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 `double`-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 en så kallad *array*, efter det engelska ordet ”array” som ungefär betyder ”uppställd rad av likadana saker”. Arrayer kallas på svenska *vektorer* eller *indexerade variabler*, och ibland också *fält*. 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 *array* (eller *vektor* eller *indexerad variabel* eller *fält*[1]) 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

```
double 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;   /* Och sista elementet har index 4, inte 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.

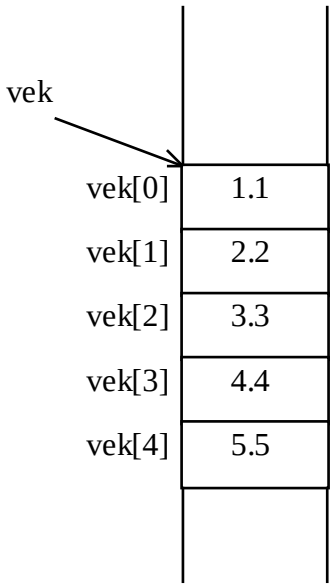
Ex: Skapa en vektor bestående av 5 reella element och initiera elementen med värdena 1.1, 2.2, 3.3, 4.4 resp. 5.5. Skriv sedan ut vektorn med ett element per rad.

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

for (i = 0; i < 5; i++)
   printf("%f\n", vek[i]);
```

1 Undvik ordet ”fält”, för det brukar ibland användas för en helt annan sak, nämligen delarna av en post.

När man definierar en vektorvariabel reserverar man utrymme i minnet för det angivna antalet element. Själva namnet på vektorn kommer i de flesta sammanhang i ett C-program att översättas till *adressen till minnesutrymmets början*, dvs *adressen till det första elementet*. 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.

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

for (i = 0; i < 5; 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:

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

for (i = 0; i < 5; 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>

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

   /* läs in data */
   for (i = 0; i < 10; i++)
   {
      printf("xdata[%d] = ", i);
      scanf("%lf", &xdata[i]);
   }

   /* sortera data */
   for (i = 0; i < 9; i++)
      for (j = i + 1; j < 10; j++)
         if (xdata[j] < xdata[i])
         {
            /* byt plats */
            temp = xdata[i];
            xdata[i] = xdata[j];
            xdata[j] = temp;
         }

   /* skriv data */
   for (i = 0; i < 10 ; i++)
   {
      if  (i % 5 == 0)
         printf("\n");
      else
         printf(" ");
      printf("%f", xdata[i]);
   }
   printf("\n");

   return 0;
}
```

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 */

int 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]);

   return 0;
}
```

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.

OBS! För det mesta brukar man i C skriva `for`-loopar på formen

```
for (i = 0; i < 6; i++) ...
```

när man vill göra något 6 gånger. (Ö*vning: Varför?*)
Här har vi i stället skrivit

```
for (i = 1; i <= 6; i++) ...
```

eftersom det var bättre just i det här fallet. (Ö*vning: Varför?*)

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>

int 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]);
   putchar('\n');

   return 0;
}
```

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*[2]*-tecknet, '\0'*, alltså det tecken som har teckenkoden 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.

2 På engelska kallas teckent för ”NUL”. (Notera: ”NUL”, inte ”NULL”.)

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

```
#include <stdio.h>

int 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 ACII-koden 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);

   return 0;
}
```

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>

int 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++;
   }
   putchar('\n');

   return 0;
}
```

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.

OBS! I riktiga program ska man egentligen aldrig använda `gets`, eftersom den alltså inte kontrollerar att inmatningen får plats i variabeln! Man kan få ett så kallat ”buffer overflow”. Även inläsning med `scanf` och formatangivelsen `%s` har samma problem. Det finns en bättre funktion som heter `fgets`, men den är lite krångligare att använda, så här nöjer vi oss med `gets`.

Elementen i en sträng är tecken och kan behandlas som vanliga teckenvariabler. Strängvariabeln däremot är en vektor. 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>

int main()
{
   double 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} };
   double radsum, totalsum = 0;
   int rad, kol;

   /* summera tabellen och skriv ut */
   for (rad = 0; rad < 3; rad++)
  {
      radsum = 0;

      /* summera raden */
      for (kol = 0; kol < 4; kol++)
         radsum += tabell[rad][kol];  /* OBS! Indexering */

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

   return 0;
}
```

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>

int main()
{
   char namnlista[10][30 + 1]; /* 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]);

   return 0;
}
```

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 *medlemmar*, *medlemsvariabler*, *fält* eller *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;
   double x;
} m1, m2;


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

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

Man kommer åt de enskilda medlemsvariablerna med *punktnotation*. Fältet 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;
   double x;
};

struct matdata m1, m2;
```

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

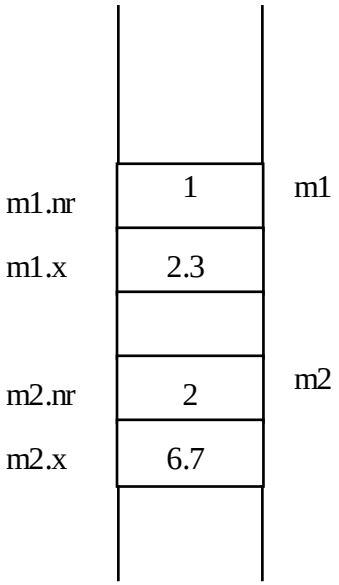
På samma sätt som för vektorer kan man vid variabeldefinitionen initiera sina poster med värden för respektive medlemsvariabel.

Ex: Skapa ovanstående poster med initiering istället för tilldelning.

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

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

När man definierar en postvariabel reserverar man utrymme i minnet för de angivna medlemsvariablerna. 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 medlemsvariabel i en post kan behandlas som en vanlig variabel av medlemsvariabelns 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 medlemsvariabel för medlemsvariabel.

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("%lf", &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 medlemsvariabeln i dessa poster, håller reda på största och minsta mätvärde samt avslutningsvis skriver ut största värdet, minsta värdet samt medelvärdet av alla posters mätvärden.

```
#include <stdio.h>

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

   int i, antal;
   double 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("%lf", &m.x);

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

   /* läs in resterande poster */
   for (i = 1; i < antal; i++)
   {
      printf("Ge postens nummer : ");
      scanf("%d", &m.nr);
      printf("Ge postens mätvärde : ");
      scanf("%lf", &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);

   return 0;
}
```

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[6 + 1];
   char agare[30 + 1];
   char marke[20 + 1];
   int arsmodell;
   double 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 medlemsdata
{
   int medlemnr;
   char namn[30 + 1];
   double avgift;
   char tel[15 + 1];
};

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

Medlemsvariablerna 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 ingår 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[11 + 1];
   char namn[30 + 1];
   char gatuadress[30 + 1];
   long postnr;
   char ortsnamn[20 + 1];
};


struct bilpost
{
   char regnr[6 + 1];
   struct agaredata agare;            /* OBS! */
   char marke[20 + 1];
   int arsmodell;
   double 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! Medlemsvariabler inuti en nästlad post kommer man åt med upprepad punktnotation.

Ett medlemsvariabel i en post kan som vi har sett vara en vektor eller en sträng. 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>

#define ANTAL_POSTER 20

int main()
{
   struct matdata
   {
      int nr;
      double temp;
   };

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

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

   /* sortera vektorn i stigande temperaturer */
   for (i = 0; i < ANTAL_POSTER - 1; i++)
      for (j = i + 1; j < ANTAL_POSTER; 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 < ANTAL_POSTER; i++)
      printf("Nr: %d Temp: %f\n", serie[i].nr, serie[i].temp);

   return 0;
}
```

OBS! Programmet arbetar med 20 stycken poster. Det betyder att talet 20 används på flera olika ställen i programmet: där arrayen skapas, där den initieras, där den sorteras och där den skrivs ut. Men i stället för att skriva talet 20 på fem olika ställen i programmet har vi definierat ett makro, ANTAL_POSTER, som sen används i fortsättningen. Det ger minst två fördelar.
(*Övning: Vilka två fördelar är det?*)

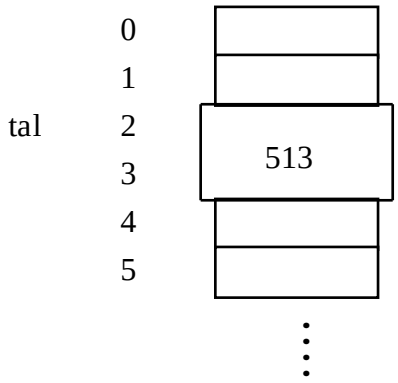
4.3 Adresser och pekare

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.

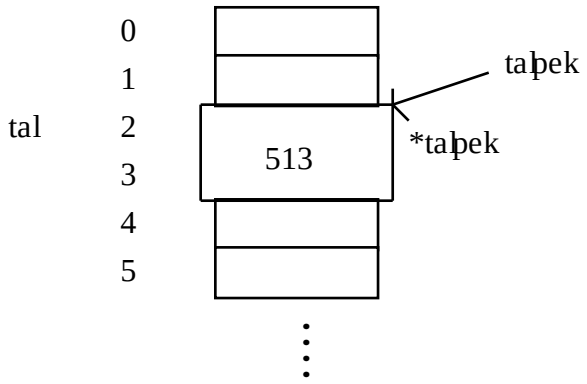
En *variabel av typen pekare kan tilldelas ett värde i form av en minnesadress*. Man säger att variabeln då pekar på en plats i minnet, därav namnet. En pekarvariabel i sig är en variabel av enkel typ eftersom den bara innehåller en heltalsadress. Varför pekare tas upp i samband med sammansatta datatyper,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

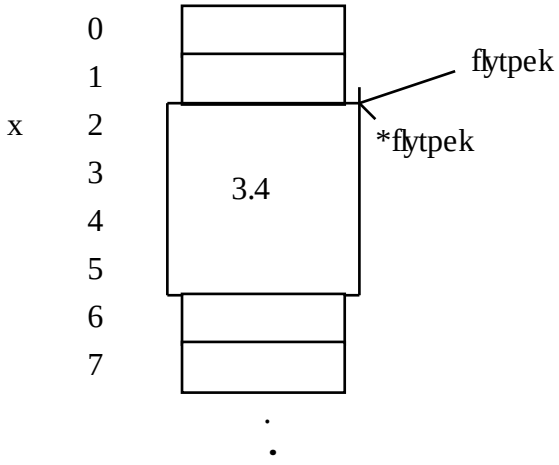
```
talpek++;
```

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.

```
double *flytpek;
double 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 double och i vårt sytem tar double 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:

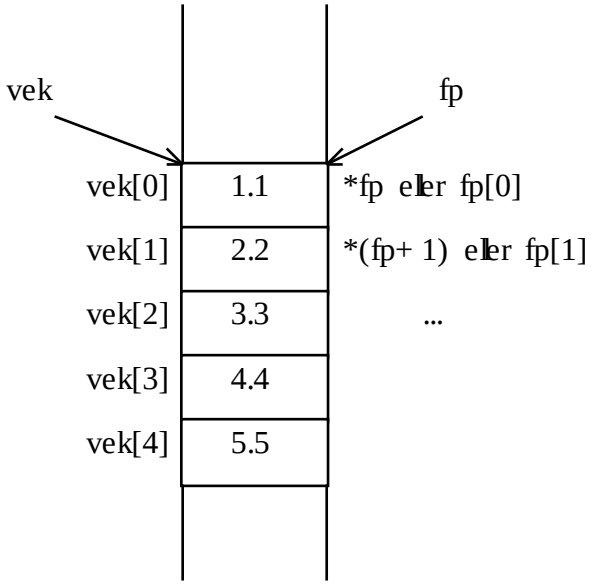
```
flytpek = NULL;
```

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.

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

double *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.