utöver standard ASCII) och används för att rita raka linjer, ramar, tabeller, skisser osv i en textbaserad miljö *(text mode)*. De kan användas tillsammans med mellanslaget för att rita labyrinten. Jämför koderna även med utskriften till programmet **`AsciiFor`** (sid 154).

Steg 3 Repetera hantering av slumptal i avsn. ***4.9 Hantering av slumptal*** (sid 94), speciellt om ***Slumptal inom ett intervall*** (sid 95).

Steg 4 Skriv ett program som med hjälp av C++:s slumpgenerator och en nästlad **`for`**-sats ritar labyrinten, en slumpmässigt ihopsatt figur bestående av de dubbla LGT samt mellanslaget.

Steg 4 i detalj

Här följer i själva verket hur ***Steg 4*** kan realiseras:

Deklarera en teckenvariabel **`letter`** och en heltalsvariabel **`randNo`**. Initiera **`letter`** till mellanslaget, dvs **`' '`**. Låt **`randNo`** anta slumpvärden mellan **`0`** och **`11`** med satsen:

```
randNo = rand() % 12;
```

Fortsätt med följande **`if`**-satser:

Om **`randNo`** blir **`0`**	ska **`letter`** tilldelas	1:a LGTs teckenkod.
Om **`randNo`** blir **`1`**	ska **`letter`** tilldelas	2:a LGTs teckenkod.
Om **`randNo`** blir **`2`**	ska **`letter`** tilldelas	3:e LGTs teckenkod.
.	.	
.	.	
.	.	
Om **`randNo`** blir **`9`**	ska **`letter`** tilldelas	10:e LGTs teckenkod.
Om **`randNo`** blir **`10`**	ska **`letter`** tilldelas	11:e LGTs teckenkod.
Om **`randNo`** blir **`11`**	ska **`letter`** tilldelas	mellanslaget, dvs **`' '`**.

Numreringen av LGT-teckenkoderna avser den ordning som är föregiven i teckentabellen, se utskriften på förra sidan. I övrigt fungerar vilken numrering som helst.

Observera att LGT-teckenkoderna är av typ **`int`**, medan variabeln **`letter`** är av typ **`char`**. Vid tilldelningen konverteras **`int`** automatiskt till **`char`**. Vid utskriften med **`cout`** skrivs ut tecknet, inte koden. Mellanslaget (**`' '`**) däremot är från början av typ **`char`**, så att vi inte behöver bry oss om koden. Se upp att **`''`** inte är mellanslaget utan den tomma strängen och ger kompileringsfel.

Skriv ut **`letter`**, så att du får ETT tecken, antingen ett LGT eller mellanslaget. Testa även om du vid varje körning får *olika* LGT eller mellanslaget.

Först när allt detta fungerar låt tilldelningen av variabeln **`randNo`** samt de 12 **`if`**-satserna ovan ingå i en enkel *loop*, t.ex. en **`for`**-sats, för att rita labyrinten.

Ersätt den enkla loopen 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.

Algoritmen ovan är enkelt genomförbar, men inte den mest eleganta lösningen. Har du klarat av den kan du gärna gå vidare till följande:

Extrauppgift (frivilligt)

Ersätt de 12 **`if`**-satserna ovan med en annan konstruktion: Samla alla dina LGT-koder och mellanslaget i en *array* och låt slumpen ta ut tecken ur denna array. För utskriften kan du fortsätta med att använda nästlad **`for`**-sats. Det blir en kortare och elegantare kod.

Kapitel 7

Funktioner

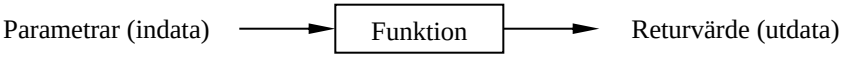
7.1 Funktionsbegreppet i programmering

Begreppet *funktion* härstammar från matematiken: Man har en formel y = f(x) som beräknar ett tal y utgående från ett annat tal x och säger: y är en funktion av x. Denna matematiska syn på funktion har tagits över till programmering som ett underliggande koncept och som en historisk utgångspunkt. Men under tiden har begreppet vidareutvecklats och fått en bredare tolkning då den inom programmering tillämpats på all datoriserad problemlösning. I programmering inkluderar funktioner även matematiska problem, men är inte begränsade till dem.

En funktion är kod som definieras som en namngiven modul och placeras utanför **`main()`**.

Koden utförs inte förrän funktionen anropas i **`main()`**.

Vid anropet kan funktionen ta emot indata, s.k. parametrar, bearbeta dem och returnera utdata, s.k. returvärde.



Man kan jämföra en funktion med en ”svart” låda i vilken man stoppar indata och får ut utdata: Indata kallas även *parametrar* (argument) och utdata *returvärde*.

En funktion kan ha inga, en eller flera parametrar. Den kan ha inget eller endast *ett* returvärde, dvs en funktion kan inte ha flera returvärden, vilket är ett arv från matematiken. Både parametrarna och returvärdet kan vara tal, tecken, strängar, sanningsvärden, objekt eller andra datatyper. Funktionen bearbetar de inkommande parametrarna och returnerar returvärdet eller inget alls. Funktioner utan returvärde kallas för **`void`**-funktioner. Som separat och namngiven modul kan en funktion anropas i **`main()`**, men även även i andra funktioner eller program. I denna bemärkelse är en funktion ett *underprogram*, på eng. *subroutine* eller *procedure*.

”Svart” är lådan så länge vi inte vet hur den fungerar ”inuti”, dvs så länge vi inte själva definierat funktionen. I så fall använder vi den endast för att lösa ett visst problem. Det gäller t.ex. för de biblioteksfunktioner som vi hittills använt i våra program: **`sizeof()`**, **`_getch()`**, **`rand()`**, **`srand()`** och **`time()`**. De är förprogrammerade och lagras i bibliotek. Vi inkluderar 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 program – är **`main()`**. Den är unik därför att den är programmets exekveringspunkt (sid 54). I alla procedurala programspråk bildar funktioner programmets byggstenar (moduler). Att C++ som är objektorienterat, även bygger på funktioner

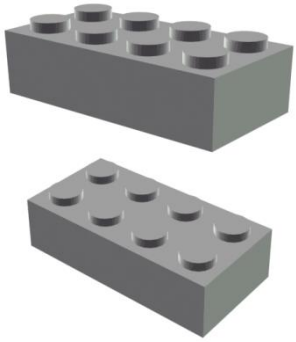
och inte bara på klasser, beror på språkets rötter i C som är proceduralt, men inte objektorienterat. C# och Java t.ex. bygger endast på klasser som inkapslar sina funktioner i klasser och döper dem till *metoder*.

Varför funktioner?

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. 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.

Modularisering eller Lego-principen

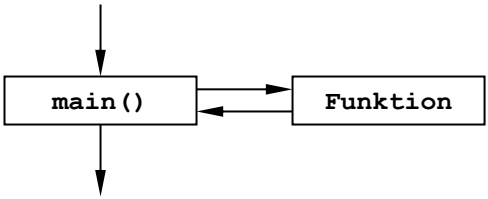


De flesta har väl någon gång som barn, eller tillsammans med sina barn, byggt ett hus, en bil eller liknande med Lego-bitar. Efter ett tag har huset kanske rasat och nya tekniska underverk har konstruerats. Men även de har någon gång plockats isär. Det enda som blivit kvar är själva Lego-bitarna som 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. Varje modul löser ett delproblem som är oberoende av andra, är mindre än det stora problemet och därmed enklare att lösa. Sedan gäller det att sätta ihop modulerna till den stora lösningen. I programmering är dessa moduler *funktioner*.

För att att sätta ihop det hela 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 *an-*

rop 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 returvärdet. 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 86). 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.

7.2 Funktioner med returvärde

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 180). Låt oss gå tillbaka till sid 86:

```
// 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:

```
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 datatyp **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. Returvärdet returneras till funktionsnamnet **totalsek()**.

Att funktione **totalsek()** innehåller en **return**-sats är anledningen till varför den är en *funktion med returvärde*.

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)   // Definitionen av fkt.
{
   /* 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. 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.
```

Vad händer när en funktion anropas?

När **totalsek()** anropas i den sista satsen av **main()** händer tre saker:

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.

```
int main()
{
   int tim, min, sek;
   cout << "\nGe timmar, minuter, sekunder "
        << "skilda med mellanslag:  ";
   cin  >> tim >> min >> sek;
   cout << '\n' << tim << " timmar, " << min <<" minuter och "
                                        // Anrop av funktion:
        << sek << " sekunder är " << totalsek(tim,  min,  sek)

        << " sekunder totalt.\n\n";
}
```

```
int totalsek(int t, int m, int s)
{
     return 3600*t + 60*m  +   s;
}
```

En översikt över dataflödet mellan den anropande funktionen **main()** och den anropade funktionen **totalsek()** visas på bilden ovan. Det som händer vid anrop av en funktion är alltså att data byts ut mellan dessa två funktioner, att koden i funktionen **totalsek()** utförs när anropet i **main()** inträffar, samt att returvärdet till sist hamnar i **main()**. Observera i *vilken ordning* funktionernas koder utförs.

Bilden ovan visar vad som *händer* när programmet **`MyFirstFct`** (sid 174) 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 193. Inledningsvis hade vi nämnt block på sid 192.

7.3 Definition och anrop av funktioner

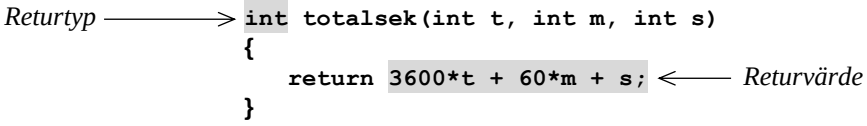
Vi ska nu i detalj gå igenom alla steg hos funktioner *med* returvärde. I avsnitt ***7.6 Funktioner utan returvärde*** kommer den andra typen av funktioner, s.k. **`void`**-funktioner, att behandlas (sid 180).

Allmän form på 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.

I programexemplet **`MyFirstFct`** är funktionen **`totalsek()`** definierad på följande sätt:



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 exemplet är **`int`** är returtypen. Funktionen returnerar ett uttryck som är av typ **`int`**. Men varför står returtypen framför funktionsnamnet? Det verkar – om man för ett ögonblick bortser från parameterlistan – som om **`int totalsek`** vore en deklaration 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 anropas. 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`**.

Placering av funktioners definition

Kan man definiera funktioner var som helst i ett C++ program? I vårt fösta exempel (sid 173) 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 kommer att behandlas i de efterföljande avsnitten.

Men först ska vi formulera 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 171).

Allmän form på 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. Den allmänna formen för anrop av en funktion med returvärde ser ut så här:

```
cout <<    funktionsnamn(apar1, apar2, ...);

                    eller

variabel = 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.

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.

I programmet **`MyFirstFct`** anropas funktionen **`totalsek()`** från funktionen **`main()`** i följande sats:

```
cout << … << totalsek(tim, min, sek) << … ;
```

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.

7.4 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, inte flera. 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 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 antingen ingen **`return`**-sats alls eller en tom **`return`**-sats, dvs: **`return;`**

I följande program definieras en **`void`**-funktion i en extern fil. En tom **`return`**-sats avslutar funktionen om talen är lika stora. Annars fortsätter funktionen. Denna funktion är inte avsedd att returnera något värde. Istället skriver den ut resultatet av en jämförelse mellan två tal – en typisk användning av en **`void`**-funktion.

Exempel på en void-funktion

```
// Compare.h
// Definierar funktion som jämför två tal på likhet, mindre
// än eller större än och skriver ut resp. meddelande.

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";
   else
      cout << "För STORT, försök igen !\n";
}
```

`void` är i C++ ett reserverat ord som kan tolkas som ”ingenting” och ersätter returtypen i funktionshuvudet. **`void`** eliminerar returvärdet och definierar en **`void`**-funktion. Även en tom **`return`**-sats avslutar funktionen, oavsett var den skrivs i funktionskroppen, vilket innebär att all kod efter den inte utförs. **`void`** kan även ersätta en tom parameterlista och betyder då att funktionen inte har några parametrar alls. T.ex. kan man skriva **`int main(void)`** istället för **`int 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”.

I **void**-funktionen **compare()** tyder meddelandetexterna på att denna funktion används i samband med ett program som gissar tal. Det är faktiskt Gissa tal-spelet som vi tidigare behandlat i två varianter: **GissaTal_1** (sid 138) och **GissaTal_2** (sid 143). Nu ska vi vidareutveckla Gissa tal-spelet genom att slumpa programmets hemliga tal och dessutom modularisera det.

Vilka moduler – logiskt meningsfulla delproblem – finns i **GissaTal_2**? En av dem är att jämföra det gissade talet med programmets hemliga slumptal och skriva ut ett passande meddelande. Detta görs i den ovan definierade **void**-funktionen **compare()**. 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. Utan **return**-satsen skulle efter grattismeddelandet även skrivas ut: **För STORT, försök igen!** vilket inte vore önskvärt.

Den andra modulen i Gissa tal-spelet är:

```
// GissaTal_3.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 funktioner är externlagrade och inkluderas här
#include <iostream>
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 **GissaTal_3** 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. Nu skriver vi den i filen **MyRand.h** som inkluderas i **GissaTal_3** med **#include "MyRand.h"**. Då måste den ligga i samma mapp som filen **GissaTal_3.cpp**. Vi kommer även i ett senare program (sid 183) använda den, fast som en funktion i samma fil.

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 intervallet **(1, 20)**. 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 tal-spelet består nu av dessa tre funktioner: **main()**, **compare()** och **myRand()**. Ett körexempel skulle likna det på sid 144. För förklaring av formlerna i funktionen **myRand()** se sid 95.

7.5 Deklaration av funktioner

Här pratar vi inte om *definition* – för, det har vi gjort tidigare (sid 177) – utan om *deklaration* av funktioner. Vad som är skillnaden kommer vi att se.

Hittills har vi placerat våra funktioners definition *före* **`main()`**, t.ex. funktionen **`totalsek()`** i programmet **`MyFirstFct`** (sid 174). 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 placera funktionen t.o.m. i en separat fil som en externlagrad funktion. Det är alltså modulariseringsprocessen som leder till deklaration av funktioner. Följande exempel demonstrerar detta:

Modularisering av tärningskast

```
// Dice_Fct.cpp
// Gör samma sak som programmet Dice (tärningskast, sid 159)
// Slumptalsgenerering har flyttats till en funktion som
// definieras efter main() och därför måste deklareras
#include <iostream>
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);
}
/*********************************************************/
```

Funktionen **myRand()** slumpar heltal i intervallet **[a, b]**, se sid 95.

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()**. Kompilatorn som går igenom koden rad för rad, stöter *först* på funktionsanropet i **main()** *innan* den hittar definitionen som står *efter* **main()**. I så fall kan kompilatorn inte tolka funktionsanropet och programmet kan inte kompileras. Deklarationen löser problemet i och med den ger en förhandsinformation 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 tolka funktionsanropen och därmed kompilera programmet **Dice_Fct**. Pga deklarationens roll som en förhandsinformation pratar man ibland om *forward declaration* eller *prototyp* av en funktion. Punkterna 1 och 3 ovan kallas för *funktionens signatur*.

Signaturen

Funktionens namn (1) och antalet parametrar samt parametrarnas datatyper (3) är funktionens igenkänningstecken och kallas därför för funktionens *signatur*. Det är anmärkningsvärt är att returtypen (2) inte ingår i signaturen. Dvs i exemplet ovan är funktionens signatur:

```
myRand(int a, int b)
```

Signaturen är alltså en del av deklarationen vars betydelse framgår av att funktioner med samma signatur anses vara *identiska*. Funktioner som skiljer sig på någon av punkterna 1 (namn) eller 3 (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. Därför är de olika.

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. Därför skriver vi i exemplet ovan deklarationen så här:

```
int myRand(int, int);
```

Deklarationen *kan* också skrivas så här: **int myRand(int a, int b);** men parametrarna **a** och **b** kan även utelämnas vilket vi gjort i **Dice_Fct**.

Vi utelämnar parametrarnas namn, för 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 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.

Placering av funktionsdeklarationer

Deklarationen kan placeras på två ställen: antingen *före* **`main()`** eller *i* **`main()`** som vi gjort. En deklarationssats *före* **`main()`**, dvs på samma plats som **`include`**-direktiven, skulle gälla inte bara i **`main()`** utan även i alla andra funktioner som ev. följer efter **`main()`**. Däremot gäller en deklaration *i* **`main()`** endast i **`main()`**.

Ett körexempel av programmet **`Dice_Fct`** ger ”samma” resultat som programmet **`Dice`** (sid 159), 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
```

7.6 Externlagrade funktioner

Deklaration av funktioner som behandlades i förra avsnitt var en led i processen – ett mellansteg – att isolera en funktion och placera den i en separat fil som en *externlagrad* funktion. Modulariseringen har inte slutförts än. Så länge en funktion är inbunden i samma fil som det anropande programmet, t.ex. **`main()`**, kan den inte användas av andra program. För att kunna dra nytta av återanvändning av kod, måste programmet brytas ned i oberoende moduler och lagras i separata filer. Sedan måste modulerna länkas ihop till ett program. I C++ är modulerna funktioner, inkl. funktionen **`main()`**. I följande uppdrag som ställs till oss av en kund, kommer vi att utveckla ett program som består av två funktioner, lagrade i två separata filer:

Projekt Ingående moms

> ”Vi har ett kassasystem som vi inte är nöjda med. För att uppdatera vårt gamla digitala kassasystem behöver vi ett program som skriver ut en tabell med information om **1.** en varas pris inkl. moms, **2.** nettopriset och **3.** den ingående momsen. Tabellsteget ska vara variabelt, för att kunna förfina tabellen. Även momssatsen ska vara variabel, eftersom vi har varor med olika momssatser. Programmet ska beräkna nettopriset och den ingående momsen utgående från bruttopriset.”
>
> ***Kundens kravspecifikation***

När vi undersöker kundens gamla kassasystem och studerar kravspecifikationen konstaterar vi att det finns en gömd problematik som ev. skulle kunna vara en anledning till missnöjet med det gamla kassasystemet. Boven i dramat är beräkningen av den ingående momsen.

Problematiken hos ingående moms

Exempel: en vara kostar i butiken 100:- kr inkl. moms. Hur stor är den ingående momsen? Den är *inte* 25:- kr om vi räknar med 25% moms. Ingående momsen är 20:- kr och nettobeloppet 80:- kr, eftersom 80 x 0,25 är 20:- kr. Men hur hittar man nettobeloppet och den ingående momsen utgående från bruttobeloppet, speciellt när momssatsen ska variera? Vi misstänker att denna fråga inte var korrekt besvarad i kundens gamla kassasystem. Vi kommer att besvara frågan genom att härleda en formel för nettobeloppet (sid 189). Men vad är egentligen ingående momsen?

Ingående momsen är den moms man lagt till ett *nettobelopp*, för att få bruttobeloppet (varans pris). Att räkna 100:- kr x 0,25 = 25:- kr är fel och ger *inte* ingående momsen. Det är *nettobelopet* som måste tas x 0,25 för att ge den ingående momsen, dvs 80:- kr x 0,25 = 20:- kr. Men 80:- kr är inte känt från början. Vi löser kundens problem med en formel som i den externlagrade funktionen **`netto()`** beräknar det okända nettobeloppet. Programmet **`IncludingVAT`** (eng.: *VAT = Value Added Tax*) inkluderar funktionen **`netto()`** och skriver ut den önskade tabellen:

```
// IncludingVAT.cpp
// Skriver ut en tabell över nettobelopp och ingående moms
// main() matar in och ut samt anropar funktionen netto()
// 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 netto och ingående moms "
        << "utgående från brutto:\n";
   cout << "\n\tStart bruttobelopp:\t";
   cin >> first;
   cout << "\tSlut bruttobelopp:\t";
   cin >> last;
   cout << "\tTabellsteget:\t\t";
   cin >> step;
   cout << "\tMomssats i % (t.ex.25):\t";
   cin >> vatRate;
   cout << fixed << setprecision(2); // Efter detta skrivs ut
                                     // alla tal med två 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";
}
```

Programmet **IncludingVAT** skriver ut följande tabell som var kundens önskemål:

```
Tabell över netto och ingående moms utgående från brutto:

        Start bruttobelopp:         125
        Slut  bruttobelopp:         150
        Tabellsteget:               5
        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
```

I körexemplet har vi valt en momssats som är vanlig i Sverige. Men programmet **IncludingVAT** tillåter vilken momssats (**vatRate**) som helst. Med denna momssats beräknar funktionen **netto()** nettobeloppet utgående från ett bruttobelopp, medan huvudprogrammet drar av **netto** från **brutto** för att få ingående momsen.

Programmet **IncludingVAT** (sid 187) läser in bruttobelopp för början och slutet på en tabell som skrivs ut. 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 finns 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.

Formatering av decimaltal

Som man ser i utskriften ovan har alla decimaltal två decimaler. Förklaringen 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**.

Funktionen netto()

Filen **Netto.h** innehåller definitionen av funktionen **netto()**:

```
// Netto.h
// Beräknar nettobeloppet utgående från ett bruttobelopp
// Definierar funktionen netto() som tar in ett beloppet
// brutto samt momsSats och returnerar nettobeloppet

float netto(float brutto, float momsSats)
{
  return 100*brutto / (100 + momsSats); // Formeln för beräk-
}                                       // ning av nettobelopp
```

Filändelsen **h** 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. **#include "Netto.h"** som står före **main()** gör att kompilatorn hittar **h***eaderfilen* **Netto.h**. 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 **IncludingVAT.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.

De formella parametrarna **brutto** och **momsSats** kommer att ta emot sina värden från de aktuella parametrarna **brutto** (OBS! En annan variabel med samma namn) och **vatRate** när **main()** anropar funktionen **netto()**. Funktionen beräknar sedan uttrycket och returnerar resultatet till **main()** som är placerad i programfilen **IncludingVAT.cpp**. Att funktionerna som är placerade i olika filer, hittar varandra, beror på att båda filer ligger i samma mapp och att **h**-filen är kopplad till **cpp**-filen via **include**-direktivet.

Härledning av formeln för nettobeloppet

I funktionen **netto()** används i **return**-satsen en formel som beräknar nettobeloppet, när bruttobeloppet och momssatsen är kända.

Här följer en matematisk härledning av denna formel:

$$\text{netto} + \frac{\text{netto} \cdot \text{momssats}}{100} = \text{brutto}$$

$$\frac{100 \cdot \text{netto} + \text{netto} \cdot \text{momssats}}{100} = \text{brutto}$$

$$\text{netto} \cdot (100 + \text{momssats}) = 100 \cdot \text{brutto}$$

$$\text{netto} = \frac{100 \cdot \text{brutto}}{100 + \text{momssats}}$$

Denna formel används i funktionen **netto()**:

```
return 100*brutto / (100 + momsSats);
```

Skulle man vilja skriva en funktion för ingående moms, kan man sätta in formeln ovan i sambandet *ingående moms = brutto – netto* och förenkla. Resultatet blir följande formel för ingående moms:

$$\text{ingående moms} = \frac{\text{brutto} \cdot \text{momssats}}{100 + \text{momssats}}$$

Genomför gärna denna härledning för att testa dina kunskaper i algebra.

I nästa avsnitt kommer vi att använda formeln i funktionen **vat()** som är en del av programmet **VAT_table** (sid 191), för att direkt beräkna den ingående momsen.

7.7 Lokala och globala variabler

Externlagrade funktioner som behandlades i förra avsnitt, leder oundvikligt till att variabler sprider sig till de olika modulerna av ett program. Dvs samma variabel kan t.ex. vara deklarerad i huvudprogrammet, men även behövas i en annan modul som är externlagrad. Kommer den fortfarande vara giltig där? Eller anropet av funktioner som är definierade externt ger upphov till blockbildning, Frågan uppstår: vilka regler gäller för livslängden av variabler, när de används i olika block?

Det är sådana och liknande frågor som vi ska ta itu med nu, för det finns väldigt strikta regler i C++ för hantering av variabler och deras livslängd samt för överskridning av gränser mellan olika block. Man skiljer mellan *lokla* och *globala* variabler som följer *olika* regler för överskrudning av blockgränser. Det nya är tt vi kommer att lära oss att deklarera och använda globala variabler och bekanta oss med deras problematik. Samtidigt kommer detta leda oss till nästa avsnitts tema: *Överskuggning av variabler*. Allt detta ska vi studera i följande projekt:

Projekt Momstabell

”Nu, när vårt digitala kassasystem fungerar tillfredsställande – tack förresten för en bra leverans – skulle vi gärna vilja vidareutveckla vårt gamla projekt *Ingående moms*. Vi önskar ett program som skriver ut en momstabell. Man ska kunna välja mellan **1.** att lägga till moms och få en tabell över *tillkommande moms* och **2.** att dra av moms och få en tabell över *ingående moms*. Den tillkommande momstabellen ska visa bruttopriset samt momsen utgående från nettopriset. Den ingående momstabellen ska visa nettopriset samt momsen utgående från bruttopriset. I båda fall ska tabellsteget och momssatsen vara variabla.”

Kundens kravspecifikation

Vi har implementerat kundens kravspecifikation i programmet **`VAT_table`** (sid 191) som har två funktioner: **`vat()`** och **`main()`**. Där finns för första gången en variabel som är deklarerad utanför både **`main()`** och **`vat()`**: Variabeln **`vatType`** kallas för *global* just därför att den är deklarerad *utanför* programmets båda funktioner. Därför är den giltig i båda. *Placeringen* gör den till en global variabel.

Men varför behöver vi variabeln **`vatType`** i båda funktionerna? Anledningen är att den globala variabeln **`vatType`** avgör mellan programmets två funktionaliteter. Vi citerar ur kundens kravspecifikation:

> ” Man ska kunna välja mellan **1.** att lägga till moms och få en tabell över *tillkommande moms* och **2.** att dra av moms och få en tabell över *ingående moms*. ”

Variabeln **`vatType`** avgör valet mellan dessa två alternativ. Därför måste den gälla både i **`vat()`** och **`main()`** och måste därför deklareras globalt. I programmet **`VAT_table`** har vi för första gången implementerat detta koncept:

```
// VAT_table.cpp
// Skriver ut en tabell över tillkommande eller ingående moms
#include <iostream>
#include <iomanip>                  // Krävs för setprecision()
using namespace std;

int vatType;                        // Global variabel

/*****************************************************************/
float vat(float a, float v)         // Definierar funktio-
{                                   // nen 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 << "\n\tStart bruttobelopp:\t";
   cin >> first;
   cout << "\tSlut bruttobelopp:\t";
   cin >> last;
   cout << "\tTabellsteget:\t\t";
   cin >> step;
   cout << "\tMomssats i % (t.ex.25):\t";
   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";
   }
}
```

Ett körexempel av **VAT_table** för tillkommande moms (1) kan se ut så här:

```
Vill du lägga till(1) eller dra av(0) moms? (1/0): 1

        Start bruttobelopp:     66.50
        Slut bruttobelopp:      70
        Tabellsteget:           1
        Momssats i % (t.ex.25): 25

Netto           Brutto          Tillkom. moms
-----------------------------------------------
66.50           83.12           16.62 kr
67.50           84.38           16.88 kr
68.50           85.62           17.12 kr
69.50           86.88           17.38 kr
```

Samma indata som ovan ger för ingående moms (0) följande utskrift:

```
Vill du lägga till(1) eller dra av(0) moms? (1/0): 0

        Start bruttobelopp:     66.50
        Slut bruttobelopp:      70
        Tabellsteget:           1
        Momssats i % (t.ex.25): 25

Brutto          Netto           Ingående moms
-----------------------------------------------
66.50           53.20           13.30 kr
67.50           54.00           13.50 kr
68.50           54.80           13.70 kr
69.50           55.60           13.90 kr
```

Skulle man välja momstypen **0** (ingående moms) och mata in samma indata som till programmet **IncludingVAT** (sid 187) skulle man få samma resultat som på sid 187. Slutsats: Programmet **IncludingVAT** är ett specialfall av programmet **VAT_table** när detta körs med momstypen **0**. **VAT_table** är en generalisering av **IncludingVAT**.

Block

I C++ kallas ett antal satser som omsluts av klamrarna **{** och **}** för *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.

Exempel på block fanns redan i vårt allra första program. **main()**-funktionens 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 att använda den första konventionen, för att lättare para ihop de inledande och avslutande klamrarna.

Exempel på blockbildning har vi sett tidigare, utan att lägga märke till dem. Som exempel går vi tillbaka till programmet **`MiniSort`** (sid 134). Där kodas en sorteringsalgoritm i tre satser i kroppen till en **`if`**-sats. Avgränsningen med blockets klamrar **`{`** och **`}`** innebär att alla tre satser hör till **`if`**-satsen och att alla tre ska utföras i fall att **`if`**-satsens villkor är sant. Om blockmarkeringen 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. Gruppering av satser i kroppen av kontrollstrukturer´– är ytterligare exempel på block.

Blockstruktur

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 **`VAT_table`** (sid 191) innehåller flera anrop av funktionen **`vat()`**. Alla anrop ger upphov till blockstruktur. Vi har ritat det första anropet:

```
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);
                    }
   .
   .
   .
}
```

Klamrarna **`{`** och **`}`** sätts ofta för att markera gränsen för variablers räckvidd. För att överskrida dem måste vissa regler om *blockstruktur* beaktas. Dessa regler kan även betraktas som regler för lokala och globala variabler.

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.

Variablers livslängd

Frågan som dyker upp är: Vad händer med en variabel när man överskrider blockgränserna? Hur långt räcker en variabels giltighet? Man pratar om variablers *livslängd* eller *räckvidd*, på engelska *scope*. Vi kommer att använda *räckvidd*.

Generellt gäller:

Regeln för räckvidden av variabler:

En variabels räckvidd börjar vid deklarationssatsen och slutar vid gränsen till det block där den är deklarerad.

Om vi tillämpar regeln ovan på t.ex. variabeln **vatType** i programmet **VAT_table**, visar blockstrukturen på förra sida att det inte finns något block som variabeln är deklarerad i och därmed inte heller någon gräns. **vatType** är ju deklarerad utanför alla block. Därav följer att variabelns räckvidd börjar vid deklarationen och slutar först när programet tar slut vilket innebär att **vatType** är en *global variabel*, medan alla andra variabler i programmet **VAT_table** (sid 191) är lokala. Regeln ovan ger oss även kriteriet för att skilja mellan lokala och globala variabler.

Lokala variabler

Sammanlagt finns det i **main()** fem lokala variabler: **first**, **last**, **step**, **vatRate**, **amount** och **vat**. Därför gäller de endast i **main()**, men inte i **vat()**.

Hur är det med lokala variabler i funktionen **vat()**, finns några där? I kroppen är inga variabler deklarerade. Hade sådana funnits hade de varit lokala. De enda variabler som används i **vat()** är parametrarna **a** och **v** som är deklarerade i parameterlistan.

Angående parametrar i en funktion gäller generellt följande regel:

Parametrarna i en funktion behandlas som lokala variabler i funktionen.

Därför gäller variablerna **a** och **v** endast i **vat()**, men inte i **main()**. Ett försök att använda dem i **main()** skulle ge kompileringsfel. Gör gärna testet! En parameter som deklareras i funktionens parameterlista tillhör funktionens huvud och därmed funktionen. Trots detta är parametrarna inte i alla avseenden likställda de lokala variabler som deklareras 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 om-

givningen dvs med andra funktioner. Kom ihåg att vi har kallat dem *formella parametrar* därför att de initieras när funktionen anropas. Dvs de tar emot sina värden från de aktuella parametrarna **amount** och **vatRate** vilka är deklarerade i **main()**. Värdena kopieras över 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.

Globala variabler

Regeln för räckvidden av variabler kan användas bl.a. för att skilja mellan globala och lokala variabler. Avgörande är *platsen* där den deklareras.

Deklareras en variabel i ett block, blir den lokal i det blocket. Deklareras den däremot utanför alla block blir den global. En global variabels räckvidd är obegränsad. Därför gäller den globala variabeln **vatType** i hela programmet och därmed i programmets alla funktioner. Därför gäller **vatType** både i **main()** och **vat()**. Globala variabler tränger genom alla blockgränser och gäller även i alla inre block, så länge deras namn inte används i nya deklarationer.

En annan viktig egenskap är:

Globala variabler initieras automatiskt till **0** om de är tal,

till nolltecknet om de är tecken och till tom sträng om de är strängar.

I vårt programxempel utnyttjas inte denna egenskap. Men generellt är detta av betydelse eftersom lokala variabler beter sig annorlunda: Som vi vet måste de initieras explicit i koden.

Problematiken hos globala variabler

Av vilken anledning har **vatType** deklarerats som global variabel i **VAT_table**? 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 **vat-**

Type 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 **VAT_table**. 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 deklarerad. Men den är inte deklarerad 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 **VAT_table**? 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. Det menas med parametrisering av globala variabler.

7.8 Överskuggning av variabler

När vi pratade om blockstruktur ställde vi upp regeln för *räckvidden* av variabler (sid 195) som i sin tur gav upphov till *lokala* och *globala variabler*. Nu ska vi komplettera våra kunskaper ytterligare med ett koncept som löser namnkonflikter:

Vad händer när lokala och globala variabler har samma namn?

Överskuggning kallas det. På köpet kommer vi att lära känna den s.k. *räckviddsoperatorn*. Följande program demonstrerar dessa begrepp:

```
// Scope.cpp
// Globala och lokala variablers räckvidd (livslängd)
// Överskuggning av variabler och räckviddsoperatorn ::
#include <iostream>
using namespace std;

int x;                          // Global variabel nollsätts
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';
}

void local()                    // Funktion dekarerar egen
{                               // lokal variabel
   int x = 55;                  // Lokal variabel 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 uppdaterar
{                               // global variabel x
   cout << "\nGlobalt x är " << x << " i början av update()"
                                                  << '\n';
   x += dx;
   cout <<"\nGlobalt x är " << x << " vid slutet av update()"
                                                   << '\n';
}
```

Tre variabler med samma namn `x`

Hur många variabler har vi i programmet **`Scope`**? Man ser bara variabeln **`x`**. Men är det en och samma variabel **`x`** i hela programmet eller är det olika variabler med samma namn? Våra kunskaper om lokala och globala variabler avslöjar: Det är *olika* variabler. De refererar till tre *olika* fysiska minnesceller med samma logiska namn **`x`** i koden. Men hur löses namnkonflikten? Vi ska undersöka alla tre fall:

1. Globalt `x`

Ovanpå **`main()`** deklareras variabeln **`x`**. Denna plats kallas för *globalt namnutrymme*, en slags globalt ”block”. Därför gäller **`x`** i hela programmet och därmed i programmets alla underblock. Blocken bildas av funktionerna **`main()`**, **`local()`** och **`update()`**, närmare bestämt av deras anrop. Den globala variabeln **`x`** får i programmet följande värden:

```
Globalt x        0  10  20
```

Som global variabel initieras den automatiskt till **`0`** när den deklareras. 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 det i **`update()`** – till skillnad från funktionen **`local()`** – inte skapas en ny lokal variabel med samma namn. Därför är det det globala **`x`** som används i **`update()`**.

2. Lokalt `x` i `main()`

Sedan har vi en lokal variabel **`x`** i **`main()`** som endast gäller där. Denna lokala variabel har samma namn som den globala variabeln **`x`** från punkt **1**. Den lokala variabeln **`x`** initieras i **`main()`** till **`5`** och ändras aldrig:

```
Lokalt x i main()        5
```

Som gloabal variabel borde den gälla i programmets alla block och därmed även i **`main()`**. Men p.g.a. omdeklarationen med samma namn uppstår en konflikt: Båda kan inte gälla samtidigt. Följande regel löser upp namnkonflikten:

> En lokal variabel i ett block slår ut (överskuggar, eng.: override) en global variabel med samma namn.

Överskuggning (eng. *overriding*) bör inte förväxlas med överskrivning (eng. *overwriting*). Överskriva kan man bara variabelns *värde* med ett nytt värde. Överskuggning har inget att göra med variabelns värde utan med variabelns *giltighet*. I ett lokalt block kastar en lokal variabel med samma namn temporärt (lokalt) en skugga över den globala variabeln.

Bilden med skuggan ska förtyddliga fenomenets temporära karaktär. I det lokala **main()**-blocket gäller gäller den ”egna” lokala variabeln, medan före och efteråt träder den globala variabeln fram ur skuggan och får tillbaka sin fulla giltighet. Hade vi valt ett annat namn för den lokala variabeln i **main()** hade förstås den globala haft full giltighet i **main()**. Men hade vi inte lärt oss överskuggning som är ett viktigt koncept inom programmering som tillämpas inte bara på variabler utan även på funktioner, vilket kommer att tas upp senare.

3. Lokalt x i local()

Slutligen används namnet **x** för att för tredje gången deklarera en variabel **x**, den här gången i funktionen **local()**. Även här överskuggar funktionens lokala variabel **x** programmets globala **x** p.g.a. omdeklarationen och namnvalet. I **local()** initieras **x** till **55**. Sedan ökas det med **1** genom **x++;** så att det blir **56** innan variabeln ”dör” när funktionen **local()** upphör:

Denna minnesbild uppstår i programmet **Scope** två gånger p.g.a. 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 deklarerats 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 4.3 (sid 108) 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. 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 π = 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} = π och omkretsen bli 2π pga 2π **`r`** = 2π.

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 86) 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 **`IncludingVAT`** (sid 187) 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 **`ta-`**

ble() och **table()** anropar **netto()**. Fundera noga på vilka parametrar den nya funktionen **table()** ska ha.

7.6 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;
}
```

7.7 Foljande program kan både kompileras och exekveras:

```
#include <iostream>
using namespace std;
void inner();
double salary, bonus;
string message;

int main()
{
     salary = 60000;
     bonus = salary * 0.20;
     message = " lämplig för bonus";
1 →

     inner();
     message = "Den anställde " + message;
2 →

}

void inner()
{
     double salary = 50000;
     double bonus = 0;
     message = "Andersson inte" + message;
     ::bonus = salary * 0.30;
3 →

}
```

a) Besvara följande frågor teoretiskt, utan att köra programmet ovan. Motivera!

1. Vilka värden har variablerna **salary**, **bonus** och **message** i position (1)?
2. Vilka värden har samma variabler i position (2)?
3. Vilka värden har **salary**, **bonus**, **::bonus** och **message** i position (3)?

b) Lägg in kod i programmet ovan som på ett anändarvänligt och meningsfullt sätt skriver ut de variabelvärden som efterfrågas i a) samt deras positioner. Av din utskrift ska framgå vilka variabler som är lokala, t.ex.:

```
Lokal salary  =  ...
```

Jämför de utskrivna värdena med de värden som du hade kommit fram till i a).

7.8 ***Kalkylatorn (Projekt)*** I denna uppgift ska ett program **`Calculator`** skapas som stödjer följande funktionaliteter: addition, subtraktion, multiplikation, division och potentiering samt att kunna ange det största och minsta av två inmatade tal.

Kalkylatorn ska vara igång kontinuerligt tills användaren väljer att stänga av den, vilket innebär att ni måste lägga in en loop. De olika räkneoperationerna ska definieras i separata funktioner och anropas i **`main()`**.

Följande funktioner ska definieras i programmet **`Calculator`**:

```
double add(double operand1, double operand2)
{
     // Additon av operand1 och operand2
}


double sub(double operand1, double operand2)
{
     // Subtraktion: operand1 - operand2
     // Även subtraktion av negativa tal
}


double mult(double operand1, double operand2)
{
     // Multiplikation av operand1 med operand2
}

double div(double operand1, double operand2)
{
     // operand1 / operand2
     // Division med 0 ska förhindras
     // Felmeddelande vid inmatning av 0 för opernad2
}


double pow(double operand1, double operand2)
{
// Beräkning av potens: operand1 upphöjt till operand2

}
```

```
double max(double operand1, double operand2)
{
    // Returnera det större värdet av operand1 och operand2
}

double min(double operand1, double operand2)
{
    // Returnera det mindre värdet av operand1 och operand2
}
```

Programmet skall exekvera kontinuerligt tills användaren väljer att avsluta körningen. För att åstadkomma detta kan ni exempelvis använda en **do**-sats. Kalkylatorn kan avslutas genom att användaren matar in t.ex. tecknet **q** (quit) istället för en operator.

Placera först, när ni börjar koda, all kod för programmet **Calculator** i en fil. Flytta sedan alla ovannämnda funktioner till en annan fil. Bibehåll endast **main()** som läser in data, anropar funktionerna och skriver ut resultat.

Frivilligt! I slutet, när allt fungerar, försök att lägga in kod som hanterar ev. felaktiga inmatningar. Detta borde inkludera både inmatning av operander, men även av operatorer.

Se till att du skriver ut meningsfulla felmeddelanden, så att användaren får möjligheten att rätta till sin felaktiga inmatning.

7.9 ***Time (Projekt)*** Programmet **Hour2Sec** (sid 86) omvandlar timmar, minuter och sekunder till totalsekunder. Programmet **Sec2Hour** nedan löser det omvända problemet: att omvandla ett givet antal totalsekunder till timmar, minuter och sekunder.

```
// Sec2Hour.cpp
// Omvandlar antal sekunder till timmar, minuter & sek.

#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?

Kapitel 8

Arrays och vektorer

8.1 Vektorer

Begreppet *array* introducerades ganska tidigt i boken (sid 88) och motiverades med att utnyttja en av datorns överlägsna egenskaper, nämligen att kunna lagra och hantera stora datamängder på ett effektivt sätt. Även programmeringstekniskt kunde vi skriva kortare och elegantare kod, när vi i olika sammanhang använde arrays. Programmen blev mer strukturerade och ofta även enklare, t.ex. i projektet *Labyrint* (sid 166). Men vi vet också att array har vissa nackdelar som vi nu vill komma över genom att ersätta den med *vektor*.

Arrayens nackdelar kan sammanfattas i två punkter:

1. Storleken av en array måste alltid anges i koden när arrayen definieras, t.ex.:

   ```
   int no[20];
   ```

   Detta beror på att C++ tillämpar *statisk minnesallokering* för arrays, vilket innebär att kompilatorn allokerar minnesutrymme för **`20`** minnesceller där varje minnescell lagrar ett **`int`**-värde. Storleken på detta minnesutrymme kan inte ändras under exekveringen. Allokeringen sker vid kompilering, inte vid exekvering.

2. En array måste tilldelas *elementvis*, vare sig med eller utan loop. Att detta inte alltid behöver vara så kunde vi se här:

   ```
   Anstalld copy = anst;
   ```

   Samma sak kan man göra med en vektor:

   ```
   vector<int> copy = no;
   ```

   Om **`no`** är en vektor som redan är definierad och initierad, kan man direkt överföra dess värden till en ny vektor **`copy`** som definieras och initieras med satsen ovan.

Vad gäller punkt **1** tillämpar C++ *dynamisk minnesallokering* för vektorer. Storleken på en vektor behöver inte, ja får inte anges på förhand. I den bemärkelsen är alltså en *vektor* en *dynamisk array*.

Vad gäller punkt **2** är denna direkta tilldelning av en vektor inte möjligt med en array. I den bemärkelsen är alltså en *vektor* ett *objekt* vars klass är fördefinierad i ett programbibliotek som måste inkluderas med:

```
#include <vector>
```

Efter denna inledning till vektorer vill vi använda den nya datatypen genom att anknyta till arrays. Vi återupptar arrayens initieringslista i programmet **`ArrayInit`** (sid 92) för att sedan gå över till vektorns initieringslista.

Arrayens initieringslista

```
// ArrayInit.cpp
// Kortform för definition och initiering av en array i en
// och samma sats med initieringslista
// Elementvis tilldelning av en array
#include <iostream>
using namespace std;

int main()
{
   int no[] = {64, 86, -6};        // Definition och initiering
                                   // med initieringslista
   int copy[3];                    // Endast definition
   copy[0] = no[0];                // Elementvis initiering
   copy[1] = no[1];
   copy[2] = no[2];
// copy = no;                      // Ger kompileringsfel!
   cout << "\n\tcopy:s 1:a element copy[0] = " << copy[0]
        << " med index 0\n"
        << "\n\tcopy:s 2:a element copy[1] = " << copy[1]
        << " med index 1\n"
        << "\n\tcopy:s 3:e element copy[2] = " << copy[2]
        << " med index 2\n";
 }
```

Array kräver elementvis initiering. Det direkta initieringsförsöket **`copy = no;`** ger kompileingsfel. 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:e element copy[2] = -6 med index 2
```

Vektorns initieringslista

Vi byter ut i programmet ovan *array* mot *vector*: Först inkluderar vi biblioteket **`<vector>`** i det nya programmet. Sedan skapar vi i **`main()`** vektorn **`no`** och initierar den i samma sats med initieringslistan:

```
vector<int> no = {64, 86, -6};
```

Därmed är vi redo för att skapa en andra vektor **`copy`** och initiera den direkt med den första vektorn **`no`** utan elementvis tilldelning:

```
vector<int> copy = no;
```

```
// VectorInit.cpp
// Definition & initiering av en vektor med initieringslista
// Ingen storlek behövs i förväg: Dynamisk minnesallokering
// Direkt tilldelning av vektorer, inte elementvis
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   vector<int> no = {64, 86, -6};   // Definition och initiering
                                    // av vektorn no med initie-
                                    // ringslista
   vector<int> copy = no;           // Kopiering av no till copy
   cout << "\n\tcopy:s 1:a element copy[0] = " << copy[0]
        << " med index 0\n"
        << "\n\tcopy:s 2:a element copy[1] = " << copy[1]
        << " med index 1\n"
        << "\n\tcopy:s 3:e element copy[2] = " << copy[2]
        << " med index 2\n";
}
```

Programmet **VectorInit** producerar förstås samma utskrift som **ArrayInit**.

8.2 Stränghantering med array

I alla våra programexempel med arrays har vi hittills använt endast arrays av **int**. Nu ska vi gå över till andra datatyper, närmare bestämt till arrays av **char**. Därifrån är det bara ett litet steg till hantering av text. Vi behöver bara bilda arrays av **char** för att representera text i våra program.

Textbehandling är ett klassiskt område för datorisering just p.g.a. datorns förmåga att effektivt och snabbt kunna hantera stora datamängder. Men:

Vad exakt är text i C++ ?

Ett antal tecken, ok. Men var går den övre gränsen? Ett ord, en mening, en rad, en sida, en hel bok eller en sammanställning av innehållet i alla böcker i ett biblitek, ... – 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 lämpar sig det vanliga språkets definitioner och verktyg inte alltid när man vill representera text i datorn. I programmering måste vi använda begrepp som är anpassade till datorns arbetssätt. Ett av dessa begrepp är *sträng*.

Så definieras sträng i C++:

> Sträng = samling av tecken som avslutas med *nolltecknet.*
> Nolltecknet har ASCII-koden 0 och kan i C++ kodas med escapesekvensen **\0**, med **NULL** eller med **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 121). Dessa tecken har olika funktioner i olika språk. I C++ har nolltecknet många funktioner, en av dem är 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, nolltecknet är inte skrivbart. Läsa kan man inte det heller, det är inte läsbart. Faktum är att nolltecknet automatiskt läggs till i slutet av en sträng, när man sätter strängen inom citationstecken **" "**. Sedan kan man spåra det med ASCII-koden **0** eller med escapesekvensen **\0**. Självklart ska man hantera strängar med datatypen **string**. Generellt är **string** stabilare och borde föredras i praktiskt sammanhang. Men då kan man inte längre på en elementär nivå manipulera tecken, vilket vi just vill göra. Dessutom vill vi lära oss mer om arrays och nolltecknet.

Vi föredrar att koda nolltecknet med **\0**, även om det är lika bra att göra det med **NULL** eller med **0**. Därför definierar vi strängar som arrays av **char**.

```
// 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";
}
```

För att definiera och tilldela en **char**-array används en initieringslista (sid 92):

```
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 programmet **ArrayChar** där vi ser stränghantering med array, manipulering med nolltecknet samt strängfunktionen **strcpy_s()** som kopierar strängar.

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 **ArrayChar** 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 på nolltecknets existens i arrayen **text** är att ASCII-koden både till arrayens 4:e tecken och till 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. Lösningen är:

Funktionen `strcpy_s()`

Den fördefinierade funktionen **strcpy_s()** kopierar sin andra parameter till sin första och sätter 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 93). 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**.

Användning av nolltecknet

För att kunna använda nolltecknet (eng. ***the NULL character***) som ett verktyg i koden, måste vi veta, när det exakt läggs till en sträng. Sker det vid definitionen eller tilldelningen av en strängvariabel? Programmet **NULLcharacter** (nedan) kommer att visa att det är vid *tilldelningen* nolltecknet tillkommer. Dessutom lägger alla C++ funktioner som har med strängar att göra, till nolltecknet automatiskt. Så gör t.ex. **strcpy_s()**, **cin.getline()** osv. Men även **string** och **" "**.

I programmet **NULLcharacter** kommer vi att separera definitionen av **char**-arrayen **char namn[20];** från tilldelningen för att kunna mata in vilken sträng som helst. Kortformen **char text[] = "C++";** 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 ty-

värr inte använda kortformen. Detta leder i sin tur till att vi redan vid definitionen måste bestämma oss hur stor arrayen ska vara. 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 variabeln **namn** med **char namn[20];** som en array av **char** med **20** element.

```
// NULLcharacter.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";
   for (int i=0; namn[i-1]!='\0'; i++)   // Skriver ut tecken
       cout << i << "\t\t" << namn[i]    // så länge tecknen
            << "\t\t" << (int) namn[i] <<'\n'; // inte är \0
}
```

char namn[20]; är ett exempel på *statisk minnesallokering*, därför att den redan vid kompilering reserverar ett sammanhängande minnesområde bestående av **20** minnesceller. Storleken på detta minnesområde är oföränderlig under hela programmet. I programmet **NULLcharacter** 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.

Stränginmatning med `cin.getline()`

Varför har vi i **NULLcharacter** 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 85). Därför skulle en **cin**-sats endast läsa in förnamnet och ignorera efternamnet när användaren matar in båda namnen skilda med mellanslag. Därför använder vi metoden **getline()** som är fördefinierad i **cin**:

```
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
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 flödesschema på sid 150.

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 körning 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.

Array i en `for`*-sats*

Här återvänder vi till programmet **`ArrayChar`** (sid 210) och tittar på **`for`**-satsen:

```
for (int i=0; i<=3; i++)
     cout << i+1 << ":a tecknet i \"C++\" är " << text[i] << '\n';
```

Den här **`for`**-satsen producerar följande utskrift:

```
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
```

`for`-satsen skriver ut en rad per varv. Arrayen **`text`** skrivs ut elementvis med koden **`text[i]`** där **`i`** är både **`for`**-satsens räknare och arrayens index.

Tekniken att använda **`for`**-satsens räknare som arrayens index är mycket vanligt. Därför är **`for`**-satsen den mest använda looptypen vid hantering av arrays, därför att antalet varv i **`for`**-satsen är känd i förväg. Samma sak gäller för antalet element i en array. Den logiska slutsatsen blir att koppla **`for`**-satsens räknare till arrayens index för att gå igenom alla element i arrayen. Därför kan vi även i fortsättningen använda denna teknik i våra program:

> Arrayens index = **`for`**-satsens

8.3 Kryptering av text

Vi ska nu dra lite praktisk nytta av våra samlade kunskaper om bl.a. slumptal, ASCII-koder, array och stränghantering, för att med ganska enkla medel skriva en liten applikation om kryptering av text. Följande program läser in en text, krypterar och återställer texten med ett slumptal som krypteringsnyckel. Algoritmen som används för kryptering är väldigt enkel, nästan primitiv, men kan lätt ersättas av mer sofistikerade algoritmer.

```
// EncryptText.cpp
// Krypterar text genom att förskjuta alla tecken i ASCII-
// tabellen med en krypteringsnyckel som slumpas fram i
// ett intervall. Återställer den krypterade texten med
// den inverterade, dvs negativa krypteringsnyckeln.
// Nollteknet styr genomsökningen av text.
#include <iostream>
using namespace std;
#include "MyRand.h"                // Innehåller myRand()

int main()
{
  srand(time(0));
  char text[80];
  cout << "\nMata in en text:\t";
  cin.getline(text, 80);
  int key = myRand(1, 1000);       // Krypteringsnyckeln

  for (int i = 0; text[i] != '\0'; i++)
      text[i] = text[i] + key;     // Kryptering med slump-
  cout << "\nKrypterad  text:\t" << text << "\n\n"; // nyckel

  for (int i = 0; text[i] != '\0'; i++)
      text[i] = text[i] - key;     // Dekryptering med negativ
  cout << "Återställd text:\t" << text << "\n\n"    // nyckel
       << "Krypteringsnyckeln den här gången:  " << key
                                                  << "\n";
}
```

Med en array av **char** allokeras minne för texten med en maximal längd på **80** tecken. För att kunna inkludera mellanslag i texten, läses den in med **cin.getline()** och lagras i arrayvariabeln **text**. Med följande sats krypteras texten:

```
for (int i = 0; text[i] != '\0'; i++)
     text[i] = text[i] + key;
```

Efter denna **for**-loop skrivs den krypterade texten ut. Sedan används **-key**, det negativa värdet av **key**, för att återställa texten som sedan skrivs ut för kontroll:

```
for (int i = 0; text[i] != '\0'; i++)
     text[i] = text[i] - key;
```

Krypteringsmetoden är väldigt enkel: tecknens ASCII-värden ökas med **key** i satsen **text[i] = text[i] + key;** genom vanlig addition. Att det verkligen adderas **key** till *ASCII-koden* till **text[i]** beror på att **text[i]** är av typ **char** och att en teckenvariabel i aritmetiska uttryck tolkas som sin ASCII-kod – ett tal man kan räkna med (sid 123). **for**-satsen som går igenom hela strängen genom att koppla räknaren till arrayens index, gör att hela texten förskjuts med **key** steg i ASCII-tabellen. **key** 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 182. Detta värde som är något heltal mellan **1** och **1000** används som krypteringsnyckel och en andra gång som **–key** för att återställa texten. Genom att ersätta **text[i] + key** med mer sofistikerade formler kan man utveckla mer avancerade krypteringsalgoritmer. Filen **MyRand.h** som innehåller funktionen **myRand()** inkluderas i **EncryptText**.

Programmet **EncryptText** 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 154 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.

Eftersom 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 **`EncryptText`** kan samma algoritm 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 av **`int`** 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. Lägg in egen kod för att ta reda på den aktuellt inmatade **`char`**-arrayens längd.

8.3 Skriv ett program som läser in text i gemener, lagrar den i en array av **`char`** och skriver ut den 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 ***Kryptering av text*** Modularisera programmet **`EncryptText`** (sid 217) genom att flytta krypteringsalgoritmen till *en* funktion med huvudet:

```
void krypt(char t[], int n)
```

Funktionen ska användas både för kryptering och dekryptering. Den formella parametern **`t`** är en array av **`char`** som tar emot den aktuella parametern **`text`** när funktionen anropas. Beakta syntaxen: Hakparentesen får inte innehålla storleken när arrayen används som parameter i en funktion. Storleken skickas vid anropet och får inte anges i funktionen.

Utforma kroppens funktion så att den formella parametern **`n`** kan användas för att ta emot både krypteringsnyckeln **`key`** för kryptering och **`-key`** för dekryptering.

I övrigt ska det nya programmet göra samma sak som det gamla **`EncryptText`**, bara nu med en modulariserad krypteringsalgoritm som kan användas även av andra progran samt vidareutvecklas till mer avancerade algoritmer, oberoende av vilket program den används av.

8.7 Skriv om programmet **ArrayChar** (sid 210) till en **string**-version. Byt ut datatypen **array av char** mot **string**. Gör ändringar i resten av programmet p.g.a. detta byte.

8.8 Skriv om programmet **NULLcharacter** (sid 213) till en **string**-version. Byt ut datatypen **array av char** till **string**. Undersök konskvenserna av denna ändring.

8.9 Skriv ett program som slumpar fram 1 000 heltal mellan 60 och 140 (tänkbara hastigheter på en motorväg), lagrar dem i en array **hastighet**, beräknar och skriver ut deras medelvärde med lämplig förklaring och rubrik.

8.10 ***Labyrint II (Projekt)*** Vidareutveckla din lösning av labyrintprojektet från inlämningsuppgift 2 (sid 166). Förslaget till algoritmen med **if**-satserna som gavs där, var inte precis den mest eleganta lösningen. Nu kan vi utnyttja våra kunskaper om array för att hitta problemets programmeringstekniskt mest eleganta lösning:

Börja med att skapa en array av **char** med 12 element som består av koderna till de 11 linjegrafiktecknen som är avbildade på sid 167 samt mellanslaget. Skriv sedan ut tecknen i en nästlad **for**-loop genom att låta slumpen bestämma arrayens index, t.ex. så här:

```
ch[rand() % 12]
```

där **ch** är namnet på den skapade **char**-arrayen. På så sätt löser en nästlad **for**-loop med en utskriftssats samt en för radbyten hela labyrintuppgiften. Några få rader kod ger den programmeringstekniskt mest eleganta lösningen. Uppgiften är identisk med frivilligdelen av inlämningsuppgift 2 (sid 166).

8.11 ***Master Mind (Projekt)*** är ett spel som låter användaren gissa ett slumpmässigt genererat fyrsiffrigt heltal genom att leda spelaren med en inbyggd hjälpprocedur vars regler beskrivs nedan. Observera att begreppet *siffra* endast omfattar 0, ..., 9, medan *tal* kan vara hur stora som helst. Alla tal skrivs (representeras) med siffror. T.ex. är 12 är ingen siffra utan ett tvåsiffrigt tal. Därför kan man beteckna siffrorna 0, ..., 9 även som ensiffriga tal.

Behandla *fyr*siffriga heltal som en serie av *fyra en*siffriga tal dvs som en array av heltal med fyra element. Kontrollera med en funktion **void input(int guessedNo [])** att endast siffror matas in. Ta hand om felaktig inmatning genom att loopa funktionens anrop i **main()**.

Skriv en funktion med huvudet **void create(int secretNo[])** som ska generera det hemliga fyrsiffriga talet och lagra det i en **int**-array **secretNo** med 4 element. Varje element i arrayen **secretNo** kan genereras som ett slumptal mellan 0 och 9. Funktionen **create()** ska kontrollera spelets regel enligt vilken alla fyra siffror måste vara *olika*.

För att implementera hjälpen till spelaren skriv en funktion med huvudet **`bool help(int guessedNo[], int secretNo[])`** som bearbetar spelarens gissning genom att kontrollera ”rätt siffra” och ”rätt siffra på rätt plats” 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	**`?`**

Är t.ex. det hemliga talet 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.

Genom valet av returtypen **`bool`** till funktionen **`help()`** kan anropet av den direkt sättas in i avslutningsvillkoret av loopar. **`bool`** är en enkel datatyp i C++ som endast kan anta sanningsvärdena **`true`** och **`false`**.

Skriv programmet så att det tillåter flera spelomgångar.

Kapitel 9

Klasser

9.1 Vad är objektorienterad programmering (OOP)?

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

Program = algoritm + data.

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

Program = Modell av verkligheten

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

Paradigmskifte

Det som i programmeringshistorien gjorde att man behövde objektorienterad programmering var den växande komplexiteten hos program under 70-talet. Programmens storlek var avgörande för den växande komplexiteten. Man insåg att det inte längre räckte till att skriva och testa program som fungerade just då. Det var nödvändigt att med rimliga kostnader kunna även *underhålla* stora program, *förnya* och *vidareutveckla* dem så att de fungerade även i flera år och att de framför allt kunde anpassas till nyuppkomna situationer utan oöverkomliga svårigheter. Det i sin tur krävde att man redan i designstadiet behövde ett annorlunda upplägg. Fokuset förskjöts från problemlösning till modellering av verkligheten. Objektorienterad design kom in i bilden. Allt detta var endast med procedural programmering inte längre möjligt. Ett s.k. *paradigmskifte* hade blivit nödvändigt, dvs en ändring av helhetssynen på programmering.

Objekt, klass, datamedlem och metod

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

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

Metoder

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

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

```
minVolvo.kör();
```

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

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

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

```
minVolvo.kör(40);
```

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

Abstraktion

Det är avgörande att skilja mellan *objekt* och *klass*. Vi tar ett annnat exempel: Pepparkakor är objekt vars klass är pepparkaks*formen*. Klassen är alltså en slags mall, en förskrift för produktion av objekt: En enda pepparkaksform kan producera tusentals pepparkaksgubbar. Gubbarna kan skiljas från varandra i vissa detaljer, t.ex. materialet, smaken osv. Man kan t.o.m. måla dem i olika färger eller modifiera på annat sätt efteråt. De förblir pepparkaksgubbar av den ursprungliga formen. I pepparkaksformen ingår det som är gemensamt hos alla pepparkaksgubbar. Man har, när man byggde formen, bortsett från oväsentliga skillnader och tagit hänsyn endast till det väsentliga, det gemensamma hos alla pepparkakor – samma abstraktionprocess som vi kunde observera hos bilar.

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

OOP:s tre hörnstenar

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

Objektorienterad programmering bygger på tre hörnstenar:

- Inkapsling
- Arv
- Polymorfism

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

Inkapsling och klassens konstruktor

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

Arv

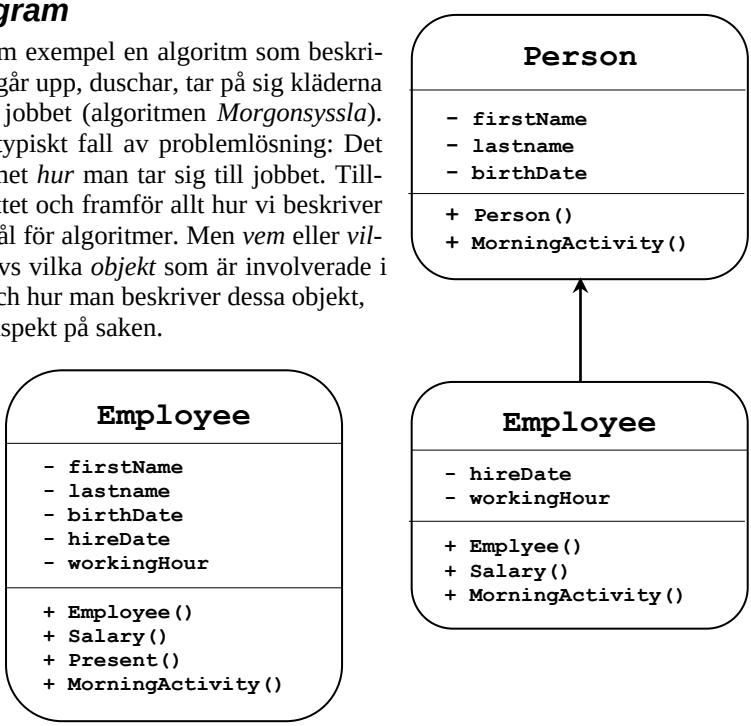
I den reala världen som vi vill efterlikna, finns inga isolerade objekt. Alla objekt är mer eller mindre relaterade till andra objekt. En klok modellering måste dra nytta av de befintliga relationer mellan objekt för att effektivisera och optimera utvecklingsarbetet. En sådan relation är arvrelationen.

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

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

Klassdiagram

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



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

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

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

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

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

Polymorfism

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

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

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

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

9.2 Vägen till objektorienterad programmering

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

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

Men *på vilket sätt* kan vi nu utnyttja det vi lärt oss hittills för att komma in i den nya filosofin? I ett antal steg ska vi exempelifiera att objektorienterad programmeringens rötter finns i procedural programmering, även om i embryonalt tillstånd. *Funktion* är nyckelkonceptet inom procedural programmering. Motsvarande nyckelkoncept inom objektorienterad programmering är *klass*.

Utan modularisering

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

1. Modularisering på funktionsnivå (procedural)

2. Modularisering på klassnivå (objektorienterad)

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

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

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

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

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

Steg 1: modularisering på funktionsnivå (procedural)

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

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

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

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

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

Parametrisering

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

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

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

#include "Procedure.h"                  // Externa moduler
                                        // area() & omkrets()
int main()                              // Modul main()
{
   float radie;                         // Lokal variabel

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

Här är bearbetningsdelen flyttad till de externa *funktionerna* **area()** och **omkrets()** som returnerar de uttryck som i det icke-modulariserade programmet **All_in_main** tilldelades *variablerna* **area** och **omkrets**. Självklart kan man även separera in- och output-delen, men typiskt är att man separerar beräkningar. Nu finns i **main()** kvar input, output samt anropet av **area()** och **omkrets()**. Fokuset vid denna modularisering är på hur man löser problemet med att beräkna cirkelns area och omkrets.

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

Vår första klass

En nackdel av programmet **Procedure** ur modelleringssynpunkt är att **main()**:s lokala variabel **radie** är separerad från funktionerna **area()** och **omkrets()** – ett resultat av modulariseringen. Nackdel, därför att **radie** är ”ur modelleringssynpunkt” relaterad till cirkeln. Samma sak gäller för **area()** och **omkrets()**. Ur problemlösningssynpunkt är det däremot ingen nackdel alls: Man har fått moduler som löser beräkningsproblemet.

Ser man på datorprogram som en modell av verkligheten – i vårt fall på ”verkligheten” *cirkel* – ska man helst avbilda en modell av cirkeln i sitt program. Man ska *beskriva* cirkeln så generellt som möjligt så att koden kan användas på alla tänkbara konkreta cirklar. Denna modellering eller beskrivning av cirkeln måste vara så *modulär* och så *generell* att ingen – inklusive oss själva – skulle behöva återuppfinna hjulet och skriva kod för cirkeln i framtiden. En sådan modul kallas för klassen **Circle**.

I objektorienterad programmering kallar man kod som på ett generellt och modulärt sätt beskriver en kategori av saker och ting i den reala världen, för *klass*. För att kunna skapa klassen **Circle** räcker det inte att flytta funktionerna **area()** och **omkrets()** från **main()** och separera dem från all annan kod, utan även variabeln **radie** som på ett naturligt sätt är en del av modellen **Circle**, borde följa med. Vi måste samla allt som är relevant för alla cirklar i en sådan klass. Följande kod som vi skriver i en headerfil och kommer att inkluderas i programmet **CircleTest** (längre fram) implementerar denna idé.

Steg 2: modularisering på klassnivå (objektorienterad)

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

class Circle                              // Klassen Circle
{
public:
   float radie;                           // Datamedlem

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

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

Om man jämför denna kod med headerfilen **`Procedure.h`** (sid 232) som var ett resultat av modularisering, ser man att det endast är några få rader som har kommit till: Variabeln **`radie`** har flyttats från **`main()`** och deklarerats här och det hela har fått en överordnad ”ram” med **`class`**. Ändå utgör dessa få ändringar ett sådant kvalitativt steg att det innebär övergången från procedural till objektorienterad programmering. Avgörande för det här steget är det reserverade ordet **`class`** som i C++ inleder deklarationen till en klass. Att vi kallar det *deklaration* och inte definition beror på att koden ovan inte reserverar en enda byte minne. Klassdeklarationen ovan beskriver *kategorin* cirkel som en abstrakt idé utan att skapa en verklig cirkel. Den är en *mall* för att skapa verkliga cirklar, en föreskrift om hur en verklig cirkel med en viss radie *skulle* se ut och hur dess area och omkrets *skulle* beräknas *om den skapades*. En verklig cirkel kallas för *objekt*. Det är objektet som behöver minnesutrymme för att lagras. Klassen definierar inga objekt utan ställer bara till förfogande modellen för framtida objektdefinitioner. Om man byter ut cirklar mot pepparkakor kan man säga att pepparkaks*formen* är klassen och själva pepparkakorna är objekten. Formen behöver ingen pepparkaksdeg – motsvarigheten till minne – den framställs bara en gång medan kakorna kan bakas i tusentals. Även klassen skriver man bara en gång, objekt kan skapas hur många som helst.

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

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

Naturligtvis har en cirkel även andra beståndsdelar (egenskaper, attribut) som t.ex. medelpunkt eller, om man vill rita den på skärmen, färgen osv. som man skulle kunna ta in som datamedlemmar i klassen. Men just nu nöjer vi oss för enkelhetens skull med datamedlemmen **`radie`**.

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

Klasskonceptet har förutom modelleringsaspekten följande fördelar i koden:

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

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

Test av klass

Klassen **`Circle`** vore ingenting värt om man inte använde den i ett program. Klassen själv är inget program och kan inte köras. I följande program används klassen för att skapa ett objekt av typ **`Circle`**, läsa in ett värde till det samt anropa klassens metoder **`area()`** och **`omkrets()`**:

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

int main()                                 // Funktionen main()
{
   Circle myCircle;                        // Objektet myCircle

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

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

```
Circle myCircle;
```

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

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

Den här syntaxen för att komma åt ett objekts datamedlem som också förekommer i programmets **cout**-sats för metoderna, kallas *punktnotation* (sid 259).

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

```
Ange radien till en cirkel: 1

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

Klassbegreppet

Här sammanfattar vi våra observationer till *en* definition på *klass*. Läs på sid 253 en *annan* definition.

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

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

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

Objekt och klass

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

Ett av syften med objektorienterad programmering är att efterlikna verkligheten så mycket som möjligt. Man vill kunna avbilda den reala världen, konstruera en modell av den i sina datorprogram för att kunna simulera verkligheten så noggrant som det går. Den reala världen, åtminstone den del som tillåter datorisering, består kort sagt av *objekt* och allt man kan *göra* med dessa objekt. *Hur* man gör och framför allt hur man beskriver det som skall göras, är föremål för algoritmer vilket behandlats tidigare. Men *vad* man sysslar med, dvs *objekt* som är involverade i algoritmerna och hur man beskriver dessa objekt, är en annan aspekt på saken. Objektorienterad programmering prioriterar objektaspekten framför algoritmaspekten. Visserligen kan man skriva **`Circle`**:s data **`radie`** på *ett* ställe och det man kan *göra* med cirkeln, metoderna **`area()`** och **`omkrets()`**, på ett *annat* ställe i sitt program. Men objektorienterad programmering gör inte så, utan sammanför till *en* **`class`** allt som är relevant för en cirkel, allt som är gemensamt för alla cirklar, allt som man skulle ta upp i en ordbok för att definiera *begreppet* cirkel. Man bildar kategorin eller *klassen* **`Circle`** som på så sätt kan användas som modell, som form, som föreskrift, som mall för att i olika sammanhang skapa *objekt* av typen **`Circle`**. På samma sätt kan en enda pepparkaksform (klass) producera tusentals pepparkaksgubbar (objekt). Gubbarna kan skiljas från varandra i vissa detaljer, t.ex. materialet, smaken osv. Man kan t.o.m. måla dem i olika färger eller modifiera på annat sätt efteråt. De förblir pepparkaksgubbar av den ursprungliga formen. I formen ingår det som är gemensamt hos *alla* pepparkaksgubbar. Man har, när man framställde formen, bortsett från oväsentliga skillnader och tagit hänsyn endast till det *väsentliga*, det *gemensamma*.

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

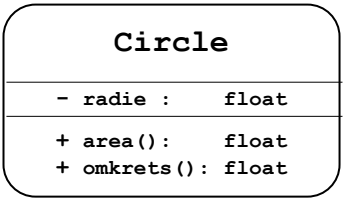
En synonym till objekt är *instans*. En instans av klassen **`Circle`** är ett exemplar av kategorin, av typen **`Circle`**.

Sammanfattningsvis kan vi säga:

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

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

Ett verktyg som har utvecklats speciellt för objektorienterade modelleringar är UML som står för *Unified Modeling Language*. Enligt det här modelleringsspråket skulle vår första klass framställas så här:



I UML-diagram sätts symbolen **–** framför datamedlemmar och **+** framför metoder.

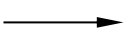
9.3 Inkapsling

Vi återvänder till frågan vi ställde i början av detta kapitel: Vad är objektorienterad programmering? Då besvarade vi den genom att säga att ett program är objektorienterat när det skapas ett objekt i det, eller: ”Först när ett program är modulariserat på klassnivå – där *modul* är en klass – kan det kallas objektorienterat.” (sid 234). Nu ska vi precisera svaret:

Som det nämndes tidigare bygger objektorienterad programmering på tre hörnstenar:

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

Låt oss återvända till vår första klass (sid 234) och lägga märke på en kod som vi hittills inte gått in på närmare, nämligen det reserverade ordet **`public`**:



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

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

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

Låt oss göra följande experiment för att förstå **`public`**:s innebörd: Kommentera bort raden med **`public`** och försök att kompilera klassen **`Circle`** genom att spara **`h`**-filen och kompilera **`cpp`**-filen **`CircleTest.cpp`** (sid 236). Resultatet är 4 kompileringsfel när vi försöker. Det första av dem är:

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

Åtkomstmodifieraren `private`

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

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

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

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

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

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

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

9.4 Konstruktor

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

Klassens konstruktor

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

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

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

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

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

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

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

Klassen **`Circle`** är en ny variant av den tidigare använda klassen **`Circle`** (sid 234). Den nya varianten har en egendefinierad konstruktor Med den vill vi testa egenskaperna **1-3** på förra sidan. Klassens konstruktor är framhävd med vit bakgrund. Som man ser är de två första ovannämnda egenskaperna givna: konstruktornamnet **`Circle`** = klassnamnet och ingen returtyp, inte ens **`void`**, vilket gör att både kompilatorn och vi kan känna igen **`Circle()`** som konstruktor och kan skilja den från klassens andra metoder **`area()`** och **`omkrets()`**. Varje försök att sätta en datatyp eller **`void`** framför metodnamnet kommer att leda till kompileringsfel.

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

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

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

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

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

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

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

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

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

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

I sin struktur liknar satsen ovan det som vi en gång nämnde för enkla datatyper och kallade för objektorienterad initiering (sid 249):

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

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

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

När konstruktorn **`Circle()`** i satsen ovan anropas i **`main()`** importerar den den aktuella parametern **`input`** via sin formella parameter **`r`** in i objektet och tilldelar den till objektets datamedlem **`radie`**, se konstruktorns definition i klassen **`Circle`** (sid 243). På så sätt blir **`radie`** initierad fast den är **`private`**. Konstruktorn tillåter

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

En körning ger följande utskrift:

```
Mata in cirkelns radie: 1.5

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

Default konstruktorn

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

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

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

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

no default constructor exists for class 'Circle'

Felmeddelandet avslöjar att det finns något som heter *default constructor*, men att en sådan inte finns i vår klass **Circle**. Det beror på att vi har definierat en egen konstruktor i **Circle** och den slår ut klassens inbyggda default konstruktorn. Men kommenterar vi bort vår egen konstruktor i klassen **Circle** (sid 243), kan vi kompilera, men får skräpvärden när vi kör. Hur ser denna default konstruktor ut?

> En default konstruktor är en automatisk konstruktor utan parametrar.

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

```
Klassnamn()
{

}
```

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

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

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

> Anropet av default konstruktorn skrivs utan parenteser.

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

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

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

Flera konstruktorer

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

Överlagring (eng. *Overloading*)

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

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

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

class Circles
{
   float radie;
   string namn;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Resultatet blir:

```
Mata in cirkelns radie: 3

Enhetscirkeln med radien 1 har arean 3.14159
                       och omkretsen 6.28318

      Cirkeln med radien 2 har arean 12.5664
                       och omkretsen 12.5664

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

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

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

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

Objektorienterad initiering

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

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

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

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

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

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

som gör exakt samma sak som:

```
int no1 = 5;
```

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

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

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

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

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

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

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

9.5 Accessmetoder

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

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

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

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

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

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

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

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

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

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

#include "Emp.h"                                // Klassen Emp

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

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

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

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

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

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

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

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

9.6 Klass som egendefinierad datatyp

På sid 237 ställdes upp en defnition för klassbegreppet. Här följer en annan:

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

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

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

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

Man får akta sig att härifrån dra slutsatsen att alla datatyper är klasser. T.ex. är alla enkla datatyper inga klasser. Av alla fördefinierade datatyper som vi använt hittills är endast **`string`** en klass i C++. Däremot gäller det omvända: Varje klass definierar en ny datatyp. T.ex. visar satsen **`Circle myCircle;`** i programmet **`CircleTest`** (sid 236) att **`myCircle`** skapas som en **`Circle`**, vilket är en deklaration av variabeln **`myCircle`**. Förutsättning är förstås att man deklaraerat klassen **`Circle`** innan. Om vi sedan pratar om *objektet* **`myCircle`** av *klassen* **`Circle`** eller om

variabeln **myCircle** *av datatypen* **Circle** är bara två olika talessätt för en och samma sak. Men satsen **int no;** som också är både deklaration och deklaration av variabeln **no** skapar inte ett objekt, därför att **int** inte är en klass. Vi har att göra med två olika typer av variabler: enkel datatyp (**no**) och klasstyp (**myCircle**).

Efter enkla datatyper behandlade vi arrays som är sammansatta datatyper. Både array och **class** sammansätter datatyper till en ny enhet. Dock har array vissa begränsningar som inte finns i **class**. En av dem är att man inte kan gruppera element av *olika* typer till en array. En annan är att man inte kan tilldela en array *direkt* utan måste göra det elementvis. Den största begränsningen dock är att man är tvungen att hålla sig till befintliga, fördefinierade datatyper, när man bildar en array. Alla dessa begränsningar faller bort i **class**. Med **class** kan man gruppera *medlemmar* – motsvarigheten till element i array – av *olika* typer, närmare bestämt data av olika typer. Dessutom kan medlemmarna vara metoder som inte finns alls i array. **class** definierar *nya* datatyper.

Tre steg måste tas när man använder **class**:

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

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

1. Deklaration av en klass

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

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

Datatyp är ett namn som vi kan välja fritt med hänsyn till de kända regler och rekommendationer för namngivning samt konventionen att inleda det med en versal. Sedan kan vi använda namnet som datatyp i våra program – med alla de ”rättigheter” som de fördefinierade datatyperna har. Med koden ovan *skapas* den nya datatypen som i sin tur kan skapa objekt. P.g.a. denna speciella styrkan betecknas **class** ofta som *abstrakt datatyp* eller *datastruktur*.

Observera att hela klassdeklarationen efter den avslutande klammern avslutas med semikolon: vi har att göra med en deklarations*sats*. Till skillnad från funktions-

definitioner ersätter den avslutande klammern inte semikolonet: Utelämnas semikolon får man kompileringsfel. Med det semikolonet däremot som står inom runda parenteser (efter *metoder*) menas att det beror på om man skriver en deklaration eller en definition av en metod. Deklaration kräver semikolon, definition inte. I exemplet **`Circle`** (sid 234) hade vi valt att skriva metodernas definition. I nästa exempel kommer vi att deklarera en metod.

I filen **`Employee.h`** definieras den nya datatypen eller klassen **`Anstalld`** som har fyra datamedlemmar **`namn`**, **`personnr`**, **`timlon`** och **`antalTimmar`** där den första är en pekare, den andra en array av **`int`** med **`2`** element, den tredje och fjärde vanliga **`double`**:s. Observera att syntaxen för deklarationen av datamedlemmarna är precis som i vanliga deklarationssatser för variabler. Man ser också att man i en klass – till skillnad från array – kan blanda data av helt olika datatyper, både enkla, sammansatta och andra typer. Man kan t.o.m. ha datamedlemmar som i sin tur är av egendefinierade typer dvs objekt. Sedan har den nya datatypen även en metod **`lon()`** som endast är deklarerad, endast med huvudet. Kroppen definieras separat. Datamedlemmarna och metoderna står inom klamrarna enligt den generella beskrivningen ovan med avslutande semikolon.

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

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

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

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

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

Varför står i klassen **Anstalld** endast deklarationen av metoden **lon()**?

I modulariseringens anda brukar man faktiskt ofta separera metodernas deklaration från deras definition, speciellt i större applikationer för att kunna använda samma deklaration med olika definitioner i olika program. Deklarationen skrivs och lagras i headerfiler, medan definitionen skrivs separat och lagras i en annan headerfil. Detta har praktiska fördelar, när man utvecklar stora program och vill testkompilera sina moduler en i taget.

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

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

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

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

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

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

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

2. Definition av ett objekt

När en klass definierar en ny datatyp kallas den nya datatypen för en *klasstyp*. Objekt av denna klass kan då anses som *variabler av klasstyp*. Att definiera ett objekt är således samma sak som att definiera variabler av klasstyp.

Följande program demonstrerar definition av objekt genom att definiera variabler av klasstypen **Anstalld**:

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

   Anstalld copy = anst;                 // Kopierar objekt

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

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

   cout << "\n\t" << "Anders Larsson" << ", personnr "
        << "590714-2493"
        << "\n\n\t" << "Gammal lön:\t" << anst.lon()
        << "\n\n\t" << "Ny lön:\t\t"   << copy.lon()
        << "\n\n\tVåra lönekostnader kommer att öka med "
        << copy.lon() - anst.lon() << " kr\n\n";

   cout << "Klassen Anstalld föreskriver "<< sizeof(Anstalld)
        << " bytes för sina objekt, därför \natt den"
        << " sammansätter datatyperna: string med "
        << sizeof(string) << " bytes\n\t\t\t\t"
        << "  float  med  " << sizeof(float) << " bytes\n";
}
```

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

```
Anstalld anst;
```

som kan jämföras med:

```
int a;
```

Här kan man direkt se analogin mellan *datatyp* och *klass*. Datatypen **int** skapar variabeln **a**. Då den är fördefinierad i C++ behöver vi inte göra det. Satsen allokerar minnesutrymme åt variabeln **a**, där **int** föreskriver storleken. På samma sätt skapar datatypen **Anstalld** variabeln **anst**. Då den inte är fördefinierad har vi definierat den på sid 255. Satsen **Anstalld anst;** allokerar minnesutrymme åt variabeln **anst**, där **Anstalld** föreskriver storleken. Ändå är den enkla datatypen **int** ingen klass, men klassen **Anstalld** är en datatyp som har exakt samma ”rättigheter” som vilken annan datatyp som helst. Generellt gäller:

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

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

```
Anstalld copy = anst;
```

precis som hos vanliga variabler: `int b = a;`

Tidigare hade vi introducerat denna teknik för enkla datatyper, vilket inte kunde utvidgas till arrays då de måste tilldelas elementvis. Men nu återupptas den röda tråden. En förutsättning för satserna ovan är förstås att variablerna höger om tilldelningstecknet är definierade, vilket räcker till för kompilering. Om **anst** och **a** dessutom är initierade kommer de vid exekvering att föra över sina värden till **copy** och **b**. Prova gärna genom att bortkommentera datamedlemmarnas initiering.

Datatyptest med `sizeof`

När man pratar om att skapa ett *objekt* av *klassen* **Anstalld**, då är det samma sak som att skapa en variabel av *datatypen* **Anstalld**. I båda fall måste *minnesutrymme reserveras* av den storlek som klassen resp. datatypen föreskriver. Detta för att kunna lagra *värden* i objektet, för det är det enda som är praktiskt relevant: en variabels eller ett objekts *existens* i programmet är identisk med motsvarande minnesutrymmes existens i datorns RAM.

För att få information om hur mycket minne t.ex. ett objekt av typ **Anstalld** behöver, måste vi titta – och det gör även kompilatorn – i klassen **Anstalld**:s deklaration (sid 255). Där finns två datamedlemmar av typ **string**: **namn** och **personnr** som var och en tar **40** bytes, samt två **float**:s med **4** var, så att sammanlagt 2 x 44 = **88** bytes allokeras åt objeket **anst**. Denna minnesstorlek mäter vi med operatorn **sizeof** som returnerar antalet bytes operanden tar i minnesutrymme. Med **sizeof** kan vi visa att klasser är egendefinierade datatyper genom att skriva dem som operander i **sizeof**. I programmet **EmployeeTest** gör vi det med:

```
sizeof(Anstalld)
```

Lika bra skulle vi kunna skriva:

```
sizeof(anst)
```

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

```
        Anders Larsson, personnr 590714-2493

        Gammal lön:     21450

        Ny lön:         24667.5

        Våra lönekostnader kommer att öka med 3217.5 kr

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

Detta visar än en gång att klasser är datatyper, sammansatta av alla möjliga datatyper (enkla, arrays, pekare, …) och objekt är variabler av klasstyp. Även klasser kan vara datamedlemmar i en annan klass. Då pratar man om nästling eller *komposition av klasser* vilket kommer att behandlas senare i detta kapitel. Värdet **88** bytes som returneras av **sizeof** visar att i det här exemplet storleken som klassen **Anstalld** föreskriver för sina objekt är lika med summan av storlekar av klassens datamedlemmar. Men generellt gäller att objektets storlek är *mindre än eller lika med* summan av alla datamedlemmars storlekar. Dvs C++ kompilatorn reserverar ibland mer minne, vilket har att göra med hanteringen av ordlängder i RAM.

3. Åtkomst till objektets medlemmar

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

Punktnotation

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

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

```
anst.namn         = "Anders Larsson";
anst.personnr     = "590714-2493";
anst.timlon       = 110;
anst.antalTimmar = 190;
```

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

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

Då ser punktnotation ut så här:

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

Till vänster om punkten måste alltid finnas namnet på ett objekt och till höger någon datamedlem tillhörande *detta* objekt. Punktnotation skrivs för att *referera* till just detta objekts datamedlem och kan därför användas antingen för att tilldela den ett värde (skriva till minnescellen) eller för att hämta värdet (läsa från minnescellen). Satserna ovan är rena tilldelningssatser, medan punktnotationen som står i programmets första **cout**-sats hämtar värdena och skriver ut dem.

Vid sidan av punktnotation finns det andra sätt att initiera objekt direkt vid skapandet. Ett av dem är konstruktorn som vi behandlat tidigare.

Anrop av metoder

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

```
anst.lon()
```

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

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

Körresultatet av programmet **EmployeeTest** visar att metoder inte allokerar minnesutrymme i objektet. När objektet skapas allokeras minne endast för datamedlemmar, inte för metoder. De är bara *deklarerade* i klassen och deklarationen skapar inget minne. Först när funktionen anropas, allokeras minne åt de parametrar

och variabler som är involverade i metoden. Men detta sker inte i objektet utan i det program som anropar metoden. En närmare titt på metoden **`lon()`**:s definition (sid 256) visar att **`lon()`** inte har några parametrar. Men den har en lokal variabel **`overtid`** som definieras i metoden, dvs skapas vid varje anrop och ”dör” direkt efter anropet. Dessutom involverar metoden **`lon()`** datamedlemmarna **`timlon`** och **`antalTimmar`** som vid anropet tas från objektet **`anst`**. Därför säger vi att metoden **`lon()`** anropas *i* objektet **`anst`** och har därmed direkt tillgång till datamedlemmarna. Det är också därför de får skrivas i metoden **`lon()`**:s kropp utan punktnotation. Båda befinner sig inuti objektet och har tillgång till varandra direkt. De är medlemmar i samma klubb – ”insiders” så att säga – och kan därför hälsa varandra utan att ange klubbens namn. Även om de hade förekommit i parameterlistan hade de angetts utan punktnotation. Punktnotation måste och får användas endast utanför objektet.

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

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

9.7 Metoder i OOP

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

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

Exempel på egendefinierade metoder är **lon()** definierad i klassen **Anstalld** samt metoderna **area()** och **omkrets()** i klassen **Circle**. Fördefinierade metoder som vi hittills använt i våra program utan att definiera dem själva, har vi haft exempel på: **_getch()** definierad i **conio**, **setprecision()** i **iomanip** osv. Det gemensamma hos alla dessa metoder var att de hade endast parametrar och returvärden av *enkla datatyper*. För att studera metodernas objektorienterade egenskaper ska vi nu ta upp ett exempel där en metod både tar in ett *objekt* som parameter och returnerar ett *objekt*.

Objekt som parameter och returvärde

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

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

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

Två programmeringstekniskt nya moment kan observeras här:

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

Punkt **1** följer av egenskapen att objekt kan skrivas överallt i programmet där även en vanlig variabel kan stå (sid 258). Variabler har hittills varit det vanliga som parametrar och returvärden. Så, varför inte objekt? Från applikationens synpunkt verkar det vara naturligt att restider kommer in som summander (input) i en algoritm som summerar tider och att även resultatet dvs summan av två restider blir en restid (output) eller åtminstone en tid, dvs ett objekt bestående av datamedlemmarna **`tim`** och **`min`**.

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

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

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

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

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

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

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

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

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

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

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

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

Här skapas först två objekt **tisdag** och **onsdag** av typ **Restid** och initieras genom att läsa in värden till deras datamedlemmar **tim** och **min**. Att skapa och initiera objekt i två separata steg är inte optimalt, men vi gör det än så länge tills vi i

nästa avsnitt lär oss att skriva objektets definition och initiering i en enda sats. När objekten **tisdag** och **onsdag** är väl definierade adderas de i följande anrop av metoden **sum()**:

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

analogt till:

```
tisdag "+" onsdag
```

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

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

Det kan vi göra därför att metoden **sum()** enligt definition (sid 262) returnerar ett objekt av typ **Restid**. Här sker definitionen och initieringen av objektet **tvadagsresa** i en enda sats precis som man definierar och initierar vanliga variabler i en enda sats. Det är möjligt, därför att **tvadagsresa** tar emot ett helt objekt: returvärdet av **sum()**.

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

1. Upprepat eller kedjeanrop.
2. Nästlat anrop.

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

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

där **torsdag** är ett **Restid**-objekt som skapats och initierats innan. Vi anropar **tvadagsresa**-objektets **sum()**-metod för att lägga till den första summan som hade bildats vid **sum()**:s första anrop, **torsdag**ens restid. Den fösta summan hade adderat **tisdag**ens och **onsdag**ens restider och lagt resultatet i **tvadagsresa**. Nu adderar vi **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 9

Besvara följande frågor om objektorienterad programmering (3.1):

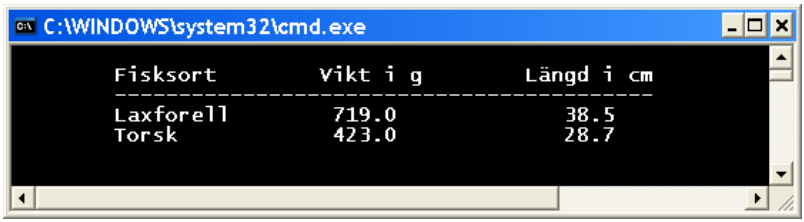
- Vad menas med *paradigmskifte* i programmeringens historia?
- Mellan vilka två programmeringsspråk går historiskt skiljelinjen mellan procedural och objektorienterad programmering? När ungefär inträffade övergången?
- Vad var anledningen till paradigmskiftet inom programutveckling?
- Vilka för- och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

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

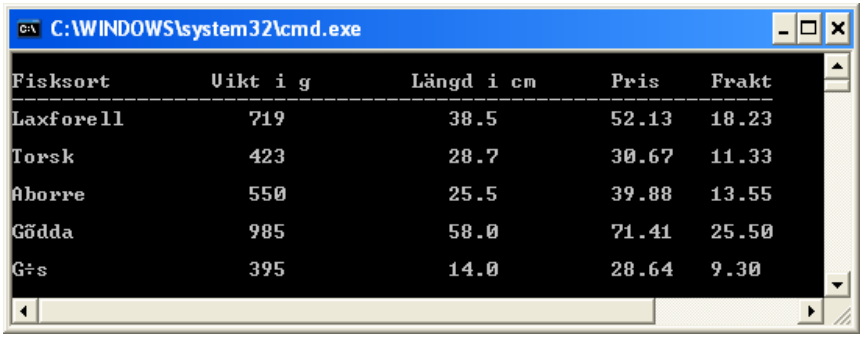
9.2 Modifiera övn 3.1 genom att *läsa in* värden till datamedlemmarna. Efter utskriften av area och omkrets, fördubbla rektangelns längd och bredd. Skriv ut en gång till rektangelns area och omkrets. Med vilken faktor växer arean resp. omkretsen?

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

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



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



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

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

9.7 Vidareutveckla programmet **`TravelTest`** (sid 264) genom att ersätta de tre **`Restid`**-objekten **`tisdag`**, **`onsdag`** och **`torsdag`** med en *array av objekt* med 3 element:

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

Använd samma klass **`Restid`** (sid 262) för att låta programmet göra samma sak som **`TravelTest`**.

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

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

Mata in timmar och minuter till varje veckodags restid. Använd klassen **`Restid`** (sid 262) och en **`for`**-loop för att summera veckodagarnas restider och skriva ut veckans totala restid.

Kapitel 10

Filhantering

10.1 Att skriva till och läsa från filer

Alla våra program hittills har en sak gemensam: Så snart vi avlutat programkörningen försvinner all data från datorn, närmare bestämt från RAM – utom källkoden som ligger på hårddisken. Vi kommer inte längre åt varken programmets in- eller output efter exekveringen. Anledningen är att källkoden laddas från hårddisken till RAM när vi startar körningen och att programmets alla variabler samt in- och utdata allokeras och bearbetas i RAM. När körningen avslutas, ”dör” programmets data i RAM. Ska utdata användas efteråt, måste den under körningen skrivas ut till filer. Samma sak gäller för indata vars mängd kanske är så stor att den praktiskt taget inte kan matas in från tangentbordet, utan måste läsas in från filer. På så sätt kan filhantering i koden bli en nödvändighet.

Innan vi konkret kan inse denna nödvändighet ska vi lära oss grunderna i filhantering. Vi börjar med följande enkelt program:

```
// WriteReadFile.cpp
// Skapar filen WriteRead.txt i projektmappen eller öppnar
// den om den redan finns och skriver över innehållet.
// Skriver text från programmet till filen.
// Läser innehållet från samma fil & skriver det till skärmen
#include <iostream>
#include <fstream>                  // Innehåller ofstream
using namespace std;                //         och ifstream

int main()
{
   char letter;                     // Objekt av typ ofstream
   ofstream fileForWrite("WriteRead.txt");  // initieras till
                                    // filen för skrivning
   fileForWrite << "\n\tDenna text finns i filen "
                << "WriteRead.txt\n";// Text skrivs till filen
   fileForWrite.close();

   ifstream fileForRead("WriteRead.txt");     // Objekt av typ
            // ifstream initieras till samma fil för läsning
   cout << "\nTexten har skrivits till filen."
                << "\n\nNu läses den från filen: \n";
   while (!fileForRead.eof())       // Så länge filslutstecknet
   {                                // inte är nått ska tecknen
      fileForRead.get(letter);      // läsas från fileForRead
      cout << letter;               // och skrivas till skärmen
   }
   fileForRead.close();
}
```

Bland allt nytt som finns i programmet ovan låt oss börja med:

```
#include <fstream>
```

Klasserna `ifstream` *och* `ofstream`

Dessa klasser lagras i biblioteksfilen **`fstream`** som står för *file stream*.

`ofstream` står för ***output*** *file stream* och **`ifstream`** för ***input*** *file stream*.

Vad betyder här in- och output? Utgångspunkten för att bestämma ”riktningen” av out- och input är alltid *C++ programmet*. Dvs:

Output = utdata från programmet till en fil, för att *skriva* till filen.

Input = indata från en fil till programmet, för att *läsa* från filen.

Därför:
```
ofstream fileForWrite("WriteRead.txt");
ifstream fileForRead("WriteRead.txt");
```

Att skriva till en fil

I programmet **`WriteReadFile`** definieras i satsen

```
ofstream fileForWrite("WriteRead.txt");
```

objektet **`fileForWrite`** av klassen **`ofstream`**, dvs en filtyp som är avsedd för output, dvs för att skriva till den. Till en sådan fil kan man endast skriva data från programmet, inte omvänt.

Parentesen **`("WriteRead.txt")`** efter objektet **`fileForWrite`** är anropet av konstruktorn: objektet skpas och initieras samtidigt till filen **`"WriteRead.txt"`**. Dvs ett *logiskt* filnamn **`fileForWrite`** kopplas till det *fysiska* filnamnet **`WriteRead.txt`**, en fil som antingen redan finns eller skapas på hårddisken. Observera att filobjektet **`fileForWrite`** tar emot en sträng som värde. Därför måste filnamnet skrivas inom citationstecken. Det som kompilatorn gör är att söka i projektmappen efter en fil med detta namn. Om den finns kommer **`ofstream`**-satsen att radera filens innehåll utan förvarning när programmet **`WriteReadFile`** exekveras. Samtidigt sätts filens markör i början av den tomma filen, redo för att skriva i den. Om filen inte finns kommer satsen att skapa en fil med namnet **`WriteRead.txt`**, sätta markören i början av filen, redo för att skriva i den.

Filobjektet **`fileForWrite`** används sedan för att skriva till filen med:

```
fileForWrite << "\n\tDenna text finns i filen...";
```

För första gången står nu utmatningsoperatorn **`<<`** inte efter **`cout`** utan efter filobjektet **`fileForWrite`**. Om man tolkar **`<<`** som en pil från höger till vänster innebär detta att data strömmar i pilens riktning till filen **`fileForWrite`**. Precis som i satsen **`cout << data;`** där **`data`** strömmar i pilens riktning till datorns standard output-enhet dvs bildskärmen. Nu går dataströmmen till en fil istället.

Slutligen stängs filen efter **`for`**-satsen med **`fileForWrite.close();`** Den explicita stängningen av filen är av betydelse då den sätter *filslutstecknet* som är avgö-

rande för filens korrekta återanvändning. När man t.ex. senare vill läsa från filen används ofta en loop vars avslutningskriterium är just detta filslutstecken som representeras på olika sätt i olika operativsystem, t.ex. *ctrl-z* i Windows och *ctrl-d* i Unix. I C++ tar funktionen **`eof()`** reda på om filslutstecknet är nått eller ej.

Att läsa från en fil

I programmet **`WriteReadFile`** definieras i satsen

```
ifstream fileForRead("WriteRead.txt");
```

objektet **`fileForRead`** av klassen **`ifstream`** – som en filtyp för input – dvs för att läsa från filen. Samtidigt initieras filobjektet till **`"WriteRead.txt"`** – samma fil som i programmets första del. Markören sätts i början av filen, redo för att läsa från den. Lägg märke till att det måste användas en annan klass för inläsning än för skrivning då operationerna för inläsning är definierade i **`ifstream`** medan de för skrivning finns i **`ofstream`**. Vi använder funktionen **`eof()`** som står för *end of file* och är definierad i klassen **`ifstream`** för att avsluta **`while`**-loopen som både läser från filen **`WriteRead.txt`** och samtidigt skriver det lästa till skärmen:

```
while (!fileForRead.eof())
{
   fileForRead.get(letter);
   cout << letter;
}
```

Så länge filslutstecknet *inte* är nått, ska **`while`**-loopen fortsätta. När det är nått ska den avslutas. Den logiska operatorn NEGATION **`!`** kan vi sätta framför anropet därför att **`eof()`** returnerar ett sanningsvärde av typ **`bool`**. Den fördefinierade funktionen **`eof()`** returnerar **`true`** när den påträffar filslutstecknet annars **`false`**. Så länge **`eof()`** returnerar **`false`** ska **`while`**-loopen leda dataströmmen från filen **`fileForRead`** till teckenvariabeln **`letter`**. Detta är innebörden i satsen **`fileForRead >> letter;`** där pilen går från vänster till höger. Precis som i satsen **`cin >> letter;`** där dataströmmen också går i pilens riktning från datorns standard input-enhet, tangenbordet, till variabeln **`letter`**. Nu kommer data från filen **`fileForRead`** istället och lagras i teckenvariabeln **`letter`**. En körning ger:

```
Texten har skrivits till filen.

Nu läses den från filen:

        Denna text finns i filen WriteRead.txt
```

Sedan kan man kolla att utskriftens tredje rad även finns i filen **`WriteRead.txt`**.

10.2 Append mode

Programmet **WriteReadFile** börjar med att skriva till filen, med följande sats:

```
ofstream fileForWrite("WriteRead.txt");
```

Om filen **WriteRead.txt** redan finns i projektmappen raderar satsen ovan filens innehåll utan förvarning varje gång programmet exekveras. Vill man inte ha det så, utan önskar att filens gamla innehåll bibehålls och det nya kommer till som ett tillägg, kan man med följande ändring åstadkomma detta:

```
ofstream fileForWrite("WriteRead.txt", ios::app);
```

Ändringen, dvs tillägget av den 2:a parametern **append:true** i konstruktorns parameterlista gör att filen **WriteRead.txt** öppnas i s.k. *append mode* vilket innebär att man kan lägga till data i filen utan att radera befintlig data. Den syntax som används för konstruktorns 2:a parameter är ny för oss:

```
ios::app
```

Detta ändrar helt och hållet filskrivningens beteende: Markören sätts inte i början utan i slutet av filen. Filens gamla innehåll överskrivs inte utan sparas. Märkören lägger till ny text till den gamla. Följande program testar detta beteende:

```
// AppendFile.cs
// Öppnar filen WriteRead.txt som skapades i programmet
// WriteReadFile utan att radera filens gamla innehåll
// Lägger till text från programmet till filen, läser sedan
// innehållet från samma fil och skriver ut det på skärmen.
#include <iostream>
#include <fstream>              // Innehåller ofstream
using namespace std;            //        och ifstream

int main()
{
   char letter;                // Objekt av klassen ofstream
   ofstream fileForWrite("WriteRead.txt", ios::app);
                               // Lägger till ny text till
   fileForWrite << "\n\tDenna text har lagts till filen "
                << "WriteRead.txt.\n"; // Bibehåller filens
   fileForWrite.close();               // gamla innehåll

   ifstream fileForRead("WriteRead.txt");     // Läsning
   cout << "\n\tFöljande text har skrivits från "
        << "programmet till filen.\n\n\t"
        << "Nu läses den från filen:\n";

   while (fileForRead.get(letter))    // Läses från fil och
      cout << letter;                 // skrivs på skärmen
   fileForRead.close();
}
```

Resultatet är följande:

```
Följande text har skrivits från programmet till filen.

Nu läses den från filen:

Denna text finns i filen WriteRead.txt.

Denna text har lagts till filen WriteRead.txt.
```

Den sista raden har kommit till i och med exekveringen av programmet **AppendFile** medan raden ovan härstammer från programmet **WriteReadFile**.

10.3 Slumplösenord i fil

Kalle som är systemadministratör önskar att få en färdig lista över ett antal användarnamn samt lösenord för att dela ut konton till sina användare. Därför skickar han följande uppdrag till oss:

Uppgiften:

> ”Skriv ett program som skriver ut två kolumner. I den första ska stå några användarnamn som t.ex. **user1**, **user2**, … . I den andra ska till varje användare stå ett slumpvis genererat lösenord med 6 tecken: 4 små bokstäver, 1 siffra och 1 specialtecken. Programmet ska båda skriva till en fil och visa filens innehåll.”

Lösningen:

```
// RandPasswTest.cpp
// Skapar en fil, skriver i den ett antal användarnamn och
// slumpvis genererade lösenord med funktionen randPasswd()
// Läser sedan från samma fil och skriver ut innehållet
#include <iostream>
#include <fstream>                  // Innehåller klasserna
using namespace std;                // ofstream och ifstream
#include "randPasswd.h"             // Innehåller randPasswd()
int main()
{
   srand(time(0));
   char password[7], letter;        // 6 tecken + nolltecknet
   int antal;
   cout << "\nHur många användarnamn med lösenord "
        << "vill du ha? ";
   cin >> antal;
   ofstream fileForWrite("userPasswd.txt");

   for (int i=1; i<=antal; i++)      // Skriver tabellen
   {
      randPasswd(password);          // Anrop av randPasswd()
      fileForWrite << "\tuser" << i // Skriver till filen
                   << "\t\t" << password << '\n';
   }
   fileForWrite.close();

   ifstream fileForRead("userPasswd.txt");
   cout << "\nVarsågod, detta står nu"
        << " i filen userPasswd.txt:\n\n";

   while (fileForRead.get(letter))     // Läser från filen och
      cout << letter;                  // skriver på skärmen
   fileForRead.close();                // så länge det finns
}                                      // tecken i filen
```

Skrivning till filen

Programmet **RandPasswTest** kopplar filobjektet **fileForWrite** till den fysiska filen **"userPasswd.txt"**. Sedan skriver det till filen med följande sats:

```
fileForWrite << "\tuser" << i
             << "\t\t" << password << '\n';
```

Satsen är inbyggd i en **for**-sats där räknaren **i** går från **1** till **antal** användare man matar in vid körning

Läsning från filen

I **ifstream**-satsen definieras objektet **fileForRead** till en filtyp för input och initieras till samma fil som vi skrev till. För läsning används här samma metod som i förra programmet, nämligen funktionen **get()** som är definierad i datatypen **ifstream**. Denna funktion anropas i villkoret till en **while**-loop:

```
while (fileForRead.get(letter))
   cout << letter;
```

för att läsa filen **fileForRead** tecken för tecken så länge det finns data i den dvs tills funktionen **get()** träffar på filslutstecknet. Funktionen **get()** gör två saker:

1. Hämtar ett tecken i taget från filen och tilldelar den till sin parameter **letter**.

2. Returnerar **true** om det hämtade tecknet *inte är* filslutstecknet och **false** om det hämtade tecknet *är* filslutstecknet. Loopens avslutningskriterium är alltså implicit inbyggd i funktionen **get()** som vid varje anrop läser ett tecken från filen och därmed inbakad i en loop läser hela filen. I **while**-loopens kropp skivs sedan de hämtade tecknen ett i taget till skärmen.

Funktionen `randPasswd()`

Följande funktion som anropas i programmet **RandPasswTest** i **for**-satsen, genererar slumplösenorden.

```
// randPasswd.h
// Skapar slumpvis genererade lösenord best. av av 6 tecken
// genom att anropa funktionen myRand() i olika intervall av
// ASCII-tabellen enligt Kalles lösenordpolicy:
// 4 små bokstäver, 1 siffra, 1 specialtecken
#include "myRand.h"                    // Innehåller myRand()
void randPasswd(char p[])
{
   for (int i=0; i<4; i++)
       p[i] = myRand(97, 122);         // 4 små bokstäver
   p[4] = myRand(48, 57);              // 1 siffra
   p[5] = myRand(33, 47);              // 1 specialtecken
   p[6] = '\0';                        // Gör p till en sträng
}
```

Funktionen **randPasswd()** tar emot som parameter arrayen **p** av **char** och tilldelar i en **for**-sats dess 4 första element tecken som slumpvis tas ur ASCII-intervallet **(97, 122)**. En blick i ASCII-tabellen (på nästa sida) visar att det är tecknen **a, b, c, ..., z** dvs det engelska alfabetet i små bokstäver. Efter anropet av funktionen **randPasswd()** sparas användarnamn och lösenord i filen.

ASCII-tabellen

	0	1	2	3	4	5	6	7	8	9
0	null	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	″	#	$	%	&	′
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

Det engelska alfabetet finns sammanhängande i ASCII-tabellen. Därför kan den här tilldelningen göras med en **for**-sats. Det 5:e elementet – med index **4** – tilldelas slumpvis ett tecken ur intervallet **(48, 57)**, det är siffrorna **0-9**. Det 6:e elementet – med index **5** – tilldelas något av specialtecknen i funktionen angivna ASCII-intervallet. I alla intervall ingår även gränserna, eftersom funktionen **myRand()** som anropas här flera gånger, även inkluderar intervallgränserna. Slutligen sätts nolltecknet som strängavslutningstecken på arrayens 7:e element – med index **6** – för att göra **char**-arrayen till en sträng. Funktionen **randPasswd()** anropas i programmet **RandPasswTest** i den **for**-sats som skriver till filen. Därvid skickas parametern **password** som är en **char**-array av längden 7. I funktionen initieras arrayen med hjälp av **p**. Efter anropet är den även initierad i **main()** pga referensanrop. Så hamnar innehållet – ett slumplösenord av 4 små bokstäver, 1 siffra och 1 specialtecken – i filen.

Det ursprungliga målet var ju att skriva en lista över användarnamn och lösenord till filen **userPasswd.txt** för att dela ut konton. Lösenorden kan initialt vara vad som helst, bara de följer en policy med vissa säkerhetskrav. Sedan kan användarna efter den första inloggningen själva bestämma sina individuella lösenord.

Ett körresultat kan se ut så här:

```
Hur många användarnamn med lösenord vill du ha? 12

Varsågod, detta står nu i filen userPasswd.txt:

        user1           wfdx6*
        user2           dgjt9+
        user3           eirb9"
        user4           wyyc5$
        user5           mgro9!
        user6           xlmx1"
        user7           pwtq2&
        user8           wvzl4+
        user9           onew6+
        user10          yxnk1(
        user11          bsnq6#
        user12          qafn9$
```

Samtidigt skapas filen **`userPasswd.txt`** på hårddisken i samma mapp som **`cpp`**-filen med ovanstående listan över 20 användarnamn och lösenord som innehåll.

Vill man placera filen på en annan plats på hårddisken, måste i den sats som skapar filen, sökvägen till denna plats anges:

```
ofstream fileForWrite("C:\\ ... \\userPasswd.txt");
```

Sökvägens syntax är plattformsberoende. Den måste börja med diskens enhetsbokstav om man väljer absoluta sökvägar. Men även relativa sökvägar av typ **`..\\userPasswd.txt`** där **`..`** betyder en nivå uppåt i mappstrukturen, är möjliga. Då placeras filen t.ex. i mappen strax ovanför den aktuella mappen. Självklart borde samma sökväg anges senare i programmet i den sats som läser filen. Anledningen till användningen av **`\\`** i sökvägen är att **`\`** är reserverad för escapesekvensernas inledningssymbol. För själva tecknet **`\`** inom en sträng måste escapesekvensen **`\\`** användas.

I funktionen **`randPasswd()`** (sid 277) anropas samma funktion som användes tidigare, för att skapa slumptal i de önskade intervallen:

```
// myRand.h
// Funktion som returnerar ETT slumptal i
// heltalsintervallet[a, b]

int myRand(int a, int b)
{
    if (a < b)
        return a + rand() % (b - a + 1);
    else
        return b + rand() % (a - b + 1);
}
```

10.4 Kryptering av filer

Tidigare behandlades kryptering av text. De verktyg som utvecklades där kan med fördel användas för att kryptera även filer nu när vi lärt oss filhantering. För avväxlingens skull presenterar vi först körresultatet av ett filkrypteringsprogram och går igenom koden sedan på nästa sida:

```
Okrypterad fil:
Denna text kommer från en fil som heter Okrypterad.txt.
C++ programmet EncryptFile läser den från hårddisken, krypterar den och skriver den krypterade texten i filen Krypterad.txt. För att testa krypteringen återställer programmet texten och skriver den återställda texten i filen Återställd.txt.

Krypterad fil:
Æ|╝╝»n┬|ã┬n╣¢╗╗ | └n┤ └3╝n|╝n┤À‖n┴¢╗nÂ|┬| └nØ╣ └Ã¥┬| └»▓|┬ã┬|Xæyyn¥
└¢Á└»╗╗ |┬nààÖ└Ã¥┬öÀ
‖n‖2┴| └n▓|╝n┤ └3╝nÃ3└▒▒À┴╣ |╝zn╣ └Ã¥┬| └» └X▓|╝n¢ÂÀn┴╣ └À─| └n▓|╝n╣ └
Ã¥┬| └»▓|n┬|ã┬|╝nÀn┤
À‖ |╝nÖ└Ã¥┬| └»▓|┬ã┬|XöD└n»┬┬n┬|┴┬»n╣ └Ã¥┬| └À╝Á|╝n3┬| └┴┬2‖‖ | └n¥└
¢Á└»╗╗ |┬n┬|ã┬|╝n¢▒Â
nX┴╣ └À─| └n▓|╝n3┬| └┴┬2‖‖▓»n┬|ã┬|╝nÀn┤À‖ |╝n!!┬| └┴┬2‖‖▓|┬ã┬|X

Återställd fil:
Denna text kommer från en fil som heter Okrypterad.txt.
C++ programmet EncryptFile läser den från hårddisken, krypterar den och skriver den krypterade texten i filen Krypterad.txt. För att testa krypteringen återställer programmet texten och skriver den återställda texten i filen Återställd.txt.

Krypteringsnyckeln:     78
```

Det här är bara ett av flera möjliga körresultat därför att krypteringsnyckeln slumpas fram vid varje körning och är **78** endast just nu.

Programstrukturen

Programmet **EncryptFile** på nästa sida som genererar utskriften ovan, är i högsta grad modulariserat och består av följande filer:

EncryptFile.cpp	innehåller	**main()** som anropar alla andra funktioner
encryptText.h		funktionen **krypt()** som krypterar text
readShowFile.h		funktionen **readShowFile()** som läser en fils innehåll och visar det på skärmen
writeFile.h		funktionen **writeFile()** som skriver till en fil

I början av **main()** skapas **char**-arrayen **fileText** för att lagra filens innehåll. Vi har förberett en liten textfil och döpt den till **Okrypterad.txt** som ska krypteras och som ligger i projektmappen.

Satsen:

```
antal = readShowFile(fileText, "Okrypterad.txt");
```

anropar funktionen **readShowFile()** som inkluderas i programmet, läser filen **Okrypterad.txt** tecken för tecken och lagrar innehållet i **char**-arrayen **fileText**. Samtidigt returnerar den ett **int**-värde som när det tilldelas variabeln **antal**, återger antalet tecken i filen. Hur den gör det kommer vi att se lite senare när vi tittar på koden. Sedan låter vi funktionen **myRand()** generera ett slumptal mellan **1** och **1000** som tilldelas variabeln **key**, slumpnyckeln som används vid kryptering. Därför skickas den tillsammans med **fileText** och **antal** till **krypt()**:

```
// EncryptFile.cpp
// Läser text från en fil, krypterar den med en slumpnyckel,
// skriver krypterade texten till en annan fil och visar den
// Slumpnyckeln ger vid varje körning en annan kryptering
// Dekrypterar texten och skriver den till en tredje fil samt
// visar både den återställda filen och slumpnyckeln
#include <iostream>
#include <fstream>
using namespace std;
#include "encryptText.h"        // Innehåller        krypt()
#include "myRand.h"             //                   myRand()
#include "readShowFile.h"       //            readShowFile()
#include "writeFile.h"          //               writeFile()

int main()
{
   srand(time(0));
   char fileText[1000];         // Arrayens storlek behöver
                                // ändras vid större filer
   int antal, key = myRand(1, 1000);        // Slumpnyckeln
   cout << "Okrypterad fil:\n";

   antal = readShowFile(fileText, "Okrypterad.txt"); // Läser
   krypt(fileText, key, antal);  // Krypterar med slumpnyckel

   writeFile(fileText, "Krypterad.txt", antal);    // Skriver
   cout << "Krypterad fil:\n";

   readShowFile(fileText, "Krypterad.txt");          // Läser
   krypt(fileText, -key, antal);                // Dekrypterar

   writeFile(fileText, "Återställd.txt", antal);   // Skriver
   cout << "\n\nÅterställd fil:\n";

   readShowFile(fileText, "Återställd.txt");         // Läser
   cout << "Krypteringsnyckeln:\t" << key << "\n";
}
```

I funktionen **krypt()** vars kod inkluderas i en headerfil i början av programmet och kommer att visas senare, förskjuts varje tecken med slumpnyckeln **key**:s värde i ASCII-tabellen – inte någon avancerad krypteringsmetod – men i och med den är

slumpbaserad får man ett annat resultat vid varje körning. Krypteringsnyckeln **key** används senare för att återställa filen genom att anropa funktionen **krypt()** med **key**:s negativa värde dvs sätta tillbaka alla tecken på sina ursprungliga platser i ASCII-tabellen. Men mellan dessa två anrop av krypteringsfunktionen – en gång för kryptering, en andra gång för dekryptering (framhävda med vit bakgrund i koden) – har vi två andra anrop, först:

```
writeFile(fileText, "Krypterad.txt", antal);
```

som skriver den krypterade texten **fileText** till filen **Krypterad.txt**. Parametern **antal** skickas för att ha ett avslutningskriterium för skrivningen till filen. Sedan:

```
readShowFile(fileText, "Krypterad.txt");
```

som läser den krypterade texten från filen och visar den på skärmen, efter att den med **writeFile()** hamnat där. Till skillnad från det första anropet av funktionen **readShowFile()** (förra sida) tilldelas här returvärdet inte till någon variabel då det inte behövs. Anropets resultat kan beskådas på sid 280 och visar att filen verkligen är krypterad. Nu återstår beviset på att krypteringen gjorts på ett sätt att vi alltid har möjligheten att återställa filen och att vi verkligen får filens ursprungliga skick. Därör anropas krypteringsfunktionen andra gången:

```
krypt(fileText, -key, antal);
```

där tecknet **-** inte ska tolkas som bindestreck i texten utan som det matematiska *tecknet minus* till variabeln **key**:s talvärde då **key** är deklarerad som ett heltal av typ **int**. Vi skickar alltså **key**:s negativa värde till samma krypteringsfunktion för att sätta tillbaka alla tecken på sina ursprungliga platser i ASCII-tabellen. En blick på funktionen som är externlagrad förklarar saken:

Funktionen krypt()

```
// encryptText.h
// Tar emot en text via arrayen t och krypterar den genom
// att förskjuta alla tecken med n steg i ASCII-tabellen
// Kontrollerar textens slut med 3:e parametern antal

void krypt(char t[], int n, int antal)
{
   for (int i = 0; i < antal; i++)
       t[i] = t[i] + n;
}
```

Den aktuella parametern **key** överförs vid anrop till den formella parametern **n**. När **n** får ett positivt **key**-värde, ökas tecknens ASCII-kod med **n**. Ett negativt **key**-värde minskar ASCII-koderna med samma belopp dvs sätter tecknen tillbaka på sina ursprungliga platser. Därför kan vi använda samma funktion även för dekryptering. Filinnehållet **fileText** som skickas till **t** är en array av **char**. För att kunna av-

sluta **for**-satsen måste vi använda oss av den tredje parametern **antal**. De andra externlagrade funktioner som anropas i **EncryptFile** är följande:

Funktionen readShowFile()

```
// readShowFile.h
// Funktion som läser innehållet i filen fileName tecken för
// tecken, lagrar det i arrayen t samt visar det på skärmen
// Returnerar dessutom antal tecken som läses och visas

int readShowFile(char t[], string fileName)
{
   int i;              // Antal tecken som läses från filen
   char tecken;
   ifstream fileForRead(fileName);
   for (i=0; fileForRead.get(tecken); i++)
   {
       t[i] = tecken;
       cout << tecken;
   }
   fileForRead.close();
   return i;
}
```

Funktionen **get()** i **for**-satsens villkor läser ett tecken från filen, lagrar det i **char**-variabeln **letter**, flyttar markören till nästa tecken i filen och returnerar **true** om det finns tecken kvar och **false** om det stöter på filslutteckenet. Så läses filen, lagras i **char**-arrayen **t** samt skickas till **cout**. Antalet tecken returneras.

Funktionen writeFile()

```
// writeFile.h
// Funktion som skriver texten t bestående av antal
// tecken till filen fileName

void writeFile(char t[], string fileName, int antal)
{
   ofstream fileForWrite(fileName);
   for(int i = 0; i < antal; i++)
      fileForWrite << t[i];
   fileForWrite.close();
}
```

I **for**-satsen skriver utmatningsoperatorn **<<** arrayen **t** tecken för tecken till filen **fileName**. Parametern **antal** – antal tecken – används för att avsluta **for**-satsen.

Övningar till kapitel 10

10.1 Skriv ett program som skapar en tom fil, skriver i den texten ”Den här texten kommer från mitt första C++ filhanteringsprogram” och sedan läser från den samt skriver ut innehållet på skärmen. Som mall kan du ta programmet **`WriteReadFile`** och modifiera den (sid 271).

10.2 Modifiera programmet från övn 11.1 ovan: Istället för att hårdkoda texten i programmet, läs in den så att programmet skriver vilken inläst text som helst till filen och läser den sedan därifrån.

10.3 Varje gång man kör programmen från övn 11.1 eller 11.2 efter första gången, rensas och återställs filen och endast den senaste texten hamnar i den. Skriv ett program som gör samma sak som övn 11.2 men bibehåller filens gamla innehåll och lägger till den nyinlästa texten utan att radera gammal data. Du kan åstadkomma det genom att öppna filen i *append mode*.

10.4 Modifiera funktionen **`randPasswd()`** (sid 277) som genererar slumplösenord genom att använda en annan lösenordpolicy: 3 små bokstäver, 2 stora bokstäver (inkl. ? och @) och 2 specialtecken. Testa din funktion i programmet **`RandPasswTest`** (sid 276) som skriver ut till en fil dessa slumpvis genererade lösenord med ett antal användarnamn.

10.5 *Filkryptering (Aktivitet)*

Bilda grupper à två studerande i klassen. Valet av gruppkompis är fritt.

Skriv ett hälsningsmeddelande eller en kort text och spara den i en oformatterad textfil, typ **`*.txt`**. Kryptera filen med funktionen **`krypt()`** på sid 282. Anteckna krypteringsnyckeln vid den aktuella körningen. Skicka den krypterade filen samt krypteringsnyckeln till din gruppkompis med uppmaningen att dekryptera filen och skicka tillbaka den återställda texten till dig.

Byt ut rollerna i gruppen och upprepa experimentet.