15 typiska uppgifter till praktiska tentan, del A.

Följande är tänkt att tjäna som instuderingsfrågor inför den praktiska tentamen. Troligtvis ligger dock uppgifterna här på en högre nivå än de som kommer på tentamen i det att det är väldigt mycket text till dessa uppgifter. Uppgifterna på tentan kommer att ha mindre text.

Historiskt sett har studenter haft svårt att förbereda sig inför den praktiska tentamen eftersom denna tidigare innehållit uppgifter av ett allmänt slag där det inte på förhand varit tydligt vilket delområde av kursen uppgifterna hört till. Studenter har många gånger haft svårt att veta hur man ska angripa dessa uppgifter även om man (enligt min mening) inhämtat kursens innehåll i en god grad. Dessutom har det visat sig att studenter som klarat praktiska tentan faktiskt haft vissa brister i programmeringskunskaperna som det är menat att man ska ha inhämtat efter genomgången programmeringskurs. För att råda bot på båda dessa svårigheter presenteras här en speciell form av problem som befattar sig med kärnkunskaperna i programmeringskursen. Problem av nedanstående typ kommer att komma på en A-del på praktiska tentan och man måste får ett godkänt betyg på A-delen för att B-delen ska rättas. B-delen kommer att innehålla generella problem utan ledningar till hur de ska lösas, så som praktiska tentor såg ut i den här kursen tidigare. De tre huvudområdena är beskrivna i ett tidigare dokument och de är:

I. Sortering/Sökning: IIII
II. Representation/Omvandling/format/protokoll: IIIII
III. Strukturering: IIIII

Antal problem i varje område är noterat efteråt, således hör de fyra första problemen till delområdet sortering/sökning, de nästföljande sex hör till representation etc. Observera dock att samtliga problem kan ha (och har ofta) inslag från samtliga tre områden.

Det här dokumentet ingår i ett förnyelsearbete som i skrivande stund (oktober 2011) är påbörjat. Det betyder att uppgifterna på den kommande tentamen kan komma att se ut på detta sätt, men de kan också komma att skilja sig något. Framförallt tror jag att uppgifterna nedan är ganska tungviktiga och således inte lämpliga som tentamensuppgifter. Några är dock absolut lämpliga och samtliga uppgifter är framtagna för att träna det centrala i kursen som absolut krävs för lägsta godkända betyget E på praktiska tentan.

I alla problem, där bokstäver och strängar förekommer, antar vi att vi bara använder det engelska alfabetet, a-z, respektive A-Z. De svenska bokstäverna å, ä, ö samt Å, Ä och Ö används inte.

I. Sortering/Sökning

1. På filen *trianglar.dat* finns ett antal trianglar sparade. Filen är en binärfil baserad på structen

```
struct triangel
{
     float a, b, c;
};
```

där `a`, `b` och `c` är flyttal som anger sidornas längder. Arean hos en triangel anges med den så kallade *Herons formel* som har följande utseende, *A=sqrt(p*(p-a)*(p-b)*(p-c))* där *p* är talet som bildas

genom *p=(a+b+c)/2*. Sortera denna fil av trianglar så att den resulterande filen består av trianglar vars areor stiger. Den sorterade filen ska heta *trianglar2.dat* och också vara en binärfil på samma format som *trianglar.dat*.

2. I en binärfil baserad på structen

```
struct person
{
     char fornamn[20];
     char efternamn[20];
     char telefonnummer[20];
};
```

finns ett antal personer lagrade. Läs in denna fil och spara innehållet i ett textfil där en person sparas på en rad enligt formatet

```
efternamn, fornamn, tel: telefonnummer
```

Filens innehåll ska vara sorterat på *efternamn*.

3. En punkt i planet är ett matematiskt objekt representerat av två tal, *(x, y)*. Avståndet mellan två punkter *(x1, y1)* och *(x2, y2)* ges av uttrycket *sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2))*. I en textfil `punkter.txt` finns ett antal punkter representerade på formen:

```
(2,4)
(5,6)
(7,8)
```

alltså koordinaterna separerade av kommatecken och omgivna av parenteser. (Det är inte exakt dessa punkter som förekommer i filen.) Skriv ett program som tar in en till punkt från användaren och sorterar textfilen baserat på avståndet till den nya punkten som användaren matar in. Om användaren till exempel matar in punkten `(5,4)` skulle ordningen bli `(5,6) (2,4) (7,8)` eftersom punkten `(5,6)` ligger närmast `(5,4)` och `(7,8)` ligger längst ifrån `(5,4)`. Om användaren matar in en annan punkt kan ordningen bli en annan, ordningen ska som sagt baseras på avståndet till den punkt som användaren matar in och den punkt som har minst avstånd till den inmatade punkten ska komma först. (I denna uppgift antar vi också att en punkts koordinater alltid är heltal, i verkligheten kan förstås en punkts koordinater vara flyttal.) Presentationen av resultatet ska vara en utskrift på skärmen på det format som anges ovan (alltså en punkt per rad).

4. Funktionen `strstr()` finns i biblioteket `string.h` tillsammans med andra funktioner som `strlen()` (som returnerar längden på en sträng) och `strcmp()` (som jämför två strängar.) Andra strängar som också finns i detta bibliotek är `strcat()` och `strcpy()`. De sista fyra funktionerna har vi troligen använt ganska flitigt i våra laborationer. Det som funktionen `strstr()` gör är att avgöra om en sträng är delsträng av en annan sträng, dess fullständiga funktionsprototyp är

```
char *strstr(const char *haystack, const char *needle);
```

och texten som beskriver funktionens beteende är

```
The  strstr()  function finds the first occurrence of the
substring needle in the string haystack. The terminating '\0'
characters are not compared. [The function] return a pointer
to the beginning of the substring, or NULL if the substring is
not found.
```

Alltså, om man skriver

```
strcpy(str1,”XXHAHAXXX”);
pos = strstr(str1, “HA”);
```

så kommer `pos` att peka in i på den position i `str1` där delsträngen `HA` börjar, det vill säga `pos` kommer att får värdet `&(str1[2])`. (Vi antar att `pos` är deklarerad som `char *`.) Om vi hade koden

```
strcpy(str1,”XXHAHAXXX”);
pos = strstr(str1, “HIHI”);
```

skulle istället `pos` fått värdet `NULL` eftersom `“HIHI”` inte är delsträng i `”XXHAHAXXX”`.

Skriv ett program som använder strstr för att ta fram alla strängar ur `SAOL12.TXT` som innehåller en viss delsträng och skriv ut dessa sorterade på stränglängde med kortaste strängen först.

En testkörning:

```
Ange delsträng: OJI

MOJIG
NOJIG
SKOJIG
STOJIG
SKOJIGHET
PROJICERA
STOJIGHET
PROJICERING
```

(Anmärkning: `SAOL12.TXT` innehåller ord med de svenska vokalerna, Å, Ä och Ö, dessa kan vara instabila under standard-*C*, i ett POSIX-system som jag använder var det svårt att skriva ut dem på skärmen, jag fick därför leta lite efter exempel som hade en skärmutskrift konsistent med hur programmet är tänkt att fungera.)

II. Representation/Omvandling/format/protokoll

5. En rätvinklig triangel kan karakteriseras av att man anger två av dess sidor: antingen två kateter, eller en katet och hypotenusan. En skriv en funktion `omvandla()` med följande prototyp:

```
omvandla (struct kh_triangel * t1, struct kk_triangel *t2);
```

Här är structarna:

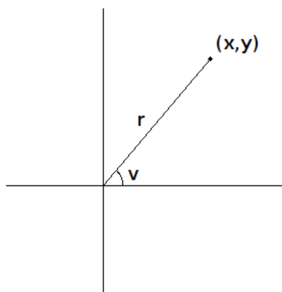
```
struct kh_triangel
{
     float k, h;
};
```

```
struct kk_triangel
{
     float a, b;
};
```

Den ena structen representerar en rätvinklig triangel som är given med en katet och hypotenusan (`kh_triangel`) och den andra structen representerar en rätvinklig triangel som är given med båda kateterna (`kk_triangel`).

Funktionen `omvandla` ska acceptera referenser till två olika structar som sina parametrar. Båda dessa structar består (som vi ser) av två flyttal. I ett korrekt anrop av funktionen ska den ena av dessa structar ska bara innehålla nollor. Den andra ska innehålla två tal skilda från noll. Den struct som bara innehåller nollor ska vara målstruct, det vill säga funktionen ska ta den andra structen och omvandla representationen av en rätvinklig triangel så att den struct med nollor får nya tal som representerar den rätvinkliga triangel som representeras av structen med tal skilda från noll. I en rätvinklig triangel anses `a` och `b` vara kateterna och `c` är hypotenusan. De är då relaterade med Pythagoras sats: `a*a + b*b = c*c`. (Om man känner två av `a`, `b` och `c` kan man därmed alltid beräkna den tredje.)

6. Punkter i planet kan anges genom rätvinkliga eller polära koordinater. Rätvinkliga koordinater är det som vi är vana vid, en x- och en y-koordinat. Med dessa kan man ange var en punkt i planet befinner sig. Med de polära koordinaterna kan man också ange var en punkt i planet befinner sig, men man anger på ett annat sätt, de polära koordinaterna består av att man anger avståndet till origo, markerat med `r` i figuren, samt vinkeln som en linje från origo till punkten bildar mot positiva x-axeln, markerat med `v` i figuren. För enkelhetens skull ska vi anta att x alltid är positivt.



De matematiska sambanden (uttryckt i *C*-funktioner) mellan `x`, `y`, `r` och `v` är `x=r*cos(v)`, `y=r*sin(v)` samt `r=sqrt(x*x+y*y)` och `v=atn(y/x)`.

I *C* kan vi uttrycka punkter på rätvinklig eller polär form så här:

```
struct point_xy {
     float x, y;
};

struct point_rv {
     float r, v;
};
```

Skriv två funktioner med följande funktionsprototyper:

```
convert_xy_to_rv(struct point_xy * pxy, struct point_rv * prv);
convert rv_to_xy(struct point_rv * prv, struct point_xy * pxy);
```

som fungerar som omvandlingsfunktioner, den övre (`convert_xy_to_rv`) tar en punkt angiven på rätvinkliga koordinater (som `pxy` pekar på) och omvandlar den till polära koordinater och lagrar resultatet i den struct som `prv` pekar på. Den undre funktionen ska göra samma sak fast åt andra hållet, det vill säga omvandla från polära koordinater till rätvinkliga och alltså ta den polärt angivna punkten som kommer in via `prv` och omvandla representationen till rätvinkliga koordinater och lagra resultatet i `pxy`. Anpassa funktionerna så att följande huvudprogram fungerar:

```
main()
{
     struct point_xy;
     struct point_rv;

     printf("Testing xy to rv.\n");
     printf("x: "); scanf("%f", &(point_xy.x));
     printf("y: "); scanf("%f", &(point_xy.y));

     convert_xy_to_rv(&point_xy,&point_rv);

     printf("\nr: %f\nv: %f\n\n", point_rv.r, point_rv.v);

     printf("Testing rv to xyrv.\n");
     printf("r: "); scanf("%f", &(point_rv.r));
     printf("v: "); scanf("%f", &(point_rv.v));

     convert_rv_to_xy(&point_rv,&point_xy);

     printf("\nx: %f\ny: %f\n\n", point_xy.x, point_xy.y);

}
```

Du får inte ändra något i ovanstående program, förutom att lägga till de funktioner som du ska skriva förstås, Omvandlingarna som utförs av funktionerna ska förstås fungera korrekt.

7. //Denna uppgift är inte riktigt klar... troligtvis är den lite för svår. // En matematisk funktion kan, under vissa förhållanden, representeras av en uppräkning av de värden som den antar. Det går dock inte så bra att göra detta för en funktion som antar flyttal, eftersom det finns så många flyttal. Men vi kan komma hyfsat nära med följande struct:

```
struct function {
     double start, end, step;
     double *values;
};
```

Vi ska beskriva hur denna struct kan representera en funktion. Vi tänker oss då att det som representeras hos en funktion är värdena i ett visst intervall, alltså alla tal mellan två gränser. Den nedre gränsen av detta intervall anges av `start` och den övre gränsen av detta intervall anges av `end`. I denna representationsform tänker vi oss också funktionens värden angivna på delintervall av intervallet med gränserna `start` och `end` och talet `step` anger hur långa delintervallen är (alla har samma längd). Om vi till exempel har `start`, `end` och `step` givna av 0, 10 och 0.5 representerar vi alltså en funktion som börjar i 0 och slutar i 10 och vi anger dess värden på delintervall av längden 0.5. Vi anger alltså värden som ska tänkas gälla i hela delintervall och vi tänker oss att värdena som anges är funktionsvärdet i de vänstra ändpunkterna av delintervallen. De värden som anges med denna representationsform blir alltså funktionsvärdena i punkterna 0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0, 7.5, 8.0, 8.5, 9.0 och 9.5. Det är totalt 20 stycken värden och det antalet (alltså hur många de är) kan fås genom att bilda uttrycket `(end-start)/step`. Vi ska alltså spara 20 funktionsvärden, dessa värden sparas då i `values` som är en array, här deklarerad som en pekare. Om vi har funktionen $y = 2*x$ och `start`, `end` och `step` givna som ovan ska alltså values vara en array med värdena 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0 och 19.0.

En funktion finns lagrad i en binärfil baserad på ovanstående struct och *C*-funktionen `loadfunc()` är given som laddar in en matematisk funktion av ovanstående sort i en variabel av typen `struct function`. Den har prototypen

```
loadfunc(FILE * funcfile, struct function * func);
```

Skriv ett program som använder sig av `loadfunc()` och som innehåller ytterligare två funktioner, som heter `value_at()` och `area()`. Den första funktionen, `value_at()`, ska ha prototypen

```
double value_at(struct function *f, double x)
```

och returnera funktionsvärdet som finns lagrat för det intervall som x ligger i. Om vi hade skickat in x=3.75 så hade vi fått tillbaka värdet 7.0 från value_at eftersom 3.75 ligger i intervallet med vänstra ändpunkt 3.5 och här anses funktionen ha värdet 2*3.5 = 7.0. Den riktiga matematiska funktionen y = 2*x har värdet 2*3.75 = 7.5 i denna punkt, men med den representationsform vi har här tappar vi en del noggrannhet. I denna representationsform anses alltså funktionen vara konstant på de delintervall som gäller i representationsformen. Den andra funktionen som heter `area()` ska ha följande prototyp:

```
double area(double a, double b, struct function *f)
```

och returnera arean under funktionsgrafen mellan x-värdena a och b. Eftersom då anses vara en summa av rektangelareor blir det ganska lätt att beräkna.

Funktionen loadfunc() ges här:... denna uppgift är inte riktigt klar... som sagt... troligtvis är den väsentligt för svår.

8. Kommandoradsargument

Då ett *C*-program startar anropas först funktionen main(). Normalt sett har vi inte sett att main() har några argument, men man kan skriva program där main() faktiskt får argument. De argument som main() tar emot skriver man på kommandoraden då man startar programmet. Det kan vara svårt att demonstrera detta i en integrerad miljö som *CodeBlocks* så vi rekommenderar att ni gör denna uppgift på kommandoraden.

För att lämna kommandoradsargument till ett *C*-program deklarerar vi main() så här:

```
int main(int antal_argument, char* argument[])
```

det vill säga main() förväntar sig först ett heltal (antal_argument) följt av en array av strängar. Här är arrayen deklarerad som en pekare, arrayer kan ju hanteras på det viset.

Kommandoradsargument hanteras som strängar och vi refererar till de enskilda strängarna med arrayen argument. Den har då elementen argument[0], argument[1] osv upp till och med argument[antal_argument-1], de är antal_argument stycken. Vi tittar genast på ett exempel:

```
#include <stdio.h>

int main(int antal_argument, char* argument[])
{
  int i;

  printf("Antal argument: %d.\n", antal_argument);

  for(i=0;i<antal_argument;i++)
    printf("Argument %d: %s.\n", i, argument[i]);
}
```

Och här är en provkörning:

```
$ ./a.out kalle olle lisa 123
Antal argument: 5.
Argument 0: ./a.out.
Argument 1: kalle.
Argument 2: olle.
Argument 3: lisa.
Argument 4: 123.
```

Observera att programmet räknar med att även programnamnet `./a.out` räknas som ett kommandoradsargument, det kan ibland vara bra för ett program att veta hur det anropats. (Vi kommer inte att ha någon användning av det i vår uppgift här dock.) Vi ser att arrayen `argument[]` då fungerar som en array av strängar. Denna array skapas automatiskt av operativsystemet och den finns innan programmet kör igång.

Uppgift: Skriv om uppgift 4 (ovan, den om delsträngar) så att man anger delsträngar att söka efter (i ord i `SAOL12.TXT`) som kommandoradsargument (istället för att mata in dem i en dialog med användaren), detta ska möjliggöra att söka efter flera delsträngar. Skriv två varianter av programmet, dels en där det krävs att alla delsträngar är med och dels en där bara en av delsträngarna ska vara med. Anropet till det ena programmet (som kräver att alla delsträngar ska finnas med) ska se ut så här:

```
$ ./sok_ord_som_har_alla TRE GS
```

ska ge alla ord som har alla delsträngarna TRE och GS, det är orden

```
FLYGSTREJK
SKUGGSTRECK
TRESTEGSHOPP
TREVGSKRAN
TREVNINGS
```

Det andra anropet kan se ut så här

```
$ ./sok_ord_som_har_nagot_av KATTER GREJ
```

och det ska resultera i alla ord som har någon av de angivna delsträngarna som angivits som kommandoradsargument. En körning skulle se ut ungefär så här:

```
GREJ
GREJA
INNEGREJ
MLARGREJER
MLARGREJOR
PANGGREJ
RAKGREJER
RAKGREJOR
SKATTERABATT
SKATTERAZZIA
SKATTEREFORM
SKATTEREGLER
SKATTERTT
SMINKGREJER
SMINKGREJOR
TOPPENGREJ
```

Återigen ser vi att Å, Ä och Ö krånglar till det lite, till exempel står ordet `SKATTERTT` för SKATTERÄTT. (Och `MLARGREJER` är förstås MÅLARGREJER etc.)

9. Vi definierar följande struct

```
struct flygplan
{
  char beteckning[19];
  double x;
  double y;
  double hojd;
};
```

för att beskriva ett flygplan. Ett flygplan har en beteckning (strängen `beteckning`) en x- och en y-koordinat och en hojd. Fyra medlemmar beskriver alltså dessa saker.

Skriv fyra funktioner som hanterar structar och arrayer av structar av flygplan enligt följande:

```
double avstand(struct flygplan *f1, struct flygplan *f2);
int samma_flygplan(struct flygplan *f1, struct flygplan *f2);
farligt_nara(struct flygplan lista[], int antal, int *f1, int *f2);
las_in_flygplan(char filnamn[], struct flygplan lista[], int * antal);
```

Vi ska beskriva vad dessa fyra funktioner ska göra. Funktionen `avstand` ska beräkna avståndet mellan två givna flygplan och avståndet ges då av

```
sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2)+(hojd1-hojd2)*(hojd1-hojd2))
```

(där koordinaterna med 1:or på hör till flygplanet som `f1` refererar till och koordinaterna med 2:or på hör till flygplanet som `f2` refererar till. Funktionen `samma_flygplan` ska returnera talet 1 om de båda flygplanen har samma `beteckning`, annars 0 och funktionen `farligt_nara` ska via referensparametrarna `f1` och `f2` i returnera index (alltså ordningsnummer i listan `lista`) på flyplan par av som befinner sig närmare varandra än 1000 meter. Man ska kunna ha respons från programmet så här:

```
Varning: Flygplanen SE-XDE och SE-KJU ligger 455 meter från varandra.
```

Om inga par av flygplan befinner sig farligt nära varandra ska funktionen via båda referensparametrarna returnera värdet -1. Funktionen `las_in_flygplan` ska ta ett filnamn som parameter och läsa in, från en textfil, ett antal flygplan givna på formatet:

```
beteckning
x
y
hojd
beteckning
x
y
hojd
...
beteckning
x
y
hojd
```

Funktionen ska returnera antal inlästa flygplan i referensparametern `antal` och själva flyplanen förstås via referensparametern `lista`.

10. Kortspel. Vi ska spela kort igen men vi ska hitta ett sätt att bättre representera kort så att vi kan ange att ett kort är i vissa positioner. Vi deklarerar därför en struct på följande sätt:

```
struct kort
{
  int farg;
  int valor;
  int position;
};
```

Heltalen `farg` och `valor` betyder som förut färg och valör, alltså 0-3 för hjärter, spader, ruter och klöver och valör har ett av värdena 1-13, alltså ess till och med kung. Nytt här är medlemmen `position` som anger var kortet befinner sig.

`position` = -1 betyder att kortet utgått ur spelet
`position` = 0 betyder att kortet är hos givaren som ska ge kort till spelare
`position` = 1 betyder att koret är hos spelare 1
`position` = 2 betyder att koret är hos spelare 2 osv.

Följande gång är tänkt: man deklarerar en array av kort på följande sätt:

```
struct kort kortlek[52];
```

denna initieras med en init-funktion så att det finns 13 kort av vardera färgerna. Sedan frågar programmet efter hur många spelare som ska vara med. Vi kan tänka oss att vi ska spela poker, det vill säga varje hand ska ha 5 kort och vi kan ha maximalt fem spelare. Sedan ska programmet göra följande saker:

1. Blanda kortleken.
2. Dela ut fem kort till varje spelare
3. Sortera varje hand för varje spelare
4. Skriva ut varje hand för alla spelare

Följande funktioner behövs:

```
initiera(struct kort kortlek[52]);
blanda(struct kort kortlek[52]);
delautkort(struct kort kortlek[52], int antal_spelare);
sortera(struct kort kortlek[52]);
skrivuthander(struct kort kortlek[52], int antal_spelare);
```

Den första funktionen ser till att alla färger och valörer kommer med, den andra blandar kortleken och den tredje delar ut kort till de ingående spelarna. Eftersom antalet spelare också skickas som parameter till denna funktion ska funktionen alltså slumpa in fem 1:or på fem olika platser i arrayen av kort och likadant för alla andra spelare. Den sista funktionen skriver ut alla spelares händer.

Sorteringsfunktionen som är den näst sista funktionen som sak anropas innan utskriften sker är lite mer komplicerad. Eftersom vi lagrar de olika händerna i kortleken, alltså varje kort håller själv reda på vilken hand den tillhör måste sorteringen för en viss hand ta hänsyn till att den handens kort finns utspridda på olika platser i kortleken... det tål att fundera på lite en del. Möjligtvis är denna uppgift lite artificiell. I ett reellt kortspelssammanhang där man vill ha en representationsform liknande denna inför man troligen extra pekararrayer för att bättre hantera korten i en hand.
III. Strukturering

11. *Format och strukturering:* Ett register innehåller uppgifter på följande form:

```
Person:Lön:Chef
```

där `Person` är ett namn bestående av alla möjliga bokstäver (A-Z, a-z), bindestreck och mellanslag. Inga andra bokstäver för förekomma i fältet `Person`. Fältet `Lön` är ett heltal i intervallet 10000 till 150000. Inga andra heltal får förekomma (observera också att ett heltal måste byggas upp av siffror, inga andra tecken). Det sista fältet, `Chef`, har samma format som det första fältet (`Person`).

Registret är givet i form av en textfil, en personuppgift lagras då på en textrad. Skriv en funktion som heter `format_ok()` som tar en sträng som inparameter som returnerar ett heltal som är 1 om textraden motsvarar en välformad personuppgift (alltså uppfyller kraven ovan) och 0 om något av kraven ovan inte är uppfyllda.

Skriv ett program som använder funktionen för att avgöra hur många personuppgifter i ett givet register som är välformade och hur många som inte är det. Programmet ska skriva de välformade personuppgifterna till en ny textfil och de som inte är välformade ska skriva till en annan textfil. Programmet ska även fråga efter filnamnen på dels textfilen som ska behandlas och den textfil som ska skapas.

12. *Sortering och strukturering.* Skriv en funktion som heter `convert()` som tar två filnamn som parametrar, det första ska vara ett filnamn till en textfil som innehåller ett register som är beskrivet i uppgiften ovan. Det andra ska vara ett nytt filnamn till en binärfil som funktionen ska skapa. Innehållet i binärfilen ska vara det som redan är innehållet i registret som fanns i textfilen vars filnamn gavs som första parametrar. Den resulterande binärfilen ska vara baserad på structen

```
struct uppgift
{
     char name[40];
     int salary;
     char boss[40];
};
```

Funktionen ska vidare sortera innehållet på fältet `name` i binärfilen.

13. Studera nedanstående program:

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

#define MAX 10

char namnlista[MAX][20];

byt(char s1[], char s2[])
{
  char tmp[20];
  strcpy(tmp,s1);
  strcpy(s1,s2);
  strcpy(s2,tmp);
}

main()
{
  int i, j;
  int antal=5;
  char newline;

  /*Inmatning:*/
  printf("Hur många namn vill du mata in? (max 10): ");
  scanf("%d", &antal); scanf("%c",&newline); //Istället för fflush(stdin);

  for(i=0;i<antal;i++)
  {
    printf("Namn %d: ", i+1);
    gets(namnlista[i]);
  }

  /*Sortering:*/
  for(i=0;i<antal-1;i++)
    for(j=i+1;j<antal;j++)
      if(strcmp(namnlista[i],namnlista[j])>0)
        byt(namnlista[i],namnlista[j]);

  /*Utskrift:*/
  printf("\nNamnen:\n");
  for(i=0;i<antal;i++)
    printf("Namn %d: %s\n", i+1, namnlista[i]);

}
```

Det tar in ett antal strängar och sorterar dem i bokstavsordning. Programmet har bara en extra funktion som heter `byt()` (som vi ser ovan) som byter innehållet i två strängar, den anropas under sorteringen. Skriv om programmet på följande sätt:

1. Deklarationen `char namnlista[MAX][20];` flyttas in i huvudprogrammet.

2. En funktion som heter "sortera" skapas med funktionsprototypen
   `sortera(char namnlista[][20], int antal);`
   som sorterar ett antal namn men som kan fungera för olika antal namn, man skickar in namnen i en array i parametern `namnlista` och hur många det är i parametern `antal`.

3. Funktioner ska även skrivas för inmatning och utskrift av namnen, de ska ha nästan samma funktionsprototyper som sortera, så här:
   `matain(char namnlista[][20], int * antal);`
   `skrivut(char namnlista[][20], int antal);`
   och de ska ska användas i programmet för att mata in och skriva ut namnen i `namnlista`. Som vi ser är antal deklarerad som referensparameter och det betyder att funktionen `matain` ska leverera värdet på variabeln antal efter att ha frågat användaren.

Funktionen `main()` i programmet ska, efter hela programmet skrivits om, ha *exakt* följande utseende:

```
main()
{
  char namnlista[MAX][10];
  int antal;

  matain(namnlista, &antal);
  sortera(namnlista, antal);
  skrivut(namnlista, antal);

}
```

14. Strukturering och sortering/klassificering

Studera nedanstående program:

```
#include <stdio.h>

main()
{
  double uppdelnings_grans, skuld, lanstorlek;
  char filnamn_lan[20], filnamn_lagalan[20], filnamn_hogalan[20],
    newline, rad[40], lantagare[40];
  FILE * lanfil;
  FILE * hogalanfil;
  FILE * lagalanfil;

  printf("Uppdelningsgrans: "); scanf("%lf", &uppdelnings_grans);
  scanf("%c", &newline);
  printf("Fil med laneuppgifter: "); scanf("%s", filnamn_lan);
  printf("Fil med laneuppgifter laga lan: "); scanf("%s", filnamn_lagalan);
  printf("Fil med laneuppgifter: "); scanf("%s", filnamn_hogalan);

  lanfil = fopen(filnamn_lan, "r");
  lagalanfil = fopen(filnamn_lagalan, "w");
  hogalanfil = fopen(filnamn_hogalan, "w");

  while(fscanf(lanfil,"%s %lf %lf",lantagare,&skuld,&lanstorlek)&&!feof(lanfil))
    if(lanstorlek<uppdelnings_grans)
      fprintf(lagalanfil, "%s %lf %lf\n", lantagare, skuld, lanstorlek);
    else
      fprintf(hogalanfil, "%s %lf %lf\n", lantagare, skuld, lanstorlek);

}
```

Programmet behandlar uppgifter om banklån, i en textfil (vars filnamn anges av användaren) finns uppgifterna lagrade på formatet "%s %lf %lf" alltså en följd av tecken (som ska vara ett namn) följt av två flyttal. Alla element är separerade av mellanslag (whitespace). Vi studerar ett exempel på en lånefil här:

```
kalle 45.0 900
lisa 35.9 1000
olle 1200.0 20000
lisa 5000 10000
anders 4000 15000
karl 34 100
```

Programmet frågar först efter en uppdelningsgräns (uppdelnings_grans) och därefter tre filnamn, det första filnamnet är den filen som uppgifter om lån ska läsas från, det andra filnamnet anger den fil där de låga lånen ska sparas och det tredje filnamnet anger den fil där de höga lånen ska sparas. Vad som är låga respektive höga lån avgörs av uppdelningsgränsen som användaren matar in.

Uppgift: Skriv om ovanstående program med följande modifikationer

a) En låneuppgift ska sparas i en struct med följande deklaration:

```
struct lan {
  char lantagare[20];
  double skuld;
  double storlek;
};
```

b) Programmet ska läsa in alla uppgifter i arrayer av dessa structar istället för att bara skriva dem direkt till två olika filer. Här är det lämpligt att införa två arrayer av structar en för låga lån och en för höga lån.

c) Innan de skrivs till filerna ska de dessutom sorteras på skuld så att minst skuld kommer först.

d) Programmet ska uppdelas i funktioner så att huvudprogrammet får följande ungefärliga utseende:

```
main() {
  double uppdelnings_grans, skuld, lanstorlek;
  char filnamn_lan[20], filnamn_lagalan[20], filnamn_hogalan[20];
  FILE * lanfil;
  FILE * lagalanfil;
  FILE * hogalanfil;

  struct lan lagalan[MAX]; int antal_lagalan;
  struct lan hogalan[MAX]; int antal_hogalan;

  mata_in_filnamn(filnamn_lan,filnamn_lagalan,filnamn_hogalan);

  lanfil = fopen(filnamn_lan, "r");
  las_in_lanuppgifter(lanfil,lagalan,&antal_lagalan,hogalan,&antal_hogalan);
  fclose(lanfil);
```

```
  sortera(lagalan,antal_lagalan);
  sortera(hogalan,antal_hogalan);

  lagalanfil = fopen(filnamn_lagalan, "w");
  skriv(lagalanfil,lagalan,antal_lagalan);
  fclose(lagalanfil);

  hogalanfil = fopen(filnamn_hogalan, "w");
  skriv(hogalanfil,hogalan,antal_hogalan);
  fclose(hogalanfil);

}
```

Observera här att funktionerna `sortera()` och `skriv()` ska vara allmänna funktioner som ska fungerar likadant oavsett om man arbetar med låga lån eller höga lån.

15. Omvandlingar mellan årsdag och datum. Gör följande globala deklarationer i ett program:

```
const int antal_dagar_i_manad[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
const char *manad[12]={"januari","februari","mars","april","maj","juni","juli",
                       "augusti","september","oktober","november","december"};

struct dag_i_manad {
  int datum;
  int manad;
};
```

Detta är två tabeller över hur många dagar det finns i en månad samt vad månaderna heter. Vi hanterar också en angivelse av en dag genom att datum och månad anges, det uttrycks i structen som också finns ovan.

Skriv funktioner för att omvandla mellan olika sätt att representera en dag. En dag representeras här som ett tal mellan 0 och 364 (det finns 365 dagar på ett år, vi numrerar dem på C-vid och börjar på 0.) En dag kan också representeras genom vilken månad den finns i tillsammans med sitt datum, till exempel den 5 mars blir då dag 4 i månad 2. (Vi börjar som sagt på 0 både när vi numrerar dagar och månader.) Följande huvudprogram får inte ändras, det ska fungera tillsammans med dina två funktioner som du ska skriva:

```
main()
{
  int dag;
  struct dag_i_manad md;

  printf("Skriv in dagnummer (0-364): "); scanf("%d", &dag);
  arsdag_till_manadsdag (dag, &md);
  printf("Datum (borjar på 0): %d %s.\n", md.datum, manad[md.manad]);

  printf("Skriv in manad (0-11):"); scanf("%d", &(md.manad));
  printf("Skriv datum (0-30):"); scanf("%d", &(md.datum));
  manadsdag_till_arsdag(&md,&dag);
  printf("Dagnummer: %d.\n", dag);

}
```

En provkörning:

```
Skriv in dagnummer (0-364): 154
Datum (borjar på 0): 3 juni.
Skriv in manad (0-11):5
Skriv datum (0-30):3
Dagnummer: 154.
```