4 Programkonstruktion

När man skriver stora program måste man, precis som man gör vid lösning av stora problem, *dela upp det hela i mindre delar*. Hur ska man göra uppdelningen av sina stora program? Här måste man utgå ifrån vad ett program egentligen är, nämligen *en plan för bearbetning av data eller information*. Dessutom ska man ha målsättningen att programmen ska bli:

- korrekta
- robusta
- lätta att använda
- lätta att underhålla
- lätta att förändra
- återanvändbara

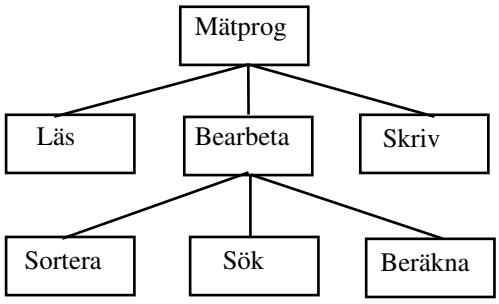
Man har använt och använder fortfarande olika metoder för att konstruera stora program. Vilken metod som används beror på vad det är för typ av program man gör och vilka målsättningar man prioriterar. Här kommer metoderna för uppdelning med hjälp av *stegvis förfining*, *dataflöden* och *abstrakta datatyper* att genomgås.

Metoderna poängterar antingen *bearbetning* (stegvis förfining), *data* (dataflöden) eller *både data och bearbetning* (abstrakta datatyper) i mer eller mindre grad och alla metoder har sina för- och nackdelar.

Ex: Ett mätsystem ska hämta data i form av mätposter till en vektor, sortera vektorn, läsa in en nyckel och söka i vektorn efter denna nyckelpost samt avslutningsvis beräkna och skriva ut viss data om denna post.

a) *Stegvis förfining:*

Vad ska programmet göra? Vilka bearbetningar är aktuella? Finfördela vid behov.

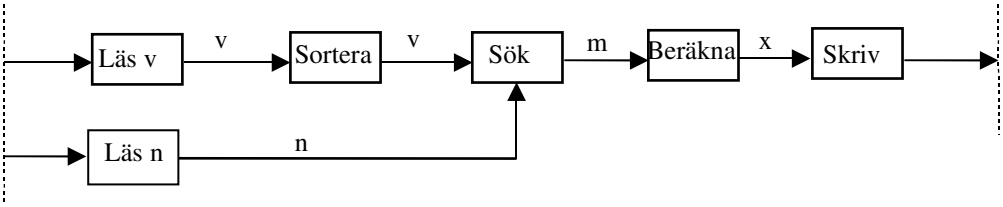


Fördelar : Logisk uppdelning i mindre delar som automatiskt blir funktioner.

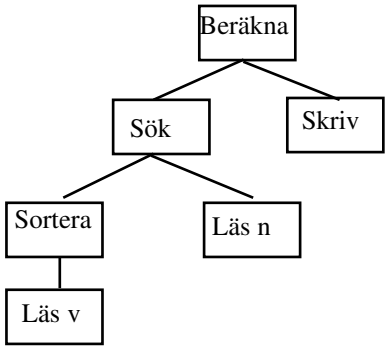
Nackdelar : Glömmer bort data. Alla datatyper måste definieras globalt och variabler av dessa transporteras som parametrar och finns då i väldigt många funktioner vilket försvårar underhåll och förändringar. Man får dessutom en dålig avbild eller modell av verkligheten, vilket också försvårar underhåll och förändringar.

b) *Dataflöden:*

Hur flödar data? Hur bearbetas data? Funktion för varje bearbetning.



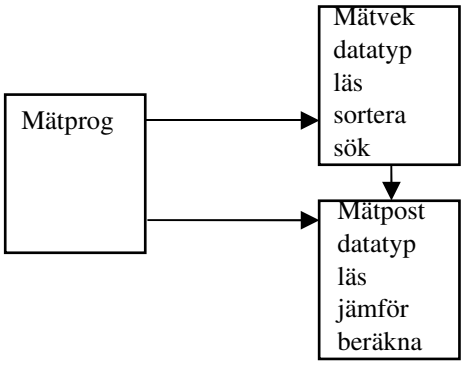
Häng upp funktionerna där inmatad data övergår till att bli utmatad, vid beräkna.



Fördelar : Bättre avbild av verkligheten där även data beaktas.
Nackdelar : Fortfarande globala datatyper som försvårar underhåll och förändringar.

c) *Abstrakta datatyper:*

Vad ska programmet hantera? Vilka abstrakta datatyper, med data och funktioner är aktuella?



Fördelar: Data och operationer samlade på samma ställe. Lätt att underhålla och förändra. Dessutom finns oftast variablerna eller objekten av de abstrakta datatyperna i verkligheten också. Bättre möjligheter till återanvändning.
Nackdelar: Ingen logisk stegvis förfining. Man måste börja tänka om i form av abstrakta datatyper.

4.1 Stegvis förfining

Man utgår ifrån *vad programmet ska göra* eller med andra ord de operationer (initiera, läsa, beräkna, mäta, skriva etc) som ingår i programspecifikationen. Varje operation blir då i princip en funktion i programmet på högsta nivån. Därefter fortsätter man uppdelningen genom att stegvis förfina operationerna till mindre operationer på lägre och lägre nivå.

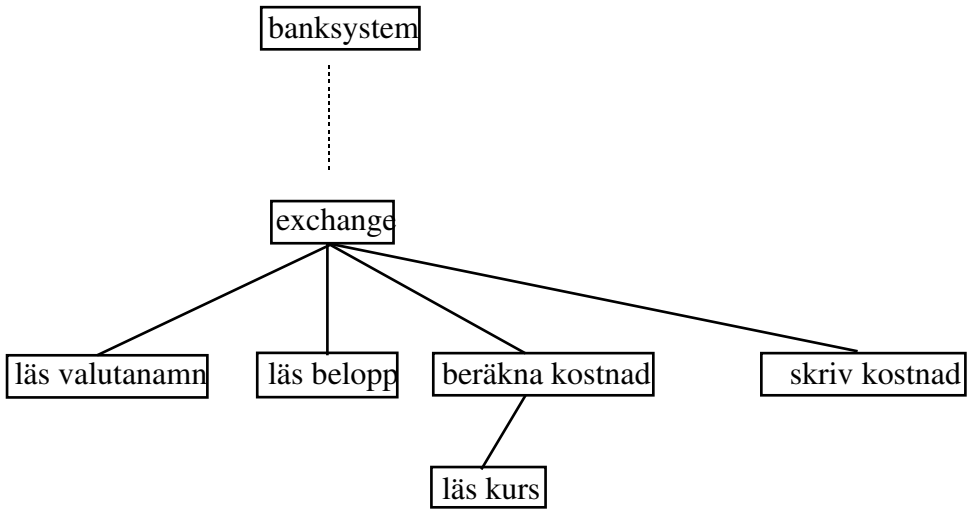
Fördelen med stegvis förfining är att man löser problem genom att *skjuta onödiga detaljer framför sig* och lösa dessa efterhand som de dyker upp. Man behöver ej ha hela problemställningen i huvudet samtidigt.

Nackdelen är den *relativt fria kompositionen*. Samma program kan skrivas på väldigt många olika sätt och programmet är ofta en dålig avbild av verkligheten. Vid underhåll och förändringar blir det svårt att hitta alla operationer som man ska ändra i, eftersom data och operationer är åtskilda.

Ex: Skriv ett program, exchange, som har hand om växling av valuta i ett banksystem. Programmet ska läsa in valutanamn och växelbelopp och beräkna och skriva ut kostnaden för valutan i svenska kronor. För beräkning av kostnad ska aktuell kurs läsas från en textfil kurs.txt som ser ut som:

pund 14.32
usdol 7.65
euro 9.12

Med stegvis förfining uppdelar vi programmet i mindre delar enligt :



När man implementerar program, som är konstruerade med stegvis förfining, går man nästan alltid uppifrån och ner (top-down). Man börjar med att implementera huvudprogrammet med globala datatyper och funktionsprototyper. Därefter kan man gå till resp funktion och implementera dessa en och en, samtidigt som man testar programmet med en funktion åt gången.

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

enum valutanamn {pund, usdol, euro};


enum valutanamn las_valutanamn(void)
{
   char str[20];
   enum valutanamn v;
   int fel;

   do
   {
      fel = 0;
      printf("Ge valutans namn : ");
      gets(str);
      if (strcmp(str, "pund") == 0)
         v = pund;
      else if (strcmp(str, "usdol") == 0)
         v = usdol;
      else if (strcmp(str, "euro") == 0)
         v = euro;
      else
      {
         printf("Felaktigt namn!\n");
         fel = 1;
      }
   } while (fel);
   return v;
}


double las_belopp(void)
{
   double b;
   char str[80];

   do
   {
      printf("Ge köpbelopp : ");
      gets(str);
      b = atof(str);
      if (b <= 0)
         puts("Fel belopp!");
   } while (b <= 0);
   return b;
}
```

```
double las_kurs(enum valutanamn sv)
{
   FILE *vfil;
   char str[20];
   enum valutanamn v;
   double kurs;

   vfil = fopen("kurs.txt","r");
   do
   {
      fscanf(vfil, "%s%lf", str, &kurs);
      if (strcmp(str, "pund") == 0)
         v = pund;
      else if (strcmp(str, "usdol") == 0)
         v = usdol;
      else if (strcmp(str, "euro") == 0)
         v = euro;
   } while ( v != sv);
   fclose(vfil);
   return kurs;
}


double berakna_kostnad(enum valutanamn v, double b)
{
   return  b * las_kurs(v);
}


void skriv_kostnad(enum valutanamn v, double b, double k)
{
   char str[20];

   switch (v)
   {
      case pund  : strcpy(str, "pund"); break;
      case usdol : strcpy(str, "usdol"); break;
      case euro  : strcpy(str, "euro");
   }
   printf("%.2f %s kostar %.2f kr", b, str, k);
}


int main()
{
   enum valutanamn val;
   double belopp, kostnad;

   val = las_valutanamn();
   belopp = las_belopp();
   kostnad = berakna_kostnad(val, belopp);
   skriv_kostnad(val, belopp, kostnad);
   return 0;
}
```

Ex: A Video Game (ur Software Engineering, A Programming Approach)

In order to illustrate the use of functional decomposition we will design the software for a simple video game, called ‘breakout’. This account can be skipped by the reader who feels they have a grasp of the method. The game make use of a keyboard and a crude visual display that can display characters (but no other graphics) at any position on the screen. The screen looks like :

The figure portrays a wall, built from ‘w’ characters, a bat ‘bbb’ and a ball the letter o. The ball bounces around the screen, moving up, down or diagonally, changing direction conventionally when it hits the wall, the sides or the bat. The player can move the bat left or right by keying ‘l’ or ‘r’ on the keyboard. The objective is to make the ball knock down the wall and so break out.
The ball is out of play and vanishes when it passes the bat. The player is supplied with a succession of 4 balls. When these have been used the best score ever attained is displayed.

Let us refer to places on the screen by a pair of x, y coordinates and assume that the software can assign a character value to a position on the screen by executing a statement like:

screen(17, 22) = ‘b’

Similarly suppose that the value at any position can be tested using a statement like:

if screen(17, 22) == ‘w’

We will refer to the coordinates of the ball as xball and yball. We begin with the overall logic:

video game
  draw sides

```
  draw bat at home position

  while someone wants to play
    draw wall
    play a game
    display best score
```

We have relegated detail to statements like

```
  draw sides
```

and we can study and check the design without worrying about detail. We probably realize that essential steps are missing so we might amend the above to:

```
video game
  draw sides
  draw bat at home position
  best score = 0
  while somone wants to play
    score = 0
    draw wall
    play a game
    if score > best score
      best score = score
    display best score
```

Now that we are satisfied with this, we can choose one of the statements and write down what it does in more detail. An obvious candidate is :

```
play a game
  balls left = 4
  while balls left > 0
    play a ball
    subtract one from balls left
```

and then:

```
play a ball
  create a new ball
  while ball is in play
    check ball position
    move ball
    move bat
  erase ball
```

Let us now work on detail of moving the bat:

```
move bat
```

```
  if key ==‘l’  move left
  if key ==’r’ move right
```

which requires:

```
move left
  erase old bat
  if bat position > left wall + 1
    bat position = bat position - 1
  draw bat
```

This completes a fair amount of the detail concerned with moving the bat. We can leave futher detail for the time being and turn to consider more of the detail of moving the ball.

```
move ball
  erase ball
  xball = xball + xstep
  yball = yball + ystep
  screen(xball, yball) = ‘o’
```

but we have to be careful to check whether it has hit something:

```
check ball position
  check out of play
  check sides
  check bat
  check brick
```

We will finish the account of this development with just two more pieces:

```
check sides
  if ball at either side
    xstep = -xstep
  if ball at end boundary
    ystep = - ystep

check brick
  if screen(xball + xstep, yball + ystep) is a brick
    screen(xball + xstep, yball + ystep) = ‘ ‘
    add one to score
    ystep = - ystep
    subtract one from bricks left
    if bricks left == 0 draw wall
```

There is still some way to go with this particular design, but this much gives the flavor of the design and of the method. Notice how, throughout, we can check the design

without having to take account of a lot of unnecessary detail. Notice the informality of the statements used and the control structures if and while.

4.2 Dataflöden

Man utgår ifrån *hur data flödar* i programmet, från indata fram till utdata. Man koncentrerar sig på data och hur data ska bearbetas av programmet enligt programspecifikationen. Varje *bearbetning blir då i princip en funktion* i programmet. Därefter tar man den funktion, där inmatad data övergår till att bli utmatad, till huvudfunktion och låter resten av funktionerna anropas utgående ifrån dataflödena, som kan liknas vid snören som binder ihop funktionerna.

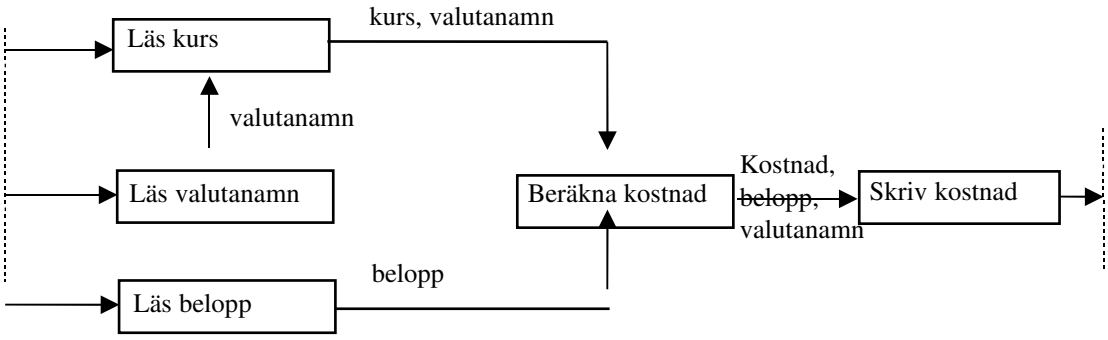
Fördelen med att använda dataflöden är att *man bygger sitt program på hur verkligheten ser ut*. Man får en väl förankrad modell av verkligheten som är bra att ha då programmet ska underhållas och förändras. Dessutom får man en *naturlig uppdelning av programmet* i mindre delar.

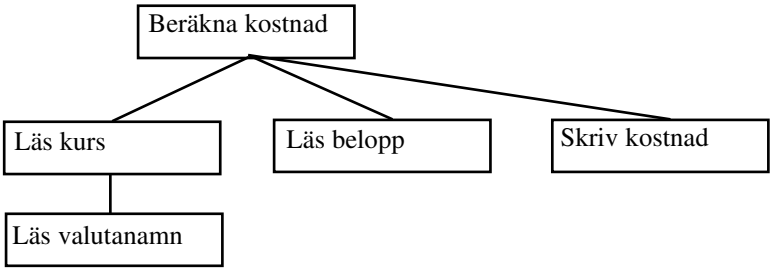
Nackdelen är dock att data och operationer på denna data fortfarande är åtskilda och det är svårt att hitta alla förändringar som ska göras då exempelvis data ska förändras.

Ex: Skriv ett program exchange som har hand om växling av valuta i ett banksystem. Programmet ska läsa in valutanamn och växelbelopp och beräkna och skriva ut kostnaden för valutan i svenska kronor. För beräkning av kostnad ska aktuell kurs läsas från en textfil kurs.txt som ser ut som:

pund 14.32
usdol 7.65
euro 9.12

Genom att betrakta dataflödet får man en naturlig uppdelning enligt :





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


enum valutanamn {pund, usdol, euro};


double las_belopp(void)
{
   double b;
   char str[80];

   do
   {
      printf("Ge köpbelopp : ");
      gets(str);
      b = atof(str);
      if (b <= 0)
        puts("Fel belopp!");
   } while (b <= 0);
   return b;
}



enum valutanamn las_valutanamn(void)
{
   char str[20];
   enum valutanamn v;
   int fel;

   do
   {
      fel = 0;
      printf("Ge valutans namn : ");
      gets(str);
      if (strcmp(str, "pund") == 0)
        v = pund;
      else if (strcmp(str, "usdol") == 0)
        v = usdol;
      else if (strcmp(str, "euro") == 0)
        v = euro;
      else
      {
        printf("Felaktigt namn!\n");
        fel = 1;
      }
   } while (fel);
   return v;
}
```

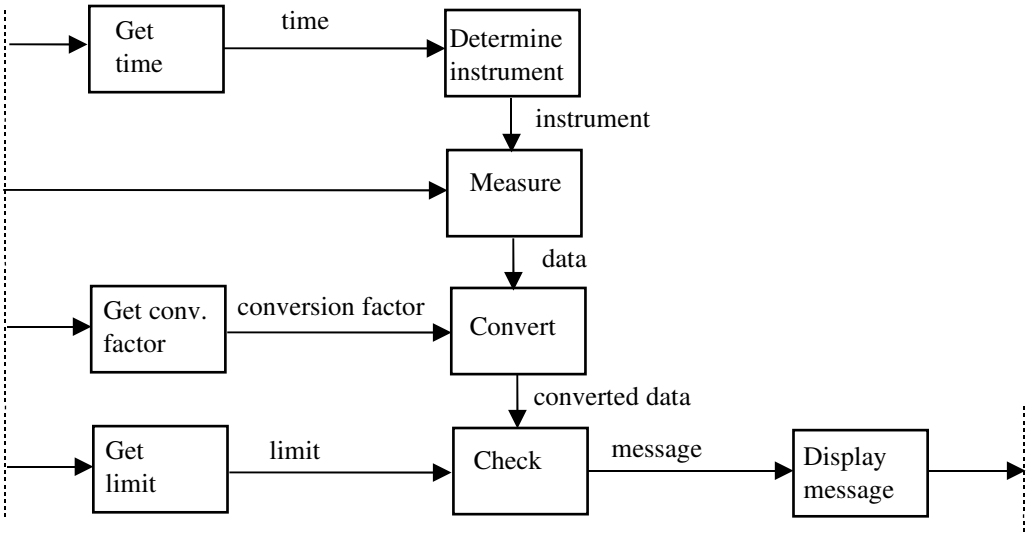
```
double las_kurs(enum valutanamn *vp)
{
   FILE *vfil;
   char str[20];
   enum valutanamn v;
   double kurs;

   v = las_valutanamn();
   vfil = fopen("kurs.txt","r");
   do
   {
      fscanf(vfil, "%s%lf", str, &kurs);
      if (strcmp(str, "pund") == 0)
         *vp = pund;
      else if (strcmp(str, "usdol") == 0)
         *vp = usdol;
      else if (strcmp(str, "euro") == 0)
         *vp = euro;
   } while ( *vp != v);
   fclose(vfil);
   return kurs;
}



void skriv_kostnad(enum valutanamn v, double b, double k)
{
   char str[20];

   switch (v)
   {
      case pund  : strcpy(str, "pund");
                   break;
      case usdol : strcpy(str, "usdol");
                   break;
      case euro  : strcpy(str, "euro");
   }
   printf("%.2f %s kostar %.2f kr", b, str, k);
}



int main()
{
   enum valutanamn v;
   double belopp, kostnad;

   belopp = las_belopp();
   kostnad = belopp * las_kurs(&v);
   skriv_kostnad(v, belopp, kostnad);
   return 0;
}
```

Ex: Monitoring A Plant (ur Software Engineering, A Programming Approach)

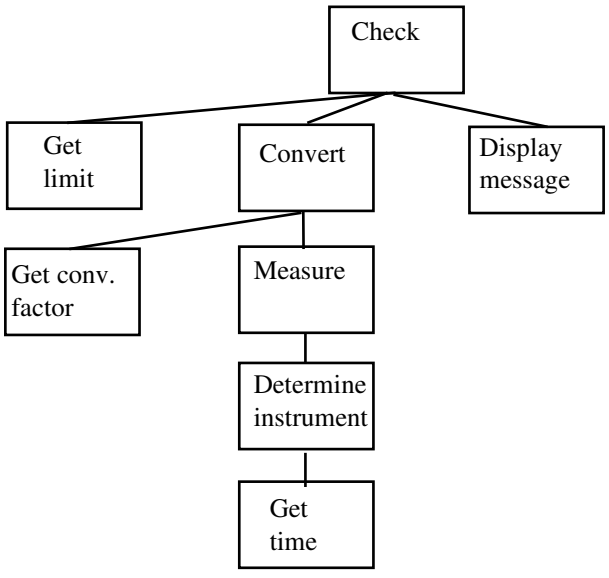
Here is the specification for a software system:

A computer is being used to monitor an industrial plant. The computer periodically inputs readings from instruments in the plant. Some of the readings require conversion to normal units of measurement (e.g microvolts into degrees). The computer checks each of the readings against permissible values. Alarm reports are displayed on a screen when a value is outside its valid range.

The data flow diagram for this problem is :



It is relatively clear that the central transform in the system is the bubble that checks the values. Hence we can derive the program structure diagram as:



4.3 Abstrakta datatyper

Vad ska *programmet hantera*? Vilka abstrakta datatyper i form av *data och operationer* på denna typ av data är aktuella? Till skillnad mot föregående metoder koncentrerar man sig här på *både data och operationer* och försöker samla dessa på ett och samma ställe. När man har konstruerat de aktuella abstrakta datatyperna, som används i programmet, kan en användare, som kan vara en annan abstrakt datatyp eller ett huvudprogram, skapa *variabler eller objekt* av dessa.

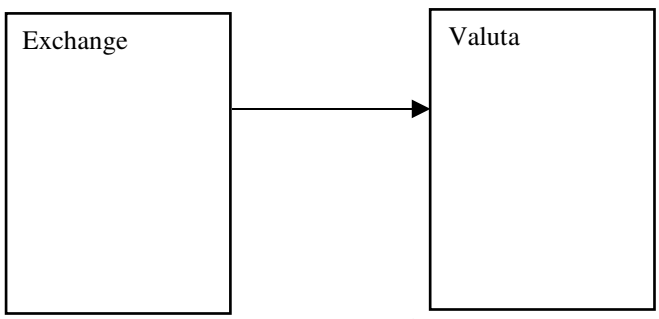
Fördelen med att använda abstrakta datatyper är att man samlar data och operationer på denna typ av data, på samma ställe, i samma modul och därför blir det *mycket lättare att underhålla och förändra* programmen. Dessutom blir det naturligt att återanvända de abstrakta datatyperna i andra program.

Nackdelen är att man måste tänka om vid programkonstruktionen. Man kan inte följa den mer logiska stegvisa förfiningen utan man börjar istället med att leta efter abstrakta datatyper alltså datatyper utöver de vanliga double, char, int etc som programmet ska hantera och som man ofta avbildar som egendefinierade poster. Efter att ha hittat lämpliga abstrakta datatyper letar man efter till dessa hörande operationer. Sedan undersöker man hur de olika abstrakta datatyperna ska samarbeta för att hela programmet ska fungera och slutligen implementerar man de olika delarna.

Ex: Skriv ett program exchange som har hand om växling av valuta i ett banksystem. Programmet ska läsa in valutanamn och växelbelopp och beräkna och skriva ut kostnaden för valutan i svenska kronor. För beräkning av kostnad ska aktuell kurs läsas från en textfil kurs.txt som ser ut som:

pund 14.32
usdol 7.65
euro 9.12

Vad ska programmet hantera? Valutor! Här finns den abstrakta datatypen valuta som man kan avbilda som en post innehållande termerna valutanamn och kurs. De operationer som är aktuella är läsa, skriva och beräkna kostnad för valuta. Eftersom man bara har en abstrakt datatyp kommer det bara att vara huvudprogrammet som använder denna och *beroendediagrammet* blir enligt:



För att implementera den abstrakta datatypen valuta använder man sig i C av en *specifikationsfil valuta.h* och en *implementationsfil valuta.c* enligt:

```
/* Specifikation av valuta -- valuta.h */

enum valutanamn {pund, usdol, euro};

typedef
struct
{
   enum valutanamn namn;
   double kurs;
} valuta;

void las_valuta(valuta *vp);
void skriv_valuta(valuta v);
double berakna_kostnad(valuta v, double belopp);
```

```
/* Implementation av valuta -- valuta.c */

#include "valuta.h"
#include <stdio.h>
#include <string.h>

enum valutanamn las_valutanamn(void)
{
   char str[20];
   enum valutanamn v;
   int fel;

   do
   {
      fel = 0;
      printf("Ge valutans namn : ");
      gets(str);
      if (strcmp(str, "pund") == 0)
         v = pund;
      else if (strcmp(str, "usdol") == 0)
         v = usdol;
      else if (strcmp(str, "euro") == 0)
         v = euro;
      else
      {
         printf("Felaktigt namn!\n");
         fel = 1;
      }
```

```
    } while (fel);
    return v;
}
```

```
double las_kurs(enum valutanamn sv)
{
   FILE *vfil;
   char str[20];
   enum valutanamn v;
   double kurs;


   vfil = fopen("kurs.txt","r");

   do
   {
      fscanf(vfil, "%s%lf", str, &kurs);
      if (strcmp(str, "pund") == 0)
         v = pund;
      else if (strcmp(str, "usdol") == 0)
         v = usdol;
      else if (strcmp(str, "euro") == 0)
         v = euro;
   } while ( v != sv);
   fclose(vfil);
   return kurs;
}


void las_valuta(valuta *vp)
{
   vp->namn = las_valutanamn();
   vp->kurs = las_kurs(vp->namn);
}


double berakna_kostnad(valuta v, double belopp)
{
   return ( belopp * v.kurs);
}

void skriv_valuta(valuta v)
{
   char str[20];

   switch (v.namn)
   {
      case pund  : strcpy(str, "pund"); break;
      case usdol : strcpy(str, "usdol"); break;
      case euro  : strcpy(str, "euro");
   }
   printf("\nValutanamn : %s\n", str);
   printf("Kurs       : %.2f\n", v.kurs);
}
```

OBS! Vid implementationen utnyttjar man egna lokala funktioner som ej exporteras och alltså ej finns med i headerfilen.

Huvudprogrammet kan nu utnyttja den abstrakta datatypen genom att skapa *variabler eller objekt* av denna datatyp och utnyttja de funktioner som exporteras för den abstrakta datatypen.

```
/* Huvudprogram -- exchange.c */

#include <stdio.h>
#include <math.h>
#include "valuta.h"


double las_belopp(void)
{
   double b;
   char str[80];

   do
   {
      printf("Ge köpbelopp : ");
      gets(str);
      b = atof(str);
      if (b <= 0)
         puts("Fel belopp!");
   } while (b <= 0);
   return b;
}



int main()
{
   valuta v;
   double b;

   las_valuta(&v);
   b = las_belopp();
   skriv_valuta(v);
   printf("Belopp: %.2f\n", b);
   printf("Kostnad: %.2f\n", berakna_kostnad(v, b));
   return 0;
}
```

OBS! Hur man även här använder andra funktioner utöver de från den abstrakta datatypen.

En av fördelarna med att använda abstrakta datatyper är att dessa ska kunna *återanvändas* av andra program. I utvecklingsmiljön för C-program brukar det finnas färdiga abstrakta datatyper som:

stdio.h – in- och utmatning, ingår i C-standarden
dos.h – doskommandon i Windows
conio.h – videofunktioner för Windows i textmode

Har man ej tillgång till färdiga återanvändbara abstrakta datatyper måste man skriva dessa själva. Exempel på sådana är listhantering i lifo.h, fifo.h och twolist.h som genomgåtts tidigare och som visar hur man kan återanvända abstrakta datatyper. Problemet vid återanvändning av exempelvis twolist.h är att den, i den form som beskrivits ovan, enbart kan användas för en enda typ av data. Vill man använda flera olika tvåvägslistor i sitt program kan man använda *poster innehållande unioner som data*.

Ex: Skriv ett program som läser in fordon med registreringsnummer, ägare och typ som kan vara personbil, lastbil med tillåten lastvikt i ton eller buss med tillåtet antal passagerare från en textfil enligt:

```
EEE555   E.E   person
AAA111   A.A   buss   10
CCC333   C.C   buss   30
GGG777   G.G   last   7.7
BBB222   B.B   person
FFF666   F.F   last   6.6
DDD444   D.D   buss   40
```

och stoppar in dessa i en tvåvägslista sorterad efter registreringsnummer. Programmet ska fortsätta med att skapa en ny tvåvägslista innehållande bara bussar och sedan skriva ut hela fordonslistan och busslistan på skärmen enligt:

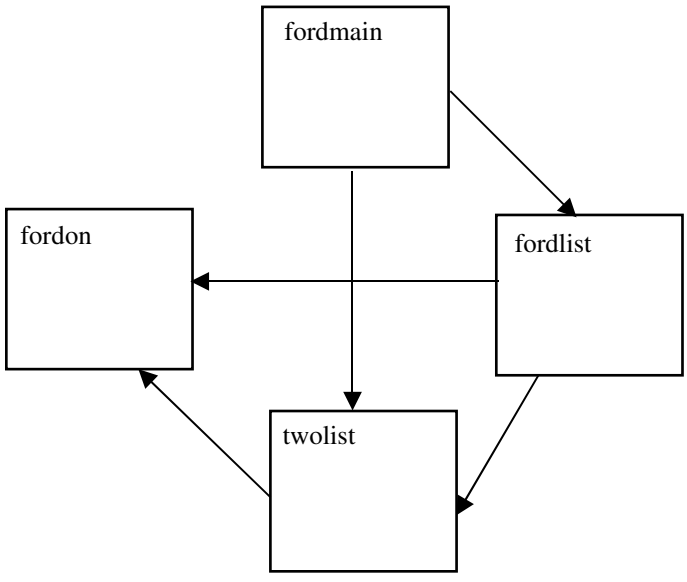
```
Fordonslista
AAA111    A.A    buss      10
BBB222    B.B    person
CCC333    C.C    buss      30
DDD444    D.D    buss      40
EEE555    E.E    person
FFF666    F.F    last      6.600000
GGG777    G.G    last      7.700000

Busslista
AAA111    A.A    buss      10
CCC333    C.C    buss      30
DDD444    D.D    buss      40
```

Man börjar med att *identifiera vilka objekt* som programmet ska hantera och därmed *vilka abstrakta datatyper* som ska skapas eller återanvändas. I specifikationen talas om fordon, fordonslistor, bussar och busslistor. Eftersom det finns en färdig abstrakt datatyp för tvåvägslistor, som man kan använda för en enda aktuell datatyp i sitt program, måste man försöka avbilda alla typer av fordon med en enda datatyp. Därmed räcker det med den abstrakta datatypen fordon för alla typer av fordon. Fordon kommer att avbildas enligt:

```
enum fordontyp {person, last, buss};

typedef
struct fordonpost
{
   char regnr[10];
   char agare[30];
   enum fordontyp ftyp;
   union
   {
      double vikt;
      int pass;
   } u;
} fordon;
```

Med ovanstående union ingående i posten markeras att innehållet i minnet kan vara *antingen en double i form av vikten eller ett heltal som håller reda på antalet tillåtna passagerare*. Man måste själv utgående ifrån ftyp hålla reda på vad som finns i u. För termen u allokeras alltid minne för det *största möjliga* innehållet.

Nu kan man rita en beroendegraf för de abstrakta datatyper som programmet kommer att innehålla.



```
/* Specifikation av fordon -- fordon.h */

      #ifndef FORDONH
      #define FORDONH

      #include <stdio.h>

      enum fordontyp {person, last, buss};

      typedef
      struct fordonpost
      {
         char regnr[10];
         char agare[30];
         enum fordontyp ftyp;
         union
         {
            double vikt;
            int pass;
         } u;
      } fordon;

      void las_fordon(FILE *infil, fordon *fp);
      void skriv_fordon(fordon f);
      int is_less_fordon(fordon f1, fordon f2);
      int is_a_bus(fordon f);

      #endif


      /* Implementation av fordon -- fordon.c */

      #include "fordon.h"
      #include <stdio.h>
      #include <string.h>

      void las_fordon(FILE *infil, fordon *fp)
      {
         char str[10];
         double x;

         fscanf(infil, "%s%s%s", fp->regnr, fp->agare, str);
         if ( strcmp(str, "person") == 0 )
            fp->ftyp = person;
         else if ( strcmp(str, "last") == 0 )
         {
            fp->ftyp = last;
            fscanf(infil, "%lf", &x);
            fp->u.vikt = x;
         }
         else if ( strcmp(str, "buss") == 0 )
         {
            fp->ftyp = buss;
            fscanf(infil, "%d", &fp->u.pass);
         }
      }
```

```
void skriv_fordon(fordon f)
{
   char str[10];

   printf("%-10s%-20s", f.regnr, f.agare);

   switch (f.ftyp)
   {
      case person : printf("person\n");break;
      case last : printf("last      %f\n", f.u.vikt); break;
      case buss : printf("buss      %d\n", f.u.pass);
   }
}


int is_less_fordon(fordon f1, fordon f2)
{
   return (strcmp(f1.regnr, f2.regnr) < 0 );
}


int is_a_bus(fordon f)
{
   return (f.ftyp == buss);

}




/* Specifikation av tvåvägslistor - twolist.h */

..
#include ”fordon.h”
typedef fordon datatyp;
..




/* Specifikation av fordonslistor -- fordlist.h */

#ifndef FORDLISTH
#define FORDLISTH

#include "fordon.h"
#include "twolist.h"

void fyll_fordlist(headtyp *fp);
void visa_fordlist(headtyp *fp);
void fyll_busslist(headtyp *fp, headtyp *bp);

#endif
```

```
/* Implementation av fordonslistor -- fordlist.c */

#include <stdio.h>
#include "fordlist.h"

void fyll_fordlist(headtyp *hfp)
{
   FILE *fil;
   fordon f;
   linktyp *lfp;

   fil = fopen("fordon.txt", "r");
   las_fordon(fil, &f);
   while ( !feof(fil) )
   {
      newlink(&lfp);
      putlink(f, lfp);
      insort(lfp, hfp, is_less_fordon);
      las_fordon(fil, &f);
   }
   fclose(fil);
}

void visa_fordlist(headtyp *hfp)
{
   fordon f;
   linktyp *lfp;

   lfp = firstlink(hfp);
   while ( lfp != NULL )
   {
      f = getlink(lfp);
      skriv_fordon(f);
      lfp = succlink(lfp);
   }
}


void fyll_busslist(headtyp *hfp, headtyp *hbp)
{
   fordon f;
   linktyp *lfp, *lbp;

   lfp = firstlink(hfp);
   while ( lfp != NULL )
   {
      f = getlink(lfp);
      if ( is_a_bus(f) )
      {
          newlink(&lbp);
          putlink(f, lbp);
          inlast(lbp, hbp);
      }
      lfp = succlink(lfp);
   }
}
/* Huvudprogram fordmain.c */
```

```
#include "twolist.h"
#include "fordlist.h"

int main()
{
   headtyp *fordhp, *busshp;

   newhead(&fordhp);
   newhead(&busshp);

   fyll_fordlist(fordhp);
   puts("Fordonslista");
   visa_fordlist(fordhp);
   fyll_busslist(fordhp, busshp);
   puts("\nBusslista");
   visa_fordlist(busshp);

   return 0;
}
```

Ovan använde man union-del i posten för att hantera listor med varierande data i sitt program. Det finns ett alternativt sätt att använda den abstrakta datatypen twolist för att hantera flera listor med olika typer av data i. Man kan som datatyp ange en void-pekare enligt:

```
/* Specifikation av tvåvägslistor - twolist.h */

..

typedef void * datatyp;

..
```

På detta sätt får man en lista innehållande data i form av pekare till aktuella data. Denna lista blir mer generell och kan återanvändas i färdig kompilerad form. Detta är en fördel när man säljer program. Man behöver bara leverera den maskinkodade exekveringsfilen.

Nackdelen med att använda pekare som data i listor är att man ej får någon kompileringssäkerhet. Hela ansvaret för att korrekt data finns i listan överlämnas åt programmeraren.