Förord

Detta material är avsett att användas på en inledande kurs i programmering med C++. Materialet ger inte alls en fullständig täckning av språket och den som vill fördjupa sig måste skaffa en mer fullständig bok.

Vissa avsnitt har märkts med asterisk (*) vilket anger att materialet är perifert och kan överhoppas åtminstone på vid första genomläsningen.

Den första versionen av detta kompendium gavs ut 1993 och skrevs av Rickard Enander. Under tidens gång har bland andra Per Foyer, Per Wahlund och Tom Smedsaas gjort uppdateringar och många fler har bidragit med synpunkter.

Sedan den första versionen av kompendiet skrevs har språket C++ förändrats med en ny standard. Vidare har målgruppen för detta kompendium också delvis förändrats. Vi har därför ansett det nödvändigt med en mer omfattande genomarbetning. Av praktiska skäl har kompendiet för närvarande delats i två delar. Den första delen täcker grunderna ungefär motsvarande medan den anndra delen tar upp klasser, pekare, filhantering mm.

Den här föreliggande versionen skall betraktas som en ”prerelease”. Vi tar tacksamt emot felrapporter och andra synpunkter via email till tom@tdb.uu.se. Tyvärr tillåter inte vår ekonomi oss att i likhet med Donald Knuth utlova en dollar per funnet fel.

Upplaga 24 september 2002

Innehåll

Kapitel 1

Inledning

1.1 Syfte

Syftet med detta kompendium är att lära ut grunderna i programmering med programmeringsspråket C++. Det behövs inte några speciella förkunskaper. Vi förutsätter att du har tillgång till något datorsystem (Unix, PC, Macintosh...) och behärskar den grundläggande hanteringen såsom inloggning, editering och filhantering.

Programmering kan man inte lära sig enbart genom att läsa en utan man måste också öva genom att skriva och uttesta egna program.

Observera att detta material inte är att betrakta som en kursbok. En bok är betydligt mer fullständig. Materialet skall istället ses som kommenterade föreläsningsanteckningar.

1.2 Introduktion till programmering

Ett program är en följd instruktioner för hur en viss uppgift skall utföras. De instruktioner som en dator kan utföra är mycket enkla som t ex att addera två tal eller att skriva ut ett tecken på skärmen. Det som gör en dator kraftfull är att dessa enkla operationer kan göras så oerhört snabbt. Datorerna kan idag t ex utföra flera tusen miljoner operationer (t ex additioner) per sekund.

För att lösa ett problem med hjälp av en dator måste lösningen beskrivas med hjälp av sådana enkla instruktioner. Lyckligtvis behöver vi inte uttrycka oss i fullt så primitiva termer som datorn förstår eftersom man har utvecklat särskilda *programmeringsspråk* (t ex C++) som är lite närmare våra vanliga formuleringssätt. Innan ett program skrivet i ett sådant språk kan köras på en dator så måste det översättas till datorns mera primitiva *maskinspråk*. Detta görs av särskilda program kallade *kompilatorer*.

Med programmering, i vid bemärkelse, avser vi hela processen att från ett löst formulerat uppgift producera ett fungerande program som löser uppgiften. Man kan urskilja några olika steg på vägen mot målet:

Analys av problemet Från början är uppgiften sällan helt klart formulerad utan kräver en mer ingående analys exakt vad som skall göras. Idealt sett skall man aldrig påbörja resten av arbetet innan detta är utklarat eftersom man då riskerar att lösa fel problem. I praktiken är det dock vanligt att man under senare faser av utvecklingen upptäcker saker som antingen inte är specificerade eller specificerade på ett dumt eller omöjligt sätt.

Design I denna del bestäms de stora strukturerna i programmet: vad som behövs, vilka övergripande datastrukturer som skall användas mm.

Algoritmformulering Här tar man fram de egentliga metoderna – algoritmerna – för hur problemen skall lösas. Dessa formuleras i ord eller med hjälp olika typer av diagram, figurer och ”pseudospråk”.

Kodning I denna fas implementeras strukturer och algoritmer i ett programmeringsspråk. Programmeringsspråk har i regel mycket större krav på exakthet än vanligt språk vilket ofta förorsakar nybörjaren problem. När man väl lärt sig språket så är ofta denna del av arbetet rutinmässigt.

Testning och felsökning I alla ovanstående faser finns möjligheter att göra fel. Innan man kan betrakta programmet som färdigt måste man försöka övertyga sig om att det är korrekt. Det är i själva verket ofta en mycket svår uppgift (stora program är sällan eller aldrig helt korrekta). Testning är ett viktigt medel att avslöja fel. När ett felaktigt beteende hos programmet är upptäckt gäller det att finna felet. Har felet uppstått vid kodningen är det ofta lätt att rätta när det väl är lokaliserat. Om det har uppstått vid tidigare faser kan det innebära att flera av stegen ovan måste göras om.

Underhåll Program som är tänkta att fungera under en längre tid behöver underhållas. Förändringar i omgivningarna, förändrade krav, utbyte av datorsystem, passage av sekelskifte m m innebär ofta att programmet måste modifieras. På sikt är denna aktivitet den mest kostnadskrävande för många stora program.

En mycket viktig del av arbetet är *dokumentation*. Det finns många olika typer av dokumentation t ex användarhandledningar, systembeskrivningar, kodkommentarer, testresultat och underhållsprocedurer. Dokumentationen skall inte ses som en särskild aktivitet som görs när allt annat är klart utan måste pågå under hela utvecklingsprocessen.

De program som skrivs under en första programmeringskurs blir naturligtvis mycket små och intresset tenderar till att koncentreras på algoritmformulering och kodning. Det är dock viktigt att man har hela bilden klar för sig eftersom de övriga stegen ofta är de mest tids- och kostnadskrävande.

1.3 Vad är en algoritm?

Ordet algoritm kommer från Al-Kwarizmi, en framstående arabisk matematiker som levde på 900-talet e Kr. Algoritmbegreppet är grundläggande i programmeringsarbetet. En algoritm för ett problem är en metod för att lösa problemet i ett ändligt antal steg. Vi stöter på många algoritmer i vårt vardagsliv: matrecept, stickbeskrivningar, instruktioner för att programmera videon och anvisningarna för att sätta samman en bokhylla är bara några exempel. Metoderna består av ett antal grundoperationer och en anvisning om i vilken ordning de skall utföras. Utmärkande för en algoritm är att den förutsätter kännedom om dessa grundoperationer. Den skall i övrigt vara så detaljerad att alla läsare som kan grundoperationerna skall kunna följa den. Vilka grundoperationerna är beror på vilket problem algoritmen löser. I matlagning är t ex steka, koka och hacka några av grundoperationerna. Vem som helst som känner till matlagningens grunder skall kunna laga t ex kalops eller baka en sockerkaka genom att följa ett recept. Om detta inte är möjligt är receptet illa formulerat.

En algoritm utgår från vissa givna indata. Om problemet är att laga fiskpudding så kan indata vara t ex 1 paket torsk, 1 paket ärtor, 2 ägg, 2 dl mjölk, salt, peppar och ris. Resultatet efter genomgång av algoritmen (d v s fiskpuddingen) kallas utdata.

I receptet för fiskpuddingen kan man tänka sig följande formulering:

> *Om fiskblocket fruset så ... annars ...*

I en stickbeskrivning kan man tänka sig:

> *Upprepa ... tills stycket är tillräckligt långt.*

Båda dessa formuleringar ger exempel på *kontrollstrukturer*. Med kontrollstrukturer styr man arbetsgången i en algoritm. Vanliga kontrollstrukturer är *alternativ* och olika typer av *slingor*. Alternativ är ett uttryck av typen ”Om villkor så ... annars...”. En slinga (eng. loop) föreskriver att en viss operation skall upprepas tills något givet villkor är uppfyllt.

Algoritmer kan beskrivas på många olika sett: med ord, med figurer (t ex IKEAs anvisningar för hur möbler sätts samman), med *pseudospråk* eller med olika typer av diagram. Några sådana metoder beskrivs i med utgångspunkt från följande exempelproblem.

1.4 Exempelproblem

Antag att vi har fått uppgiften att skriva ett program som räknar ut rötterna till ett andragradspolynom.

1.4.1 Analys

Det första vi måste göra är att reda ut vad problemställningen innebär.

- Vad är ett andragradspolynom?

$$ax^2 + bx + c$$

- Rötter till:

$$ax^2 + bx + c = 0$$

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

- Observation: Koefficienterna a, b och c bestämmer polynomet, dvs de är *indata* till problemet.
- Observation: x_1 och x_2 är rötter, d v s *utdata* från problemet.

Symboliskt kan vi se vårt program på följande sätt:

Alla kombinationer av koefficienter går inte bra.

- a får inte vara 0, annars division med 0!
- $b^2 \geq 4ac$, ty annars ej reella rötter.

1.5 Algoritmformulering

Vi skall nu lösa problemet och det första steget är att formulera en algoritm:

Det finns olika sätt att beskriva eller representera en algoritm. Det viktiga är egentligen inte hur utan att man verkligen formulerar sig. Det är viktigt att tänka efter före.

Hur detaljerat? Varje del i algoritmen skall vara ett basproblem, d v s ett problem till vilket lösningen är känd. Detaljeringsgraden blir olika för olika personer beroende på hur skicklig eller erfaren programmeraren är.

När vi idag formulerar andragradsproblemet kanske vi måste ha 5 steg i algoritmen. Om vi skall lösa en andragradare i morgon som en del i ett annat större

problem så är andragradsekvationen i sig ett basproblem. Programmering och livet går ut på att samla basproblem på hög!

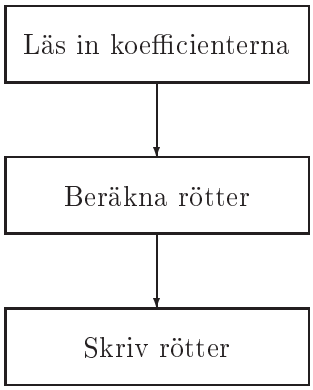
Algoritmformuleringen har två syften:

- Att dela in problemet lagom stora delproblem som var för sig är lösbara. Våra hjärnor är klarar inte att överblicka hur stora problem som helst.
- Att anpassa mina tankegångar till datorns enkla programmeringsspråk.

Låt oss nu se hur algoritmer kan se ut.

1.5.1 Blockschema

Ett blockschema är grov, schematisk beskrivning av huvudstegen i uppgiften, t ex



1.5.2 Pratalgoritm

1. Läs in koefficienterna a, b och c.
2. Kolla att $a \neq 0$. Om inte FEL, SLUTA.
3. Kolla att rötterna x_1 och x_2 är reella. Om inte FEL, SLUTA.
4. Beräkna rötterna enligt formel.
5. Presentera resultatet på skärmen.

En algoritm behöver inte vara mer komplicerad eller snyggare än så här. Varje punkt ovan skall vara ett basproblem.

1.5.3 Pseudokodning

Vi skriver ett i något som liknar ett programmeringsspråk (ofta likande språken algol och Pascal) men gör det enklare genom att utlämna detaljer:

```
program andragradare( input,output );

begin
     ReadCoefficients( a,b,c );
     if (a=0) then Error('a=0');
     if (b*b < 4*a*c ) then Error("Ej reella rötter");
     CalculateRoots( a,b,c,x1,x2 );
     PresentResults( x1,x2 );
end.
```

1.5.4 Flödesschema

Flödesscheman var en av de första metoderna att grafiskt beskriva en algoritm. Dessa har dock kommit alltmer ur bruk ur bruk bl a eftersom de saknar bra struktureringsprimitiver.

1.5.5 Struturdiagram

Det har gjorts olika försök med att ersätta flödesscheman med med diagram som är bättre passar för modernt tänkande och moderna programmeringsspråk. Någon allmänt accepterad metod har dock inte vuxit fram.

1.6 Kom ihåg

- Algoritmbegreppet, algoritmformulering, basproblem.

Kapitel 2

C++, en första lektion

2.1 Vad är ett programmeringsspråk

Programmering och programmeringsspråk är en ung vetenskap, och till skillnad mot t ex egyptologi kommer den att fortsätta att utvecklas och att vara ett synnerligen aktuellt område för forskning och utveckling. En kort historik av utvecklingen av datorer och programmeringsspråk kan se ut som följer:

De första datorerna såg dagens ljus på 1940-talet. De var till omfånget mycket imponerande, men till prestanda skrattretande ur dagens perspektiv. En ordinär tvättmaskin idag har betydligt mer datorkraft än 40-talets monster. Sverige låg ett tag i världstoppen av datorutvecklingen med BARK och BESK. BARK står för Binär Automatisk ReläKalkylator och dess efterföljare för Binär Elektronisk SekvensKalkylator. BESK blev klar 1953 och klarade av ca 3000 multiplikationer per sekund och kunde lagra 8000 tal. Den programmerades genom att rattar ställdes in för data och instruktioner innan programmet startade. Datorn programmerades under denna tid i maskinkod, d v s som en följd av ettor och nollor. Ett program kunde se ut som följer:

rad 0	0000001010 1111001010	S(10) -> Ac+
rad 1	0000001011 1111001000	S(11) -> Ah+
rad 2	0000001100 1110101000	At -> S(12)
rad 3	0000001010 1111001100	S(10) -> R
rad 4	0000001011 1111000110	S(11)*R -> A
rad 5	0000001100 1111001000	S(12) -> Ah+
rad 6	0000001100 1110101000	At -> S(12)

Här står det vänstra 10-bitars-ordet för en adress till en minnescell och det högra för en instruktion. Som synes är denna beskrivning av ett program mer eller mindre obegripligt. Låt oss se vad programmet betyder i ”klarspråk”.

rad 0	Rensa ackumulatorn och addera innehållet i minnescell 10.	S(10) -> Ac+
rad 1	Addera innehållet i minnescell 11 till innehållet i ackumulatorn.	S(11) -> Ah+
rad 2	Flytta innehållet i ackumulatorn till minnescell 12.	At -> S(12)
rad 3	Rensa innehållet i aritmetiska registret och fyll på med innehållet i minnescell 12.	S(10) -> R
rad 4	Rensa ackumulatorn och multiplicera innehållet i minnescell 11 med innehållet i aritmetiska registret. Placera resultatet i ackumulatorn.	S(11)*R -> A
rad 5	Addera innehållet i minnescell 12 till innehållet i ackumulatorn.	S(12) -> Ah+
rad 6	Flytta innehållet i ackumulatorn till minnescell 12.	At -> S(12)

Vi ser att programmet är lagrat i minnescell 0–6 och data i 10–12. Minnet används till både data och instruktioner. Dessutom finns det register för speciella operationer som t ex addition och multiplikation. När programmen blev större sågs behovet av att ha ett mer lätthanterligt sätt att instruera datorn. Den binära koden skrevs då mnemoniskt d v s med namn på instruktionerna. Dessa ”språk” kallas ofta för assemblerspråk.

rad 0	GETA 10	S(10) -> Ac+
rad 1	ADDA 11	S(11) -> Ah+
rad 2	MOVEA 12	At -> S(12)
rad 3	GETR 10	S(10) -> R
rad 4	MULTA 11	S(11)*R -> A
rad 5	ADDA 12	S(12) -> Ah+
rad 6	MOVEA 12	At -> S(12)

Observera att varje rad i assemblerprogrammet motsvarar en rad i maskinkoden. Än idag används assembler för att programmera avsnitt där det är viktigt att det går fort, ty med assembler har man fullständig kontroll över vad som datorn gör. Assemblerprogrammet översätts till maskinspråk med ett program som brukar kallas för assemblator.

Att programmera med assembler blir snabbt opraktiskt om programmen blir miljontals instruktioner långa. Vad de ovanstående programsatserna egentligen uttrycker är den matematiska formeln:

$$c = ab + a + b$$

Observera att formeln är 1 rad lång och programmet 7 rader. Moderna programmeringsspråk låter oss skriva den matematiska formeln utan att behöva bekymra oss om register hit eller dit eller om minnesceller. Variablerna a, b och c associeras automatiskt med någon minnescell. För att göra programmet mer begripligt kan vi också ha längre symboliska namn som t ex:

Salary = hours * pay_per_hour;

Nästa steg i programmeringsutvecklingen var att kunna översätta matematiska formler, som ovan, till assembler/maskinkod. Ett av de första programmeringsspråken var FORTRAN, FORmula TRANslation system, som introducerades 1954. FORTRAN är fortfarande ett mycket viktigt språk inom industrin och forskningen, främst för att så mycket pengar är investerade i program.

I vissa situationer är det viktigt att språket så långt som möjligt ser till att programmeraren inte kan göra fel. Ett sådant språk sägs vara *hårt typat*. En situation då detta är extra viktigt är då programmeraren är ovan såsom t.ex. i en introduktionskurs! Ett språk som är hårt typat och avsett speciellt för undervisning är Pascal. Pascal är också använt för administrativa program och på persondatorer (IBM-PC och Macintosh). Fördelen med Pascal är att kompilatorn är så pass sträng att få överraskningar drabbar programmeraren. Pascal introducerades i början på 70-talet och har sedan dess varit det vanligaste undervisningsspråket.

I början av 70-talet började också utvecklingen av ett nytt operativsystem, UNIX. Avsikten var att ha ett operativsystem för många olika datortyper och att program på en maskin enkelt skulle kunna flyttas över till en annan datortyp. Det fanns då behov av ett språk som var lättanvänt. Ett sådant språk brukar benämnas ett högnivåspråk. Ur detta behov utvecklades programspråket C. Föregångaren till C var naturligtvis B!

FORTRAN, Pascal och C tillhör de språk som brukar kallas procedurella språk, vilket innebär att man skiljer på algoritm och data. Ett program består då av ett antal funktioner och procedurer som utför något enligt en given algoritm. Dessa rutiner anropas med olika datamängder. Om vi har ett program som hanterar 10 olika kundregister kan det finnas en subrutin som plockar ut den rikaste kunden ur ett givet register. Nyckelordet här är data och algoritm. Procedurella språk visar sig ha begränsningar då stora program skall konstrueras.

En ny infallsvinkel på problemet ”stora program” är att se data och algoritm som en enhet, ett objekt, vilket kallas objektorienterade programmeringsspråk, förkortat OOP. I fallet kundregister ser vi ett register som ett objekt som har data, t ex namn, telefonnummer och egenskaper, metoder, som t ex att kunna ur sig självt plocka fram den rikaste kunden eller att lägga till en ny kundkontakt.

De första objektorienterade språken var Simula och Smalltalk som utvecklades under 70-talet men det var inte förrän under 90-talet som de objektorienterade språken blev ordentligt på modet och kommer att vara det dominerande sättet att programmera på 90-talet. Det finns ett flertal OOP och det vi väljer att arbeta med i detta material är C++. Andra exempel på OOP är Simula, Smalltalk, ADA95 och Object-Pascal.

Java är det stora tillskottet till programmeringsspråken under 1990-talet. Ytligt sett ser Java och C++ ganska lika ut. Behärskar man det ena språket kan man i långa stycken läsa och förstå program skrivna i det andra. Dock finns dock flera betydande skillnader.

2.2 Varför väljer vi C++

Valet av programmeringsspråk har alltid varit ett hett diskussionsämne bland programmerare och lärare. Det finns flera skäl för att inte använda C++ som ett nybörjarspråk: lätt att göra fel, delvis krånglig och kryptisk syntax och komplicerat. C++ är ett mycket rikt språk med stora uttrycksmöjligheter för användaren, men som nybörjare behövs inte alla finesser användas. Motivet till att börja med C++ är att det idag är det mest använda språket i många programmeringssituationer, t ex Microsoft Windows och Macintosh. Detta material kommer inte att lära ut hur ett fönsterprogram på Mac eller SUN programmeras, men vi kommer inte heller att stänga några dörrar till fönstren! Vi väljer C++ för att ni skall kunna växa som datoranvändare, programmerare och människor i framtiden. Dessutom introducerar C++ ett sätt att formulera problem, med hjälp av objekt, som kan vara användbart i andra situationer. Parollen kommer att vara: härska genom att söndra. Med det menas att stora problem skall slås sönder i mindre mer hanterbara delar. Dessa mindre delar skall sedan bli vänner och kunna kommunicera med varandra.

2.3 Hur skall vi lära oss C++

Vi kommer genomgående att välja det minst krångliga sättet att skriva programmen på. För att inte falla i onödiga gropar kommer jag att peka ut speciella finesser som i annat fall kan råkas på av misstag. Ni skall dock sträva efter att använda så få finesser som möjligt, om ni inte vill utmana ödet. Ett problem med nya koncept, som OOP, och s k förståsigpåare är att många modeord används utan att man begriper vad de står för. Vi skall under kursens gång försöka ge följande ord en substansiell innebörd : metod, meddelande, arv, polymorfi, virtuella metoder, datainkapsling, klass, objekt, instans.

De programexempel som finns i texten finns åtkomliga på nätet. Tag hem dessa och provkör dem samt prova att göra olika ändringar i dem!

2.4 Programexempel

2.4.1 Exempel 1

Vi kastar oss direkt över vårt första C++ program:

Program 2.1

```
//  Filnamn: .../helloWorld.cc
//
//  Ett enkelt program i C++
//  Det skriver bara ut ett meddelande
```

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

int main() {
  cout << "Hello world! ";
  cout << "Hello world!" << endl;
  return 0;
}
```

Detta program skriver bara ut ett meddelande på skärmen. Låt oss gå igenom programmet uppifrån och ner:

- Allt som står efter `//` till slutet av raden behandlas som en kommentar, d v s inte som en del av instruktionerna. De 4 första raderna är i detta fall bara information till dig som läsare av programmet.
- `#include <iostream>` Detta är ett direktiv till kompilatorn att börja läsa från filen med namnet *iostream*, som innehåller information om in- och utmatning till terminal och från tangentbord. Du behöver inte veta vad som står i filen, bara att den måste inkluderas på detta sätt i programmet.
- `using namespace std;` Denna rad anger att vi vill använda namnrymden `std` (som står för ”standard”). Liksom `include`-raden ovan så är det här en rad som vi alltid får skriva utan att tills vidare förstå varför.
- `int main()` Alla C++ program måste innehålla en huvudfunktion som heter `main`. `int` står för att main skall returnera ett heltalsvärde, mer om det senare. De tomma parenteserna står för att funktionen inte använder några argument.
- `cout << " ";` `cout` är ett objekt som sköter utmatning till skärmen. Det som står mellan " representerar en sträng och kommer bokstavligen att skrivas ut. I den andra `cout`-satsen har vi två strängar: `cout << " "<< " " ;` Flera element ”slås ihop” med `<<`. Den sista elementet `endl` ger en radframmatning på skärmen.
- `return 0;` avslutar programmet och ger `main` funktionsvärdet 0. Värdet används inte i programmet, men kan avläsas i operativsystemet.

Vidare noterar vi följande i programmet.

- Det som står mellan { } hör till funktionen main och kallas funktionskroppen.
- Alla satser avslutas med ett `;` (semikolon).

Tills vidare kommer alla våra program ha följande struktur:

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

```
int main() {
    övriga satser
    return 0;
}
```

2.4.2 Exempel 2

Låt oss titta på ett program som räknar ut något!

Program 2.2

```
//  Filnamn: .../someComp.cc
//
//  Ett andra exempel på ett program. Det läser in ett tal,
//  beräknar sinus och cosinus för talet samt
//  skriver ut resultaten.

#include <iostream>
#include <cmath>

using namespace std;

int main() {
  double  x, sinval, cosval;

  cout << "Hello again, please give me a number > ";
  cin  >> x;
  cout << "Thanks, I will make some calculations \n";

  sinval = sin(x);
  cosval = cos(x);
  cout << "The number you gave me was " << x << "\n";
  cout << "The cosine of that number is  " << cosval << "\n";
  cout << "The sine   of that number is  " << sinval << "\n";
  cout << "Bye for this time!\n";

  return 0;
}
```

Detta program läser in ett tal, `x`, och beräknar sinus- och cosinus-värdena. Vi noterar följande för programmet:

- Vi har inkluderat ännu en fil, `cmath`. Den innehåller matematiska definitioner som t ex sin och cos.
- Numeriska variabler kan definieras (deklareras). I vårt fall har vi deklarerat tre variabler av typen `double`, `x`, `sinval` och `cosval`. `double` är en typ av reella tal. Mer om det senare.

- Programmet kan läsa in ”saker”, i detta fall ett tal från användaren. Objektet `cin` används för inläsningar.
- `cout` kan inte bara skriva ut text utan också siffervärden.

När programmet körs kan det se ut som följer:

```
Hello again, please give me a number > 0.25 <Return>
Thanks, I will make some calculations
The number you gave me was 0.25
The cosine of that number is 0.968912
The sine of that number is 0.247404
Bye for this time!
```

Den text som är understruken har användaren av programmet. <Return> innebär inte att du skall skriva detta bokstavligt utan att du skall trycka på <Return>-tangenten. Observera att programmet självt, dvs operatorn `<<` i `cout`, bestämmer hur många decimaler som skrivs ut för respektive tal. Vidare räknar C++ i radianer.

2.5 Kom ihåg

- Historik är allmänbildande.
- Hur ett litet C++ program ser ut.
- `#include` satsen.
- `using namespace std;`
- `main`-funktionen.
- `cout` och `cin`.

Kapitel 3

Datatyper, variabler och aritmetik

3.1 Datatyper

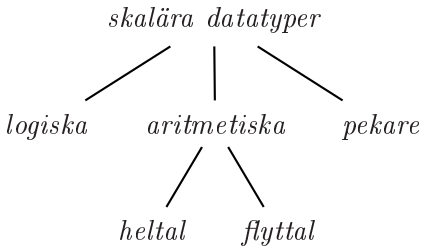
Information kan vara av olika typer. Exempel på olika typer (numeriska) *tal* och *text*. I programmeringsspråken brukar man precisera typerna närmare, t ex *heltal*, *decimaltal* och *tecken* (dvs bokstäver, siffror, skiljetecken ...). De olika typerna skiljer sig åt dels hur de *representeras* (lagras) i datorn och dels vilka operationer man kan utföra på dem ("vad man kan göra med dem"). Det är också vanligt att man sätter samman flera värden, eventuellt av olika typer, till nya typer. Typen *text* kan ju t ex sammansättas av flera värden av typen *tecken*.

I ett bra programmeringsspråk bör man kunna definiera *egna typer* för att beskriva och hantera de begrepp som man vill behandla. Om man skall skriva ett program som utför en trafiksimulering så vill man kanske kunna skapa typer som *bil*, *väg*, *vägkorsning*, *kö* medan ett banksystem kan behöva typer som *konto* och *kund*. I C++ använder man huvudsakligen *klassbegreppet* för detta.

Det finns ett antal fördefinierade klasser som vi kommer använda oss av i detta kompendium (t ex är `cout` och `cin` i exemplen objekt ur en fördefinierad klass). Hur man gör egna klasser diskuteras i del II.

De datatyper som vi först skall studera är de inbyggda så kallade *skalära* typerna. Man skulle också kunna kalla gruppen för de *fundamentala* typerna eftersom de är grundstenar för att bygga nya datatyper eller *atomära* för att de, precis som atomer, är odelbara (i själva verket kan de, precis som atomer, med visst våld delas upp i mindre beståndsdelar)

Bland de skalära datatyperna kan man urskilja olika grupper: aritmetiska, logiska och pekare. Den aritmetisk gruppen kan i sin tur uppdelas i *heltal* och *flyttal*. Detta leder till följande bild av de fördefinierade skalära datatyperna i C++:



Man kan förundras över att det till verkar saknas någon datatyp för att hantera tecken och text. Så är dock inte fallet. Enskilda tecken hanteras som heltalstyp kallad `char` medan text hanteras av en fördefinierad sammansatt typ kallad `string`. Mer om detta senare.

Vi skall börja med att titta på de aritmetiska typerna. Den logiska typen använder man huvudsakligen i tester och villkor. Pekartypen behandlas i del II av kompendiet.

3.2 Aritmetiska typer

Aritmetiska typer är sådana man kan räkna med. Enligt beskrivningen ovan skiljer man mellan heltalstyper och flyttalstyper. Heltalstyperna kan bara användas för att representera hela tal (t ex 491 och -17) medan flyttalstyperna även kan representera decimaltal (t ex 4.56 och -0.04).

Varför behöver man skilja mellan heltal och flyttal? Skulle man inte kunna använda bara flyttal eftersom 1 och 1.0 är samma sak? Svaret är att datorerna har olika sätt att representera dessa typer. Med heltalen kan man använda mindre utrymme, räkna exakt samt, ofta, räkna snabbare än med flyttalen. Flyttalen kan uttrycka decimaldelar och representera mycket större tal än heltalstyperna till priset av mer utrymme, avrundningsfel och långsammare räkningar.

Det finns flera varianter av såväl heltals- som flyttalstyperna men tills vidare kommer vi endast använda typerna `int` och `double`. Varianter på dessa (t ex `short`, `long` och `float`) presenteras senare. Typerna `int` och `double` är, i någon mening, *standardtyper* dvs systemet förutsätter att heltal är av typen `int` och flyttal av typen `double` om man inte anger något annat. Skriver man talet `-26` så blir det en konstant av typen `int` medan talet `47.8` blir en konstant av typen `double`.

3.3 Variabler, tilldelningar och enkla uttryck

Variabler är ett viktigt begrepp i alla programmeringsspråk. En variabel är ett utrymme i datorns minne som har försetts med ett *namn* (eller, som man också säger, en *identifierare*). I detta utrymme kan man lagra ett *värde* av någon viss *typ*. I C++ (och andra så kallade *starkt typade* programmeringsspråk) så måste

man ange vilken typ av värden en variabel skall kunna lagra. Detta gör man i en så kallad *deklaration*.

Exempel:

```
int antal;
double summa, medel;
```

Dessa satser beskriver tre variabler — en (`antal`) av heltalstyp och två (`summa` och `medel`) av flyttalstyp. Variablerna är nu *deklarerade* dvs kända till namn och typ. Det gör att systemet kan reservera utrymme i minnet för dem. Däremot har vi inte talat om vilka värden som de skall innehålla. Man brukar säga att variablerna är *odefinierade* (även om detta ord också kan ha en annan betydelse).

Till skillnad från föregångaren C så är man i C++ ganska fri att placera variabeldeklarationerna var man vill. En variabel måste dock alltid deklareras innan den används.

För att ge en variabel ett värde används en operation som kallas *tilldelning* och skrivs med likhetstecken (`=`). Likhetstecknet kallas vanligen för *tilldelningsoperatorn*. Exempel:

```
antal = 5;
```

När detta utförs lagras 5 som värde i det utrymme som är reserverat för variabeln.

Till vänster om tilldelningsoperatorn skall det finnas en variabel (eller något annat som går att tilldela värde). Till höger kan ett helt uttryck stå. Exempel:

```
summa = 1.2 + 2.1 + 3.3;
medel = summa/3.0;
cout << "Medelvärdet är: " << medel << endl;
```

Här ser vi exempel på hur man bygger upp uttryck av *konstanter*, *variabler* och *operatorer*. När satserna utförts kommer variabeln `summa` innehålla 6.6 och `medel` 2.2. Det senare värdet kommer också ha skrivits ut.

En variabel bibehåller sitt värde tills det ändras. Satsen

```
antal = antal + 1;
```

kommer först att beräkna värdet på uttrycket till höger om tilldelningsoperatorn varvid det gamla värdet på `antal` (5) används och därefter tilldela variabeln till vänster värdet av uttrycket. Efter denna sats kommer således `antal` ha värdet 6.

Man kan redan i deklarationen ge en variabel värde (ett *initialvärde*). Exempel:

```
int    n     = 10;
double left  =  0.0,
       right =  1.0,
       mid   = (right-left)/2.0;
```

För att man skall få initiera en variabel med ett uttryck som på sista raden i exemplet så måste de variabler som ingår i uttrycket vara både deklarerade och

definierade. Variabeln `mid` kommer här att få värdet 0.5.

De aritmetiska uttrycken byggs upp av vanliga artimetiska operatorer: `+` för addition, `-` för subtraktion, `*` för multiplikation och `/` för division. Det gäller samma prioritetsregler som i matematik ("gånger före plus", ...) och, precis som i matematiken, använder man parenteser för att ändra beräkningsordning. Vid lika prioritet utförs operationerna i ordning från vänster till höger. Följande tabell innehåller exempel på uttryck:

Uttryck	Värde	Kommentar
`5 - 4 + 3 - 2 + 1`	3	Från vänster till höger ty samma prioritet
`5 - (4 + 3)`	-2	Parenteserna först
`8 + 2*4`	16	`*` före `+`
`8/2*4`	16	Först `/` sedan `*` eftersom samma prioritet
`8/(2*4)`	1	Parentesen först
`7.5/2.5`	3.0	Flyttalsoperationer med flyttalsresultat

Det är viktigt att veta att när man utför en aritmetisk operation så blir resultatet av heltalstyp om operanderna är av heltalstyp och av flyttalstyp om operanderna är av flyttalstyp (även om, som i sista exemplet i tabellen visar, resultatet är ett heltal). Vad som händer om man har olika datatyper återkommer vi till.

När divisionsoperatorn (`/`) används på heltalstyper är det inte säkert att divisionen "går jämnt ut". Eftersom resultatet skall vara av heltalstyp så måste man på något sätt göra sig av med decimalerna. Detta görs genom *trunkering* dvs decimalerna stryks helt enkelt. Värdet av uttrycket `1/2 + 3/4` är således 0 eftersom båda divisionerna görs först (högst prioritet) och resulterar i 0. Detta sätt att dividera på brukar benämnas *heltals*-division för att skilja den från "vanlig" division. En god regel är att alltid sätta ut decimalpunkt om man avser att ange ett flyttal!

Förutom de redan nämnda operationerna för addition, subtraktion, multiplikation och division finns en *rest*-operator (`%`). Den kan bara användas med heltalsoperander och ger resten vid heltalsdivision. Exempel: Satsen

```
cout << 17/5 << "  " << 17%5 << "  " << 5%17 << endl;
```

kommer att skriva ut raden

```
3  2  5
```

Ett uttryck kan också innehålla anrop till *funktioner*. I C++ finns ett antal funktioner fördefinierade. Till dessa hör bl a

sin(x), cos(x), tan(x)	trigonometriska funktioner, x i radianer
exp(x)	exponentialfunktionen e^x
sqrt(x)	kvadratroten $\sqrt{x}$
log(x)	naturliga logaritmen $\ln x$
log10(x)	10-logaritmen $\log_{10} x$
fabs(x)	absolutbeloppet $\lvert x \rvert$
pow(x,y)	"x upphöjt till y" x^y

Dessa funktioner förväntar sig flyttalsargument och ger flytalsresultat.

Exempel på hur man skriver uttryck:

$\frac{1}{x+y}$	`1/(x+y)`
$\sin x + \cos y$	`sin(x) + sin(y)`
$x^2 + y^{i-1}$	`x*x + pow(y, i-1)`
$\frac{x}{yz}$	`x/y/z` eller `x/(y*z)`
$\frac{e^{\lvert x+y\rvert}}{2}$	`0.5*exp(fabs(x+y))`

De matematiska funktionerna är inte inbyggda i själva språket utan finns i ett standardbibliotek. För att kunna använda dem måste man inkludera en deklarationsfil som heter `cmath`. Detta görs med raden

```
#include <cmath>
```

som placeras i början av filen. (I gamla versioner av C++ och i språket C heter filen `math.h`)

Exempel:

Program 3.1

```
// Filnamn: .../pythagoras.cc

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

int main() {
  double a = 1.0, b =2.0, c;
  c = sqrt( a*a + b*b );
  cout << "Hypotenusan är " << c << endl;
  return 0;
}
```

3.4 Namngivning av variabler

Variabelnamn får inte se ut hur som helst utan måste följa vissa regler:

1. De måste börja med en bokstav.

2. Därefter ett godtyckligt antal alfanumeriska tecken (dvs bokstäver eller siffror) inklusive understrykningstecknet (_).

3. Reserverade ord får inte användas som namn. Exempel på sådana är: `double`, `int`, `for`, `while`. Förutom de reserverade orden finns ett antal fördefinierade begrepp (t ex `cout`). Det är inte förbjudet att använda dessa men oftast mycket olämpligt.

Dessa regler gäller inte bara för variabelnamn utan för alla typer av namn (så kallade *identifierare*) på begrepp som senare kommer att diskuteras (funktioner, klasser, typer ...).

Exempel på godkända variabelnamn:

```
int    lineNumber, linesPerPage;
double x, x_1, x_2;
double Delta, delta;
```

Observera det är skillnad på *gemena* och *VERSALER* (dvs "små" och "stora" bokstäver) — tecknen 'a' och 'A' betraktas som olika tecken. Således är `Delta` och `delta` olika variabelnamn.

Exempel på icke godkända eller olämpliga variabelnamn:

```
int     3i;             // Första ej bokstav
double  arne-kalle;     // Minustecken ej tillåtet
double  lines per page; // Blanktecken ej tillåtet
double  motstånd;       // å är ingen bokstav...
double  if;             // Reserverat ord
int     If;             // Korrekt men olämpligt
double  cout;           // Korrekt men olämpligt
int     If;             // Fel - redan använt
```

Variabelnamnen skall väljas så att de beskriver vad variabeln används till. Om en variabel skall beteckna x-koordinaten hos något objekt så kalla den t ex för `x_coord` och inte för `fasterAnki`. Man kan använda versaler eller understrykningstecken (_) för att öka läsbarheten hos långa namn (t ex `linesPerPage` och `x_coord`).

3.5 In- och utmatning av grundläggande datatyper

För att ett program skall ha något större intresse måste det kunna visa användaren någon form av resultat. Detta görs oftast genom utskrifter t ex på en datorskärm.

Vi har i de inledande exemplen sett hur man använder en speciell variabel `cout` och en speciell operator `<<` för att åstadkomma sådan utskrifter. Satsen

`cout <<` *uttryck*;

kommer att beräkna värdet av uttrycket och sedan skriva ut det på "standard output" som oftast är terminalens skärm. Man kan också skriva ut flera uttryck

i samma sats genom att använda operatorn flera gånger. Exempel:

```
cout << x << y << z;
```

Detta blir dock inte så bra eftersom värdena kommer skrivas ihop. För att undvika detta kan man t ex skriva ut blanktecken eller komma mellan vilket kan göras mha *textsträngar* som omges av citationstecken ". Exempel:

```
cout << x << ", " << y << ", " << z;
```

Ofta vill man ha någon mer förklarande text och se till att man får ny rad efter utskriften. Exempel:

```
cout << "Koordinater:" << x << ", " << y << ", " << z << endl;
```

Identifieraren `endl` anger att raden skall avslutas här så att nästa utskrift kommer först på ny rad.

Utmatningsoperatorn `<<` anpassar sig till den datatyp som skall skrivas ut — är det en text så skrivs tecknen i texten ut, är det ett heltal så skrivs dess siffror ut och är det ett flyttal så skrivs siffror med decimalpunkt. Det finns många möjligheter att styra antal siffror, antal decimaler mm i utskrifterna men vi tar inte upp detta nu.

Hittills har vi använt tilldelningsoperatorn `=` för att ge variabler värden. Detta måste skrivas in i programtexten så om man vill använda ett annat värde på en variabel så måste man ändra i själva programmet. Ett mer flexibelt sätt är att låta den som kör programmet ge värden som *indata* (eng *input*). Man säger att man låter programmet *läsa in* värden från t ex tangentbordet. Detta görs med hjälp av inmatningsoperatorn `>>` och den variabel som står för "standard input" och som heter `cin`. Satsen

`cin >>` *variabel*;

läser in tecken från tangentbordet och lagrar motsvarande värde i den angivna variabeln. Även här kan man upprepa operatorn `>>` för att läsa in flera variabler.

Program 3.2

```
// Filnamn: .../io.cc
#include<iostream>
using namespace std;

int main() {
  double radie, yta, omkrets;
  double pi = 3.14159;

  cout << "Ange en cirkelradie: ";
  cin >> radie;
  yta     = pi*radie*radie;
  omkrets = 2.*pi*radie;
  cout << "Cirkelns yta är " << yta
       << " och dess omkrets är " << omkrets << endl;
```

```
  int m, n;
  cout << "Gå två heltal: ";
  cin >> m >> n;
  cout << m << "*" << n << " = " << m*n << endl
       << m << "/" << n << " = " << m/n << endl;

}
```

När detta program körs blir resultatet:

```
Ange en cirkelradie: 9
Cirkelns yta är 254.469 och dess omkrets är 56.5486
Gå två heltal: 7 4
7*4 = 28
7/4 = 1
```

(Understruken text anger inmatning dvs sådant som användaren har skrivit på tangentbordet.)

Exakt vilka tecken som läses och hur dessa tolkas beror på typen av den variabel som man läser till. I de flesta fall så ignoreras inledande blank-tecken (inklusive radbyte och tabulatortecken dvs så kallade *vita tecken*). Därefter läser systemet så många tecken som kan tolkas som hörande till ett värde i aktuell datatyp. Det innebär t ex att om man skriver `123.456` som indata till en heltalsvariabel så läses bara `123` medan alla sju tecknen läses om inmatningen görs till en flyttalsvariabel.

3.6 Symboliska konstanter

Ibland vill man kunna ge speciella konstanter namn (t ex π) som man vill använda i stället för att skriva 3.141593 överallt. Man kan naturligtvis deklarera en variabel `pi` och initiera denna till rätt värde:

```
double pi = 3.141593;
```

Förfaringssättet är dock lite osäkert eftersom det skulle vara möjligt att av misstag ändra värdet på `pi`.

```
pi = -13.7;
```

Detta skulle naturligtvis ge mycket märkliga resultat om man t ex använder `pi` för att beräkna omkretsen och areor av cirklar.

För att öka säkerheten finns det kan man sätta inte `const` först i deklarationenen:

```
const double pi = 3.141593;
```

Kompilatorn kommer nu att ge felmeddelande om vi försöker ändra på `pi`.

3.7 Typblandning och typkonvertering

Det är i viss utsträckning tillåtet att ha olika typer i ett uttryck. Man kan antingen låta C++ gör typomvandlingarna *automatiskt* eller själv *explicit* ange vilka omvandlingar som skall göras.

När C++ skall utföra en aritmetisk operation (t ex en addition) så tittar systemet på operandernas typer. Om inte båda är av samma typ (t ex om den ena är `int` och den andra `double`) så måste värdet av den ena operanden konverteras så att de har samma typ som den andra. Vid blandning av heltal och flyttal så är det alltid heltalsvärdet som konverteras till flyttal (13 blir 13.0 t ex). Det är viktigt att komma ihåg att man tittar på varje operation för sig och inte konverterar allt till flyttal bara för att det finns ett flyttal med i ett uttryck. Således blir uttrycket

`5/3 + 3.0/4` = `1.75`

Beräkningarna sker i följande ordning:

1. `5/3` räknas ut med heltalsdivision och får värdet 1
2. `4` räknas om till 4.0
3. Flyttalsdivisionen `3.0/4.0` utförs och får värdet 0.75
4. Resultatet av den första divisionen 1 konverteras till 1.0
5. En flyttalsaddition av 1.0 och 0.75 resulterar värdet 1.75

(I själva verket vet man inte vilken av de två divisionerna som utförs först. I detta fall spelar det ingen roll för resultatet men det finns situationer när man måste se upp med det.)

Automatisk konvertering från flyttal till heltal uppstår t ex om man tilldelar ett flyttalsvärde till en heltalsvariabel. Konverteringen sker genom så kallad *trunkering* dvs decimalerna strykes helt enkelt.

Exempel:

```
int    n;
double x, y;
n = 5/3 + 3.0/4;  // n får värdet 1
x = n/2 + 4;      // x får värdet 4.0
x = n/2 + 4.0;    // x får värdet 4.0 (igen);
x = n/2.0 + 4;    // x får värdet 4.5
n = n/2.0 + 4;    // n får värdet 4
```

Det är också viktigt att veta att konverteringar av variabelvärden inte påverkar själva variabeln. I näst sista raden i exemplet ovan kommer visserligen värdet av variabeln `n` konverteras till flyttal när uttrycket skall beräknas men `n` kommer fortfarande ha heltalsvärdet 1.

Om ett uttryck inom parentes föregås av ett typnamn (t ex `int`) så konverteras värdet till den angivna typen. Exempel:

```
int    m = 3, n = 4;
double g, h;
g = m/n;                  // g får värdet 0
h = double(m)/double(n); // h får värdet 0.75
h = double(m)/n;          // h får värdet 0.75 (igen)
h = int(h + 0.5)          // h får värdet 1.0
```

Ibland kan man se en konstruktion där typnamnet satts inom parentes i stället för uttrycket efter typnamnet (t ex `(double) m`). Denna form har ärvts från språket C men kan behöva användas i vissa situationer i C++.

Vissa kompilatorer ger varningsutskrifter om man tilldelar ett flyttalsvärde till en variabel av heltalstyp. För att slippa varningarna använder man explicit typkonvertering. Exempel:

```
int n = int( ln(123.47) );
```

3.8 Mer om tilldelningar

Som vi redan sett har tilldelningen den generella formen

variabel = *uttryck*

Detta är i själva verket också ett uttryck som har ett värde och värdet är det som variabeln får. Det betyder att man kan upprepa användningen av tilldelningsoperatorn. Exempel: Satsen

```
x = y = z = 1.0;
```

kommer att ge alla variablerna värdet 1.0

Till skillnad från andra operatorer vi sett så utförs tilldelningsoperatorerna *från höger till vänster* dvs som om vi satt ut parenteserna

```
x = (y = (x = 1.0));
```

Observera att det som står till vänster om en tilldelningsoperator måste vara något som går att tilldela ett värde — ett så kallat *lvalue*. De enda exemplen på sådana som vi hittills stött på är variabler. Följande sats är således felaktig:

```
y = x + 1 = 4;
```

Vissa tilldelningar är så vanliga att man har definierat egna operatorer för dem. Exempel:

Exempel	Alternativt skrivsätt
`x += 7`	`x = x + 7`
`y -= 2`	`y = y - 2`
`z *= 0.5`	`z = z*0.5`
`n /= m + 2`	`n = n/(m + 2)`
`++i`	`i = i + 1`
`--j`	`j = j - 1`

Observera att tvåteckenskombinationerna `+=` etc är egna operatorer och inte två operatorer som råkar stå intill varandra. Det går t ex inte att ha ett blanktecken mellan de två tecknen.

De sista två operatorerna (`++` och `--`) finns också i en *postfix* form (ovanstående form kallas *prefix*) dvs de kan skrivas efter variabeln i stället för före. Dessa har samma effekt som prefixvarianterna men värdet av uttrycket är det gamla värdet. Exempel: Satserna

```
i = j = 1;
m = ++i;
n = i++;
```

kommer att sätta variablerna `i`, `j` och `m` till 2 medan `n` får värdet 1.

Att använda dessa operatorer inuti uttryck är tillåtet men bör absolut undvikas eftersom det skapar förvirring och emellanåt också odefinierade resultat. (Vilket värde ger satsen

```
i -= i += -i+++i--;
```

till variabeln `i`?)

3.9 Varianter på heltalstyper typer*

Vi har hittills sett heltalstyperna `int` och `char`. Det finns ytterligare två typer: `short int` och `long int` (kan förkortas till `short` respektive `long`). Det som skiljer typerna åt är hur mycket plats de tar och därmed hur stora tal de kan lagra. Utrymmet mäts vanligen i *byte* där en byte lagrar 8 binära siffror (bitar). En byte kan därför lagra $2^8 = 256$ olika värden. Vidare kan alla heltalstyper förses med attributet `unsigned` för att ange att värdet skall betraktas som icke-negativt. (Man kan också ange `signed` men, förutom för typen `char`, så är det alltid standard.) Exakt vilka storlekar som används för respektive typ beror på implementation men följande tabell anger vanliga värden:

`char`	1 byte	-128 – 127 eller 0 – 255
`unsigned char`	1 byte	0 – 255
`signed char`	1 byte	-128 – 127
`short`	2 byte	-32768 – 32767
`unsigned short`	2 byte	0 – 65536
`long`	4 byte	-2147483648 – 2147483647
`unsigned long`	4 byte	0 – 4294967295

Typen `int` är oftast antingen 2 (vanligt i Windows) eller 4 (vanligt i Unix) byte. Att detta är implementationsberoende kan ställa till problem när man flyttar program mellan olika system.

När man skriver konstanter så utgår systemet från att de är i decimal form. Man kan dock ange konstanter i både *oktal* och *hexadecimal* form. För att ange att en

konstant skall tolkas oktalt eller hexadecimalt inleds den med `0` respektive `0x`. För att skriva tal hexadecimalt används bokstäverna A-F för siffrorna 10–15.

Exempel:

```
unsigned int   ui     = 17;
unsigned short us     = -1;         // Förbjudet
int            octal1 = 011;        // 9
int            octal2 = 099;        // Förbjudet
int            hex1   = 0x11;       // 17
int            hex2   = 0xff;       // 255
unsigned long  ul     = 0xffffffff; // 4294967295
long           sl     = 0xffffffff; // -1
```

Observera att värdena representeras binärt i datorn oavsett hur vilken bas vi använt när vi skrivit konstanterna. Vi väljer oktal eller hexadecimal sätt bara när vi själva tycker att det underlättar för oss.

3.10 Mer om flyttal och flyttalsaritmetik

Flyttalen skall försöka efterlikna matematikens reella tal men har begränsningar i både noggrannhet och talområde. Det innebär alltså att tal (ofta) inte kan lagras exakt samt att man inte kan ha hur stora tal som helst. Precis vilka tal som inte kan representeras exakt beror på detaljer i implementationen som vi inte skall gå in på här. Det är dock bra att veta att hela tal kan representeras exakt som flyttal om de inte är alltför stora medan även enkla tal med decimaldelar (t ex 0.1) ofta får avrundningsfel.

Det finns två varianter av flyttal i C++, typen `double` som vi redan sett och typen `float`. (Det finns faktiskt också en tredje typ `long double`) Ofta har värden av typen `float` ungefär 7 decimala siffrors noggrannhet medan värden av typen `double` har ungefär 15 decimala siffrors noggrannhet. Vid omfattande beräkningar med flytal är det viktigt att ha kontroll på hur dessa begränsningar i noggrannheter påverkar resultaten men detta ligger utanför ramen för detta kompendium. Använd typen `double` om du inte har speciella skäl för något annat.

För att en konstant skall vara av flyttalstyp så måste den skrivas med decimalpunkt och/eller med *exponentdel* som är ett e eller E följt av en tal som anger tiopotens. Om lägger till ett f eller F sist så blir konstanten av typen `float` annars blir den av typen `double`. Exempel

Exempel:

Konstant	Värde	Typ
`1.5`	1.5	`double`
`.23`	0.23	`double`
`-10.`	-10.0	`double`
`0.5e5`	$0.5 \cdot 10^5$	`double`
`1e-10`	10^{-10}	`double`
`-.45f`	-0.45	`float`

Om båda operanderna vid en operation (t ex en addition) är av typen `float` så blir resultatet av typen `float`. Om den ena operanden är av typen `double` så konverteras den andra till samma typ och resultatet blir av typen `double`. Som tidigare påpekats så returnerar de matematiska funktionerna (`sin`, `exp`, ...) resultat av typen `double`.

3.11 Kom ihåg

- Variablers grundläggande egenskaper, namn, typ och värde.
- Typerna `int` och `double`.
- Skillnaden mellan heltalsaritmetik och flyttalsaritmetik
- Hur variabler får namnges.
- Begränsat antal värdesiffror och största värde.
- `cout` och `cin` kan användas för tal.
- De fyra vanliga räknesätten.
- Heltalsdivision
- Explicit och implicit typkonvertering.
- Tilldelningsoperatorerna
- `const`-kvalificeraren.
- Matematikfunktioner i `cmath`.

Kapitel 4

Villkors- och iterationssatser

4.1 Programflöde och kontrollstrukturer

Hittills har vi gått igenom tilldelningar samt satser för in- och utmatning. Dessa satser har utförts i *sekvens* dvs i tur och ordning som de står i programtexten. Vi vill ofta kunna bryta detta strikt sekvensiella utförande t ex genom att bara göra en sats under vissa villkor ("Om det regnar så stannar vi hemma annars går vi en promenad") Ett annat vanligt önskemål är att kunna upprepa en sats tills något speciellt händer ("så länge spiken inte är helt inslagen så slå på den") eller ett visst antal gånger ("läs in och summera 100 heltal").

Satser som bryter den löpande ordningsföljden brukar med ett gemensamt ord kallas *kontrollstrukturer*. Det finns två huvudtyper: *villkorssatser* som väljer väg och *iterations-* eller *repetitionssatser* som upprepar en angiven sats flera gånger. I C++ (liksom i det flesta programmeringsspråk) kallas den väsentliga villkorssatsen för *if-sats*. Egentligen skulle det räcka med en iterationssats också men av bekvämlighetsskäl finns det tre som skiljer sig i när och hur man avgör om iterationen skall fortsätta eller ej. Dessa satser kallas *while-*, *for-* och *do-while-*sats.

4.2 Villkorssatsen

Villkorssatsen används således för att välja vilken eller vilka satser som skall utföras beroende på någon omständighet. Vi tittar direkt på ett exempel:

```
double x;
cout << "Ange ett tal: ";
cin >> x;
if ( x<0 )
   cout << "Talet var negativt!" << endl;
else
   cout << "Talet var positivt" << endl;
```

Programmet läser således in ett tal. Beroende på det villkor som står efter `if` väljes en av grenarna ("then"-grenen eller "else"-grenen). (Observera att man *inte*, som i vissa andra programmeringsspråk, skall skriva `then`)

Ibland behöver man ingen `else`-gren. Exempel: Koden

```
cout << "Talet är";
if ( x>=0 )               // Större än eller lika med
   cout << " icke";
cout << " negativt" << endl
```

skriver ut texten "`Talet är negativt`" respektive "`Talet är icke negativt`" beroende på värdet av variabeln `x`.

De allmänna formerna av satserna ser ut som

```
if ( villkor )
     sats
else
     sats
```

respektive

```
if ( villkor )
     sats
```

Om man vill ha flera satser i någon av grenarna så måste man omge dessa med så kallade *satsparenteser* dvs `{` och `}`. På detta sätt skapas en *sammansatt sats* som utåt sett är en enda sats. Exempel:

```
if ( alder < 18 ) {
   cout << "Du är för ung!" << endl;
   return 0;                          // Avbryt
}
// Fortsätt
. . .
```

Satserna i grenarna kan vara vilka satser som helst — t ex nya villkorssatser. Exempel:

```
if ( x < 10 )
   cout << "För lite!" << endl;
else
  if ( x < 20 )
     cout << "Lagom!" << endl;
  else
     cout << "För mycket! << endl;
```

Om man har flera olika villkor att testa av brukar man strukturera koden enligt exemplet:

```
if ( x == 1 )                    // lika med 1?
   cout << "Grönt" << endl;
else if ( x == 2 )               // lika med 2?
   cout << "Gult" << endl;
else if ( x==3 )                 // lika med 3?
```

```
    cout << "Rött" << endl;
else                                // annars
    cout << "Fel värde! << endl;
```

4.3 Villkor och typen `bool`

I exemplen har vi sett hur villkoren i `if`-satsen kan anges som *relationsuttryck* dvs som en storleksjämförelse mellan två uttryck. Följande jämförelseoperatorer (relationsoperatorer) finns:

C++	Matematisk symbol	Betydelse
`>`	$>$	större än
`>=`	$\geq$	större än eller lika med
`<`	$<$	mindre än
`>=`	$\leq$	mindre än eller lika med
`==`	$=$	lika med
`!=`	$\neq$	ej lika med

Observera särskilt att test på likhet görs med operatorn `==` och inte med `=` som ju står för tilldelning. Att av misstag använda tilldelning i stället för jämförelse kan vara ett mycket svårfunnet fel. Exempel: Koden

```
int x = 4;
if ( x=5 )
    cout << "x är 5" << endl;
else
    cout << "x är inte 5" << endl;
```

kommer att skriva ut `x är fem` vilket också `x` kommer att vara. När villkoret i `if`-satsen beräknas kommer nämligen `x` tilldelas värdet 5 och värdet av tilldelningsuttrycket blir då 5. Som värde på ett villkor betraktas allt som inte är 0 som sant varför then-grenen utföres.

I äldre versioner av C++ och i C så används just heltalstypen på detta sätt i tester och jämförelser. Nyare versioner av C++ har tillfogat en särskild *logisk* datatyp med namnet `bool` med de två möjliga värdena *sant* och *falskt* som skrivs `true` respektive `false` . En jämförelse resulterar således numera i ett värde av typen `bool`. För att gamla program (och gamla programmerare...) fortfarande skall fungera finns det en automatisk typkonvertering från de aritmetiska typerna till typen `bool` sådan att 0 eller 0.0 blir `false` och allt annat `true`.

Man kan naturligtvis deklarera variabler av typen `bool`. Exempel:

```
bool negative;
int  n;
cin >> n;
negative = n < 0;    // negative blir true om n < 0 annars false
...
if ( negative ) {    // om negative är true
```

```
    ...
}
```

Logiska uttryck kan kombineras med *logiska operatorer*:

C++	Alternativ form	Betydelse
`&&`	`and`	logiskt *och*
`||`	`or`	logiskt *eller*
`!`	`not`	logiskt *icke*

Alla kompilatorer godkänner inte de alternativa formerna (`and`, `or` och `not`) och då dessa inte heller finns i C eller Java så avstår vi från att använda dem.

Exempel:

```
int number;

cout << "Ge mig ett tal mellan 1 och 6: ";
cin >> number;

if ( number>=1 && number<=6 )
   cout << number << " du sa, ganska bra!" << endl;
else
   cout << number << " du sa, inget att ha!" << endl;
```

Observera att `&` och `|` är helt andra operatorer — man måste dubblera tecknen för att få de logiska operatorerna.

Om man vill kombinera flera av de logiska operatorerna bör man veta att `!` har högst prioritet följt av `&&` och `||` som således har lägst prioritet. Uttrycket

```
x>0 || y>0 && z>0
```

tolkas således som

```
x>0 || (y>0 && z>0
```

dvs uttrycket är sant antingen om `x` är positiv eller om *både* `y` och `z` är positiva (eller om alla tre är positiva — operatorn `||` svarar mot *och/eller* i dagligt tal).

4.4 Repetitionssatser

Repetitionssatser används för att upprepa en viss sekvens flera gånger. En sådan programkonstruktion brukar kallas en *iteration*, *loop* eller *slinga*.

4.4.1 `while`-satsen

Vi tittar direkt på ett exempel:

Program 4.1

```
// Filnamn: .../squares.cc
// Beräkna kvadrater på ett antal inlästa tal

#include <iostream>
using namespace std;

int main() {
  double tal;
  cout << "Ge tal att kvadrera. Avbryt med 0.";
  cin >> tal;                        // Läs det första talet
  while ( tal!=0 ) {                 // Så länge inte 0 lästs
    cout << "Kvadraten är " << tal*tal << endl;
    cin >> tal;                      // Läs nytt tal
  }
  cout << "Tack för mig!" << endl;
  return 0;
}
```

Programmet börjar med att skriva ut en instruktion och läser sedan in det första talet. I `while`-satsen testar man först huruvida talet är noll eller ej. Om det inte är noll så görs en utskrift och ett nytt tal läses varefter programmet fortsätter med att göra en ny test. När talet noll läses så avbryts iterationen och programmet fortsätter med den avslutande utskriften.

Satsen har den generella formen

`while (` *logiskt uttryck* `)`
  sats

där man som vanligt får använda satsparenteserna `{` och `}` om det är flera satser som skall upprepas.

Det logiska uttrycket beräknas varje gång och så länge det är sant så utförs satsen. Det är naturligtvis väsentligt att uttrycket någon gång blir falskt — i annat fall stannar aldrig programmet och man har skapat en *oändlig* loop. (Det finns i och för sig möjlighet att avbryta iterationen på andra sätt men normalt bör den avbrytas genom att uttrycket blir falskt.)

Ytterligare ett exempel:

```
double x;
cout << "Ge ett positivt tal: ";
cin  >> x;
while ( x<0 ) {
  cout << "Talet var inte positivt! Försök igen: ";
  cin  >> x;
}
...
```

I detta fall vägrar programmet att gå vidare innan användaren matat in ett positivt tal.

Ett tredje exempel: Ett program som beräknar summan $1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{10}$

Program 4.2

```
// Filnamn: .../harmonicWhile.cc
// Beräkna den harmoniska summan

#include <iostream>
using namespace std;

int main() {

  int    n;
  cout << "Antal termer: ";
  cin >> n;

  double h = 0.0;
  while ( n>0 ) {
    h = h + 1./double(n);
    n--;
  }
  cout << "Summan är: " << h << endl;
  return 0;
}
```

En iteration kan i sig innehålla en iteration. Antag t ex att vi vill kunna köra ovanstående program med olika antal. Då kan vi ha en yttre iteration som läser in antalet termer och utför beräkningen med just detta antal.

Program 4.3

```
// Filnamn: .../harmonicWhile2.cc

#include <iostream>
using namespace std;

int main() {
  int    n;

  cout << "Antal termer: ";
  cin >> n;
  while ( n>0 ) {
    double h = 0.0;
    while ( n>0 ) {
      h = h + 1./double(n);
      n--;
    }
    cout << "Summan är: " << h << endl;
    cout << "Nytt antal (0 om du vill sluta): ";
    cin  >> n;
  }
```

```
  return 0;
}
```

4.4.2 `for`-satsen

I de många programmeringsspråk är *for*-satsen inriktad mot att repetera en viss sats ett bestämt antal gånger och så använder vi den i de första exemplen. Satsen är dock mer generell.

Exempel: Program som skriver en tabell med kvadrater och kuber på de hela talen från 1 till 10.

```
int i;
for ( i = 1; i <=10; i++ )
    cout << i << "   " << i*i << i*i*i << endl;
```

Variabeln `i` sättes först till 1. Varje varv i loopen inleds med att testa om `i` är mindre än eller lika med 10. Om detta är fallet skrivs en rad varefter `i` ökas med 1 och en ny test görs etc.

`for`-satsen har som synes en mer komplicerad syntax:

`for (` *initiering*`;` *repetitionsvillkor*`;` *repetitionsuttryck* `)`
sats

Funktionen går att beskriva i punktform:

1. Utför *initieringen*
2. Beräkna *repetitionsvillkor*
3. Om värdet blev falskt så fortsätt vid punkt 7
4. Utför *satsen*
5. Utför *repetitionsuttrycket*
6. Fortsätt vid punkt 2
7. . . . (första sats efter `for`-satsen)

Ett exempel till: Följande program beräknar hur ett kapital växer med en viss ränta under ett visst antal år.

Program 4.4

```
// Filnamn:.../raentaFor.cc
// Hur stort blir kapitalet efter ett visst antal år

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

```
int main() {
  double kapital = 1.;                // Startkapital
  double raenta;                      // Räntesats i procent
  int    aar;                         // Antal gångna år

  cout << "Räntesats i procent: ";
  cin >> raenta;
  cout << "Antal år           : ";
  cin >> aar;

  for ( int i = 1; i<=aar; i++ )
    kapital = kapital + kapital*raenta/100.;

  cout << "Kapitalet har blivit " << kapital << endl;

  return 0;
}
```

Hittills har vi i `for`-satserna använt någon *räknare* som hållit reda på hur många satser som skall utföras. Vi kan dock använda vilket logiskt uttryck som helst som repetitionsvillkor. Antag t ex att vi vill veta hur många år det tar att fördubbla ett kapitalet med en viss ränta. Detta program skulle kunna skrivas:

```
for ( ; ; kapital < 2. )

dumt exempel...
```

Funktionen av `for`-satsen kan beskrivas med hjälp av `while`-satsen:

initiering;
`while (` *repetitionsvillkor* `) {`
    sats
    repetitionsuttryck;
`}`

Således hade `for`-satsen i ränteberäkningen i programmet ovan lika gärna kunnat skrivas som

```
int i = 1;
while ( i <=aar ) {
  kapital = kapital + kapital*raenta/100.;
  i++;
}
```

Omvänt kan alltid en `while`-sats skrivas som en `for`-sats. Summaberäkningen i exempel ? hade kunnat skrivas

```
double h;
for ( h = 0.0; n>0; n-- )
    h = h + 1./double(n);
```

Observera att vi i detta exempel *inte* kan deklarera variabeln `h` inuti `for`satsen! Detta beror på att vi vill använda variabeln i utskriften *utanför* iterationen.

Däremot hade vi kunnat göra initieringen i deklarationen och lämnat tomt på initieringsuttryckets plats:

```
double h = 0.0;
for ( ; n>0; n-- )
    h = h + 1./double(n);
```

Huruvida man väljer en `for`- eller en `while`-sats är mest en fråga om personlig smak.

4.4.3 `do`-satsen

Funktionen för `do`-satsen är som för `while`-satsen förutom att testen på fortsättning görs sist i ställe för först. Detta innebär att satserna i loopen alltid utförs minst en gång. Exempel

```
int i
do {
  cout << "Svara 0 eller 1: ";
  cin >> i;
} while ( i!=0 && i!=1 );
```

Här kommer programmet begära att användaren matar in 0 eller 1 tills han gör det.

4.5 `switch`-satsen*

Det finns ytterligare en villkorssats som kan vara användbar om man har många olika likhetsfall att testa. Exempel:

```
switch ( x ) {
 case 1:                    // lika med 1?
   cout << "Grönt" << endl;
   break;
 case 2:                    // lika med 2?
   cout << "Gult" << endl;
   break;
 case 3:                    // lika med 3?
   cout << "Rött" << endl;
   break;
 default:                   // annars
   cout << "Fel värde! << endl;
}
```

Denna programkod kommer att fungera exakt lika som den på sid 30 och den blir faktiskt heller inte kortare eller enklare.

Man kan flera värden i `case`-uttrycken. Följande kod visar hur man kan beräkna antalet dagar i en månad:

```
case (month) {
 11, 4, 6, 9 : days = 30;  // Trettio dagar har november,
               break;      // april, juni och september,
           2 : days = 28;  // februari tjugoåtta allen
               break;
 default     : days = 31;  // alla de övriga trettioen
}
```

Syntaxen för switch-satsen har följande form:

```
    switch (uttryck av heltalstyp) {
        case konstantlista :
            satser;
        case konstanterlista :
            satser ;
        case konstantlista :
            satser ;
            ...
        default :
            satser ;
}
```

Vad som händer i switch-satsen är följande:

- Uttrycket beräknas.
- Programmet hoppar till en matchande `case`-rad.
- Om ingen matchande rad påträffas hoppar programmet till `default`-raden, om en sådan finns.
- Från det ställe som man hoppat till (`case` eller `default`) utförs alla satser tills `switch`-satsen i sin helhet tar slut eller tills en `break`-sats påträffas.

Satsen `break` betyder att något bryts, vilket i C++ mening innebär att programmet hoppar ut ur pågående block av satser.

Observera att uttrycket i `switch`-satsen måste vara av heltalstyp. Om vi skulle vilja göra samma sak för ett uttryck av `double`-typ får vi måste vi använda nästlade `if`-satser.

4.6 Kom ihåg

- `if`-satsen.
- Nästlade `if`-satser.
- Satsparenteser `{ }` och sammansatt sats.
- `while`-satsen.

- `for`-satsen.
- `do`-satsen.
- `switch`-satsen

Kapitel 5

Tecken och strängar

5.1 Tecken och typen `char`

För att representera tecken (enskilda bokstäver, siffror, punkt, komma, ...) ges varje tecken en heltalskod. Det vanligaste systemet är den s k ASCII-koden. Den ursprungliga versionen av ASCII använder talen 0 till 127. I denna kod saknas bland annat de svenska bokstäverna å, ä och ö. För att få med dessa och andra tecken har koden utökats så att man använder värden 128 till 255. Det finns en internationell standard med beteckningen ISO 8859 som kallas LATIN_1 för denna kod.

Lyckligtvis behöver man inte komma ihåg koderna utan man använder sig av *teckenkonstanter*. Sådan skrivs omgivna av apostrofer. Exempel:

```
int m, n;
m = 'a';              // 49 (ascii-kod för a)
n = 'c';              // 51 (ascii-kod för c)
cout << m   << endl;  // Skriver ut 49
cout << n   << endl;  // Skriver ut 51
cout << n-m << endl;  // Skriver 2 (skillnaden mellan 51 och 49)
```

Förmodligen vill vi dock ha bokstäverna själva utskrivna och inte deras ascii-koder. Detta kan vi åstadkomma genom att använda typen `char` i stället `int`. Det är fortfarande en heltalstyp men utskriftsoperatorn `<<` hanterar den annorlunda — den kommer att skriva ut tecknet som har det lagrade värdet som ascii-kod. Samma exempel som ovan men med `char` i stället för `int`:

```
char m, n;
m = 'a';
n = 'c';
cout << m   << endl;  // Skriver tecknet a
cout << n   << endl;  // Skriver tecknet c
cout << n-m << endl;  // Skriver 2 som förut
```

Att utskriften av `n-m` fortfarande blir 2 beror på att resultatet av en aritmetisk operation mellan heltal (inklusive `char`) är av typen `int`.

Motsvarande gäller vid inläsning — om vi läser till en variabel av typen `char` så kommer *ett* tecken att läsas och dess ascii-kod lagras i variabeln. Exempel:

Program 5.1

```
// Filnamn: .../char.cc

#include <iostream>
using namespace std;

int main() {
   char c;
   cout << "Ge ett tecken: ";
   cin  >> c;
   cout << "Kod för tecknet " << c
        << " är " << int(c) << endl;
   cout << "Nästa tecken är " << char(c+1)
        << " med kod " << c+1 << endl;
   return 0;
}
```

Körexempel:

```
Ge ett tecken: +
Kod för tecknet + är 43
Nästa tecken är , med kod 44
```

Observera hur vi har använt `char` och `int` för konvertering mellan tecken- och heltalstyp som beskrevs i avsnitt 3.7.

Ovanstående program fungerar inte för riktigt alla inmatade tecken. Undantagen är de så kallade *vita* tecknen dvs blank, tab och radslut. (Prova!) Detta beror på att `>>`-operatorn alltid börjar med att läsa förbi alla vita tecken. Vanligen vill vi att det skall fungera så när vi läser tal som indata men det är inte säkert att vi vill ha det så för tecken. Vi kan i så fall använda en annan funktionen `get` i `cin`. Exempel:

Program 5.2

```
// Filnamn: .../get.cc

#include <iostream>
using namespace std;

int main() {
   char c;
   cout << "Ge ett tecken: ";
   cin.get(c);          // Läser ett tecken och lagrar det i c
   cout << "Kod för tecknet " << c
```

```
        << " är " << int(c) << endl;
   return 0;
}
```

Prova att mata in blank, tab och retur till detta program. (Observera att du i de två första fallen också måste trycka på RETURN för att något skall hända.)

Det finns ett antal otryckbara tecken som man vill kunna ange. Detta kan göras med så kallade *escape*-sekvenser som alltid har en ”backslash” (\) som första tecken. Några vanliga sådana:

Kod	Engelskt namn	Betydelse
\n	new line	radbyte
\t	tab	nästa tabulatorstopp
\a	alert	pip
\b	backspace	markören åt vänster
\f	formfeed	sidframmatning
\\	backslash	\
\’	single quote	apostrof (’)
\"	double quote	citationstecken (”)

Exempel: Raden

```
cout << "Se" << '\n' << '\t' << "upp!" << '\a' << '\n';
```

ger utskriften

```
Se
        upp!
```

ackompanjerat av ett pip.

Man kan även använda dessa escape-sekvenser i textsträngar varför ovanstående effekt betydligt enklare erhålles av satsen

```
cout << "Se\n\tupp\a\n";
```

Vi kan naturligtvis jämföra tecken med de vanliga relationsoperatorerna. Det är värt att hålla i huvudet att tecken är (små) heltal! Exempel:

Program 5.3

```
// Filnamn: .../countDigits.cc
// Räknar antalet siffror på en rad
#include <iostream>
#include <cctype>

using namespace std;

int main() {
  int ndigits = 0;
```

```
  char c;
  cin.get(c);               // Läs första tecknet
  while ( c!='\n' ) {       // Så länge ej radslut
    if ( c>='0' && c<='9' ) // Om siffra så
      ndigits++;            // räkna
    cin.get(c);             // Läs nästa tecken
  }
  cout << "Antal siffror: " << ndigits << endl;

  return 0;
}
```

I programmet läser vi successivt in tecken. Så länge vi inte hittar ett radslutstecken så ser vi om det lästa tecknet ligger mellan heltalen '0' (dvs 48 i ascii) och '9' (dvs 57 i ascii). I så fall är det en siffra (detta är sant inte bara i ascii utan i alla vanligt förekommande teckensystem).

Observera att hade kunnat skriva testen

```
if ( c >= 48 && c <= 57 )
   ndigits++;
```

men dels är detta mer kryptiskt (det kräver ju att läsaren kan teckenkoderna) och dels fungerar det inte för andra teckensystem (t ex EBCDIC).

5.1.1 Funktioner för tecken

Biblioteket `cctype` (`ctype.h` i C och gamla versioner av C++) innehåller ett antal funktioner för att hantera tecken:

`int isalpha(c)`	*true* om c bokstav
`int isdigit(c)`	*true* om c siffra
`int islower(c)`	*true* om c gemen
`int isupper(c)`	*true* om c versal
`int isspace(c)`	*true* om c vitt tecken
`int tolower(c)`	motsvarande gemen
`int toupper(c)`	motsvarande versal

Exempel:

Program 5.4

```
// Filnamn:  .../countChars.cc
// Läser en rad och räknar antalet bokstäver och siffror
#include <iostream>
#include <cctype>

using namespace std;
```

```
int main() {
  int nletters = 0, ndigits = 0, ntotal = 0;
  char c;
  cin.get(c);
  while ( c!='\n' ) {
    ntotal++;
    if ( isalpha(c) )
      nletters++;
    else if ( isdigit(c) )
      ndigits++;
    cin.get(c);
  }
  cout << "Totalt antal lästa tecken: " << ntotal << endl;
  cout << "Antal bokstäver          : " << nletters << endl;
  cout << "Antal siffror            : " << ndigits << endl;

  return 0;
}
```

Observera att det är både enklare, effektivare och säkrare att använda funktioner som `isdigit` och `isalpha` i stället för tester av den typ vi gjorde i programmet (`countDigit`) ovan.

5.2 Typen string

Ett värde av typen `char` innehåller exakt *ett* tecken (eller, snarare, ascii-koden för ett tecken). Från första början i kompendiet har vi sett exempel på hur man vid utskrifter kunnat ge hela *textsträngar* dvs ett antal tecken omgivna av citationstecken (`"`). Man kan (numera) hantera sådana textsträngar med en särskild typ kallad `string`. Exempel:

Program 5.5

```
// Filnamn: .../string1.cc
// Läser ett namn, skriver en hälsning
#include <iostream>
#include <string>
using namespace std;

int main() {
  string s = "Hej på dig";
  string namn;

  cout << "Vad heter du? ";
  cin >> namn;

  cout << s << ", " << namn << '!' << endl;
```

```
  return 0;
}
```

Programmet deklarerar två variabler av `string`-typ, ger den ena ett värde i deklarationen, frågar om och läser in ett namn samt skriver ut en hälsning.

Observera hur vi använt två `string`-variabler, en `string`-konstant och en `char`-konstant i utmatningssatsen. Vi hade lika gärna kunnat skriva `"!"` i stället för `'!'` men *inte* `', '` i stället för `", "`. En sträng kan bestå av noll, ett eller flera tecken men teckenkonstanter består alltid av exakt *ett* tecken.

Som vi ser i programmet kan vi hantera strängar på samma sätt som vi i tidigare kapitel har hanterat tal: deklaration, tilldelning, inläsning och utskrift. Den enda nyheten är att vi måste inkludera filen `string` när vi vill använda denna typ.

Ett exempel till:

Program 5.6

```
// Filnamn: .../wordlength.cc
// Räknar ordlängder
#include <iostream>
#include <string>
using namespace std;

int main() {

   string w;
   cout << "Ge ett ord: ";
   cin  >> w;
   while ( w != "STOPP" ) {
     cout << "Ordet " << w << " innehåller "
          << w.length() << " tecken" << endl;
     // Läs ett nytt ord
     cout << "Ge ett ord: ";
     cin  >> w;
   }
   return 0;
}
```

Vi ser exempel på dels hur strängar kan jämföras med relationsoperatorer (i detta fall `!=`) och dels hur man kan ta reda på strängens längd genom att använda `length`-funktionen.

När man använder relationsoperatorerna `<`, `<=` ... så måste man tänka på att dessa jämförelser baseras på de ingående tecknens ascii-koder vilket bl a innebär att ordningen mellan de svenska bokstäverna å, ä och ö blir felaktig. Vidare

så bör man veta att stora bokstäver (versaler) är ”mindre än” små bokstäver (gemena).

Observera hur `length`-funktionen anropas. När vi tidigare har anropat funktioner så har vi givit argumentet inom parentes efter funktion (t ex `sin(x)`). Nu har vi skrivit `w.length()` i stället för `length(w)`. Detta beror på att typen `string` är en *klass*. När man definierar en klass kan man förse den med ett antal funktioner. Dessa funktioner hör då till klassen och anropas vanligen för ett visst värde (oftast kallat *objekt*). Man kan uppfatta punkten ungefär som en genitivkonstruktion ”*w's length*”. Funktioner som på placeras i klasser kallas ofta för klassens *metoder*.

Det finns en stor mängd andra operationer som vill kunna göra på strängar (sätta ihop strängar, leta efter delsträngar, ta reda på längden, byta ut enskilda tecken etc). Många av dessa är skrivna som metoder i klassen `string` medan en del finns som operatorer (t ex tilldelning) och några som funktioner (t ex inläsning av en rad till till en sträng). Följande tabell innehåller några operationer. Det finns betydligt flera!

Några operationer för klassen `string`	
`string s`	Deklaration av s som får längden noll initialt (tom sträng).
`string s = t`	Deklaration av `s` som får en kopia av strängen `t` som värde.
`s.at(i)`	Returnerar tecken i position `i`. Observera att positionerna räknas från 0 och uppåt!
`s.length()`	Returnerar antalet tecken i s.
`s.append(t)`	Lägger till strängen `t` sist i `s`.
`s.insert(i,t)`	Skjuter in strängen `t` i position `i` i `s`.
`s.substr(i,n)`	Returnerar den delsträng av `s` som börjar i position `i` och består av `n` tecken (dock högst så många som ”finns”)
`s.find(t)`	Letar efter strängen `t` i `s`. Om strängen hittas returneras dess position, annars värdet `string::npos` (som normalt är -1)
`s.find(t,i)`	Som `find` ovan men börjar söka i position `i`.
`cout << s`	Skriver ut `s`.
`cin >> s`	Läser in ett ”ord” till `s`. Ord avgränsas med ”vita” tecken dvs blank, tabulator och nyrad.
`getline(cin,s)`	Läser in en hel rad till `s`

Exempel:

Program 5.7

```
// Filnamn: .../stringOperationer.cc
// Exempel på operationer i typen string
#include <iostream>
#include <string>
using namespace std;
```

```
int main() {
  string s = "mississippi";
  string t;

  s.append("missourie");
  cout << "s: " << s << endl;

  int pos1, pos2, pos3;
  pos1 = s.find("miss");
  pos2 = s.find("miss", pos1+1 );
  pos3 = s.find("miss", pos2+1 );
  cout << "\"miss\" finns i positionerna: " << pos1
       << ", " << pos2 << " och " << pos3 << endl;

  s.replace(0, 1, "M");
  s.replace(pos2, 1, "M");
  s.insert(pos2,"-");   // Skjut in ett bindestreck

  cout << s << endl;

  int len = s.length();
  int pos = s.find("-");
  cout << "pos, len: " << pos << ", " << len << endl;
  t = s.substr(pos+1, len-pos-1);
  cout << t << endl;
  return 0;
}
```

Programmet ger följande utskrifter:

```
s: mississippimissourie
”miss” finns i positionerna: 0, 11 och -1
Mississippi-Missourie
pos, len: 11, 21
Missourie
```

5.3 Mer om input

Hittills har vi vid all inmatning antingen vetat hur många saker vi skulle läsa in eller så har vi använt något speciellt slutvärde (t ex talet noll, strängen `"STOP"` eller tecknet `'\n'`). En vanlig situation är dock att vi inte vet hur mycket som skall läsas eller vad som ligger sist. Vi skulle vilja skriva något i stil med

```
while ( cin.moreData() ) {
  ...
}
```

Någon sådan metod finns inte men däremot är flera av inläsningsmetoderna gjorda så att de returnerar ett värde som tolkas som *true* om läsningen gick bra (dvs om det fanns något att läsa) annars *false*. Detta är ofta bra att utnyttja. Exempel:

Program 5.8

```
//  Filnamn: .../countChars1.cc
// Läser och räknar antalet bokstäver och siffror.
// Avbryter när inget mer finns att läsa.

#include <iostream>
#include <cctype>

using namespace std;

int main() {
  int nletters = 0, ndigits = 0, ntotal = 0;
  char c;
  while ( cin.get(c) ) {
    ntotal++;
    if ( isalpha(c) )
      nletters++;
    else if ( isdigit(c) )
      ndigits++;
  }
  cout << "Totalt antal lästa tecken: " << ntotal << endl;
  cout << "Antal bokstäver          : " << nletters << endl;
  cout << "Antal siffror            : " << ndigits << endl;

  return 0;
}
```

Observera här att vi nu bara har anrop till `cin.get` på *ett* ställe mot två i den första versionen.

När vi skall köra detta program så har vi ett problem: Den första versionen av läste fram till och med radslutstecken och stannade där men denna version läser allt som vi skriver. Om vi läser indata från en fil (vilket vi inte gått igenom än) så har ju filen ett slut och när man kommer dit (kallas ”end of file” eller EOF) returnerar `cin.get` *false* och iterationen avbryts. När vi läser från tangentbordet så läser och räknar programmet alla tecken vi skriver. Hur man markerar att det inte finns mer data från tangenbordet har inget med C++ att göra och varierar mellan olika system. Vanligt är CTRL-D (på Unix och Linux) och CTRL-Z på Windows. Detta är alltså inget tecken som kan läsas in till programmet utan en signal om att det inte finns mer data. (Livet är i själva verket mer komplicerat än så — de som använder Emacs vet att CTRL-D i alla fall kan läsas av Emacs...)

Även inläsningsoperatorn `>>` returnerar ett värde som kan användas i tester. Det

skulle gå bra att använda denna teckenräkningsprogrammet men då kommer inte de vita tecknen att räknas. Vi exemplifierar i stället med ett ordräkningsprogram:

Program 5.9

```
// Filnamn: .../wordCount.cc
// Räknar "ord". Med "ord" avses en följd av tecken som
// inte innehåller något vitt tecken
// Avbryter när inget mer finns att läsa.

#include <iostream>
#include <string>

using namespace std;

int main() {
  int nwords = 0;
  string w;;
  while ( cin >> w ) {     // Läs ett ord
    nwords++;
  }
  cout << "Antal lästa ord: " << nwords << endl;
  return 0;
}
```

5.4 Kom ihåg

- Typen `char` - en heltalstyp.
- Teckenkonstanter omges med apostrofer.
- Vita tecken.
- Specialtecken med *escape*-sekvenser.
- Typen `string` som är en *klass*.
- Operationer som medlemsfunktioner i klassen `string`
- Att `>>` hoppar över vita tecken.
- Att `>>` och `cin.get` returnerar ett värde som kan användas i tester.

Kapitel 6

Funktioner

En *funktion* är en namngiven programenhet som utför en viss specificerad uppgift. Uppgiften kan gälla att göra en viss beräkning, skriva ut en tabell, sköta om viss inläsning etc.

Vi har redan sett exempel på hur man kan använda de matematiska funktionerna *sin*, *cos*, ... Fördelen med sådana funktioner är uppenbar: någon har tänkt efter hur dessa skall fungera och skrivit den erforderliga programkoden. Vi kan sedan använda funktioner utan att behöva veta hur beräkningarna tillgår — funktionerna blir som svarta lådor som utför sina uppgifter. Det vi behöver veta för att kunna använda en funktion är

- vad funktionen heter,
- vad funktionen behöver för indata och
- vad funktionen lämnar för utdata.

De flesta matematiska funktionerna (*sin*, *cos* ...) skall ha ett värde av typen `double` som indata och lämnar ett värde av samma typ som utdata. Som vi redan sett så använder man funktionerna genom att skriva ett *anrop* bestående av funktionens namn följt av de indata man vill ge till funktionen inom parentes. Exempel:

```
y = exp(x) + exp(-x);
```

En funktion kan emellertid behöva flera värden som indata som t ex `pow`. Indata till en funktion ges som *parametrar*

Funktioner är ett sätt att göra programmeringsarbetet mer modulärt och på en högre abstraktionsnivå. Med det menas att kunna gömma detaljer angående t ex en matematisk uträkning i en funktion varefter vi kan använda denna funktion som ett språkelement och utan att tänka på detaljerna. Vi har redan stött på funktionen *cos* i ett tidigare exempel. Hur funktionen beräknar cosinus för en given vinkel behöver vi som användare inte bry oss om. Den enda som måste känna till detaljerna är den som skrev *cos*-funktionen från början. Det vi behöver veta är vad den heter, vad den skall ha för parameter och vad den returnerar.

Svaret ligger i hur cos är deklarerad:

```
double cos( double );
```

Detta anger att

- `cos` returnerar ett värde av typen `double` (det första `double` ovan).
- `cos` behöver *ett* argument av typen `double` (det mellan parenteserna).

I C++ finns det en rad standardfunktioner av denna typ. I detta kapitel ska vi lära oss hur egna funktioner definieras och används. Processen kan ses som att vi utvecklar språket efter våra specifika behov.

6.1 Funktionens syntax

En funktion *deklareras* på följande sätt:

returtyp funktionsnamn `(` *parameterlista* `);`

Regler:

- Den angivna *returtypen* kan vara vilken datatyp som helst, t ex `double` eller `int`. Returtypen kan också anges som `void` vilket innebär att funktionen inte returnerar något värde.
- *Funktionsnamnet* skall vara unikt och följer samma regler som namnet på variabler, d v s börja med en bokstav följt av ett godtyckligt antal alfanumeriska tecken.
- *Parameterlistan* beskriver antalet och typen på det indata som behövs. Parameterlistan kan vara tom.
- Deklarationen avslutas med semikolon. Observera att deklarationen inte innehåller några utförande programsatser.

En funktion *definieras* på följande sätt:

returtyp funktionsnamn `(` *parameterlista* `) {`
    satser;
    `return` *värde*`;`
`}`

- Om inte returtypen är `void` måste funktionen returnera ett värde med `return`-satsen.
- En funktion måste alltid definieras, men definitionen kan också fungerara som en deklaration.

En definition och en deklaration hör ihop vilket innebär att sekvensen

returtyp funktionsnamn `(` *parameterlista* `)`

måste vara lika. Låt oss ta ett par exempel:

Program 6.1

```
// Filnamn: .../maxex.cc

#include <iostream>
using namespace std;

int max ( int a, int b ) {
  if ( a > b )
     return a;
  else
     return b;
}

int main() {
   int tal1, tal2;
   cout << "Give two numbers : ";
   cin >> tal1 >> tal2;
   cout << "The largest of " << tal1 << " and " << tal2 ;
   cout << " is " << max(tal1,tal2) << endl;
   return 0;
}
```

I exemplet har vi ingen deklaration av funktionen utan definierar `max` överst i programmet. Funktionen måste definieras eller deklareras före `main` om `main` skall kunna anropa `max`. Anledningen är att kompilatorn måste veta exakt hur max ser ut när den anropas för att kunna kontrollera att du inte har gjort något fel. Kompilatorn läser uppifrån och ner, precis som vi själva gör! Om du i main hade skrivit

```
max( tal1,tal2,tal1 )
```

hade kompilatorn protesterat, ty antalet parametrar stämmer inte med definitionen.

Vi sade tidigare att det kan finnas fler än ett return i en funktion. I ovanstående fall har vi return i båda grenarna av if-satsen. Betrakta följande variant av max:

```
int max ( int a, int b ) {
   if ( a > b )
      return a;
   return b;
}
```

Kommer denna variant att fungera som det är tänkt? Svaret är JA, ty return returnerar ett värde och avslutar funktionen på en gång, d v s inga satser efter return kommer att exekveras. Om $a > b$ i exemplet returneras a och funktionen avbryts. Satsen `return b` kommer aldrig att utföras. Om inte $a > b$ utförs inte

`return a;` i `if`-satsen. Funktionen går då vidare till `return b;` och allt fungerar som det är tänkt!

Låt oss titta på samma exempel då vi använder oss av en funktionsdeklaration:

Program 6.2

```
// Filnamn: .../maxdekl.cc

#include <iostream>
using namespace std;

int max ( int a, int b ); // deklaration

int main() {
   int tal1, tal2;
   cout << "Give two numbers : ";
   cin >> tal1 >> tal2;
   cout << "The largest of " << tal1 << " and " << tal2 ;
   cout << " is " << max(tal1,tal2) << endl;
   return 0;
}

int max ( int a, int b )  { // definition
  if ( a > b )
     return a;
  else
    return b;
}
```

Funktionen `max` är deklarerad före `main` som därmed vet hur anropet skall se ut. Däremot vet inte `main` vad och hur `max` gör, men det spelar ingen roll! Efter `main` definieras `max`, C++ måste ju veta hur vi har tänkt oss funktionens utförande någon gång. Normalt används deklaration och definition på detta sätt, speciellt då programmen blir så pass stora att vi har mer än en programfil. Vidare är det bra att ha väsentligheterna först i filen och gömma detaljerna långt ner.

I exemplet behöver vi inte i deklarationen namnge parametrarna utan hade kunnat skriva

```
int max ( int , int );       // deklaration
```

Vad parametrarna heter har ingen betydelse för main, det enda viktiga är antalet och typerna. Om namnet skall skrivas ut är en smaksak. Det kan vara förklarande om man väljer parameternamn som är beskrivande. Följande funktionsdeklaration är mindre beskrivande

```
double avbetalning( double, double, double );
```

än denna variant

```
double avbetalning(double raenta, double tid_i_aar, double total_skuld);
```

I definitionen måste parametrarna naturligtvis namnges eftersom de skall användas explicit i funktionen.

Den sista varianten är att föredraga eftersom namngivningen fungerar ju som dokumentation. Om betydelsen av parametrarna är helt uppenbar som t ex i max-exemplet kan man dock använda den första varianten.

6.2 Tilldelning av returvärdet

Som tidigare sagts returnerar funktionen ett värde med `return`-satsen. Denna sats avbryter dessutom exekveringen av funktionen och återvänder till anropar. Du kan ha hur många `return`-satser som helst och var som helst, men med många `return` blir programkoden svårare att läsa och därmed svårare att rätta om något blir fel. En bra ledstjärna är att försöka se till att funktionen har högst tre returnsatser. Om den har fler än tre bör nog funktionen skrivas om.

Ofta benämns en funktion som returnerar en `double` ”en funktion av typen `double`” och motsvarande för andra typer. Anledningen är att en funktion som returnerar en `double` kan användas i uttryck precis som en variabel av typen `double`. Om en funktion är av typen `void` behöver den inte ha någon `return`-sats. Om du skulle försöka returnera något kommer C++-kompilatorn att säga ifrån. En sådan funktion benämns i vissa programspråk *procedur* eller *subrutin*.

6.3 Funktionens parametrar

Först lite terminologi. Parametrarna i deklarationen och definitionen av en funktion kallas för *formella parametrar* (eng. *formal parameters*). De parametrar som används i själva anropet kallas för *aktuella parametrar* (eng. *actual parameters*). De formella parametrarna i en definition finns inte ”fysiskt” utan har betydelsen: om jag hade en `int`-parameter a så skulle jag göra detta. Det är först vid ett anrop som det blir möjligt/verkligt/aktuellt att göra något.

Program 6.3

```
// Filnamn: .../ex03.cc

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

double tangens ( double );  // deklaration

int main( void ) {
   cout << "table of sin, cos and tangens: " << endl;
```

```
   for ( double x=0.0; x<=1.2; x +=0.1 ) {
         cout << x << " " << sin(x) << " " << cos(x);
         cout << " " << tangens(x) << endl ;
   }
   return 0;
}

double tangens ( double v ) // definition
{
  if (fabs(cos(v))<1e-6)
     return 1e6;
  else
     return sin(v)/cos(v);
}
```

I exemplet är `double v` formell parameter. Den finns inte förrän funktionen anropas från main med ett verkligt värde representerat av den aktuella parametern `x`. I anropet övertar `v` `x`:s värde.

Det finns två sätt på vilket parametrar kan förmedlas/överföras till en funktion, *anrop med värde* eller *anrop med referens*. Termerna på engelska är *call by value* och *call by reference*. Skillnaden är att:

- Vid *värdeanrop* överförs bara själva värdet av den aktuella parametern till den formella. Värdet kopieras vilket innebär att aktuell och formell parameter lagras i två olika minnespositioner.
- Vid *referensanrop* överförs istället adressen från den aktuella parametern till den formella. Aktuell och formell parameter kommer att vara lagrade i samma minnesposition.

6.4 Värdeanrop

C++ överför parametrar by value om inget annat sägs. Det finns en del undantag till denna regeln som vi kommer till senare. Låt oss titta på ett exempel, (från Kernighan, Ritchie: The C Progamming Language):

Program 6.4

```
// Filnamn: .../power.cc

double power( double x, int n ) {
  double  p;

  for ( p = 1.0; n > 0; n-- )
    p = p*x;
  return p;
}
```

Funktionen räknar ut värdet av `x` upphöjt till `n` , där `n` antas vara icke-negativt. Den löpande produkten ackumuleras i den lokala variabeln `p` , och rätt antal varv i loopen erhålls genom att räkna ned parametern `n` från sitt ursprungliga värde till 0 . Säg att vi nu vill använda funktionen för att beräkna värdena 2.7 upphöjt till 3 samt 2 upphöjt till `k` , där `k` är ett inläst tal. Det kan i ett huvudprogram eller motsvarande se ut så här:

```
int      k;
double   x, y, z;
  .
  .
  .
x = power( 2.7, 3 );
cin >> k;
y = 2.0;
z = power( y, k );
  .
  .
```

Vad kommer att hända med det värde som står som andra parameter, dvs. konstanten 3 resp. variabeln `k` , när funktionen anropas? Funktionen räknar ju ner detta värde till 0. Svaret är: ingenting, eftersom det är en värdeanropad parameter, dvs. endast en kopia av värdet skickas till funktionen, och vad denna sedan gör med detta värde påverkar inte motsvarande storhet i det anropande programmet.

6.5 Referensanrop

Hur ska man då förfara om man verkligen vill att en funktion skall kunna påverka värdet av sina parametrar, så att detta får effekt även i det program som anropar funktionen? Då använder man referensanropade parametrar. Ett klassiskt exempel är en funktion som kan få två variabler att byta värde med varandra, som exempel kan vi ta två heltalsvariabler.

Program 6.5

```
// Filnamn: .../swap.cc

void swap( int &a, int &b ) {
  int  temp;
  temp = a;
  a    = b;
  b    = temp;
}
```

Vi ser skillnaden jämfört med föregående exempel, nämligen att parametrarna är referensdeklarerade. Med `&`-tecknet har vi gjort parametrarna till referensparametrar, mer om det nedan.

Nu kommer förändringarna av parametrarnas värden verkligen att avspeglas i ett anropande program:

```
int  i, j;
  .
i = 5; j = 7;
cout << "i and j before swap: " << i << " " << j << endl;
swap( i,j );
cout << "i and j after swap: " << i << " " << j << endl;
  .
```

Utskriften av detta programavsnitt kommer att bli

```
i and j before swap : 5 7
i and j after swap : 7 5
```

Eftersom nu en funktion med referensparametrar kan påverka värdet av motsvarande aktuella parameter, får dessa endast vara sådana storheter som får tilldelas värden, dvs. i princip variabler. Det går alltså inte att byta värde på två konstanter. Det är fullt möjligt, och även vanligt förekommande, att låta en funktion ha både värde- och referensparametrar.

6.6 Använda värdeanrop eller referensanrop

Man kan nu fråga sig om det är bättre att anropa med värde- eller med referensparametrar. Om det krävs att den aktuella parametern skall kunna modifieras av funktionen finns det inget val, vi måste ha referensanrop. Om vi däremot inte kräver modifieringsmöjlighet kan vi välja vilket som. I de flesta fall används då värdeanropet eftersom man då inte av misstag kan ändra den aktuella parametern ty vi hanterar bara en kopia. Det finns dock fall då värdeanrop är klart olämpligt, nämligen då det är stora datamängder som skall hanteras. Vi har hittills bara gått igenom enkla datatyper, men kommer senare till att definiera egna typer, som kan vara mycket stora. Antag att vi har definierat en typ som innehåller en hel skärmsida. En sådan datatyp kan, beroende på storlek och antal färger, innehålla ca 1 Mbyte. Om vi nu har en funktion

```
void PrintScreen( ScreenType OneScreen )
```

och anropar med värdeanrop måste datorn göra en kopia på alla ca 1,000,000 bytes, vilket tar tid. Det vore då bättre att endast skicka en referens:

```
void PrintScreen( ScreenType & OneScreen )
```

Vän av ordning ser nu att det blir tillåtet att ändra i OneScreen i funktionen , även om det inte är meningen. Detta kan man förhindra med hjälp av `const`-kvalificeraren:

```
void PrintScreen( const ScreenType & OneScreen )
```

Tillägget av `const` gör nu att det inte är tillåtet att ha någon sats i `PrintScreen` som kan ändra värdet av `OneScreen`.

```
OneScreen = 3;  // genererar kompileringsfel.
```

Tillsammans ger `const` och & säkerheten att inget kan ändras i aktuell parameter tillsammans med fördelen att stora datamängder inte behöver kopieras vid anropet.

6.7 Default-parametrar

Den bästa översättningen till *default parameters* blir nog underförstådda eller standardparametrar. Vi skall nu titta på vår funktion `power` igen (med en lokal variabel som loopvariabel, vilket brukar vara det vanligaste). Antag att vår power-funktion normalt anropas med `n=2` och att vi skulle vilja slippa att skriva denna 2:a varje gång. Lösningen blir:

Program 6.6

```
// Filnamn: .../power2.cc
#include <iostream>
using namespace std;

double power( double x, int n=2 ) { // default parameter
   double p = 1.0;
   for (int i=1; i<=n; i++) {
          p *=x;
   }
   return p;
}

int main() {
   cout << "Call 1 : " << power( 3.5 , 3 ) << endl;
   cout << "Call 2 : " << power( 3.5 , 2 ) << endl;
   cout << "Call 3 : " << power( 3.5 ) << endl;
   return 0;
}
```

Vi har i definitionen angivit `n=2`. I det sista anropet har vi utelämnat motsvarande aktuella parameter. C++ ”stoppar då in” 2 för att göra anropet fullständigt. Fler än en parameter kan ha default-värde. Regeln är att dessa parametrar måste komma sist i parameterlistan. Följande är tillåtet :

```
void Example( int i, int j=3, double k=4.5 )
```

Följande är inte tillåtet:

```
void Example2( int i=3, int j, double kk=24.5 );
```

Om det finns en deklaration före själva definitionen är det deklarationen som skall ha default-parametrarna utskrivna, i definitionen får de inte finnas.

Program 6.7

```
// Filnamn: .../power3.cc

#include <iostream>
using namespace std;

double power( double x, int n=2 );  // default here!!

int main() {
   cout << "Call 1 : " << power( 3.5 , 3 ) << endl;
   cout << "Call 2 : " << power( 3.5 , 2 ) << endl;
   cout << "Call 3 : " << power( 3.5 ) << endl;
   return 0;
}

double power( double x, int n)  // no default here !!
{
  double p = 1.0;
  for (int i=1; i<=n; i++) {
    p *=x;
  }
  return p;
}
```

6.8 Räckvidd, synlighet och livstid

Detta avsnitt handlar om var i programmet olika variabler kan användas, d v s deras *räckvidd* (eng. *scope*). Vi skall också upptäcka att variabler som finns inte alltid syns, man talar om *synlighet* (eng. *visibility*). Detta beror på att C++ skapar och ”dödar” variabler under programmets gång enligt vissa givna regler. Variablerna har således en viss *livslängd* (eng. *lifetime*).

6.8.1 Globala variabler

Hittills har vi alltid definierat variabler inuti en funktion (`main` eller någon annan). Variabler kan också definieras utanför funktionerna. De kallas då för *globala* variabler. En global variabel kan användas i alla funktioner som definieras efter densammas deklaration. De variabler som deklareras inuti en funktion kallas för lokala variabler och kan inte användas utanför ”sin egen funktion”. Funktioner deklareras/definieras alltid globalt i C++, vilket innebär att en funktion

inte kan finnas inuti en annan funktion. Som huvudregel skall globala variabler undvikas.

6.8.2 Räckvidd

Dessa termer beskriver de regler som avgör då ett namn (på en funktion eller en variabel) kan användas. Den generella regeln säger:

- Ett namn kan användas från deklarationspunkten till slutet av det block, begränsas av {}, i vilken den är deklarerad.

Vi illustrerar med ett exempel:

Program 6.8

```
// Filnamn: .../scope.cc
#include <iostream>
using namespace std;

double d1 = 17.5;       // global variable, scope from here

int main() {
  int i1 = 1;           // local variables
  int i2 = 2;           // scope from here in main function
  if (i1 != i2 ) {
    int i3 = 3;         // Scope within this {}
    cout << i3 << endl;
    int i4 = 4;         // Scope from here
    cout << d1 << endl; //    |
    cout << i4 << endl; // To here
  }
  else {
    cout << i1 << i2 << d1 << endl;    // OK in scope
    cout << i3 << i4 << endl;          // NOT OK, compile error
  }
  return 0;
}
```

Ett speciellt problem uppstår i samband med att man definierar loop-variabeln i en forsats-initiering. Betrakta följande kodavsnitt.

```
int main() {                 // början på yttersta blocket
    for (int i=1; i<5; i++)
        {
          cout << "Loop 1 : << i << endl;
        }
    for (int i=1; i<5; i++)  // Skall bli korrekt!
        {
          cout << "Loop 2 : << i << endl;
```

```
        }

    return 0;
}                           // slut på det yttersta blocket
```

Enligt den nya standarden skall en variabel, som deklareras i en `for`-sats anses tillhöra det block som `for`-satsen omfattar, och när detta lämnas existerar variabeln inte längre. Därför är ovanstående konstruktion tillåten, där det är två olika variabler i som förekommer. Det kan fortfarande finnas en del kompilatorer som följer tidigare standard, och då får man kompileringsfel för denna kod.

6.8.3 Synlighet

C++ tillåter inte att fler än en variabel med samma namn deklareras i samma block. Samma namn kan finnas i nästlade block. Exempel:

Program 6.9

```
// Filnamn: .../nested.cc

#include <iostream>
using namespace std;

double x = 17.5;                    // första x:et

int main() {
  int x = 4;                        // andra x:et
  cout << "Out 1 " << x << endl;
  if ( x > 3 ) {
    double x = 3.5;                 // tredje x:et
    cout << "Out 2 " << x << endl;
  }
  cout << "Out 3 " << x << endl;
  cout << "Out 4 " << ::x << endl; // scope resolution
  return 0;
}
```

I exemplet har vi tre giltiga deklarationer av `x`. Frågan är vilket `x` som gäller var. Programmet kommer att skriva ut följande:

```
Out 1 4
Out 2 3.5
Out 3 4
Out 4 17.5
```

- Vid `Out 1` gäller `int x=4` deklarationen. Denna deklaration kan sägas ha ”lagt sig över” den globala deklarationen.

- Vid `Out 2` gäller `double x=3.5` deklarationen som på samma sätt har lagt sig över `int x=4`.
- Vid `Out 3` har `double x=3.5`:s räckvidd upphört, detta `x` har dött, och `int x=4` gäller.
- Vid `Out 4` har vi använt `::`-operatorn, vilken kallas *scope resolution operator*. Denna "plockar fram" det globala `x`:et.

Vi hade kunna ha en sats `cout << "Out 2.5 " << ::x << endl;` precis efter `Out 2` som naturligtvis hade skrivit ut 17.5. Vi kan dock aldrig komma åt `int x = 4;` deklarationen i `if`-satsens block.

6.8.4 Livstid

Vi har gått igenom räckvidd- och synlighetsregler som avgör vilka variabler som existerar vid en given situation och vilka som är åtkomliga vid namnkollisioner. Variabler har en livstid som inte alltid sammanfaller med dess räckvidd. Vi tar ett trivialt programexempel:

Program 6.10

```
// Filnamn: .../lifetime.cc

#include <iostream>
using namespace std;

double gx = 1;

void Print() {
  int lx = 1;
  lx++;
  gx++;
  cout << "Global=" << gx << " Local=" << lx << endl;
}

int main() {
  for (int i=0; i<5 ; i++ )
    Print();
  return 0;
}
```

Vi har en global variabel `gx` och en lokal `lx`. Utskriften från programmet blir:

```
Global=2 Local=2
Global=3 Local=2
Global=4 Local=2
Global=5 Local=2
Global=6 Local=2
```

Den globala variabeln `gx` ”minns” sitt värde mellan funktionsanropen och inkrementeras från föregående värde. Den lokala variabeln `lx` minns inget mellan funktionsanropen. Det fungerar på följande sätt:

- Globala variabler allokeras i minnet och initieras en gång innan `main`-funktionen börjar exekvera.
- Lokala variabler allokeras och initieras varje gång funktionen anropas. Då funktionen slutar exekvera ”dör” alla lokala variabler.

Ibland vill vi att lokala variabler skall ”få minne” mellan funktionsanropen vilket kan uppnås med `static`-kvalificeraren:

Program 6.11

```
// Filnamn: .../static.cc

#include <iostream>
using namespace std;

double gx = 1;

void Print() {
  static int lx = 1;  // static deklaration
  lx++;
  gx++;
  cout << "Global=" << gx << " Local=" << lx << endl;
}

int main() {
   for (int i=0; i<5 ; i++ )
     Print();
   return 0;
}
```

Vad vi har gjort är att ”kvalificera” den lokala variabeln `lx` som `static`. Detta innebär att den fortfarande fungerar som en lokal variabel vad avser räckvidd och synlighet, men att den har fått minne mellan funktionsanrop. Den initieras på samma sätt som en global variabel och finns hela tiden som programmet exekverar. Utskriften från programmet blir följande:

```
Global=2 Local=2
Global=3 Local=3
Global=4 Local=4
Global=5 Local=5
Global=6 Local=6
```

6.9 Rekursiva funktioner*

En funktion kan anropa sig själv, vilket ger en del intressanta möjligheter. Detta kallas för *rekursiva anrop* och används då problemet i sig kan formuleras rekursivt. Ett enkelt exempel är hur fakultetsberäkning kan definieras:

```
n! beräknas som:
        Om n<=1 så är n!=1      // basfall
        annars n! = n * (n-1)!  // rekursivt fall
```

Basfallet är att $n = 1$ då 1 returneras. Alla andra fall, $n > 1$, är rekursiva fall.

Om vi t ex skall beräkna 2! så är enligt definitionen 2! = 2*(1!). 1! i sin tur är 1. Alltså är 2! = 2*1=2. På samma sätt blir 3! = 3 * 2! = 3 *2 * 1! = 3 * 2 * 1 = 6.

I C++ kan vi definiera en fakultetsfunktion på följande sätt:

Program 6.12

```
// Filnamn: .../fakult.cc

#include <iostream>
using namespace std;

int fakultet( int x ) {
  if (x<=1)
    return 1;
  else
    return x * fakultet(x-1);
}

int main() {
  cout << "Fakultet 4 : " << fakultet(4) << endl;
  cout << "Fakultet 7 : " << fakultet(7) << endl;
  return 0;
}
```

Här ser vi att funktionen fakultet anropar sig själv med inparametern nedräknad ett steg. Denna nedräkning garanterar att vi kommer till stoppvillkoret (`x<=1`) någon gång. Låt oss lägga in en del utskrifter i programmet och sedan köra det för att se vad som händer.

Program 6.13

```
// Filnamn: .../mod_fakult.cc

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

```
void PrintChar( char c, int n ) {
  for (int i=1; i<=n; i++)
    cout << c;
  cout << endl;
}

int fakultet( int x ) {
  int temp;
  PrintChar( '>', x );

  if (x<=1)
    temp = 1;
  else
    temp = x * fakultet(x-1);
  PrintChar( '<',x );
  return temp;
}

int main() {
  int result = fakultet(12);
  cout << "fakultet(12) : " << result << endl;
  return 0;
}
```

Vi har här lagt in utskrifter före och efter själva uträkningen i fakultet. Observera att vi måste formulera om funktionen så att return kommer sist för att alla utskrifter ska komma med. Utskriften från programmet blir:

```
»»»»»»
»»»»»>
»»»»»
»»»»>
»»»»
»»»>
»»»
»»>
»»
»>
»
>
<
«
«<
««
«««<
«««
«««<
««««
«««««<
«««««
«««««<
«««««
fakultet(12) : 479001600
```

Funktionen anropar sig själv 12 gånger innan den ”går ur sig själv” 12 gånger.

6.10 Överlagrade funktioner

Den engelska termen är *overloaded functions*. Idén är följande: Antag att vi vill att vår funktion

```
int max( int,int );
```

även skall gälla för typen `double`. Vid en första betraktelse verkar det omöjligt att definiera en funktion

```
double max( double,double );
```

eftersom C++ inte tillåter att samma namn definieras inom samma block. Funktioner definieras ju alltid globalt och därmed inom samma block. Som parentes kan sägas att C++ inte skulle tillåta att två globala variabler deklareras med samma namn. Funktionerna skulle kunna tänkas deklareras som:

```
int maxi( int,int );
double maxd( double,double );
```

Detta sätt att namnge funktionerna olika blir dock snabbt otympligt, antag t ex att vi vill ha en max-funktion som tar maximum över tre `int`:s.

```
int maxi3( int,int,int );
```

C++ hjälper oss dock ur dilemmat. Det är tillåtet att ha följande deklaration:

```
int max( int,int );
int max( int,int,int );
double max( double,double );
```

C++ skiljer på funktionerna genom att de har olika parameterlistor eller olika signatur. Observera att det inte hjälper att ha olika returvärde om parameterlistan är lika. Följande är inte tillåtet:

```
int max( int,double );
double max( int,double );
```

Låt oss avsluta kapitel med ett max-exempel:

Program 6.14

```
//  Filnamn: .../maxover.cc
#include <iostream>
using namespace std;

int    max( int, int );
int    max( int, int, int );
double max( double, double );
double max( double, double, double );

int main() {
  cout  << " max of 6 3 : " << max(6,3) << "\n";
  cout  << " max of 2 6 3 : " << max(2,6,3) << "\n";
  cout  << " max of 3.5 6.5 : " << max(3.5,6.5) << "\n";
  cout  << " max of 9.8 3.5 6.5 : " << max(9.8,3.5,6.5) << "\n";
  return 0;
}

int max( int a,int b ) {
  if   ( a>b )  return a;
  else          return b;
}

int max( int a,int b,int c ) {
  int temp;
  if   ( a>b )  temp = a;
  else          temp = b;
  if ( temp>c ) return temp;
  else          return c;
}

double max( double a,double b ) {
  if   ( a>b )  return a;
  else          return b;
}

double max( double a,double b,double c ) {
```

```
  return  max( a, max(b,c) );
}
```

Observera speciellt hur den sista max-funktionen två gånger använder sig av den tidigare max-funktionen, den med bara två `double`-parametrar. Detta är inte ett rekursivt anrop.

6.11 Kom ihåg

- Funktioner är bra för att bryta ner programmet i mindre delar.
- Funktioner höjer abstraktionsnivån, vi gömmer detaljerna.
- Returvärde.
- Parameterlista.
- Deklaration kontra definition.
- Namn på parametrar (eller inte).
- `return`-satsen.
- Formella och aktuella parametrar.
- Värdeanrop.
- Referensanrop.
- Stora datamängder som parametrar.
- `const`-kvalificeraren vid funktionsparametrar.
- Flera parametrar, ordning och typ.
- Default-parametrar.
- Räckvidd.
- Synlighet.
- Livstid.
- Globala variabler.
- `static`-kvalificeraren för lokala variabler.
- Rekursiva anrop.
- Överlagrade funktioner.

Kapitel 7

Arrayer

Hittills har vi behövt en variabel för varje värde (heltal, flyttal, sträng) vi vill behandla. Antag att vi har en stor textmängd lagrad – t ex en roman. Antag vidare att vi för att göra viss språkstatistik vill läsa in orden i texten och skriva ut dem i bokstavsordning. Programmet måste då läsa in *alla* ord innan något kan skrivas ut eftersom vi inte vet vilket som skall skrivas först innan alla ord beaktats. En roman kan innehålla hundratusentals ord och det vore klumpigt, minst sagt, att behöva hitta på ett variabelnamn för vart och ett av de lästa orden. För att hantera sådana situationer finns ett begrepp som kallas *array*. En array är en slags tabell som kan innehålla många olika värden (dock skall alla vara av samma typ). Hela arrayen har ett namn och sedan kan vi komma åt de olika värdena genom ett *index* dvs ett heltal. Det första värdet i en array har index 0, det andra index 1 osv.

7.1 Endimensionella arrayer

En endimensionell array är en indicerad (indexerad) sekvens av värden som alla är av samma typ. Vi kastar oss direkt över ett exempel.

Program 7.1

```
// Filnamn: .../ex01.cc

#include <iostream>
using namespace std;

int main() {
  int NumOfItems[3];      // Deklarera en array med 3 int
  NumOfItems[0] = 3;
  NumOfItems[1] = 5;
  NumOfItems[2] = 7;

  double ItemPrice[3] = { 2.50, 7.35, 3.35 };
```

```
  for ( int i=0; i<3; i++ )
    cout << "Value of items with number # " << i
         <<   " is : " << ItemPrice[i]*NumOfItems[i] << endl;

  double StockValue = 0.0;
  for ( i=0; i<3; i++)
    StockValue += NumOfItems[i]*ItemPrice[i];

  cout << "Value of whole stock : " << StockValue << endl;
  return 0;
}
```

- I exemplet börjar vi med att deklarera en array `NumOfItems` som består av 3 st `int`. Därefter tilldelar vi elementen i arrayen värden.
- Observera att elementen indexeras från 0 till 2.
- Vi deklarerar därefter en array av 3 st `double`:s, `ItemPrice`, och initierar denna i deklarationen, jämför skalära variabler.
- Efter deklarationerna används elementen i arrayerna i diverse beräkningar.
- Observera att ett element, t ex `ItemPrice[1]`, används precis som en vanlig skalär variabel.
- Vi kunde i programmet enkelt ha hanterat 100,000 items istället för 3 stycken.

Utskriften från programmet blir:

```
Value of items with number # 0 is : 7.5
Value of items with number # 1 is : 36.75
Value of items with number # 2 is : 23.45
Value of whole stock : 67.7
```

Deklarationen av en array

En deklaration av en endimensionell array ser ut på följande sätt:

typ namn[storlek];

Exempel:

```
int    exA[78];
double exB[94*78];
```

- *typ* är en giltig datatyp, t ex `int`, `double` eller `long`. Senare kommer vi att kunna definiera egna datatyper.
- *namn* följer samma regler för namngivning som skalära variabler.

- *storlek* måste vara en konstant som 10 eller eller ett konstantuttryck som 19*47. *storlek* kan inte vara en variabel, ty kompilatorn måste kunna räkna ut storleken på arrayen och reservera minne (vissa kompilatorer tillåter dock detta).

7.1.1 Index i arrayen

Vi kan visualisera en array som en följd, en vektor, av värden. Så här kan `ItemPrice` illustreras:

index	0	1	2
värden	2.50	7.35	3.35

`ItemPrice` har tre element numrerade från 0 till 2. Dessa ligger direkt efter varandra i minnet. Numreringen brukar benämnas indicering och en array kallas ibland för en indexerad variabel.

- En arrays första element har alltid index 0 och dess sista *storlek* − 1.
- Varje element, ruta i figuren, kan hanteras som en skalär variabel av arrayens datatyp.

De två kanske vanligaste felen som en nybörjare gör är:

- Att blanda ihop ett elements index med dess värde.
- Att glömma att första elementet har index 0 och inte 1!

Låt oss titta på ett program som hanterar en array felaktigt!

Program 7.2

```
// Filnamn: .../ex02.cc

#include <iostream>
using namespace std;

int main() {
  double ItemPrice[3] = { 2.50, 7.35, 3.35 };
  int i = 3;
  int NumOfItems[3];      // declaring an array of ints
  NumOfItems[0] = 3;
  NumOfItems[1] = 5;
  NumOfItems[2] = 7;
  NumOfItems[3] = 42;

  cout << "Value of i : " << i << endl;
  return 0;
}
```

Utskriften från exemplet kan bli (olika system kan ge olika resultat):

```
Value of i : 42
```

Skälet är att vi har adresserat oss utanför arrayen med satsen

```
NumOfItems[3]=42;
```

Variabeln i ”råkar” i exemplet ligga direkt efter arrayen och får därför värdet 42.

Vi kastar oss över ett exempel till:

Program 7.3

```
// Filnamn: .../ex03.cc

#include <iostream>
using namespace std;

int main() {
  const int max = 5;
  int temp[max];

  cout << "Give " << max << " numbers: " << endl;

  for (int i=1; i<=max; i++) {
    cout << "Give number " << i << " : ";
    cin >> temp[i-1];
  }
  cout << "The given numbers in reversed order : " << endl;

  for ( i = max ; i>=1; i-- )
    cout << temp[i-1] << endl;

  cout << "Thanks for today " << endl;
  return 0;
}
```

Programmet läser in 5 tal och skriver ut dem i omvänd ordning:

- Som *storlek* har vi använt en konstant.
- Observera att programmet loopar från 1 till 5, men indicerar arrayen med (`i-1`) dvs 0 till 4. Glöm inte 0:an!
- Om vi vill skriva om programmet så att det läser in 13 tal behövs bara konstanten 5 ändras till 13 och omkompilering av programmet. Om vi inte hade använt en konstant hade vi varit tvungna att ändra på 3 ställen i programmet. I ett större program hade vi behövt ändra på 100-tals ställen. Konstanter är bra!

Arrayer som parameter till funktioner

Ett exempel som visar hur en array överförs som parameter till en funktion:

Program 7.4

```
// Filnamn: .../ex04.cc

#include <iostream>
using namespace std;

double sumArray( const int num, double data[] ) {
  double sum = 0.0;
  for (int i=0; i<num; i++ )
    sum += data[i];
  return sum;
}

int main() {
  const int max = 5;
  double temp[max];

  cout << "Give " << max << " numbers: " << endl;

  for (int i=1; i<=max; i++) {
    cout << "Give number " << i << " : ";
    cin >> temp[i-1];
  }

  cout << "The numbers added : " << sumArray( max,temp) << endl;
  return 0;
}
```

Till funktionen `sumArray` skickar vi dels antalet element i arrayen och dels arrayen själv. Observera att vi måste deklarera `double data[]` för att C++ ska förstå att det är en array och inte en skalär parameter.

Normalt överförs parametrar till funktioner med värdeanrop om inte & anges i deklarationen. För arrayer fungerar det lite annorlunda, se följande exempel.

Program 7.5

```
// Filnamn: .../ex04b.cc

#include <iostream>
using namespace std;

void tenPercent( int num, double data[] ) {
  for (int i=0; i<num; i++ )
```

```
    data[i] *= 1.10;
}

int main() {
  const int max = 3;
  double temp[max] = { 1.0, 2.0, 3.0 };

  cout << "Values before function call : " << endl;
  for (int i=1; i<=max; i++ )
    cout << temp[i-1] << " ";

  tenPercent( max, temp );

  cout << "\nValues after function call : " << endl;
  for ( i=1; i<=max; i++ )
    cout << temp[i-1] << " ";

  cout << endl;
  return 0;
}
```

Om vi nu rekapitulerar reglerna för parameteröverföring ser vi att data inte är deklarerat med & och vi har värdeanrop. Funktionen hanterar en kopia av den aktuella parametern och alla förändringar som sker i `tenPercent` skall inte synas utanför funktionen. När programmet körs får vi emellertid:

```
Values before function call :
1 2 3
Values after function call :
1.1 2.2 3.3
```

Det som skedde inuti funktionen syns i `main`! Förklaringen är att när man har arrayer som parametrar skickas inte arrayens *adress* (med värdeanrop). Vi skjuter upp diskussionen hur detta egentligen fungerar den ett tag. Vad vi ska komma ihåg är att:

- Arrayer som parametrar till funktioner fungerar som om de är &-deklarerade d v s som om vi använder referensanrop

7.2 Flerdimensionella arrayer

Endimensionella arrayer organiserar data i en lång rad, en vektor. Ibland är problemet av den typen att data naturligt organiseras i en rektangel d v s tvådimensionellt. Ett exempel är schack, det vore otympligt om alla pjäser stod på en lång rad istället för en kvadrat. Ett annat exempel är om vi vill mäta temperaturen i ett rum i ett gitter med 1 dm mellan varje punkt. Ett rum är

tredimensionellt och vi skulle då vilja organisera data på så sätt. I C++ kan vi definiera *flerdimensionella* arrayer. En tvådimensionell array deklareras på följande sätt:

```
int twoDarray[2][3] = { { 1 , 2 , 3 },
                        { 4 , 5 , 6 } };
```

Denna array har två rader och tre kolumner. Observera på det sätt som arrayen initieras, som om den vore en endimensionell array med två element som i sin tur är endimensionella arrayer med tre element.

Program 7.6

```
// Filnamn: .../ex05.cc

#include <iostream.h>

int main() {
  int twoD[8][6];

  for (int i=0; i<8; i++ )
    for (int j=0; j<6; j++)
      twoD[i][j] = i*10 + j;

  for (i=0; i<8; i++ ) {
    for (int j=0; j<6; j++) {
      cout.width(8);    // ser till bredd=8 på utskrift
      cout << twoD[i][j];
    }
    cout << endl;
  }
  return 0;
}
```

- Observera att en tvådimensionell array används på samma sätt som en endimensionell med skillnaden att elementen åtkoms med två index: `twoD[i][j]`.
- Egentligen lagras elementen i en tvådimensionell array som en endimensionell array, rad för rad. C++ hjälper oss att ”hitta rätt” genom det dubbla indexet.
- En array kan i C++ ha hur många dimensioner som helst:

  ```
  double Big[100][100][100][100];
  ```

  Kom dock ihåg att `Big` innehåller 100 miljoner `double`, vilket troligen inte får plats i minnet!

7.2.1 Ett större exempel med en 2D-array

Antag att vi ska göra ett program som ska rita diagram på en textskärm genom att rita ut '*' på rätt ställen. I figur 8 har vi plottat en sinuskurva. Vi har inga skalor än men strunt i det tills vidare. Problemet med att göra en sådan här plottning i ett textfönster är att vi måste använda objektet cout vilken har begränsningen att endast kunna skriva åt höger och hoppa en rad ner. Det är inte möjligt att gå åt vänster eller hoppa "uppåt" på skärmen. En lösning är att inte plotta direkt på skärmen utan använda sig av en tvådimensionell array av char som buffert. Denna buffert har då fördelen att vi kan hoppa hur vi vill inom den genom index. Låt oss titta närmare på hur det kan gå till.

Först måste vi bestämma oss för hur stor vår skärm och buffert ska vara. Låt oss deklarera följande konstanter:

```
const int maxRow = 25;       // 25 rader text i diagrammet
const int maxCol = 80;       // 80 tecken bred skärm.
```

En buffertskärm kan då deklareras som:

```
char AScreen[maxCol][maxRow];
```

Vi har kolumner som första index (dvs. x-axeln) och rader som andra index (d v s y-axeln). Låt oss vidare bestämma att origo (dvs. `index[0][0]` ligger längst ner till vänster och `index [maxCol-1][maxRow-1]` ligger längst upp till höger).

I figur 9 är markerat vilka index som hörnen har. Q-rutan har ett index på ca [8][19]. Ett sinusdiagram byggs genom att alla element i bufferten initieras till ett blanktecken (mellanslag) och därefter fylls tecknet '*' i på "rätt" ställen. Låt oss titta på en funktion som initierar hela bufferten med ett givet tecken.

```
void InitScreen( char Screen[maxCol][maxRow], char c ) {
  for (int i=1; i<=maxCol; i++ )
    for (int j=1; j<=maxRow; j++ )
      Screen[i-1][j-1] = c;
}
```

Funktionen InitScreen tar en buffert Screen och ett tecken c som inparametrar. Genom en dubbelloop sätts alla element i Screen till värdet c. Observera att vi inte behöver deklarera Screen med & eftersom arrayer som standard hanteras som referensanrop. I figur 8 som visar en sinuskurva har vi också en ram bestående av B i botten, T i toppen, L till vänster och R till höger. Låt oss se hur en sådan funktion kan se ut:

```
void FrameScreen( char Screen[maxCol][maxRow] ) {
   int i;
   for (i=1; i<=maxRow; i++ ) {
       Screen[0][i-1] = 'L';
       Screen[maxCol-1][i-1] = 'R';
   }

   for (i=1; i<=maxCol; i++ ) {
```

```
      Screen[i-1][maxRow-1] = 'T';
      Screen[i-1][0] = 'B';
    }
}
```

Genom anrop till InitScreen och FrameScreen har vi ett tomt diagram med en ram. Nu kan vi sätta in en sinuskurva:

```
void MakeSinus( char Screen[maxCol][maxRow] ) {
  for (int i=10; i<=70; i++ ) {
    double y = sin( 6.28*(i-10)/60 );
    int col = 5 + int( ( y+1 )/2.0 * 15 );
    Screen[i-1][col-1] = '*';
  }
}
```

Här har vi lagt in diagrammet i kolumnerna 10 till 70. För varje x-värde skalar vi om och beräknar ett sinusvärde i intervallet -1 till 1. Nu måste värdet omvandlas till ett radnummer, `int col`. Vi har valt att använda raderna 5 till 20 för diagrammet. När sedan både kolumn (`i`) och rad (`col`) är uträknade är det bara att stoppa in ett '*'-tecken på. Observera att vi subtraherar 1 i varje index.

När vi har diagrammet "färdigritat" i bufferten måste vi skriva ut den på skärmen. En funktion som utför detta kan se ut som:

```
void PrintScreen( char Screen[maxCol][maxRow] ) {
  for (int j=maxRow; j>=1; j-- )  { // raderna baklänges
    for (int i=1; i<=maxCol; i++ )
      cout.put(Screen[i-1][j-1]);   // tecken för tecken
      cout << endl;                 // radframmatning
    }

  cout << "Give <Return> to continue : ";
  char ch;
  cin.get(ch);
}
```

Den yttre loopen avser raderna eftersom `cout` skriver ut uppifrån och ner. Observera att vi loopar baklänges ty rad 25 måste skrivas ut före rad 24 o s v. För varje rad skrivs varje kolumn ut tecken för tecken. Efter alla tecken i raden skrivs ett radframmatningstecken ut. De tre sista raderna i funktionen är till för att användaren skall få se diagrammet i lugn och ro. När han/hon trycker på <Return>-tangenten avslutas funktionen.

Låt oss plocka samman dessa funktioner till ett helt program.

Program 7.7

```
// File name : .../ex06.cc

#include <iostream.h>
#include <math.h>
```

```
const int maxRow = 25;    // 25 rader text i diagrammet
const int maxCol = 80;    // 80 tecken bred skärm.

//              Deklaration av funktioner

void PrintScreen( char Screen[maxCol][maxRow] );
void InitScreen( char Screen[maxCol][maxRow], char c );
void FrameScreen( char Screen[maxCol][maxRow] );
void MakeSinus( char Screen[maxCol][maxRow] );

int main() {
  char AScreen[maxCol][maxRow];   // Our screen buffer
  InitScreen( AScreen, ' ' );
  FrameScreen( AScreen );
  MakeSinus( AScreen );
  PrintScreen( AScreen );
  return 0;
}

//         Härefter kommer definitionerna av funktionerna
//         Av platsbrist i manuset hoppar vi över dem
//         Programmet i sin helhet kan studeras, se filnamnet.
//*************************************************************
```

Som vi ser blir programmet ganska kort. Det är ganska bra att kunna dela upp ett program på detta sätt. Vi förstår precis vad det gör, men gömmer detaljerna på ett annat ställe, i en annan fil eller som i detta fall längre ner i samma fil.

7.3 Arrayer med tecken — strängar*

Detta avsnitt beskriver hur teckensträngar hanterades i äldre versioner av C++ samt i språket C. Dessa lagrades i arrayer med tecken och hanterades med vissa funktioner. I den nu gällande standarden används den redan diskuterade typen `string`. Vi behåller dock avsnittet i kompendiet tills vidare. Programexemplen i detta avsnitt följer den gamla standarden för `include`-filer.

7.3.1 Initiering av strängar

En speciell typ av arrayer är char-arrayer. En `char` används för att lagra ett teckenvärde t ex 'A' eller '%'. En array av char skulle då kunna tänkas lagra ett ord eller en hel mening:

```
char name[7]       = {'R','i','c','k','a','r','d'};
char profession[9] = {'a',' ','t','e','a','c','h','e','r'};
```

Sådana arrayer skulle sedan kunna tänkas skrivas ut på skärmen med cout:

```
cout << name << " is " << profession << endl;
```

Det sades nyss att en hel array inte kan skrivas ut på en gång. Teckenfält är dock ett undantag (som bekräftar regeln). Det är tillåtet att skriva ut tecken-arrayer med <<. Exemplet ovan skulle dock inte fungera, låt oss exemplifiera med ett program:

Program 7.8

```
// Filnamn: .../cstring01.cc
// Felaktig hantering av strängar
#include <iostream.h>

int main() {
  int i = 0xffffffff;    // största talet på hexadecimal form
  char name[4];
  name[0] = 'R';
  name[1] = 'i';
  name[2] = 'c';
  name[3] = 'k';
  cout << "Hello " << name << endl;
  return 0;
}
```

Utskriften från programmet blir:

```
Hello Rickÿÿÿÿ
```

eller ännu värre.

Anledningen är att << inte vet hur lång arrayen är och således inte vet när `name` tar slut. För att råda bot på problemet har man kommit överens om att definiera en sträng, engelska string, som en array av `char` som avslutas med ett \0 (ett null-tecken). Operatorn << skriver då ut tecken för tecken tills ett '\0'-tecken påträffas.

'R'	'i'	'c'	'k'	'a'	'r'	'd'	Ej en sträng
'R'	'i'	'c'	'k'	\0	'r'	'd'	En sträng med värde "Rick"

Om vi nu vill fixa till `name`- och `profession`-strängarna sen tidigare får vi göra så här:

```
char name[8]        = {'R','i','c','k','a','r','d','\0'};
char profession[10] = {'a',' ','t','e','a','c','h','e','r','\0'};
```

Observera att vi måste ha 8 tecken i `name` för att få plats med '\0'.

Strängar kan initieras på ett enklare sätt:

```
char name[8]      = "Rickard";  // '\0' kommer med underförstått
char profession[] = "a teacher";// C++ räknar själv ut längden
```

Observera skillnaden mellan `'R'` och `"R"`. `'R'` betecknar `char`-värdet och är av storleken en `char` medan `"R"` betecknar strängen, d v s implicit `{'R','\0'}` med storleken 2 tecken.

7.3.2 Funktioner för strängar

Några av de strängfunktioner som C++ tillhandahåller kommer nedan.

Program 7.9

```
// Filnamn: .../cstring02.cc
// Exempel på strängar

#include <iostream.h>
#include <string.h>    // headerfil för stränghantering

int main() {
  char s1[20] = "Rickard";
  char s2[20] = " Filippa";
  char s3[20] = ".";

  cout << "Size of s1 : " << sizeof(s1) << endl;
  cout << "Lenght of s1 : " << strlen(s1) << endl;  // strlen
  cout << "Lenght of s2 : " << strlen(s2) << endl;

  strcat( s1,s2 );                          // strcat
  cout << "Result of strcat : " << s1 << endl ;

  strcpy( s3,"Rickard Enander" );           // strcpy
  cout << "Comparing : " << s1 << " and " << s3 << endl;

  cout << "Result : " << strcmp( s1,s3 ) << endl;        // strcmp
  cout << "Result (3) : " << strncmp( s1,s3,3 ) << endl; // strncmp
  cout << "Result (9) : " << strncmp( s1,s3,9 ) << endl;
  return 0;
}
```

I exemplet använder vi oss av strängfunktionerna `strcpy`, `strcmp`, `strncmp`, `strlen`, och `strcat`. Funktionen `strcpy` kopierar en sträng till en annan. Vi slipper på så sätt flytta tecken för tecken. Funktionerna `strcmp` och `strncmp` jämför strängar och talar om huruvida de är lika eller inte. Med `strlen` kan vi ta reda på hur lång en sträng är och med `strcat` kan vi slå ihop strängar. ”Cat” i `strcat` står för catenation och kan populäröversättas med ”att sätta samman strängar” eller ”konkatenera”. Utskriften från programmet blir:

```
  Size of s1 : 20
  Lenght of s1 : 7
  Lenght of s2 : 8
  Result of strcat : Rickard Filippa
  Comparing : Rickard Filippa and Rickard Enander
  Result : 1
  Result (3) : 0
  Result (9) : 1
```

När det gäller resultatet av `sizeof()` ska man se upp lite, mer om det när vi kommer till dynamiska arrayer.

Ibland vill vi översatta ett strängvärde till talvärden. Ett exempel:

Program 7.10

```
// Filnamn: .../cstring03.cc
// Exempel på sträng till tal

#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

int main() {
  char ns1[20] , ns2[20];
  strcpy( ns1,"38" );          // ns1 has string value "38"
  strcpy( ns2,"3.14" );        // ns2 has string value "3.14"

  int i = atoi( ns1 );         // conversion of str to int
  double d = atof( ns2 );      // conversion of str to double

  cout << "Value of int i : " << i << endl;
  cout << "Value of double d : " << d << endl;
  return 0;
}
```

Utskriften från programmet blir:

```
  Value of int i : 38
  Value of double d : 3.14
```

Omvändningen d v s ett talvärde till en sträng kan också behövas:

Program 7.11

```
// Filnamn: .../cstring04.cc
// Exempel på omvandling från tal till sträng
#include <iostream.h>
```

```
#include <string.h>

int main() {
  char ns1[20] , ns2[20];

  int i = 38;
  double d = 3.14;

  sprintf(ns1,"%6d",i);     // conversion of int to string
  sprintf(ns2,"%8.3f",d);   // conversion of double to string

  cout << "Value of ns1 : " << ns1 << endl;
  cout << "Value of ns2 : " << ns2 << endl;
  return 0;
}
```

Utskriften från programet blir:

```
value of ns1 : 38
value of ns2 : 3.140
```

Observera i dessa exempel vilka header-filer som måste inkluderas.

7.3.3 In- och utmatning av strängar

När strängar skall matas in från tangentbordet finns det en del saker att tänka på. Betrakta följande program och körexempel:

Program 7.12

```
// Filnamn: .../cstring05.cc
// Felaktig inmatning av strängar med cin

#include <iostream.h>

int main() {
  char name[30], car[30];

  cout << "What is your name ? ";
  cin >> name;
  cout << "What car do you drive ? ";
  cin >> car;

  cout << name << " drives a " << car << endl;
  return 0;
}
```

Körexempel:

```
What is your name ? Clint Eastwood
What car do you drive ? Clint drives a Eastwood
```

Programmet tillät mig inte tala om att jag kör en Volvo och dessutom hugger det av mitt namn till Clint samt tror att jag kör en Eastwood!

Förklaringen är att en sträng skall avslutas med ett `'\0'` som inte går att skriva in på tangentbordet. Därför har `>>` ett annat sätt att avskilja inmatningsenheter från varandra, i vårt fall de två strängarna `name` och `car`. Den använder *vita tecken* (eng. *white spaces*) som avskiljare. Ett vitt tecken är ett mellanslag, ett tabulator- eller ett returtecken. I vårt fall tyds mellanslaget mellan Clint och Eastwood som avskiljare mellan två strängar. Strängen `name` får värdet `Clint` och strängen `car` värdet `Eastwood`, vilket förklarar utskriften av mitt bilval.

Nu finns det en väg ur problemet. Objektet `cin` har flera alternativa sätt att läsa in. Här är ett exempel:

Program 7.13

```
// Filnamn: .../cstring06.cc
// Inläsning med cin.getline

#include <iostream.h>

int main() {
  const int Size = 20;
  char name[Size], car[Size];

  cout << "What is your name ? ";
  cin.getline( name,Size );
  cout << "What car do you drive ? ";
  cin.getline( car,Size );

  cout << name << " drives a " << car << endl;
  return 0;
}
```

Med `cin.getline( name, Size )` läser vi in tills dess att ett radslutstecken (d v s <Return>) påträffas eller tills dess att `Size` tecken har lästs in. På detta sätt kan vi aldrig gå utanför det minnesutrymme som `name` har sig tilldelat. Metoden `cin.getline` byter sedan automatiskt ut radslutstecknet mot ett `'\0'` och strängen är definierad.

I de två följande exemplen ser vi att `cin` har ytterligare varianter. Vi betraktar dessa två exempel som lite av överkurs!

Program 7.14

```
// Filnamn: .../cstring07.cc
// Felaktigt exempel med cin.get()

#include <iostream.h>
int main() {
  const int Size = 20;
  char name[Size], car[Size];

  cout << "What is your name ? ";
  cin.get( name,Size );
  cout << "What car do you drive ? ";
  cin.get( car,Size );

  cout << name << " drives a " << car << endl;
  return 0;
}
```

Här har vi bara bytt ut `cin.getline` mot `cin.get`. Exemplet blir fel som vi ser i det övre körexemplet i figuren nedan:

```
> cstring07
What is your name ? Clint Eastwood
What car do you drive ? Clint Eastwood drives a
>
>
> cstring08
What is your name ? Clint Eastwood
What car do you drive ? Volvo
Clint Eastwood drives a Volvo
>
```

När vi kör programmet ignoreras den andra frågan helt och hållet. Anledningen här är att `cin.get` läser fram till radslutstecknet, men inte själva tecknet. Tecknet ligger därför kvar till den andra inläsningen. Denna läser då fram till just det gamla radslutstecken, men det finns inget kvar att läsa och `car`-strängen blir därför tom!

Nu kan `cin.get` klara av att även ”läsa bort” radslutstecknet och vilka andra enskilda tecken som helst. I exemplet nedan, som också körs i rutan ovan har vi fixat programmet så att det fungerar som tänkt.

Program 7.15

```
// Filnamn: .../cstring08.cc
// Fixat program med cin.get
```

```
#include <iostream.h>

int main() {
  const int Size = 20;
  char name[Size], car[Size];

  cout << "What is your name ? ";
  cin.get( name,Size );         // cin.get för sträng
  cin.get();                    // läs bort <Return>
  cout << "What car do you drive ? ";
  cin.get( car,Size ).get();    // som ovan fast elegantare!
  cout << name << " drives a " << car << endl;
  return 0;
}
```

7.3.4 Ett litet strängproblem

Inläsning av strängar brukar vara ett knepigt område i de flesta programmeringsspråk. Hittills har stränginläsningen varit tämligen rakt på i C++. Använd `cin.getline( str,size)` så fixar sig det mesta. Nu kommer vi dock till ett litet aber när inläsning av tal med `cin >> tal;` och inläsning av strängar med `cin.getline(str, size)` blandas. Tyvärr kan lite olustiga saker hända i den situationen vilket illustreras i programmet och körexemplet nedan.

Program 7.16

```
// Filnamn: .../cstrin09.cc
// Exempel med cin och cin.getline (felaktigt)

#include <iostream.h>

int main() {
  const int Size = 20;
  char name[Size], car[Size];
  int num;

  cout << "How many cars do you have : ";
  cin >> num;                       // först ett tal
  cout << "What is your name ? ";
  cin.getline( name,Size );         // sedan en sträng
  cout << "What car do you drive ? ";
  cin.getline( car,Size );

  cout << name << " drives a " << car << endl;
  cout << "and has " << (num-1) << " other cars" << endl;
  return 0;
}
```

Körexempel:

```
How many cars do you have : 4
What is your name ? What car do you drive ? Volvo
drives a Volvo
and has 3 other cars
```

Situationen liknar den i föregående exemplet då en sträng lästes in som tom (med längd 0) beroende på att `cin.get()` inte läser in radslutstecknet <Return>. I exemplet läses den första strängen efter `cin >> num;` in som tom och anledningen är i princip densamma: `cin >> num;` läser inte in vita tecken dvs <Return> i vårt fall. <Return>-tecknet ligger kvar till nästa inläsning att ta hand om, `cin.getline(name, size);` som genast påträffar detta. Aha, säger `cin.getline()`, inläsningen är klar och avslutad med längd 0! Nästa inläsning med `cin.getline()` klarar sig däremot bra eftersom den föregående såg till att ta bort det "hängande" <Return>:et. Det finns nu flera lösningar att ta till.

1. Den samvetsgranna programmeraren skulle antagligen inte läsa in tal med `cin >> num`, ty denna inläsning är felkänslig. Prova t ex att läsa in "kurt" istället för ett tal. Hon skulle antagligen läsa in en sträng, kolla att det verkligen är ett tal och i så fall omvandla denna sträng till talet. Om den inlästa strängen visar sig vara ett "icke-tal" skulle han meddela användaren och kräva en ny inläsning. Allt detta skulle hon placera i en funktion som han anropar vid behov. Proceduren låter lite krånglig så låt oss titta på en genväg.

2. Vi som är lite lat gör på följande sätt. Eftersom problemet är att ett <Return> inte försvinner efter `cin >> num` tar vi helt enkelt och läser bort det med `cin.get()`, precis som tidigare. Programmet kan då se ut som följer:

Program 7.17

```
// Filnamn: .../cstring10.cc
// Hur inläsning av tal och strängar klaras

#include <iostream.h>

int main() {
  const int Size = 20;
  char name[Size], car[Size];
  int num;

  cout << "How many cars do you have : ";
  cin >> num;
  cin.get();                  // saved by the bell!
  cout << "What is your name ? ";
  cin.getline( name,Size );
```

```
  cout << "What car do you drive ? ";
  cin.getline( car,Size );

  cout << name << " drives a " << car << endl;
  cout << "and has " << (num-1) << " other cars" << endl;
  return 0;
}
```

Nu fungerar programmet som det ska. Det finns dock ytterligare ett aber, användaren kan ”trilskas” genom att ge ett eller flera mellanslag efter det tal som skall läsas in.

```
How many cars do you have : 45    <Return>
```

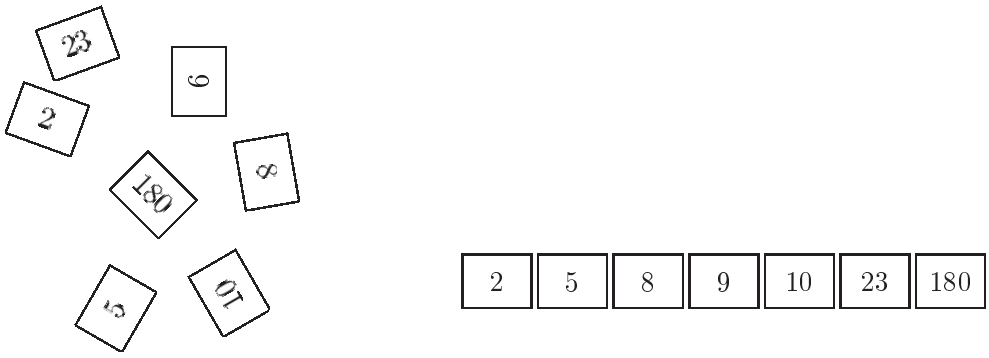
Då kommer `cin.get` att läsa bort det första mellanslaget och det förhatliga <Return>:et finns kvar och gäckar oss.

7.4 Kom ihåg

- Array är en följd av likadana element.
- Deklaration med storlek. Exempel: `double Arr[78]`.
- Indicering. Exempel: `Arr[i]` hanteras som en skalär variabel.
- Flerdimensionella arrayer.
- Strängar: array med tecken med ett null-tecken sist.
- Funktioner för strängar.
- Omvandling mellan tal och sträng.
- Inmatning med `>>`, `cin.getline()` ocherb!cin.get()!.
- Blandad inmatning, tal och strängar.

Kapitel 8

Sökning och sortering



Sortering och sökning hör till de klassiska algoritmproblemen. Det finns en stor mängd metoder men här skall vi bara titta på några av de enklaste. Med *sökproblemet* menas problemet att bland en mängd element lokalisera ett element med ett visst innehåll och med *sorteringsproblemet* avses problemet att ordna om ett antal element i storleksordning (i någon mening). Vi kommer bara behandla data lagrade i arrayer men det finns andra datastrukturer som används i dessa sammanhang (typiskt filer och listor). För enkelhetens skull kommer vi anta att de data vi behandlar är heltal. I verkligheten har man i regel mer komplexa data men de grundläggande algoritmerna är oberoende av datatyp.

8.1 Sökning

Med *linjär sökning* menas att man börjar i ena ”änden” av mängden och i tur och ordning tittar på elementen för att se om det är det eftersökta. Följande funktion illustrerar algoritmen.

Program 8.1

```
// Filnamn: .../linSearch.cc
// Linjär sökning i array

int linSearch( int data[], int antal, int key ) {
  // Om key finns med i data returneras dess index, annars -1
    for ( int i=0; i<antal; i++ )
        if ( data[i]==key ) return i;
    return -1;
}
```

Ovanstående funktion kan behöva göra ganska många jämförelser för stora arrayer. Om elementen i arrayen är sorterade i storleksordning kan en väsentligt snabbare metod som kallas binär sökning användas. Idén här är att man börjar att titta i på elementet i mitten av arrayen. Om det sökta elementet är mindre än det mittersta så kan man koncentrera arbetet på första halvan, om det är större så fortsätter man med den andra halvan. Detta upprepas antingen tills man hittar elementet eller tills intervallet inte längre innehåller några element.

Program 8.2

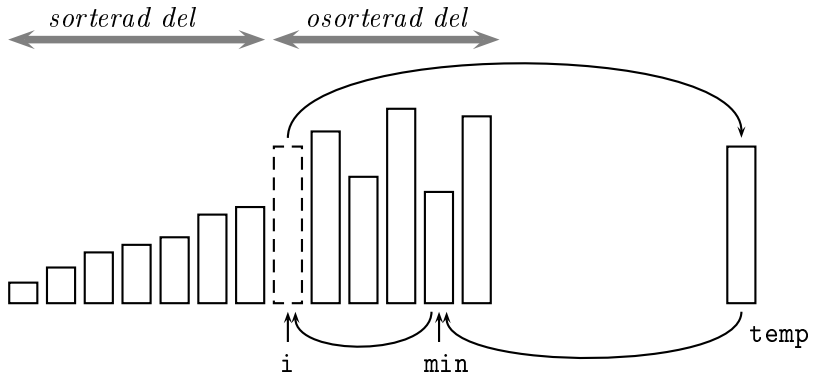
```
// Filnamn: .../binSearch.cc
// Binär sökning

int binSearch( int data[], int antal, int key ) {
    int first = 0, last = antal - 1;
    while ( first <= last ) {
       int mid = (first + last)/2;
       if ( data[mid] == key )
          return mid;
       else if ( key < data[mid] )
          last = mid - 1;
       else
          first = mid + 1;
     }
     return -1;
}
```

8.2 Sortering

För att binärsökning skall fungera måste elementen i arrayen vara sorterade. Nedanstående program demonstrerar en metod som brukar kallas *urvalssortering*. Den går till så att man letar upp och det minsta av elementen och byter

plats på det minsta och det första. Därefter letar man upp det minsta av de kvarvarande elementen och byter plats på det och det näst första o s v.

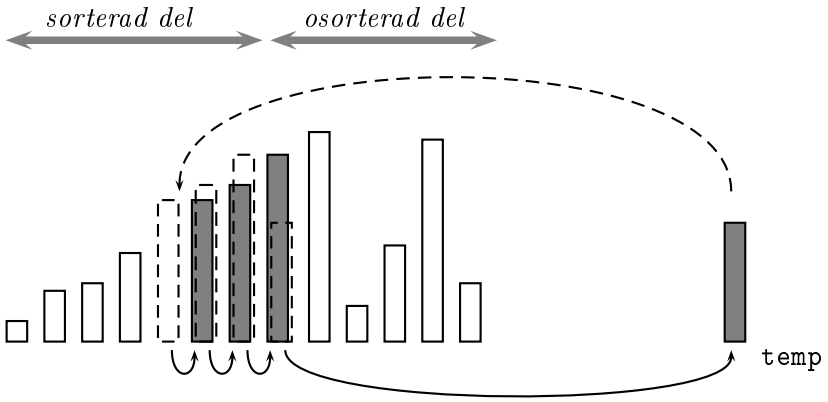


Program 8.3

```
// Filnamn: .../selectionSort.cc
// Urvalssortering

 void selSort( int data[], int antal ) {
      for ( int i=0; i<antal-1; i++ ) {
      int min = i; // "Gissa" att minsta finns på plats i
      for ( int j=i+1; j<antal; j++ ); // Se om det finns mindre
          if ( data[j] < data[min] )
             min = j;
                 // Byt plats på det minsta och det "första"
             int temp = data[i];
             data[i] = data[min];
             data[min] = temp;
      }
}
```

En nackdel med ovanstående metod är att den inte drar fördel av att elementen redan ligger ”bra”. Nedanstående metod, kallad *insticksortering* klarar detta. Idén är att man betraktar arrayen som bestående av två delar: den vänstra, redan sorterade delen och den högra, osorterade. Från början består den sorterade delen av endast det första elementet. Sedan bygger infogar man elementen från den osorterade del i den sorterade delen så att sorteringen bibehålls. När alla element är infogade är sorteringen klar.



Program 8.4

```
// Filnamn: .../insertionSort.cc
// Instickssortering

void insSort( int data[], int n ) {
     for (int i=1; i<n; i++) {
            // Element på plats i skall infogas bland
            // elementen på platserna 0, 1, ... i-1
         int temp = data[i];
            // Flytta undan tills rätt plats funnen
         for ( int j = i; j>0 && k < data[j-1]; j-- )
             data[j] = data[j-1];
         data[j] = temp;
     }
}
```

Ingen av ovanstående metoder är bra om verkligt stora mängder skall sorteras. En sortering av en miljon element med ovanstående metod tar timmar medan en ”avancerad” metod som quicksort, mergesort eller heapsort klarar den uppgiften på några få sekunder. Dessa metoder ligger dock utanför denna kurs.

Sakregister