6 Lagring av data på fil

(Senaste ändring av detta kapitel: 16 december 2006)

Vid inmatning av ett fåtal data till ett program kan man använda tangentbordet. Har man större datamängder som ska matas in brukar man istället ha dessa på en *fil i sekundärminnet* och låta programmet hämta data direkt från filen. Samma sak gäller även vid utmatning. Normalt sker all utmatning till skärmen. Vid stora utdatamängder eller då utdata ska sparas för att användas vid ett senare tillfälle måste man från sitt program kunna skriva ut data på en *fil i sekundärminnet*.

Ett typiskt exempel på ovanstående är ett program som ska ta reda på alla bilar som man ska skicka ut besked om kontrollbesiktning till. Bilarna finns i ett register på en fil i sekundärminnet. Programmet måste kunna läsa data från denna fil, kontrollera om bilen ska besiktigas, och skriva ut information om bilen på en annan fil eller alternativt mata ut brev som kallar till kontrollbesiktning.

För att från ett program kunna hämta data från eller skicka data till filer i sekundärminnet finns i alla generella programmeringsspråk möjligheter att skapa kommunikationskanaler eller buffertar som sköter om transporten av data mellan primärminnet och sekundärminnet. *I C kallas dessa kanaler för strömmar*. (Ej att förväxla med de ”strömmar”, eller på engelska ”streams”, som finns i C++. De används för samma sak, men fungerar annorlunda.)

6.1 Strömmar och filer

Med en *ström* i C ska man kunna transportera data mellan sitt program och en *fil* i sekundärminnet. I header-filen stdio.h finns en fördeklarerad strömtyp FILE för detta.

Ex: Definiera en inström för att läsa data från filen indata.txt och en utström för att skriva data på filen utdata.txt.

```
#include <stdio.h>

/* ... */

/* definiera pekare till strömmarna */
FILE *instrom, *utstrom;

/* skapa och öppna strömmarna */
instrom = fopen("indata.txt", "r");   /* r som i read */
utstrom = fopen("utdata.txt", "w");   /* w som i write */

/* läs data från filen indata.txt via instrom */

/* skriv data till filen utdata.txt via utstrom */

/* töm och stäng strömmarna */
fclose(instrom);
fclose(utstrom);
```

I sitt program skapar man en ström genom att först definiera en pekare till datatypen FILE, som finns i stdio.h, och sedan öppna strömmen eller kanalen till filen med fopen. I fopen anger man med den *första strängparametern* vilken fil i sekundärminnet som strömmen ska öppnas till. Om det av någon anledning ej gick att öppna strömmen, *returnerar fopen pekarvärdet NULL*. Detta kan exempelvis inträffa ovan, om filen indata.txt inte finns i sekundärminnet eller om den är lässkyddad.

Med den *andra strängparametern* i anropet av fopen anger man vilken *typ av ström* som ska skapas. Man kan öppna *textströmmar* eller *binärströmmar* och man kan ange om man ska läsa eller skriva data via dessa strömmar Den andra parametern kan ha värden enligt:

r	----	Textström för enbart läsning. Filen måste finnas.
w	----	Textström för enbart skrivning. Om filen finns skrivs den över. Om filen ej finns skapas en ny fil.
a	----	Textström för enbart skrivning. Om filen finns lägger man till data på slutet. Om filen ej finns skapas en ny fil.
r+	----	Textström för både läsning och skrivning. Filen måste finnas.
w+	----	Textström för både skrivning och läsning. Om filen finns skrivs den över. Om filen ej finns skapas en ny fil.
a+	----	Samma som w+ men man lägger till i slutet om filen finns.

I C skiljer man på *textströmmar* och *binärströmmar.* En textström omvandlar vid skrivning data, som finns i binär form i primärminnet, till läsbara tecken exempelvis i ASCII-format på filen i sekundärminnet och omvänt från ASCII-form[1] i sekundärminnet till binär form i primärminnet vid läsning. Filerna i sekundärminnet som på detta sätt blir ASCII-kodade kallas för *textfiler* och kan exempelvis ändras med en editor och skrivas ut på en printer. Textfiler behövs för att kommunikationen mellan *människa och dator* ska fungera.

Binärströmmar gör ingen omvandling utan de transporterar bara data i binärform mellan programmet i primärminnet och filen i sekundärminnet och tvärtom. Alla strömmar i C blir som *default textströmmar*. Vill man ha en binärström måste man ange detta explicit i strängen för strömtypen med ett *b* enligt exempelvis:

r+b	----	Binärström för både läsning och skrivning. Filen måste finnas.
wb	----	Binärström för skrivning. Om filen finns skrivs den över. Om filen ej finns skapas en ny fil.

1 Eller vilken teckenkodning man nu använder. Man säger ofta ”ASCII-koder”, men egentligen är det bara den ursprungliga amerikanska 7-bitarskoden som heter ASCII.

6.2 Textströmmar

När ett program ska kommunicera med en människa måste in- och utdata vara i sådan form att människan förstår. Programmet använder internt ett speciellt binärt format för exempelvis det reella talet 3.4. Ska ett program skriva ut ett reellt tal på en fil i sekundärminnet så att en människa kan förstå vad filen innehåller, måste det binära formatet omvandlas till tre ASCII-kodade tecken i form av 3.4.

Ex: Skriv ett program som slumpar 1000 tärningskast som skrivs ut på en textfil utfall.txt med 25 utfall per rad.

```
#include <stdio.h>    /* FILE, fopen, fprintf, fclose */
#include <stdlib.h>
#include <time.h>

int main()
{
   FILE *filout;
   int i, utfall;

   filout = fopen("utfall.txt", "w");

   srand((unsigned)time(NULL));
   for (i = 1; i <= 1000; i++)
   {
      utfall = rand()%6 + 1;
      fprintf(filout, "%2d", utfall);
      if (i % 25 == 0)
         fprintf(filout, "\n");
   }

   fclose(filout);

   return 0;
}
```

OBS! Funktionen *fprintf* skriver ut på den ström som anges som första parameter. Formatsträngens omvandlingsspecifikation anger hur det slumpade tärningsutfallets ettor och nollor ska tolkas och omvandlas innan det skrivs ut som ASCII-kodade tecken på textfilen utfall.txt via strömmen filout. Eftersom man använder en textström, och skriver på den med fprintf, blir utfall.txt en textfil, som kan *skrivas ut på en skrivare eller visas på en skärm. Textfiler brukar markeras med filtypen .txt.*

OBS! Funktionen fprintf fungerar på samma sätt som den vanliga utskriftsfunktionen printf. Skillnaden är att med *fprintf kan man själv välja vilken ström man ska skriva ut på*. Funktionen printf däremot skriver alltid ut på samma fördefinierade ström, *stdout*, vilken som default är ihopkopplad med skärmen.

Ex: Skriv ett program som läser de slumpade tärningsutfallen från textfilen utfall.txt och på skärmen skriver ut utfallens medelvärde.

```
#include <stdio.h>    /* FILE, fopen, fscanf, fclose */

int main()
{
   FILE *filin;
   int i, utfall, sum = 0;

   filin = fopen("utfall.txt", "r");
   if (filin != NULL)
   {
      for (i = 0; i < 1000; i++)
      {
         fscanf(filin, "%d", &utfall);
         sum += utfall;
      }

      fclose(filin);

      printf("Medelvärde: %.2f\n", (double)sum/1000);

   }
   else
      printf("Det gick inte att läsa filen utfall.txt!\n");

   return 0;
}
```

OBS! Funktionen *fscanf* läser från den ström som anges som första parameter. Formatsträngens omvandlingsspecifikation anger hur ASCII-koderna i textfilen utfall.txt ska tolkas och hur omvandlingen till binärform ska ske innan de instoppas på adressen för variabeln utfall.

OBS! Filen läses i *fritt format* med utnyttjande av separatortecknen blank, tab och nyrad, precis som vid läsning med scanf från tangentbordet. Skillnaden mellan funktionen fscanf och scanf är att fscanf kan utnyttja en egendefinierad ström kopplad till vilken textfil som helst, medan *scanf* alltid läser från den fördefinierade strömmen *stdin*, vilken som default är ihopkopplad med tangentbordet.

I ovanstående exempel visste man hur många tal som fanns i filen och man kunde anpassa läsningen efter detta. *Vad ska man göra om man inte vet hur många tal som finns i en fil,* som man ska läsa data från? Det finns olika metoder. När man läser från tangentbordet kan man exempelvis avsluta inläsningen med talet 0. Detta kan man naturligtvis också göra vid läsning från fil om man ser till att ha 0 som sista tal i filen.

Ex. Skriv ett program som läser alla reella tal ifrån textfilen retal.txt och skriver ut talens medelvärde. Tittar man på filen retal.txt på skärmen ser den ut som:

3.45 3.67 3.56
3.34 3.48
3.56 3.61 3.24 3.56
0.0

```
#include <stdio.h>

int main()
{
   FILE *filin;
   double x, sum = 0;
   int nr = 0;

   filin = fopen("retal.txt", "r");
   if (filin != NULL)
   {

      fscanf(filin, "%lf", &x);
      while(x != 0.0)
      {
         sum += x;
         nr++;
         fscanf(filin, "%lf", &x);
      }

      fclose(filin);
      if (nr > 0)
         printf("Medelvärde: %.2f\n", sum/nr);
      else
         printf("Antalet inlästa tal är 0!\n");
   }
   else
      printf("Det gick inte att läsa filen retal.txt!\n");

   return 0;
}
```

OBS! Funktionen fscanf hoppar först över alla separatorer i form av blanka, tabbar och nyradtecken och läser därefter tecken *så länge det passar ihop* med ett reellt tal.

OBS! Textfilen retal.txt kan vara gjord av ett annat program eller med ett editorprogram.

Man behöver ej själv markera filslut, som man gjorde ovan med 0.0. I ett C-program kommer *fscanf-funktionen, som normalt returnerar antalet lyckade läsningar, att returnera värdet -1* då den läser förbi slutet på filen. (Operativsystemet håller reda på hur lång varje fil är.) Konstanten EOF med värdet -1 finns som makro i stdio.h.

Notera att det är först *den första misslyckade läsningen* som returnerar EOF. Vid inläsningen av det sista talet i filen returnerar fscanf 1.

Ex: Skriv ett program som letar upp och skriver ut det största och minsta talet i textfilen heltal.txt. I filen finns alltid minst ett tal enligt:

234 123 345 236 678 456
567 891 234 567
125 567 234 456 234
234 567

```
#include <stdio.h>

int main()
{
   FILE *filin;
   int tal, min, max, status;

   filin = fopen("heltal.txt", "r");
   if (filin != NULL)
   {
      fscanf(filin, "%d", &tal);
      min = max = tal;

      status = fscanf(filin, "%d", &tal);
      while(status != EOF)
      {
         if ( tal < min )
            min = tal;
         else if (tal > max)
            max = tal;
         status = fscanf(filin, "%d", &tal);
      }

      fclose(filin);

      printf("Minsta tal: %d\n", min);
      printf("Största tal: %d\n", max);
   }
   else
      printf("Det gick inte att läsa filen heltal.txt!\n");

   return 0;
}
```

OBS! Funktionen fscanf returnerar ett heltal som är lika med antalet lyckligt inlästa tal eller EOF (-1), då filslut nåtts. Detta gäller även scanf-funktionen som läser från tangentbordet. Hur man gör för att från tangentbordet tala om att ”filen”, dvs inmatningen, är slut varierar mellan olika operativsystem. I Windows trycker man CTRL-Z och i Unix (och Linux) CTRL-D.

OBS! Ovanstående lösning, med två olika fscanf-anrop och en variabel (status) för att spara returvärdet från fscanf-anropet, är inte det vanliga sättet att skriva läs-loopar i C. Det vanliga, och idiomatiska (dvs, så som man alltid brukar göra) är i stället det här, med ett enda fscanf-anrop i koden:

```
while(fscanf(filin, "%d", &tal) != EOF)
{
   if ( tal < min )
      min = tal;
   else if (tal > max)
      max = tal;
}
```

I fil-inmatningsfunktionen fscanf och fil-utmatningsfunktionen fprintf kan man, på samma sätt som i vanliga scanf och printf, använda andra omvandlingsspecifikationer, som exempelvis *%c för att läsa/skriva tecken* och *%s för att läsa/skriva strängar*. Dessa har dock som framgått tidigare vissa nackdelar åtminstone för strängar. Därför finns det speciella funktioner för in- och utmatning av tecken och strängar.

För tecken heter funktionerna *fgetc och fputc*. Dessa ska motsvara getchar och putchar som enbart gäller för stdin (tangentbord) resp stdout (skärm).

Ex: Skriv ett program som kopierar en textfil tecken för tecken till en annan fil. Namnet på filerna ska läsas in från tangentbordet.

```
#include <stdio.h>

int main()
{
   FILE *fin, *fout;
   char infilnamn[30 + 1], utfilnamn[30 + 1];
   int ch;

   printf("Ge infilnamn: ");
   gets(infilnamn);[2]
   fin = fopen(infilnamn, "r");

   /* här borde kontrolleras att filen gick att öppna */

   printf("Ge utfilnamn: ");
   gets(utfilnamn);
   fout = fopen(utfilnamn, "w");

   ch = fgetc(fin);
   while ( ch != EOF )
   {
      fputc(ch, fout);
      ch = fgetc(fin);
   }

   fclose(fin);
   fclose(fout);

   return 0;
}
```

OBS! Förutom tecken kan `fgetc` även returnera värdet `EOF`, som inte är ett tecken. Därför måste variabeln `ch` ovan vara en `int`, och inte en `char`.

OBS! Även nyradtecken läses från infilen och kopieras till utfilen vilket innebär att raderna bibehålls.

OBS! Ovanstående lösning, med två olika fgetc-anrop, är inte det vanliga sättet att skriva läs-loopar med fgetc i C. Det vanliga, och idiomatiska (dvs, så som man alltid brukar göra) är i stället det här, med ett enda fgetc-anrop. Det fungerar eftersom ett tilldelningsuttryck får samma värde som det tilldelade värdet.

```
while ( (ch = fgetc(fin)) != EOF )
{
    fputc(ch, fout);
}
```

2 Använd inte gets i riktiga program. Den kontrollerar inte att den inlästa strängen får plats i variabeln.

Ex: Skriv ett program som läser textfilen matdata.txt och beräknar och på skärmen skriver ut summan av varje rad och en totalsumma. Filen ser ut som:

23.45 34.56 12.34
56.78 23.56
12.45 45.67 13.56 25.67

```
#include <stdio.h>

int main()
{
   FILE *rfil;
   double x, radsum = 0.0, totalsum = 0.0;
   char rfilnamn[20 + 1];
   int ch;

   rfil = fopen("matdata.txt", "r");
   while (rfil == NULL)
   {
      printf("Gick inte att öppna filen!\n");
      printf("Ge nytt filnamn: ");
      gets(rfilnamn);
      rfil = fopen(rfilnamn, "r");
   }

   /*  läs tal från fil och summera */
   while ( fscanf(rfil, "%lf", &x) != EOF )
   {
      radsum += x;

      /* kontrollera nästa tecken */
      ch = fgetc(rfil);
      ungetc(ch, rfil);

      if ( ch == '\n' || ch == EOF )
      {
         /* ny rad */
         printf("%f\n", radsum);
         totalsum += radsum;
         radsum = 0;
       }
   }
   fclose(rfil);
   printf("%f\n", totalsum);

   return 0;
}
```

OBS! Med fgetc läses nästa tecken i filen och med ungetc flyttas filpekaren tillbaka så att *aktuell filposition ej ändras.* Man kan fortsätta att läsa nästa tal som vanligt. I ovanstående exempel är användandet av ungetc egentligen onödigt eftersom fgetc bara läser blanktecken, radslutstecken eller filslutstecken, som ej har någon inverkan på de reella talens storlek.

OBS! Programmet är känsligt för formatet på indata. Det räcker med att man råkar få med ett blanktecken på slutet av en rad i filen matdata.txt, vilket i de flesta texteditorer inte syns, så kommer programmet att räkna fel. Så bör ett program förstås inte fungera, men här har vi förenklat för att principerna med filläsning ska framgå tydligare.

För strängar finns funktionerna *fgets och fputs* som läser resp skriver strängar på valfria textströmmar. Dessa motsvarar funktionerna gets och puts som gör motsvarande på stdin (tangentbord) resp stdout (skärm).

Ex: Skriv ett program som läser textfilen medlem.txt som innehåller namn, telefonnummer och medlemsavgift enligt:

Ulla Karlsson
019/111111
300
Per Jonsson
0586/222222
0
....

och som för alla medlemmar som har medlemsavgiften 0 skriver ut namn och telefonnummer på en ny fil ejbetalt.txt.

```
#include <stdio.h>

int main()
{
   char namn[31], tel[21];
   int avgift;
   FILE *infil, *utfil;

   infil = fopen("medlem.txt", "r");
   utfil = fopen("ejbetalt.txt", "w");
   while ( fgets(namn, 31, infil) != NULL)
   {
      fgets(tel, 21, infil);
      fscanf(infil, "%d", &avgift);
      fgetc(infil);        /* förbi radslut */
      if ( avgift == 0 )
      {
         /* ej betalt */
         fputs(namn, utfil);
         fputs(tel, utfil);
      }
   }
   fclose(infil);
   fclose(utfil);

   return 0;
}
```

OBS! Funktionen fgets har tre parametrar där den i mitten anger det maximala antal tecken som ryms i strängen, som man ska läsa till. Ovan *läser funktionen maximalt 30 tecken till strängen namn*, eftersom det ska finnas plats för ett avslutande '\0'-tecken. Läsningen avslutas dock alltid då *ett radslutstecken* lästs. *Radslutstecknet ingår i den inlästa strängen* till skillnad från gets-funktionen.

OBS! I stället för att skriva ut storleken som ett tal, kan man använda sizeof-operatorn: `fgets(namn, sizeof namn, infil)`

OBS! Funktionen fgets *returnerar NULL* då det inte finns någon sträng att läsa, exempelvis vid filslut.

När man definierar egna strömmar kan man koppla *dessa från sitt program till vilken fil som helst i sekundärminnet*. Man kan också koppla strömmarna till andra *externa enheter* som printrar, skärm, tangentbord, portar etc. I C-program kommer man åt dessa enheter genom att använda de fördefinierade filnamn, som dessa har i det aktuella operativsystemet. I DOS har exempelvis printern i den första parallellporten filnamnet lpt1, skärm och tangentbord filnamnet con etc. I Unix kan man oftast komma åt skärmen och tangentbordet som filen /dev/tty.

Ex: Skriv om ovanstående program med medlemmarna så att utskriften kan väljas att fås på skärm eller printer. Anger man inget kommer utskriften som vanligt på filen ejbetalt.txt.

```
#include <stdio.h>

int main()
{
   FILE *infil, *utfil;
   char namn[30 + 1], tel[30 + 1], svar;
   int avgift;

   infil = fopen("medlem.txt", "r");

   /* fråga efter utskriften */
   printf("Utskrift på printer/skärm (p/s) : ");
   svar = getchar();
   if ( svar == 'P' || svar == 'p' )
      utfil = fopen("lpt1", "w");
   else if ( svar == 'S' || svar == 's' )
   {
      utfil = fopen("con", "w");
   }
   else
      utfil = fopen("ejbetalt.txt", "w");

   while ( fgets(namn, sizeof namn, infil) != NULL )
   {
      fgets(tel, sizeof tel, infil);
      fscanf(infil, "%d", &avgift);
      while (fgetc(infil) != '\n' && !feof(infil))
         ;
      if ( avgift == 0 )
      {
         /* ej betalt */
         fputs(namn, utfil);
         fputs(tel, utfil);
      }
   }
   fclose(infil);
   fclose(utfil);

   return 0;
}
```

6.3 Binärströmmar

Binärströmmar används då man från sitt program vill spara undan data på en binärfil i sekundärminnet och det *inte finns något behov av att skriva ut denna data på skärm eller skrivare*. På detta sätt sparar man både programtid och (ofta) minne. Exempelvis tar talet 23456 upp 5 byte (5 tecken) i en textfil och kanske 2 byte (datatypen short int) i en binärfil. Då man använder en binärfil behövs inte heller någon omvandling från binär form i primärminnet till ASCII-kod i sekundärminnet. Strömmen skyfflar bara över byten precis som de ser ut i primärminnet.

Det finns också nackdelar med binärfiler. Exempelvis kan binärformatet se olika ut på olika typer av datorer, så binärfiler kan inte flyttas mellan datorer hur som helst.

Ex: Skriv ett program som läser in mätdata i form av reella tal från tangentbordet och skriver in dessa i en binärfil, mat.dat. Inläsningen av data avslutas med ett CTRL-Z (eller motsvarande) från tangentbordet, vilket innebär EOF i stdin.

```
#include <stdio.h>

int main()
{
   FILE *bfout;
   double x;

   bfout = fopen("mat.dat", "wb");

   /* läs data från tangentbordet */
   printf("Ge mätdata. Avsluta med EOF (CTRL-Z)!\n");
   while ( scanf("%lf", &x) != EOF )
   {
      /* skriv till fil */
       fwrite(&x, sizeof(double), 1, bfout);
   }

   fclose(bfout);

   return 0;
}
```

OBS! Filen öppnas med filtypen *wb*. Detta innebär att man öppnar en binärfil för enbart skrivning.

OBS! Funktionen *fwrite* skriver från den adress i primärminnet som anges av den första parametern de antal byte som anges av produkten av den andra och den tredje parametern till den ström som anges av den fjärde parametern.

OBS! Filen mat.*dat* är en binärfil som ej kan skrivas ut på printer eller skärm utan den innehåller data i samma binära form som variablerna i primärminnet. Med filtypen *.dat* markeras i forsättningen att det är en binärfil till motsats mot *.txt* för textfiler.

OBS! Operatorn *sizeof* som ger storleken i byte av en datatyp eller variabel. (Egentligen mäts storleken i char och inte i byte, men det är för det mesta samma sak.)

Binära filer innehåller binärkodad information. Detta innebär att om man ska kunna tolka data i en binärfil måste man skriva ett program som läser filen.

Ex: Skriv ett program som läser binärfilen mat.dat och beräknar och skriver ut medelvärdet av all data. Filnamnet ska inläsas.

```
#include <stdio.h>

int main()
{
   FILE *bfin;
   char filnamn[20 + 1];
   double x, sum = 0.0;
   int nr = 0;

   /* läs in filnamn och öppna fil */
   do
   {
      printf("Ge filnamn: ");
      gets(filnamn);
      bfin = fopen(filnamn, "rb");
      if (bfin == NULL)
         printf("Kunde inte öppna filen!\n");
   }
   while (bfin == NULL);

   /* läs från fil */
   fread(&x, sizeof(double), 1, bfin);
   while ( !feof(bfin) )
   {
      sum += x;
      nr++;
      fread(&x, sizeof(double), 1, bfin);
   }

   fclose(bfin);

   /* skriv ut medelvärde */
   if ( nr > 0 )
      printf("Medel: %f\n", sum / nr);
   else
      printf("Antalet data är 0!\n");

   return 0;
}
```

OBS! Funktionen *fread* läser till adressen som anges av den första parametern, så många byte data som anges av produkten av den andra och den tredje parametern från strömmen som anges av den sista parametern.

OBS! Funktionen *feof* returnerar sant då filslut nåtts i en *binärfil*. För textfiler är det säkrare att testa om inläsningsfunktionens returvärde är EOF.

OBS! Det är först *efter den första misslyckade läsningen* som feof returnerar sant. Om man anropar feof direkt efter att man läst den sista posten, returnerar den falskt, trots att det inte finns mer data att läsa i filen.

OBS! Ovanstående lösning, med feof och två olika fread-anrop, är inte den idiomatiska lösningen i C. Se nästa exempel för hur man brukar göra!

Funktionerna fwrite och fread *skyfflar byte* mellan primärminne och sekundärminne och tvärtom. Dessa funktioner är ej begränsade till att endast användas för binära strömmar utan kan också användas även för textfiler om inga konverteringar behöver göras. Man kan exempelvis utnyttja dessa funktioner till att kopiera alla typer av filer, både text- och binärfiler.

Ex: Skriv ett program som kopierar en fil till en annan. Filnamnen ska läsas in och test ska finnas så att infilen finns.

```
#include <stdio.h>
#include <stdlib.h>        /* exit */

int main()
{
   FILE *bfin, *bfout;
   char infilnamn[30 + 1], utfilnamn[30 + 1], ch;

   printf("Ge infilnamn: ");
   gets(infilnamn);
   bfin = fopen(infilnamn, "rb");
   if ( bfin == NULL)
   {
      /* om filen ej finns */
       printf("Kunde inte öppna filen!\n");
       exit(EXIT_FAILURE);
   }

   printf("Ge utfilnamn: ");
   gets(utfilnamn);
   bfout = fopen(utfilnamn, "wb");

   while (fread(&ch, sizeof(char), 1, bfin) == 1)
      fwrite(&ch, sizeof(char), 1, bfout);

   fclose(bfin);
   fclose(bfout);

   return EXIT_SUCCESS;
}
```

OBS! Hur man, via ett tecken i primärminnet, skyfflar över alla byte, precis som de ser ut, från infilen till utfilen. Detta program fungerar för alla typer av filer.

OBS! Använder man textströmmar och funktioner för dessa vid kopiering fungerar det ej för binärfiler i alla operativsystem. I exempelvis DOS så omvandlar skriv- och läsfunktionerna för textströmmar nyradtecken i primärminnet till två tecken (CR och LF) i sekundärminnnet och tvärtom.

Databaser eller register använder sig ofta av binärfiler. Oftast är det poster, som man arbetar med. En databashanterare för ett bilregister ska exempelvis kunna hantera bilposter som skyfflas mellan primärminne och sekundärminne vid sökning, sortering etc.

Ex: Skriv ett program som läser in bilposter med registreringsnummer och ägare och skriver in dessa i en binärfil som avslutningsvis skrivs ut på skärmen.

```
#include <stdio.h>

int main()
{
   struct bilpost
   {
      char regnr[10 + 1];
      char agare[30 + 1];
   } bil;

   FILE *bilfil;

   bilfil = fopen("bilar.dat", "w+b");

   /* läs in bilar och skriv på fil */
   printf("Ge regnr (Avsluta med bara RETURN) : ");
   gets(bil.regnr);
   while (bil.regnr[0] != '\0')
   {
      printf("Ge ägare: ");
      gets(bil.agare);
      fwrite(&bil, sizeof(struct bilpost), 1, bilfil);
      printf("Ge regnr (Avsluta med bara RETURN) : ");
      gets(bil.regnr);
   }

   /* läs fil och skriv ut bilarna */
   rewind(bilfil);
   while (fread(&bil, sizeof(struct bilpost), 1, bilfil) == 1)
   {
      printf("\n”);
      printf("Regnummer: %s\n", bil.regnr);
      printf("Ägare: %s\n", bil.agare);
   }
   fclose(bilfil);

   return 0;
}
```

OBS! Varje skrivning i filen fyller på med de antal byte som sizeof-operatorn ger. När man fyllt på med sista posten står man på filslut. Ska man börja att läsa filen från början måste man med *rewind*-funktionen ställa tillbaka den aktuella filpositionen till filens början.

OBS! När man öppnar en ström med w innebär det att man *suddar ut* eventuell gammal fil, om sådan finns. *Vill man ha kvar informationen i gamla filer och istället fylla på i slutet av filen*, ska man istället öppna strömmen med a, där a står för append (lägga till).

6.4 Filåtkomst eller filaccess

Normalt använder man *sekventiell åtkomst* (access) till filerna. Med sekventiell åtkomst menas att man manipulerar byten på filen i den ordning de står. Ska man läsa filen läser man från filens början till filens slut. Skriver man på filen fyller man på byte på byte. Denna rent sekventiella åtkomst till byten i filen kan man bryta med exempelvis rewind och med append enligt ovanstående exempel.

Det finns även andra möjligheter att *komma åt speciella byte i filen direkt*, så kallad *direkt åtkomst* (access). Man kan tänka sig att man har en filpekare som man flyttar på och placerar på önskad byte. Detta har man exempelvis nytta av då man ska uppdatera poster på en fil. Man läser information från filen, ändrar det och skriver tillbaka informationen i rätt position på filen. Har man ej möjligheter till direkt åtkomst får man kopiera över allt till en slaskfil samtidigt som man utför ändringar på lämpliga ställen och sedan kopiera tillbaka till den ursprungliga filen.

Ex: Skriv en enkel databashanterare för ett lager som innehåller ett antal varor med varubeteckning, pris och antal. Hanteraren ska kunna skapa ny vara, visa vara med viss beteckning och ändra priset på en vara.

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

struct varupost
{
   char vb[10 + 1];
   double pris;
   int antal;
};

void ny_vara(void);
void visa_vara(void);
void nytt_pris(void);
void skipline(void) { while (getchar() != '\n') ; }

int main()
{
   char svar;

   do
   {
      printf("N    Ny vara\n");
      printf("V    Visa vara\n");
      printf("P    Prisändring\n");
      printf("S    Sluta\n");
      printf("Välj >> ");
      svar = getchar();
      skipline();
```

```
      switch (svar)
      {
         case 'N':
            ny_vara();
            break;
         case 'V':
            visa_vara();
            getchar();
            skipline();
            break;
         case 'P':
            nytt_pris();
            break;
         case 'S':;
      }
   }
   while (svar != 'S');

   return 0;
}
```

OBS! Funktionsdeklarationer (prototyper, huvuden) före huvudprogrammet för att filen ska kunna kompileras med funktionsdefinitioner (kroppar) efter huvudprogrammet.

```
void ny_vara(void)
{
   FILE *lfil;
   struct varupost vara;

   /* läs in vara */
   printf("\nGe beteckning: ");
   gets(vara.vb);
   printf("Ge pris: ");
   scanf("%lf", &vara.pris);
   printf("Ge antal: ");
   scanf("%d", &vara.antal);

   /* rensa stdin*/
   skipline();

   /* skriv in varan på fil */
   lfil = fopen("lager.dat", "ab");
   fwrite(&vara, sizeof(struct varupost), 1, lfil);
   fclose(lfil);
}
```

OBS! Funktionen ny_vara öppnar strömmen med *a, som i append,* för att lägga till varan på slutet.

```
void visa_vara(void)
{
   FILE *lfil;
   struct varupost vara;
   char varubet[10 + 1];

   /* läs in sökt vara */
   printf("\nGe beteckning: ");
   gets(varubet);

   /* sök efter vara i lager */
   lfil = fopen("lager.dat", "rb");

   while (fread(&vara, sizeof(struct varupost), 1, lfil)
          && strcmp(varubet, vara.vb) != 0)
       ;

   if ( !feof(lfil) )
   {
       /* varan finns */
       printf("Beteckning: %s\n", vara.vb);
       printf("Pris: %.2f\n", vara.pris);
       printf("Antal: %d\n", vara.antal);
   }
   else
      printf("Varan finns ej i lagret !\n");
   fclose(lfil);
}
```

OBS! Funktionen strcmp som jämför två strängar och returnerar 0 om strängarna är lika. Vill man jämföra utan hänsyn till stora eller små bokstäver måste man använda stricmp som ignorerar typen på tecknen. (stricmp ingår inte i C-standarden, utan är en utökning som finns i vissa system. Den kan också heta strcasecmp.)

```
void nytt_pris(void)
{
   FILE *lfil;
   struct varupost vara;
   char varubet[10 + 1];

   /* läs in sökt vara */
   printf("\nGe beteckning: ");
   gets(varubet);

   /* sök efter vara i lager */
   lfil = fopen("lager.dat", "rb+");

   /* läs poster tills vi hittar den rätta,
       eller filen tar slut */
   while (fread(&vara, sizeof(struct varupost), 1, lfil) == 1
          && strcmp(varubet, vara.vb) != 0)
     ;
```

```
   if ( !feof(lfil) )
   {
       /* varan finns och ser ut som */
       printf("Beteckning: %s\n", vara.vb);
       printf("Pris: %.2f\n", vara.pris);
       printf("Antal: %d\n", vara.antal);

       /* nytt pris */
       printf("Nytt pris: ");
       scanf("%lf", &vara.pris);
       /* rensa */
       skipline();

       /* tillbaka till lagret */
       fseek(lfil, -1*sizeof(struct varupost), SEEK_CUR);
       fwrite(&vara, sizeof(struct varupost), 1, lfil);
   }
   else
      printf("Varan finns ej i lagret !\n");
   fclose(lfil);
}
```

OBS! Funktionen fseek som flyttar till önskad filpositionen. Med den andra parametern anger man offset i förhållande till någon utgångsposition som anges som tredje parameter. Utgångsposition anges med någon av konstanterna:

SEEK_CUR	----	om offset i förhållande till aktuell position
SEEK_SET	----	om offset i förhållande till filens början
SEEK_END	----	om offset i förhållande till filslut.

När man uppdaterar en post i ett register söker man först upp posten sekventiellt d.v.s man startar från filens början och letar tills man har funnit posten. Om posten finns på filen läser man in den i primärminnet och utför önskade förändringar i den. Därefter är det dags att skriva tillbaka posten till filen. Här får man tänka på att en läsning flyttat fram filpositionen till nästa post och man måste flytta tillbaka positionen de antal byte som posten består av. Det är detta som kallas direkt access och då använder man funktionen *fseek* enligt ovan.

Vill man ha reda på den filposition som man står på för tillfället använder man funktionen *ftell*, som returnerar nummer på den byte, räknat från filens början, som filpekaren står på just då. Funktionen ftell returnerar long int och börjar att räkna från 0.

Ex: Skriv de satser som behövs för att ta reda på en fils storlek i byte. Använd fseek och ftell.

```
.....
fseek(lfil, 0, SEEK_END);
storlek = ftell(lfil);
.....
```