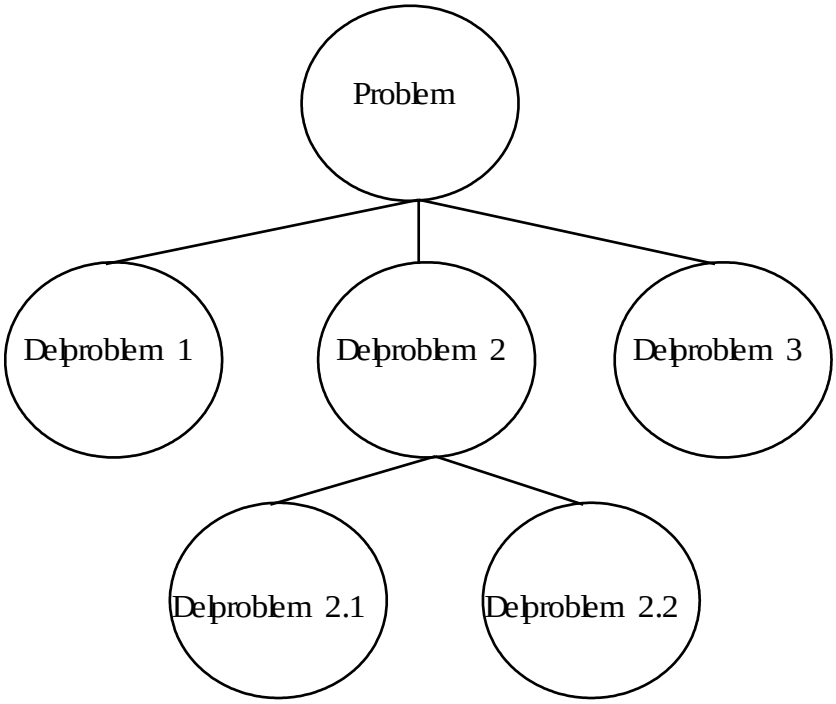
5 Funktioner

(Senaste ändring av detta kapitel: 14 december 2006)

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 dem på lägre nivå senare. 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 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 <stdlib.h>        /* atof */
#include <stdio.h>         /* printf, gets */
#include <math.h>          /* pow */

int main()
{
   double 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\n", pow(x, 2));         /* anrop av printf, pow*/

   return 0;
}
```

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.

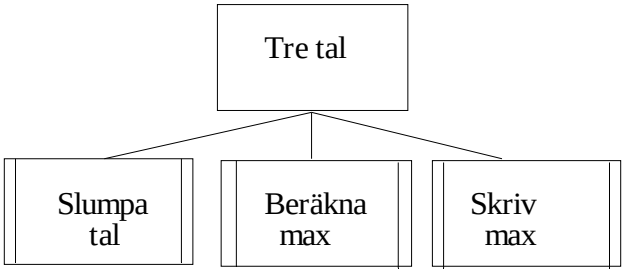
Till vissa funktioner kan man skicka med information. Till exempelvis funktionen pow ska man vid anropet skicka information om vilket tal som ska upphöjas till vad. Ska man ej skicka med någon information anges detta med tomma parenteser.

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:

```
int main()
{
   slumpa_tal();
   berakna_max();
   skriv_max();

   return 0;
}
```

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 före 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);
}
```

```
int main()
{
   slumpa_tal();
   berakna_max();
   skriv_max();

   return 0;
}
```

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);
}

int 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);

   return 0;
}
```

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*:

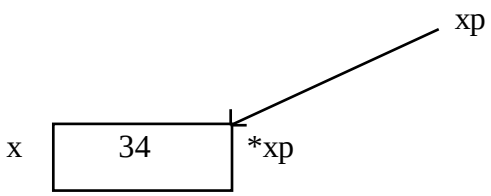
```
void slumpa_tal(int *xp, int *yp, int *zp)
{
   srand((unsigned)time(NULL));
   *xp = rand()%100;
   *yp = rand()%100;
   *zp = rand()%100;
}
```

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 :

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

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

   return 0;
}
```

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>

int main()
{
   char liten_bokstav, stor_bokstav;

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

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

   return 0;
}
```

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.

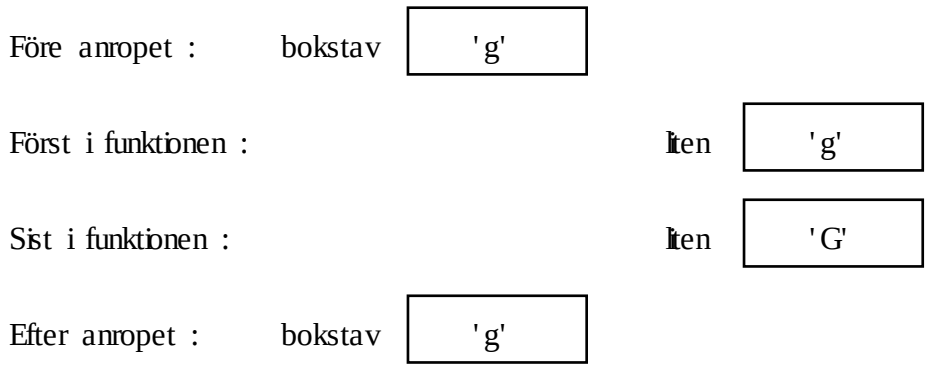
(Ett tips: C-standarden innehåller en färdig funktion som heter `toupper`, och som översätter från en liten till en stor bokstav.)

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>

int main()
{
   char bokstav;

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

   stor(bokstav);
   printf("Motsvarande stora bokstav: %c\n", bokstav);

   return 0;
}
```

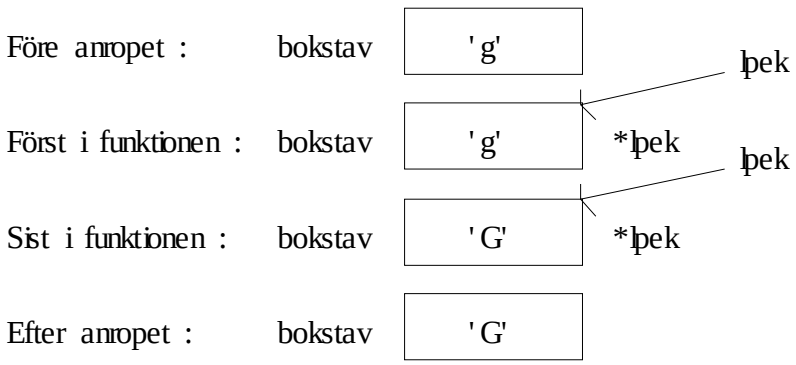
Programmet gör ingen omvandling utan skriver alltid ut samma lilla bokstav. Detta beror på att värdet av den aktuella parametern, bokstav, visserligen kopieras till den formella parametern, liten, som i funktionen omvandlas till en stor bokstav, men aldrig kopieras tillbaka till bokstav.



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>

int main()
{
   char bokstav;

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

   stor(&bokstav);
   printf("Motsvarande stora bokstav: %c\n", bokstav);

   return 0;
}
```

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 kommer själva vektor-namnet (normalt) att översättas till en adress, när man skriver det i ett C-program. *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]);
   printf(”\n”);
}

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;
   }
}
```

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

   printf("Hur många tal? (Max 1000!) : ");
   scanf("%d", &antal);
   slumpa(vek, antal);
   printf("\n");
   printf("Slumpad vektor\n");
   skriv(vek, antal);
   sortera(vek, antal);
   printf("\n");
   printf("Sorterad vektor\n");
   skriv(vek, antal);

   return 0;
}
```

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.

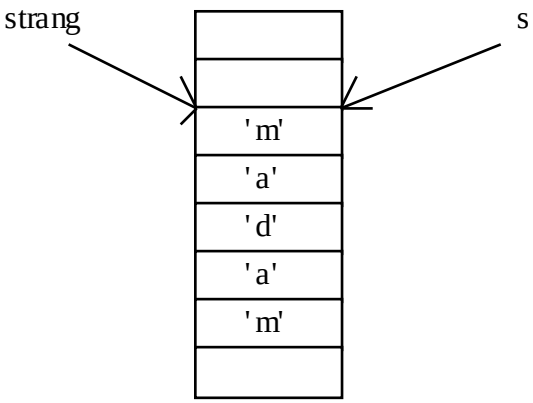
När det gäller strängar, alltså vektorer av tecken, gäller samma sak som för vektorer. Här använder man dock nästan alltid en * istället för tomma hakparenteser [] som formell parameter.

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 palindrom(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;
}

int main()
{
   char strang[80 + 1];

   printf("Ge en sträng: ");
   gets(strang);
   if ( palindrom(strang) )
      printf("Strängen är ett palindrom!\n");
   else
      printf("Strängen är ej ett palindrom!\n");

   return 0;
}
```

Här har man i funktionen `palindrom`, som kontrollerar om en sträng är ett 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 + 1];

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

int main()
{
   char a_namn[80 + 1], b_namn[80 + 1];

   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);

   return 0;
}
```

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;
   }
}

int 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);

   return 0;
}
```

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! Här har man en vektor av pekare, 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>

struct medlem
{
   int nr;
   char namn[30 + 1];
};

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;
}

int main()
{
   struct medlem med1, med2;

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

   byt(&med1, &med2);

   skriv(med1);
   skriv(med2);

   return 0;
}
```

OBS! *När man avrefererar en post-pekare kan man använda en förkortad syntax med -> istället för *.*
*”a->b” betyder samma sak som ”(*a).b”.*

5.4 Makron

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

Ex: Skriv ett makro 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 makron.

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

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

Ex: Skriv ett makro som definierar makro-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') };
```

(En varning: Såna här komplicerade makron bör man se upp med i riktiga program. Det är lätt att det blir fel! Använd hellre funktioner.)

makron kan precis som funktioner ha parametrar:

Ex: Definera ett makro 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 makron 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 makro kan ersätta funktioner. Många standarfunktioner som finns i header-filer som stdio.h, stdlib.h, är implementerade som makron, istället för funktioner. Man ska dock undvika att ha för långa makron 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, makron 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".