Laboration 3 – Sista laborationen – Register, igen, fast uppsnyggat
Denna uppgift benämns A6.

Vi ska nu ta det resulterande programmet från den stora övningen inför denna laboration och snygga till det så att följande krav är uppfyllda:

* Inga globala variabler.

Det enda som får vara globalt är `#define`-direktiv och det blir bara konstanter och *include*-filer samt en deklaration på den struct som ska användas. Variabler kommer att behövas, men dessa ska då **alltid** deklareras som lokala variabler inuti funktioner, många variabler blir då deklarerade i `main()`, men inte alla. Funktionen `main()` kommer givetvis att behöva anropa andra funktioner för att ändra på innehållet i variablerna men då måste de berörda variablerna skickas med som parametrar. Till exempel, om vi ska läsa in innehållet från en fil till variabeln `register` kan `main()` behöva göra anropet

```
las_in_register(filnamn, register, &antal_personuppgifter);
```

Alltså ”*Läs in register*”. Här antar vi att filnamnet (`filnamn`) där personuppgifterna är lagrade är bestämt innan anropet till `las_in_register()`. Vi ser också att addressen till variabeln `antal_personuppgifter` skickas med vilket vi får tolka som att funktionen `las_in_register()` förändrar dess värde, då `las_in_register()` kört klart innehåller variabeln `antal_personuppgifter` förstås just det antal personuppgifter som registret innehåller. (Och det beror ju i sin tur på hur många personuppgifter som filen innehåller

* Array av structar, inte tre separata strängarrayer.

I den stora övningen till denna laboration hanterades också ett personuppgiftsregister. Då behandlades data i form av tre *separata* arrayer av strängar:

```
char name[MAX][20];
char email[MAX][30];
char phone[MAX][20];
```

Vi ska inte ha det så nu. Vi ska istället arbeta med structar och samla alla personuppgifter hörande till en person *i en enhet*, structen, och sedan hantera hela registret som en array av structar av denna typ. Deklarationerna

```
struct personuppgift {
  char name[20];
  char email[30];
  char phone[20];
};
```

och

```
struct personuppgift register[MAX];
```

kommer då att vara grunden för detta program. Se boken för hur man hanterar ett sådant register.

* Binärfiler istället för textfiler.

I den stora övningen inför denna laboration arbetade vi med textfiler. Vi ska nu gå över till binärfiler istället. Boken visar tydligt hur man anänder ett sådant register, men i bokens exempel sker ändringar i filen direkt i respons till användarens kommandon. Det är lättare att göra alla ändringar i arrayen i datorns primärminne under programkörningen och sedan, just innan programmet avslutas skriver man alla ändringar till filen.

* Programmet ska vara organiserat på ett bra sätt. Det betyder 3 saker

1. Perfekt indentering genom hela programmet. Järndisciplin på det!

2. Variabelnamn och funktionsnamn ska vara *väl valda*, variabelnamn ska *beskriva det som de innehåller* och funktionsnamn ska *beskriva det som de utför*. Ett exempel kan vara funktionen `sortera()`. Om registret är en array av structar med namnet `register` kan ett bra anrop till funktionen `sortera()` se ut så här:

   ```
   sortera(register, antal_personuppgifter, NAMN);
   ```

   Vi ser här flera saker: variabeln `antal_personuppgifter` innehåller (förstås) det antal personuppgifter som registret för närvarande innehåller. När man lägger till en personuppgift ökas den med 1 och när man tar bort en personuppgift minskas den med 1. Den ska alltid förstås befinna sig under eller lika med `MAX` som är storleken av arrayen `register` och därmed det största antalet personuppgifter som programmet kan hantera. Funktionen själv heter `sortera` och det beskriver då vad den utför för uppgift: den sorterar registret. Konstanten `NAMN` är en vanlig heltalskonstant troligtvis definierad via ett makro av typ `#define NAMN 0`, och det används som tredje parameter till `sortera()` för att specificera att det är namnen som ska avgöra sorteringsordningen. Två andra konstanter kan tänkas vara användbara, `EPOST`, som kanske har värdet 1 som avgör att sortering ska ske på epostadresser och `TELEFON` som kanske har värdet 2 som avgör att sortering ska ske på telefonnummer. Man kan med dessa konstanter göra anropen

   ```
   sortera(register, antal_personuppgifter, EPOST);
   ```

   respektive

   ```
   sortera(register, antal_personuppgifter, TELEFON);
   ```

   som förklarar sig själva. Om ni tycker att detta verkar svår är det helt OK att skriva tre separata funktioner som anropas så här:

   ```
   sortera_efter_namn (register, antal_personuppgifter);
   sortera_efter_epost(register, antal_personuppgifter);
   sortera_efter_telefon(register, antal_personuppgifter);
   ```

   Dessa funktioner förklarar också sig själva.

3. Kommentarer ska finnas löpande genom hela programmet och samverka med indentering och väl valda namn på variabler och funktioner så att programmet blir väldigt läsligt för en som inte sett det innan. Det är svårt att precisera detta, men ni får experiementera med detta för att nå goda resultat. Sträva efter att koden ska *förklara sig själv* (som ovan).

4. Sista överkursuppgift: fundera på roller som olika funktioner i programmet har. Kan du separera rollerna och göra kvalitativa bedömningar om vad som är lämpligt ska finnas i olika funktioner? Exempel: Det kan vara lämpligt att låta huvudprogrammet sköta alla kommunikation med användaren och låta alla beräkningar och sådant ske i funktioner. Hur är detta uttryckt i ditt program? Det här är en mera reflekterande uppgift så det är inte lätta formulera den på ett strikt sätt, det är inte heller ett krav att man löst den.