ÖREBRO UNIVERSITET

INSTITUTIONEN FÖR TEKNIK

Gunnar Joki Oru-Te-2005-41

Programmering C, 5p

Innehållsförteckning

1 Datorer och program

Idag är datorn ett av de viktigaste och vanligaste verktyget i vårt samhälle. På de flesta arbetsplatser finns det idag datorer. Vad använder man datorerna till? Vad gör en dator? Datorer kallades tidigare för datamaskiner. Den första delen av ordet datamaskin är data. Vad är data? *Data och information* hänger ihop enligt :

Ex: Data : 5, A, R, ?, >, ...
Information : Hej, 222222-2222, ...

Som exemplet visar kan man sammansätta data till information. Data kan vara bärare av information. En dator har med information att göra och eftersom information har hög prioritet i dagens samhälle har datorn blivit ett viktigt verktyg.

Vad gör en dator med informationen? Den andra delen av ordet datamaskin är maskin. När man hör ordet maskin tänker man på bearbetning och automatik. En symaskin bearbetar tyger och tråd så att man får kläder. *En dator eller datamaskin bearbetar information till ny information*.

En dator är en maskin som kan

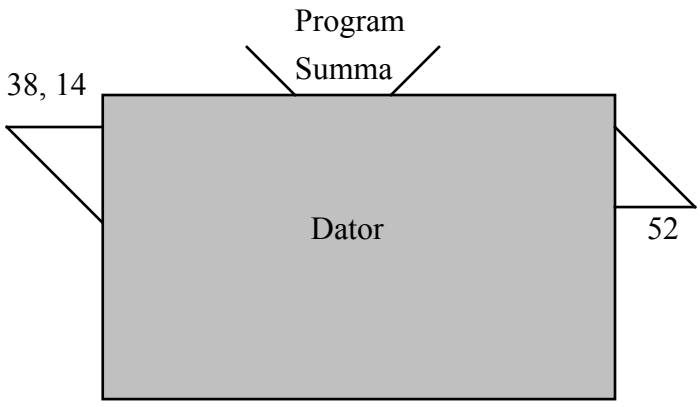
- hämta in information

- bearbeta information

- skicka ut ny information

Hur datorn ska hämta in, bearbeta och skicka ut information bestäms av det *program* som datorn för tillfället är laddat med.

Ex: Programmet summa som tar in information i form av två tal och skickar ut ny information om summan av talen.



1.1 Program

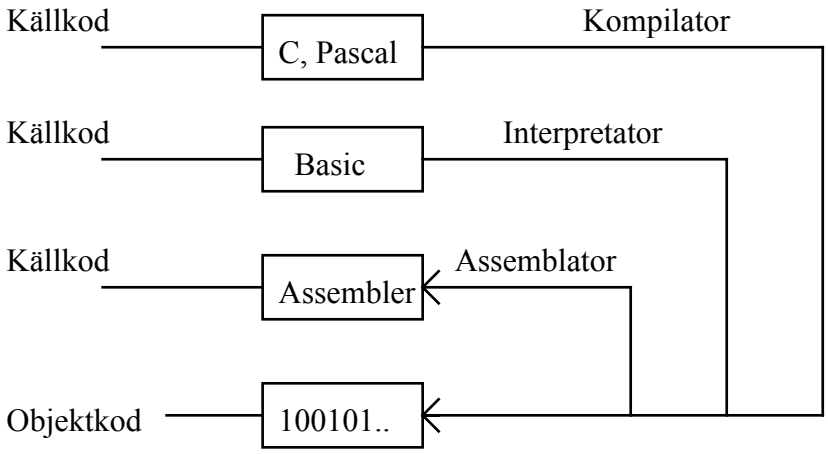
Programmet är datorns arbetsplan. *Steg för steg måste programmet ange vad som ska göras för att korrekt informationsbehandling ska fås*. Hårdvaran eller elektroniken i en dator är uppbyggd av digitala komponenter som bara kan anta två lägen, nämligen 0 eller 1.

Slutprodukten, av de data eller det program som datorn laddas med, får därför bara innehålla ettor och nollor. Man säger att programmets instruktioner är i binär- eller maskinkod. Varje datortyp har sin egen speciella maskinkod. I datorns barndom fick man skriva sina program direkt i den maskinkod, som den aktuella datorn förstod. Detta var mycket tidskrävande då det krävs många ettor och nollor för att skriva den enklaste information.

Ex: Talet 38, som matades in till datorn ovan, skrivs i binärkod som 100110

Bytte man datortyp fick man snällt skriva om hela programmet i nya kombinationer av ettor och nollor.

Idag finns det inbyggda program i datorerna som översätter från språk på högre nivå, med fler symboler som i Basic, C och Pascal, till en slutprodukt i aktuell maskinkod. Det finns tre olika huvudtyper av översättningsprogram interpretatorer, kompilatorer och assemblatorer.



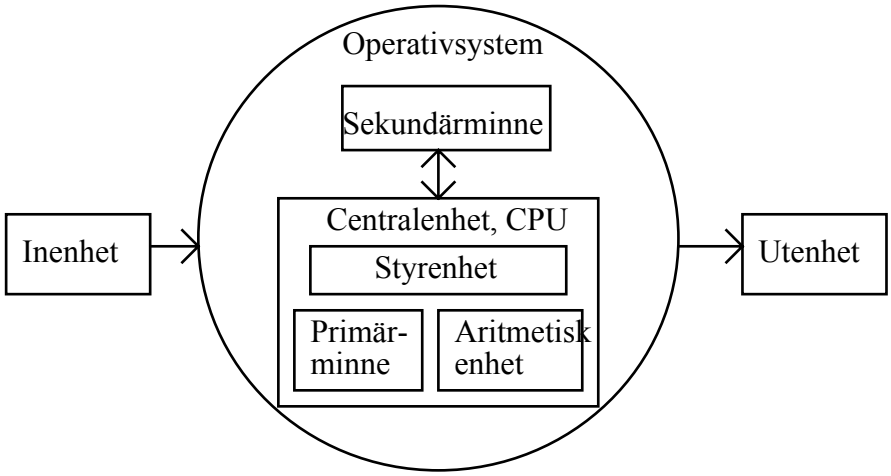
Kompilatorn är ett översättningsprogram som översätter hela den skrivna programtexten (källkoden) till maskinkod (objektkod). Vid körningen används den översatta objektkoden.

Interpretatorn översätter programtexten radvis till maskinkod. Vid körning används källkoden och översättning till objektkod måste göras rad för rad. Körningen blir därför långsammare än då en kompilator används.

Assemblatorn översätter lågnivåspråket assembler till maskinkod. Assembler är ett språk som enkelt och direkt kan översättas till ettor och nollor. Med en omvänd assemblator kan man översätta åt andra hållet från maskinkod till assembler. Det finns inga motsvarande omvända kompilatorer eller interpretatorer som översätter från maskinkod till högnivåspråk.

1.2 Datorns funktion

En dator består av hårdvara (maskinvara) och mjukvara (programvara). Den består av huvuddelarna inenhet, utenhet, centralenhet, sekundärminne och operativsystem.



I centralenheten, CPU:n, finns styrenhet, aritmetisk enhet och primärminne. Styrenheten har direkt kontakt med alla enheter. Den styr och fördelar arbetet inne i centralenheten.

Den aritmetiska enheten utför beräkningar och gör jämförelser. I primärminnet lagras program och data under körning.

Sekundärminnet används för lagring av program och data mellan körningarna.

In- och utenhet är idag oftast en terminal med tangentbord och skärm.

Då man ska använda en dator och köra ett program måste man hämta programmet från ett utrymme (fil) i sekundärminnet och placera det i primärminnet. När programmet körs måste körningen övervakas och eventuella in- och utmatningar registreras etc.

I datorns barndom sköttes allt detta av en operatör. Det tog lång tid och var mycket besvärligt att köra ett program. Med tiden har det utvecklats speciella serviceprogram, *operativsystem*, som ersatt operatörerna. Det finns många olika operativsystem bland annnat:

DOS
Unix
WindowsNT

Man använder dessutom hjälpsystem, *Windows*, som kan ses som ett skal ovanför operativsystemet, för att få ett *användarvänligare gränsnitt* mot datorn. I WindowsNT är dessa båda, operativsystem och Windows, sammanfogade till ett fullständigt och användarvänligt operativsystem.

1.3 Minnen

Minnet i centralenheten kallas primärminne. Det består oftast av två delar:

1) ROM-del (Read Only Memory)

2) RWM-del (Read Write Memory)

ROM-delen är programmerad av tillverkaren med exempelvis operativsystemet eller delar av detta. ROM-delen kan användaren endast läsa. RWM-delen är den del som användarens program och data laddas in i under körning. I stället för RWM används ofta beteckningen RAM (Random Access Memory). I ett RAM är åtkomsttiden oberoende av var i minnet data är lagrat. Detta gäller förvisso även ROM. Primärminnet är oftast av halvledartyp (chips). Härmed blir minnet snabbt men relativt dyrt per lagrat tecken. RWM-delen töms då datorn slås av.

För att lagra data och program då datorn är avslagen används sekundärminnet, som oftast är av magnetisk typ. Detta är relativt långsamt men billigt.

Ett minnes kapacitet mäts i Byte (B). 1B består av 8 bitar i form av ettor eller nollor enligt exempelvis :

01000001

Informationen som sparas i minnet är kodad i någon form av kod bestående av ettor och nollor. Exempelvis kan de 8 bitarna ovan vara koden för ett tal eller koden för en bokstav enligt :

Kod	Tal	Bokstav
01000001	65	A

För större datamängder som texter, ljud och bilder behövs naturligvis flera Byte. Man brukar säga att en A4-sida skriven text innehåller ungefär 1500 tecken alltså 1500B. För större minnesutrymmen används enheterna kilobyte, megabyte resp gigabyte enligt :

$$1\text{KB} = 2^{10}\text{B} = 1024\text{B}$$
$$1\text{MB} = 2^{10} \cdot 2^{10}\text{B}$$
$$1\text{GB} = 2^{10} \cdot 2^{10} \cdot 2^{10}\text{B}$$

En vanlig diskett kan innehålla 1.44 MB alltså ca 1000 A4-sidor text.

1.4 Filer

När man stänger av datorn ska man spara sin information på sekundärminnet (hårddisk eller diskett), annars försvinner den. För att hitta informationen nästa gång man ska använda den, måste man på något sätt ge den en identitet. Man delar upp sekundärminnet i *filer* och sparar informationen på en sådan. Varje fil ger man ett namn. Hur namnet får se ut bestäms av det aktuella operativsystemet.

Ex: I WindowsNT anges namnet på en fil av *filnamn.filtyp* enligt:

personregister.c
personregister.cpp
persondata.dat
persondata.doc
adressbok.txt

Filnamnet ska tydligt ange vad filen innehåller för information. Filtypen, som skrivs efter punkten, anger vilken typ av information som finns i filen. Filtypen c anger att filen innehåller källkoden till ett program skrivet i språket C och filtypen cpp anger att källkoden är skriven i språket C++. Filtyperna doc och txt brukar användas för textfiler som exempelvis skapats med ordbehandlingsprogram och som kan visas på skärmen eller skrivas ut på printer. Filtypen dat markerar att filen innehåller information i binärkod och därför exempelvis ej kan skrivas ut med printer.

Det finns ingen direkt begränsning vad gäller antalet tecken eller vilka tecken som ska användas i WindowsNT. Ska man däremot utnyttja sina filer i operativsystemet DOS måste man hålla sig till *maximalt 8 tecken före punkten* och *maximalt 3 tecken efter punkten*.

Ex: I DOS anges namnet på en fil av *filnamn.filtyp* där filnamn får vara *maximalt 8* och filtyp *maximalt 3* tecken enligt:

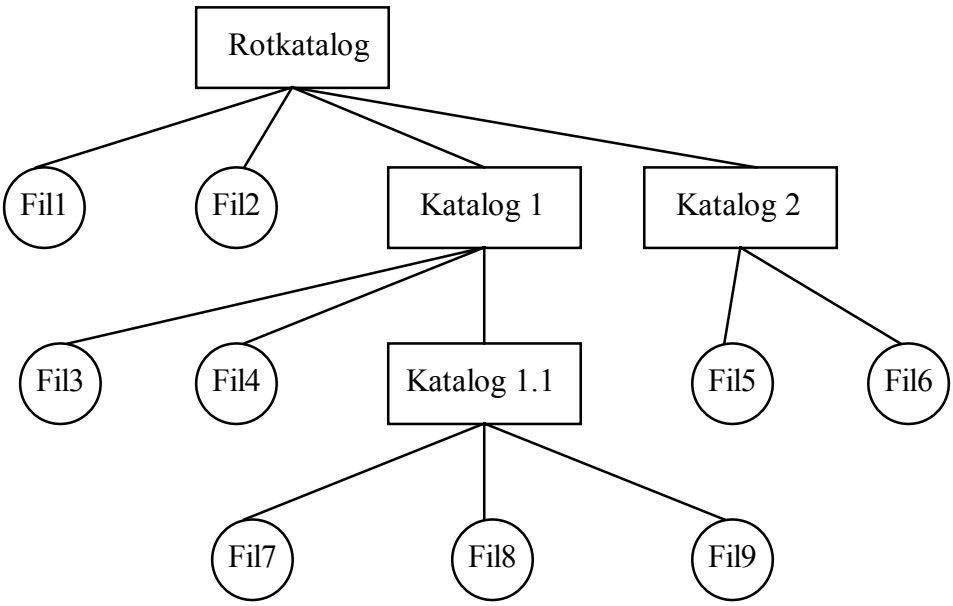
persreg.c
persreg.cpp
persdata.dat
persdata.doc
adrbok.txt

Sparar man information på samma fil en gång till försvinner den gamla versionen. Vissa editorprogram brukar dock döpa om den gamla filen till en fil med samma namn men med filtypen bak istället.

Filer kan man skapa på olika sätt. Man kan använda ett editorprogram eller ett ordbehandlingsprogram med vars hjälp man kan skriva sina egna filer. Man kan skriva egna program som skapar filer. Man kan också köpa filer på diskett eller CD etc.

Har man många filer kan det vara svårt att hålla ordning på dessa och svårt att hitta en speciell fil, som man vill titta på. Det blir långa söktider om man har alla filer i en enda hög. Jämför med en pärm som man har en massa papper i utan någon ordning. Det är inte alldeles lätt att hitta det papper man letar efter, även om pappren är namngivna.

För att snabba upp sökningen av filer brukar man dela in sina filer i kataloger. Man har en huvudkatalog eller rotkatalog (pärm) som är indelad i ett antal underkataloger (flikar) som i sin tur är indelade i underkataloger o.s.v. Man får ett filträd där det blir lättare att hitta enligt:



Med hjälp av operativsystemet kan man exempelvis:

- skapa underkataloger
- flytta filer mellan kataloger
- kopiera filer
- titta vilka filer som finns i resp katalog
- flytta sig till en viss katalog

Rotkatalogen betecknas på olika sätt i olika operativsystem. Disponerar man flera enheter sekundärminne exempelvis diskettstationer, hårddiskar etc kan man naturligtvis ha flera filträd (pärmar) med olika beteckningar.

1.5 Programutveckling

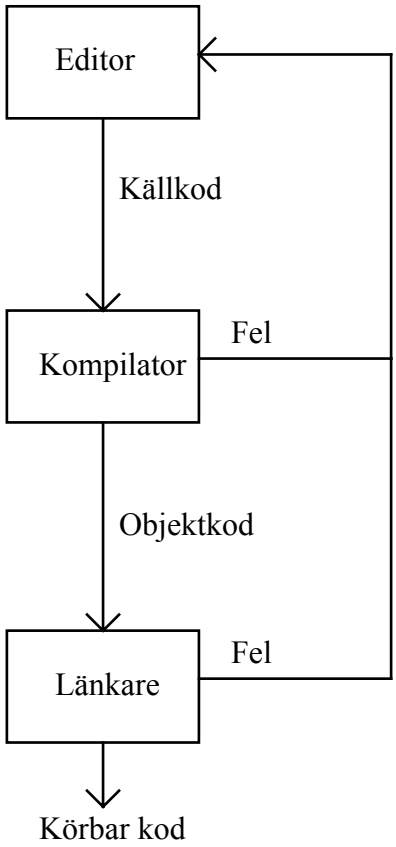
När man ska utveckla egna program i något språk måste man ha verktyg i form av *editor*, *kompilator* och *länkare*.

Med *editorn*, som är ett ordbehandlingsprogram, skriver man sin källkod. När man skrivit färdigt går man ur editorn och sparar sitt program eller sin källkod i en fil.

Nästa steg i programutvecklingsprocessen är att man använder *kompilatorn* för att kompilera sitt program. Är källkoden felaktigt skriven får man ändra den med hjälp av editorn och sedan kompilera om tills källkoden är felfri. Går kompileringen igenom utan fel har man fått sin källkod översatt till maskin- eller objektkod.

Maskinkoden är ej direkt körbar utan man måste med *länkaren* länka ihop den med vissa standardrutiner för exempelvis in- och utmatning etc. Går länkningen bra och utan fel har man fått en körbar eller exekverbar kod.

Gången vid programutveckling är:



Ex: Man vill skriva ett C-program som på skärmen skriver ut texten Hej!

Man börjar med att skriva källkoden med hjälp av en editor enligt :

```
#include <stdio.h>

void main()
{
   printf(”Hej!”);
}
```

Sedan sparar man med hjälp av ett editorkommando ovanstående kod i en fil exempelvis *hej.c*.

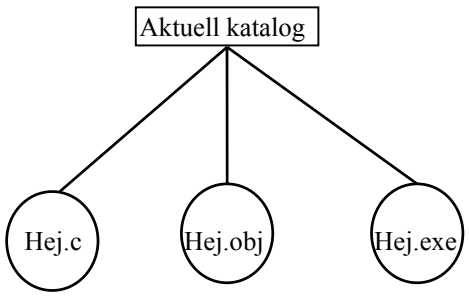
Filen hej.c ska man nu kompilera. Kompilatorn är idag ofta inbyggd i samma interaktiva miljö som editorn, vilket innebär att man kan kompilera programmet med ett kommando i denna interaktiva miljö. Efter kompileringen får man i sin aktuella katalog filen *hej.obj*, som innehåller objektkoden (maskinkoden) för programmet hej i binärt format.

När kompileringen går felfritt är det dags att länka ihop programmet med eventuella extra program som behövs. I ovanstående fall så behövs utskriftsfunktionen printf, som länkaren då länkar in tillsammans med den kompilerade filen hej.obj. Går länkningen bra får man en exekverbar (körbar) fil i sin katalog som heter *hej.exe*. Länkaren finns också idag i samma miljö som editorn och kompilatorn.

Nu är det dags att exekvera (köra) programmet som skapats ovan. Detta kan man också göra från samma interaktiva miljö som man editerat, kompilerat och länkat i. Man väljer kör-kommandot och programmet körs. På skärmen skrivs :

```
Hej!
```

Tittar man i katalogen som man arbetat i ser man följande filer:



Man kan skapa olika exe-filer beroende på under vilket operativsystem de ska köras. Man kan exempelvis skapa en exe-fil för DOS och en för WindowsNT. I den interaktiva miljön som man arbetar i brukar man kunna ange vilken typ av plattform man ska skapa program för.

1.6 Programspråket C

Språket C utvecklades då man fick ett behov av ett språk på högre eller mer generell nivå än assembler. På Bell-laboratoriet i USA utvecklade man på 70-talet operativsystemet Unix. Man skrev det i assembler vilket innebar att man måste skriva nya versioner av operativsystemet för varje ny typ av datorsystem. För att höja nivån på sitt språk utvecklade man först språket B, som ganska snart gjordes om och fick då namnet C.

Språket C utvecklades med följande målsättningar:

- möjligheten att programmera på låg nivå skulle finnas kvar
- standardiserade anrop av systemrutiner som tidshantering mm
- vara ett generellt högnivåspråk av samma typ som Pascal, Fortran etc

Möjligheten att kunna programmera på låg nivå var en viktig målsättning eftersom C skulle användas för realtidstillämpningar typ operativsystem etc. I många av de högnivåspråk som fanns då på 70-talet fick man lämna den aktuella högnivåmiljön och gå över till assembler-programmering, för den aktuella processorn, för att komma åt exempelvis enskilda bitar i register.

Ex: Lågnivåegenskaperna har man infört i form av funktioner eller operatorer på C-nivå. Normalt kan man i ett högnivåspråk bara komma åt hela Byte men i C kan man exempelvis testa om en bit är 1 eller man kan skifta bitarna enligt:

```
c1 = 5;          /* 00000101 */
c2 = c1 << 3     /* 00101000 */
```

Man har lyckats bra med lågnivåmålsättningen. Från att tidigare varit tvungen att skriva halva koden i högnivåspråk och resten i assembler kan man idag med C skriva 90% på hög nivå. Detta sparar mycket tid då systemet ska skrivas om för en ny datortyp.

När det gäller standardiserade systemanrop har man infört ett gränssnitt med vars hjälp man kan komma åt dessa med samma C-namn oberoende av vilken dator man använder. Varje C-kompilator anropar den korrekta funktionen just för den datorn.

Ex:
```
#include <stdlib.h>   /* Här finns bl.a slumptalsgenerering */
#include <time.h>     /* Här finns tidsfunktioner */
#include <stdio.h>    /* Här finns i/o-rutiner */
```

När det gäller målsättningen att vara ett generellt högnivåspråk finns exempelvis möjligheterna att utnyttja selektioner (val) och iterationer (upprepningar), som i alla andra högnivåspråk.

1.7.1 Programexempel

Ex: Skriv ett program som läser in radien för en cirkel och beräknar och skriver ut cirkelns area.

```
/* Cirkel.c v1.0           */
/* Gunnar Joki GJI         */
/* ANSI C                  */
/* Arean av en cirkel      */

#include <stdio.h>    /* printf, scanf */

void main()
{
   const float pi = 3.14159;
   float radie, area;

   /* hämta in ett värde till radie */
   printf("Ge radie : ");
   scanf("%f", &radie);

   /* beräkna och skriv ut area */
   area = pi*radie*radie;
   printf("Arean = %f", area);
}
```

Vid körning av detta program kommer utskriften att bli:

Ge radie : 2
Arean = 12.566360

OBS! C-kompilatorer skiljer mellan *små* och *stora* bokstäver. Exempelvis betyder ej *Area* samma sak som *area*. De flesta kompilatorer kräver dessutom att fördefinierade ord (const, float ..) ska skrivas med små bokstäver. *Använd alltså små bokstäver!*

```
/* Cirkel.c v1.0 */
```

Är en kommentar som kan innehålla vilken text som helst. Kompilatorn bryr sig ej om innehållet.

OBS! Man bör skriva programmets *filnamn* i en kommentar eftersom detta ej behöver skrivas ut i C-programmet. Alla huvudprogram heter main.

```
#include <stdio.h>
```

Är ett direktiv till en förkompilator. Detta direktiv innebär att förkompilatorn hämtar standardinkluderingsfilen stdio.h, som måste inkluderas för att vi ska kunna använda in/ut-rutiner som scanf och printf på ett korrekt sätt.

OBS! Vid inkludering av egna filer skriver man istället

```
#include "myfil.h"
```

```
void main()
```

Varje körbar modul måste innehålla ett huvudprogram vars start markeras med main. Vid körning startar alltid programmet vid denna punkt. Egentligen är main en funktion som ej returnerar något värde, vilket markeras av void och saknar parametrar, vilket framgår av de tomma parenteserna efter main().

```
{
   ...;
   ...;
}
```

Block-parenteserna eller måsvingarna { och } markerar blockstart resp blockslut och visar i detta fall var huvudprogrammet main börjar och var det slutar. Programmet innehåller ett antal satser. Varje sats avslutas med ett semikolon (;).

```
const float pi = 3.14159;
```

Ett minnesutrymme ges namnet pi och tilldelas det reella värdet 3.14159. Minnesutrymmets värde får ej ändras vilket anges med const.

OBS! Vi kunde ha deklarerat konstanten pi som ett makro istället enligt:

```
#define PI 3.14159
```

Detta makro expanderas sedan av förkompilatorn till det angivna värdet på alla ställen i programmet där PI står. Stora bokstäver brukar användas till makron.

```
float radie, area;
```

Två minnesutrymmen (variabler) definieras med namnen radie resp area. Ordet float markerar att variablerna kan tilldelas reella värden.

```
printf("Ge radie: ");
```

Utskriftsrutinen printf anropas och den skriver ut orden som finns mellan citationstecknen ("). Till funktionen printf skickar man information med hjälp av en parameter eller ett argument i form av strängen "Ge radie :".

```
scanf( "%f", &radie);
```

Inmatningsrutinen scanf anropas och den läser värdet som matas in från tangentbordet och tilldelar radie detta värde. Rutinen scanf har alltid en formatsträng som första parameter. Där ska man med en speciell kod ange hur inbufferten ska tolkas. Formatet %f innebär att man ska hämta ett flyttal (reellt tal) från tangentbordet. Den andra parametern &radie anger att flyttalet ska placeras i minnet i den adress som radie har. Ampersand (&) markerar adressen för en variabel.

```
area = pi * radie * radie;
```

Datorn räknar ut ett värde för produkten av talen i minnesutrymmena pi, radie och radie och tilldelar variabeln area (minnesutrymmet som har namnet area) detta värde med hjälp av tlldelningstecknet (=).

OBS! *Tilldelningstecknet är = och likhetstecknet är == i C.*

```
printf("Arean = %f", area);
```

Utskriftsrutinen printf anropas och den skriver först ut strängen "Arean = " följt av värdet av den reella variabeln area. Var och hur värdet av area skrivs ut markeras av %f som även kan innehålla information om antal positioner och antal decimaler ex %5.2f.

Ex: Modifiera programmet cirkel så att man kan mata in radien och få arean beräknad för ett godtyckligt antal cirklar. Matar vi in radien 0 ska programmet avslutas.

```
/* Cirklar.c v1.0          */
/* Gunnar Joki GJI         */
/* ANSI C                  */
/* Arean av cirklar        */

#include <stdio.h> /* printf, scanf */

void main()
{
   const float pi = 3.14159;
   float radie, area;


   /* läs in första värde till radie */
   printf("Ge radie (avsluta med 0) : ");
   scanf("%f", &radie);

   /* beräkna area och läs in nytt värde på radie */
   while ( radie != 0 )
   {
      /* beräkna och skriv ut area */
      area = pi*radie*radie;
      printf("Arean = %f\n", area);

      /* läs in nytt värde till radie */
      printf("Ge radie (avsluta med 0) : ");
      scanf("%f", &radie);
   }
}
```

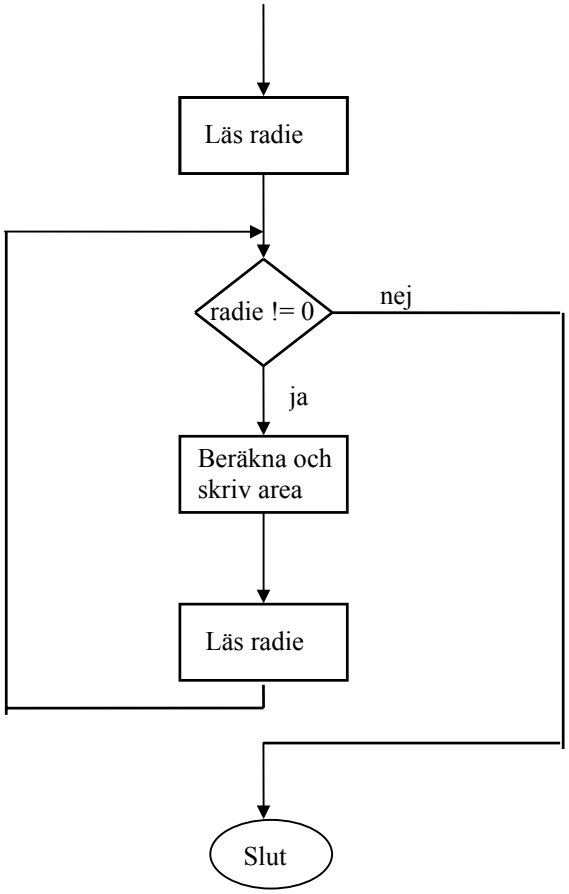
En körning av programmet kan se ut som:

Ge radie (avsluta med 0) : 1
Arean = 3.141590
Ge radie (avsluta med 0) : 2
Arean = 12.565360
Ge radie (avsluta med 0) : 0

```
while (  radie != 0 )
{
   ...
}
```

Så länge radie är skilt ifrån 0 upprepas alla satser i blocket mellan { och }.

Man kan rita ett flödesschema som visar programflödet, alltså i vilken ordning satserna kring while-loopen körs i ovanstående program enligt:



För att flytta fram till nästa rad på skärmen måste man skicka ett RETURN-tecken till den. Detta gör man själv vid buffrade inmatningar, som alltid avslutas med att man trycker RETURN. Vid utmatning måsta man skriva ut RETURN-tecknet (\n) i printf-satsen enligt:

```
printf("Arean = %f\n", area);
```

Ex: Skriv ett program som skriver ut hur många rabattfrimärken som man ska använda sig av för olika tunga brev enligt tabellen:

Max vikt(g): 100 250 500 1000
Märken(st) : 1 3 4 5

Är vikten större än 1000 g ska programmet skriva ut att brevet ska skickas som paket.

```
/* Porto.c v1.0                              */
/* Gunnar Joki GJI                           */
/* ANSI C                                    */
/* Rabattfrimärken för olika tunga brev      */

#include <stdio.h>

void main()
{
   int vikt;

   /* läs in vikt */
   printf(" Ge vikt : ");
   scanf("%d", &vikt);

   /* välj rätt antal rabattfrimärken */
   if (vikt <= 100)
      printf(" Antal rabattfrimärken = 1\n");
   else if (vikt <= 250)
      printf(" Antal rabattfrimärken = 3\n");
   else if (vikt <= 500)
      printf(" Antal rabattfrimärken = 4\n");
   else if (vikt <= 1000)
      printf(" Antal rabattfrimärken = 5\n");
   else
      printf(" Skickas som paket!\n");
}
```

En körning av programmet kan se ut som:

Ge vikt : 200
Antal rabattfrimärken = 3

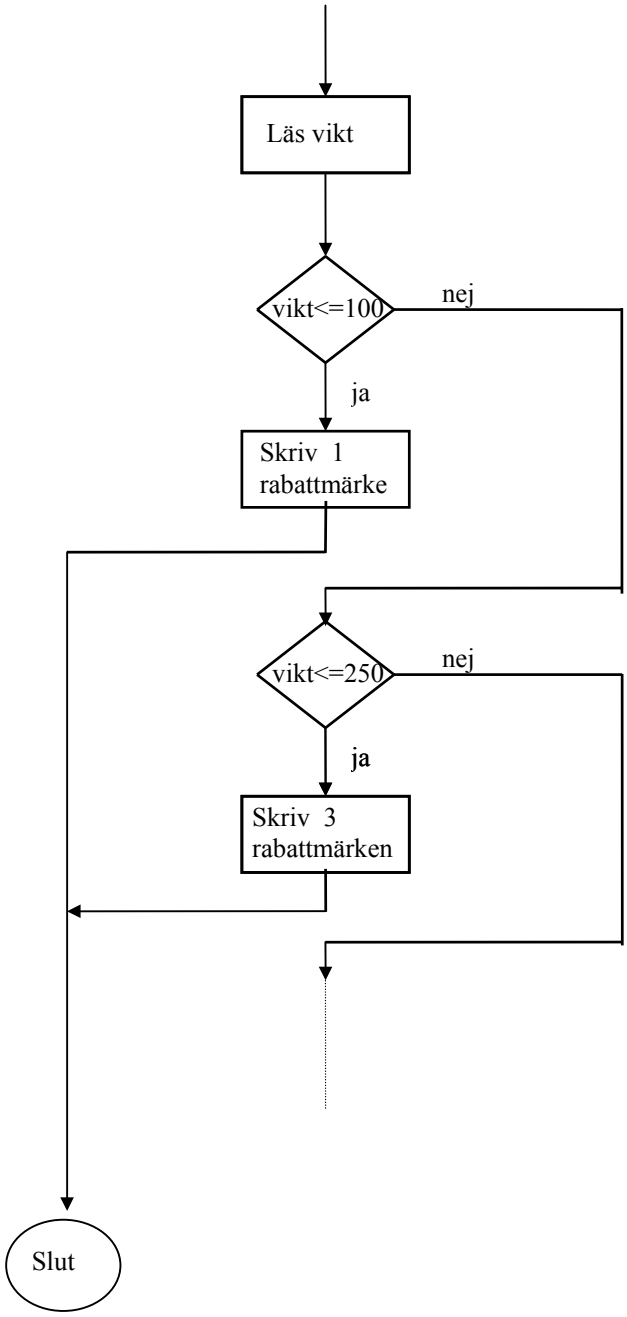
En annan körning av programmet kan se ut som:

Ge vikt : 600
Antal rabattfrimärken = 5

```
if ( vikt <= 100 )
   ...
else if ( vikt <= 250)
   ...
```

Här prövas villkoren för alternativen från början och det alternativ vars villkor först blir korrekt väljs och utförs. Endast ett alternativ utförs.

Ett flödesschema som visar hur satserna i detta program körs ser ut som:



I printf-satsen anger man formatet `%d` vilket innebär att vanliga heltal (basen 10) kan inmatas. I föregående program hade man %f när reella tal skulle inmatas.

Ex: Skriv ett program som skriver ut en trigonometrisk tabell för vinklar mellan 0 och 90 enligt:

```
VINKEL      SINUS      COSINUS      TANGENS
------      -----      -------      -------
   0       0.0000       1.0000       0.0000
  10       0.1736       0.9848       0.1763
  ..       ......       ......       ......
  90       1.0000       0.0000       ******
```

```
/* Trigtab.c v1.0                            */
/* Gunnar Joki GJI                           */
/* AANSI C                                   */
/* Trigonometrisk tabell 0 - 90 grader       */

#include <stdio.h>
#include <math.h>    /* sin, cos, tan, atan */

void main()
{
   const float pi = 4*atan(1);
   float vinkel;
   int nr;

   /* skriv tabellrubrik */
   printf(" VINKEL      SINUS      COSINUS      TANGENS\n");
   printf(" ------      -----      -------      -------\n");


   /* beräkna och skriv tabellvärden */
   for ( nr = 0; nr <= 90; nr += 10)
   {
      printf("%5d",nr);
      vinkel = nr * pi/180;
      printf("%13.4f",sin(vinkel));
      printf("%13.4f",cos(vinkel));
      if (nr < 90)    // om nr mindre än eller lika med 90
         printf("%13.4f\n",tan(vinkel));
      else            // annars
         printf("        ******\n");
   }
}
```

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

Efter denna inkludering har man tillgång till de vanliga matematiska funktionerna som exempelvis :

```
printf("%13.4f",sin(vinkel));
printf("%13.4f",cos(vinkel));
```

OBS! Vinkeln måste skickas i radianer för att få tillbaks korrekta värden.

OBS! Formateringen av utskriften med %13.4f, vilket innebär utskrift i 13 positioner högerjusterat med 4 decimaler. Positionerna fylls ut med blanka i början.

```
for ( nr = 0; nr <= 90; nr += 10 )
{
   ...
}
```

Här ges först nr värdet 0. Sedan kontrolleras om villkoret nr <= 90 är sant. Är detta sant körs satserna mellan { och }. Därefter ökas nr med 10 och ny kontroll sker. Är villkoret fortfarande sant körs satserna igen och det hela upprepas så länge villkoret är sant.

Denna for-loop är helt ekvivalent med följande while-loop:

```
nr = 0;
while (nr <= 90)
{
   ...
   nr += 10;
}
```

Satsen

```
nr += 10;
```

är detsamma som

```
nr = nr + 10;
```

dvs öka det gamla nr med 10 och tilldela detta till det nya nr.

2 Enkla datatyper

Att skriva ett program innebär att man ska tillverka en plan för att bearbeta data eller information på något sätt. Programmering handlar om *data och bearbetning av data*. Data kan vara av enkel typ som tal och tecken eller av mer sammansatt typ som text, tabeller, ljud, bilder etc. När det gäller att tillfälligt spara undan data under bearbetningen använder man sig av *variabler*.

2.1 Variabler

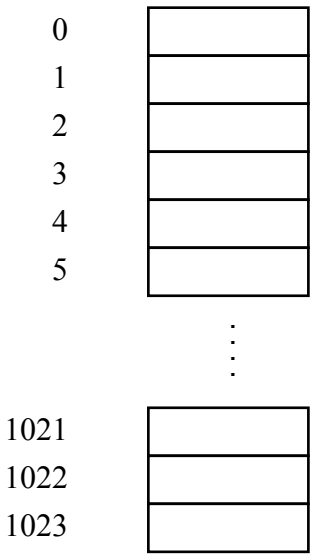
Variabler använder man sig av för att under programkörningen spara data eller information i primärminnet. En variabel har *typ, namn och värde*.

Ex: En variabel av typen int med namnet tal och initialvärdet 2314 skapas enligt:

```
int tal = 2314;
```

Med namnet på variabeln hänvisar man till en speciell plats i primärminnet. Primärminnet kan liknas vid ett stort antal postfack. Storleken på varje fack är 1 Byte eller 8 bitar (oftast). För att kunna stoppa in data i eller läsa data från rätt fack har varje fack en unik adress.

Ex: Ett minne på 1 KB = 1024 B kan ges adresser enligt:

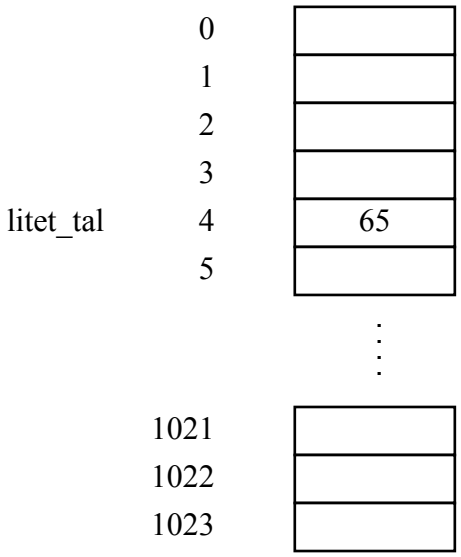


Det är inte lätt att hålla reda på alla fack och deras adresser. Därför har man i högnivåspråken infört variabelbegreppet . Man definierar en variabel med ett unikt namn som kompilatorn automatiskt kopplar ihop med av typen angivet utrymme i minnet. På detta sätt slipper man att hålla reda på exakt var i minnet data finns. Man använder istället variabelnamnet då man vill komma åt data.

Ex: Har man i sitt program definierat variabeln litet_tal av typen char och tilldelat den värdet 65 enligt:

```
char litet_tal;
litet_tal = 65;
```

kan minnesbilden se ut som:



Istället för att vara tvungen att komma ihåg att litet_tal finns på adressen 4 använder man variabelnamnet litet_tal för att komma åt data enligt exempelvis:

```
litet_tal = 34;                 /* nytt värde 34 */
litet_tal = litet_tal + 12;     /* nytt värde blir 46 */
```

Variabelnamnet får innehålla bokstäver, siffror och understrykningstecken. Namnet måste dock börja med en bokstav eller ett understrykningstecken. Antalet tillåtna tecken i variabelnamnen beror på vilken kompilator man använder. Oftast får man dock använda åtminstone 31 tecken.

Ex: x_koordinat
minsta_tal
max_tal_3

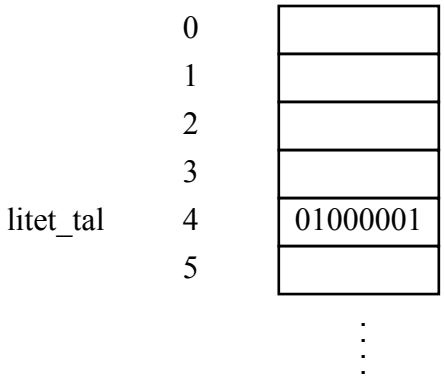
OBS! Man får ej ha de svenska bokstäverna Å, å, Ä, ä, Ö, ö i variabelnamnen.

OBS! Använd variabelnamn som utsäger något. Var inte rädd för att använda långa variabelnamn. En programtext skrivs en gång men läses betydligt fler gånger. Använd understrykningstecknet för att göra långa namn tydligare.

En variabels värde skriver vi i vårt högnivåspråk på det sätt som är brukligt när man skriver tal, tecken, text etc. Kompilatorn måste dock koda vårt data på något sätt till ettor och nollor. Tal kodas i vanlig binärkod där 2 används som bas istället för 10.

Ex: Ange hur de enskilda bitarna ser ut i minnes-facket litet_tal ovan

$$65 = 0 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 01000001$$



Vad kan man då spara för variabelvärden i ett minnesfack bestående av 8 bitar? Detta beror naturligtvis på vilken form av kod som används. Allmänt kan man spara 2*2*2*2*2*2*2*2 = 256 olika kodkombinationer i 8 bitar eftersom varje bit kan anta två värden. Detta innebär att man kan spara 256 olika heltal.

2.1.1 Heltal

Ex: Hur stora positiva heltal kan man spara i 1 Byte om vanlig binärkod används?

0	00000000
1	00000001
2	00000010
3	00000011
.	
.	
255	11111111

Alltså kan man spara tal från 0 till 255.

Vill man även använda negativa tal måste man på något sätt i koden markera att talet är negativt. Man kan exempelvis låta den första biten markera tecknet och då finns det bara plats för 128 olika tal i de resterande 7 bitarna. Den vanligaste koden för negativa tal är dock tvåkomplementmetoden.

Ex: Vilka heltal kan man spara i 1 Byte om kodformen tvåkomplementmetoden används för negativa tal?

0	00000000
1	00000001
.	
.	
127	01111111
-128	10000000
-127	10000001
.	
.	
-1	11111111

Alltså kan man spara tal mellan -128 och 127. Tvåkomplementmetoden utgår från binärkoden för det positiva talet och byter alla ettor mot nollor och nollor mot ettor. Efter addition med 1 har man fått den nya koden.

Vad gör man om man vill använda större tal än vad som ryms i 8 bitar? Man utnyttjar naturligtvis flera minnesfack och får då tillgång till flera bitar och ett större antal kombinationer.

Ex: Hur stora positiva heltal kan man spara i 16 bitar?

8 bitar rymmer talen 0 till $2^8 - 1 = 255$

16 bitar rymmer talen 0 till $2^{16} - 1 = 65535$

Ex: Hur stora heltal kan man spara med tvåkomplementmetoden i 16 bitar?

8 bitar rymmer talen $-2^7 = -128$ till $2^7-1 = 127$

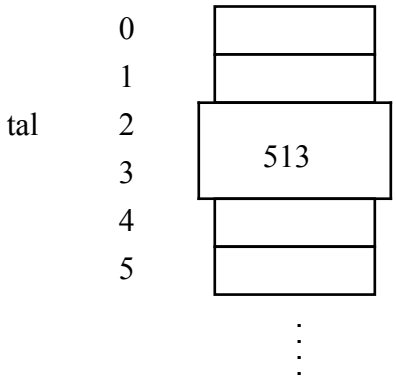
16 bitar rymmer talen $-2^{15} = -32768$ till $2^{15}-1 = 32767$

Det är den angivna datatypen som bestämmer hur många minnesfack och vilken kodning som ska användas för den definierade variabeln. Med typen short anger man exempelvis att man ska använda binärkodade tal i 16 bitar. Antalet bitar för olika datatyper kan variera något mellan olika kompilatorer.

Ex: Variabeldefinitionen

```
short tal = 513;
```

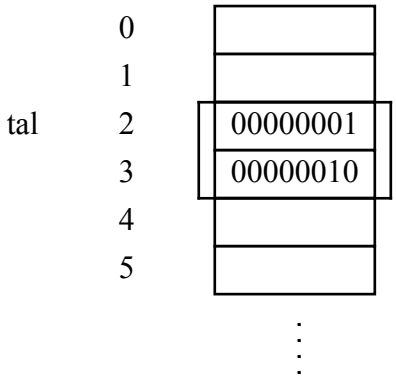
ger minnesbilden:



och eftersom

513 = 0000 0010 0000 0001 = 0 2 0 1 (hexadecimalt)

blir bitmönstret i minnet:



Vilka typer av variabler som finns beror oftast på vilken dialekt av C man har. Exempelvis kan det finnas följande datatyper att tillgå för hela tal:

unsigned char	8 bitar	0 .. 255
char	8	-128 .. 127
unsigned short	16	0 .. 65 535
short	16	-32 768 .. 32 767
unsigned int	32	0 .. 4 294 967 295
int	32	-2 147 483 648 .. 2 147 483 647

2.1.2 Tecken

När det gäller tecken använder man i högnivåspråket de vanliga beteckningarna för bokstäver och tecken omgivna av enkla apostrofer som i 'A', '+', '7'. Den vanligaste koden för tecken är ASCII-koden. Den standardiserade ASCII-koden är på 7-bitar och kan alltså innehålla 128 tecken men de flesta kompilatorer använder 8-bitars ASCII-kod och då kan man hantera 256 tecken.

Ex: Har man i sitt program definierat variabeln tecken av typen char kan man tilldela den bokstaven A enligt:

```
char tecken;
tecken = 'A';
```

Kompilatorn kodar teckenvärdet till ASCII-koden som för tecknet 'A' är 65. Variabeln tecken är av heltalstypen char och kan ses som:

eller

Ex: Skriv ett program som läser in ett tecken och skriver ut tecknet och dess ASCII-kod.

```
#include <stdio.h>

void main()
{
   char tecken;

   /* läs tecken */
   printf("Ge ett tecken : ");
   scanf("%c", &tecken);

   /* skriv tecken och dess ASCII-kod */
   printf("Tecknet %c har ASCII-koden %d\n", tecken, tecken);
}
```

Matar man in bokstaven A blir utskriften:

Tecknet A har ASCII-koden 65

OBS! Med omvandlingspecifikationen %c i utskriftssatsens formatsträng anges att minnesutrymmets ettor och nollor ska tolkas som koden för ett tecken och skivas ut som ett tecken . Omvandlingsspecifikationen %d anger att samma minnesutrymme ska tolkas som koden för ett tal och skivas som decimalt tal.

2.1.3 Reella tal

Alla tal är inte hela tal. När man ska hantera data i form av reella tal i datorn används oftast koder där man skriver om talet som en potens av 2 och sedan i minnet sparar mantissan (talet framför potensen) och exponenten i binärkod.

Ex: Talet 5.5 kan exempelvis skrivas om som:

$$5.5 = 2^2 + 2^0 + 2^{-1} = (1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4}) \cdot 2^3$$

Här sparar man mantissans koefficienter 1011 och exponenten 3 = 11 i binär form. Man ser att ju fler decimaler eller ju större tal desto fler bitar behövs. Även tecknet på talen kräver en bit.

I de flesta miljöer finns följande typer för reella tal med angivna antalet bitar och storleksordningar:

float	32	$\pm 1.2 \cdot 10^{-38}$	..	$\pm 3.4 \cdot 10^{38}$	(7 värdesiff)
double	64	$\pm 2.2 \cdot 10^{-308}$	..	$\pm 1.8 \cdot 10^{308}$	(15 värdesiff)

Ex: Skriv ett program som frågar efter en kropps massa och hastighet och beräknar och skriver ut dess kinetiska energi som är lika med massa*hastighet*hastighet/2.

```
#include <stdio.h>

void main()
{
   float massa, hastighet, kinetisk_energi;

   /* läs massa */
   printf("Massa = ");
   scanf("%f", &massa);

   /* läs hastighet */
   printf("Hastighet = ");
   scanf("%f", &hastighet);

   /* beräkna och skriv ut kinetisk energi */
   kinetisk_energi = massa*hastighet*hastighet/2;
   printf("Kinetisk energi : %f", kinetisk_energi);
}
```

2.1.4 Egenuppräknade

Man kan även definiera egna variabeltyper där man ger namn åt varje enskilt värde.

Ex: Definiera en egenuppräknad variabel som ska kunna anta tillståndsvärdena upp, ner och stilla för en hiss.

```
/* definiera en egen datatyp enum hiss */
enum hiss {upp, ner, stilla};

/* definiera en variabel med namnet a_hiss */
enum hiss a_hiss;

/* tilldela a_hiss värdet stilla */
a_hiss = stilla;
```

För egendefinierade typer används vanlig binärkod där det först uppräknade värdet får koden 0 om inget annat anges.

Ex: Definiera en egen variabeltyp som ska kunna ha värden i form av månadsnamn och där månadsnamnen ska motsvaras av motsvarande månadsnummer. Månaden januari ska alltså ha värdet 1 och december värdet 12.

```
/* definiera en egen datatyp enum manad */
enum manad {januari = 1, februari, mars, april, maj, juni,
juli, augusti, september, oktober, november,
december};

/* variabel mm av typen enum manad och några andra */
enum manad mm;
int yy, dd, datum;

/* tilldela värden */
yy = 2001;
mm = september;
dd = 14;

/* skriv ut datum på formen 010914 */
datum =(yy - 2000)*10000 + mm*100 + dd;
printf("%06d", datum); /* utfyllnad med 0 till 6 positioner */
```

Egenuppräknade värden används för att göra koden lättare att läsa. För den som ska tyda koden då den ska underhållas eller ändras är det lättare att förstå sammanhanget om det står september istället för bara talet 9.

2.2 Konstanter

Variabler ska kunna ändra sina värden under programkörningen genom att man utför vissa operationer på dessa. Man kan exempelvis läsa in ett nytt värde till en variabel eller man kan multiplicera det med 2 etc.

Ibland vill man se till att ett minnesutrymmes värde ej får ändras under programkörning. Man kan då med *const* framför typnamnet kvalificera eller märka detta minnesfack. Det värde som minnesfacket ska ha måste man ge direkt vid definitionen genom initiering.

Ex: Man ska ha ett program med priser och moms.

```
const float moms = 25.0;
const int pris_per_st = 50;
int st;
float pris, pris_med_moms;

printf("Ge antal : ");
scanf("%d", &st);

pris = pris_per_st*st;
pris_med_moms = pris + pris*moms/100;
```

Varför ska man använda ordet moms eftersom man lika gärna kan skriva 25.0 istället. Fördelen med konstanter är att man har dessa samlade i början av programmet. Ändras momsen räcker det med att ändra på ett ställe istället för att leta upp alla ställen där moms-värdet finns i programmet.

Ett annat sätt att ge namn åt konstanter som ofta används speciellt i äldre C-program är att definiera ett makro. Ett makro är ett direktiv till kompilatorn att vid förkompileringen byta ut alla makro-namn mot motsvarande värde.

Ex: Skriv ett program som läser in en radie och beräknar volymen av ett klot med denna radie. Använd ett macro för PI.

```
#include <stdio.h>
#define PI 3.14159

void main()
{
   float radie;

   printf("Ge radie : ");
   scanf("%f", &radie);

   printf("Klotvolym = %f", 4*PI*radie*radie*radie/3);
}
```

PI byts ut mot 3.14159 vid förkompileringen.

2.3 In- och utmatning

Ett progam som ska bearbeta data eller information måste på något sätt få tillgång till denna data via någon kanal som exempelvis tangentbord, sekundärminne, port, ljudkort, video-kamera etc. Processen att hämta in data till ett program kallas för *inmatning*.

Efter bearbetningen måste man också kunna presentera resultatet på någon enhet som exempelvis bildskärm, sekundärminne, printer, port, ljudkort etc. Processen att skicka ut data eller presentera data kallas för *utmatning*.

Vanliga enkla in- och utmatningar till variabler av enkla typer gör man via tangentbordet och skärmen. I C finns i *stdio.h* funktionerna *scanf* och *printf* som sköter in- resp utmatning av enkla variabler. Dessutom finns funktionerna *getchar* och *putchar* för in- resp utmatning av tecken.

2.3.1 Inmatning med scanf-funktionen

Inmatning från tangentbordet går till så att alla tecken man skriver samlas i en buffert, *stdin*, samtidigt som de också ekas på skärmen. När man matat in färdigt trycker man på RETURN och då scannar (läser och omvandlar) programmets scanf-sats bufferten och flyttar över data till angiven plats i minnet.

Ex: Inmatning av ett heltalsvärde till variabeln tal från tangentbordet.

```
#include <stdio.h>

void main()
{
   int tal;

   printf("Ge ett heltal : ");
   scanf("%d", &tal);
}
```

En körning:

Ge ett heltal : 234

Vid körning stannar programmet då den kommer till scanf-satsen och väntar på att den som kör programmet ska skriva in ett heltal och trycka RETURN. När man gjort detta scannar scanf-satsen inmatningen enligt *formatsträngen* "%d", som ovan endast inne--håller en *omvandlingsspecifikation* %d, vilket anger att tecknen i bufferten ska *tolkas som ett decimalt heltal* och talet ska sedan placeras i minnet *kodat som en int* på den adress som tal har.

Man kan läsa in värden till flera olika variabler med samma scanf-sats. Scanf-satsen innehåller då flera variabeladresser och varje variabeladress ska ha en motsvarande omvandlingsspecifikation i formatsträngen

Ex: Inmatning av två heltalsvärden till variablerna a_tal och b_tal i en enda sats.

```
#include <stdio.h>

void main()
{
   int a_tal, b_tal;

   printf("Ge två heltal : ");
   scanf("%d%d", &a_tal, &b_tal);
}
```

En körning:

Ge två hela tal : 345 268

Här måste man *separera* de inmatade talen med någon *tillåten separator* annars tolkar datorn de inmatade tecknen som ett enda tal och stannar och väntar på nästa tal. Som separatorer kan man använda blanktecken (som ovan), RETURN (styrtecknet '\n') eller TAB (styrtecknet '\t'). Programmet gör så att den börjar med att hoppa över alla eventuella separatorer som den hittar i början. Sedan läser den fram till nästa separator och placerar det lästa talet i den första variabeln. Proceduren upprepas sedan för nästa variabel o.s.v.

Inmatningsbufferten läses utgående ifrån formatsträngen. Har man exempelvis angivit %d innebär detta att programmet läser tecken så länge de passar in i ett decimalt heltal. Skulle den stöta på ett tecken som ej passar in, exempelvis bokstaven A, avbryts inläsningen och det dittills godkända tecknen omvandlas till heltal och placeras i variabeln. Hittas inga godkända tecken avbryts inläsningen direkt utan att något lästs in.

Ex: Några körningar av programmet ovan med felaktiga inmatningar.

a) Ge två heltal : 12S 321

Detta ger a_tal = 12 och b_tal har kvar sitt gamla odefinierade värde.

b) Ge två heltal : 123 3F21

Detta ger a_tal = 123 och b_tal = 3.

OBS! De tecken som ej accepteras blir kvar i inmatningsbufferten och kan ställa till besvär vid en eventuell ny inläsning. Man bör därför alltid se till att man har en tom inmatningsbuffert efter varje inläsning. I vissa lägen bör man även tömma bufferten på eventuella kvarvarande separatortecken.

Man kan med scanf-funktionen även läsa in reella tal och tecken. Dock kan man inte läsa in egenuppräknade variabler.

Ex: Skriv ett program som läser in ett additonsuttryck bestående av två reella tal enligt exempelvis 4.0+3.2 och beräknar och skriver ut resultatet.

```
#include <stdio.h>

void main()
{
   float a_tal, b_tal, sum;
   char op;

   /* läs in uttrycket */
   printf("Ge uttrycket på formen x+y : ");
   scanf("%f%c%f", &a_tal, &op, &b_tal);

   /* skriv resultatet */
   if ( op == '+' )
   {
      sum = a_tal + b_tal;
      printf("Resultat = %.1f\n", sum);
   }
   else
      printf("Fel operator!\n");
}
```

En körning ger följande :

Ge ett uttryck : 3.2+4.0
Resultat = 7.2

Formatsträngen "%f%c%f" i scanf-funktionen läser först fram till +-tecknet eftersom 3.2 kan omvandlas till ett reellt tal. Därefter läses ett tecken nämligen +-operatorn och slutligen det andra reella talet.

OBS! Omvandlingsspecifikationen %c hoppar ej över några separatorer utan läser varje tecken i inmatningsbufferten. Hade man exempelvis skrivit ett blanktecken före +-tecknet skulle op blivit ett blanktecken och programmet skriver "Fel operator!". Vill man hoppa över eventuella blanktecken får man ange detta i formatsträngen med "%f %c %f"och då fungerar en inmatning av typen :

Ge ett uttryck : 3.2 + 4.0
Resultat = 7.2

2.3.2 Utmatning med printf-funktionen

Utmatning till skärmen sker med printf-funktionen som fyller bufferten *stdout*, som sedan skickas ut till skärmen på den plats där cursorn står. Hur bufferten ska fyllas ska framgå av formatsträngen i printf-funktionen.

Ex: Inmatning av ett heltalsvärde till variabeln tal från tangentbordet och utmatning av det dubbla värdet.

```
#include <stdio.h>

void main()
{
   int tal, dubbelt_upp;

   printf("Ge ett heltal : ");
   scanf("%d", &tal);

   dubbelt_upp = 2 * tal;
   printf("Tal = %d\nDubbelt upp = %d\n", tal, dubbelt_upp);
}
```

En körning:

Ge ett heltal : 35
Tal = 35
Dubbelt upp = 70

Formatsträngen kan innehålla text, styrtecken och omvandlingsspecifikationer. Varje variabel ska ha en omvandlingsspecifikation som anger hur minnesutrymmet ska tolkas och skrivas ut. *Omvandlingsspecifikationen %d anger att minnesutrymmets kod ska tolkas som en int och skrivas ut decimalt i default format*. Texten i övrigt skrivs ut som den står förutom styrtecknen som skärmen reagerar för på ett speciellt sätt. Exempelvis flyttar styrtecknet '\n' skärmens cursor till början på nästa rad.

Formatsträngens omvandlingsspecifikation kan även innehålla en formatering av utskriften där man anger antal positioner som utskriften ska skrivas ut på.

Ex: Vill man ha en utskrift vid körning ovan enligt:

Ge ett heltal : 35
Tal = 35
Dubbelt upp = 70

får man skriva formatsträngen så att utskriften sker högerjusterad i 12 positioner enligt:

```
printf("Tal = %12d\nDubbelt_upp = %d\n", tal, dubbelt_upp);
```

Utskriftsatsen kan innehålla mer sammansatta uttryck än enstaka variabler. Vill man bara skriva ut ett resultat av ett uttryck är det onödigt att skapa en extra variabel för detta utan man kan låta printf-funktionen både räkna ut uttryckets värde och skriva ut detta.

Ex: Skriv ett program som läser in årsräntesats i % och kapital i kr och beräknar och skriver ut årsräntan i kr.

```
#include <stdio.h>

void main()
{
   float procent, kapital;

   printf("Årsränta i %: ");
   scanf("%f", &procent);
   printf("Kapital i kr : ");
   scanf("%f", &kapital);

   printf("Årsräntan blir %.2f kr\n", kapital*procent/100);
}
```

En körning:

Årsränta i % : 7.5
Kapital i kr : 1250
Årsräntan blir 93.75 kr

Istället för att använda en variabel för räntan beräknar man och skriver ut uttrycket med printf-funktionen. Formatsträngens omvandlingsspecifikation %.2f anger att minnesutrymmet, som här ej är namngivet med en variabel men innehåller det uträknade värdet, ska tolkas som ett reellt tal och skrivas ut med 2 decimaler.

Ex: Skriv ett program som läser in en liten bokstav a .. z och skriver ut motsvarande stora bokstav. Observera att i ASCII-tabellen är koden för en liten bokstav 32 enheter större än för motsvarande stora.

```
#include <stdio.h>

void main()
{
   char tecken;

   printf("Ge en liten bokstav : ");
   scanf("%c", &tecken);
   printf("Motsvarande stora bokstav är %c\n", tecken - 32);
}
```

Körning:

Ge en liten bokstav : b
Motsvarande stora bokstav är B

2.3.3 In- och utmatning av tecken med getchar- resp putchar-funktionen.

För att läsa in och skriva ut tecken kan man använda funktionerna scanf resp printf med omvandlingsspecifikationen %c. Det finns också speciella funktioner för just in- och utmatning av tecken. Dessa finns i *stdio.h* och heter *getchar* och *putchar*.

Funktionen getchar läser nästa tecken i inmatningsbufferten och returnerar tecknet som funktionsvärde, *alltså själva funktionsnamnet motsvarar tecknet*. Funktionen putchar skriver ut det tecken som står mellan parenteserna, parametern eller argumentet, på skärmen.

Ex: Skriv om ovanstående teckenomvandling från små till stora bokstäver och använd getchar och putchar istället för scanf och printf.

```
#include <stdio.h>

void main()
{
   char tecken;

   printf("Ge en liten bokstav : ");
   tecken = getchar();

   printf("Motsvarande stora bokstav är ")
   putchar(tecken - 32);
   putchar('\n');
}
```

Fördelen med att använda getchar och putchar är att man slipper formatsträngen med dess omvandlingsspecifikationer. Getchar tolkar alltid inmatningsbufferten som ett tecken och putchar skriver alltid ut den angivna parametern som ett tecken på skärmen.

OBS! De tomma parenteserna när en funktion anropas utan parametrar som i getchar().

Ex: Ibland måste man se till att inmatningsbufferten är tom för att nästa läsning ska fungera. Det som oftast finns kvar i bufferten är ett RETURN-tecken och då räcker det med att man anropar funktionen getchar en gång enligt:

```
getchar();
```

Har man även annat skräp före RETURN-tecknet måste man upprepat anropa getchar-funktionen enligt:

```
while ( getchar() != '\n' );
```

Här rensas alla tecknen från inmatningsbufferten fram till och med RETURN-tecknet.

2.4 Uttryck

Ett uttryck består av operander och operatorer. Det enklaste uttrycket består av ett enda värde eller en variabel.

Ex: litet_tal + 23 är ett uttryck bestående av operanderna litet_tal och 23 och operatorn +.

Operander kan i sig vara uttryck.

Ex: x * (y - 3.2) är ett uttryck bestående av operanderna x och (y-3.2).

Alla utryck har ett värde av en viss typ av data. Vilken typ av värde ett uttryck får beror av de ingående operatorernas värden och typer. Man brukar skilja på *aritmetiska uttryck* som har värden i form av typen heltal eller reella tal och *logiska uttryck* som bara har två värden falskt eller sant av typen heltal. I C är egentligen även logiska uttryck aritmetiska eftersom *falskt motsvaras av heltalet 0* och *sant av 1*.

Ex: pi*radie*radie är aritmetiskt med ett värde i form av ett reellt tal

tal < 0 är logiskt med ett värde 0 (falskt) eller 1 (sant)

2.4.1 Aritmetiska uttryck

Aritmetiska uttryck innehåller de vanliga operatorerna +, -, * och /. Dessutom finns en del speciella operatorer som % (resten vid division), ++ (öka med 1) etc.

Ex:	5 + 3	har värdet 8 av typen heltal
	5 / 2	har värdet 2 av typen heltal (kvoten vid heltalsdivision)
	(float)5 / 2	har värdet 2.5 av typen reellt tal (explicit typomvandling)
	5 % 2	har värdet 1 av typen heltal (resten vid heltalsdivision)
	5 + 3 * 2	har värdet 11 av typen heltal
	(5 + 3) * 2	har värdet 16 av typen heltal

OBS! När man blandar olika typer av operander i uttryck omvandlas först alla operander till den högsta typen varefter beräkning sker. Med högsta typ menas den som har störst minnesutrymme. Blandar man som i uttrycket (float)5 / 2 ett heltal med ett reellt tal omvandlas heltalet automatiskt till ett reellt tal varefter division utförs för reella tal.

OBS! Samma operator kan ha olika innebörd beroende på vilka datatyper som den ska verka på. Divisionsoperatorn / utför heltalsdivision (hur många hela gånger

går den högra operanden i den vänstra) om den verkar på två heltal och vanlig reell division om den verkar på reella tal.

OBS! De vanliga matematiska prioriteringsreglerna gäller där /, * och % har högre prioritet eller utförs före + och -. Med parenteser ändras prioriteten så att parentesen utförs först. *Använd parenteser för att få tydligare kod!*

Efter beräkning av ett värde på ett uttryck sker ofta en tilldelning av detta värde till en variabel. Som tilldelningsoperator används likhetstecken.

Ex:
```
int a_tal, b_tal;

a_tal = 6 * 2 % 5;    /* a_tal får värdet 2 */
b_tal = 2.3 * 2;/* b_tal får värdet 4 */
a_tal =  b_tal  = 0;  /* b_tal och a_tal får båda värdet 0 */
```

OBS! Först beräknas det högra uttyckets värde och typ. Är det högra uttryckets typ samma som typen hos den vänstra variabeln sker tilldelning direkt. Är däremot typerna olika sker en *automatisk (implicit) typomvandling* av det högra uttryckets värde till den vänstra variabelns typ.

OBS! En tilldelning kan ses som ett uttryck med ett värde som är lika med det tilldelade värdet.

Tilldelningsoperatorn finns i att antal sammansatta former som +=, *= etc.

Ex:
```
float x = 2.3;
int a = 3;

x = x + 2  /* x blir 4.3 */
x += 2;    /* samma operation, x blir 6.3 */

a = a * 2; /* a blir 6 */
a *= 2;    /* samma operation, a blir 12 */
```

OBS! Istället för en tilldelningssats kan man initiera variabelvärden vid definitionen. Ger man inga värden vid definitionen är variabelvärdet odefinierat.

Ofta när man skriver programkod så ingår satser där man ökar eller minskar en variabels värde med ett. I C har man infört speciella operatorer för detta.

Ex: Skriv ett program som skriver ut alla stora bokstäver mellan A och Z.

```
#include <stdio.h>

void main()
{
   char ch = 'A';

   do
   {
      putchar(ch);
      ch++;           /* nästa bokstav */
   }
   while ( ch <= 'Z' );
}
```

Körning ger:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

När man arbetar med aritmetiska uttryck har man ofta användning av de vanliga matematiska funktionerna sin, cos, log, exp etc. Dessa finns i *math.h*.

Ex: Skriv ett program som frågar efter ett reellt tal och skriver ut kvadratroten ur talet och kvadraten på talet.

```
#include <stdio.h>
#include <math.h>     /* sqrt, pow */

void main()
{
   float tal;

   printf("Ge ett tal : ");
   scanf("%f", &tal);

   printf("Kvadratroten ur talet = %f\n", sqrt(tal));
   printf("Kvadraten på talet = %f\n", pow(tal, 2));
}
```

Exempel på körning:

Ge tal : 5.3
Kvadratroten ur talet = 2.302173
Kvadraten på talet = 28.090000

Ex: och Skriv ett program som frågar efter en rätvinklig triangels hypotenusa och en vinkel beräknar triangelns övriga sidor.

```
#include <stdio.h>
#include <math.h>     /* sin, cos, atan */

void main()
{
   float hypotenusa, vinkel;
   const float pi = 4*atan(1);

   printf("Ge hypotenusan och vinkeln : ");
   scanf("%f%f", &hypotenusa, &vinkel);

   printf("Ena sidan = %f\n",hypotenusa*sin(pi*vinkel/180));
   printf("Andra sidan = %f\n",hypotenusa*cos(pi*vinkel/180));
}
```

Körning:

Ge hypotenusan och vinkeln : 35.6 30
Ena sidan = 17.800000
Andra sidan = 30.830504

OBS! Vinkeln måste skickas i radianer vid anrop av de trigonometriska funktionerna.

2.4.2 Logiska uttryck

Ett logiskt uttryck innehåller alltid någon av *relationsoperatorerna* < (mindre än), > (större än), == (lika med), != (skilt ifrån), <=(mindre än eller lika med) och >= (större än eller lika med). Värdet av ett logiskt uttryck blir alltid 0 (falskt) eller 1 (sant).

Ex:	3 < 5	har värdet 1 och typen heltal
	5 == 2	har värdet 0 och typen heltal
	5 != 2	har värdet 1 och typen heltal
	3 >= 6	har värdet 0 och typen heltal
	3 + 4 <= 2 + 6	har värdet 1 och typen heltal

OBS! Relationsoperatorerna har lägre prioritet än de vanliga aritmetiska operatorerna. Man behöver ej prioritera med parenteser i 3 + 4 <= 2 + 6. För tydlighetens skull kan man dock göra detta med (3 + 4) <= (2 + 6).

Flera logiska uttryck kan sammanfogas med de *logiska operatorerna* && (och), || (eller), ! (inte).

Ex:	3 < 5 && 5 == 2	har värdet 0 eftersom det krävs att båda uttrycken är sanna
	3 < 5 || 5 == 2	har värdet 1 eftersom det räcker om ett uttryck sant
	! (3 < 5)	har värdet 0 eftersom inte sant är falskt

OBS! De logiska operatorerna && och || har lägre prioritet än relationsoperatorerna medan ! har högre. Är man osäker på prioriteten ska man använda parenteser. Det gör inget om man har några onödiga parenteser det ökar bara tydligheten.

Logiska uttryck förekommer ofta i villkoren för upprepningar (while) och val (if) i programkoden.

Ex: Skriv den del av ett program som inte ger sig förrän ett korrekt månadsnummer mellan 1 och 12 har inlästs.

```
....
printf("Ge ett månadsnummer mellan 1-12 : ");
scanf("%d", &mm);
while ( mm < 1 || mm > 12)
{
   printf("Fel månadsnummer! Vi gör ett nytt försök!\n");
   printf("Ge ett månadsnummer mellan 1-12 : ");
   scanf("%d", &mm);
}
....
```

I *ctype.h* finns speciella funktioner för att hantera tecken. Många av dessa funktioner är av typen att man ska kontrollera om ett tecken exempelvis är skrivbart, siffra etc. Dessa funktioner returnerar ett logiskt värde sant eller falskt och kan alltså sägas vara logiska uttryck.

Ex: Skriv ett program som läser in ett antal tecken som avslutas med RETURN-tecknet och som skriver ut alla tecken som är siffror.

```
#include <stdio.h>
#include <ctype.h>    /* isdigit */

void main()
{
   char ch;

   printf("Skriv en rad med tecken : ");
   ch = getchar();
   while ( ch != '\n' )
   {
      if ( isdigit(ch) )
         putchar(ch);
      ch = getchar();
   }
}
```

Körning:

Skriv en rad med tecken : Abjhacjj1332dasdd2114dd212
13322114212

Man ser ofta i C-program att man skriver ovanstående kod på ett förkortat sätt genom att utnyttja att *tilldelningar får värdet av det tilldelade uttrycket*. Ovanstående kod kan istället skrivas som:

```
#include <stdio.h>
#include <ctype.h>    /* isdigit */

void main()
{
   char ch;

   printf("Skriv en rad med tecken : ");
   while ( (ch = getchar()) != '\n' )
   {
      if ( isdigit(ch) )
         putchar(ch);
   }
}
```

OBS! Vill man skriva flera rader av tecken får man istället avsluta inskrivningen med en filslutsmarkering (CTRL/Z i DOS) och i programmet testa på EOF istället för RETURN-tecknet '\n'.

3 Styrning av programflöde

Ett program består av ett antal satser. När programmet körs exekveras satserna i den ordning som de står.

Ex:
```
{
  sats1;
  sats2;
  sats3;
}
```

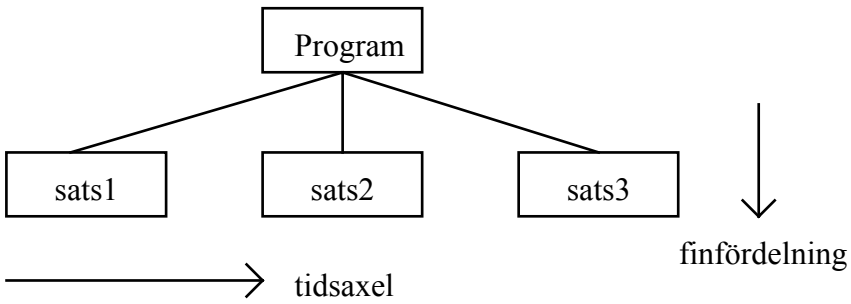
Här exekveras satserna enligt först sats1 sedan sats2 och sist sats3.

Ofta måste man i programmet kunna förändra ovanstående programflöde för att få en korrekt och önskad bearbetning av data. Man ska kanske bara utföra sats2, om något villkor är sant eller man vill upprepa sats3 fem gånger.

I ett programspråk måste det finnas möjligheter att styra det normala eller sekventiella programflödet genom att välja (selektera) eller upprepa (iterera) vissa satser. De styrmekanismer som behövs för att bryta *sekvenserna* är *selektioner* och *iterationer*. Med hjälp av dessa styrmöjligheter kan man lösa alla bearbetningar av data som kan behövas.

När man konstruerar sina program eller databearbetningar är det ofta lämpligt att först tänka ut en plan hur bearbetningen ska göras. Istället för att då använda ett programmeringsspråk kan man rita ett *strukturdiagram* eller bara skriva sin bearbetning på ren svenska i *halvkod*.

Ex: Ett strukturdiagram som visar ovanstående sekvens.



Ex: Halvkod för ovanstående sekvens.

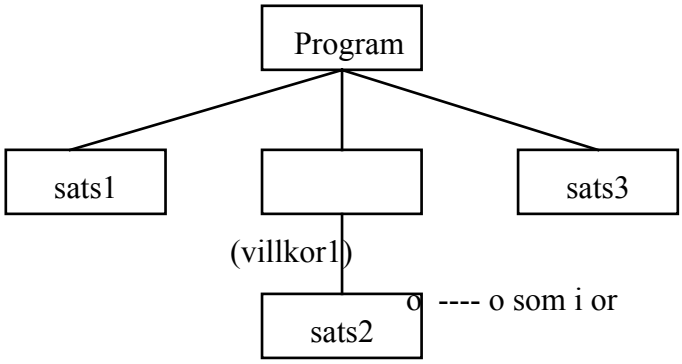
```
Program
  sats1
  sats2
  sats3
```

3.1 Selektion

Med styrmekanismen *selektion* kan man välja att utföra satser *under förutsättning att vissa villkor är uppfyllda*. Man *väljer ut* eller *selekterar* vilka satser som ska utföras.

Ex: I ovanstående exempel ska sats2 endast utföras om villkor1 är sant.

Strukturdiagram:



Halvkod:

```
Program
  sats1
  om villkor1
    sats2
  sats3
```

C-kod:

```
sats1;
if (villkor1)
   sats2;
sats3;
```

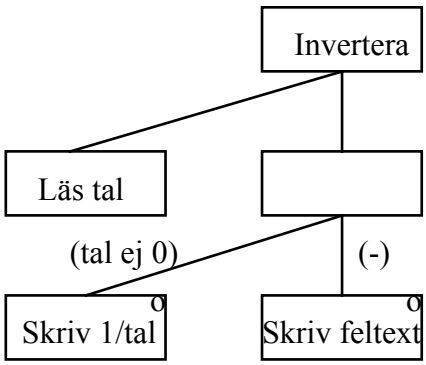
Villkor1 är ett logiskt uttryck som kan anta värdet falskt (0) eller sant (1). Då programmet kommer till if-satsen kontrollerar det värdet på villkor1. Är detta sant utförs sats2 annars inte.

OBS! I strukturdiagrammet ska *varje ruta ha en entydig struktur*. Den ska antingen vara sekvens, selektion eller iteration.

OBS! *Indragningen (indenteringen)* i halvkoden och C-koden för att *markera att man styrt programflödet på ett speciellt sätt.*

Ex: Skriv ett program som läser in ett reellt tal och beräknar och skriver ut dess inverterade värde. Är det inlästa talet 0 ska feltexten 'Saknar inverterat värde!' skrivas ut.

Stukturdiagram:



Halvkod:

```
Invertera
  läs tal
  om tal ej 0
    skriv 1/tal
  annars
    skriv feltext
```

C-kod:

```
#include <stdio.h>

void main()
{
   float tal;

   printf("Ge tal : ");
   scanf("%f", &tal);

   if ( tal != 0 )
      printf ("Inverterat tal = %f\n", 1/tal);
   else
      printf("Saknar inverterat värde!\n");
}
```

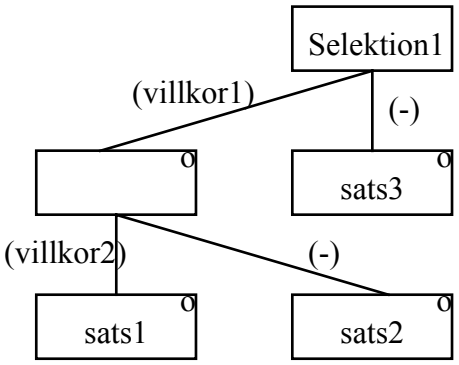
OBS! Annars markeras i strukturdiagrammet med ett streck (-).

OBS! Man kan om man vill skriva text i den tomma burken. I ovanstående exempel kan man exempelvis skriva bearbeta tal. Denna text kan sedan återkomma som rubrik i koden.

Selektioner kan nästlas d.v.s man kan ha selektioner inuti selektioner. Vid nästlade selektioner kan det uppstå problem med och veta till vilken del som annars-delen hör. Regeln är att annars-delen alltid hör till den *närmast föregående selektionen som saknar* annars-del.

Ex: En första nästlad selektion.

Strukturdiagram:

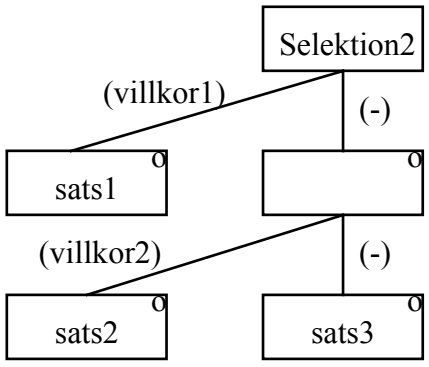


C-kod:

```
if (villkor1)
   if (villkor2)
      sats1;
   else
      sats2;
else
   sats3;
```

Ex: En andra nästlad selektion.

Strukturdiagram:



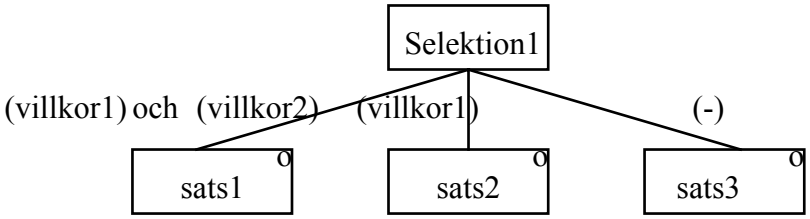
C-kod:

```
if (villkor1)
   sats1;
else
   if (villkor2)
      sats2;
   else
      sats3;
```

Nästlade selektioner har ofta en tendens att bli krångliga. Ser man att en selektion börjar bli väldigt djup i strukturdiagrammet bör man tänka om. Man ska då göra om selektions-villkoren så att en *flervalsselektion* fås med bredd istället för djup. I en flervalsselektion kan man välja mellan fler saker än två och endast en av sakerna utförs.

Ex: Skriv om den första selektionen ovan som en flervalsselektion.

Strukturdiagram:

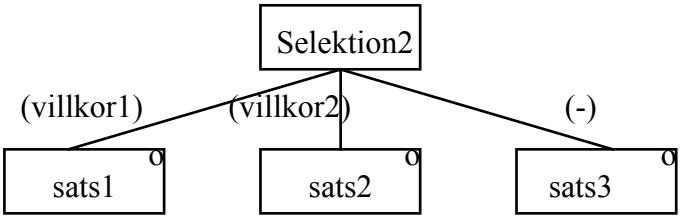


C-kod:

```
if (villkor1 && villkor2)
   sats1;
else if (villkor1)
   sats2;
else
   sats3;
```

Ex: Den andra selektionen skriven som en flervalsselektion.

Strukturdiagram:



C-kod:

```
if (villkor1)
   sats1;
else if (villkor2)
   sats2;
else
   sats3;
```

Ex: Skriv ett program som löser andragradsekvationen

$$x^2 + ax + b = 0$$

som har lösningarna

$$x = -a/2 \pm \sqrt{a^2/4 - b}$$

Halvkod:

```
Andragrad
  läs a, b
  c = a^2 / 4 - b
  om c > 0
    skriv x1
    skriv x2
  annars om c = = 0
    skriv x
  annars
    skriv 'Saknar reell lösning'
```

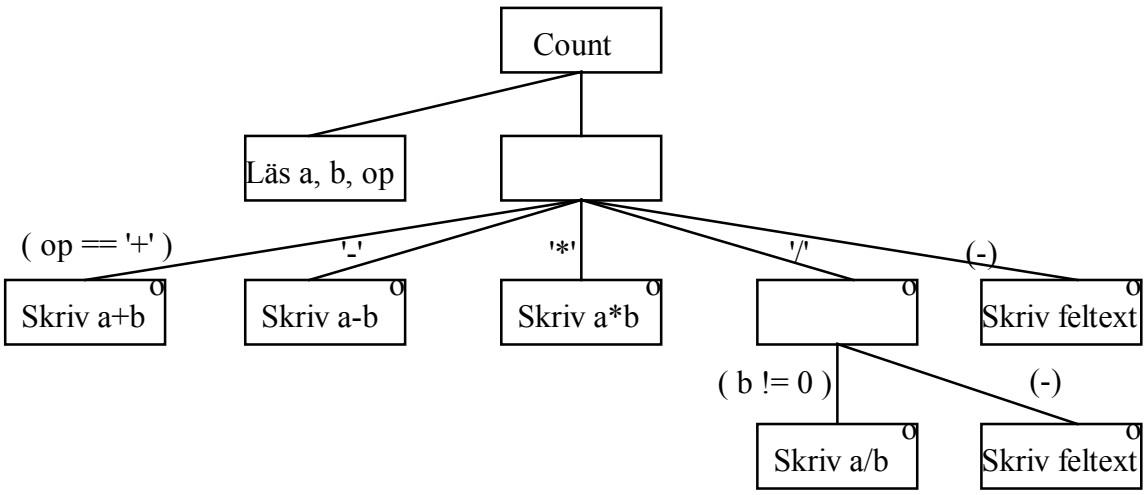
C_kod:

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

void main()
{
   float a, b, c;

   printf("Ge a och b : ");
   scanf("%f%f", &a, &b);

   c = pow(a, 2)/4 - b;

   if ( c > 0 )
   {                                              /* OBS! */
      printf("x1 = %f\n", -a/2 - sqrt(c);
      printf("x2 = %f\n", -a/2 + sqrt(c);
   }                                              /* OBS! */
   else if ( c == 0 )
      printf("x1 = x2 = %f\n", -a/2);
   else
      printf("Ekvationen saknar reell lösning!\n");
}
```

OBS! Ska flera satser utföras inuti en selektionsdel måste *blockparenteserna* { } användas.

Ex: Skriv ett program som läser in ett uttryck av typen 2.3 op 3.4 där op kan vara +, -, * eller /. Programmet ska sedan skriva ut resultatet av operationen. Matas en felaktig operator in ska ett felmeddelande skrivas ut. Ett felmeddelande ska också skrivas ut om man försöker dividera med 0.

Strukturdiagram:



C_kod:

```
#include <stdio.h>

void main()
{
   float a, b;
   char op;

   printf("Ge ett uttryck på formen a op b : ");
   scanf("%f %c %f", &a, &op, &b);

   if (op == '+')
      printf("%f", a + b);
   else if (op == '-')
      printf("%f", a - b);
   else if (op == '*')
      printf("%f", a * b);
   else if (op == '/')
      if ( b != 0)
         printf("%f", a / b);
      else
         printf("Division med 0!");
   else
      printf ("Fel operator!");
}
```

Flervalsselektioner kan i vissa fall kodas med en *switch-sats* istället för en if-sats i C.

Ex: Skriv om programmet count ovan med en switch-sats istället för en if-sats.

```
#include <stdio.h>

void main()
{
   float a, b;
   char op;

   printf("Ge ett uttryck på formen a op b : ");
   scanf("%f %c %f", &a, &op, &b);

   switch (op)
   {
      case '+':
         printf("%f", a + b);
         break;                          /* OBS! */
      case '-':
         printf("%f", a - b);
         break;
      case '*':
         printf("%f", a * b);
         break;
      case '/':
         if ( b != 0)
            printf("%f", a / b);
         else
            printf("Division med 0!");
         break;
      default :
         printf ("Fel operator!");
   }
}
```

Switch-satsen fungerar så att satsens styruttryck, som ovan är variabeln op och som måste vara uppräknelig (heltal), beräknas. Därefter hoppar programmet in i den första case-del som överenstämmer med detta värde och *alla efterföljande satser utförs*. Om det ej finns något överenstämmande värde utförs satserna i default-delen.

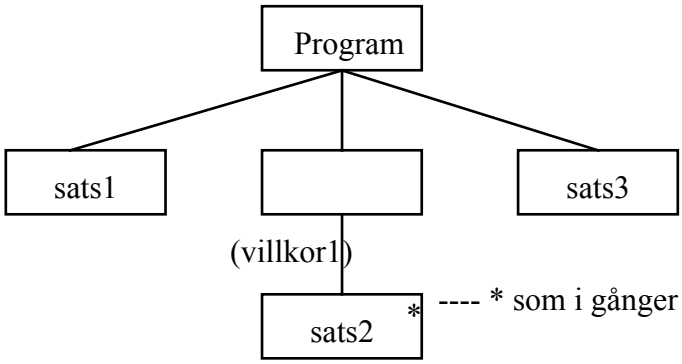
OBS! *Man måste hoppa ur switch-satsen med break*. Har man ej med break kommer alla efterföljande delar att också utföras. Har man exempelvis läst in ett *-tecken kommer programmet att utföra multiplikation, division och även skriva ut ett felmeddelande om felaktig operator om den aktuella case-delen skulle sakna break.

3.2 Iteration

Med styrmekanismen *iteration* kan man välja att upprepa satser *så länge något villkor är uppfyllt*.

Ex: I ovanstående första exempel ska sats2 upprepas så länge villkor1 är sant.

Strukturdiagram:



Halvkod:

```
Program
  sats1
  så länge villkor1
    sats2
  sats3
```

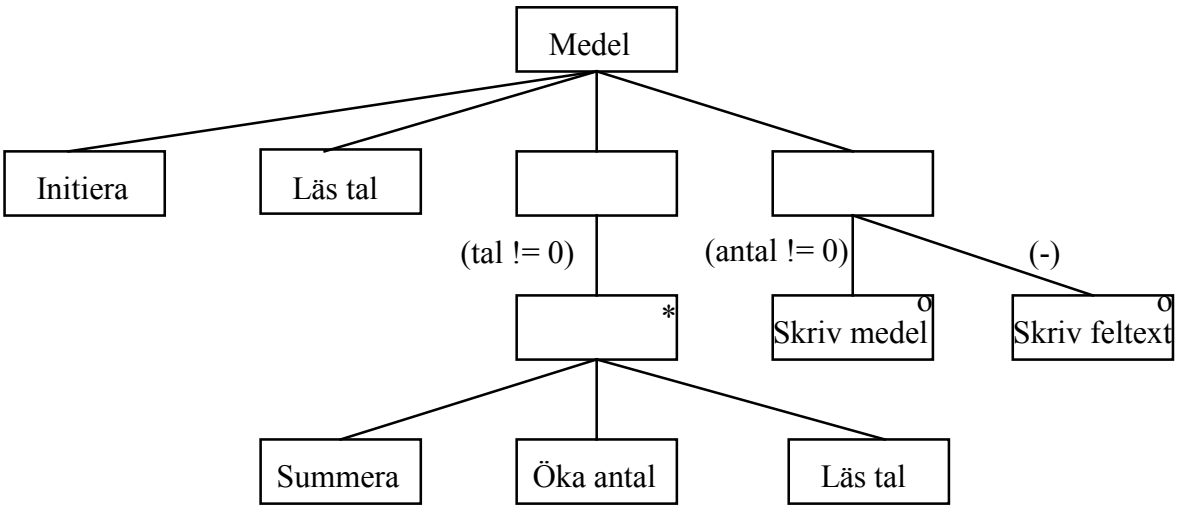
C-kod:

```
sats1;
while (villkor1)
   sats2;
sats3;
```

Villkor1 är ett logiskt uttryck som kan anta värdet falskt (0) eller sant (1). Då programmet kommer till while-satsen kontrollerar det värdet på villkor1. Är detta sant utförs sats2 och ny kontroll av villkor1 görs o.s.v. Då villkoret blir falskt fortsätter programmet med sats3.

Ex: Skriv ett program som läser in tal och summerar talen så länge de inlästa talet är skilt ifrån 0. Efter inläsning ska medelvärdet skrivas ut om något tal lästs in annars feltext.

Strukturdiagram:



C_kod:

```
#include <stdio.h>

void main()
{
   float tal, summa = 0.0;
   int antal = 0;

   printf("Ge tal (avsluta med 0) : ");
   scanf("%f", &tal);

   while (tal != 0)
   {                                          /* OBS! */
      summa += tal;
      antal++;
      printf("Ge tal (avsluta med 0) : ");
      scanf("%f", &tal);
   }                                          /* OBS! */

   if (antal != 0)
      printf("Medel = %f\n", summa / antal);
   else
      printf("Finns inget medel att beräkna!\n");
}
```

OBS! Om flera satser ska upprepas måste *blockparenteserna* { } användas.

En *verklig iteration ska kunna upprepas 0, 1 eller flera gånger*. Ibland kanske man vill att en sats alltid ska utföras minst en gång och sedan eventuellt upprepas så länge ett villkor är sant. I C finns en speciell konstruktion för detta som heter *do ... while*.

Ex: I vårt arbetsexempel vet man att sats2 ska köras minst en gång och sedan upprepas så länge villkor1 är sant.

Halvkod:

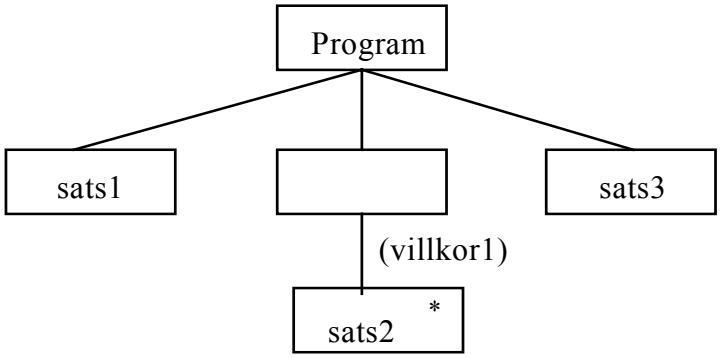
```
Program
  sats1
  gör
    sats2
  så länge (villkor1)
  sats3
```

C-kod med do .. while:

```
sats1;
do
   sats2;
while (villkor1);
sats3;
```

Strukturdiagram:

I strukturdiagrammet markerar man med texten do .. while (villkor1) istället för med bara while.



Ex: Ekvationen $x^3 - 25x^2 + 18x - 450 = 0$ har en lösning som är ett litet positivt heltal. Skriv ett program som hittar denna lösning genom att pröva med talen 1, 2, 3 o.s.v.

C-kod med while:

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

void main()
{
   int y, x = 0;

   x++;
   y = pow(x, 3) - 25* pow(x, 2) + 18*x - 450;

   while (y != 0)
   {
      x++;
      y = pow(x, 3) - 25* pow(x, 2) + 18*x - 450;
   }

   printf("x = %d\n", x);
}
```

C_kod med do .. while:

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

void main()
{
   int y, x = 0;

   do
   {
      x++;
      y = pow(x, 3) - 25* pow(x, 2) + 18*x - 450;
   }
   while (y != 0);

   printf("x = %d\n", x);
}
```

Ex: Från en *meny ska man upprepat kunna välja olika alternativ*. I menyn ska finnas ett alternativ som man kan avsluta upprepningen med. Skriv ett program som upprepat visar menyn:

K Kvadrat
R Rot
S Sluta

Välj >>

mitt på en tom skärm. Programmet ska kunna läsa reella tal, skriva ut kvadraten eller kvadratroten mitt på skärmen.

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

/* macro för att rensa bufferten */
#define SKIPLINE while (getchar() != '\n')

void main()
{
   float tal;
   char svar;

   do
   {
      /* menytext */
      gotoxy(30, 10);
      printf("K   Kvadrat");
      gotoxy(30, 11);
      printf("R   Roten");
      gotoxy(30, 12);
      printf("S   Sluta");
      gotoxy(30, 14);
      printf("Välj >> ");

      /* läs svar och rensa bufferten */
      svar = getchar();
      SKIPLINE;
```

```
      switch (svar)
      {
         case 'k':
         case 'K':
            /* läs tal och rensa */
            printf("Ge ett tal : ");
            scanf("%f", &tal);
            SKIPLINE;

            /* skriv kvadraten */
            printf("Kvadraten : %f\n", pow(tal, 2));
            printf("Tryck RETURN!!\n");
            SKIPLINE;
            break;

         case 'r':
         case 'R':
            /* läs tal och rensa */
            printf("Ge ett tal : ");
            scanf("%f", &tal);
            SKIPLINE;

            /* skriv roten */
            if ( tal >= 0 )
               printf("Kvadratroten : %f\n", sqrt(tal));
            else
               printf("Kvadratroten ej definierad!\n");
            printf("Tryck RETURN!!\n");
            SKIPLINE;
            break;

         case 's':
         case 'S':
            /* gör ingenting */
            break;
         default:
            /* skriv feltext */
            printf("Fel val!!\n");
            printf("Tryck RETURN!!\n");
            SKIPLINE;
      }
   }
   while (svar != 's' && svar != 'S');
}
```

Ibland kanske man vill att en sats ska *upprepas ett visst antal gånger*. Istället för att använda en räknare som initieras och räknas upp så länge den är mindre än antalet kan man i C använda en konstruktion med *for*.

Ex: I vårt arbetsexempel vill man att sats2 ska upprepas 20 gånger.

Halvkod:

```
Program
  sats1
  nr = 1
  så länge (nr <= 20)
    sats2
    nr++
  sats3
```

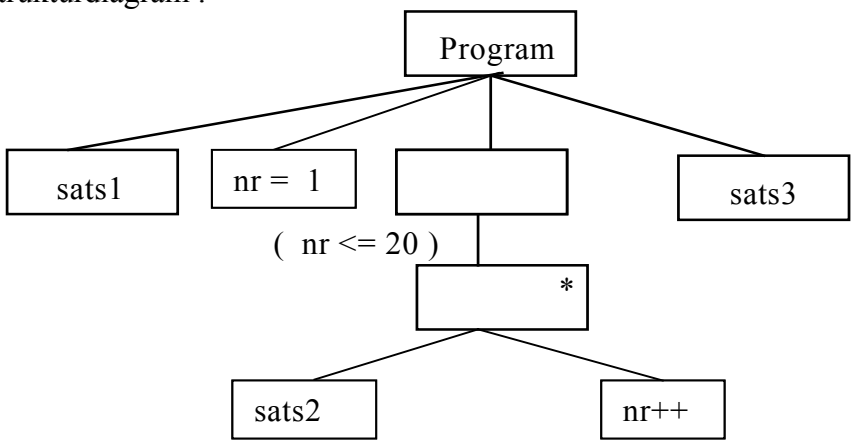
C-kod med while-konstruktion:

```
int nr;

sats1;
nr = 1;
while (nr <= 20)
{
   sats2;
   nr++;
}
sats3;
```

C-kod med for-konstruktion:

```
int nr;

sats1;
for(nr = 1; nr <= 20; nr++)
   sats2;
sats3;
```

Strukturdiagram :



Ex: Skriv ett program som frågar efter ett positivt heltal och skriver ut multiplikationstabellen för talet. Matar man exempelvis in talet 5 ska utskriften bli:

0 * 5 = 0
1 * 5 = 5
2 * 5 = 10
3 * 5 = 15
4 * 5 = 20
5 * 5 = 25

C-kod med while-sats:

```
#include <stdio.h>

void main()
{
   int tal, mult;

   printf("Ge talet : ");
   scanf("%d", &tal);

   mult = 0;
   while (mult <= tal)
   {
      printf("%d * %d = %d\n", mult, tal, mult*tal);
      mult++;
   }
}
```

C-kod med for-sats:

```
#include <stdio.h>

void main()
{
   int tal, mult;

   printf("Ge talet : ");
   scanf("%d", &tal);

   for (mult = 0; mult <= tal; mult++)
      printf("%d * %d = %d\n", mult, tal, mult*tal);
}
```

Man kan naturligvis ha nästlade iterationer precis som man kan ha nästlade selektioner. Man pratar om yttre och inre iteration. Den inre iterationen kommer att upprepas *fullständigt* varje gång som den yttre iterationen upprepas.

Ex: Skriv ett program som tillverkar ett mönster på skärmen enligt:

```
XXXXXXXXXXXXXXXXX
XXXXXXXX XXXXXXXX
XXXXXXX   XXXXXXX
XXXXXX     XXXXXX
XXXXX       XXXXX
XXXX         XXXX
XXX           XXX
XX             XX
X               X
```

```
#include <stdio.h>
void main()
{
   int rad, kol;

   /* för alla rader */
   for (rad = 1; rad <= 9; rad++)
   {
      /* för alla kolumner */
      for (kol = 1; kol <= 17; kol++)
      {
         if (kol > (10 - rad) && kol < (rad + 8))
            printf(" ");
         else
            printf("X");
       }
      printf("\n");
   }
}
```

OBS! I en nästlad loop körs alltid den innersta loopen färdigt innan man går till den yttre loopen och kör nästa varv. I exemplet ovan innebär det att man för varje rad kör färdigt alla kolumner innan man går till nästa rad.

3.3 Övriga styrmekanismer

Man kan även styra programflödet med hjälp av hopp. I C finns möjligheter att hoppa till en angiven plats i programmet med goto.

Ex:

```
.....
if (villkor1)
   goto slut;
.....
slut : printf("Ett fel har uppstått. Vi måste avbryta!");
```

Här finns slut som sista sats i programmet så att programmet avsutas om villkor1 är uppfyllt. Man kunde ha uppnått samma sak genom att använda funktionen exit i process.h enligt:

```
if (villkor1)
{
   printf("Ett fel har uppstått. Vi måste avbryta!");
   exit(1);
}
```

Man kan hoppa ur en *iteration* eller en selektion i form av en *switch-sats* med hjälp av break.

Ex:

```
.....
while (villkor1)
{
   .....
   if (villkor2)
      break;
}
.....
```

Här hoppar programmet ur while-loopen om villkor2 är sant.

Man kan hoppa över en upprepning med continue.

Ex:

```
.....
for (nr = 0; nr <= 10; nr++)
{
   if (nr == 5)
      continue;
   sats1;
}
```

Här utföres sats1 ej när nr är 5 utan man fortsätter direkt med nr = 6.

OBS! **Hopp försvårar programunderhåll. Försök att undvika hopp i största möjliga utsträckning. Använd istället strukturerna sekvens, selektion och iteration.**

4 Sammansatta datatyper

De enkla datatyper som vi hittills använt är otillräckliga när man ska hantera stora datamängder. Vill man exempelvis läsa in 100 reella mätvärden, som man tillfälligt vill spara under bearbetningen, måste man definiera 100 float-variabler och skriva 100 inläsningssatser. Här märker man att det finns ett behov av *sammansatta datatyper*.

Består datamängden av ett antal data av *samma typ* använder man *vektorer eller indexerade variabler*. Med hjälp av indexet kan man komma åt de enskilda datadelarna. Man har också ofta behov av variabler som kan innehålla *olika typer* av element exempelvis en medlem i en förening ska ha medlemsnummer, namn, telefon etc. För att kunna hantera denna typ av data finns det möjligheter att samla ihop olika typer av data till en *post eller struct*.

4.1 Vektorer

En *vektor* eller *indexerad variabel* eller *array* eller *fält* kan innehålla ett antal element av *samma typ*.

Ex: Skapa en vektor bestående av 5 reella element och tilldela elementen värdena 1.1, 2.2, 3.3, 4.4 resp. 5.5

```
float vek[5];   /* 5 reella element */

vek[0] = 1.1;   /* OBS! Första elementet har index 0 */
vek[1] = 2.2;
vek[2] = 3.3;
vek[3] = 4.4;
vek[4] = 5.5;
```

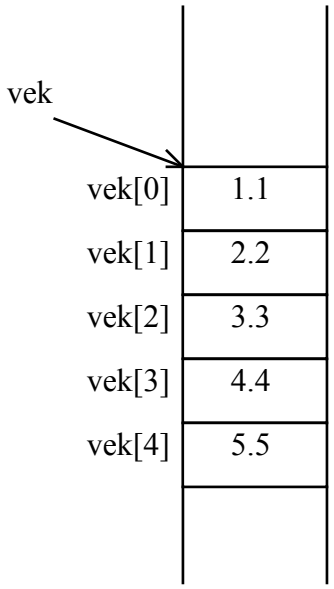
OBS! I *definitionen anges antalet element* och de enskilda elementen kommer man åt med indexering. Det första elementet har *alltid index 0*.

I ovanstående exempel ser man inte direkt fördelarna med att använda vektorer. Man kan dock förenkla koden genom att exempelvis initiera vektorn direkt vid definitionen och skriva ut hela vektorn med hjälp av en enda iteration.

Ex: Skapa en vektor bestående av 5 reella element och initiera elementen med värdena 1.1, 2.2, 3.3, 4.4 resp. 5.5. Skriv sedan ut vektorn med ett element per rad.

```
float vek[5] = {1.1, 2.2, 3.3, 4.4, 5.5};
int i;

for (i = 0; i <= 4; i++)
   printf("%f\n", vek[i]);
```

När man definierar en vektorvariabel reserverar man utrymme i minnet för det angivna antalet element. Själva namnet på vektorn kommer att vara *adressen till minnesutrymmets början*. Elementen i vektorn kommer sedan att ligga i minnet med början på denna adress. För ovanstående vektor kommer minnet att se ut som:



Varje element i en vektor kan behandlas som en vanlig variabel av elementens typ. Fördelen mot att använda helt nya variabelnamn för varje element är att man kan utnyttja samma variabel och räkna upp index med hjälp av en iteration istället.

Ex: Skriv de satser som summerar vektorn ovan.

```
float vek[5] = {1.1, 2.2, 3.3, 4.4, 5.5}, sum = 0;
int i;

for (i = 0; i <= 4; i++)
   sum += vek[i];
```

Själva vektorvariabeln kan man inte göra mycket med. Man kan exempelvis *inte tilldela en hel vektor till en annnan vektor* och därmed få en kopia utan man måste *tilldela varje element* för sig.

Ex: Skriv de satser som kopierar ovanstående vektor till en ny vektor xvek med lika många element:

```
float vek[5] = {1.1, 2.2, 3.3, 4.4, 5.5}, xvek[5];
int i;

for (i = 0; i <= 4; i++)
   xvek[i] = vek[i];
```

Ex: Skriv ett C-program som läser in 10 reella mätvärden till en vektor, sorterar dessa i stigande ordning och skriver ut de sorterade värdena med 5 mätvärden per rad.

```
#include <stdio.h>

void main()
{
   float xdata[10], temp;
   int i, j;

   /* läs in data */
   for (i = 0; i <= 9; i++)
   {
      printf("xdata[%d] = ", i);
      scanf("%f", &xdata[i]);
   }

   /* sortera data */
   for (i = 0; i <= 8; i++)
      for (j = i + 1; j <= 9; j++)
         if (xdata[j] < xdata[i])
         {
            /* byt plats */
            temp = xdata[i];
            xdata[i] = xdata[j];
            xdata[j] = temp;
         }

   /* skriv data */
   for (i = 0; i <= 9 ; i++)
   {
      if  (i % 5 == 0)
         printf("\n");
      printf("%f", xdata[i]);
   }
}
```

Sorteringen i exemplet går till så att man jämför första elementet i vektorn med alla resterande och byter plats så fort man hittar ett element som är mindre. Efter ett varv har det minsta elementet placerats först. Därefter fortsätter man med det andra elementet o.s.v. När man kört igenom alla element så har man en sorterad vektor. Detta är ingen snabb sorteringsalgoritm eftersom man gör en mängd onödiga byten. Det finns mycket snabbare algoritmer för sortering.

OBS! Användandet av en temporär variabel vid bytet för att inte tappa bort information. Ska man byta plats mellan variablerna x och y måste man göra i tre steg enligt:

```
temp = x;
x = y;
y = temp;
```

Ex: Skriv ett program som slumpar 1000 tärningskast mellan 1 och 6 och sedan skriver ut en tabell med antalet utfallna ettor, tvåor etc enligt:

```
1       178
2       201
3       154
4       167
5       189
6       111
```

```
# include <stdio.h>
# include <stdlib.h>  /* srand, rand */
# include <time.h>    /* srand använder tidsfunktion */

void main()
{
   int antal[7], utfall, nr;

   /* nolla antal-tabellen */
   for (utfall = 1; utfall <= 6; utfall++)
      antal[utfall] = 0;

   /* starta slumpningen slumpmässigt */
   srand((unsigned)time(NULL));

   /* slumpa 1000 kast och uppdatera antal-tabellen */
   for (nr = 1; nr <= 1000; nr++)
   {
      utfall = rand()%6 + 1;
      antal[utfall]++;
   }

   /* skriv ut antal-tabellen */
   for (utfall = 1; utfall <= 6; utfall++)
      printf("%d        %d\n", utfall, antal[utfall]);
}
```

OBS! Med srand-funktionen fås olika slumpserier varje gång programmet körs. Funktionen använder klockan i time.h för att åstadkomma detta.

OBS! Funktionen rand() returnerar ett slumpat heltal mellan 0 och RAND_MAX. Med rand()%6 fås tal mellan 0 och 5 och med +1 fås tal mellan 1 och 6.

OBS! Antal-tabellen som har ett extra element antal[0] som aldrig utnyttjas. Detta är ej nödvändigt men ger tydligare och mer lättläst kod, eftersom indexet då kommer att överensstämma med utfallen 1 till 6.

Som index i vektorer måste man alltid ha heltal som börjar med 0. Accepterar kompilatorn att egenuppräknade variabler betraktas som heltal kan man även indexera med sådana. När det gäller elementens typ finns det inga begränsningar utan där kan man ha *vilken typ som helst*. Man kan exempelvis ha en vektor där elementen är bokstäver eller tecken och man kan på detta sätt behandla sammansatt *data i form av flera tecken alltså ord eller text*.

Ex: Skriv ett program som läser in en rad av text från tangentbordet och skriver ut texten baklänges. Raden kan maximalt innehålla 80 tecken.

```
#include <stdio.h>

void main()
{
   char text[80], ch;
   int nr = 0, i;

   /* läs in texten tecken för tecken */
   printf("Skriv en rad med text!\n");
   ch = getchar();
   while (ch != '\n')
   {
      text[nr] = ch;
      nr++;
      ch = getchar();
   }

   /* skriv texten baklänges */
   printf("Texten baklänges!\n");
   for (i = nr - 1; i >= 0; i--)
      putchar(text[i]);
}
```

Vektorn text innehåller exempelvis tecknen enligt:

text[0]	'H'
text[1]	'e'
text[2]	'j'
text[3]	' '
text[4]	'p'
text[5]	'å'
text[6]	' '
text[7]	'D'
text[8]	'i'
text[9]	'g'
text[10]	'!'

Vektorer av tecken eller strängar som det också kallas är ofta förekommande i programmen och man har därför gett dessa speciella egenskaper utöver vanliga vektorer.

4.1.1 Strängar

Vektorer av tecken eller *strängar*, som det kallas, förekommer ofta i program som bearbetar text.

Ex: En strängvariabel kan ses som en vektor av tecken enligt exempelvis:

```
char strang[20] = {'H', 'e', 'j'};
```

Strang kan innehålla maximalt 20 tecken och varje tecken kommer man åt med hjälp av indexering, som exempelvis strang[0] som är tecknet 'H'.

Oftast har man plats för fler tecken i strängen än vad den innehåller just för ögonblicket. För att man ska kunna hålla reda på hur många tecken som strängen för ögonblicket innehåller har man i *C infört konventionen att det sista tecknet i strängen alltid är noll-tecknet, '\0'*, alltså det tecken som har ASCII-koden 0. Man brukar prata om noll-terminerade strängar. Alla funktioner som bearbetar strängar kommer sedan att utnyttja '\0'-tecknet vid sina bearbetningar.

Ex: Strängen i exemplet ovan är en vektor av tecken men egentligen ingen sträng eftersom den saknar '\0'-tecknet. För att det ska bli en riktig sträng kan man sätta dit ett '\0'-tecken explicit eller också använda beteckningen med citat-tecken för en strängkonstant enligt:

```
char strang[20] = {'H', 'e', 'j', '\0'};

char strang[20] = "Hej";
```

Ska man exempelvis beräkna strängens längd i sitt program kan man loopa fram till '\0'-tecknet samtidigt som man räknar upp en räknare enligt:

```
i = 0;
while ( strang[i] != '\0')
   i++;
```

Längden av strängen kommer att vara slutvärdet på i, som ovan blir 3. Man måste *alltid dimensionera strängens längd så att '\0'-tecknet får plats*.

Det avslutande '\0'-tecknet *sätts ofta in automatiskt* när man exempelvis *initierar en sträng* eller *läser in* en sträng med någon färdig inmatningsfunktion. Läser man in en sträng tecken för tecken måste man dock själv sätta dit '\0'-tecknet.

Eftersom strängar är ofta förekommande i program finns det speciella in- och utmatningsfunktioner för dessa så att man slipper att läsa eller skriva strängen tecken för tecken. Med de vanliga I/O-funktionerna scanf och printf kan man genom att använda omvandlingsspecifikationen %s läsa in resp skriva ut strängar.

Ex: Skriv ett program som läser in en sträng från tangentbordet och skriver ut strängen med stora bokstäver.

```
#include <stdio.h>

void main()
{
   char strang[80];
   int i;

   /* läs in sträng */
   printf("Ge en sträng : ");
   scanf("%s", strang);          /* OBS! Inget & före strang */

   /* gå igenom hela strängen */
   i = 0;
   while (strang[i] != '\0')
   {
      /* om liten bokstav byt till stor */
      /* som har ASCII liten - 32 */
      if (strang[i] >= 'a' && strang[i] <= 'z')
        strang[i] -= 32;
      i++;
   }

   /* skriv ut den nya strängen */
   printf("Strängen med stora bokstäver : %s\n", strang);
}
```

OBS! Ingen & före strang-variabeln eftersom strang redan är en adress.

Ett första körexempel:
Ge en sträng : Hej
Strängen med stora bokstäver : HEJ

Ett andra körexempel:
Ge en sträng : Hej Eva
Strängen med stora bokstäver : HEJ

Det andra körexemplet visar att programmet ej fungerar så bra för alla strängar. Detta beror på att inläsningen med scanf bara läser in tecken till strängen *fram till första separatorn som kan vara blanktecken, TAB-tecken eller RETURN-tecken*. I ovanstående fall kommer ett blanktecken efter j och då slutar scanf att läsa och lämnar kvar resten av tecknen i inmatnings-bufferten. För att även kunna läsa in blanktecken kan man använda funktionen gets som läser fram till RETURN-tecknet.

Ex: Skriv ett program som läser in ditt namn och skriver ut dina initialer med stora bokstäver. Exempelvis ska inmatningen Bertil Jonsson skriva ut BJ och Ulla Maria Karlsson skriva ut UMK.

```
#include <stdio.h>

void main()
{
   char namn[80];
   int i;

   /* läs ett namn */
   printf("Ge ditt namn : ");
   gets(namn);

   /* plocka ut och skriv initialerna */
   i = 0;
   putchar(namn[i]);
   while (namn[i] != '\0')
   {
      if (namn[i] == ' ')
         putchar(namn[i + 1]);
      i++;
   }
}
```

OBS! Inläsningen av namnet med funktionen gets som läser fram till RETURN-tecknet. RETURN-tecknet *tas bort från inmatnings-bufferten men placeras inte i strängen*. Man måste själv kontrollera att den inlästa strängen ryms i motsvarande variabeln. *Glöm ej att även '\0'-tecknet måste rymmas*.

Det finns en motsvarande funktion för utskrift av strängar också som heter puts. Funktionen puts skriver ut strängen och *även ett avslutande RETURN-tecken*.

Gets och puts för in- och utmatning av strängar kan jämföras med getchar och putchar för in- och utmatning av tecken.

Elementen i en sträng är tecken och kan behandlas som vanliga teckenvariabler. Sträng-variabeln däremot är en vektor, *alltså en adress i minnet till strängens första tecken*. Här gäller alltså precis samma restriktioner vad gäller operationer som för vanliga vektorer. Man kan exempelvis *inte tilldela en sträng till en annan sträng* och man kan *inte jämföra två* strängar med varandra.

Vill man kopiera en sträng till en annan eller jämföra två strängar måste man göra detta tecken för tecken eller också använda *de färdiga funktioner som finns i string.h*. Där finns bl.a funktionen *strcpy* som kopierar, *strcmp* som jämför och *strlen* som beräknar längden av en sträng.

4.1.2 Flerdimensionella vektorer

Har man data i form av tabeller och vill spara dessa variabler i sitt program kan man använda flerdimensionella vektorer.

Ex: Skriv ett program som skapar en variabel som motsvarar nedanstående tabell och beräknar och skriver ut tabellens radsummor och den totala summan.

2.3	4.2	3.4	5.6
1.8	3.5	5.8	2.2
7.8	6.7	4.5	1.3

```
#include <stdio.h>

void main()
{
   float tabell[3][4] =    { { 2.3, 4.2, 3.4, 5.6},
                             { 1.8, 3.5, 5.8, 2.2},
                             { 7.8, 6.7, 4.5, 1.3} };
   float radsum, totalsum = 0;
   int rad, kol;

   /* summera tabellen och skriv ut */
   for (rad = 0; rad <= 2; rad++)
  {
      radsum = 0;

      /* summera raden */
      for (kol = 0; kol <= 3; kol++)
         radsum += tabell[rad][kol];  /* OBS! Indexering */

      /* skriv ut radsumman */
      printf ("Summan av rad %d = %f", rad + 1, radsum);

      totalsum += radsum;
   }
   printf("Totalsumman = %f", totalsum);
}
```

OBS! Indexeringen med *dubbla hakparenteser* av flerdimensionella vektorer som börjar med tabell[0][0] och fortsätter med tabell[0][1] o.s.v. Egentligen består en flerdimensionell vektor av ett antal enkla vektorer eller radvektorer. Den första raden ovan är vektorn tabell[0], den andra tabell[1] o.s.v.

Tvådimensionella vektorer kallas också matriser. Man kan ha fler dimensioner än två i sina tabeller.

En tabell innehållande namn kan ses som en tvådimensionell vektor av tecken eller en vektor av strängar.

Ex: Skriv ett program som läser in ett antal namn , max 10, till en tabell och sedan skriver ut namnen i omvänd ordning. Namnen kan innehålla maximalt 30 tecken. Inläsningen ska avslutas med ett tomt namn alltså bara RETURN.

```
#include <stdio.h>

void main()
{
   char namnlista[10][30]; /* 10 namn med max 30 tecken */
   int i, nr = 0;

   /* läs in namnen och avsluta med tom sträng */
   printf("Ge ett namn (Avsluta med bara RETURN) : ");
   gets(namnlista[nr]);
   while (namnlista[nr][0] != '\0')
   {
      nr++;
      printf("Ge ett namn (Avsluta med bara RETURN) : ");
      gets(namnlista[nr]);
   }

   /* skriv ut namnen i omvänd ordning */
   printf("Namnen i omvänd ordning!\n");
   for (i = nr - 1; i >= 0; i--)
      puts(namnlista[i]);
}
```

Ett körexempel:

Ge ett namn (Avsluta med bara RETURN) : Eva Olsson RETURN
Ge ett namn (Avsluta med bara RETURN) : Olle Karlsson RETURN
Ge ett namn (Avsluta med bara RETURN) : RETURN

Namnen i omvänd ordning!
Olle Karlsson
Eva Olsson

Namnlista är en vektor av strängar eller en matris av tecken. Vid ovanstående körning kommer listan att se ut som:

namnlista[0] ----- "Eva Olsson" ----- {'E', 'v', 'a', ' ', 'O', 'l', 's', 's', 'o', 'n', '\0'}
namnlista[1] ----- "Olle Karlsson" ----- {'O', 'l', 'l', 'e', ' ', 'K', 'a', 'r', 'l', 's', 's', 'o', 'n', '\0'}
namnlista[2] ----- "" ----- {'\0'}

Alltså är namnlista[2][0] lika med '\0' och loopen avslutas.

4.2 Poster

En *post eller struct* är en sammansatt datatyp vars delar kan bestå av *olika datatyper*. Delarna i en post kallas för *termer*.

Ex: Man ska hantera data i form av mätvärden där varje mätvärde ska ha ett nummer. Skapa två sådana mätposter och tilldela dessa några godtyckliga värden.

```
struct matdata
{
   int nr;
   float x;
} m1, m2;


m1.nr = 1;
m1.x = 2.3;

m2.nr = 2;
m2.x = 6.7;
```

Man kommer åt de enskilda termerna med *punktnotation*. Termen m2.x har samma egenskaper som vilken flyttalsvariabel som helst.

Istället för att skriva variabelnamnen direkt vid typen kan man dela upp definitionen precis som man kunde göra för egenuppräknade variabler enligt:

```
struct matdata
{
   int nr;
   float x;
};

struct matdata m1, m2;
```

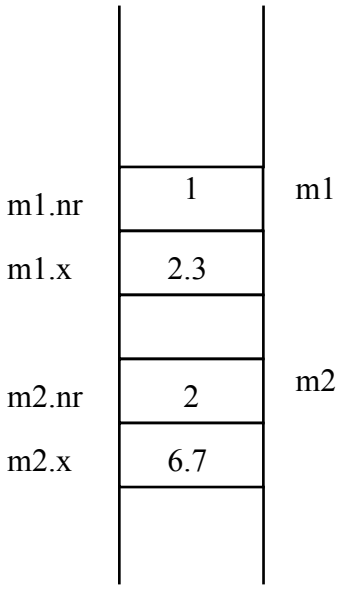
Här har vi definierat variablerna m1 och m2 av typen *struct matdata*.

På samma sätt som för vektorer kan man vid variabeldefinitionen initiera sina poster med värden för resp term.

Ex: Skapa ovanstående poster med initiering istället för tilldelning.

```
struct matdata
{
   int nr;
   float x;
};

struct matdata m1 = {1, 2.3}, m2 = {2, 6.7};
```

När man definierar en postvariabel reserverar man utrymme i minnet för de angivna termerna. Själva namnet på posten kommer att vara ett samlingsnamn för en variabel som motsvarar hela minnesutrymmet. Minnesbilden för våra mätposter ovan kommer att se ut som:



Varje term i en post kan behandlas som en vanlig variabel av termens typ.

Ex: Skriv de satser som skapar en post med nummer 12 som innehåller summan av mätvärdena i post m1 och m2 ovan.

```
struct matdata m1 = {1, 2.3}, m2 = {2, 6.7}, msum;

msum.nr = 12;
msum.x = m1.x + m2.x;
```

Själva postvariabeln kan man se som namnet på hela variabeln. Man kan exempelvis, i motsats till vektorer, *tilldela hela poster till varandra* och därmed få en kopia. Däremot kan man inte läsa in eller skriva ut hela poster, utan man måste göra detta term för term.

Ex: Skriv de satser som läser in värden till en mätpost enligt ovan, kopierar posten till en ny post och skriver ut den nya postens värden.

```
struct matdata m, mkopia;

printf("Ge mätpostens nummer : ");
scanf("%d", &m.nr);                    /* OBS! Adress */
printf("Ge mätpostens värde : ");
scanf("%f", &m.x);

mkopia = m;

printf("Kopians nummer : %d\n", mkopia.nr);
printf("Kopians värde  : %f\n", mkopia.x);
```

Ex: Skriv ett program som börjar med att fråga efter antalet mätposter och sedan läser in värden till termerna i dessa poster, håller reda på största och minsta mätvärde samt avslutningsvis skriver ut största värde , minsta värde samt medelvärde av alla posters mätvärden.

```
#include <stdio.h>

void main()
{
   struct matdata
   {
      int nr;
      float x;
   } m, min, max;

   int i, antal;
   float xsum;

   /* läs in antal poster */
   printf("Ge antalet poster : ");
   scanf("%d", &antal);

   /* läs in första posten */
   printf("Ge postens nummer : ");
   scanf("%d", &m.nr);
   printf("Ge postens mätvärde : ");
   scanf("%f", &m.x);

   /* initiera med hjälp av första posten */
   xsum = m.x;
   min = max = m;

   /* läs in resterande poster */
   for (i = 2; i <= antal; i++)
   {
      printf("Ge postens nummer : ");
      scanf("%d", &m.nr);
      printf("Ge postens mätvärde : ");
      scanf("%f", &m.x);

      /* summera och byt ev ut min eller max */
      xsum += m.x;
      if (m.x < min.x)
         min = m;
      else if ( m.x > max.x)
         max = m;
   }
   printf("Största : %f i post nr %d\n", max.x, max.nr);
   printf("Minsta : %f i post nr %d\n", min.x, min.nr);
   printf("Medelvärde : %f av %d poster\n" ,xsum/antal,antal);
}
```

De data som programmen ska bearbeta är ofta sammansatta av olika datatyper. För att avbilda sådana data i datorn använder man poster.

Ex: Bilar i ett bilregister ska hålla reda på

-- registreringsnummer
-- ägare
-- bilmärke
-- årsmodell
-- skatt

En variabel bil som kan hålla reda på dessa data om bilen kan då definieras enligt:

```
struct bilpost
{
   char regnr[7];
   char agare[30];
   char marke[20];
   int arsmodell;
   float skatt;
};

/* definiera en variabel bil som initieras */
struct bilpost bil = {"ESA345", "Karin Larsson",
                      "VOLVO 240", 1985, 675};
```

Ex: För medlemmar i en förening ska man hålla reda på

-- medlemsnummer
-- namn
-- medlemsavgift
-- telefon

En variabel medlem som ska avbilda en medlem definieras då som:

```
struct medlemdata
{
   int medlemnr;
   char namn[30];
   float avgift;
   char tel[15];
};

struct medlemdata medlem = {120, "Kurt Olsson",
                            300, "019-122121"};
```

Termerna i posterna kan vara av vilken datatyp som helst. De kan som exemplen ovan visar vara strängar. De kan också vara vektorer eller poster.

När man har poster som termer i andra poster pratar man om nästlade poster.

Ex: I bilpost ovan kunde man istället för att bara ha bilägaren som en sträng, som bara innehåller namnet, ha den som en post som innehåller

-- personnummer
-- namn
-- gatuaadress
-- postnummer
-- ortsnamn

En variabel bil skulle då se ut som:

```
struct agaredata
{
   char pnr[12];
   char namn[30];
   char gatuadress[30];
   long postnr;
   char ortsnamn[20];
};


struct bilpost
{
   char regnr[7];
   struct agaredata agare;            /* OBS! */
   char marke[20];
   int arsmodell;
   float skatt;
};


/* skapa en variabel som initieras */
struct bilpost bil = { "ESA345",
                       {"111111-1111", "Ove Olsson",
                       "Xvägen 12", 73234, "Örebro"},
                       "VOLVO 240", 1985, 675};


/* nytt postnummer */
bil.agare.postnr = 12345;

/* ny skatt */
bil.skatt = 685;
```

OBS! En nästlad posts termer kommer man åt med upprepad punktnotation.

En posts term kan vara en vektor eller sträng som vi har sett. Man kan också ha en vektor bestående av ett antal poster.

Ex: Skriv ett program som simulerar mätning av temperatur på 20 st numerade mätställen. Temperaturen på de olika platserna slumpas fram som flyttal med en decimal mellan 20 och 30 grader. Posterna sorteras sedan i stigande temperatur och skrivs ut. Använd en vektor av poster där varje post har ett nummer och en temperatur.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void main()
{
   struct matdata
   {
      int nr;
      float temp;
   };

   struct matdata serie[20], slask;
   int i, j;

   /* slumpa temperaturer för mätställena  100 till 119 */
   srand((unsigned)time(NULL));
   for (i = 0; i <= 19; i++)
   {
      serie[i].nr = 100 + i;
      serie[i].temp = (rand()%101 + 200) / (float)10.0;
   }

   /* sortera vektorn i stigande temperaturer */
   for (i = 0; i <= 18; i++)
      for (j = i + 1; j <= 19; j++)
         if (serie[j].temp < serie[i].temp)
         {
            /* byt värde på posterna */
            slask = serie[i];
            serie[i] = serie[j];
            serie[j] = slask;
         }

   /* skriv ut vektorn */
   for (i = 0; i <= 19; i++)
      printf("Nr:%d  Temp:%f \n",serie[i].nr,serie[i].temp);
}
```

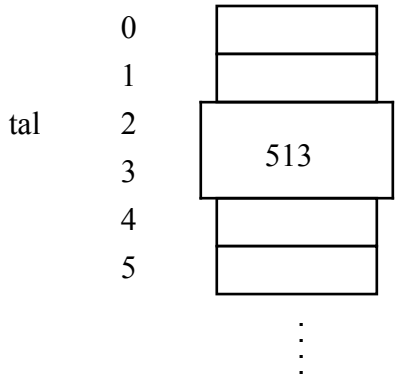
4.3 Adresser och pekare

Som vi har sett är en variabel ett symboliskt namn för en plats i minnet. Det är systemet som bestämmer var någonstans i minnet variabeln hamnar. Man kan i sitt C-program ta reda på variabelns adress genom att använda *adressoperatorn & (ampersand)*.

Ex: Definierar vi variabeln tal enligt:

```
short int tal = 513;
```

Om systemet placerar tal på adressen 2 ser en tänkt minnesbild ut som:



Värdet av variabeln tal, som man kommer åt med variabelnamnet tal, är 513 och *adressen för variabeln tal, som fås med &tal, är 2*.

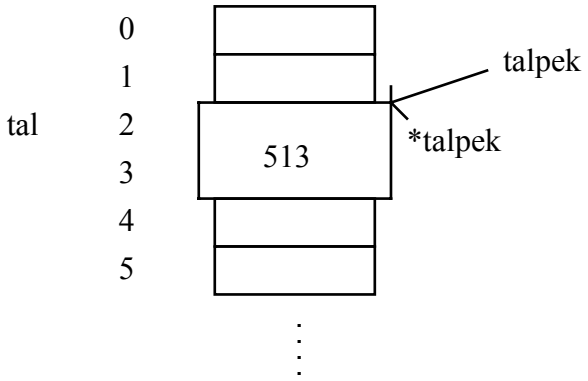
Det finns en speciell typ av variabel, *pekare*, *som kan ha värden i form av adresser*. Man tycker att ett adressvärde borde vara en unsigned int och att det inte skulle behövas någon ny typ för detta. Man har dock infört en speciell typ av variabel, pekare, för att hantera adresser eftersom man vill ha vissa *speciella operationer* för dessa.

En *variabel av typen pekare kan tilldelas ett värde i form av en minnesadress*. Man säger att variabeln då pekar på en plats i minnet, därav namnet. En pekarvariabel i sig är en variabel av enkel typ eftersom den bara innehåller en heltalsadress. Varför pekare tas upp i samband med sammansatta datatyper, beror på att man med hjälp av pekare kan hantera stora datamängder i minnet.

Ex: Definiera en pekarvariabel som tilldelas adressen för heltalet tal ovan.

```
short int tal = 513;
short int *talpek;

talpek = &tal;
```

Vi har fått en pekare till tal enligt:



Nu kan vi komma åt heltalet 513 på två olika sätt antingen med variabelnamnet tal eller via en *avreferering av pekaren talpek enligt *talpek*.

```
tal = 345;       /* nytt värde på tal 345 */

*talpek =  123; /* nytt värde på tal 123 */
```

OBS! *Tecknet * som används vid både definition av pekaren och avreferens av värdet som pekaren pekar på*.

När man definierar en pekarvariabel måste man alltid ange *vilken typ av data den ska peka på*. Varför man kräver detta beror på att systemet måste veta hur många byte som pekaren ska räknas upp då man adderar 1 till den. Att räkna upp en pekare med 1 innebär alltså inte att man ökar adressen med 1 utan man *ökar adressen med så många byte som den typ, som pekaren pekar på, tar upp i minnet*.

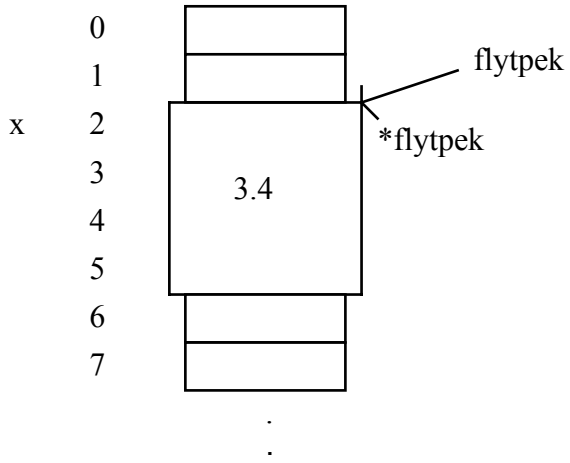
Ex: Pekaruppräkningen

```
talpek++;
```

innebär att talpek nu innehåller adressen 4 eftersom short int tar upp 2 byte i vårt system.

Ex : Definiera en pekarvariabel som kan peka på reella tal eller med andra ord kan tilldelas en adress till en reell variabel.

```
float *flytpek;
float x;

x = 3.4;
flytpek = &x;
```



Variabeln flytpek har värdet 2 och pekar på värdet 3.4. Vill man öka talet 3.4 med 2.3 kan man antingen använda x eller *flytpek enligt:

```
*flytpek = *flytpek + 2.3;
```

Variabeln flytpek har fortfarande värdet 2 och pekar nu på talet 5.7.

Vi kan öka pekarvärdet med 1 enligt:

```
flytpek++;
```

Variabeln flytpek har nu värdet 6 eftersom det är en pekare till float och i vårt sytem tar float upp 4 Byte. En avreferering av pekaren enligt:

```
*flytpek = 1.2;
```

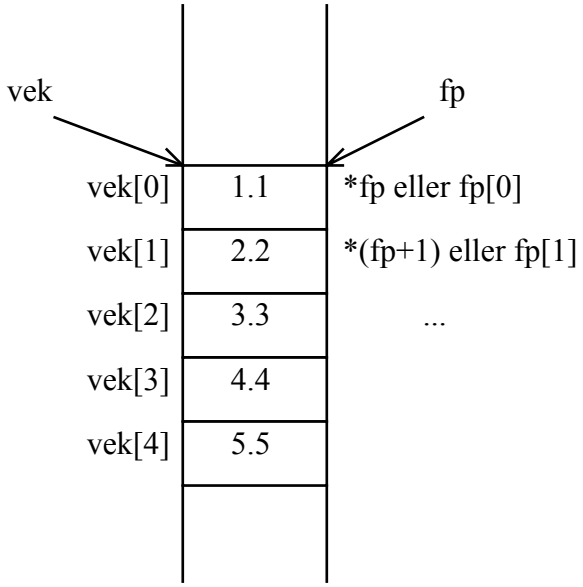
skriver nu i 4 byte med början på adressen 6. *Där kan finnas något väsentligt för att systemet ska fungera och datorn kraschar*. Man bör alltid se till att pekare har väldefinierade och tillåtna värden. Har man inget annat värde att tilldela kan man använda det fördefinierde och tillåtna pekarvärdet NULL enligt:

```
flytpek = NULL;
```

När det gäller vektorer och strängar så är *själva variabelnamnet redan en adress* till vektorns första element. Denna adress kan man naturligtvis tillldela till en pekarvariabel som då också kommer att peka på vektorns första element.

Ex: Definera en vektor med 5 flyttalselement och en pekare som sätts att peka på vektorns första element.

```
float vek[5] = {1.1, 2.2, 3.3, 4.4, 5.5 };

float *fp = vek;
```



Med indexering kan man nu exempelvis summera vektorn som tidigare enligt:

```
int i, sum = 0;

for (i = 0; i <= 4; i++)
   sum += vek[i];
```

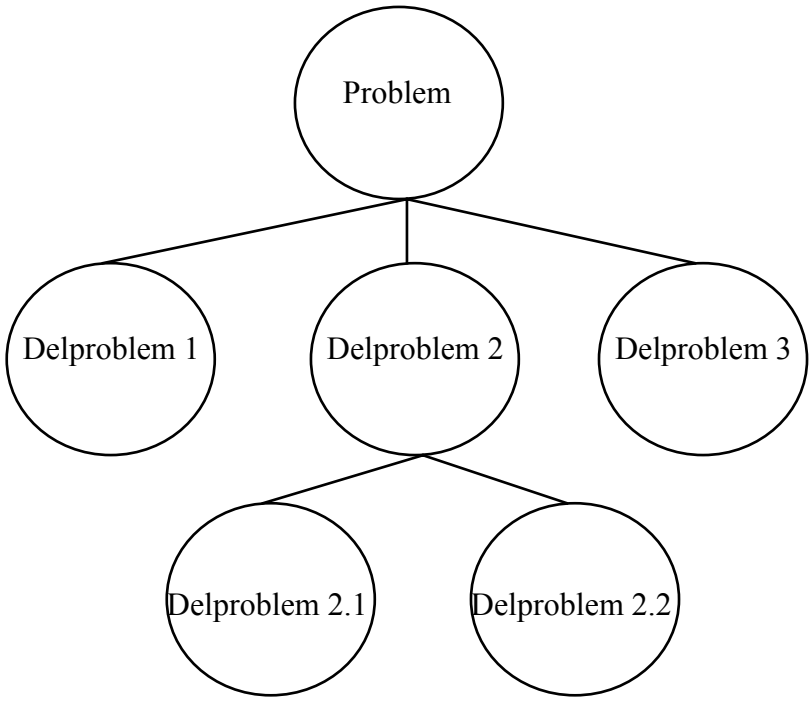
Samma sak kan man uppnå med pekaruppräkning enligt:

```
for(i = 0; i <= 4; i++)
{
   sum += *fp;
   fp++;
}
```

Pekaren fp pekar nu på elementet efter 5.5 och borde återställas till att peka på vektorns start eller också ges värdet NULL.

5 Funktioner

När man ska lösa stora komplexa problem måste man dela upp problemlösningen i hanterbara delar. Blir även dessa nya delar för stora för att lösas direkt får man dela upp ytterligare o.s.v



Att skriva program handlar om problemlösning och när det gäller att hantera stora program stöter man på samma svårigheter, som vid problemlösning. Det är svårt att hantera hela det stora programmet på en gång utan man blir tvungen att dela upp det i mindre delar, delprogram eller underprogram, precis som vid problemlösning.

I programmering brukar man kalla metoden att dela upp sitt program i delar för *stegvis förfining*. Man bryter ner sitt program till mindre och mindre delar där varje del får en speciell *avgränsad programmeringsuppgift* att lösa. Vissa delar måste kanske brytas ner i flera steg för att bli hanterbara.

När man har brutit ner sitt program i hanterbara delar är det dags att implementera eller koda dessa. Här kan man skapa delprogrammen högst upp först och sedan gå neråt *(top-down)* eller de längst ner först och fortsätta uppåt *(bottom-up)*. I verkligheten använder man sig av en blandning av dessa implementationsmetoder. För vissa delar av programmmet kodar och testar man delprogrammen på högre nivå först och skriver de på lägre nivå senare och för andra delar gör man tvärtom.

5.1 Fördefinierade funktioner

För att på ett enkelt sätt kunna dela upp sina program i mindre delar är det möjligt i de flesta språk att använda olika typer av delprogram eller underprogram. I C har man bara en enda typ av underprogram, nämligen *funktion*. Varje C-program är uppbyggd av ett antal funktioner. *Huvudprogrammet main , där programexekveringen alltid börjar*, är också en funktion.

Huvudprogrammet kan i sin tur innehålla ett antal anrop av andra funktioner. Dessa funktioner kan i sin tur ha anrop av andra funktioner o.s.v. För att kunna anropa en funktion i C måste den finnas *definierad* eller åtminstone *deklarerad* innan. Med funktionens *definition menas hela funktionen* medan funktionens *deklaration endast består av funktionens prototyp eller huvud.*

I C-språket ingår ett stort antal färdiga funktioner som kan användas om man plockar in deras deklarationer som finns i header-filer. Exempelvis finns deklarationen för printf i stdio.h. Själva funktionens kropp eller definition kommer in automatiskt vid länkningen.

Ex: Skriv ett program som läser in ett flyttal som en sträng, omvandlar strängen till tal, kvadrerar talet och skriver ut det mitt på skärmen.

```
#include <math.h>          /* pow, atof */
#include <stdio.h>         /* printf, gets */

void main()
{
   float x;
   char fstr[20];

   printf("Ge ett reellt tal : ");    /* anrop av printf */
   gets(fstr);                        /* anrop av gets */
   x = atof(fstr);                    /* anrop av atof */
   printf("%f", pow(x, 2));           /* anrop av printf, pow*/
}
```

Programmet ovan är uppbyggt av ett antal funktionsanrop till redan färdiga funktioner i C, som printf , gets etc. Då en funktion anropas hoppar programmet till koden för den anropade funktionen, som körs från början till slut. Efter att funktionen kört färdigt, fortsätter programmet med satsen efter funktionsanropet.

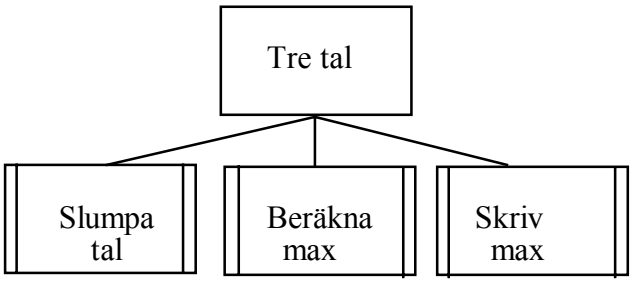
Till vissa funktioner kan man skicka med information. Till exempelvis funktionen pow ska man vid anropet skicka information om vilket tal som ska upphöjas till vad. Ska man ej skicka med någon information anges detta med tomma parenteser.

Vissa funktioner skickar tillbaks information. Funktionen atof returnerar exempelvis det omvandlade reella talet som funktionsvärde.

5.2 Egna funktioner

Finns inte färdiga funktioner att tillgå, kan man skriva sina egna funktioner i C. Speciellt ska man göra detta om man har *stora program* som ska delas upp i mindre delar enligt ovan. Det är också motiverat att skriva egna funktioner, om man har *generella delar* i sina program, alltså sådana delar som kan *återanvändas* av andra program. Det kan exempelvis vara sortering, sökning, menyer etc. Ytterligare orsak till att skriva egna funktioner har man om man i sitt program *har kod som upprepas ofta.*

Ex: Skriv ett program som slumpar tre hela tal mellan 0 och 99 och beräknar och skriver ut det största av dessa tre tal. Detta program är inget stort program där man är tvungen att dela upp i delprogram. Vi gör en uppdelning i alla fall bara för att visa principen.



Huvudprogrammet kommer då i princip bara att innehålla *anrop* av de tre funktionerna enligt:

```
void main()
{
   slumpa_tal();
   berakna_max();
   skriv_max();
}
```

Nästa steg i programutvecklingen är att man skriver de enskilda funktionerna. Dessa måste skrivas *före huvudprogrammet*. Man kan ej i C *skriva* funktioner inuti funktioner. Man kan bara *anropa* funktioner inuti andra funktioner.

Skriver man ej funktionerna före anropet måste det i alla fall finnas en deklaration av funktionen föra anropet. En funktionsdeklaration innehåller enbart funktionens prototyp, som består av funktionshuvudet avslutat med ett semikolon.

5.3 Informationsöverföring mellan funktioner

När man ska skriva funktionerna dyker det genast upp ett problem när det gäller *informationsöverföringen* mellan de olika funktionerna. *Hur får berakna_max reda på av slumpa_tal vilka tal som slumpats och hur skickar den det största talet till skriv_max för utskrift?*

Det finns i princip två sätt att lösa detta kommunikationsproblem. Man kan använda sig av *globala variabler,* d.v.s variabler som är definierade på global nivå ovanför main och som då kommer att gälla i alla funktioner eller man kan använda sig av *parametrar (argument)* och *returvärden från funktionerna*.

5.3.1 Globala och lokala variabler

Ex: Implementera informationsöverföringen i exemplet tre tal ovan med hjälp av globala variabler.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int x, y, z, max;          /* Globala variabler */


void slumpa_tal(void)
{
   srand((unsigned)time(NULL));
   x = rand()%100;
   y = rand()%100;
   z = rand()%100;
}


void berakna_max(void)
{
   if (x > y && x > z)
      max = x;
   else if (y > z)
      max = y;
   else
      max = z;
}


void skriv_max(void)
{
   printf("Största talet : %d\n", max);
}
```

```
void main()
{
   slumpa_tal();
   berakna_max();
   skriv_max();
}
```

OBS! Funktionerna måste implementeras före huvudprogrammet i samma fil annars förstår inte kompilatorn anropen av funktionerna som finns i huvudprogrammet main.

OBS! Ingen informationsöverföring sker med returvärden från funktionerna eller på annat sätt, därför markerar man med *void* vid implementationen av funktionerna både före funktionsnamnet och inuti parenteserna.

OBS! De globala variablerna x, y, z och max definieras allra först så att de är kända i alla funktioner. Hade man definierat variablerna x, y, z och max inuti main hade de blivit *lokala variabler* inuti main-blocket och bara kända där. Något informationsutbyte mellan huvudprogrammet och dess funktioner hade man inte fått då.

OBS! Globala variabler kan initieras med startvärden. Gör man ej detta blir de automatiskt noll-ställda. Lokala variabler kan också initieras men de får odefinierade värden om de lämnas oinitierade.

En nackdel med att använda globala variabler är att funktionerna som man tillverkar *ej blir generella och användbara för andra program*. För att exempelvis funktionen berakna_max ovan ska fungera tillsammans med något annat program måste man alltid se till att man använder samma globala variabelnamn x, y, z och max.

En annan nackdel med globala variabler är att man kan *få mycket svårfunna fel* i sina program eftersom en variabel kan ändras på så många olika ställen.

Ytterligare en nackdel är att program som innehåller globala variabler blir mycket *svåra att underhålla och göra förändringar* i. Det blir svårt att hitta alla platser i programmet där ändringar ska göras.

En bra regel : Använd globala variabler endast i undantagsfall !

5.3.2 Parametrar eller argument

Ett mycket bättre, säkrare och mer generellt sätt att föra över information mellan olika funktioner är att använda sig av *parametrar eller argument och returvärden från funktioner*.

Ex: Implementera ovanstående funktioner i programmet tre tal så att all informationsöverföring sker med parametrar och funktionsvärden.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void slumpa_tal(int *xp, int *yp, int *zp)
{
   srand((unsigned)time(NULL));
   *xp = rand()%100;
   *yp = rand()%100;
   *zp = rand()%100;
}

int berakna_max(int xf, int yf, int zf)
{
   int maxl;    /* lokal variabel gäller bara i berakna_max */

   if (xf > yf && xf > zf)
      maxl = xf;
   else if (yf > zf)
      maxl = yf;
   else
      maxl = zf;

   return maxl;
}

void skriv_max(int maxf)
{
   printf("Största talet : %d\n", maxf);
}

void main()
{
   int x, y, z, max;  /* lokala variabler gäller bara i main */

   slumpa_tal(&x, &y, &z);
   max = berakna_max(x, y, z);
   skriv_max(max);
}
```

I C sker all parameteröverföring med sk *värdeanrop* d.v.s de aktuella *parametrarnas* värden, som vid exempelvis anropet av berakna_max är x, y och z , kopieras över till motsvarande *formella parametrar* xf, yf och zf. *De aktuella parametrarna måste vara lika många och av samma typ som de formella*. Överkopieringen sker mellan samma positioner vilket innebär att x:s värde kopieras över till xf etc.
De tre funktionerna slumpa_tal, berakna_max och skriv_max har olika sätt att överföra information. Skillnaden beror på vilken riktning informationsöverföringen har. Ska

man bara skicka information till en funktion eller vill man också ha tillbaka information?

Funktionen skriv_max har den enklaste informationsöverföringen. Här behöver man enbart skicka över det värde som ska skrivas ut.

Funktion med formell parameter:

```
void skriv_max(int maxf)
{
   printf("Största talet : %d\n", maxf);
}
```

Anrop med aktuell parameter:

```
skriv_max(max);
```

Värdet av variabeln max kopieras över till maxf, som sedan skrivs ut.

Funktionen berakna_max har informationsöverföring i båda riktningarna. Det anropande programmet ska skicka tre tal till funktionen och funktionen i sin tur ska skicka tillbaks det största talet.

Funktionen med formella parametrar och returvärde:

```
int berakna_max(int xf, int yf, int zf)
{
   int maxl;

   if (xf > yf && xf > zf)
      maxl = xf;
   else if (yf > zf)
      maxl = yf;
   else
      maxl = zf;

   return maxl;
}
```

Anrop med aktuella parametrar och tilldelning av funktionsvärdet:

```
max = berakna_max(x, y, z);
```

Värdet av x, y och z kopieras över till xf, yf resp zf. Funktionen beräknar det största av dessa tal och tilldelar en *lokal variabel, maxl,* detta största värde som slutligen skickas tillbaka som funktionsvärde. I huvudprogrammet tilldelas detta funktionsvärde sedan till max. Jfr med en vanlig matematisk funktion.

Den besvärligaste informationsöverföringen ovan sker mellan huvudprogrammet och slumpa_tal. Funktionen slumpa_tal ska skicka tillbaka de tre slumpade värdena till x, y och z. Här kan man inte på ett enkelt sätt returnera värden med hjälp av funktionens returvärde eftersom de är 3 st olika värden som ska returneras. Istället skickar huvudprogrammet adresserna till x, y och z som funktionen slumpa_tal tar emot som pekare. Genom att avreferera dessa pekare eller adresser med exempelvis *xp som då är detsamma som variabeln x, kan funktionen slumpa värden till x, y resp z.

Funktionen med formella parametrar i form av *pekare*:

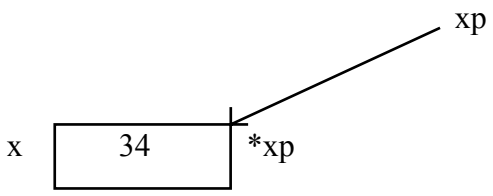
```
void slumpa_tal(int *xp, int *yp, int *zp)
{
   srand((unsigned)time(NULL));
   *xp = rand()%100;
   *yp = rand()%100;
   *zp = rand()%100;
}
```

Anropet med aktuella parametrar i form av *adresser*:

```
slumpa_tal(&x, &y, &z);
```

Adresserna för x, y och z skickas till slumpa_tal. Via dessa adresser kan slumpa_tal slumpa värden till variablerna x, y resp z.

Vi kan illustrera hur det ser ut för variabeln x enligt:



Vid anropet får pekaren xp ett värde som är adressen för x d.v.s xp kommer att peka på det utrymme i minnet som heter x. Via xp kommer man åt detta utrymme i minnet som *xp, som man slumpar ett värde till exempelvis 34.

Informationsflödet för hela programmet :

```
void main()
{
   int x, y, z, max;

   slumpa_tal(&x, &y, &z);
   max = berakna_max(x, y, z);
   skriv_max(max);
}
```

fungerar nu så att slumpa_tal slumpar värden till x, y och z. Dessa värden skickas vidare till berakna_max som returnerar det största talet vilket tilldelas max. Slutligen skickas detta största värde till skriv_max som skriver ut det.

Informationsöverföringen är nu gjord på ett mera generellt och säkert sätt än med globala variabler. *De program som exempelvis vill använda funktionen berakna_max behöver bara se funktionens prototyp (huvud) för att använda funktionen*. Resten av funktionen är ointressant och kan ses som en svart låda (black box).

```
int berakna_max(int xf, int yf, int zf)
/* returnerar det största av talen xf, yf och zf */
{
/*************************************/
/* Denna del ointressant för användare */
/*************************************/
}
```

Exempel på anrop:

a)

```
max = berakna_max(56, 78, 34);
```

b)

```
int a, b, c;

printf("Ge 3 heltal : ");
scanf("%d%d%d", &a, &b, &c);
if (  berakna_max(a, b, c) > 100 )
{
   ....
```

c)

```
max = berakna_max(12, a);   /* Fel! För få parametrar */
```

Vill man ändra på ett parametervärde måste man i C skicka den aktuella adressen och ta emot med en formell pekare som sedan avrefereras i funktionen.

Ex: Skriv en funktion som omvandlar en liten bokstav till en motsvarande stor.

```
char stor(char liten)
/* Om liten bokstav returneras stor */
{
   char ch;

   if (liten >= 'a' && liten <='z')
      ch = liten - 32;
   else
      ch = liten;

   return ch;
}


#include <stdio.h>

void main()
{
   char liten_bokstav, stor_bokstav;

   printf ("Ge en liten bokstav : ");
   liten_bokstav = getchar();

   stor_bokstav = stor(liten_bokstav);
   printf("Motsvarande stora bokstav : %c", stor_bokstav);
}
```

I exemplet ovan sker informationsöverföringen om vilken liten bokstav som ska omvandlas med hjälp av en parameter. Vid anropet kopieras värdet av den aktuella parametern liten_bokstav till motsvarande formella parameter liten.

Informationen tillbaka från funktionen sker med hjälp av funktionsvärdet. I funktionen tar man fram motsvarande stora bokstav och returnerar den som funktionsvärdet stor.

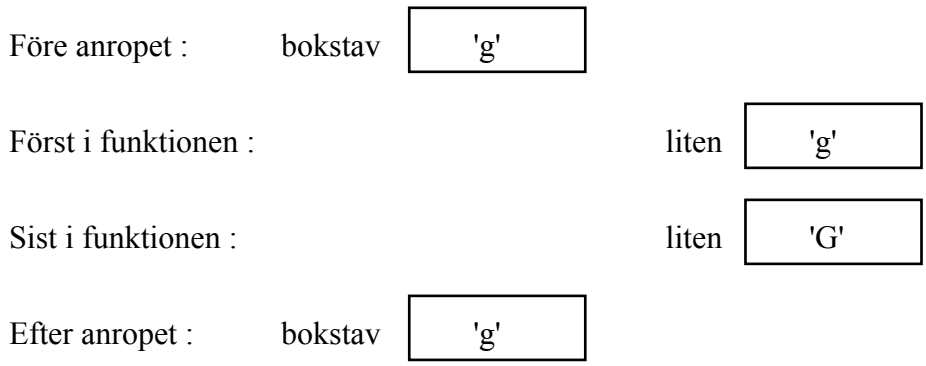
Ovanstående sätt att ordna informationsöverföringen är det absolut bästa när man ska returnera ett enda värde. Man får *ej heller några bieffekter* på den aktuella parametern liten_bokstav som bibehåller sitt värde, vilket ibland är önskvärt.

Man kan också tänka sig en funktion som ej returnerar något värde utan omvandlar den aktuella parametern direkt istället.

Om man försöker med nedanstående funktion *fungerar den inte*.

```
void stor(char liten)
/* Om liten bokstav omvandlas den till stor */
{
   char ch;

   if (liten >= 'a' && liten <='z')
      ch = liten - 32;
   else
      ch = liten;

   liten = ch;
}


#include <stdio.h>

void main()
{
   char bokstav;

   printf ("Ge en liten bokstav : ");
   bokstav = getchar();

   stor(bokstav);
   printf("Motsvarande stora bokstav : %c", bokstav);
}
```

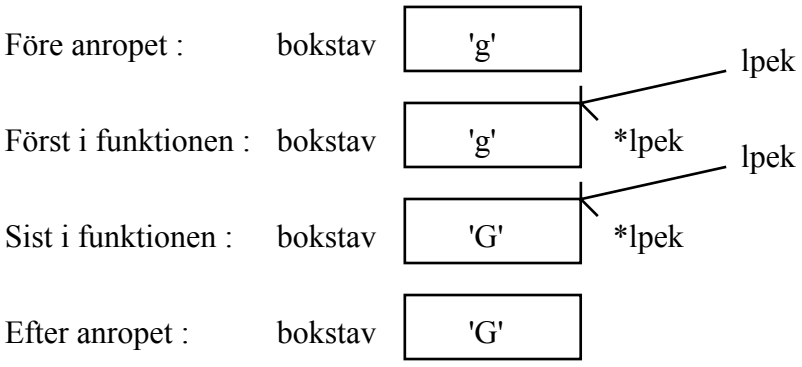
Programmet gör ingen omvandling utan skriver alltid ut samma lilla bokstav. Detta beror på att värdet av den aktuella parametern, bokstav, visserligen kopieras till den formella parametern, liten, som i funktionen omvandlas till en stor bokstav, men aldrig kopieras tillbaka till bokstav.



Ska man få tillbaka den nya informationen om stor bokstav via parametern, måste man skicka adressen till bokstav som aktuell parameter och ta emot den med en pekare som formell parameter enligt:

```
void stor(char *lpek)
/* Om liten bokstav omvandlas den till stor */
{
   char ch;

   if (*lpek >= 'a' && *lpek <='z')
      ch = *lpek - 32;
   else
      ch = *lpek;

   *lpek = ch;
}


#include <stdio.h>

void main()
{
   char bokstav;

   printf ("Ge en liten bokstav : ");
   bokstav = getchar();

   stor(&bokstav);
   printf("Motsvarande stora bokstav : %c", bokstav);
}
```

Genom att skicka över adressen för bokstav till funktionen kan man från funktionen via lpek komma åt den lilla bokstaven och ändra den till motsvarande stora enligt:



Man kan även ha sammansatta typer som parametrar eller funktionsvärden. När det gäller vektorer så är själva vektor-namnet egentligen en adress. *När man har en vektor som aktuell parameter skickar man alltså en adress och måste ta emot den med en formell parameter i form av en pekare.*

Ex: Skriv ett program som slumpar en heltalsvektor med ett inläst antal tresiffriga tal, skriver ut vektorn, sorterar vektorn och slutligen skriver ut den igen.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>


void slumpa(int v[], int nr)
{
   int i;

   srand((unsigned)time(NULL));
   for (i = 0; i < nr; i++)
      v[i] = rand()%900 + 100;
}


void skriv(int v[], int nr)
{
   int i;

   for (i = 0; i < nr; i++)
      printf("%4d", v[i]);
}

void sortera(int v[], int nr)
{
   int i, j, minind, temp;

   for (i = 0; i < nr - 1; i++)
   {
      minind = i;
      for (j = i + 1; j < nr; j++)
      {
         if (v[j] < v[minind])
            minind = j;
      }
      temp = v[i];
      v[i] = v[minind];
      v[minind] = temp;
   }
}
```

```
void main()
{
   int vek[1000], antal;

   printf("\nHur många tal? (Max 1000!) : ");
   scanf("%d", &antal);
   slumpa(vek, antal);
   printf("\nSlumpad vektor\n");
   skriv(vek, antal);
   sortera(vek, antal);
   printf("\n\nSorterad vektor\n");
   skriv(vek, antal);
}
```

När man använder formella parametrar som ska anropas av aktuella parametrar i form av vektorer brukar man använda en *tom hakparentes* för att markera att den ska anropas med en vektor. Man kan skriva antal element inuti hakparentesen men detta antal är helt betydelselöst.

Man kan också använda enbart en pekare som formell parameter och programmet fungerar lika bra för det. Tittar man på funktionen slumpa ovan enligt:

```
void slumpa(int v[], int nr)
{
   int i;

   srand((unsigned)time(NULL));
   for (i = 0; i < nr; i++)
      v[i] = rand()%900 + 100;
}
```

så kan den lika gärna skrivas enligt:

```
void slumpa(int *v, int nr)
{
   int i;

   srand((unsigned)time(NULL));
   for (i = 0; i < nr; i++)
      v[i] = rand()%900 + 100;
}
```

Indexeringen med hakparenteser fungerar eftersom det egentligen är pekaruppräkning.

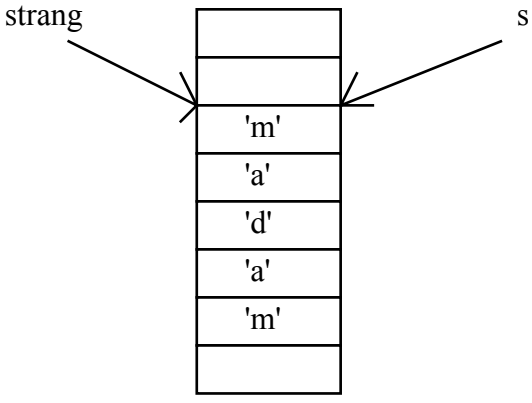
När det gäller strängar, alltså vektorer av tecken, gäller samma sak som för vektorer. Här använder man dock nästan alltid en * istället för tomma hakparenteser [] som formell parameter.

Ex: Skriv ett program som läser in en sträng och kontrollerar om strängen är ett palindrom d.v.s ser likadan ut både fram och baklänges.

```
#include <string.h>   /* strlen */
#include <stdio.h>

int palin(char *s)
{
   int i, j;

   for (i = 0, j = strlen(s) - 1; i < j; i++, j--)
      if (s[i] != s[j])
         return 0;          /* OBS! Uthopp */
   return 1;
}

void main()
{
   char strang[80];

   printf("\nGe en sträng : ");
   gets(strang);
   if ( palin(strang) )
      printf("\nSträngen är ett palindrom!\n");
   else
      printf("\nSträngen är ej ett palindrom!");
}
```

Här har man i funktionen palin, som kontrollerar om palindrom, en formell parameter som är pekare till char. Man kunde lika gärna ha skrivit char s[] och programmet hade fungerat precis på samma sätt.

Tittar man närmare på parameteröverföringen, då man har vektorer eller strängar som parametrar, ser det ut enligt nedanstående skiss som gäller för exemplet med palindrom.



För strängar finns det ett stort antal färdiga funktioner att använda i string.h. Ovan användes funktionen strlen för att bestämma strängens längd. I string.h finns även funktioner för att kopiera en sträng till en annan (strcpy), lägga ihop två strängar (strcat), jämföra två strängar (strcmp) etc.

Ex: Skriv ett program som läser in två namn och i en funktion byter plats mellan strängarna som sedan skrivs ut.

```
#include <stdio.h>
#include <string.h>   /* strcpy */

void byt(char *s1, char *s2)
{
   char temp[80];

   strcpy(temp, s1);
   strcpy(s1, s2);
   strcpy(s2, temp);
}

void main()
{
   char a_namn[80], b_namn[80];

   printf("Ge ett namn : ");
   gets(a_namn);

   printf("Ge ett till namn : ");
   gets(b_namn);

   byt(a_namn, b_namn);

   puts(a_namn);
   puts(b_namn);
}
```

OBS! Man kan *inte använda tilldelning mellan strängar* när man ska kopiera en sträng till en annan sträng. Man måste kopiera tecken för tecken eller som här använda den färdiga funktionen strcpy som finns i string.h.

OBS! *Funktionen strcpy kopierar den andra parametern till den första. Man måste se till att det finns plats (minne) i den första strängen för den andra strängen.*

Ex: Skriv ett program som sorterar en vektor av ett antal initierade namn.

```
#include <stdio.h>
#include <string.h>        /* strcpy, strcmp */

void skriv(char *v[], int nr)
{
   int i;

   for (i = 0; i < nr; i++)
      printf("%s\n", v[i]);

}

void sortera_namn(char *v[], int nr)
{
   int i, j, minind;
   char *temp;

   for (i = 0; i < nr - 1; i++)
   {
      minind = i;
      for (j = i + 1; j < nr; j++)
      {
         if ( strcmp(v[j], v[minind]) < 0)
            minind = j;
      }
      temp = v[i];
      v[i] = v[minind];
      v[minind] = temp;
   }
}

void main()
{
   char *namn[10] = {"Karlsson Bert", "Olsson Vera",
                     "Persson Gulli", "Andersson Per",
                     "Larsson Kurt", "Persson Kurt",
                     "Månsson John", "Nilsson Gun",
                     "Jonsson Gun", "Larsson Lena" };

   putchar('\n');
   skriv(namn, 10);
   sortera_namn(namn, 10);
   putchar('\n');
   skriv(namn, 10);
}
```

OBS! Användandet av funktionen strcmp som jämför strängtecknens ASCII-värden tecken för tecken från vänster tll höger och returnerar -1 så fort vänster sträng har ett tecken med mindre ASCII-värde än höger, 0 om alla tecken är lika och
1 så fort något tecken har större värde.

OBS! Hur man har en vektor av strängar som man sorterar genom att ställa om pekarna i vektorn. Strängarna finns kvar i samma minnesutrymmen hela tiden.

När man har poster som parametrar ska man hantera dessa på samma sätt som vanliga enkla variabler.

Ex: Skriv ett program som läser in data till två poster som ska avbilda medlemmar med nummer och namn och som byter plats mellan posterna och sedan skriver ut dessa.

```
#include <stdio.h>
#include <conio.h>

struct medlem
{
   int nr;
   char namn[30];
};

void skriv(struct medlem m)
{
   printf("Nummer : %d\n", m.nr);
   printf("Namn   : %s\n", m.namn);
}

void las(struct medlem *mp)
{
   printf("Ge nummer : ");
   scanf("%d", &mp->nr);         /* samma som &(*mp).nr */
   getchar();                    /* rensa */
   printf("Ge namn : ");
   gets(mp->namn);               /* samma som (*mp).namn */
}

void byt(struct medlem *mp1, struct medlem *mp2)
{
   struct medlem temp;

   temp = *mp1;
   *mp1 = *mp2;
   *mp2 = temp;
}

void main()
{
   struct medlem med1, med2;

   las(&med1);
   las(&med2);

   byt(&med1, &med2)

   skriv(med1);
   skriv(med2);
}
```

OBS! *När man avrefererar en post-pekare kan man använda en förkortad syntax med*

*-> istället för *.*

5.4 Macron

I vissa fall kan man istället för att använda en funktion använda ett macro. Ett macro-namn ersätts av förkompilatorn med det definierade macro-värdet. Man kan, som visats tidigare, använda macron för att exempelvis definiera konstanter.

Ex: Skriv ett macro som definierar MINUS_ETT som (-1).

```
#define MINUS_ETT (-1)

....
ett = MINUS_ETT*MINUS_ETT;
```

Efter förkompileringen är ovanstående sats expanderad till

```
ett = (-1)*(-1);
```

som sedan kompileras av den egentliga C-kompilatorn.

OBS! Man brukar använda STORA BOKSTÄVER för macron.

OBS! Allt som kommer efter första blanktecknet som följer efter macro-namnet kommer att ingå i macro-värdet.

Har man korta små funktioner kan det ibland vara motiverat att använda macron istället för funktioner, eftersom ett funktionsanrop tar tid att göra. Macron kan precis som funktioner återanvändas och koden blir lättare att läsa.

Ex: Skriv ett macro som definierar macro_namnet SKIPLINE för att rensa inmatningsbufferten fram t.o.m närmaste RETURN-tecken.

```
#define SKIPLINE while (getchar() != '\n')

....
SKIPLINE;
```

som efter expanderingen kommer att se ut som

```
while (getchar() != '\n');
```

Macron kan precis som funktioner ha parametrar:

Ex: Definera ett macro för kvadrering av ett tal.

```
#define KVADRAT(n) ((n)*(n))

....
a_tal = KVADRAT(5);
```

expanderas vid förkompileringen till

```
a_tal = ((5)*(5));
```

OBS! Man ska vara försiktig vid användningen av macron med parametrar. Har man sidoeffekter på dessa blir det helt fel!

```
b_tal = KVADRAT(i++);
```

expanderas till

```
b_tal = ((i++)*(i++));
```

som kanske inte ger det resultat man tänkt sig.

Ett macro kan ersätta funktioner. Många standarfunktioner som finns i header-filer som stdio.h, stdlib.h, är implementerade som macron, istället för funktioner. Man ska dock undvika att ha för långa macron och kontrollera att sidoeffekter inte kan ge fel resultat. I dessa fall är det bättre att använda en funktion istället.

Header-filer som stdio.h innehåller en mängd färdiga typer, macron och funktioner som man kan utnyttja sig av i sitt program. När det gäller funktioner finns oftast inte hela funktionen med utan bara *funktionens deklaration eller prototyp* som består av funktionshuvudet följt av ett semikolon.

Ex: Funktionen pow i math.h har exempelvis deklarationen eller prototypen:

```
double pow(double x, double y);
```

I deklarationen kan namnet på de formella parametrarna , x och y ovan, utelämnas enligt:

```
double pow(double, double);
```

Deklarationen används av kompilatorn för att kontrollera att anropet är korrekt och att korrekt kod läggs ut. Själva funktionsdefinitionen eller funktionskroppen tas sedan in som objektkod vid länkningen. Man kan även tillverka egna headerfiler, som inkluderas med #include "fil.h".

6 Lagring av data på fil

Vid inmatning av ett fåtal data till ett program används tangentbordet. Har man större datamängder som ska matas in brukar man istället ha dessa på en *fil i sekundärminnet* och låta programmet hämta data direkt från filen. Samma sak gäller även vid utmatning. Normalt sker all utmatning till skärmen. Vid stora utdatamängder eller då utdata ska sparas för att användas vid ett senare tillfälle måste man från sitt program kunna skriva ut data på en *fil i sekundärminnet*.

Ett typiskt exempel på ovanstående är ett program som ska ta reda på alla bilar som man ska skicka ut besked om kontrollbesiktning till. Bilarna finns i ett register på en fil i sekundärminnet. Programmet måste kunna läsa data från denna fil, kontrollera om bilen ska besiktigas och skriva ut information om bilen på en annan fil eller alternativt mata ut brev som kallar till kontrollbesiktning.

För att från ett program kunna hämta data från eller skicka data till filer i sekundärminnet finns i alla programmeringsspråk möjligheter att skapa kommunikationskanaler eller buffertar som sköter om transporten av data mellan primärminnet och sekundärminnet. *I C kallas dessa kanaler för strömmar.*

6.1 Strömmar och filer

Med en *ström* i C ska man kunna transportera data mellan sitt program och en *fil* i sekundärminnet. I header-filen stdio.h finns en fördeklarerad strömtyp FILE för detta.

Ex: Definiera en inström för att läsa data från filen indata.txt och en utström för att skriva data på filen utdata.txt.

```
#include <stdio.h>

/* definiera pekare till strömmarna */
FILE *instrom, *utstrom;

/* skapa och öppna strömmarna */
instrom = fopen("indata.txt","r");    /* r som i read */
utstrom = fopen("utdata.txt","w");    /* w som i write */

/* läs data från filen indata.txt via instrom */

/* skriv data till filen utdata.txt via utstrom */

/* töm och stäng strömmarna */
fclose(instrom);
fclose(utstrom);
```

I sitt program skapar man en ström genom att först definiera en pekare till datatypen FILE, som finns i stdio.h och sedan öppna strömmen eller kanalen till filen med fopen. I fopen anger man med den *första strängparametern* vilken fil i sekundärminnet som strömmen ska öppnas till. Om det av någon anledning ej gick att öppna strömmen, *returnerar fopen pekarvärdet NULL*. Detta kan exempelvis inträffa ovan, om inte filen indata.txt finns i sekundärminnet.

Med den *andra strängparametern* i anropet av fopen anger man vilken *typ av ström* som ska skapas. Man kan öppna *textströmmar* eller *binärströmmar* och man kan ange om man ska läsa eller skriva data via dessa strömmar Den andra parametern kan ha värden exempelvis enligt:

r	----	Textström för enbart läsning. Filen måste finnas.
w	----	Textström för enbart skrivning. Om filen finns skrivs den över. Om filen ej finns skapas en ny fil.
a	----	Textström för enbart skrivning. Om filen finns lägger man till data på slutet. Om filen ej finns skapas en ny fil.
r+	----	Textström för både läsning och skrivning. Filen måste finnas.
w+	----	Textström för både skrivning och läsning. Om filen finns skrivs den över. Om filen ej finns skapas en ny fil.
a+	----	Samma som w+ men man lägger till i slutet om filen finns.

I C skiljer man på *textströmmar* och *binärströmmar.* En textström omvandlar vid skrivning data, som finns i binär form i primärminnet, till läsbara tecken exempelvis i ASCII-format på filen i sekundärminnet och omvänt från ASCII-form i sekundärminnet till binär form i primärminnet vid läsning. Filerna i sekundärminnet som på detta sätt blir ASCII-kodade kallas för *textfiler* och kan exempelvis ändras med en editor och skrivas ut på en printer. Textfiler behövs för att kommunikationen mellan *människa och dator* ska fungera.

Binärströmmar gör ingen omvandling utan de transporterar bara data i binärform mellan programmet i primärminnet och filen i sekundärminnet och tvärtom. Alla strömmar i C blir *default textströmmar*. Vill man ha en binärström måste man ange detta explicit i strängen för strömtypen med ett *b* enligt exempelvis:

r+b	----	Binärström för både läsning och skrivning. Filen måste finnas.
wb	----	Binärström för skrivning. Om filen finns skrivs den över. Om filen ej finns skapas en ny fil.

6.2 Textströmmar

När ett program ska kommunicera med en människa måste in- och utdata vara i sådan form att människan förstår. Programmet använder internt ett speciellt binärt format för exempelvis det reella talet 3.4. Ska ett program skriva ut ett reellt tal på en fil i sekundärminnet så att en människa kan förstå vad filen innehåller, måste det binära formatet omvandlas till tre ASCII-kodade tecken i form av 3.4.

Ex: Skriv ett program som slumpar 1000 tärningskast som skrivs ut på en textfil utfall.txt med 25 utfall per rad.

```
#include <stdio.h>    /* FILE, fopen, fprintf, fclose */
#include <stdlib.h>
#include <time.h>

void main()
{
   FILE *filout;
   int i, utfall;

   filout = fopen("utfall.txt", "w");

   srand((unsigned)time(NULL));
   for (i = 1; i <= 1000; i++)
   {
      utfall = rand()%6 + 1;
      fprintf(filout, "%2d", utfall);
      if (i % 25 == 0)
         fprintf(filout, "\n");
   }

   fclose(filout);
}
```

OBS! Funktionen *fprintf* skriver ut på den ström som anges som första parameter. Formatsträngens omvandlingsspecifikation anger hur det slumpade tärningsutfallets ettor och nollor ska tolkas och omvandlas innan det skrivs ut som ASCII-kodade tecken på textfilen utfall.txt via strömmen filout. Eftersom man använder en textström blir utfall.txt en textfil, som kan *skrivas ut på en skrivare eller visas på en skärm. Textfiler brukar markeras med filtypen .txt.*

OBS! Funktionen fprintf fungerar på samma sätt som den vanliga utskriftsfunktionen printf. Skillnaden är att med *fprintf kan man själv välja vilken ström man ska skriva ut på*. Funktionen printf däremot skriver alltid ut på samma fördefinierade ström, *stdout*, som default är ihopkopplad med skärmen.

Ex: Skriv ett program som läser de slumpade tärningsutfallen från textfilen utfall.txt och på skärmen skriver ut utfallens medelvärde.

```
#include <stdio.h>    /* FILE, fopen, fscanf, fclose */

void main()
{
   FILE *filin;
   int i, utfall, sum = 0;

   filin = fopen("utfall.txt", "r");
   if (filin != NULL)
   {
      for (i = 1; i <= 1000; i++)
      {
         fscanf(filin, "%d", &utfall);
         sum += utfall;
      }

      fclose(filin);

      printf("Medelvärde : %.2f\n", (float)sum/1000);

   }
   else
      printf("Finns ingen fil att läsa ifrån!\n");
}
```

OBS! Funktionen *fscanf* läser från den ström som anges som första parameter. Formatsträngens omvandlingsspecifikation anger hur ASCII-koderna i textfilen utfall.txt ska tolkas och hur omvandlingen till binärform ska ske innan de instoppas på adressen för variabeln utfall.

OBS! Filen läses i *fritt format* med utnyttjande av separatortecknen blank, tab och nyrad, precis som vid läsning med scanf från tangentbordet. Skillnaden mellan funktionen fscanf och scanf är att fscanf kan utnyttja en egendefinierad ström kopplad till vilken textfil som helst, medan *scanf* alltid läser från den fördefinierade strömmen *stdin*, som default är ihopkopplad med tangentbordet.

I ovanstående exempel visste man hur många tal som fanns i filen och man kunde anpassa läsningen efter detta. *Vad ska man göra om man inte vet hur många tal som finns i en fil,* som man ska läsa data från? Det finns olika metoder. När man läser från tangentbordet kan man exempelvis avsluta inläsningen med talet 0. Detta kan man naturligtvis också göra vid läsning från fil om man ser till att ha 0 som sista tal i filen.

Ex. Skriv ett program som läser alla reella tal ifrån textfilen retal.txt och skriver ut talens medelvärde. Tittar man på filen retal.txt på skärmen ser den ut som:

3.45 3.67 3.56
3.34 3.48
3.56 3.61 3.24 3.56
0.0

```
#include <stdio.h>

void main()
{
   FILE *filin;
   float x, sum = 0;
   int nr = 0;

   filin = fopen("retal.txt", "r");
   if (filin != NULL)
   {

      fscanf(filin, "%f", &x);
      while(x != 0.0)
      {
         sum += x;
         nr++;
         fscanf(filin, "%f", &x);
      }

      fclose(filin);
      if (nr > 0)
         printf("Medelvärde : %.2f\n", sum/nr);
      else
         printf("Antalet inlästa tal är 0!\n");
   }
   else
      printf("Finns ingen fil att läsa ifrån!\n");
}
```

OBS! Funktionen fscanf hoppar först över alla separatorer i form av blanka, tabbar och nyradtecken och läser därefter tecken *så länge det passar ihop* med ett reellt tal.

OBS! Textfilen retal.txt kan vara gjord av ett annat program eller med ett editorprogram.

Man behöver ej själv markera filslut, som man gjorde ovan med 0.0. *Varje operativsystem har en egen markering för filslut*. I ett C-program ska *fscanf-funktionen, som normalt returnerar antalet lyckade läsningar, returnera värdet -1* då den har läst filslutstecknet i systemet. Konstanten EOF med värdet -1 finns som macro i stdio.h

Ex: Skriv ett program som letar upp och skriver ut det största och minsta talet i textfilen heltal.txt. I filen finns alltid minst ett tal enligt:

234 123 345 236 678 456
567 891 234 567
125 567 234 456 234
234 567

```
#include <stdio.h>
void main()
{
   FILE *filin;
   int tal, min, max, status;

   filin = fopen("heltal.txt", "r");
   if (filin != NULL)
   {
      fscanf(filin, "%d", &tal);
      min = max = tal;

      status = fscanf(filin, "%d", &tal);
      while(status != EOF)
      {
         if ( tal < min )
            min = tal;
         else if (tal > max)
            max = tal;
         status = fscanf(filin, "%d", &tal);
      }

      fclose(filin);

      printf("Minsta tal : %d\n", min);
      printf("Största tal: %d\n", max);
   }
   else
      printf("Finns ingen fil att läsa ifrån!\n");
}
```

OBS! Funktionen fscanf returnerar ett heltal som är lika med antalet lyckligt inlästa tal eller EOF (-1), då filslut nåtts. Detta gäller även scanf-funktionen som läser från tangentbordet. Hur operativsystemets tecken för filslut ser ut kan variera. I DOS är det CTRL/Z och i UNIX är det CTRL/D.

OBS! Istället för att ovan använda en extra variabel för att kontrollera status på fscanf-funktionen hade man kunnat skriva läsningen förkortat som:

```
while(fscanf(filin, "%d", &tal) != EOF)
{
   if ( tal < min )
      min = tal;
   else if (tal > max)
      max = tal;
}
```

I fil-inmatningsfunktionen fscanf och fil-utmatningsfunktionen fprintf kan man, på samma sätt som i vanliga scanf och printf, använda andra omvandlingsspecifikationer, som exempelvis *%c, för att läsa/skriva tecken* och *%s för att läsa/skriva strängar*. Dessa har dock som framgått tidigare vissa nackdelar åtminstone för strängar. Därför finns det speciella funktioner för in- och utmatning av tecken och strängar.

För tecken heter funktionerna *fgetc och fputc*. Dessa ska motsvara getchar och putchar som enbart gäller för stdin (tangentbord) resp stdout (skärm).

Ex: Skriv ett program som kopierar en textfil tecken för tecken till en annan fil. Namnet på filerna ska läsas in från tangentbordet.

```
#include <stdio.h>

void main()
{
   FILE *fin, *fout;
   char infilnamn[30], utfilnamn[30], ch;

   printf("Ge infilnamn : ");
   gets(infilnamn);
   fin = fopen(infilnamn, "r");

   /* här borde kontrolleras att filen finns */

   printf("Ge utfilnamn : ");
   gets(utfilnamn);
   fout = fopen(utfilnamn, "w");

   ch = fgetc(fin);
   while ( ch != EOF )
   {
      fputc(ch, fout);
      ch = fgetc(fin);
   }

   fclose(fin);
   fclose(fout);
}
```

OBS! Även nyradtecken läses från infilen och kopieras till utfilen vilket innebär att raderna bibehålls.

OBS! Eftersom ett tilldelningsuttryck får samma värde som det tilldelade värdet kan ovanstående kod även skrivas:

```
while ( (ch = fgetc(fin)) != EOF )
{
    fputc(ch, fout);
}
```

Ex: Skriv ett program som läser textfilen matdata.txt och beräknar och på skärmen skriver ut summan av varje rad och en totalsumma. Filen ser ut som:

23.45 34.56 12.34
56.78 23.56
12.45 45.67 13.56 25.67

```
#include <stdio.h>

void main()
{
   FILE *rfil;
   float x, radsum = 0.0, totalsum = 0.0;
   char ch, rfilnamn[20];

   rfil = fopen("matdata.txt", "r");
   while (rfil == NULL)
   {
      printf("Finns ingen sådan fil!\n");
      printf("Ge nytt filnamn : ");
      gets(rfilnamn);
      rfil = fopen(rfilnamn, "r");
   }

   /*  läs tal från fil och summera */
   while ( fscanf(rfil, "%f", &x) != EOF )
   {
      radsum += x;

      /* kontrollera nästa tecken */
      ch = fgetc(rfil);
      ungetc(ch, rfil);

      if ( ch == '\n' || ch == EOF)
      {
         /* ny rad */
         printf("%f\n", radsum);
         totalsum += radsum;
         radsum = 0;
       }
   }
   fclose(rfil);
   printf("%f\n", totalsum);
}
```

OBS! Med fgetc läses nästa tecken i filen och med ungetc flyttas filpekaren tillbaka så att *aktuell filposition ej ändras.* Man kan fortsätta att läsa nästa tal som vanligt. I ovanstående exempel är användandet av ungetc egentligen onödigt eftersom fgetc bara läser blanktecken, radslutstecken eller filslutstecken, som ej har någon inverkan på de reella talens storlek.

För strängar finns funktionerna *fgets och fputs* som läser resp skriver strängar på valfria textströmmar. Dessa motsvarar funktionerna gets och puts som gör motsvarande på stdin (tangentbord) resp stdout (skärm).

Ex: Skriv ett program som läser textfilen medlem.txt som innehåller namn, telefonnummer och medlemsavgift enligt:

Ulla Karlsson
019/111111
300
Per Jonsson
0586/222222
0
....

och som för alla medlemmar som har medlemsavgiften 0 skriver ut namn och telefonnummer på en ny fil ejbetalt.txt.

```
#include <stdio.h>

void main()
{
   char namn[30], tel[20];
   int avgift;
   FILE *infil, *utfil;

   infil = fopen("medlem.txt", "r");
   utfil = fopen("ejbetalt.txt", "w");
   while ( fgets(namn, 30, infil) != NULL)
   {
      fgets(tel, 20, infil);
      fscanf(infil, "%d", &avgift);
      fgetc(infil);        /* förbi radslut */
      if ( avgift == 0)
      {
         /* ej betalt */
         fputs(namn, utfil);
         fputs(tel, utfil);
      }
   }
   fclose(infil);
   fclose(utfil);
}
```

OBS! Funktionen fgets har tre parametrar där den i mitten anger det maximala antal tecken som ryms i strängen, som man ska läsa till. Ovan *läser funktionen maximalt 29 tecken till strängen namn*, eftersom det ska finnas plats för ett avslutande '\0'-tecken. Läsningen avslutas dock alltid då *ett radslutstecken* lästs. *Radslutstecknet ingår i den inlästa strängen* till skillnad från gets-funktionen.

OBS! Funktionen fgets *returnerar NULL* då det inte finns någon sträng att läsa, exempelvis vid filslut.

När man definierar egna strömmar kan man koppla *dessa från sitt program till vilken fil som helst i sekundärminnet*. Man kan också koppla strömmarna till andra *externa enheter* som printrar, skärm, tangentbord, portar etc. I C-program kommer man åt dessa enheter genom att använda de fördefinierade filnamn, som dessa har i det aktuella operativsystemet. I DOS har exempelvis printern i den första parallellporten filnamnet lpt1, skärm och tangentbord filnamnet con etc.

Ex: Skriv om ovanstående program med medlemmarna så att utskriften kan väljas att fås på skärm eller printer. Anger man inget kommer utskriften som vanligt på filen ejbetalt.txt.

```
#include <stdio.h>

void main()
{
   FILE *infil, *utfil;
   char namn[30], tel[30], svar;
   int avgift;

   infil = fopen("medlem.txt", "r");

   /* fråga efter utskriften */
   printf("Utskrift på printer/skärm (p/s) : ");
   svar = getchar();
   if ( svar == 'P' || svar == 'p' )
      utfil = fopen("lpt1", "w");
   else if ( svar == 'S' || svar == 's' )
   {
      clrscr();
      utfil = fopen("con", "w");
   }
   else
      utfil = fopen("ejbetalt.txt", "w");

   while ( fgets(namn, 30, infil) != NULL )
   {
      fgets(tel, 20, infil);
      fscanf(infil, "%d", &avgift);
      fgetc(infil);
      if ( avgift == 0)
      {
         /* ej betalt */
         fputs(namn, utfil);
         fputs(tel, utfil);
      }
   }
   fclose(infil);
   fclose(utfil);
}
```

6.3 Binärströmmar

Binärströmmar används då man från sitt program vill spara undan data på en binärfil i sekundärminnet och det *inte finns något behov av att skriva ut denna data på skärm eller skrivare*. På detta sätt sparar man både programtid och minne. Exempelvis tar talet 23456 5 Byte (5 tecken) i en textfil och 2 Byte (typen short int) i en binärfil. Då man använder en binärfil behövs ej heller någon omvandling från binär form i primärminnet till ASCII-kod i sekundärminnet. Strömmen skyfflar bara över byten precis som de ser ut i primärminnet.

Ex: Skriv ett program som läser in mätdata i form av reella tal från tangentbordet och skriver in dessa i en binärfil, mat.dat. Inläsningen av data avslutas med ett CTRL/Z från tangentbordet vilket innebär EOF i stdin.

```
#include <stdio.h>

void main()
{
   FILE *bfout;
   float x;

   bfout = fopen("mat.dat", "wb");

   /* läs data från tangentbordet */
   printf("Ge mätdata. Avsluta med EOF(CTRL/Z)!\n");
   while ( scanf("%f", &x) != EOF )
   {
      /* skriv till fil */
       fwrite(&x, sizeof(float), 1, bfout);
   }

   fclose(bfout);
}
```

OBS! Filen öppnas med filtypen *wb*. Detta innebär att man öppnar en binärfil för enbart skrivning.

OBS! Funktionen *fwrite* skriver från den adress i primärminnet som anges av den första parametern de antal byte som anges av produkten av den andra och den tredje parametern till den ström som anges av den fjärde parametern.

OBS! Filen mat.*dat* är en binärfil som ej kan skrivas ut på printer eller skärm utan den innehåller data i samma binära form som variablerna i primärminnet. Med filtypen *.dat* markeras i forsättningen att det är en binärfil till motsats mot *.txt* för textfiler.

OBS! Operatorn *sizeof* som ger storleken i byte av en datatyp eller variabel.

Binära filer innehåller binärkodad information. Detta innebär att om man ska kunna tolka data i en binärfil måste man skriva ett program som läser filen.

Ex: Skriv ett program som läser binärfilen mat.dat och beräknar och skriver ut medelvärdet av all data. Filnamnet ska inläsas.

```
#include <stdio.h>

void main()
{
   FILE *bfin;
   char filnamn[20];
   float x, sum = 0.0;
   int nr = 0;

   /* läs in filnamn och öppna fil */
   do
   {
      printf("Ge filnamn : ");
      gets(filnamn);
      bfin = fopen(filnamn, "rb");
      if (bfin == NULL)
         printf("Filen finns ej!\n");
   }
   while (bfin == NULL);

   /* läs från fil */
   fread(&x, sizeof(float), 1, bfin);
   while ( !feof(bfin) )
   {
      sum += x;
      nr++;
      fread(&x, sizeof(float), 1, bfin);
   }

   fclose(bfin);

   /* skriv ut medelvärde */
   if ( nr > 0 )
      printf("Medel : %f", sum / nr;
   else
      printf("Antalet data är 0!\n");
}
```

OBS! Funktionen *fread* läser till adressen som anges av den första parametern, så många byte data som anges av produkten av den andra och den tredje parametern från strömmen som anges av den sista parametern.

OBS! Funktionen *feof* returnerar sant alltså 1 då filslut nåtts i en *binärfil*. För textfiler är det säkrare att testa om inläsningsfunktionens returvärde är EOF.

Funktionerna fwrite och fread *skyfflar byte* mellan primärminne och sekundärminne och tvärtom. Dessa funktioner är ej begränsade till att endast användas för binära strömmar utan kan naturligtvis också användas även för textfiler om inga konverteringar behöver göras. Man kan exempelvis utnyttja dessa funktioner till att kopiera alla typer av filer både text- och binärfiler.

Ex: Skriv ett program som kopierar en fil till en annan. Filnamnen ska läsas in och test ska finnas så att infilen finns.

```
#include <stdio.h>
#include <stdlib.h>        /* exit */

void main()
{
   FILE *bfin, *bfout;
   char infilnamn[30], utfilnamn[30], ch;

   printf("Ge infilnamn : ");
   gets(infilnamn);
   bfin = fopen(infilnamn, "rb");
   if ( bfin == NULL)
   {
      /* om filen ej finns */
       printf("Filen finns ej!\n");
       exit(1);
   }

   printf("Ge utfilnamn : ");
   gets(utfilnamn);
   bfout = fopen(utfilnamn, "wb");

   fread(&ch, sizeof(char), 1, bfin);
   while ( !feof(bfin) )
   {
       fwrite(&ch, sizeof(char), 1, bfout);
       fread(&ch, sizeof(char), 1, bfin);
   }

   fclose(bfin);
   fclose(bfout);
}
```

OBS! Hur man, via ett tecken i primärminnet, skyfflar över alla byte, precis som de ser ut, från infilen till utfilen. Detta program fungerar för alla typer av filer.

OBS! Använder man textströmmar och funktioner för dessa vid kopiering fungerar det ej för binärfiler i alla operativsystem. I exempelvis DOS så omvandlar skriv- och läsfunktionerna för textströmmar nyradtecken i primärminnet till ENTER och nyradtecken i sekundärminnnet och tvärtom.

Databaser eller register använder sig av binärfiler. Oftast är det poster, som man arbetar med. En databashanterare för ett bilregister ska exempelvis kunna hantera bilposter som skyfflas mellan primärminne och sekundärminne vid sökning, sortering etc.

Ex: Skriv ett program som läser in bilposter med registreringsnummer och ägare och skriver in dessa i en binärfil som avslutningsvis skrivs ut på skärmen.

```
#include <stdio.h>

void main()
{
   struct bilpost
   {
      char regnr[10];
      char agare[30];
   } bil;

   FILE *bilfil;

   bilfil = fopen("bilar.dat", "w+b");

   /* läs in bilar och skriv på fil */
   printf("\nGe regnr (Avsluta med bara RETURN) : ");
   gets(bil.regnr);
   while (bil.regnr[0] != '\0')
   {
      printf("Ge ägare : ");
      gets(bil.agare);
      fwrite(&bil, sizeof(struct bilpost), 1, bilfil);
      printf("\nGe regnr (Avsluta med bara RETURN) : ");
      gets(bil.regnr);
   }

   /* läs fil och skriv ut bilarna */
   rewind(bilfil);
   fread(&bil, sizeof(struct bilpost), 1, bilfil);
   while ( !feof(bilfil) )
   {
      printf("\nRegnummer : %s\n", bil.regnr);
      printf("Ägare     : %s\n", bil.agare);
      fread(&bil, sizeof(struct bilpost), 1, bilfil);
   }
   fclose(bilfil);
}
```

OBS! Varje skrivning i filen fyller på med de antal byte som sizeof-operatorn ger. När man fyllt på med sista posten står man på filslut. Ska man börja att läsa filen från början måste man med *rewind*-funktionen ställa tillbaka den aktuella filpositionen till filens början.

OBS! När man öppnar en ström med w innebär det att man *suddar ut* eventuell gammal fil, om sådan finns. *Vill man ha kvar informationen i gamla filer och istället fylla på i slutet av filen*, ska man istället öppna strömmen med a, där a står för append (lägga till).

6.4 Filåtkomst eller filaccess

Normalt använder man *sekventiell åtkomst* (access) till filerna. Med sekventiell åtkomst menas att man manipulerar byten på filen i den ordning de står. Ska man läsa filen läser man från filens början till filens slut. Skriver man på filen fyller man på byte på byte. Denna rent sekventiella åtkomst till byten i filen kan man bryta med exempelvis rewind och med append enligt ovanstående exempel.

Det finns även andra möjligheter att *komma åt speciella byte i filen direkt*, så kallad *direkt åtkomst* (access). Man kan tänka sig att man har en filpekare som man flyttar på och placerar på önskad byte. Detta har man exempelvis nytta av då man ska uppdatera poster på en fil. Man läser information från filen, ändrar det och skriver tillbaka informationen i rätt position på filen. Har man ej möjligheter till direkt åtkomst får man kopiera över allt till en slaskfil samtidigt som man utför ändringar på lämpliga ställen och sedan kopiera tillbaka till den ursprungliga filen.

Ex: Skriv en enkel databashanterare för ett lager som innehåller ett antal varor med varubeteckning, pris och antal. Hanteraren ska kunna skapa ny vara, visa vara med viss beteckning och ändra priset på en vara.

```
#include <stdio.h>
#include <string.h>
#define SKIPLINE while ( getchar() != '\n')

struct varupost
{
   char vb[10];
   float pris;
   int antal;
};

void ny_vara(void);
void visa_vara(void);
void nytt_pris(void);

void main()
{
   char svar;

   do
   {
      printf("N    Ny vara\n");
      printf("V    Visa vara\n");
      printf("P    Prisändring\n");
      printf("S    Sluta\n");
      printf("Välj >> ");
      svar = getchar();
      SKIPLINE;
```

```
      switch (svar)
      {
         case 'N':
            ny_vara();
            break;
         case 'V':
            visa_vara();
            getchar();
            SKIPLINE;
            break;
         case 'P':
            nytt_pris();
            break;
         case 'S':;
      }
   }
   while (svar != 'S');
}
```

OBS! Funktionsdeklarationer (prototyper, huvuden) före huvudprogrammet för att filen ska kunna kompileras med funktionsdefinitioner (kroppar) efter huvudprogrammet.

```
void ny_vara(void)
{
   FILE *lfil;
   struct varupost vara;

   /* läs in vara */
   printf("\nGe beteckning : ");
   gets(vara.vb);
   printf("Ge pris : ");
   scanf("%f", &vara.pris);
   printf("Ge antal : ");
   scanf("%d", &vara.antal);

   /* rensa stdin*/
   SKIPLINE;

   /* skriv in varan på fil */
   lfil = fopen("lager.dat", "ab");
   fwrite(&vara, sizeof(struct varupost), 1, lfil);
   fclose(lfil);
}
```

OBS! Funktionen ny_vara öppnar strömmen med *a, som i append,* för att lägga till varan på slutet.

```
void visa_vara(void)
{
   FILE *lfil;
   struct varupost vara;
   char varubet[10];

   /* läs in sökt vara */
   printf("\nGe beteckning: ");
   gets(varubet);

   /* sök efter vara i lager */
   lfil = fopen("lager.dat", "rb");
   fread(&vara, sizeof(struct varupost), 1, lfil);
   while ( !feof(lfil) && strcmp(varubet, vara.vb) != 0)
   {
      fread(&vara, sizeof(struct varupost), 1, lfil);
   }

   if ( !feof(lfil) )
   {
       /* varan finns */
       printf("Beteckning : %s\n", vara.vb);
       printf("Pris : %.2f\n", vara.pris);
       printf("Antal : %d\n", vara.antal);
   }
   else
      printf("Varan finns ej i lagret !\n");
   fclose(lfil);
}
```

OBS! Funktionen strcmp som jämför två strängar och returnerar 0 om strängarna lika. Vill man jämföra utan hänsyn till stora eller små bokstäver måste man använda stricmp som ignorerar typen på tecknen.

```
void nytt_pris(void)
{
   FILE *lfil;
   struct varupost vara;
   char varubet[10];

   /* läs in sökt vara */
   printf("\nGe beteckning: ");
   gets(varubet);

   /* sök efter vara i lager */
   lfil = fopen("lager.dat", "rb+");
   fread(&vara, sizeof(struct varupost), 1, lfil);
   while ( !feof(lfil) && strcmp(varubet, vara.vb) != 0)
   {
      fread(&vara, sizeof(struct varupost), 1, lfil);
   }
```

```
    if ( !feof(lfil) )
    {
        /* varan finns och ser ut som */
        printf("Beteckning : %s\n", vara.vb);
        printf("Pris : %.2f\n", vara.pris);
        printf("Antal : %d\n", vara.antal);

        /* nytt pris */
        printf("Nytt pris : ");
        scanf("%f", &vara.pris);
        /* rensa */
        SKIPLINE;

        /* tillbaka till lagret */
        fseek(lfil, -1*sizeof(struct varupost), SEEK_CUR);
        fwrite(&vara, sizeof(struct varupost), 1, lfil);
    }
    else
       printf("Varan finns ej i lagret !\n");
    fclose(lfil);
}
```

OBS! Funktionen fseek som flyttar till önskad filpositionen. Med den andra parametern anger man offset i förhållande till någon utgångsposition som anges som tredje parameter. Utgångsposition anges med någon av konstanterna:

SEEK_CUR	----	om offset i förhållande till aktuell position
SEEK_SET	----	om offset i förhållande till filens början
SEEK_END	----	om offset i förhållande till filslut.

När man uppdaterar en post i ett register söker man först upp posten sekventiellt d.v.s man startar från filens början och letar tills man har funnit posten. Om posten finns på filen läser man in den i primärminnet och utför önskade förändringar i den. Därefter är det dags att skriva tillbaka posten till filen. Här får man tänka på att en läsning flyttat fram filpositionen till nästa post och man måste flytta tillbaka positionen de antal byte som posten består av. Det är detta som kallas direkt access och då använder man funktionen *fseek* enligt ovan.

Vill man ha reda på den filposition som man står på för tillfället använder man funktionen *ftell*, som returnerar nummer på den byte, räknat från filens början, som filpekaren står på just då. Funktionen ftell returnerar long int och börjar att räkna från 0.

Ex: Skriv de satser som behövs för att ta reda på en fils storlek i byte. Använd fseek och ftell.

```
.....
fseek(lfil, 0, SEEK_END);
storlek = ftell(lfil);
.....
```

7 Programmeringsteknik

Att skriva ett program innebär att man skriver en plan för hur bearbetningen av data ska utföras. Vilken typ av data och vilken typ av bearbetning, som ska göras, ska vara bestämt i en *specifikation,* innan man börjar programmera. Man bör också se till att hela tiden under programmeringen hålla kontakt med beställaren eller den som har skrivit specifikationen för eventuella frågor.

Att tillverka programmet innebär inte enbart att skriva koden rakt upp och ner. Man bör ha någon form av strategi eller teknik som man följer då programmet skrivs. En modell kan vara att man delar upp programmeringen i ett antal moment där varje moment dokumenteras noggrannt. Följande moment bör åtminstone ingå :

1)	Uppdelning i delprogram	---	exempelvis kan man använda *stegvis förfining* som man dokumenterar med *halvkod eller strukturdiagram,* som visar den grova uppdelningen av programmet i funktioner.
2)	Algoritmformulering	---	*problemlösning* där man gör nya funktioner över delproblem eller man använder färdiga funktioner, som man har i sitt bibliotek.Varje ny funktion bör dokumenteras med *halvkod* eller *strukturdiagram.*l Detta gäller åtminstone algoritmer som kommmer att återanvändas.
3)	Kodning	---	man realiserar algoritmen genom att koda den i något språk. *Koden* utgör dokumentet och ska vara snyggt skriven och kommenterad.
4)	Testning	---	programmet testas genom att enskilda funktioner först testas var för sig och sedan testas allt större och större ihopsatta delar. Testningen ska dokumenteras med ett *testprotokoll som visar testvärden och testresultat*. Observera att det aldrig går att testa sig fram till ett korrekt program. Det är mycket bättre att skapa sin algoritm för att testa sig fram.

Hur ska man dela upp i funktioner vid stegvis förfining? Detta är mycket indivudellt och beror bland annat på vilken erfarenhet man har som programmerare och vilka färdiga funktioner man har tillgång till. Varje programmerare bör ha en uppsättning färdiga funktioner som löser ofta förekommande problem vid programmeringen. Hit hör bland annat algoritmer för basproblem som sökning, sortering och användargränssnitt (menyer etc).

7.1 Sökning

Sökproblemet innebär att i en mängd av element hitta den eftersökta (nyckeln). Beroende på hur mängden ser ut kan man använda sig av olika metoder. I en helt oordnad mängd måste man använda *linjär sökning* d.v.s söka från början så länge man ej hittat nyckeln. Har man en ordnad mängd kan man använda *binär sökning* , vilket innebär att man alltid kan söka i rätt halva av mängden.

7.1.1 Linjär sökning

När man ska lösa ett problem är det alltid bra att *lösa ett konkret exempel eller ett liknande men enklare problem först*. Problemet linjär sökning kan lösas enligt:

Konkret exempel:

Element:	52 41 36 14 22 34 12
Sökt nyckel:	14
Lösning:	Jämför först 14 med 52, 14 med 41 o.s.v så länge nyckel skilt ifrån element. Om nyckeln finns i mängden anger man exempelvis vilket ordningsnummer den har annars anges att den ej finns i mängden.

Halvkod:

```
hämta nyckel
hämta första elementet i mängden

så länge ej slut på mängden och nyckel skilt ifrån elementet i mängden
  hämta nästa element i mängden

om ej slut på mängden
  nyckel finns på elementets plats i mängden
annars
  nyckel finns ej
```

Kod:

```
/* söker efter key i v med nr element och returnerar på */
/* vilken plats key finns eller 0 om den inte finns     */

int linsearch(int v[], int nr, int key)
{
   int i, svar;

   i = 0;

   while (i < nr && v[i] != key)
      i++;

   if (i < nr)
      /* hittat */
      svar = i + 1;
   else
      /* ej hittat */
      svar = 0;

   return svar;
}
```

OBS! Samma princip kan användas för sökning av andra typer av data. Man måste dock exempelvis vid sökning av namn använda jämförelsefunktionerna för strängar, stricmp, som jämför oberoende av liten- eller stor bokstav.

OBS! Alternativt kan man koda ovanstående sökfunktion med en flagga, hittat, som sätts till falskt från början och sedan till sant när man hittat nyckeln.

Ex: Skriv ett program som initierar en vektor och läser in ett tal som man ska söka efter i vektorn. Om talet finns i vektorn anges vilket ordningsnummer det har annars skrivs att talet finns ej i vektorn.

```
#include <stdio.h>

void main()
{
   int nyckel, nr, vek[10] = { 12, 34, 56, 23, 98,
                               16, 14, 25, 67, 38 };


   printf("Vilket tvåsiffrigt tal söks ? ");
   scanf("%d", &nyckel);

   nr = linsearch(vek, 10, nyckel);
   if ( nr > 0 )
      printf("Talet finns och har nummer %d!\n", nr );
   else
      printf("Talet finns ej!\n");
}
```

7.1.2 Binär sökning

Vid binär sökning måste man ha en *ordnad mängd* som man letar i. Genom att hela tiden söka i rätt halva av mängden hittas nyckeln snabbare än vid linjär sökning.

Konkret exempel:

Mängd:	14 36 50 53 58 65 72
Nyckel:	50
Lösning:	Jämför nyckeln 50 med mittersta elementet 53. Eftersom 50 är mindre än 53 fortsätter man att söka efter 50 i den vänstra mängden 14, 36 och 50. Jämför 50 med det mittersta elementet i den nya mängden 36. Eftersom 50 är större än 36 så sök i den högra mängden 50. Jämför 50 med 50 och nyckeln hittad.
Nyckel:	67
Lösning:	Jämför nyckeln 67 med mittersta elementet 53. Eftersom 67 är större än 53 fortsätter man att söka efter 50 i den högra mängden 58, 65 och 72. Jämför 67 med det mittersta elementet i den nya mängden 65. Eftersom 67 är större så sök i den högra mängden 72. Jämför 67 med 72. Eftersom 67 mindre än 72 och ingen fler mängd att söka i så finns ej nyckeln i mängden.

Halvkod:

```
hämta nyckel
hämta första elementets index och sista elementets index

om nyckel mindre än första elementet och nyckel större än sista elementet
  nyckeln finns ej
annars
  gör
    bestäm mittersta elementets index
    om nyckel mindre än mittersta
      byt sista elementidex till ett under mittersta
    annars om nyckel större än mittersta
      byt första elementindex till ett över mittersta
  så länge nyckel skilt ifrån mittersta element och mängd kvar att dela

  om nyckel lika med mittersta
    nyckel finns i mittersta
  annars
    nyckel finns ej
```

Kod:

```
/* söker efter key i v med nr sorterade element i     */
/* stigande ordning och returnerar på vilken plats    */
/* key finns eller 0 om den inte finns                */

int binsearch(int v[], int nr, int key)
{
    int mitti, mini = 0, maxi = nr - 1, svar;

    if (key < v[mini] || nr <= 0 || key > v[maxi])
       svar = 0;
    else
    {
       do
       {
          mitti = (mini + maxi) / 2;
          if (key < v[mitti] )
             maxi = mitti - 1;
          else if (key > v[mitti])
             mini = mitti + 1;
       }
       while (v[mitti] != key && mini <= maxi);

       if (v[mitti] == key)
          svar = mitti + 1;
       else
          svar = 0;
    }

    return svar;
}
```

Ex: efter — Skriv ett program som initierar en ordnad vektor och läser in ett tal som ska sökas i vektorn. Finns nyckeln i vektorn skrivs dess ordningstal ut annars att den inte finns.

```
#include <stdio.h>

void main()
{
   int nyckel, nr, vek[10] = { 12, 14, 23, 38, 56,
                               67, 78, 83, 88, 92 };


   printf("Vilket tvåsiffrigt tal söks ? ");
   scanf("%d", &nyckel);

   nr = binsearch(vek, 10, nyckel);
   if ( nr > 0 )
      printf("Talet finns och har nummer %d!\n", nr );
   else
      printf("Talet finns ej!\n");
}
```

7.2 Sortering

En mängd kan sorteras i stigande eller fallande storleksordning. Man kan sortera tal i storleksordning eller namn i bokstavsordning. Det finns ett sort antal algoritmer för sortering. En algoritm passar bättre för en typ av mängd än för en annan och en annan algoritm är snabbare än den andra. När det gäller snabbhet bör man för att få en snabb algoritm hålla nere *antalet byten* i första hand och *antalet jämförelser* i andra hand.

7.2.1 Urvalssortering

Urvalssortering är en mycket enkel sorteringsalgoritm som passar bra till en helt oordnad mängd.

Konkret exempel:

Mängd: 8 18 -1 7 0 -19 3

Lösning: Sök efter det minsta elementet i hela mängden som är -19 och byt plats mellan detta och det första elementet 8 enligt:

-19 18 -1 7 0 8 3

Fortsätt sedan med resten av mängden och sök efter minsta element som blir - 1. Byt plats enligt:

-19 -1 18 7 0 8 3

o.s.v så länge ej slut på mängden.

Halvkod:

```
för alla element från första till näst sista i mängden

  sätt minindex till elementets index

  för alla resterande element i mängden
    om resterande element mindre än minindexelementet
      sätt minindex till resterande elements index

  byt plats mellan minindexelement och element
```

Kod:
```
/* sorterar v med nr st element i stigande ordning */
void ursort(int v[], int nr)
{
   int i, j, minelemi, temp;

   for (i = 0; i < nr - 1; i ++)
   {
      minelemi = i;
      for (j = i + 1; j < nr; j++)
      {
         if ( v[j] < v[minelemi])
            minelemi = j;
      }
      temp = v[i];
      v[i] = v[minelemi];
      v[minelemi] = temp;
   }
}
```

Ex: Skriv ett program som slumpar 100 heltal till en vektor, sorterar och skriver ut vektorn.

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
void slump(int v[], int nr)
{
   int i;

   srand((unsigned)time(NULL));
   for (i = 0; i < nr; i++)
      v[i] = 100 + rand()%900;
}

void skriv(int v[], int nr)
{
   int i;

   for (i = 0; i < nr; i++)
   {
      if (i % 15 == 0)
         putchar('\n');
      printf("%4d", v[i]);
   }
   putchar('\n');
}

void main()
{
   int vek[100];

   slump(vek, 100);
   skriv(vek, 100);
   ursort(vek, 100);
   skriv(vek, 100);
}
```

7.2.2 Bubbelsortering

Bubbelsortering är en lämplig sorteringsmetod då *endast några element ligger i fel ordning*, exempelvis efter en uppdatering.

Konkret exempel:

Mängd: 8 18 -1 7 0 -19 3

Lösning: Jämför 8 med 18. Byt ej. Jämför 18 med -1 och byt till:

8 -1 18 7 0 -19 3

Jämför 18 med 7 och byt till:

8 -1 7 18 0 -19 3

Efter fortsatta jämförelser och byten har det största elementet 18 *bubblat upp till ytan*. Starta sedan från början och låt det näst största elementet bubbla upp till näst sista plats o.s.v. så länge det bubblar

Kod:

```
/* sorterar v med nr st element i stigande ordning*/

void bubbsort(int v[], int nr)
{
   int i = 0, maxi = nr - 1, bubbel = 1, temp;

   while ( bubbel && maxi > 0)
   {
      bubbel = 0;
      for (i = 0; i < maxi; i++)
      {
         if ( v[i] > v[i + 1] )
         {
            temp = v[i];
            v[i] = v[i + 1];
            v[i + 1] = temp;
            bubbel = 1;
         }
      }
      maxi--;
   }
}
```

7.2.3 Insticksortering

Insticksortering användes då man ska fylla på en tom mängd. Genom att se till att alltid stoppa in det nya elementet i mängden på rätt ställe har man alltid en sorterad mängd.

Konkret exempel:

Osorterad mängd :	8	18	-1	7	0	-19	3
Sorterad mängd:	8						
	8	18					
	-1	8	18				
	-1	7	8	18			
	o.s.v						

Lösning: Sätt in 8 på första plats. Jämför 18 med 8 och sätt in 18 efter 8. Jämför -1 med 18 och flytta fram 18. Jämför -1 med 8 och flytta fram 8. Sätt in -1 på första plats o.s.v så länge element att sätta in.

Kod:

```
/* sorterar vektorn v till vektorn s med nr st element i    */
/* stigande ordning                                         */
void insort(int v[], int s[], int nr)
{
   int i, j, klar;

   s[0] = v[0];
   for (i = 1; i < nr; i++)
   {
      klar = 0;
      j = i - 1;
      do
      {
         /* flytta fram */
         if (v[i] < s[j])
         {
            s[j+1] = s[j];
            j--;
         }
         /* sätt in */
         else
         {
            s[j+1] = v[i];
            klar = 1;
         }
      }
      while (!klar && j > -1);

      /* om första plats */
      if (!klar)
         s[0] = v[i];
   }
}
```

Ovan användes en extra vektor som parameter för att sortera med instickssortering. Man kunde istället ha använt en lokal vektor som sorteras och sedan kopieras över till den ursprungliga vektorn eller också kunde man ha *sorterat direkt då man skapade* mängden vid inläsningen eller slumpningen.

Ex : Skriv ett program som slumpar en sorterad vektor med 1000 hela tresiffriga tal och skriver ut vektorn.

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

void slumpsort(int v[], int nr, int min, int max)
{
   int i, j, klar, tal;

   srand((unsigned)time(NULL));
   v[0] = rand()%(max-min+1) + min;
   for (i = 1; i < nr; i++)
   {
      klar = 0;
      j = i - 1;
      tal = random(max-min+1) + min;
      do
      {
         if (tal < v[j])
         {
            v[j+1] = v[j];
            j--;
         }
         else
         {
            v[j+1] = tal;
            klar = 1;
         }
      }
      while (!klar && j > -1);

      if (!klar)
         v[0] = tal;
   }
}


void main()
{
   int vek[1000];

   slumpsort(vek, 1000, 100, 999);

   /* antag att skriv finns */
   skriv(vek, 1000);
}
```

7.3 Kodning

Har man gjort ett bra arbete vid algoritmformuleringen, som resulterat i en väl genomtänkt halvkod eller strukturdiagram, brukar ej kodningen ställa till något problem. Däremot är det viktigt att ha en *god stil* vid kodningen eftersom koden är ett viktigt dokument för underhåll och förändringar.

Regel: **Man ska skriva sitt program som om man skrev det för någon annan för nästa gång man läser programmet är man en annan!**

Hur ska man då lära sig en god kodningsstil. Det bästa sättet är att studera andras kod och kritisera det man tycker är fel samt hoppas på att andra läser ens egen kod och kritiserar den.

Ex:

```
for (i = 0; i < n; i++)
   for (j = 0; j < n; j++)
      v[i, j] = ((i+1)/ (j+1)) * ((j+1) /( i+1));
```

Vad gör ovanstående kod? Här får man verkligen tänka efter vad koden gör. Koden tillverkar en enhetsmatris med ettor i diagonalen och nollor för övrigt. Vad smart! Eller?

Här ska man åtminstone ha en kommentar före koden enligt:

```
/* tillverka en enhetsmatris */
for (i = 0; i < n; i++)
   for (j = 0; j < n; j++)
      v[i, j] = ((i+1)/ (j+1)) * ((j+1) /( i+1));
```

Frågan är också om man inte ska koda på ett klarare och tydligare sätt och verkligen markera vad koden gör enligt:

```
/* tillverka en enhetsmatris */
for (rad = 0; rad < max; rad++)
{
   for (kol = 0; kol < max; kol++)
   {
      /* sätt in 0 utom i diagonalen som sätts till 1 */
      if (rad != kol)
         v[rad, kol] = 0;
      else
         v[rad, kol] = 1;
   }
}
```

Regel: **Skriv tydligt och klart! Var ej för smart vid kodningen! Tänk på att någon annan ska förändra och underhålla koden! Kommentera vettigt!**

Kommentarer är viktiga för att göra en kod tydligare och mera lättläst. En annan detalj som också är viktig att tänka på när man skriver kod är *variabelnamnen*. Väl valda variabelnamn, som klart utsäger vad variabeln användes till, ger klarare kod. Jämför a) med b) nedan. Vilken kod är lättast att förstå?

Ex: a)
```
/* beräkning av poäng i bowling */
s = 0;
b = 1;
for (i = 1; i <= 10; i++)
{
   if ( p[b] == 10 )
   {
      s = s + 10 + p[b+1] + p[b+2];
      b++;
   }
   else if ( p[b] + p[b+1] == 10 )
   {
      s = s + 10 + p[b+2];
      b += 2;
   }
   else
   {
      s = s + p[b] + p[b+1];
      b += 2;
   }
}
```

b)
```
/* beräkning av poäng i bowling */
score = 0;
ball = 1;
for (frame = 1; frame <= 10; frame++)
{
   /* strike */
   if ( pins[ball] == 10 )
   {
      score += 10 + pins[ball+1] + pins[ball+2];
      ball++;
   }
   /* spare */
   else if ( pins[ball] + pins[ball+1] == 10 )
   {
      score += 10 + pins[ball+2];
      ball += 2;
   }
   else
   /* regular */
   {
      score += pins[ball] + pins[ball+1];
      ball += 2;
   }
}
```

Regel: **Använd variabelnamn som utsäger något och skiljer sig från varandra på ett tydligt sätt!**

En viktig del av den kod som man skriver utgörs av strukturerna selektion och iteration. Här gäller det att *tänka först och koda sedan*. Man bör ha tänkt ut en vettig struktur i exempelvis halvkod först innan man sätter sig ner för att koda.

När det gäller *iterationer* ser man ofta i litteraturen att man *använder for-loopar i alla lägen istället för den mer logiska while-konstruktionen*. För iterationer är följande regel lämplig när man ska välja loop-sats.

Regel: **Man ska använda for-loopar endast då man vet hur många gånger något ska upprepas! I övrigt ska man använda while-loopar.**

Selektioner kräver ofta extra eftertanke för att bli klara och tydliga.

Ex:

```
if (pris <= 50.0)
   rabatt = 0.0;
if (pris > 50.0)
   if (pris <= 100.0)
      rabatt = 0.02*pris;
if (pris > 100.0)
   rabatt = 0.03*pris;
```

Flera if-satser inuti varandra kan ofta sättas ihop till en, genom att använda villkor med de logiska operatorerna &&, || och !.

```
if (pris <= 50.0)
   rabatt = 0.0;
if (pris > 50.0 && pris <= 100)
      rabatt = 0.02*pris;
if (pris > 100.0)
   rabatt = 0.03*pris;
```

Vid uteslutande händelser, som det handlar om ovan, är det ännu klarare och tydligare att skriva en flervalsselektion.

```
if (pris <= 50.0)
   rabatt = 0.0;
else if (pris <= 100.0)
   rabatt = 0.02 * pris;
else
   rabatt = 0.03*pris;
```

Även när man använder flervalsselektioner kan det bli snårigt, om man ej tänkt igenom selektionerna ordentligt i förväg.

Ex:

```
if (length >= 30 && length < 50)
   if (wing < 0.6*length)
      weight1 = (1 + 0.08 - 0.037)* weight;
   else
      weight1 = (1 + 0.08 + 0.045)*weight;
else if (length >= 50 && length < 60)
   if (wing < 0.6*length)
      weight1 = (1 + 0.09 - 0.037) * weight;
   else
      weight1 = (1 + 0.09 + 0.045) * weight;
else if (length >= 60 && length < 80)
   if (wing < 0.6*length)
      weight1 = (1 + 0.105 - 0.037) * weight;
   else
      weight1 = (1 + 0.105 + 0.045) * weight;
else
   if (wing < 0.6*length)
      weight1 = (1 + 0.122 - 0.037) * weight;
   else
      weight1 = (1 + 0.122 + 0.045) * weight;
```

Tänker man efter här ser man att selektionen kan skrivas betydligt enklare enligt:

```
/* correction for wing */
if (wing < 0.6 * length)
   corr = 1.0 - 0.037;
else
   corr = 1.0 + 0.045;

/* correction for length */
if (length >= 80)
   corr += 0.122;
else if (length >= 60)
   corr += 0.105;
else if (length >= 50)
   corr += 0.09;
else if (length >= 30)
   corr += 0.08;

/* new weight */
weight1 = corr * weight;
```

Regel: **Man ska tänka efter ordentligt innan man kodar selektioner. Undvik nästlade if-satser och använd flervalsselektioner där det går!**

Data som ska matas in till ett program ställer alltid till trassel. Orsaken till detta är att det är människor som matar in data och människor gör fel. Därför ska programmet konstrueras så att det *även klarar av att hantera felaktiga indata*.

Den metod som man ofta använder sig av är ett *inmatningsfilter som inte släpper igenom annat än korrekt och acceptabel indata*.

Ex: Skriv ett program som beräknar en triangels area med Herons formel. Alla triangelsidor ska inläsas säkert och man får ej acceptera sidor som ej bildar någon triangel.

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

void main()
{
   float sid[3], omk, area;
   char sidstr[10];
   int nr;

   /* säker inmatning av tre triangelsidor */
   do
   {
      for (nr = 0; nr < 3; nr++)
      {
         do
         {
            printf("Ge sida %d : ", nr);
            gets(sidstr);
            sid[nr] = atof(sidstr);
            if (sid[nr] <= 0)
               printf("Fel data!\n");
         }
         while (sid[nr] <= 0);
      }

      omk = (sid[0] + sid[1] + sid[2]) / 2;

      if (sid[0] > omk || sid[1] > omk || sid[2] > omk)
         printf("Ingen triangel!\n");
   }
   while (sid[0] > omk || sid[1] > omk || sid[2] > omk);

   /* beräkning av triangelns area med Herons formel */
   area = sqrt(omk*(omk-sid[0])*(omk-sid[1])*(omk-sid[2]));
   printf("Arean = %.2f ", area);
}
```

OBS! Funktionen atof returnerar normalt det omvandlade reella talet. Går strängen ej att omvandla returneras 0. Ovan kan vi ej heller acceptera negativa resultat.

Välj indata som är mänskliga. Ord från vardagslivet, ord som säger något, ord som har en mening. Ofta använder man tal och siffror till allting, Använd istället strängar eller egenuppräknade variabler.

Ex: Skriv ett program som läser in metaller och deras vikt samt beräknar priset. Använd ej 1, 2 och 3 för metallen utan använd metallens kemiska beteckning.

```
#include <stdio.h>
#include <math.h>
#include <string.h>

void main()
{
   enum metalltyp { Al, Sn, Cu } metall;
   float vikt, pris[3] = { 3.0, 2.0, 5.0 };
   char viktstr[10], metallstr[10];
   int ok;

   /* säker inmatning av metall */
   do
   {
      ok = 1;
      printf("Ge metall (Al, Sn, Cu) : ");
      gets(metallstr);
      if ( stricmp(metallstr, "Al") == 0)
         metall = Al;
      else if (stricmp(metallstr, "Sn") == 0)
         metall = Sn;
      else if (stricmp(metallstr, "Cu") == 0)
         metall = Cu;
      else
      {
         ok = 0;
         printf("Fel data!\n");
      }
   }
   while ( !ok );

   /* säker inmatning av vikt */
   do
   {
      printf("Ge vikt : ");
      gets(viktstr);
      vikt = atof(viktstr);
      if (vikt <= 0)
         printf("Fel data!\n");
   }
   while (vikt <= 0);

   /* beräkning av pris */
   printf("Pris = %.2f\n", pris[metall]*vikt);
}
```

Regel: **Kräv indata som är mänskliga! Testa alltid att indata är korrekt! Släpp ej igenom felaktig indata!**

7.4 Testning

En viktig del av programmeringen är att se till att programmet klarar av att hantera felaktiga indata. Ännu viktigare är naturligtvis att se till att programmet bearbetar rätt indata på ett korrekt sätt.

Har man tänkt igenom sina strukturer ordentligt, brukar det mesta fungera. Testningen är ett sista moment i programmeringen och ska naturligtvis utföras på ett genomarbetat sätt och dokumenteras med testdata och testresultat. Speciellt viktigt är det att testa på extremvärden och randvärden.

Räknare i loopar måste man se upp med så att de räknas upp korrekt.

Ex: Försök till en medelvärdesberäkning:

```
sum = 0;
antal = 1;
printf("Ge tal : ");
scanf("%f", &tal);
while (tal != 0.0)
{
   sum += tal;
   antal++;
   printf("Ge tal : ");
   scanf("%f", &tal);
}
printf("Medelvärde = %f", sum/antal);
```

Antalet räknas upp fel. Man har ett för mycket i antal vid iterationens slut. Korrigera till:

```
antal = 0;
```

Ex: Försök till summering av 100 vektorelement:

```
sum = 0;

for(i = 0; i <= 100; i++)
   sum += vek[i];

printf("Summan = %f\n", sum);
```

Här summeras ett extra element i minnet som kan innehålla vilket värde som helst. Programmet fungerar om detta extra element skulle råka vara 0. Korrigera till:

```
for(i = 0; i < 100; i++)
```

Regel: **Testa speciellt på randvärden och extremvärden samt se upp med att initieringarna är riktiga och att räknare räknas upp korrekt!**

Det svåraste felen att hitta vid testning är sådana fel som *endast dyker upp ibland* för vissa typer av data. Ett sådant fel brukar vara att man testar likhet mellan flyttal. Att jämföra ett flyttal med 0 brukar alltid gå bra. Att däremot jämföra om två flyttal är lika eller olika kan lyckas för vissa värden och misslyckas för andra.

Ex: Skriv ett program som övar division av flyttal.

```
#include <stdio.h>

void main()
{
   float taljare, namnare, kvot, svar;
   int antal = 0;

   printf("Ge täljare : ");
   scanf("%f", &taljare);
   printf("Ge nämnare : ");
   scanf("%f", &namnare);

   kvot = taljare / namnare;

   do
   {
      printf("Vad blir kvoten ? ");
      scanf("%f", &svar);
      if (svar == kvot)
         printf("Svaret är rätt!\n");
      else
      {
         printf("Svaret är fel!\n");
         antal++;
         if (antal == 3)
            printf("Korrekt svar = %f", kvot);
      }
   }
   while (svar != kvot && antal < 3);
}
```

Programmet ovan fungerar för exempelvis 1.8/0.9 om man ger svaret 2.0 men inte för 1.8/0.6 om man ger svaret 3.0. Det fel som man gör ovan är jämförelserna:

```
if (svar == kvot)

while (svar != kvot && antal < 3);
```

Man ska aldrig jämföra likhet mellan flyttal utan man ska kontrollera skillnaden mellan talen och se om den är mindre än den noggrannhet som man arbetar med enligt:

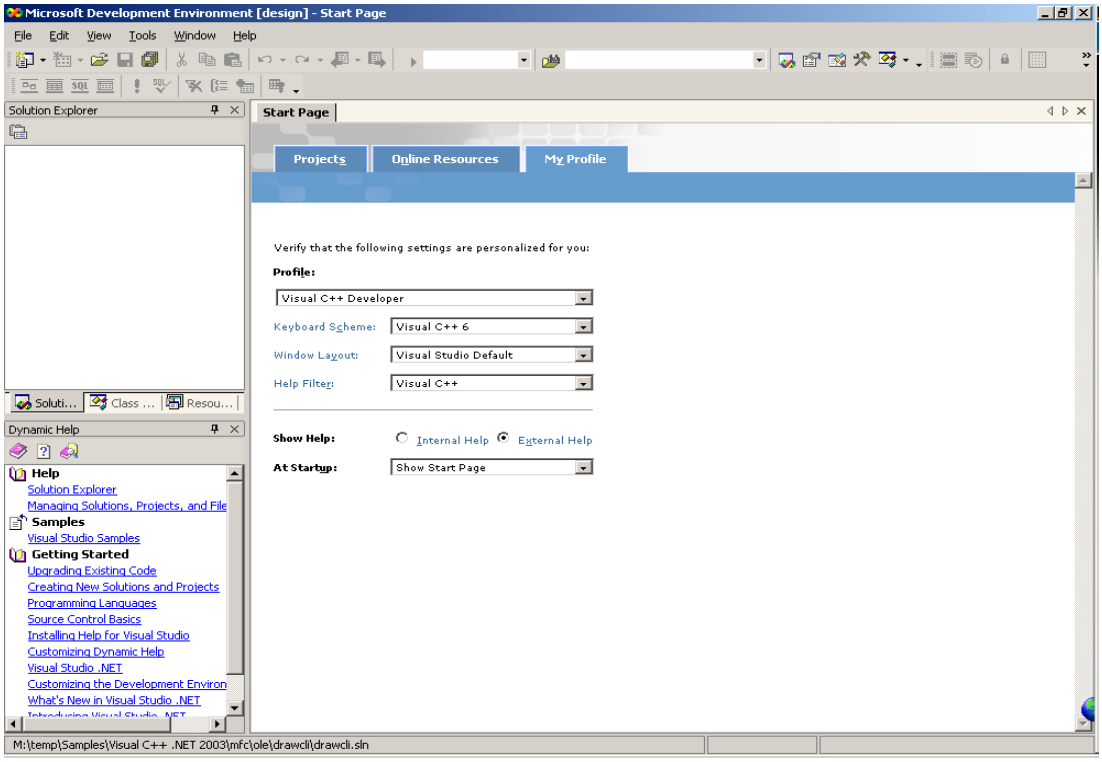
```
if (fabs(svar - kvot) <= fabs(kvot*1e-6))

while (fabs(svar - kvot) > fabs(kvot*1e-6) && antal < 3);
```

Regel: Testa aldrig direkt likhet eller olikhet mellan flyttal!

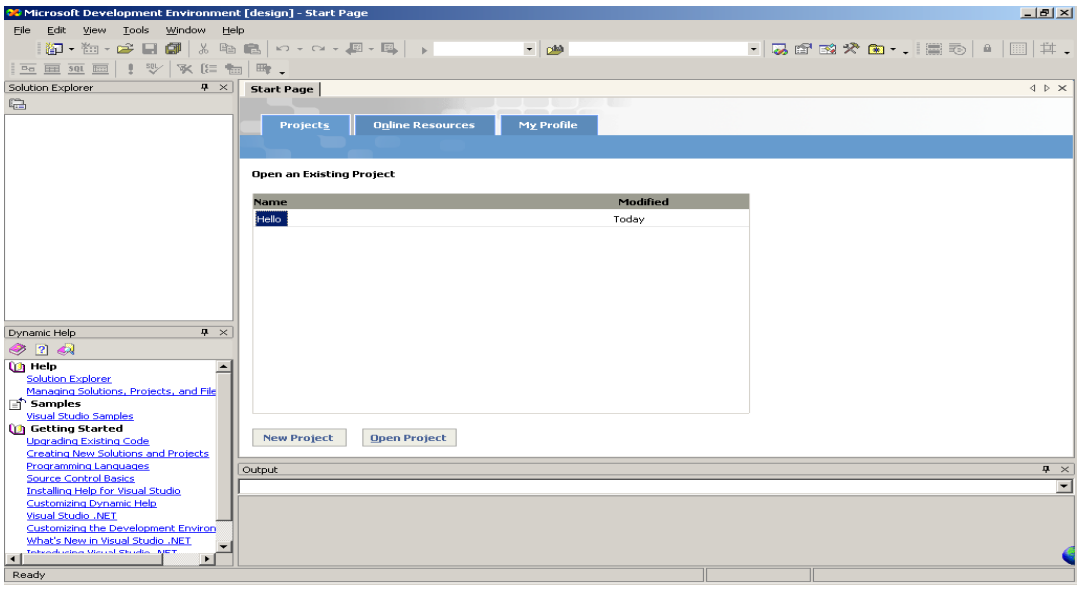
Datorövning 1

A) Logga in på din användare. Välj från startmenyn Start | Program | Microsoft Visual Studio.NET 2003 | Microsoft Visual Studio.NET 2003. En startsida dyker upp där du genom att klicka på fliken MyProfile kan sätta din aktuella profil enligt fönstret nedan.

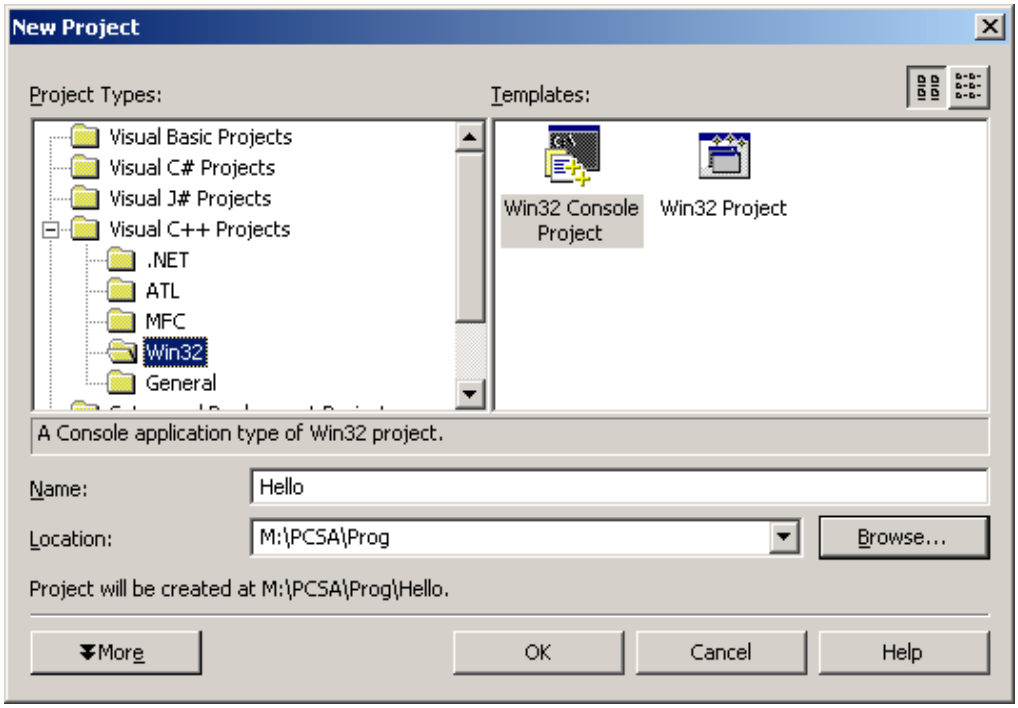
Får du ej fram någon startsida startar du den från huvudmenyn Help | Show Start Page.



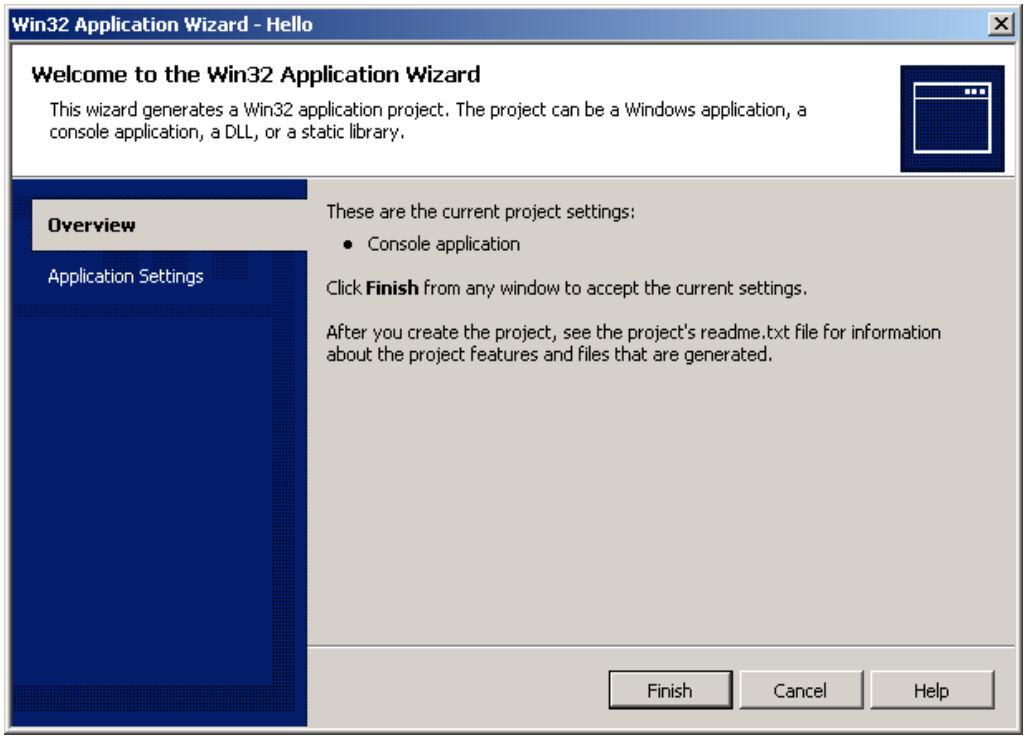
Klicka på fliken Projects på startsidan och du får upp en sida där du kan välja mellan att starta ett nytt projekt eller öppna ett gammalt projekt enligt:



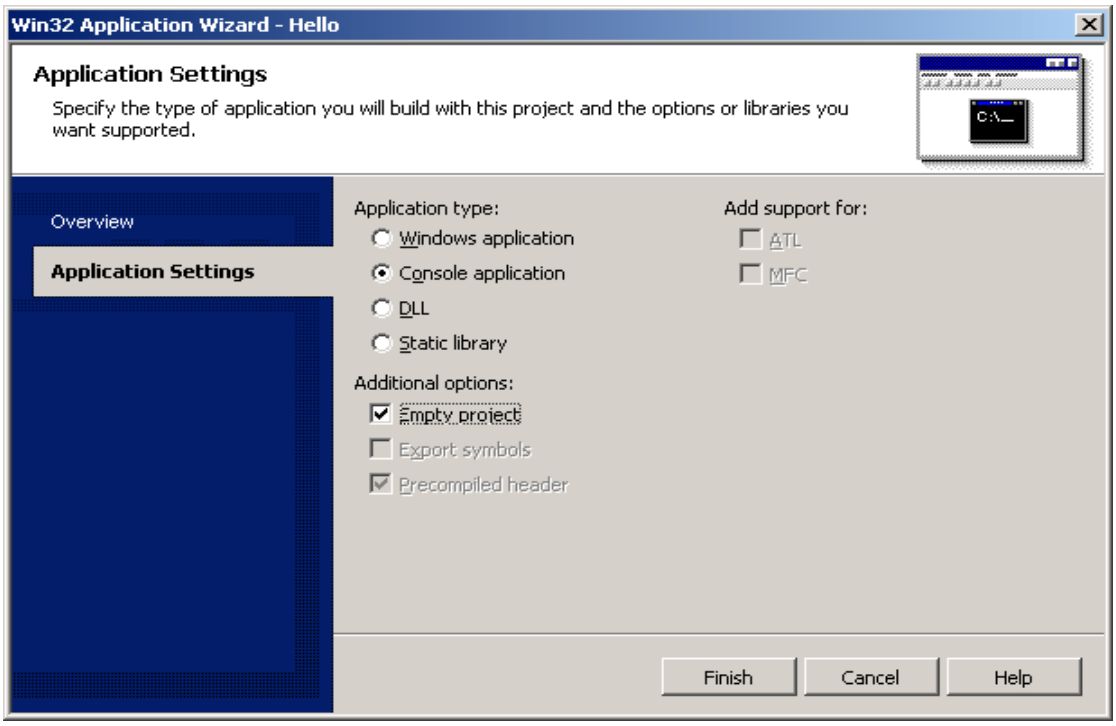
Välj att starta ett nytt projekt genom att klicka på New Project. Det dyker upp ett nytt fönster. Här ska du välja vilken typ av program som du ska skriva. Välj Visual C++ Win32 och klicka på Win32 Console Project. Sätt projektnamnet till Hello. Klicka sedan på Browse och bläddra till M:\PCSA där du skapar katalogen Prog genom att i fönstret trycka ner höger musknapp och välja Nytt och sedan Ny Mapp. Programmet eller projektet kommer att hamna i katalogen M:\PCSA\Prog\Hello\ och heta Hello.



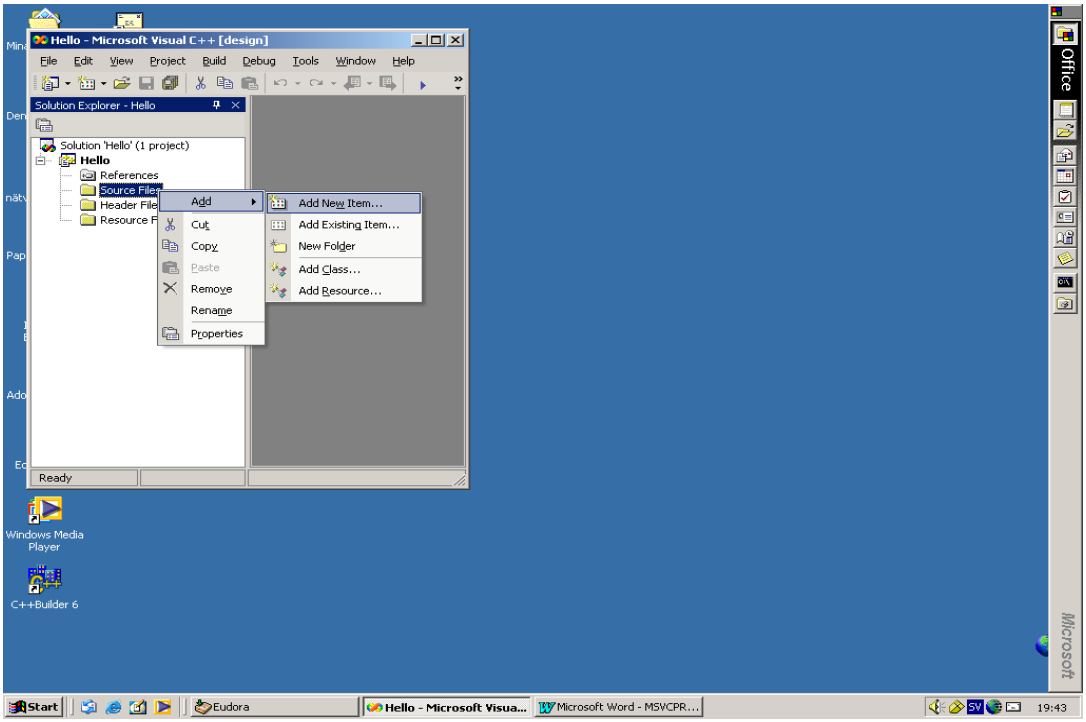
Klicka sedan på OK och du får fram ett nytt fönster enligt:



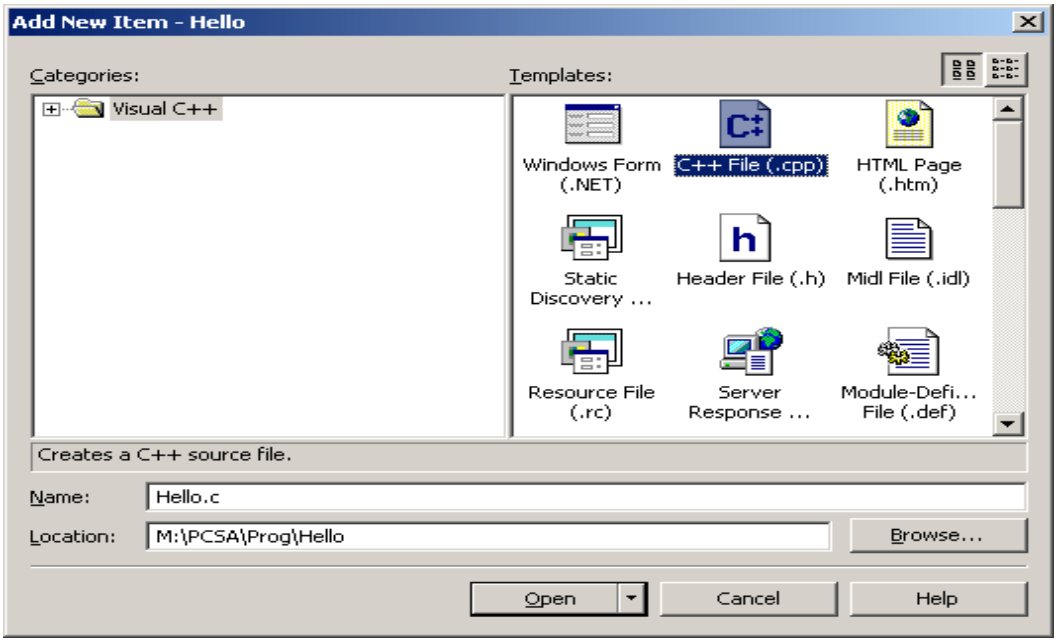
Klicka på Application Settings i detta fönster och då dyker det upp ett nytt där du väljer Empty Project och klickar på Finish enligt:



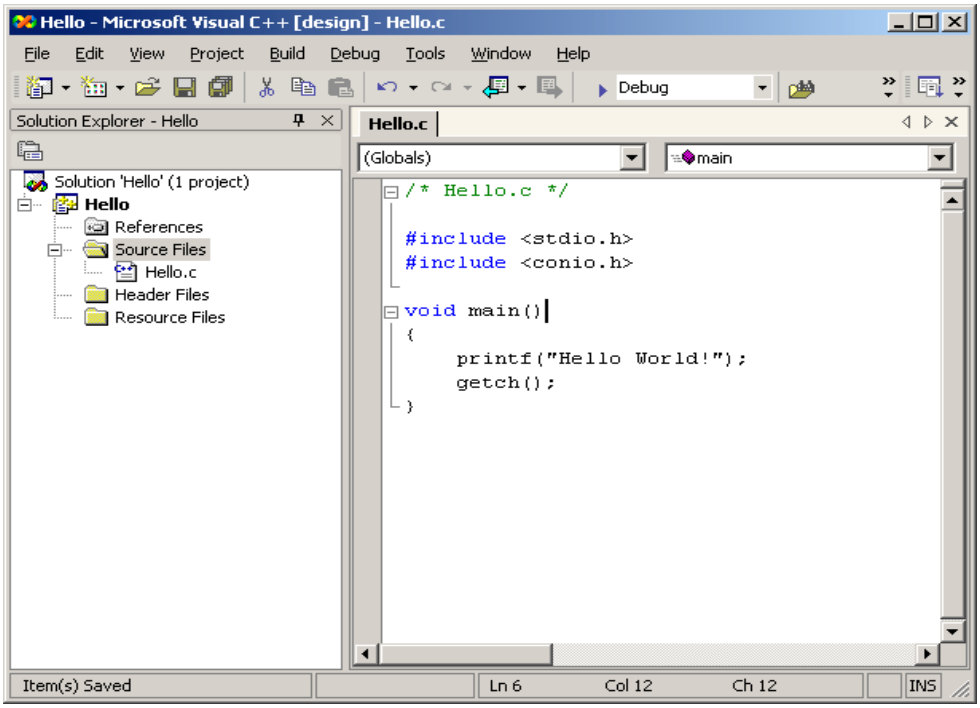
Nu är du inne i den interaktiva miljön där du ska skriva ditt första C-program. Klicka med höger musknapp på Source Files i Soloution Explorer och välj Add | Add New Item enligt:



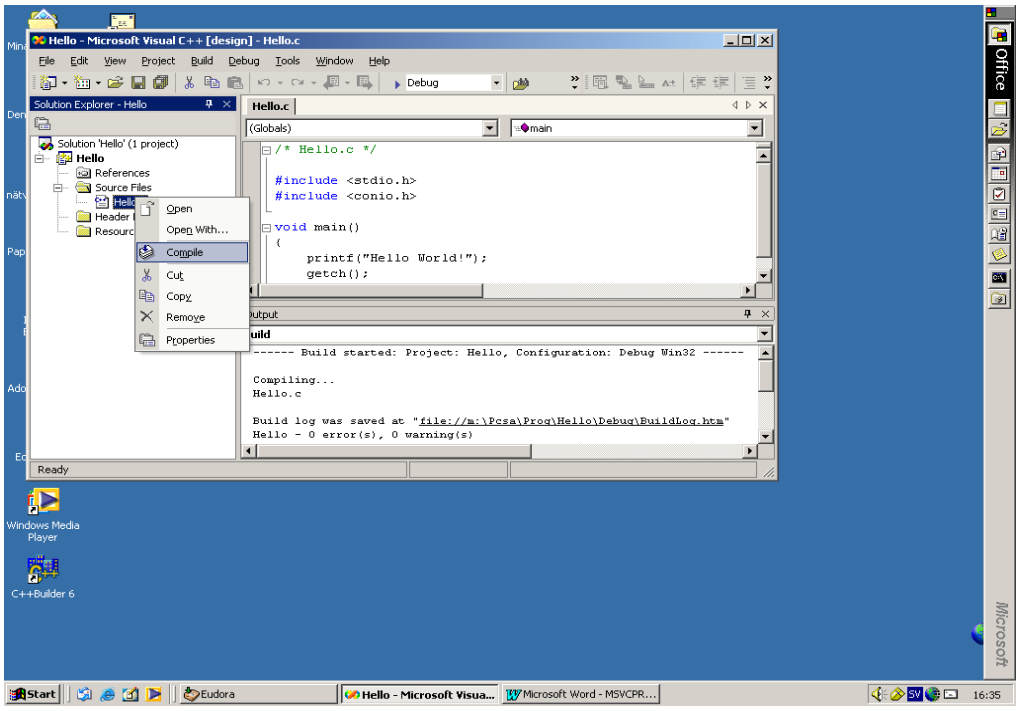
Får du inte fram Solution Explorer automatiskt, använd View på huvudmenyn. I fönstret Add New Item - Hello väljer du C++File och ger filen namnet Hello.c. (OBS! .c)



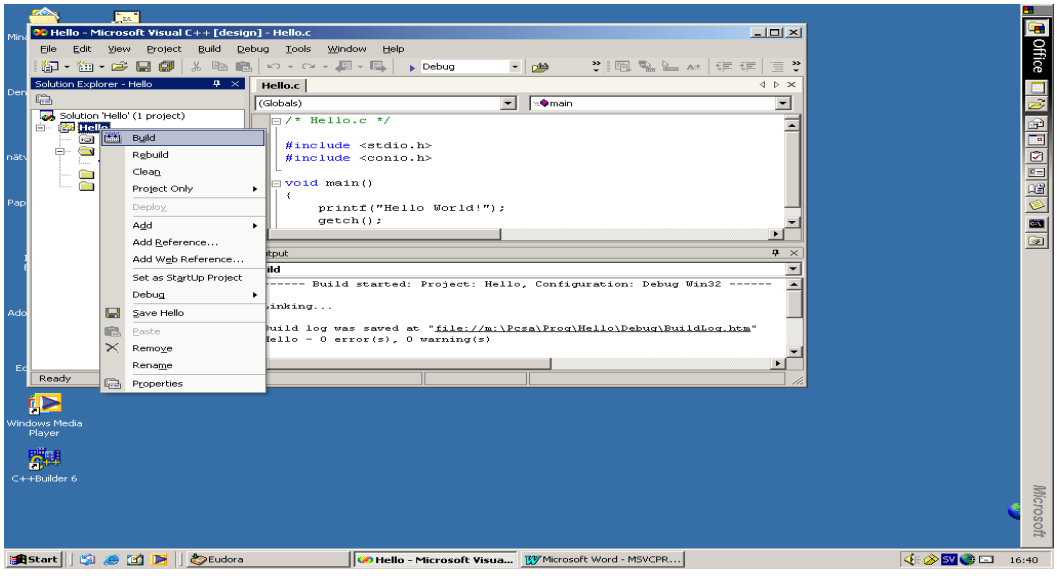
Nu ska du skriva koden för ditt C-program. Klicka på filen Hello.c i Solution Explorer så dyker editorfönstret upp, där du ska skriva programmet Hello.c enligt:



När du skrivit koden färdigt sparar du den med File | Save Hello.c. Nästa steg är kompilering. Klicka med höger musknapp på Hello.c i Solution Explorer och välj Compile enligt:

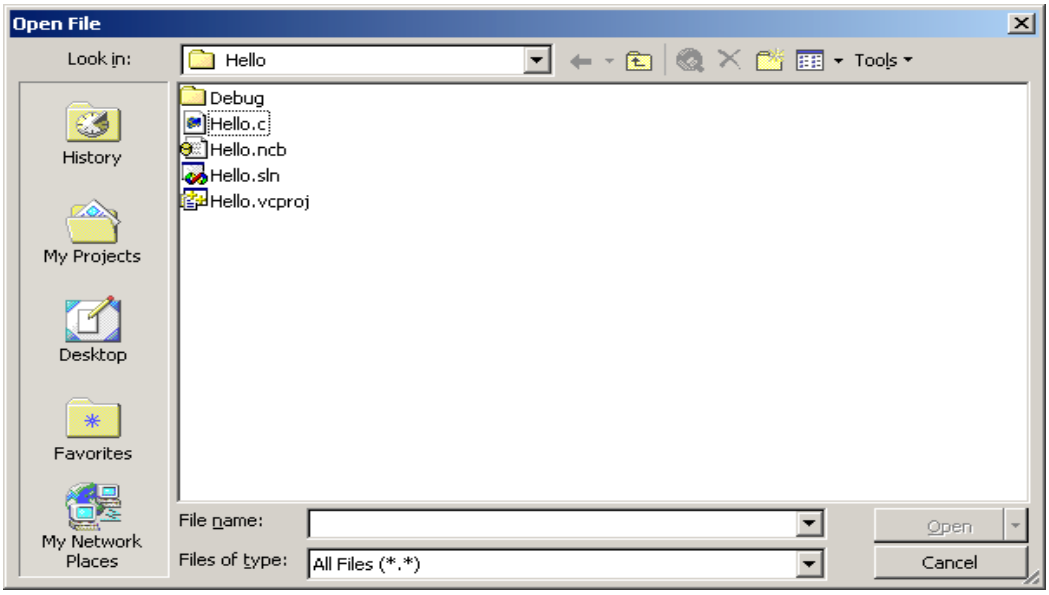


Kompileringen har gått bra om det står 0 errors och 0 warnings i Outputfönstret nedanför Editorfönstret. Får du kompileringsfel kan du dubbelklicka på felet och felets position i koden kommer att markeras. För att slippa en massa onödiga varningar kan du välja Project | Proporties och C/C++-general och sätta Warning Leval till l. Efter att ha rättat till eventuella kompileringsfel länkar du ihop programmet till körbar fil genom att i Solution Explorer med höger musknapp klicka på Hello och välja Build enligt:

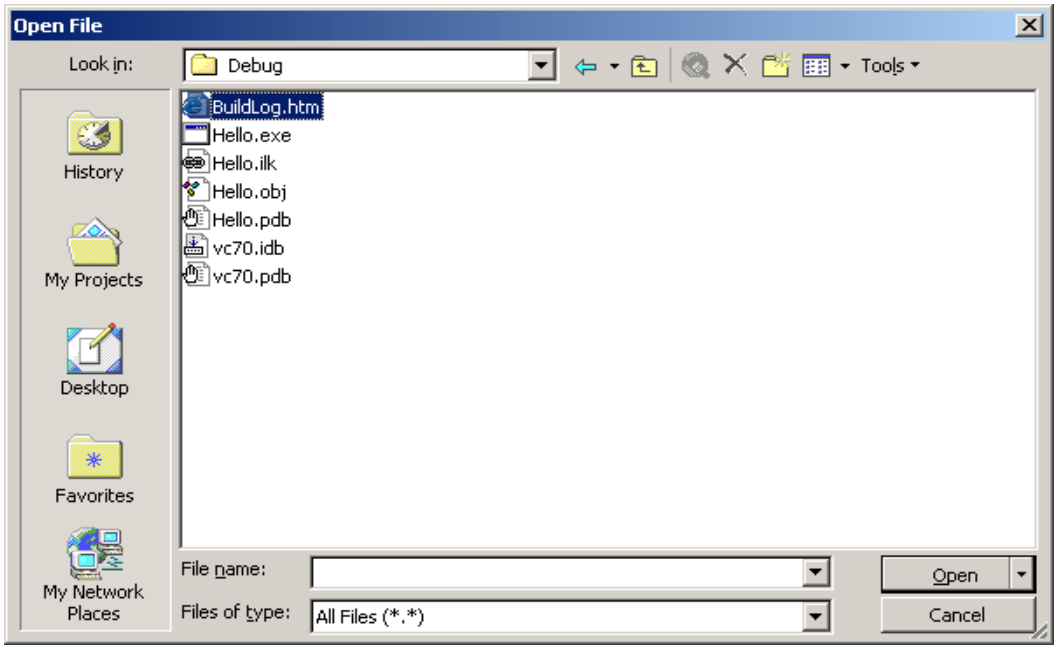


Får du felmeddelanden vid länkningen måste du rätta felen och länka på nytt. Felmeddelanden visas i samma fönster som kompileringsfelen. Glöm ej att spara på nytt med File | Save, så fort du ändrat i koden.

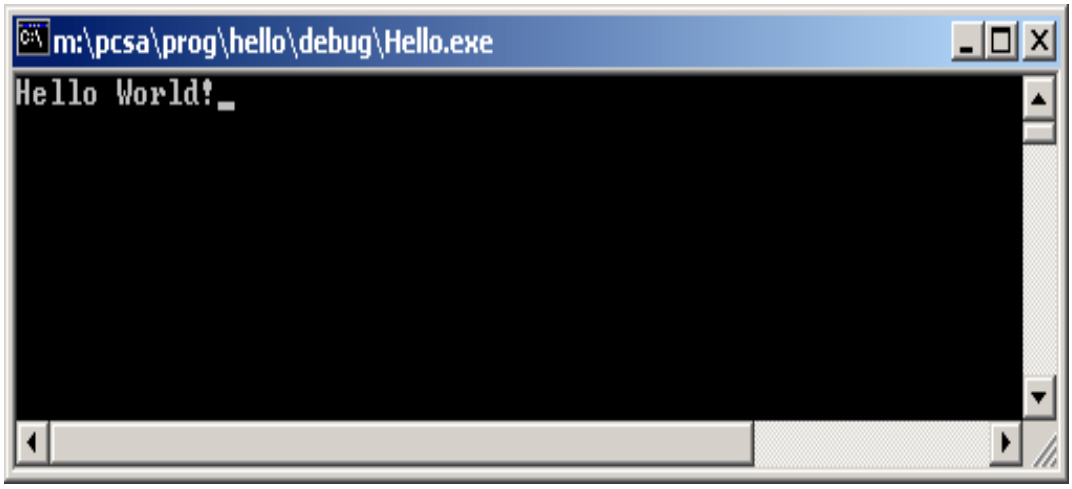
Kontrollera att du har nedanstående filer i katalogen M:\PCSA\Prog\Hello genom att välja File | Open | File i huvudmenyn och få fram fönstret.



där Debug-katalogen ska innehålla



Du kan köra (exekvera) programmet genom att från huvudmenyn välja Debug | Start eller klicka på den gråa pilen till vänster om Debug i verktygsfältet. Programmet körs i ett fönster enligt:



Programmet skriver ut angiven text och väntar på att du ska trycka på någon tangent. Då programmet kört färdigt kommer fönstret att stängas. Det program som vi skapat är ett Console-program, som körs i ett Console-fönster. Kommentera bort anropet av funktionen getch och kompilera länka och kör. Nu hinner du ej se resultatet av körningen eftersom det är getch som stoppar körningen och väntar på tangenttryck.

Innan du börjar skriva nästa program ska du från huvudmenyn välja File | Close Solution och sedan välja New Project antingen från huvudmenyn eller Startsidan.

B) Skriv, spara, kompilera, länka och kör nedanstående program. Börja med att välja File | New osv.

```
/* Bensin.c v1.0                    */
/* Ditt namn och din användare      */
/* MSVisual C++                     */
/* Beräkning av bensinpris          */

#include <stdio.h>
#include <conio.h>

void main()
{
   float literpris, liter, totalpris;

   /* läs in literpris */
   printf("Literpris = ");
   scanf("%f", &literpris);

   /* läs in antal liter */
   printf("Antal liter = ");
   scanf("%f", &liter);

   /* beräkna och skriv ut totalpris */
   totalpris = literpris * liter;
   printf("Totalpris = %.2f kr\n", totalpris);

   /* vänta på tangent */
   getch();
}
```

Efter att du kört programmet och sett att det fungerar, skriver du ut källkoden genom att välja File | Print från huvudmenyn. Du måste först lägga till den aktuella salens skrivare med Start | Inställningar | Skivare | Lägg till skrivare.

C) Skriv ett program i filen summa.c som läser in två reella tal, beräknar och skriver ut talens summa, skillnad, produkt och kvot.

D) Komplettera programmet summa så att kvoten bara beräknas om nämnaren (det andra talet) är skilt ifrån 0. Är nämnaren 0 ska istället ett felmeddelande skrivas ut.

E) Öppna filen bensin.c. Komplettera programmet i filen så att det upprepat frågar efter antalet liter och skriver ut totalpriset. Avslutning av programmet ska ske då 0 liter ges.

*F) Sätt på motsvarande sätt in en upprepning i programmet i filen summa.c så att den upprepas så länge det första talet ej är 0. Matar man in 0 på det första talet ska programmet avslutas direkt utan att det andra talet efterfrågas.

*G) Komplettera bensin-programmet så att inläsningen av antal liter alltid sker mitt på en tom skärm. Använd funktionerna clrscr och gotoxy i MSViscon.h och MSViscon.c, som du kan hämta från kurssidans länk Studprog och placera i baskatalogen M:\PCSA\Prog. Filen MSViscon.h inkluderar du med #include ”..\MSViscon.h” och med Add Existing Item på SourceFiles i Solution Explorer adderar du MSViscon.c till projektet.

Datorövning 2

A) Skriv ett program som läser in massa och höjd för en kropp och beräknar och skriver ut dess potentiella energi enligt g*massa*höjd. Definiera konstanten g = 9.81 först med define och sedan med const.

B) Skriv ett program som läser in antalet timmar, minuter och sekunder samt beräknar och skriver ut motsvarande tid i sekunder.

C) Skriv ett program som läser in två sidor och mellanliggande vinkel för en triangel samt beräknar och skriver ut triangelns area enligt 0.5*sida1*sida2*sin(v). Vinkeln v ska läsas in i grader.

D) Skriv ett program som läser in en tid i hela sekunder och skriver ut den i timmar, minuter och sekunder.

E) Skriv ett program som läser in ett datum på formen yymmdd (6-siffrigt heltal exempelvis 010124 och på skärmen skriver ut :

År : 20yy
Mån : mm
Dag : dd

F) Låt oss kalla ett tresiffrigt positivt heltal för ett spegeltal om entalssiffran är lika med hundratalssiffran. Talet 474 är exempelvis ett spegeltal. Skriv ett program som läser in ett 3-siffrigt tal och kontrollerar om det är ett spegeltal.

G) Skriv ett program som läser in en stor bokstav mellan A och Z och skriver ut nästa stora bokstav i alfabetet. Bokstaven Z måste du hantera separat och skriva ut texten 'Detta är den sista stora bokstaven!'.

*H) Komplettera programmet i D) så att det upprepat frågar efter tid. Programmet ska avslutas då tiden 0 matas in.

*I) Skriv ett program som läser in en entals- och en tiotalssiffra som tecken och sedan beräknar motsvarande heltal som multipliceras med 25 och skrivs ut.

**J) Skriv ett program som inkluderar headerfilerna limits.h och float.h, som innehåller information om ditt aktuella C-system. Skriv sedan ut största och minsta tal som kan sparas i en variabel av typerna char, int, long, float, float och long float.

**K) Komplettera uppgift G) ovan så att även Å, Ä och Ö behandlas.

Datorövning 3

A) Skriv ett program som frågar efter ett heltal och skriver ut om talet är udda eller jämnt.

B) Skriv ett program som frågar efter en positiv vinkel i hela grader och skriver ut om vinkeln är spetsig (mindre än 90°), trubbig (större än 90°) eller rät.

C) Skriv ett program som frågar efter ett reellt tal och skriver ut talets kvadratrot med 3 decimaler. Om det inlästa talet är negativt ska ett felmeddelande skrivas ut.

D) Skriv ett program som frågar efter en siffra mellan 0 och 5 och skriver ut motsvarande morsekod. Använd switch-sats. Morsekoderna för 0 till 5 är:

```
0    -----     1         .----     2         ..---
3    ...--     4         ....-     5         .....
```

E) Under vissa förutsättningar gäller att bromssträckan för en bil med hastigheten v i km/h ges av formeln :

$$s = 0.015v^2 \text{ (meter)}$$

Skriv ett program som skriver ut en tabell som visar bromsträckorna för hastigheterna 30, 40, 50, ..., 130 km/h.

F) Lägg till en upprepning i uppgift C) så att programmet upprepat frågar efter ett reellt tal och skriver ut kvadratroten. Upprepningen avslutas om talet 0 inmatas.

G) Skriv ett program som beräknar antalet termer (n) som behövs för att summan av den harmoniska serien 1 + 1/2 + 1/3 ..... + 1/n ska överstiga 10.

*H) Skriv ett program som skriver ut delsumman 1 - 1/2 + 1/3 - 1/4 ...1/n för ett inläst värde på n. Inläsningen ska upprepas och avslutas då 0 inläses.

*I) Skriv ett program som slumpar ett tal mellan 0 och 99 och frågar efter en gissning av talet. Beroende på storleken av det gissade talet skall texten RÄTT, FÖR STORT eller FÖR LITET skrivas ut på skärmen. Programmet ska upprepa så länge man gissar fel och vid avslut ska antalet gissningar skrivas ut. För att slumpa tal ska du använda funktionerna srand som startar slumpgeneratorn slumpmässigt och rand som returnerar slumptalet. Använd hjälpen med F1 för dessa funktioner.

Datorövning 4

A) Skriv ett program som definierar en vektor innehållande 4 heltalselement, läser in värden till alla element och skriver ut elementen i omvänd ordning.

B) Skriv ett program som läser in 5 reella tal till en vektor och därefter skriver ut vektorns summa, största och minsta element.

C) Skriv ett program som slumpar 100 tresiffriga heltal till en vektor, skriver ut vektorn, sorterar vektorn samt skriver ut den sorterade vektorn.

D) Skriv ett program som med funktionen gets läser in en sträng innehållande ett antal ord som åtskiljs med blanktecken. Skriv sedan ut strängen tecken för tecken och byt rad efter varje ord alltså vid blanktecken. Strängen kan maximalt innehålla 79 tecken.

E) Skriv ett program som läser in en sträng med gets och skriver ut strängen baklänges. För att bestämma strängens längd kan du använda den färdiga funktionen strlen i string.h.

F) Skriv ett program som läser in data till en post som innehåller termerna medlemsnummer, namn och telefonnummer och skriver ut postens termer på skärmen.

G) Skriv ett program som läser in data till två poster av typen

```
struct rtal
{
   int taljare;
   int namnare;
};
```

som ska avbilda bråktal som exempelvis 3/5. Programmet ska efter inläsning addera ihop bråken till ett summabråk och sedan skriva ut detta summabråk.

*H) Skriv ett program som skapar en skruv-vektor innehållande skruvdiametrar med värdena 1.80, 2.15, 2.50, 2.80, 3.15, 3.45, 3.80, 4.15, 4.50, 4.80, 5.45, 6.15, 6.80, 7.50, och 8.15, frågar efter en håldiameter och skriver ut den skruvdiameter som är närmast större.

*I) Skriv ett program som läser in flera medlemsposter enligt F) ovan till en vektor av poster innehållande max 10 medlemmar. Inläsningen avslutas då medlemsnummer 0 inläses. Programmet ska avslutningsvis fråga efter ett medlemsnummer och skriva ut namn och telefonnummer för denna medlem.

**J) Skriv ett program som läser in ett heltal i form av en sträng, omvandlar strängen till ett tal och slutligen skriver ut talet multiplicerat med 2. Exempelvis ska "123" omvandlas till 123 och utskriften ska bli 246. Större än 4-siffriga tal behöver ej hanteras.

**K) Skriv ett program som slumpar en lottorad bestående av 7 tal mellan 1 och 35. Observera att samma tal ej får återkomma. Lottoraden skrivs slutligen ut sorterad.

Datorövning 5

A) Skriv ett program som läser in ett tecken och antalet gånger som tecknet ska skrivas ut samt anropar en funktion för utskriften med huvud enligt :

```
void skriv_tecken(char tecken, int antal)
/* skriver ut tecken antal gånger */
```

Testa att köra programmet i Debuggern genom att med F11 köra programmet radvis och växla mellan kodfönster och console. Kolla variabelvärden genom att ställa dig med muspekaren på variabeln och vänta.

B) Skriv ett program som läser in ett positivt heltal och kontrollerar om talet är ett primtal dvs. att det ej är jämnt delbart med något annat heltal större än 1 utom sig självt. Primtalstestet ska göras av en funktion som tar talet som parameter och returnerar 1 (sant) om primtal annars 0 (falskt). I funktionen ska du upprepat dividera med först 2 sedan 3 osv. till tal-1 och returnera 0 så fort det går jämnt upp. Går det inte jämnt upp någon gång returneras 1.

C) Skriv ett program som läser in ett pris i kr och avrundar priset till närmaste 50 öre. Exempelvis ska priset 23.36 avrundas till 23.50 och 23.78 till 24.00. Avrundningen ska ske i en funktion som tar priset som parameter och returnerar det avrundade värdet Börja med att i funktionen plocka ut hela kr med hel = (int)pris och sedan decimaldelen med dec = pris - hel.

*D) Skriv ett program som skapar en vektor initierad med 10 tresiffriga vinstnummer i ett lotteri och som sedan läser in ett lottnummer och kontrollerar om lotten har vinst eller är en nitlott. Programmet ska innehålla en funktion som tar vektorn och lottnumret som parametrar och returnerar numret på vinsten (1 till 10) om vinstlott annars 0.

*E) Skriv ett program som läser in ett personnummer i form av en sträng med 11 tecken och som kontrollerar och skriver ut om personnumrets alla tecken är ok dvs alla tecken utom det 7:e, som kan vara + eller -, är siffror. Kontrollen ska göras i en funktion som returnerar 1 om allt ok annars 0.

**F) Komplettera programmet i E) ovan med en funktion som anropas efter teckenkontrollen och som kontrollerar om datum är korrekt angivet med månad mellan 1 och 12 och dag mellan 1 och 31.

**G) Skriv ett program som i en funktion slumpar värden till en vektor bestående av 10 poster numrerade från 100 till 109 med värden mellan 10.0 och 20.0 av typen:

```
struct matdata
{
   int nr;
   float x;
};
```

Sortera sedan posterna efter värdet på x och skriv ut dessa. Slumpning, sortering och utskrift ska ske i funktioner med lämpliga parametrar.

Datorövning 6

A) Skriv ett program som från tangentbordet läser in ett antal (avslutas med 0.0) reella tal och skriver in dessa tal i en textfil rtal.txt, ett tal per rad. Kontrollera filen i editorn efter det att du kört programmet.

B) Skriv ett program som läser filen rtal.txt ovan och beräknar talens medelvärde.

C) Skriv ett program som räknar antalet semikolon i en textfil vars namn inläses. Testa på något av dina C-program.

D) Skriv med editorn en textfil innehållande ett antal personers namn, ett per rad. Skriv sedan ett program som läser filen och skriver ut namnen numrerade på skärmen. Numret (börja med 1) ska stå framför namnet på samma rad.

E) Skriv ett program som slumpar 500 tärningskast och skriver in dessa i en binärfil tarning.dat.

F) Skriv ett program som läser binärfilen tarning.dat ovan och skriver ut tärningsutfallens medelvärde som ska bli nära 3.5 om slumpningen är tillfredsställande.

*G) Skriv ett program som slumpar temperaturer mellan 20 och 30 grader till posterna med nummer 100 till och med 199 av typen :

```
struct matpost
{
   int nr;
   float temp;
};
```

och skriver in posterna i en binärfil temp.dat.

*H) Skriv ett program som läser filen temp.dat ovan och skriver ut posterna med lägsta resp högsta temperatur på skärmen.

**I) Skriv med editorn en textfil bil.txt, som innehåller ett antal bilar med registrerings-nummer, ägare och bilmärke, på var sin rad. Skriv sedan ett program som läser värden från bil.txt till poster av lämplig typ och skriver in posterna i en binärfil bil.dat.

**J) Skriv ett program som läser in ett registreringsnummer, söker efter denna bil i bil.dat ovan och uppdaterar bilens ägare. Den nya ägarens namn läses in.

Datorövning 7

*A) I filen slump.c (finns i din katalog om du kopierat alla filer från kurssidans Studprog) finns funktionen slump som slumpar en vektor med heltal mellan ett minsta och ett största värde och i filen search.c finns funktionen linsearch som söker linjärt efter en nyckel i en vektor och om nyckeln finns returnerar nyckelns platsnummer. Skriv ett program som slumpar en vektor innehållande ett inläst antal(max 100) tresiffriga heltal, skriver ut vektorn och sedan frågar efter ett heltal och söker efter dess plats i vektorn. Kopiera in de funktioner som du behöver från slump.c och search.c till ditt program.

*B) I filen slump.c i din katalog finns funktionen slumpsort som slumpar en sorterad vektor med heltal mellan ett minsta och ett största värde och i filen search.c finns funktionen binsearch som söker binärt efter en nyckel i en vektor och om nyckeln finns returnerar nyckelns platsnummer. Skriv ett program som slumpar en sorterad vektor med ett inläst antal (max 100) fyrsiffriga heltal, skriver ut vektorn och sedan frågar efter ett heltal och söker efter dess plats i vektorn.

*C) I filen sort.c i din katalog finns sorteringsrutinen ursort som sorterar vektorer i stigande ordning. Skriv ett program, som använder slump-funktionen i slump.c för att slumpa ett inläst antal (max 100) tre-siffriga heltal till en vektor, skriver ut vektorn på skärmen, sorterar vektorn med ursort samt skriver ut den sorterade vektorn.

**D) Använd funktionen clock i time.h för att betämma hur lång tid i sekunder det tar för ursort att sortera en slumpad vektor med 1000, 2000 resp. 3000 slumpade heltal. Hur ändras sorteringstiden med antalet element i vektorn. Använd hjälpen med F1 för att få reda på hur clock-funktionen används.

**E) Jämför sorteringstiderna för att sortera en slumpad vektor med 2000 element för de tre sorteringsalgoritmerna ursort, bubbsort och insort som alla finns i sort.c.

**F) Skriv om sök-funktionerna i search.c så att de söker efter en nyckelsträng i en vektor av strängar. Spara funktionerna i stsearch.c och skriv ett huvudprogram där du initierar en vektor med 10 strängar och sedan läser in en sträng som man söker efter i vektorn och om den finns skriver ut platsnummer för.

**G) Skriv om sorterings-funktionerna i sort.c så att de sorterar strängar. Spara funktionerna i strsort.c och skriv sedan ett program som initierar en vektor av strängar och skriver ut strängarna sorterade på skärmen.

**H) Skriv en textfil med editorn innehållande ett antal namn, en per rad. Skriv sedan ett program som läser in namnen från filen till en vektor, sorterar vektorn och skriver ut namnen sorterade på samma fil.

Inlämningsuppgift 1, Simhopp

Specifikation

Skriv ett program som läser in en simhoppares startnummer, hoppnummer, svårighetsgrad och 7 domarpoäng. Efterhand som domarpoängen läses summeras dessa och största och minsta domarpoäng bestäms. Slutligen beräknas hoppets poäng och skrivs ut. Skärmens ska se ut enligt nedan där du har matat in de understrukna.

Startnummer (avsluta med 0) : 235
Hoppnummer : 7
Svårighetsgrad : 2.3

Domarpoäng 1 : 6.5
Domarpoäng 2 : 7.0
Domarpoäng 3 : 7.5
Domarpoäng 4 : 5.5
Domarpoäng 5 : 6.0
Domarpoäng 6 : 7.0
Domarpoäng 7 : 6.5

Hoppoäng : 45.54

Tryck tangent för nytt hopp!

Hoppoängen räknas ut genom att man efter inläsningen minskar summan av alla domarpoäng med minsta och största domarpoäng och beräknar ett medelvärde av de resterande domarpoängen. Slutligen multipliceras detta medelvärde med 3 och med svårighetsgraden.

Programmet ska upprepa och avslutas direkt om startnummer 0 inläses. Inga vektorer ska användas.

Extra (ej obligatoriskt) : Säker inmatning av domarpoäng mellan 0 och 10 i steg om 0.5.

Redovisning

Programmet redovisas genom att det uppvisas och provkörs för läraren. Godkänd redovisning utgör en del av delkurs 2 i Programmering C.

Inlämningsuppgift 2, Ishockey

Specifikation

Kopiera alla filer från kurssidans Studprog till din aktuella katalog. I din aktuella katalog finns nu demo-programmet ishodemo.exe. Provkör programmet genom att dubbelklicka på det i utforskaren.

Programmet simulerar ett antal omgångars spel i elitserien i ishockey. Det slumpar resultatet för 6 matcher i varje omgång, där *lagen spelar mot varandra parvis i den ordning som de står i tabellen*. Man slumpar antalet mål, mellan 0 och 8, för hemmalaget resp bortalaget. Hemmalag är alltid det lag som står före i tabellen.

Efter varje omgång redovisas matchresultatet och lagen tilldelas gjorda mål, insläppta mål och poäng. Vinnande lag får 3 poäng och förlorande lag 0 poäng. Vid oavgjort resultat får båda lagen 1 poäng. Efter omgångens resultat redovisas en serietabell där lagen är sorterade i första hand efter poäng, om samma poäng, efter målskillnad (gjorda - insläppta) och om både samma poäng och samma målskillnad, efter antalet gjorda mål.

I filen ishostud.c, som nu finns i din aktuella katalog, finns en början på ovanstående program. Din uppgift är att fullborda programmet, med de delar som saknas, så att programmet fungerar på samma sätt som demo-programmet.

Extra (ej obligatoriskt) : Vid oavgjort resultat i dagens elitserie förlänger man matchen med 5 minuter följt av eventuella straffar tills något lag vunnit. Det vinnande laget får 2 poäng och det förlorande laget får behålla sin enda poäng. Komplettera programmet så att det vid oavgjort resultat även simulerar en förlängning.

Redovisning

Programmet redovisas genom att det uppvisas och provkörs för läraren. Godkänd redovisning utgör en del av delkurs 2 i Programmering C.

Inlämningsuppgift 3, Luffarschack.

Specifikation

I din aktuella katalog finns demoprogrammet Luffdemo.exe som är en förenklad variant av luffarschack med enbart 9 rutor. Spelet spelas mellan två spelare , en som markerar kryss(x) och en annan som markerar ring(o). Det är alltid spelaren med kryss som börjar. Den spelare som först får en hel rad eller en hel kolumn eller en hel diagonal med sina markeringar har vunnit. Om ingen vinnare finns blir resultatet remi.

Provkör programmet i utforskaren genom att flytta runt med piltangenterna och när du vill ha en markering trycker du på ENTER.

Programmet i filen Luffstud.c , som finns i din aktuella katalog är en del av ovanstående program. Programmet saknar dock funktionen kontrollera som kontrollerar om spelplanen har någon vinnare. Din uppgift är att skriva funktionen kontrollera så att den kontrollerar om någon vinnare finns och rapporterar detta omedelbart.

Redovisning

Programmet redovisas genom att det uppvisas och provkörs för läraren. Godkänd redovisning utgör en del av delkurs 2 i Programmering C.

Inlämningsuppgift 4, Kvalitetsstatistik

Specifikation

På en komponentfabrik tillverkas motstånd i ett antal serier. Efter tillverkningen kontrollmäter man motståndens resistanser. För att få en jämnare kvalitet på tillverkningen vill man följa upp mätningarna genom att se hur avvikelserna från det nominella resistansvärdet varierar med tiden för varje serie.

I din aktuella katalog finns en textfil measure.txt som innehåller ett antal serier med uppmätta resistansvärden för motstånd med nominella resistansen 220 Ω. Varje serie avslutas med resistansvärdet 0.0. Demoprogrammet kvaldemo.exe , som finns i samma katalog, läser mätfilen och ritar upp en trendkurva, som visar den procentuella avvikelsen för varje serie från det nominella resistansvärdet. Provkör programmet i utforskaren och välj från menyn genom att flytta dig dit med piltangenterna och verkställa med RETURN. Skriv in namnet på mätfilen enligt ovan och välj sedan 'Visa statistik' från menyn. Avsluta programmet.

Programmet i filen kvalstud.c, som också finns i din katalog, är en del av ovanstående program. Din uppgift är att fullborda programmet genom att skriva funktionerna las_data och visa_stat**.**

Las_data läser resistansvärden från mätfilen och fyller vektorn serie med ett antal serieposter, en per serie. Varje seriepost ska hålla reda på seriens totala antal motstånd, totala antalet som avviker mer än 5% från det nominella värdet och medelvärdet av alla resistanser i serien. Funktionen ska returnera antalet serier.

Visa_stat ska i diagramform rita en trendkurva som visar medelvärdenas avvikelser för varje serie från det nominella värdet samt, avslutningsvis, det totala antalet motstånd och det totala antalet utanför 5%-gränsen både absolut och i procent. För ritning av kurvan ska du använda gotoxy i conio.h

Redovisning

Programmet redovisas genom att en fullständig rapport inlämnas innehållande försättsblad, innehållsförteckning, sammanfattning av vad programmet gör samt bilagor i form av användaranvisning (manual), strukturdiagram för funktionerna las_data och visa_stat, källkod och en körbar fil kvalstud.exe i din aktuella katalog. Glöm inte att ange din användaridentitet och i vilken katalog den körbara filen finns. Dessutom ska trendkurvan ingå som bilaga. Detta gör du genom att med PrintScrn-tangenten kopiera skärmen då grafen visas till Clipboard och sedan i Word klistra in grafen på rätt plats i rapporten.

Godkänd rapport utgör en del av delkurs 2 i Programmering C.

Library Facilities Alphabetized by Name

Syntax	Header File	Purpose
`void abort(void);`	`stdlib.h`	Abnormally terminates a program.
`int abs(int x);`	`stdlib.h`	Returns the absolute value of an integer.
`double acos(double x);`	`math.h`	Returns the arc cosine of the input value (argument must be in the range –1 to 1).
`char *asctime (const struct tm *tblock);`	`time.h`	Converts a time stored as a structure in `*tblock` to a 26-character string.
`double asin(double x);`	`math.h`	Returns the arc sine of the input value (argument must be in the range–1 to 1).
`void assert(int test);`	`assert.h`	If `test` evaluates to zero, `assert` prints a message on `stderr` and aborts the program.
`double atan(double x);`	`math.h`	Calculates the arc tangent of the input value.
`double atan2(double y, double x);`	`math.h`	Calculates the arc tangent of `y/x`.
`int atexit(void (*func)(void));`	`stdlib.h`	Registers a function to be called at normal program termination.
`double atof(const char *s);`	`math.h`	Converts a string pointed to by `s` to `double`.
`int atoi(const char *s);`	`stdlib.h`	Converts a string pointed to by `s` to `int`.
`long int atol(const char *s);`	`stdlib.h`	Converts a string pointed to by `s` to `long int`.
`void *bsearch(const void *key, const void *base, size_t nelem, size_t width, int (*fcmp)(const void *, const void *));`	`stdlib.h`	Binary search of the sorted array `base`: returns the address of the first entry in the array that matches the search key using the comparison routine `*fcmp`; if no match is found, returns 0.

Library Facilities Alphabetized by Name

Syntax	Header File	Purpose
`void *calloc(size_t nitems, size_t size);`	stdlib.h	Allocates a memory block of size `nitems` × `size`, clears the block to zeros, and returns a pointer to the newly allocated block.
`double ceil(double x);`	math.h	Returns the smallest integer not less than `x`.
`void clearerr(FILE *stream);`	stdio.h	Resets `stream`'s error and end-of-file indicators to 0.
`clock_t clock(void);`	time.h	Returns processor time elapsed since the beginning of program invocation.
`double cos(double x);`	math.h	Calculates the cosine of a value (angle in radians).
`double cosh(double x);`	math.h	Calculates the hyperbolic cosine of a value.
`char *ctime (const time_t *time);`	time.h	Converts date and time value pointed to by `time` (the value returned by function `time`) into a 26-character string representing local time.
`double difftime (time_t time2, time_t time1);`	time.h	Calculates the difference between two times in seconds.
`div_t div(int numer, int denom);`	stdlib.h	Divides two integers, returning quotient and remainder in a structure whose components are `quot` and `rem`.
`void exit(int status);`	stdlib.h	Terminates program. Before termination, all files are closed, buffered output (waiting to be output) is written, and any registered "exit functions" (posted with `atexit`) are called; status of 0 indicates normal exit; a nonzero status indicates some error.
`double exp(double x);`	math.h	Calculates the exponential function e^x.
`double fabs(double x);`	math.h	Calculates the absolute value of a floating-point number.
`int fclose(FILE *stream);`	stdio.h	Closes the named stream.
`int feof(FILE *stream);`	stdio.h	Predicate that detects end of file on a stream.
`int ferror(FILE *stream);`	stdio.h	Predicate that detects errors on a stream.
`int fflush(FILE *stream);`	stdio.h	Flushes a stream: If the stream has buffered output, `fflush` writes the output for `stream` to the associated file.
`int fgetc(FILE *stream);`	stdio.h	Gets a character from a stream.

Library Facilities Alphabetized by Name

Syntax	Header File	Purpose
`int fgetpos(FILE *stream, fpos_t *pos);`	stdio.h	Gets the current file pointer and stores it in the location pointed to by pos.
`char *fgets(char *s, int n, FILE *stream);`	stdio.h	Copies characters from stream into the string s until it has read n-1 characters or newline character, whichever comes first. Marks the end of s with the null character.
`double floor(double x);`	math.h	Returns the largest whole number not greater than x.
`double fmod(double x, double y);`	math.h	Calculates x modulo y, the remainder of x divided by y.
`FILE *fopen (const char *filename, const char *mode);`	stdio.h	Opens file named by filename and associates a stream with it. Modes and meanings "r" is read, "w" is write, "a" is append, "r+" is existing file update (reading and writing), "w+" is new file update (reading and writing), "a+" is update at the end of the file
`int fprintf(FILE *stream, const char *format [, argument, ...]);`	stdio.h	Writes formatted output to a stream.
`int fputc(int c, FILE *stream);`	stdio.h	Outputs a character to a stream.
`int fputs(const char *s, FILE *stream);`	stdio.h	Outputs a string to a stream.
`size_t fread(void *ptr, size_t size, size_t n, FILE *stream);`	stdio.h	Reads up to n items of data, each of length size bytes, from the given stream into a block pointed to by ptr; returns number of items read.
`void free(void *block);`	stdlib.h	Deallocates a memory block allocated by a previous call to calloc, malloc, or realloc
`FILE *freopen (const char *filename, const char *mode, FILE *stream);`	stdio.h	Associates a new file with an open stream; often used for redirecting standard streams.
`double frexp(double x, int *exponent);`	math.h	Splits a double number into mantissa and exponent.
`int fscanf(FILE *stream, const char *format [, address, ...]);`	stdio.h	Scans and formats input from a stream.

Syntax	Header File	Purpose
`int fseek(FILE *stream, long int offset, int whence);`	stdio.h	Repositions the file pointer associated with `stream` to a new position that is `offset` bytes from the file location given by whence.
`int fsetpos(FILE *stream, const fpos_t *pos);`	stdio.h	Positions the file pointer of a stream to a new position that is the value obtained by a previous call to `fgetpos` on that stream.
`long int ftell(FILE *stream);`	stdio.h	Returns the current file pointer for `stream` as the number of bytes from the beginning of the file.
`size_t fwrite (const void *ptr, size_t size, size_t n, FILE *stream);`	stdio.h	Writes n × `size` bytes to `stream` from the memory block pointed to by `ptr`.
`int getc(FILE *stream);`	stdio.h	Gets a character from `stream`.
`int getchar(void);`	stdio.h	Gets a character from `stdin`.
`char *getenv(const char *name);`	stdlib.h	Returns the value of a specified variable.
`char *gets(char *s);`	stdio.h	Gets a string (one line) from `stdin`; discards any newline character.
`struct tm *gmtime (const time_t *timer);`	time.h	Converts date and time to Greenwich mean time (GMT).
`int isalnum(int c);`	ctype.h	Predicate returning nonzero if `c` is a letter or a decimal digit.
`int isalpha(int c);`	ctype.h	Predicate returning nonzero if `c` is a letter.
`int iscntrl(int c);`	ctype.h	Predicate returning nonzero if `c` is a delete character or an ordinary control character.
`int isdigit(int c);`	ctype.h	Predicate returning nonzero if `c` is a decimal digit.
`int isgraph(int c);`	ctype.h	Predicate returning nonzero if `c` is a printing character other than a space.
`int islower(int c);`	ctype.h	Predicate returning nonzero if `c` is a lowercase letter.
`int isprint(int c);`	ctype.h	Predicate returning nonzero if `c` is a printing character.
`int ispunct(int c);`	ctype.h	Predicate returning nonzero if `c` is a punctuation character.

Library Facilities Alphabetized by Name

Syntax	Header File	Purpose
`int isspace(int c);`	`ctype.h`	Predicate returning nonzero if `c` is a space, tab, carriage return, new line, vertical tab, or form feed.
`int isupper(int c);`	`ctype.h`	Predicate returning nonzero if `c` is an uppercase letter.
`int isxdigit(int c);`	`ctype.h`	Predicate returning nonzero if `c` is a hexadecimal digit (0 to 9, A to F, a to f).
`long int labs(long int x);`	`math.h`	Computes the absolute value of the parameter `x`.
`double ldexp(double x, int exp);`	`math.h`	Calculates $x \times 2^{exp}$.
`ldiv_t ldiv(long int numer, long int denom);`	`stdlib.h`	Divides two `long ints`, returning quotient and remainder in a structure whose components are `quot` and `rem`.
`struct lconv *localeconv(void);`	`locale.h`	Sets up country-specific monetary and other numeric formats.
`struct tm *localtime (const time_t *timer);`	`time.h`	Accepts the address of a value returned by `time` and returns a pointer to a structure of type `tm` in which the time is corrected for the time zone and possible daylight savings time.
`double log(double x);`	`math.h`	Calculates the natural logarithm of `x`.
`double log10(double x);`	`math.h`	Calculates $\log_{10}(x)$.
`void longjmp(jmp_buf jmpb, int retval);`	`setjmp.h`	Restores the task state captured by the last call to `setjmp` with the argument `jmpb`; then returns in such a way that `setjmp` appears to have returned with the value `retval`.
`void *malloc(size_t size);`	`stdlib.h`	Allocates a block of `size` bytes from the memory heap and returns a pointer to the newly allocated block.
`int mblen (const char *s, size_t n);`	`stdlib.h`	Returns the size in bytes of the multibyte character pointed to by `s` (`n` is the maximum size of the character).
`size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);`	`stdlib.h`	Converts up to `n` multibyte characters from string `s` to wide characters stored in array `pwcs`.
`int mbtowc(wchar_t *pwc, const char *s, size_t n);`	`stdlib.h`	Converts the multibyte character accessed by `s` to a wide character.

Library Facilities Alphabetized by Name

Syntax	Header File	Purpose
`void *memchr(const void *s, int c, size_t n);`	`string.h`	Searches the first n bytes of the block pointed to by `s` for first occurrence of character `c`.
`int memcmp(const void *s1, const void *s2, size_t n);`	`string.h`	Compares two blocks for a length of exactly n bytes; return value < 0 means `s1` less than `s2`, value = 0 means same as, and value > 0 means greater than.
`void *memcpy(void *dest, const void *src, size_t n);`	`string.h`	Copies a block of n bytes from `src` to `dest` (behavior undefined if `src` and `dest` overlap); returns `dest`.
`void *memmove(void *dest, const void *src, size_t n);`	`string.h`	Copies a block of n bytes from `src` to `dest` (copy is correct even if `src` and `dest` overlap); returns `dest`.
`void *memset(void *s, int c, size_t n);`	`string.h`	Sets the first n bytes of the array `s` to the character `c`.
`time_t mktime(struct tm *t);`	`time.h`	Converts the time in the structure pointed to by `t` into a calendar time.
`double modf(double x, double *ipart);`	`math.h`	Splits a `double` into integer and fractional parts, both with the same sign as `x`.
`void perror(const char *s);`	`stdio.h`	Prints to the `stderr` stream the system error message for the last library routine that produced the error.
`double pow(double x, double y);`	`math.h`	Calculates x^y.
`int printf(const char *format [, argument, ...]);`	`stdio.h`	Writes formatted output to `stdout`.
`int putc(int c, FILE *stream);`	`stdio.h`	Outputs a character to `stream`.
`int putchar(int c);`	`stdio.h`	Outputs a character to `stdout`.
`int puts(const char *s);`	`stdio.h`	Outputs a string to `stdout`; terminates output by a newline character.
`void qsort(void *base, size_t nelem, size_t width, int (*fcmp)(const void *, const void *));`	`stdlib.h`	Sorts array `base` using the quicksort algorithm based on the comparison function pointed to by `fcmp`.
`int raise(int sig);`	`signal.h`	Sends a signal of type `sig` to the program. If the program has installed a signal handler for the signal type specified by `sig`, that handler will be executed.

Syntax	Header File	Purpose
`int rand(void);`	`stdlib.h`	Returns successive pseudorandom numbers in the range from 0 to `RAND_MAX` (constant defined in `stdlib.h`).
`void *realloc(void *block, size_t size);`	`stdlib.h`	Attempts to shrink or expand the previously allocated block to `size` bytes, copying the contents to a new location if necessary.
`int remove (const char *filename);`	`stdio.h`	Deletes the file specified by `filename`.
`int rename(const char *oldname, const char *newname);`	`stdio.h`	Changes the name of a file from `oldname` to `newname`.
`void rewind(FILE *stream);`	`stdio.h`	Repositions a file pointer to the beginning of a stream.
`int scanf(const char *format [, address, ...]);`	`stdio.h`	Scans and formats input from `stdin` stream.
`void setbuf(FILE *stream, char *buf);`	`stdio.h`	Causes the buffer `buf` to be used for I/O buffering instead of an automatically allocated buffer.
`int setjmp(jmp_buf jmpb);`	`setjmp.h`	Captures the complete task state in `jmpb` and returns 0.
`char *setlocale(int category, char *locale);`	`locale.h`	Selects a locale; if selection is successful, returns a string indicating the locale that was in effect prior to invoking the function.
`int setvbuf(FILE *stream, char *buf, int type, size_t size);`	`stdio.h`	Causes the buffer `buf` to be used for I/O buffering instead of an automatically allocated buffer. The `type` parameter may be `_IOFBF` (fully buffered), `_IOLBF` (line buffered), or `_IONBF` (unbuffered).
`void (*signal(int sig, void (*func)(int sig))) (int);`	`signal.h`	Specifies signal-handling actions.
`double sin(double x);`	`math.h`	Calculates the sine of the input value (angles in radians).
`double sinh(double x);`	`math.h`	Calculates hyperbolic sine.
`int sprintf(char *buffer, const char *format [, argument, ...]);`	`stdio.h`	Writes formatted output to a string.
`double sqrt(double x);`	`math.h`	Calculates the positive square root of a nonnegative input value.

Syntax	Header File	Purpose
`void srand(unsigned int seed);`	`stdlib.h`	Initializes random number generator.
`int sscanf(const char *buffer, const char *format [, address, ...]);`	`stdio.h`	Scans and formats input from a string.
`char *strcat(char *dest, const char *src);`	`string.h`	Appends a copy of `src` to the end of `dest`; returns `dest`.
`char *strchr(const char *s, int c);`	`string.h`	Returns a pointer to the first occurrence of the character `c` in the string `s` (or null).
`int strcmp(const char *s1, const char *s2);`	`string.h`	Compares one string to another; return value < 0 means `s1` less than `s2`, value = 0 means same as, and value > 0 means greater than.
`int strcoll(const char *s1, const char *s2);`	`string.h`	Compares two strings according to the collating sequence set by `setlocale`; return value < 0 means `s1` less than `s2`, value = 0 means same as, and value > 0 means greater than.
`char *strcpy(char *dest, const char *src);`	`string.h`	Copies string `src` to `dest`, stopping after copying the terminating null character; returns `dest`.
`size_t strcspn(const char *s1, const char *s2);`	`string.h`	Returns the length of the initial segment of string `s1` that consists entirely of characters *not* from string `s2`.
`char *strerror(int errnum);`	`string.h`	Returns a pointer to an error message string associated with `errnum`.
`size_t strftime(char *s, size_t maxsize, const char *fmt, const struct tm *t);`	`time.h`	Formats time for output according to the `fmt` specifications; returns the number of characters placed into `s`.
`size_t strlen(const char *s);`	`string.h`	Returns the number of characters in `s`, not counting the null terminating character.
`char *strncat(char *dest, const char *src, size_t maxlen);`	`string.h`	Copies at most `maxlen` characters of `src` to the end of `dest` and appends a null character.
`int strncmp(const char *s1, const char *s2, size_t maxlen);`	`string.h`	Compares a portion (no more than `maxlen` characters) of one string to a portion of another; return value < 0 means portion of `s1` less than portion of `s2`, value = 0 means same as, and value > 0 means greater than.

Syntax	Header File	Purpose
`char *strncpy(char *dest, const char *src, size_t maxlen);`	`string.h`	Copies up to `maxlen` characters from `src` into `dest`, truncating or null-padding `dest` (which might not be null-terminated).
`char *strpbrk(const char *s1, const char *s2);`	`string.h`	Returns a pointer to the first occurrence in `s1` of any of the characters in `s2` (or returns null).
`char *strrchr(const char *s, int c);`	`string.h`	Returns a pointer to the last occurrence of the character `c` in string `s` (or returns null).
`size_t strspn(const char *s1, const char *s2);`	`string.h`	Returns the length of the initial segment of `s1` that consists entirely of characters from `s2`.
`char *strstr(const char *s1, const char *s2);`	`string.h`	Scans `s1` for the first occurrence of the substring `s2`.
`double strtod(const char *s, char **endptr);`	`stdlib.h`	Converts string `s` to a `double` value; if `endptr` is not null, it sets `*endptr` to point to the character that stopped the scan.
`char *strtok(char *s1, const char *s2);`	`string.h`	Searches `s1` for tokens, which are separated by delimiters defined in `s2`.
`long int strtol(const char *s, char **endptr, int radix);`	`stdlib.h`	Converts a string `s` to a `long int` value in the given radix; if `endptr` is not null, it sets `*endptr` to point to the character that stopped the scan.
`unsigned long int strtoul (const char *s, char **endptr, int radix);`	`stdlib.h`	Converts a string `s` to an `unsigned long int` value in the given radix; if `endptr` is not null, it sets `*endptr` to point to the character that stopped the scan.
`size_t strxfrm(char *s1, const char *s2, size_t n);`	`string.h`	Transforms strings so that `strcmp` of new strings has the same result as `strcoll` of original strings. Changes up to `n` characters of `s1`.
`int system (const char *command);`	`stdlib.h`	Executes an operating system command.
`double tan(double x);`	`math.h`	Calculates the tangent of an angle specified in radians.
`double tanh(double x);`	`math.h`	Calculates the hyperbolic tangent.
`time_t time(time_t *timer);`	`time.h`	Gives the current time, in seconds, elapsed since 00:00:00 GMT, January 1, 1970, and stores that value in the location pointed to by `timer`.

Library Facilities Alphabetized by Name

Syntax	Header File	Purpose
`FILE *tmpfile(void);`	`stdio.h`	Creates a temporary binary file and opens it for update.
`char *tmpnam(char *s);`	`stdio.h`	Creates a unique file name.
`int tolower(int ch);`	`ctype.h`	Converts an integer `ch` to its lowercase value. Non-uppercase letter values are returned unchanged.
`int toupper(int ch);`	`ctype.h`	Converts an integer `ch` to its uppercase value. Non-lowercase letter values are returned unchanged.
`int ungetc(int c, FILE *stream);`	`stdio.h`	Pushes a character back into an open input stream.

`void va_start(va_list ap, lastfix);`
type `va_arg(va_list ap, type);`
`void va_end(va_list ap);`
	`stdarg.h`	Macros for implementing a variable argument list.

`int vfprintf(FILE *stream, const char *format, va_list arglist);`	`stdio.h`	Writes formatted output to a stream: Writes the values of a series of arguments, applying the format specifiers from the format string.
`int vprintf(const char *format, va_list arglist);`	`stdio.h`	Writes formatted output to `stdout`: Writes the values of a series of arguments, applying the format specifiers from the format string.
`int vsprintf(char *buffer, const char *format, va_list arglist);`	`stdio.h`	Writes formatted output to a string: Writes the values of a series of arguments, applying the format specifiers from the format string.
`size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);`	`stdlib.h`	Converts a string of wide characters to a string of multibyte characters (changes no more than `n` bytes of `s`).
`int wctomb(char *s, wchar_t wchar);`	`stdlib.h`	Stores in `s` the multibyte representation of wide character `wchar`.

Skrivbara ASCII-tecken

I ASCII-tabellen nedan är inte teckenkoderna 0-31 medtagna, ty dessa är styrtecken och inte skrivbara i vanlig mening (se ASCII-styrkoder).

Vilka tecken som du kommer att se i nedanstående tabell med 8-bitars (s k utvidgad ASCII) beror faktiskt på vilket operativsystem och vilket verktyg du använder dig av. Det är nämligen så att det är enbart de "ursprungliga" 7-bitars ASCII-koderna, dvs tecknen med ASCII-koderna 0-127, som är gemensamma för alla s k *teckentabeller*. Resten (koderna 128-255) finns det många olika varianter av.

Om du tittar på ASCII-tabellen med en grafiskt baserad webbläsare, som Netscape eller Internet Explorer, då är det de tecken som ingår i *Latin-1* (ISO 8859-1) som du ser. Denna teckenuppsättning är idag också den vanligaste.

Om du i stället studerar HTML-koden från MS-DOS, då blir det tecknen som ingår i någon variant av IBM:s utvidgade ASCII som ses (bortsett från några tecken som har specialbeydelse i HTML och som därför angetts med specialkoder för att inte feltolkas, detta gäller t ex `&` för att visa &-tecknet).

Om du är riktigt uppmärksam (och använder en vanlig webbläsare), då kommer förmodligen en del tecken saknas i tabellen, eller rättare sagt det visas små rutor (eller frågetecken) i stället för vissa tecken. Detta beror på att teckenkoderna 128-159 inte används i *Latin-1* och 127 är en styrkod (och därför inte visbart).

För att ytterligare komplicerat till det: Det kan mycket väl hända att du ändå ser tecken som tillhör detta intervall. Detta gäller t ex Windows 98/NT/2000/XP, vilket beror på att Microsoft utökat *Latin 1* med dessa tecken. Detta gäller även Linux, om du använder nyare webbläsare (Netscape 6.x, Opera, Konqueror). Det där med att hålla sig till standarder är, som du förstår, inte så lätt.

kod	tkn	kod	tkn	kod	tkn	kod	tkn	kod	tkn	kod	tkn	kod	tkn
32		64	@	96	`	128	€	160		192	À	224	à
33	!	65	A	97	a	129	□	161	¡	193	Á	225	á
34	"	66	B	98	b	130	,	162	¢	194	Â	226	â
35	#	67	C	99	c	131	*f*	163	£	195	Ã	227	ã
36	$	68	D	100	d	132	„	164	¤	196	Ä	228	ä
37	%	69	E	101	e	133	...	165	¥	197	Å	229	å
38	&	70	F	102	f	134	†	166	¦	198	Æ	230	æ
39	'	71	G	103	g	135	‡	167	§	199	Ç	231	ç
40	(	72	H	104	h	136	ˆ	168	¨	200	È	232	è
41	)	73	I	105	i	137	‰	169	©	201	É	233	é
42	*	74	J	106	j	138	Š	170	ª	202	Ê	234	ê
43	+	75	K	107	k	139	‹	171	«	203	Ë	235	ë

44	,	76	L	108	l	140	Œ	172	¬	204	Ì	236	ì
45	-	77	M	109	m	141	□	173		205	Í	237	í
46	.	78	N	110	n	142	Ž	174	®	206	Î	238	î
47	/	79	O	111	o	143	□	175	¯	207	Ï	239	ï
48	0	80	P	112	p	144	□	176	°	208	Đ	240	ð
49	1	81	Q	113	q	145	‘	177	±	209	Ñ	241	ñ
50	2	82	R	114	r	146	’	178	²	210	Ò	242	ò
51	3	83	S	115	s	147	“	179	³	211	Ó	243	ó
52	4	84	T	116	t	148	”	180	´	212	Ô	244	ô
53	5	85	U	117	u	149		181	µ	213	Õ	245	õ
54	6	86	V	118	v	150	–	182	¶	214	Ö	246	ö
55	7	87	W	119	w	151	—	183	·	215	×	247	÷
56	8	88	X	120	x	152	˜	184	¸	216	Ø	248	ø
57	9	89	Y	121	y	153	™	185	¹	217	Ù	249	ù
58	:	90	Z	122	z	154	š	186	º	218	Ú	250	ú
59	;	91	[	123	{	155	›	187	»	219	Û	251	û
60	<	92	\	124	|	156	œ	188	¼	220	Ü	252	ü
61	=	93	]	125	}	157	□	189	½	221	Ý	253	ý
62	>	94	^	126	~	158	ž	190	¾	222	Þ	254	þ
63	?	95	_	127	□	159	Ÿ	191	¿	223	ß	255	ÿ

Skrivbara IBM-ASCII-tecken

Denna ASCII-tabell visar den IBM:s utvidgade ASCII (den flerspråkiga teckentabellen 850) som används i MS-DOS (de tecken som är specifika för IBM-ASCII är de med teckenkoder över 127).

Teckenkoderna 0-31 är inte medtagna, ty dessa är styrtecken och normalt inte skrivbara, även om åtskilliga av dem faktiskt tilldelats symboler i IBM-ASCII.

Anmärkning: För att DOS-tecknen ska visas korrekt med en vanlig webbläsare (som kan visa Latin 1, men inte IBM-ASCII), har de enskilda tecknen handkodats med hjälp av *Latin-1* (ISO 8859-1, som är standard i Windows och Unix). Korrekt återgivning av DOS-tecknen förutsätter därför att webbläsaren använder denna teckenuppsättning. De speciella grafiska tecknen i IBM-ASCII (hörn, sidor, block) saknar dock motsvarigheter i Latin 1, varför dessa inte kan visas. Däremot finns alla bokstäver (inkl grekiska) med.

kod	tkn	kod	tkn	kod	tkn	kod	tkn	kod	tkn	kod	tkn	kod	tkn
32		64	@	96	`	128	Ç	160	á	192	+	224	Ó
33	!	65	A	97	a	129	ü	161	í	193	-	225	ß
34	"	66	B	98	b	130	é	162	ó	194	-	226	Ô
35	#	67	C	99	c	131	â	163	ú	195	+	227	Ò
36	$	68	D	100	d	132	ä	164	ñ	196	-	228	õ
37	%	69	E	101	e	133	à	165	Ñ	197	+	229	Õ
38	&	70	F	102	f	134	å	166	ª	198	ã	230	µ
39	'	71	G	103	g	135	ç	167	º	199	Ã	231	þ
40	(	72	H	104	h	136	ê	168	¿	200	+	232	Þ
41	)	73	I	105	i	137	ë	169	®	201	+	233	Ú
42	*	74	J	106	j	138	è	170	¬	202	-	234	Û
43	+	75	K	107	k	139	ï	171	½	203	-	235	Ù
44	,	76	L	108	l	140	î	172	¼	204	¦	236	ý
45	-	77	M	109	m	141	ì	173	¡	205	-	237	Ý
46	.	78	N	110	n	142	Ä	174	«	206	+	238	¯
47	/	79	O	111	o	143	Å	175	»	207	¤	239	´
48	0	80	P	112	p	144	É	176	_	208	ð	240	
49	1	81	Q	113	q	145	æ	177	_	209	Ð	241	±
50	2	82	R	114	r	146	Æ	178	_	210	Ê	242	_
51	3	83	S	115	s	147	ô	179	¦	211	Ë	243	¾
52	4	84	T	116	t	148	ö	180	¦	212	È	244	¶

53	**5**	85	**U**	117	**u**	149	**ò**	181	**Á**	213	**i**	245	§
54	**6**	86	**V**	118	**v**	150	**û**	182	**Â**	214	**Í**	246	÷
55	**7**	87	**W**	119	**w**	151	**ù**	183	**À**	215	**Î**	247	¸
56	**8**	88	**X**	120	**x**	152	**ÿ**	184	**©**	216	**Ï**	248	°
57	**9**	89	**Y**	121	**y**	153	**Ö**	185	**¦**	217	**+**	249	¨
58	**:**	90	**Z**	122	**z**	154	**Ü**	186	**¦**	218	**+**	250	·
59	**;**	91	**[**	123	**{**	155	**ø**	187	**+**	219	**_**	251	_
60	**<**	92	****	124	**|**	156	**£**	188	**+**	220	**_**	252	³
61	**=**	93	**]**	125	**}**	157	**Ø**	189	**¢**	221	**_**	253	²
62	**>**	94	**^**	126	**~**	158	**×**	190	**¥**	222	**Ì**	254	_
63	**?**	95	**_**	127	□	159	***f***	191	**+**	223	**_**	255	

Svenska tecken i DOS som styrtecken i C

Å	**\217**
å	**\206**
Ä	**\216**
ä	**\204**
Ö	**\231**
ö	**\224**

The PC Keyboard

The computer keyboard produces codes that are associated with letters and symbols. One key, however, can produce a different set of codes when you press other keys at the same time. For example, the A key normally produces the letter "a" (ASCII code 97), but when pressed along with the SHIFT key, it produces the letter "A" (ASCII code 65). Two other keys—CTRL and ALT—produce even more codes!

Some keys, such as the function keys (F1 through F10), produce two codes: a scan code (≠0) and another code that indicates the key pressed.

The following table lists the keys on the PC keyboard and the codes they return.

Key	Normal		SHIFT		CTRL		ALT	
F1	0	59	0	84	0	94	0	104
F2	0	60	0	85	0	95	0	105
F3	0	61	0	86	0	96	0	106
F4	0	62	0	87	0	97	0	107
F5	0	63	0	88	0	98	0	108
F6	0	64	0	89	0	99	0	109
F7	0	65	0	90	0	100	0	110
F8	0	66	0	91	0	101	0	111
F9	0	67	0	92	0	102	0	112
F10	0	68	0	93	0	103	0	113
←	0	75		52	0	115		none
→	0	77		54	0	116		none
↑	0	72		56		none		none
↓	0	80		50		none		none
HOME	0	71		55	0	119		none
END	0	79		49	0	117		none
PGUP	0	73		57	0	132		none
PGDN	0	81		51	0	118		none
INS	0	82		48		none		none
DEL	0	83		46	0	255		none
ESC		27		27		27		none
BACKSPACE		8		8		127		none
TAB		9	0	15		none		none
ENTER		13		13		10		none
A		97		65		1	0	30
B		98		66		2	0	48
C		99		67		3	0	46
D		100		68		4	0	32
E		101		69		5	0	18
F		102		70		6	0	33
G		103		71		7	0	34
H		104		72		8	0	35
I		105		73		9	0	23
J		106		74		10	0	36
K		107		75		11	0	37

Key	Normal	SHIFT	CTRL	ALT
L	108	76	12	0 38
M	109	77	13	0 50
N	110	78	14	0 49
O	111	79	15	0 24
P	112	80	16	0 25
Q	113	81	17	0 16
R	114	82	18	0 19
S	115	83	19	0 31
T	116	84	20	0 20
U	117	85	21	0 22
V	118	86	22	0 47
W	119	87	23	0 17
X	120	88	24	0 45
Y	121	89	25	0 21
Z	122	90	26	0 44
[	91	123	27	none
\	92	124	28	none
]	93	125	29	none
`	96	126	none	none
0	48	41	none	0 129
1	49	33	none	0 120
2	50	64	0 3	0 121
3	51	35	none	0 122
4	52	36	none	0 123
5	53	37	none	0 124
6	54	94	30	0 125
7	55	38	none	0 126
8	56	42	none	0 127
9	57	40	none	0 128
*	42	none	0 114	none
Keypad +	43	43	none	none
Keypad -	45	45	none	none
=	61	43	none	0 131
/	47	63	none	none
;	59	58	none	none
-	45	95	31	0 130

Sakregister