1 Modulär programutveckling.

När man ska utveckla *stora* program blir det otympligt och ohanterligt att ha all kod i samma fil, speciellt om man är flera personer som utvecklar programmet. Det blir också svårt att hitta i en stor fil då man ska göra förändringar eller rätta fel.

För att underlätta felsökning och underhåll ska man dela upp sin kod i flera separata filer. Hur ska man då göra uppdelningen? I princip skulle man kunna ha en fil för varje funktion, som kompilerades separat och sedan länkades ihop med huvudprogrammet, till ett program. Detta skulle dock innebära en ny svårighet, nämligen att man får för många filer.

För att underlätta införandet av förändringar och felsökning ska man istället försöka samla *alla funktioner som hör ihop* till en fil. Man kan exempelvis samla alla funktioner som *bearbetar samma data*, i form av en abstrakt datatyp, eller också samla alla funktioner som *utför liknande arbetsuppgifter* som in- och utmatning, sortering etc, i form av ett funktionsbibliotek.

För att kunna göra denna typ av uppdelning måste man ha tillgång till en miljö där man har möjlighet att *kompilera enheter separat* och sedan länka ihop delarna till ett exekverbart (körbart) program.

1.1 Separatkompilering

Skriver man stora program ska man dela upp programmet i mindre delar genom att använda funktioner. Man kan ha allt i samma fil vilket fungerar bra för mindre program.

Ex: Skriv ett program som beräknar medelvärdet av en vektor genom att anropa en funktion för medelvärdesberäkningen.

```
/* medvek1.c */

#include <stdio.h>

double medvek(double vek[], int nr)
{
   int i;
   double sum = 0.0;

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

   return sum/nr;
}
```

```
int main()
{
   double vektor[7] = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0 };

   printf("Medelvärdet = %.1f\n", medvek(vektor, 7));

   return 0;
}
```

OBS! Här *definieras* funktionen *före* anropet. Med en funktions *definition* menar man hela funktionen.

(Om man provkör det här exemplet i Microsoft Visual Studio, kan det hända att utskriften dyker upp i ett fönster, som försvinner innan man hinner läsa texten. Lägg in exempelvis ett anrop till funktionen *getch* sist i *main*-funktionen.)

(Kom ihåg att det är *double* som är den ”vanliga” flyttalstypen i C, och den som man normalt ska använda. Datatypen *float* använder man mest om man behöver spara plats.)

Man behöver inte ha hela funktionen definierad före anropet. Det räcker *med funktionens deklaration*. Funktionens deklaration består av *funktionshuvudet följt av ett semikolon* enligt:

```
double medvek(double vek[], int nr);
```

eller bara

```
double medvek(double *, int);
```

Funktionen printf finns på detta sätt *deklarerad* i inkluderings- eller headerfilen stdio.h.

Ex: I ovanstående exempel kan man flytta funktionens definition sist i filen om vi stoppar in en funktionsdeklaration före anropet enligt:

```
/* medvek2.c */

#include <stdio.h>

/* Funktionsdeklaration */
double medvek(double vek[], int nr);

int main()
{
   double vektor[7] = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0 };

   printf("Medelvärdet = %.1f\n", medvek(vektor, 7));

   return 0;
}
```

```
/* Funktionsdefinition */
double medvek(double vek[], int nr)
{
   int i;
   double sum = 0.0;

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

   return sum/nr;
}
```

Varje funktion i C utgör en enhet eller modul som *kan kompileras separat under förutsättning att alla använda datatyper och anropade funktioner är kända.* Efter det att man kompilerat alla funktioner kan man länka ihop alla delar till en körbar enhet. Hur detta görs beror på vilken miljö man arbetar i.

Ex: Skriv ovanstående exempel så att funktionen för beräkning av vektorns medelvärde kompileras separat i en egen fil.

Konstruerar man sitt program enligt principen top-down börjar man med att skriva huvudprogrammet i en separat fil enligt:

```
/* medvmain1.c */

#include <stdio.h>

double medvek(double vek[], int nr);

int main()
{
   double vektor[7] = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0 };

   printf("Medelvärdet = %.1f\n", medvek(vektor, 7));

   return 0;
}
```

Ovanstående program kan man kompilera helt separat och det bildas en *objektkodsfil* som innehåller maskinkoden för programmet. På Windows får den filen namnet *medvmain1.obj* och på Linux och Unix *medvmain1.o*.

Sedan skriver man funktionen medvek i en *ny fil* medvek.c enligt:

```
/* medvek.c */

double medvek(double vek[], int nr)
{
   int i;
   double sum = 0.0;

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

   return sum/nr;
}
```

Denna fil kan också kompileras för sig och det skapas en ny objektkodsfil, medvek.obj (eller medvek.o), som innehåller maskinkoden för funktionen.

För att skapa en *körbar eller exekverbar enhet* har man i sin miljö oftast *laddare eller länkare* som sköter om att plocka ihop delarna och svetsa ihop dessa till en enhet som kan köras. Olika miljöer har olika typer av hjälpmedel för länkningen.

I Visual Studio kan man addera filer till sitt *projekt.* I projektet kompilerar man fil för fil och sedan länkas filerna ihop till en exekverbar fil med kommandot Build

För att huvudprogrammet ska kunna kompileras separat måste funktionsdeklarationen för medvek finnas med i samma fil. Istället för att skriva in funktionsdeklarationer i samma fil som huvudprogrammet skriver man dessa, speciellt om de är många, i en egen headerfil medvek.h enligt :

```
/* medvek.h */

double medvek(double vek[], int nr);
```

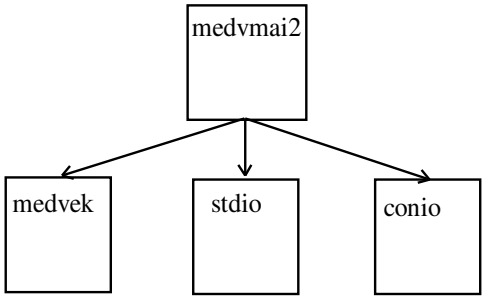
Denna fil sparar man i sin katalog och i huvudprogrammet tas den in av preprocessorn (även kallad förkompilatorn) med hjälp av include-direktivet enligt:

```
/* medvmain2.c */

#include <stdio.h>
#include ”medvek.h”      /* OBS! */

int main()
{
   double vektor[7] = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0 };

   printf("Medelvärdet = %.1f\n", medvek(vektor, 7));

   return 0;
}
```

OBS! *Egna* headerfiler tas in med citationstecken runt filnamnet medan de färdiga biblioteksheaderfilerna tas in som exempelvis <stdio.h>.

OBS! Headerfiler kompileras ej separat utan tas in som vanliga textfiler.

När man använder separatkompilering ska man alltid i *sin dokumentation rita ett beroendediagram som visar vilka delar (moduler, enheter) programmet består av och hur beroendet mellan dessa delar ser ut*. För ovanstående program skulle man på enklast möjliga sätt kunna rita ett beroendediagram enligt:

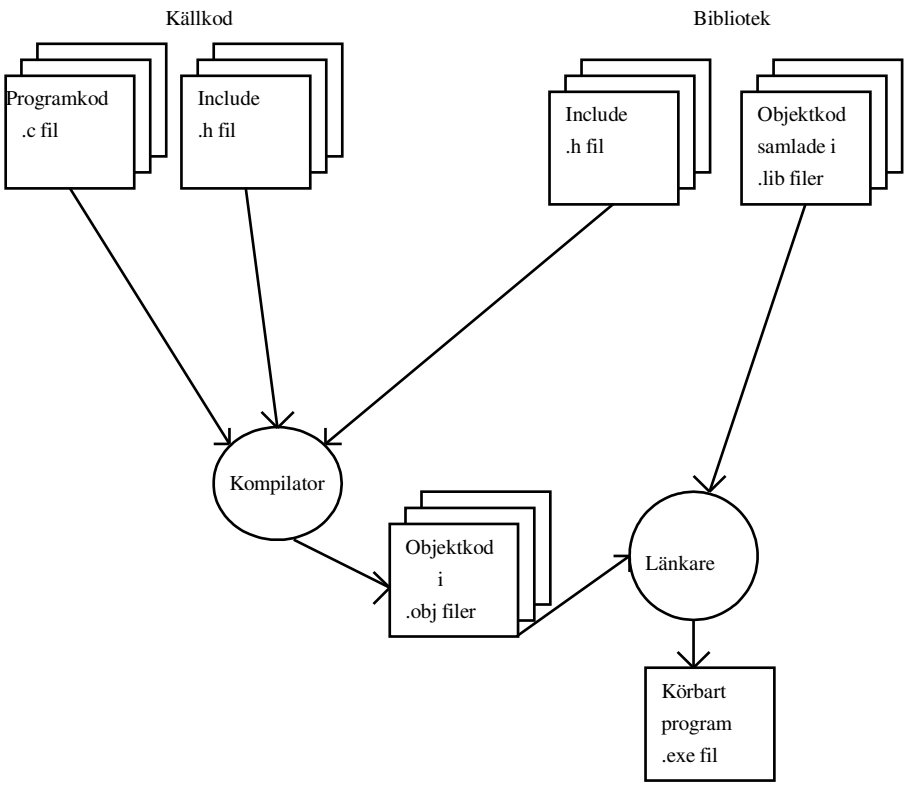


Beroendediagrammet visar vilka delar som programmet består av. Varje del i sin tur består av en specifikationsfil (headerfil, inkluderingsfil) med *deklarationer* och en implementationsfil med *definitioner*.

Ex: Enheten medvek i beroendediagrammet ovan består av filerna medvek.h och medvek.c där medvek.h inkluderas av huvudprogrammet och medvek.c, som kompilerats separat *tas in vid länkningen* med projekthanteraren.

Ex: Standardbiblioteket stdio är också gjort på detta sätt. Den är uppdelad i en specifikationsfil stdio.h och en implementationsfil stdio.c där stdio.h inkluderas av användaren och använda delar av stdio.c, som finns i standardbiblioteket i kompilerad form som stdio.lib*, tas in automatiskt* vid länkningen.

En sammanfattande bild av det hela ser ut som:



1.2 Abstrakta datatyper

Ett sätt att dela upp sitt program i separata delar är att samla data och bearbetning av data i en och samma enhet. *En sådan enhet kallas för en abstrakt datatyp*. En abstrakt datatyp innehåller både datatyp och operationer eller funktioner på data av denna typ. *Samlar man data och operationer i samma enhet blir programmet mycket lättare att underhålla och förändra*. Detta är mycket viktigt idag eftersom *underhållet utgör den största kostnaden för programvara*.

Ex: Skriv ett program som läser in, jämför och skriver ut mätposter innehållande mätvärde och mätställets nummer.

Man börjar med att skriva en headerfil som innehåller deklarationer av typ och funktioner för denna typ av data enligt:

```
/* matvarde.h */

typedef
struct matpost
{
   double varde;
   int nr;
} matvarde;

void las_matvarde(matvarde *mvp);
/* Läser in ett mätvärde från tangentbordet */

void skriv_matvarde(matvarde mv);
/* Skriver ut ett mätvärde på skärmen */

int jfr_varden(matvarde mv1, matvarde mv2);
/* Returnerar 1 om mv1 har mindre mätvärde än mv2 */
```

OBS! Användningen av typedef för att definiera datatypen matvarde. Typedef definierar ett nytt kortare namn matvarde, som ersätter det långa namnet struct matpost på datatypen.

I headerfilen samlar man deklarationerna för data som ska avbilda den abstrakta datatypen samt de funktioner som är aktuella. *Headerfilen är bara en textfil som innehåller typdefinitioner och funktionsdeklarationer. För att fullborda den abstrakta datatypen måste man också implementera operationerna*. Detta gör man i en separat fil matvarde.c, som man sedan kompilerar separat.

```
/* matvarde.c*/

#include <stdio.h>
#include "matvarde.h"    /* OBS */


void las_matvarde(matvarde *mvp)
{
   printf("Ge mätvärde : ");
   scanf("%lf", &mvp->varde);
   printf("Ge mätnummer : ");
   scanf("%d", &mvp->nr);
}


void skriv_matvarde(matvarde mv)
{
   printf("Mätvärde = %f\n", mv.varde);
   printf("Mätnummer = %d\n", mv.nr);
}


int jfr_varden(matvarde mv1, matvarde mv2)
{
   return (mv1.varde < mv2.varde);
}
```

OBS! Man måste inkludera hederfilen matvarde.h för att filen matvarde.c ska kunna kompileras separat. Det som behövs är datatypen matvarde.

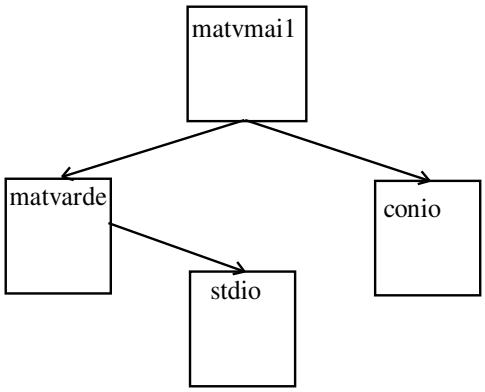
OBS! Normalt länkas funktionerna från standarbiblioteken in automatiskt. Vissa länkare försöker dock optimera koden vid länkningen och bara ta med de funktioner som verkligen används vid körning av programmet. Gamla versioner av Microsoft C kunde glömma att länka med flyttalsversioner av scanf och printf, om man som ovan har pekare till poster som innehåller flyttal. Man måste då använda sig av temporära variabler enligt:

```
void las_matvarde(matvarde *mvp)
{
   double ftemp;

   printf("Ge mätvärde : ");
   scanf("%lf", &ftemp);
   mvp->varde = ftemp;
   ....
}
```

Nu är den abstrakta datatypen matvarde klar och en användare, som kan vara ett huvudprogram eller också en annan abstrakt datatyp, kan utnyttja dess operationer om headerfilen inkluderas och implementationsfilen inlänkas.

```
/* matvmain1.c */

#include "matvarde.h"    /* OBS! */

int main()
{
   matvarde x1, x2;

   las_matvarde(&x1);
   las_matvarde(&x2);
   if (jfr_varden(x1, x2))
   {
      skriv_matvarde(x1);
      skriv_matvarde(x2);
   }
   else
   {
      skriv_matvarde(x2);
      skriv_matvarde(x1);
   }
   return 0;
}
```

Beroendediagrammet för ovanstående program blir:



Har man en gång tillverkat en abstrakt datatyp ska man naturligtvis utnyttja den då man använder denna typ av data. Abstrakta datatyper är , förutom att de är lätta att underhålla, en viktig ingrediens i möjligheten att återanvända kod.

Ex: Skriv ett program som skapar en vektor av mätvärden enligt ovan och sorterar dessa efter mätvärdena i stigande ordning och skriver ut de sorterade mätvärdena.

Man skapar en ny abstrakt datatyp som hanterar operationer för vektorer av mätvärden enligt:

```
/* matvek.h */

#include "matvarde.h"

void las_matvek(matvarde v[], int nr);
/* Läs nr st mätvärden */

void skriv_matvek(matvarde v[], int nr);
/* Skriv nr st mätvärden */

void sort_matvek(matvarde v[], int nr);
/* Sortera vektorn v */
```

```
/* matvek.c */

#include "matvarde.h"

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

   for ( i = 0; i < nr; i++)
      las_matvarde(&v[i]);
}


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

   for ( i = 0; i < nr; i++)
      skriv_matvarde(v[i]);
}
```

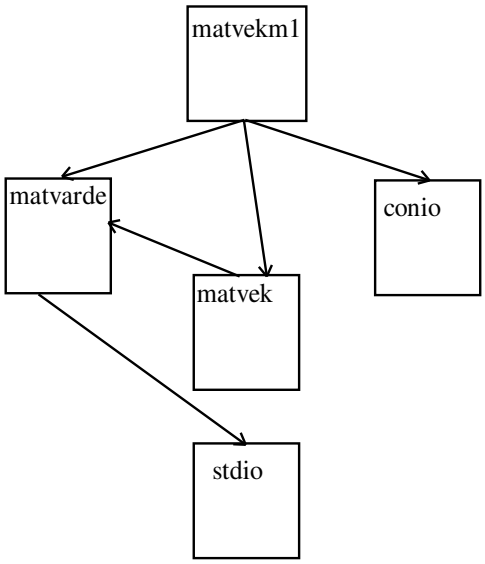
```
void sort_matvek(matvarde v[], int nr)
{
   int i, j, minelemi;
   matvarde temp;

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

I huvudprogrammet inkluderar man de enheter som man behöver och utnyttjar data och funktioner för de abstrakta datatyperna enligt:

```
/* matvekm1.c */

#include "matvarde.h"    /* OBS! */
#include "matvek.h"      /* OBS! */

int main()
{
   matvarde vek[6] = { {3.4, 12}, {2.3, 23}, {1.2, 13},
                       {5.6, 34}, {4.5, 25}, {3.1, 16} };

   sort_matvek(vek, 6);
   skriv_matvek(vek, 6);
   return 0;
}
```

OBS! Man måste ha med matvarde.h för att kompilatorn ska känna igen datatypen matvarde och matvek.h för funktionerna sort_matvek och skriv_matvek ska finnas deklarerade före anropet.

Beroendediagrammet för ovanstående program blir :



Här stöter man på ett problem vid kompileringen av huvudprogrammet matvekm1.c. Man får ett felmeddelande om att man definierat typen matvarde dubbelt. Detta beror på att filen matvarde.h *inkluderas dubbelt,* först direkt av huvudprogrammet och sedan indirekt av matvek.h. För att undvika ovanstående fel ska man bara inkludera matvek.h i huvudprogrammet. Har man många inkluderingsfiler, blir det dock svårt att hålla reda på vad som ska inkluderas och vad som inte ska. Ett bättre sätt är att i inkluderingsfilen matvarde.h sätta in ett kompilatordirektiv om *villkorlig kompilering* enligt :

```
/* matvarde.h */

#ifndef MATVARDE_H
#define MATVARDE_H

typedef
struct matpost
{
   double varde;
   int nr;
} matvarde;

void las_matvarde(matvarde *mvp);
void skriv_matvarde(matvarde mv);
int jfr_varden(matvarde mv1, matvarde mv2);

#endif
```

OBS! Har man *en gång* definierat namnet MATVARDE_H, kommer kompilatorn fortsättningsvis att hoppa över inkluderingsfilen fram till endif. Man bör *ta som vana* att sätta in villkorlig kompilering *i alla sina inkluderingsfiler*.

Vill man i ovanstående exempel sortera mätvektorn efter mätnummer istället för efter mätvärde måste man gå in i funktionen sort_matvek och ändra jämförelsen till att gälla mätvärdets nummer istället för värde enligt :

```
.....

if ( jfr_nr(v[j], v[minelemi]) )
   minelemi = j;

.....
```

Vill man sortera både efter mätvärde och mätnummer måste man då skriva två olika sorteringsfunktioner, med var sin aktuella jämförelse. Detta verkar vara onödig kopiering av kod och onödigt arbete. Det finns ett bättre sätt i språket C. Istället för att skriva två sorteringsfunktioner använder man en *mer generell sådan dit man skickar med den aktuella jämförelsefunktionen som parameter*.

Ex: Skriv ett program som initierar en mätvektor med mätvärden och mätnummer och sedan skriver ut vektorn först sorterad efter mätvärden och sedan efter mätnummer.

Eftersom det mesta av programmet redan gjorts kan man betrakta programmet top-down och börja med huvudprogrammet:

```
/* matvekm2.c */

#include "matvarde.h"
#include "matvek.h"

int main()
{
   matvarde vek[6] = { {3.4, 12}, {2.3, 23}, {1.2, 13},
                       {5.6, 34}, {4.5, 25}, {3.1, 16} };

   sort_matvek(vek, 6, jfr_nr);
   skriv_matvek(vek, 6);

   sort_matvek(vek, 6, jfr_varden);
   skriv_matvek(vek, 6);

   return 0;
}
```

Samma sorteringsfunktion anropas men med olika aktuella jfr-funktioner. Denna sorteringsfunktion måste naturligtvis finnas med i den abstrakta datatypen matvek enligt:

```
/* matvek.h */

#include "matvarde.h"

void las_matvek(matvarde v[], int nr);
/* Läs nr st mätvärden */

void skriv_matvek(matvarde v[], int nr);
/* Skriv nr st mätvärden */

void sort_matvek(matvarde v[], int nr,
                 int (*jfr)(matvarde mv1, matvarde mv2));
/* Sortera v med aktuell jämförelsefunktion jfr */
```

```
/* matvek.c */

#include "matvek.h"

......

void sort_matvek(matvarde v[], int nr,
                 int (*jfr)(matvarde mv1, matvarde mv2))
{
   int i, j, minelemi;
   matvarde temp;

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

Dessutom måste de två jämförelsefunktionerna jfr_nr och jfr_varden finnas med för mätvärden enligt:

```
/* matvarde.h */

#ifndef MATVARDEH
#define MATVARDEH

typedef
struct matpost
{
   double varde;
   int nr;
} matvarde;

void las_matvarde(matvarde *mvp);
void skriv_matvarde(matvarde mv);
int jfr_varden(matvarde mv1, matvarde mv2);
int jfr_nr(matvarde mv1, matvarde mv2);

#endif



/* matvarde.c*/

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


......


int jfr_varden(matvarde mv1, matvarde mv2)
{
   return (mv1.varde < mv2.varde);
}

int jfr_nr(matvarde mv1, matvarde mv2)
{
   return (mv1.nr < mv2.nr);
}
```

Nu har man fått en något så när generell abstrakt datatyp, som också lätt kan utökas med nya data i form av termer till posten. Ändrar man på data i posten matvarde räcker det nu att göra ändringar i den abstrakta datatypen matvarde som består av två filer matvarde.h och matvarde.c. *Ju färre filer man måste gå in i när man gör förändringar desto bättre*. Filerna får dessutom inte vara för stora.

1.3 Funktionsbibliotek

Ett annat sätt att dela upp sitt program i mindre delar är att samla ihop *funktioner av samma sort* i samma enhet. *Dessa enheter kallas för funktionsbibliotek*. Funktionsbibliotek kan man ha för sortering, sökning, beräkning, användargränssnitt mm. Sådana bibliotek kan användas gång på gång och man sparar mycket tid som programmerare.

Funktionsbibliotek skapar man i form av en specifikationsfil med tillhörande implementationsfil.

Ex: Sorteringsfunktioner kan man samla tillsammans i en enhet enligt :

```
/* sort.h */

void ursort(int v[], int nr);
/* Sorterar nr st element i v med urvalssortering */

void bubbsort(int v[], int nr);
/* Sorterar nr st element i v med bubbelsortering */
```

```
/* sort.c */

#include ”sort.h”

void ursort(int v[], int nr)
{
   int i, j, minelemi, temp;

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

```
void bubbsort(int v[], int nr)
{
   int i = 0, maxi = nr - 1, bubbel = 1, temp;

   while ( bubbel && maxi > 0)
   {
      bubbel = 0;
      for (i = 0; i < maxi; i++)
      {
         if ( v[i] > v[i + 1] )
         {
            temp = v[i];
            v[i] = v[i + 1];
            v[i + 1] = temp;
            bubbel = 1;
         }
      }
      maxi--;
   }
}
```

Ovanstående funktionsbibliotek med sorteringsfunktioner kan kompileras separat och länkas in i ett projekt tillsammans med ett huvudprogram enligt exempelvis:

```
#include <stdio.h>
#include ”sort.h”        /* OBS! */

int main()
{
   int i, vek[8] = { 23, 54, 12, 45, 67, 18, 24, 45};

   bubbsort(vek, 8);
   for(i = 0; i < 8; i++)
      printf(”%5d\n”, vek[i]);
   return 0;
}
```

Sorteringsfunktionerna ovan är ej generella. Den enda typ av vektor som går att sortera är en heltalsvektor. Man kan ej anropa bubbsort för att sortera våra mätvärden i mätvektorn ovan exempelvis. *Ett sätt att göra funktionsbibliotek mer generella är att skriva funktionerna för en generell datatyp och sedan i headerfilen anpassa aktuell data till den generella datatypen*.

Dessutom ska man naturligtvis som ovan visats *införa jämförelsefunktionen som parameter* till sorteringsfunktionen.

Ex: Skriv om rutinerna i sort-modulen så att de kan användas för sortering av olika typer av data som exempelvis för vektorer av våra mätvärden.

```
/* sort.h */

/* OBS! Denna del ändras till den aktuella datatypen */

#include ”matvarde.h”
typedef matvarde dtyp;

/* OBS! Slut på den del som ska ändras */


void ursort(dtyp v[], int nr, int (*jfr)(dtyp d1, dtyp d2));
/* Sorterar nr st element i v med urvalssortering */

void bubbsort(dtyp v[], int nr, int (*jfr)(dtyp d1, dtyp d2));
/* Sorterar nr st element i v med bubbelsortering */
```

```
/* sort.c */

#include ”sort.h”

void ursort(dtyp v[], int nr, int (*jfr)(dtyp d1, dtyp d2))
{
   int i, j, minelemi;
   dtyp temp;

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

```
void bubbsort(dtyp v[], int nr, int (*jfr)(dtyp d1, dtyp d2))
{
   int i = 0, maxi = nr - 1, bubbel = 1;
   dtyp temp;

   while ( bubbel && maxi > 0)
   {
      bubbel = 0;
      for (i = 0; i < maxi; i++)
      {
         if ( jfr(v[i], v[i + 1]) )
         {
            temp = v[i];
            v[i] = v[i + 1];
            v[i + 1] = temp;
            bubbel = 1;
         }
      }
      maxi--;
   }
}
```

```
/* matvekm3.c */

#include "matvarde.h"
#include "matvek.h"
#include ”sort.h”

int main()
{
   matvarde vek[6] = { {3.4, 12}, {2.3, 23}, {1.2, 13},
                       {5.6, 34}, {4.5, 25}, {3.1, 16} };

   ursort(vek, 6, jfr_nr);
   skriv_matvek(vek, 6);
   bubbsort(vek, 6, jfr_varden);
   skriv_matvek(vek, 6);
   return 0;
}
```

OBS! Hur man måste gå in i sort.h för att ändra inkluderingsfil och aktuell datatyp. Ändringarna begränsas dock till detta. Man behöver ej gå in i sorteringsalgoritmerna och göra ändringar.

(Det här kändes kanske lite krångligt, och det beror ganska mycket på att detaljerna blir krångliga i C. Själva tanken med generiska funktioner och generiska datatyper är dock rätt enkel, för det handlar bara om att man kan använda samma funktion till att göra samma sak med flera olika sorters data. Språk som C++ och Java har mer lättanvända mekanismer för att göra samma sak.)

Ex: Skriv ett funktionsbibliotek som beräknar nollställen till en funktion samt använd detta för att lösa ekvationen

$$e^{-x} = 2\cos x$$

i intervallet mellan x = 1 och x = 2.

```
/* nollfunk.h */


double intervall(double x1, double x2, double (*f)(double x),
                 double e);
/* Intervallhalvering ger nollställe till f i
   intervallet x1 < x < x2 med fel på max e */



double newton(double x1, double (*f)(double x),
              double (*d)(double x), double e);
/* Newton-Raphsons metod ger nollställe till f
   vars derivata är d med ett fel på max e */



double sekant(double x1, double x2, double (*f)(double x),
              double e);
/* Sekantmetoden ger nollställe till f med fel på max e */
```

Eftersom headerfilen är den fil som en användare tittar på, då man söker lämpliga funktioner, ska den innehålla information om vad funktionerna gör och hur de anropas. Man bör skriva kortfattat men så att användaren förstår. Hur de anropas framgår oftast av parameterlistan.

Funktionerna implementeras sedan i filen nollfunk.c enligt:

```
/* nollfunk.c */

#include <math.h>
#include ”nollfunk.h”

double intervall(double x1, double x2, double (*f)(double x),
                 double e)
{
   double xm, ym;

   do
   {
      xm = (x1 + x2) / 2;
      if ( f(xm)*f(x1) > 0 )
        x1 = xm;
      else
        x2 = xm;
   }
   while ( fabs(x2-x1) >= e);
   return xm;
}


double newton(double x1, double (*f)(double x),
              double (*d)(double x), double e)
{
   double x = x1 - f(x1) / d(x1);

   while ( fabs(x - x1) >= e )
   {
      x1 = x;
      x = x1 - f(x1)/d(x1);
   }
   return x;
}


double sekant(double x1, double x2, double (*f)(double x),
               double e)
{
   double x = x2 - f(x2)*(x1 - x2) / (f(x1) -f(x2));

   while ( fabs(x - x2) >= e )
   {
      x1 = x2;
      x2 = x;
      x = x2 - f(x2)*(x1 - x2) /(f(x1) -f(x2));
   }
   return x;
}
```

Huvudprogrammet, som ska lösa ekvationen ovan, skrivs nu så att den anropar alla funktioner i biblioteket för att testa dessa. Annars skulle det naturligtvis räcka med att anropa en. Den funktion som används mest är sekant-metoden.

```
/* nollst.c */

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

double funk(double x)
{
   return exp(-x) - 2*cos(x);
}

double der(double x)
{
   return 2*sin(x) - exp(-x);
}

int main()
{
   printf("Med intervall blir x = %.3f\n",
            intervall(1, 2, funk, 0.0005));

   printf("Med Newton-Raphson blir x = %.3f\n",
            newton(1, funk, der,  0.0005));

   printf("Med sekantmetoden blir x = %.3f\n",
            sekant(1, 2,funk, 0.0005));

   return 0;
}
```

OBS! Den aktuella funktionen vars nollställe utgör lösningen på ekvationen skrivs ovan in i samma fil som huvudprogrammet. Man kan naturligtvis ta in den som en separat enhet om man vill det.