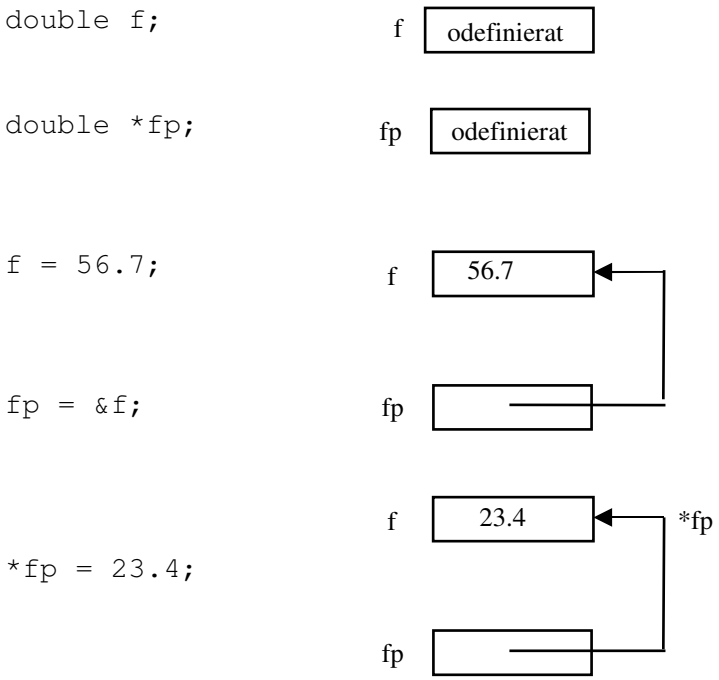
2 Pekare och dynamiska variabler.

När man definierar en variabel reserverar man samtidigt minne för variabelns värde. Detta minnesutrymme kommer man sedan åt med hjälp av variabelns namn. Definierar man en sträng med 80 tecken, *reserveras eller allokeras* det minne för 80 byte, oberoende av om hela minnesutrymmet utnyttjas eller ej. *Detta är en statisk modell för att allokera minne*. Det finns ett mer dynamiskt sätt. Man definierar först en vanlig variabel i form av *en pekare* och sedan, under körning, *när man vet hur mycket minne man behöver*, reserverar man minne som man *sätter pekaren att peka på*. Detta minne kommer man sedan åt genom att *avreferera* pekaren. Denna typ av minne kan man under körningen *avallokera*, när det ej behövs längre.

2.1 Pekare

Pekare är vanliga variabler som defineras till att peka på en speciell typ av data, eller mera generellt på vad som helst (med nyckelordet void). Pekarvariabler kan sedan ges värden i form av *adresser*.

Ex: Definiera en flyttalsvariabel och en pekarvariabel som kan peka på flyttal. Tilldela flyttalet värdet 56.7 och tilldela pekaren adressen för detta flyttal. Ändra sedan flyttalets värde med hjälp av pekaren till 23.4.

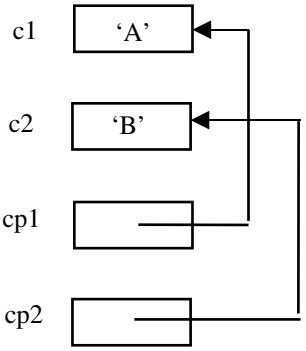


OBS! *Avreferering av pekaren med *fp*, för att komma åt det som pekaren pekar på.

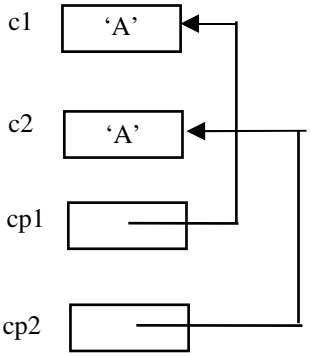
Man kan naturligtvis definiera pekare att peka på andra typer av data. Man kan till och med definiera pekare att peka på vilken typ av data som helst, nämligen void-pekare.

Ex: Definiera två tecken samt två pekare som sätts att peka på dessa tecken. Tilldela sedan *med hjälp av pekaren* det första tecknet till det andra och därefter det första pekarvärdet till det andra pekarvärdet.

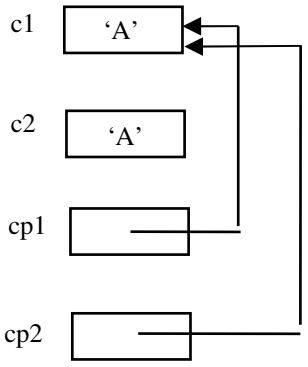
```
char c1 = 'A';

char c2 = 'B';

char *cp1 = &c1;

char *cp2 = &c2;
```




```
/* Tilldela tecken */

*cp2 = *cp1;
```



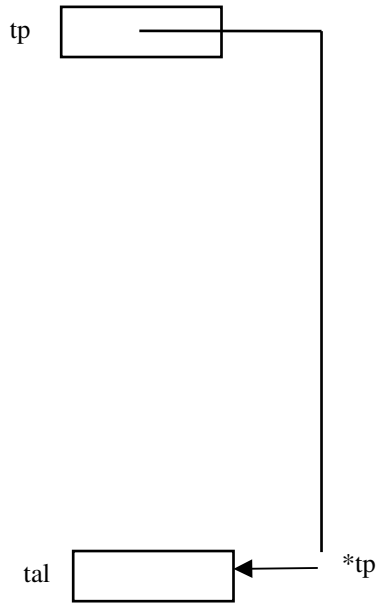

```
/* Tilldela pekarvärden */

cp2 = cp1;
```



Pekare använder man exempelvis då man ska *förändra värdet* på en aktuell parameter med en funktion.

Ex: Skriv ett program som läser in ett heltalsvärde och skriver ut talet kvadrerat. Kvadreringen ska utföras av en void-funktion som tar talet som parameter och förändrar talet genom att kvadrera det.

```
#include <stdio.h>

void kvadrera(int *tp)
{
   *tp = (*tp) * (*tp);
}

int main()
{
   int tal;

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

   kvadrera(&tal);

   printf("Talet kvadrerat : %d\n", tal);
   return 0;
}
```



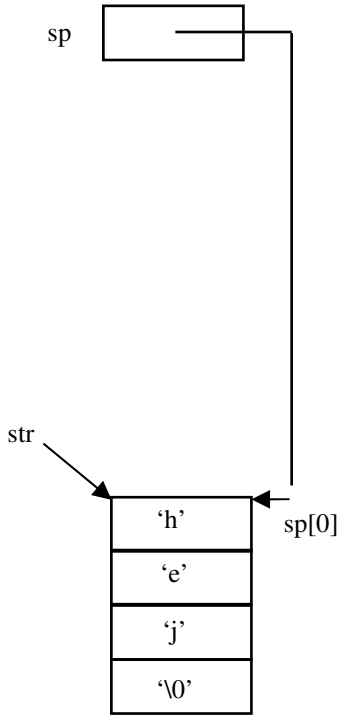
OBS! Vid anropet *skickas adressen för tal* till funktionen som *tar emot med en pekare*. I funktionen har man nu tillgång till den *aktuella parameterns minnesutrymme via pekaren.*

OBS! Då man, som ovan, enbart ska returnera ett enda värde är det bättre att använda en int-funktion som returnerar det kvadrerade talet. Då man använder funktionsvärdet som informationsöverförare slipper man också ifrån nackdelen att den aktuella parameterns värde förändras efter anropet. Denna typ av *sidoeffekt är oftast ej önskvärd*.

Vektorer och strängar är inga vanliga variabler utan *kan hellre ses som konstanta adresser* eller *konstanta pekarvärden*. Har man vektorer eller strängar som aktuella parametrar ska man alltid ta emot dessa *formellt som pekare*.

Ex: Skriv ett program som läser in en sträng och som byter ut alla stora bokstäver till motsvarande små.

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

void smastr(char *sp)
{
   int i;

   for (i = 0; i < strlen(sp); i++)
      if ( sp[i] >= 'A' && sp[i] <= 'Z')
         sp[i] = sp[i] + 32;
}



int main()
{
   char str[80];

   printf("Ge sträng : ");
   gets(str);


   smastr(str);


   printf("Strängen : %s \n", str);
   return 0;
}
```



OBS! *Indexering* av pekaren sp som egentligen är en *pekaruppräkning*. Exempelvis kommer man åt det andra tecknet i strängen med indexeringen sp[1] eller alternativt med avrefereringen *(sp + 1).

Pekaruppräkningen *lägger automatiskt till det antal byte som gäller för den datatyp* som den aktuella pekaren är definierad att peka på.

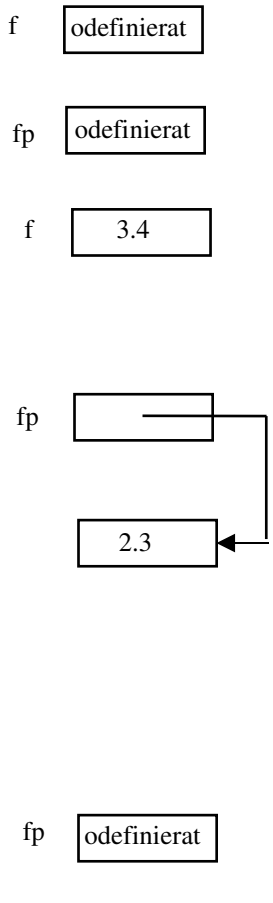
(Men kom ihåg att man *egentligen bör undvika funktionen gets*, eftersom den inte kollar att den inlästa strängen får plats i variabeln! Använd *fgets* i stället. Dessutom vill man kanske titta lite på funktionen *tolower*, som finns i standardbiblioteket.)

2.2 Dynamiska variabler

För variabler som definieras på vanligt sätt, exempelvis inuti en funktion som main, reserveras det minnesutrymme för *automatiskt direkt vid definitionen*. Det finns ett annat sätt att reservera eller allokera minnesutrymme, där man som programkonstruktör *själv bestämmer när och hur* mycket minne som ska reserveras. Dessa variabler eller minnesutrymmen som är mer flexibla, kallas *dynamiska variabler*. När det gäller att ge tillbaka det reserverade minnesutrymmet gäller motsvarande. Vanliga variabler avallokeras *alltid vid funktionens slut,* medan de dynamiska *avallokeras av programmet* när de ej behövs längre.

Ex: Allokera två reella variabler den första automatiskt och den andra dynamiskt. Tilldela dessa sedan värden samt skriv ut summan av dessa värden.

```
double f ;

double *fp;

f = 3.4;


/* Allokera minne */
fp = malloc(sizeof(double));


*fp = 2.3;

printf("Summan = %f", f + (*fp));


/* Avallokera minne*/
free(fp);
```



OBS! För att allokera minne dynamiskt måste man *först definiera en pekare,* som man sedan sätter att peka på det allokerade minnet. Eftersom detta *minne är anonymt, dvs saknar namn*, måste *man alltid använda denna pekare* för att komma åt detta minne.

OBS! Funktionen malloc finns i stdlib.h. Den tar antalet byte som ska allokeras som parameter och returnerar en void-pekare. Denna pekare omvandlas ovan automatiskt till en double-pekare vid tilldelningen. Vissa kompilatorer *kräver en explicit* typomvandling enligt:

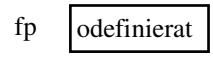
```
fp = (double *)malloc(sizeof(double));
```

Efter avallokeringen eller frigörandet av minne kommer pekaren att ha ett odefinierat värde, vilket innebär att den kan *peka på ett förbjudet minnesområde*. Därför är det alltid säkrast att se till att *pekare har väl definierade värden* då man skapar dem eller då man frigör minne som de pekar på. Det finns ett sådant väl definierat konstant värde, nämligen värdet NULL, som också finns i stdlib.h.

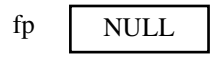
I exemplet ovan ska man använda detta NULL-värde, då man definierar pekarvariabeln och då man avallokerar minne enligt:

```
double *fp = NULL;
```


```
...........
...........

/* Avallokera minne*/

free(fp);
```




```
fp = NULL;
```



Funktionen malloc som allokerar minne *returnerar ett pekarvärde* till det allokerade minnet. Misslyckas allokeringen av minne, vilket exempelvis inträffar då minnet är slut*, returnerar malloc värdet NULL*. Detta kan man testa på efter allokeringen enligt:

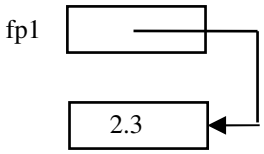
```
fp = malloc(sizeof(double));
if (fp == NULL)
   /* Kunde ej allokera */;
else
   /* Gick bra */;
```

Har man odefinierade pekare i sitt program är det lätt att *man skriver över väsentliga* delar av minnet och datorn hänger sig. Se till att alltid ha väl definierade värden på alla pekare i programmet. Väl definierade pekare får man genom att tilldela dessa värdet NULL eller tilldela dessa adresser för variabler eller dynamiskt allokera minne.

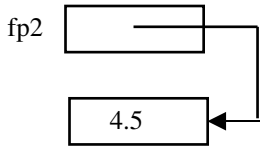
Regel : Se alltid till att ha väl definierade pekare i programmet. Har man inget annat värde, ger man pekarna värdet NULL.

Man kan naturligtvis tilldela en pekare värdet av en annan pekare. Här bör man också se upp eftersom man först tappar det minnesutrymme som den vänstra pekaren pekar på och sedan får två pekare att peka på samma minnesutrymme.

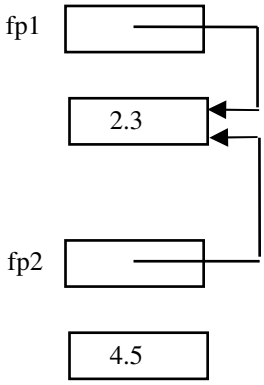
```
/* Allokera minne */

fp1 = malloc(sizeof(double));

*fp1 = 2.3;
```




```
fp2 = malloc(sizeof(double));

*fp2 = 4.5;
```




```
/* Pekartilldelning */

fp2 = fp1;
```



OBS! Minnesutrymmet som innehåller talet 4.5 har man *tappat bort*. Man kan inte komma åt detta mer under programmets gång. Minnet är dock fortfarande upptaget. *Detta är ett fel som medför att programmet äter minne*.

OBS! Frigör man minne, som två pekare pekar på, med hjälp av den ena pekaren som man sedan NULL-ställer har man fortfarande kvar en pekare som pekar på *otillåtet* (redan avallokerat) minne.

OBS! Skillnaden mellan tilldelning av reella tal

```
*fp2 = *fp1;
```

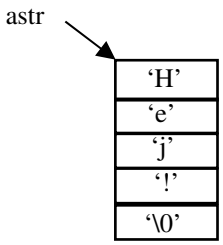
och mellan pekare

```
fp2 = fp1;
```

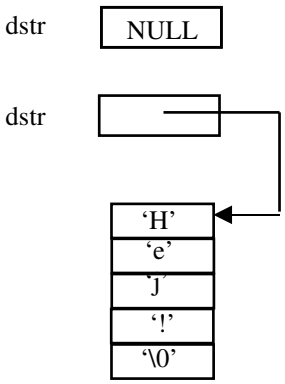
Man kan naturligtvis allokera minne dynamiskt även för sammansatta datatyper som strängar, vektorer och poster.

Ex: Allokera två strängar som maximalt kan innehålla 4 tecken (plus strängavslutningstecknet). Den första ska allokeras automatiskt och initieras. Den andra ska allokeras dynamiskt och sedan ska den första strängens värde kopieras över till denna. Slutligen skrivs den andra strängen ut på skärmen.

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

int main()
{
   char astr[5] = "Hej!";
```




```
   char *dstr = NULL;
```




```
   dstr = malloc(5*sizeof(char));

   strcpy(dstr, astr);

   puts(dstr);

   free(dstr);
```


```
   dstr = NULL;
}
```

OBS! Istället för att använda malloc kan man använda calloc enligt:

```
dstr = calloc(5, sizeof(char));
```

Här nollställs också minnesutrymmet automatiskt.

(Egentligen är sizeof(char) onödigt, för det är alltid 1, enligt C-standarden. Storlekar mäts alltså egentligen i char-platser, och inte i bytes! Men man kan ha med det för tydlighets skull.)

Ex: Skriv ett program som läser in hur många reella tal som ska läsas in till en vektor, allokerar utrymme för denna vektor dynamiskt samt läser in värden och skriver ut medelvärdet.

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

int main()
{
   int nr, i;
   double *vek = NULL, sum = 0.0;

   /* Läs in antal element */
   printf(”Ge antalet element : ”);
   scanf(”%d”, &nr);

   /* Allokera vektor med nr st element */
   vek = calloc(nr, sizeof(double));

   /* Läs in vektorelementen */
   for (i = 0; i < nr; i++)
   {
      printf(”Ge vek[%d]: ”, i);
      scanf(”%lf”, &vek[i]);
   }

   /* Addera elementen i vektorn */
   for (i = 0; i < nr; i++)
      sum += vek[i];

   /* Skriv ut medelvärdet */
   printf(”Medelvärdet = %f\n”, sum / nr);

   /* Frigör minne */
   free(vek);
   vek = NULL;

   return 0;
}
```

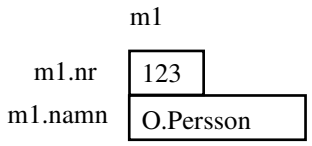
Här kan man inte skapa vektorn automatiskt eftersom man inte vet hur många element man ska ha i vektorn innan programmet körs. Därför definierar man först en pekare och, när man vet hur många element vektorn ska innehålla, skapar man dynamiskt utrymme för denna vektor.

Man kunde naturligtvis ha tagit till lite extra och definierat en vektor automatiskt med exempelvis 1000 element. Den dynamiska modellen sparar dock minne eftersom man bara allokerar upp så mycket minne som man för tillfället verkligen behöver.

Ex: Skapa två poster för medlemmar med medlemsnummer och namn. Den första ska skapas automatiskt (statiskt) och den andra dynamiskt. Initiera den första posten med lämpliga värden samt tilldela dessa sedan till den andra posten och skriv ut den andra postens termer.

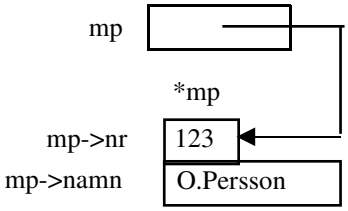
```
struct medlem
{
   int nr;
   char namn[30];
};
```

```
struct medlem m1 = {123, ”O.Persson”};
```




```
struct medlem *mp = NULL;
```


```
mp = malloc(sizeof(struct medlem));

*mp = m1;
```




```
printf(”Nummer : %d\n”, mp->nr);
printf(”Namn : %s\n”, mp->namn);
```

OBS! För att avreferera hela posten används på vanligt sätt *mp. För att komma åt enskilda termer i posten, som exempelvis medlemsnummer, kan man skriva

```
(*mp).nr
```

eller kortare med piloperatorn

```
mp->nr.
```

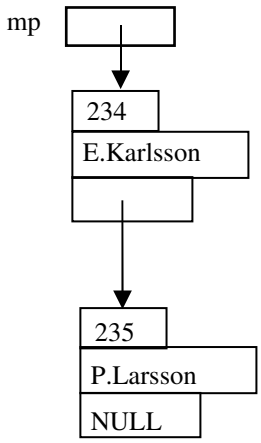
Poster kan länkas ihop till en kedja i minnet om man sätter in en extra term i posten i form av en pekare till nästa post.

Ex: Skapa två poster av medlemmar som är ihoplänkade så att den första pekar på den andra.

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

```
struct medlem *mp = NULL;
```

```
mp = malloc(sizeof(struct medlem));

mp->nr = 234;
strcpy(mp->namn, "E.Karlsson");

mp->next = malloc(sizeof(struct medlem));

mp->next->nr = 235;
strcpy(mp->next->namn, "P.Larsson");
mp->next->next = NULL;
```



OBS! Strängkopiering med strcpy.

OBS! Termen next som är en pekare till samma posttyp, som den ingår i. Ovanstående definition är tillåten även om man tycker att struct medlem ej är definierad innan man använder den för att definiera next.

OBS! Har man en pekare i sin post som kan sättas att peka på en ny post av samma typ kan man *länka ihop posterna till en kedja*. För att detta ska fungera för flera poster måste man dock använda någon annan metod än ovanstående, något begränsade metod.

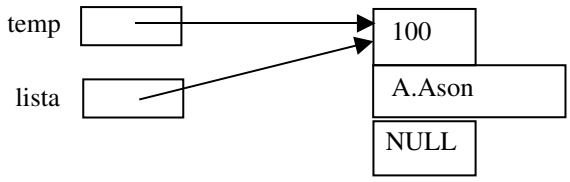
2.3 Länkade strukturer

För att avbilda datastrukturer som *köer, träd mm på ett dynamiskt sätt,* använder man sig av poster med pekare som i sin tur pekar på nästa post osv. Man behöver aldrig allokera mer minne än vad som *för tillfället behövs* för exempelvis en kö. När det dyker upp en ny person som ska stoppas in i kön, allokerar man bara upp nytt minne och placerar in personen i kön.

Ex: Skissa på ett program som läser in data till medlemsposter innehållande nummer och namn och länkar ihop posterna med hjälp av en pekare till nästa post.

```
/* Definiera data */

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

struct medlem *lista = NULL, *temp = NULL;


/* Skapa den första medlemmen */

temp = malloc(sizeof(struct medlem));
temp->nr = 100;
strcpy(temp->namn, "A.Ason");
temp->next = NULL;


/* Sätt lista att peka på den första posten */

lista = temp;
```

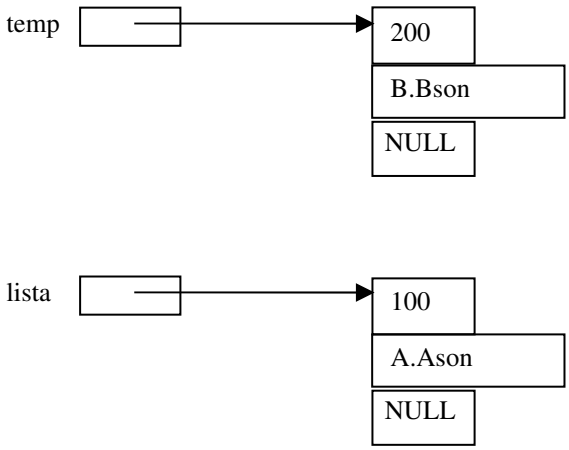
Läget är följande:



Eftersom man nu har två pekare som pekar på samma post kan man fortsätta med att allokera upp nytt minne för nästa post med hjälp av pekaren temp. Eftersom pekaren lista finns, tappar man ej bort den första posten.

```
/* Skapa en ny medlem */

temp = malloc(sizeof(struct medlem));
temp->nr = 200;
strcpy(temp->namn, ”B.Bson”);
temp->next = NULL;
```

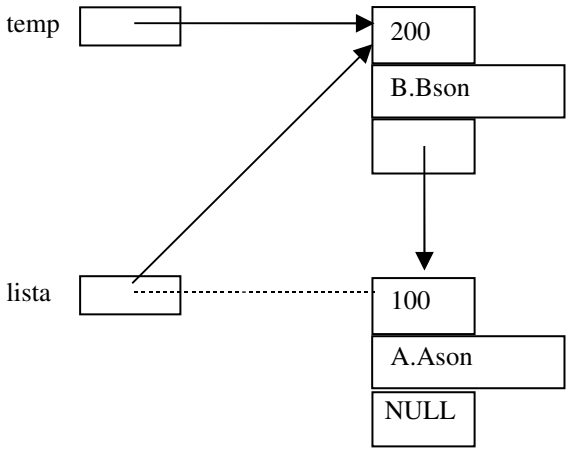
Läget är nu följande:



Hur ska man göra för att länka ihop listan?

```
/* Länka ihop */

temp->next = lista;
lista = temp;
```

Nu har listan länkats ihop enligt:

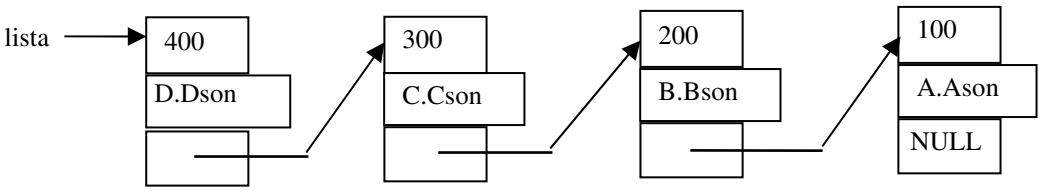


Ovanstående länkade struktur blir en lista av samma typ som en tallriksstapel. Man lägger på tallrikar på varandra och kommer bara åt den översta direkt. Man kan stoppa in nya länkar (tallrikar) i listan med en loop, där man läser in data enligt:

```
/* Skapa en länkad medlem */

/* Läs nytt medlemsnummer */
printf(”Ge nr ( Avslut 0 ) : ”);
scanf(”%d”, &tal);
while ( tal != 0 )
{
   /* Skapa en ny medlem */
   temp = malloc(sizeof(struct medlem));
   temp->nr = tal;
   getchar();
   printf(”Ge namn : ”);
   gets(temp->namn);

   /* Länka ihop med föregående */
   temp->next = lista;
   lista = temp;

   /* Läs nytt medlemsnummer */
   printf(”Ge nr ( Avslut 0 ) : ”);
   scanf(”%d”, &tal);
}
```

OBS! Ihoplänkningen som även kan användas för en tom lista om lista *initierats till NULL.*

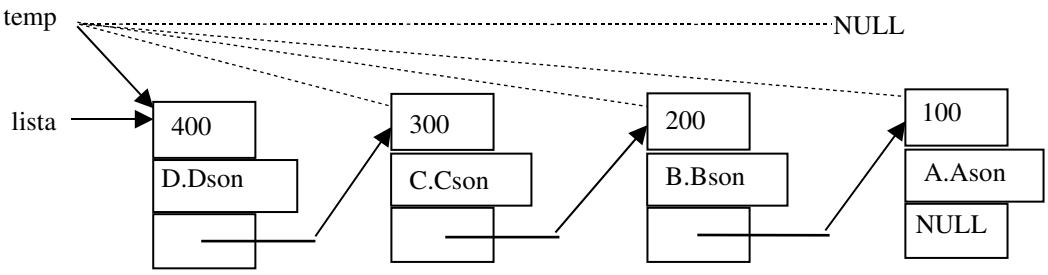
Efter inläsning av ytterligare två medlemmar ser listan ut som:



Vill man ha en utskrift av alla länkar i listan börjar man med att skaffa sig en temporär pekare, som man sätter att peka på den första länken. Sedan kan man stega framåt i listan och skriva ut varje länk.

```
/* Skriv ut listans data */

temp = lista;

while ( temp != NULL )
{
   printf("\nMedlemsnr : %d\n", temp->nr);
   printf("Medlemsnamn : %s\n", temp->namn);

   /* Flytta till nästa länk */
   temp = temp->next;
}
```

Pekaren temp används som en iterator, för att skriva ut alla länkar i listan. Då temp vid uppräkningen tilldelas värdet NULL, vilket inträffar då den sista länkens temp->next tilldelas, har man skrivit ut hela listan.

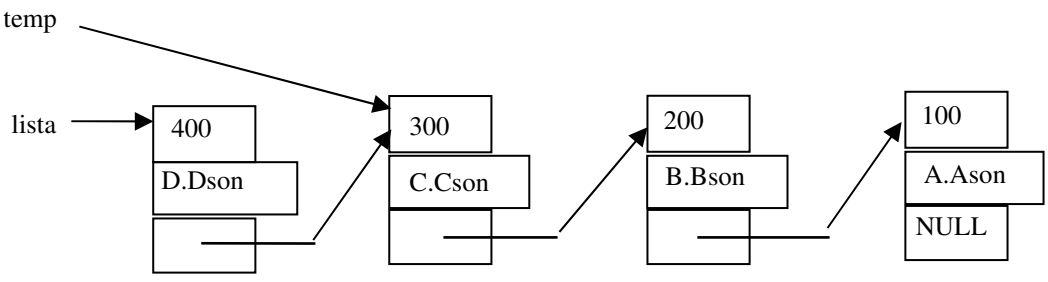


OBS! Använd *ej pekaren lista för att flytta framåt* i listan för då *tappar man bort* början på listan!

Man kan söka efter en medlem i listan på motsvarande sätt som när man skriver ut listan. Man har en temporär pekare som man flyttar framåt i listan så länge man inte hittat medlemmen och listan inte slut.

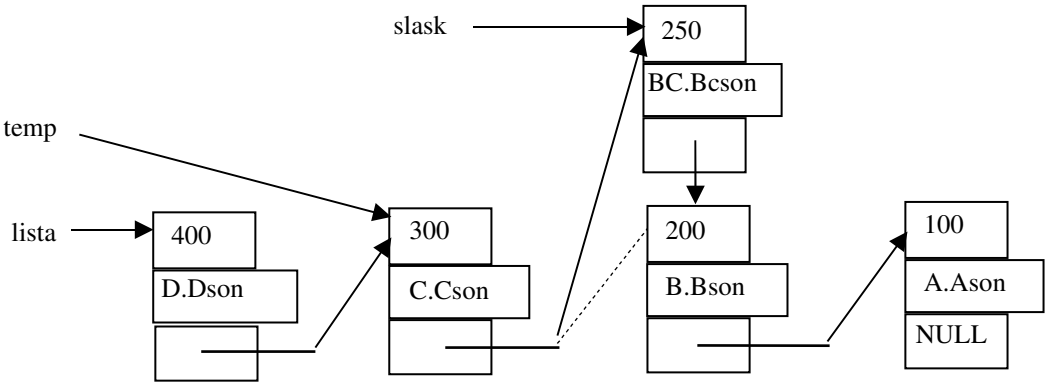
```
int snr, hittat = 0;

/* Läs in medlemsnummer som ska sökas */

printf(”Ge medlemsnummer : ”);
scanf(”%d”, &snr);


/* Sök i listan */

temp = lista;
while ( temp != NULL && !hittat )
{
   if ( snr == temp->nr )
      /* Hittat länken */
      hittat = 1;
   else
      /* Flytta till nästa länk */
      temp = temp->next;
}


/* Skriv ut sökresultat */

if ( hittat )
   printf(”Medlemsnamn : %s\n”, temp->namn);
else
   printf(”Medlemmen finns ej i listan!\n”);
```

Läser man exempelvis in medlemsnumret 300 stannar sökningen med temp pekande på länken:



och C.Cson skrivs ut.

Vill man stoppa in nya medlemmar i listan på speciella ställen efter en länk söker man sig fram i listan länk för länk, skriver ut medlemsnumret och frågar om instoppning ska ske efter länken. Ska instoppning ske skapar man en ny medlem, läser in data till den och stoppar in den i listan enligt:

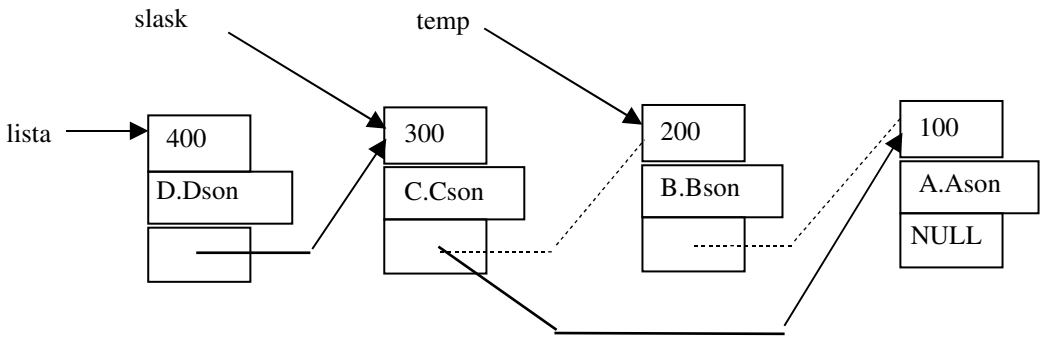



```
struct medlem *slask;
char svar;

/* Sök dig fram i listan */

temp = lista;
while ( temp != NULL )
{
   printf(”Medlem nr : %d\n”,temp->nr);
   printf(”Ska du länka in (J/N)? ”);
   svar = toupper(getchar());
   getchar();
   if ( svar == ‘J’ )
   {
      slask = malloc(sizeof(struct medlem));
      printf(”Ge medlemsnummer : ”);
      scanf(”%d”, &slask->nr);
      getchar();
      printf(”Ge medlemsnamn :”);
      gets(slask->namn);

      /* Stoppa in länken */
      slask->next = temp->next;
      temp->next = slask;
   }
   temp = temp->next;
}
```

OBS! Man måste ställa om pekarna i *rätt ordning* annars kan man tappa bort resten av listan. Innan man ställer om temp->next att peka på slask måste man sätta

slask->next att peka på temp->next.

Vill man ta bort en medlem från listan kan man på motsvarande sätt leta sig fram i listan till aktuell medlem och sedan plocka bort denna enligt:




```
temp = slask = lista;
while ( temp != NULL )
{
   printf(”Medlem nr : %d\n”,temp->nr);
   printf(”Ska du ta bort (J/N)? ”);
   svar = toupper(getchar());
   getchar();
   if ( svar == ‘J’ && temp == lista)
   {
      /* Första medlemmen ska bort */
      slask = lista = lista->next;
      free(temp);
      temp = lista;
   }
   else if ( svar == ‘J’ )
   {
      /* Medlem inuti listan ska bort*/
      slask->next = temp->next;
      free(temp);
      temp = slask->next
   }
   else
   {
      /* Medlemmen ska ej bort */
      slask = temp;
      temp = temp->next;
   }
}
```

OBS! Hur man måste skilja på att ta bort den första medlemmen jämfört med resten av medlemmarna.