7 Programmeringsteknik

(Senaste ändring av detta kapitel: 18 december 2006)

Att skriva ett program innebär att man skriver en plan för hur bearbetningen av data ska utföras. Vilken typ av data och vilken typ av bearbetning, som ska göras, ska vara bestämt i en *specifikation,* innan man börjar programmera. Man bör också se till att hela tiden under programmeringen hålla kontakt med beställaren eller den som har skrivit specifikationen för eventuella frågor.

Att tillverka programmet innebär inte enbart att skriva koden rakt upp och ner. Man bör ha någon form av strategi eller teknik som man följer då programmet skrivs. En modell kan vara att man delar upp programmeringen i ett antal moment där varje moment dokumenteras noggrannt. Följande moment bör åtminstone ingå:

1)	Uppdelning i delprogram	---	Exempelvis kan man använda *stegvis förfining* som man dokumenterar med *pseudokod* (även kallad *halvkod)* eller *strukturdiagram,* som visar den grova uppdelningen av programmet i funktioner.
2)	Algoritmformulering	---	*Problemlösning* där man gör nya funktioner av delproblem eller använder färdiga funktioner som man har i sitt bibliotek.Varje ny funktion bör dokumenteras med pseudokod eller strukturdiagram. Detta gäller åtminstone algoritmer som kommmer att återanvändas.
3)	Kodning	---	Man *realiserar* (*implementerar*) algoritmen genom att *koda (skriva)* den i något språk. *Koden* (*källkoden*) är inte bara instruktioner till datorn, utan också ett dokument som ska läsas av andra, så den ska vara snyggt skriven och bra kommenterad.
4)	Testning	---	Programmet testas genom att enskilda funktioner först testas var för sig, och sedan testas allt större och större ihopsatta delar. Testningen ska dokumenteras med ett *testprotokoll som visar testvärden och testresultat*. Observera att det aldrig går att testa sig fram till ett korrekt program. Det är mycket bättre att tänka ut en algoritm ordentligt, än att prova sig fram och tycka att man är klar när det verkar som att programmet fungerar.

Hur ska man dela upp i funktioner vid stegvis förfining? Detta är mycket individuellt och beror bland annat på vilken erfarenhet man har som programmerare och vilka färdiga funktioner man har tillgång till. Varje programmerare bör ha en uppsättning färdiga funktioner som löser ofta förekommande problem vid programmeringen. Hit hör bland annat algoritmer för basproblem som sökning, sortering och användargränssnitt (menyer etc).

7.1 Sökning

Sökproblemet innebär att i en mängd av element hitta det eftersökta elementet (den så kallade *nyckeln*). Beroende på hur mängden ser ut kan man använda sig av olika metoder. I en helt oordnad mängd måste man använda *linjär sökning*, d.v.s. söka från början så länge man ej hittat nyckeln. Har man en ordnad mängd kan man använda *binär sökning*, vilket innebär att man alltid kan söka i rätt halva av mängden.

7.1.1 Linjär sökning

När man ska lösa ett problem är det alltid bra att *lösa ett konkret exempel eller ett liknande men enklare problem först*. Problemet linjär sökning kan lösas enligt:

Konkret exempel:

Element:	52	41	36	14	22	34	12
Sökt nyckel:	14						
Lösning:	Jämför först 14 med 52, 14 med 41 o.s.v så länge nyckel skilt ifrån element. Om nyckeln finns i mängden anger man exempelvis vilket ordningsnummer den har, annars anges att den ej finns i mängden.						

Pseudokod:

```
hämta nyckel
hämta första elementet i mängden

så länge ej slut på mängden och nyckel skilt ifrån elementet i mängden
  hämta nästa element i mängden

om ej slut på mängden
  nyckel finns på elementets plats i mängden
annars
  nyckel finns ej
```

Kod:

```
/* söker efter key i v med nr element och returnerar på */
/* vilken plats key finns, eller -1 om den inte finns   */

int linsearch(int v[], int nr, int key)
{
   int i, svar;

   i = 0;

   while (i < nr && v[i] != key)
      i++;

   if (i < nr)
      svar = i; /* hittat */
   else
      svar = -1; /* ej hittat */

   return svar;
}
```

En alternativ (och mer idiomatisk!) lösning, som utnyttjar att `return` returnerar direkt:

```
int linsearch(int v[], int nr, int key)
{
   for (int i = 0; i < nr; ++i)
      if (v[i] == key)
         return i;
   return -1;
}
```

OBS! Samma princip kan användas för sökning av andra typer av data. Man måste dock exempelvis vid sökning av namn använda jämförelsefunktionerna för strängar, som strcmp eller stricmp (eller strcasecmp).

Ex: Skriv ett program som initierar en vektor och läser in ett tal som man ska söka efter i vektorn. Om talet finns i vektorn anges vilket ordningsnummer det har, annars skrivs att talet ej finns i vektorn. (Vi tänker oss att programmet är avsett för vanliga användare och inte C-programmerare, så det första talet har nummer ett!)

```
#include <stdio.h>

int main()
{
   int nyckel, nr, vek[10] = { 12, 34, 56, 23, 98,
                               16, 14, 25, 67, 38 };


   printf("Vilket tvåsiffrigt tal söks? ");
   scanf("%d", &nyckel);

   nr = linsearch(vek, 10, nyckel);
   if (nr >= 0)
      printf("Talet finns och har nummer %d!\n", nr + 1);
   else
      printf("Talet finns ej!\n");

   return 0;
}
```

7.1.2 Binär sökning

Vid binär sökning måste man ha en *ordnad mängd* som man letar i. Genom att hela tiden söka i rätt halva av mängden hittas nyckeln snabbare än vid linjär sökning.

Konkret exempel:

Mängd:	14 36 50 53 58 65 72
Nyckel:	50
Lösning:	Jämför nyckeln 50 med mittersta elementet 53. Eftersom 50 är mindre än 53 fortsätter man att söka efter 50 i den vänstra mängden 14, 36 och 50. Jämför 50 med det mittersta elementet i den nya mängden 36. Eftersom 50 är större än 36 så sök i den högra mängden 50. Jämför 50 med 50 och nyckeln hittad.
Nyckel:	67
Lösning:	Jämför nyckeln 67 med mittersta elementet 53. Eftersom 67 är större än 53 fortsätter man att söka efter 50 i den högra mängden 58, 65 och 72. Jämför 67 med det mittersta elementet i den nya mängden 65. Eftersom 67 är större så sök i den högra mängden 72. Jämför 67 med 72. Eftersom 67 mindre än 72 och ingen mer mängd att söka i så finns ej nyckeln i mängden.

Pseudokod:

```
hämta nyckel
hämta första elementets index och sista elementets index

om nyckel mindre än första elementet och nyckel större än sista elementet
  nyckeln finns ej
annars
  gör
    bestäm mittersta elementets index
    om nyckel mindre än mittersta
      byt sista elementidex till ett under mittersta
    annars om nyckel större än mittersta
      byt första elementindex till ett över mittersta
  så länge nyckel skilt ifrån mittersta element och mängd kvar att dela

  om nyckel lika med mittersta
    nyckel finns i mittersta
  annars
    nyckel finns ej
```

Kod:

```
/* söker efter key i v med nr sorterade element i     */
/* stigande ordning och returnerar på vilken plats    */
/* key finns eller -1 om den inte finns               */

int binsearch(int v[], int nr, int key)
{
    int mitti, mini = 0, maxi = nr - 1, svar;

    if (key < v[mini] || nr <= 0 || key > v[maxi])
       svar = -1;
    else
    {
       do
       {
          mitti = (mini + maxi) / 2;
          if (key < v[mitti] )
             maxi = mitti - 1;
          else if (key > v[mitti])
             mini = mitti + 1;
       }
       while (v[mitti] != key && mini <= maxi);

       if (v[mitti] == key)
          svar = mitti;
       else
          svar = -1;
    }

    return svar;
}
```

På liknande sätt som vi tidigare såg med funktionen `linsearch`, kan man förenkla koden genom att returnera direkt när man hittat det sökta elementet.

Ex: Skriv ett program som initierar en ordnad vektor och läser in ett tal som ska sökas efter i vektorn. Finns nyckeln i vektorn skrivs dess ordningstal ut annars att den inte finns. (Även här tänker vi oss att programmet är avsett för vanliga användare och inte C-programmerare, så det första talet har nummer ett.)

```
#include <stdio.h>

int main()
{
   int nyckel, nr, vek[10] = { 12, 14, 23, 38, 56,
                               67, 78, 83, 88, 92 };

   printf("Vilket tvåsiffrigt tal söks ? ");
   scanf("%d", &nyckel);

   nr = binsearch(vek, 10, nyckel);
   if (nr >= 0)
      printf("Talet finns och har nummer %d!\n", nr + 1);
   else
      printf("Talet finns ej!\n");

   return 0;
}
```

7.2 Sortering

En mängd kan sorteras i stigande eller fallande storleksordning. Man kan sortera tal i storleksordning eller namn i bokstavsordning. Det finns ett stort antal algoritmer för sortering. En algoritm passar bättre för en typ av mängd än för en annan, och en annan algoritm är snabbare än den andra. När det gäller snabbhet bör man för att få en snabb algoritm hålla nere *antalet byten* i första hand och *antalet jämförelser* i andra hand. (Det beror dock på hur data är lagrat. Ibland tar jämförelsen längre tid än bytet, och då bör man förstås i första hand hålla nere antalet jämförelser.)

7.2.1 Urvalssortering

Urvalssortering är en mycket enkel sorteringsalgoritm som passar bra till en helt oordnad mängd.

Konkret exempel:

Mängd: 8 18 -1 7 0 -19 3

Lösning: Sök efter det minsta elementet i hela mängden som är -19 och byt plats mellan detta och det första elementet 8 enligt:

-19 18 -1 7 0 8 3

Fortsätt sedan med resten av mängden och sök efter minsta element som blir - 1. Byt plats enligt:

-19 -1 18 7 0 8 3

o.s.v så länge ej slut på mängden.

Pseudokod:

```
för alla element från första till näst sista i mängden

  sätt minindex till elementets index

  för alla resterande element i mängden
    om resterande element mindre än minindexelementet
      sätt minindex till resterande elements index

  byt plats mellan minindexelement och element
```

Kod:
```
/* sorterar v med nr st element i stigande ordning */
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;
   }
}
```

Ex: Skriv ett program som slumpar 100 heltal till en vektor, sorterar och skriver ut vektorn.

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

#define ANTAL_TAL 100

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

   srand((unsigned)time(NULL));
   for (i = 0; i < nr; i++)
      v[i] = 100 + rand()%900;
}

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

   for (i = 0; i < nr; i++)
   {
      if (i % 15 == 0)
         putchar('\n');
      printf("%4d", v[i]);
   }
   putchar('\n');
}

int main()
{
   int vek[ANTAL_TAL];

   slump(vek, ANTAL_TAL);
   skriv(vek, ANTAL_TAL);
   ursort(vek, ANTAL_TAL);
   skriv(vek, ANTAL_TAL);

   return 0;
}
```

7.2.2 Bubbelsortering

Bubbelsortering är en lämplig sorteringsmetod då *endast några element ligger i fel ordning*, exempelvis efter en uppdatering. Annars är den mycket långsammare än andra metoder.

Konkret exempel:

Mängd: 8 18 -1 7 0 -19 3

Lösning: Jämför 8 med 18. Byt ej. Jämför 18 med -1 och byt till:

8 -1 18 7 0 -19 3

Jämför 18 med 7 och byt till:

8 -1 7 18 0 -19 3

Efter fortsatta jämförelser och byten har det största elementet 18 *bubblat upp till ytan*. Starta sedan från början och låt det näst största elementet bubbla upp till näst sista plats o.s.v. så länge det bubblar.

Kod:

```
/* sorterar v med nr st element i stigande ordning*/

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--;
   }
}
```

OBS! Eftersom bubbelsortering alltså är mycket långsammare än andra sorteringsmetoder, utom när elementen redan ligger i nästan rätt ordning, bör man normalt undvika bubbelsortering. Bubbelsortering har dock fördelen att vara ganska enkel att förklara, jämfört med andra sorteringsmetoder, och därför ser man den ofta i grundkurser i programmering.

7.2.3 Instickssortering

Instickssortering används då man ska fylla på en tom mängd. Genom att se till att alltid stoppa in det nya elementet i mängden på rätt ställe har man alltid en sorterad mängd.

Konkret exempel:

Osorterad mängd :	8	18	-1	7	0	-19	3
Sorterad mängd:	8						
	8	18					
	-1	8	18				
	-1	7	8	18			
	o.s.v						

Lösning: Sätt in 8 på första plats. Jämför 18 med 8 och sätt in 18 efter 8. Jämför -1 med 18 och flytta fram 18. Jämför -1 med 8 och flytta fram 8. Sätt in -1 på första plats o.s.v så länge element att sätta in.

Kod:

```
/* sorterar vektorn v till vektorn s med nr st element i   */
/* stigande ordning                                        */
void insort(int v[], int s[], int nr)
{
   int i, j, klar;

   s[0] = v[0];
   for (i = 1; i < nr; i++)
   {
      klar = 0;
      j = i - 1;
      do
      {
         /* flytta fram */
         if (v[i] < s[j])
         {
            s[j+1] = s[j];
            j--;
         }
         /* sätt in */
         else
         {
            s[j+1] = v[i];
            klar = 1;
         }
      }
      while (!klar && j > -1);

      /* om första plats */
      if (!klar)
         s[0] = v[i];
   }
}
```

Ovan användes en extra vektor som parameter för att sortera med instickssortering. Man kunde istället ha använt en lokal vektor som sorteras och sedan kopieras över till den ursprungliga vektorn eller också kunde man ha *sorterat direkt då man skapade* mängden vid inläsningen eller slumpningen.

Ex : Skriv ett program som slumpar en sorterad vektor med 1000 hela tresiffriga tal och skriver ut vektorn.

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

void slumpsort(int v[], int nr, int min, int max)
{
   int i, j, klar, tal;

   srand((unsigned)time(NULL));
   v[0] = rand()%(max-min+1) + min;
   for (i = 1; i < nr; i++)
   {
      klar = 0;
      j = i - 1;
      tal = rand()%(max-min+1) + min;
      do
      {
         if (tal < v[j])
         {
            v[j+1] = v[j];
            j--;
         }
         else
         {
            v[j+1] = tal;
            klar = 1;
         }
      }
      while (!klar && j > -1);

      if (!klar)
         v[0] = tal;
   }
}


#define ANTAL_TAL 1000

int main()
{
   int vek[ANTAL_TAL];

   slumpsort(vek, ANTAL_TAL, 100, 999);

   /* antag att skriv finns */
   skriv(vek, ANTAL_TAL);

   return 0;
}
```

Man kan också använda instickssortering för att sortera en array ”på plats”, på samma sätt som vi gjorde med bubbelsortering. Då låter man *början på arrayen* vara den sorterade delen, och flyttar fram gränsen mellan sorterad och osorterad del ett steg för varje element som stoppas in i den sorterade delen. Det går vi dock inte igenom i den här kursen.

7.3 Kodning

Har man gjort ett bra arbete vid algoritmformuleringen, som resulterat i väl genomtänkt pseudokod eller strukturdiagram, brukar ej kodningen ställa till något problem. Däremot är det viktigt att ha en *god stil* vid kodningen eftersom koden är ett viktigt dokument för underhåll och förändringar.

Regel: **Man ska skriva sitt program som om man skrev det för någon annan, för nästa gång man läser programmet är man en annan!**

Hur ska man då lära sig en god kodningsstil. Det bästa sättet är att studera andras kod och kritisera det man tycker är fel samt hoppas på att andra läser ens egen kod och kritiserar den.

Ex:

```
for (i = 0; i < n; i++)
   for (j = 0; j < n; j++)
      v[i][j] = ((i+1)/ (j+1)) * ((j+1) /( i+1));
```

Vad gör ovanstående kod? Här får man verkligen tänka efter vad koden gör. Koden tillverkar en enhetsmatris med ettor i diagonalen och nollor för övrigt. Vad smart! Eller?

Här ska man åtminstone ha en kommentar före koden enligt:

```
/* tillverka en enhetsmatris */
for (i = 0; i < n; i++)
   for (j = 0; j < n; j++)
      v[i][j] = ((i+1)/ (j+1)) * ((j+1) /( i+1));
```

Frågan är också om man inte ska koda på ett klarare och tydligare sätt och verkligen markera vad koden gör enligt:

```
/* tillverka en enhetsmatris */
for (rad = 0; rad < max; rad++)
{
   for (kol = 0; kol < max; kol++)
   {
      /* sätt in 0 utom i diagonalen som sätts till 1 */
      if (rad != kol)
         v[rad][kol] = 0;
      else
         v[rad][kol] = 1;
   }
}
```

Regel: **Skriv tydligt och klart! Var ej för smart vid kodningen! Tänk på att någon annan ska förändra och underhålla koden! Kommentera vettigt!**

Kommentarer är viktiga för att göra koden tydligare och mera lättläst. En annan detalj som också är viktig att tänka på när man skriver kod är *variabelnamnen*. Väl valda variabelnamn, som klart utsäger vad variabeln användes till, ger klarare kod. Jämför a) med b) nedan. Vilken kod är lättast att förstå?

Ex: a)
```
/* beräkning av poäng i bowling */
s = 0;
b = 1;
for (i = 1; i <= 10; i++)
{
   if ( p[b] == 10 )
   {
      s = s + 10 + p[b+1] + p[b+2];
      b++;
   }
   else if ( p[b] + p[b+1] == 10 )
   {
      s = s + 10 + p[b+2];
      b += 2;
   }
   else
   {
      s = s + p[b] + p[b+1];
      b += 2;
   }
}
```

b)
```
/* beräkning av poäng i bowling */
score = 0;
ball = 1;
for (frame = 1; frame <= 10; frame++)
{
   /* strike */
   if ( pins[ball] == 10 )
   {
      score += 10 + pins[ball+1] + pins[ball+2];
      ball++;
   }
   /* spare */
   else if ( pins[ball] + pins[ball+1] == 10 )
   {
      score += 10 + pins[ball+2];
      ball += 2;
   }
   else
   /* regular */
   {
      score += pins[ball] + pins[ball+1];
      ball += 2;
   }
}
```

Regel: **Använd namn på variabler med mera som utsäger något och skiljer sig från varandra på ett tydligt sätt!**

En viktig del av den kod som man skriver utgörs av strukturerna selektion och iteration. Här gäller det att *tänka först och koda sedan*. Man bör ha tänkt ut en vettig struktur i exempelvis pseudokod först innan man sätter sig ner för att koda.

När det gäller *iterationer* ser man i en del program att man *använder for-loopar i alla lägen* istället för den kanske mer logiska while-konstruktionen. För iterationer är följande regel lämplig när man ska välja loop-sats.

Regel: **Man ska använda for-loopar endast då man vet före loopen hur många gånger något ska upprepas. Annars ska man använda while-loopar.**

Selektioner kräver ofta extra eftertanke för att bli klara och tydliga.

Ex:

```
if (pris <= 50.0)
   rabatt = 0.0;
if (pris > 50.0)
   if (pris <= 100.0)
      rabatt = 0.02*pris;
if (pris > 100.0)
   rabatt = 0.03*pris;
```

Flera if-satser inuti varandra kan ofta sättas ihop till en, genom att använda villkor med de logiska operatorerna &&, || och !.

```
if (pris <= 50.0)
   rabatt = 0.0;
if (pris > 50.0 && pris <= 100)
      rabatt = 0.02*pris;
if (pris > 100.0)
   rabatt = 0.03*pris;
```

Vid uteslutande händelser, som det handlar om ovan, är det ännu klarare och tydligare att skriva en flervalsselektion.

```
if (pris <= 50.0)
   rabatt = 0.0;
else if (pris <= 100.0)
   rabatt = 0.02*pris;
else
   rabatt = 0.03*pris;
```

Även när man använder flervalsselektioner kan det bli snårigt, om man ej tänkt igenom selektionerna ordentligt i förväg.

Ex:

```
if (length >= 30 && length < 50)
   if (wing < 0.6*length)
      weight1 = (1 + 0.08 - 0.037)* weight;
   else
      weight1 = (1 + 0.08 + 0.045)*weight;
else if (length >= 50 && length < 60)
   if (wing < 0.6*length)
      weight1 = (1 + 0.09 - 0.037) * weight;
   else
      weight1 = (1 + 0.09 + 0.045) * weight;
else if (length >= 60 && length < 80)
   if (wing < 0.6*length)
      weight1 = (1 + 0.105 - 0.037) * weight;
   else
      weight1 = (1 + 0.105 + 0.045) * weight;
else
   if (wing < 0.6*length)
      weight1 = (1 + 0.122 - 0.037) * weight;
   else
      weight1 = (1 + 0.122 + 0.045) * weight;
```

Tänker man efter här ser man att selektionen kan skrivas betydligt enklare enligt:

```
/* correction for wing */
if (wing < 0.6 * length)
   corr = 1.0 - 0.037;
else
   corr = 1.0 + 0.045;

/* correction for length */
if (length >= 80)
   corr += 0.122;
else if (length >= 60)
   corr += 0.105;
else if (length >= 50)
   corr += 0.09;
else if (length >= 30)
   corr += 0.08;

/* new weight */
weight1 = corr * weight;
```

Regel: **Man ska tänka efter ordentligt innan man kodar selektioner. Undvik nästlade if-satser och använd flervalsselektioner där det går!**

Data som ska matas in till ett program ställer alltid till trassel. Orsaken till detta är att det är människor som matar in data och människor gör fel. Därför ska programmet konstrueras så att det *även klarar av att hantera felaktiga indata*.

Den metod som man ofta använder sig av är ett *inmatningsfilter som inte släpper igenom annat än korrekt och acceptabel indata*.

Ex: Skriv ett program som beräknar en triangels area med Herons formel. Alla triangelsidor ska inläsas säkert och man får ej acceptera sidor som ej bildar någon triangel.

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

int main()
{
   double sid[3], omk, area;
   char sidstr[30 + 1];
   int nr;

   /* säker inmatning av tre triangelsidor */
   do
   {
      for (nr = 0; nr < 3; nr++)
      {
         do
         {
            printf("Ge sida %d: ", nr);
            fgets(sidstr, sizeof(sidstr), stdin);
            sid[nr] = atof(sidstr);
            if (sid[nr] <= 0)
               printf("Inte en tillåten sidlängd."
                      " Försök igen.\n");
         }
         while (sid[nr] <= 0);
      }

      omk = (sid[0] + sid[1] + sid[2]) / 2;

      if (sid[0] > omk || sid[1] > omk || sid[2] > omk)
         printf("Den triangeln är omöjlig. Försök igen.\n");
   }
   while (sid[0] > omk || sid[1] > omk || sid[2] > omk);

   /* beräkning av triangelns area med Herons formel */
   area = sqrt(omk*(omk-sid[0])*(omk-sid[1])*(omk-sid[2]));
   printf("Arean = %.2f\n", area);

   return 0;
}
```

OBS! När vi nu ska kontrollera inmatningen, använder vi förstås funktionen `fgets` och inte `gets`, eftersom `gets` inte kontrollerar att inlästa data faktiskt får plats i strängvariabeln. Det kan ge upphov till besvärliga och svårhittade fel.

OBS! Funktionen atof returnerar normalt det omvandlade reella talet. Går strängen ej att omvandla returneras 0. Ovan kan vi ej heller acceptera negativa resultat.

Välj indata som är mänskliga. Ord från vardagslivet, ord som säger något, ord som har en mening. Ofta använder man tal och siffror till allting, Använd istället strängar eller egenuppräknade variabler.

Ex: Skriv ett program som läser in metaller och deras vikt samt beräknar priset. Använd ej 1, 2 och 3 för metallen utan använd metallens kemiska beteckning.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <strings.h> /* för strcasecmp – ej standard */

int main() {
   enum metalltyp { Al, Sn, Cu } metall;
   double vikt, pris[3] = { 3.0, 2.0, 5.0 };
   char viktstr[30 + 1], metallstr[30 + 1];
   int ok;

   /* säker inmatning av metall */
   do {
      ok = 1;
      printf("Ge metall (Al, Sn, Cu): ");
      fgets(metallstr, sizeof(metallstr), stdin);
      /* ta bort (eventuellt) radslutstecken ur strängen */
      if (strrchr(metallstr, '\n') != NULL)
          *strrchr(metallstr, '\n') = '\0';
      if (strcasecmp(metallstr, "Al") == 0)
         metall = Al;
      else if (strcasecmp(metallstr, "Sn") == 0)
         metall = Sn;
      else if (strcasecmp(metallstr, "Cu") == 0)
         metall = Cu;
      else {
         ok = 0;
         printf("Inte en tillåten metall. Försök igen.\n");
      }
   } while ( !ok );

   /* säker inmatning av vikt */
   do {
      printf("Ge vikt: ");
      fgets(viktstr, sizeof(viktstr), stdin);
      vikt = atof(viktstr);
      if (vikt <= 0)
         printf("Inte en tillåten vikt. Försök igen.\n");
   } while (vikt <= 0);

   /* beräkning av pris */
   printf("Pris = %.2f\n", pris[metall]*vikt);

   return 0;
}
```

Regel: **Kräv indata som är mänskliga! Testa alltid att indata är korrekt! Släpp ej igenom felaktig indata!**

7.4 Testning

En viktig del av programmeringen är att se till att programmet klarar av att hantera felaktiga indata. Ännu viktigare är naturligtvis att se till att programmet bearbetar rätt indata på ett korrekt sätt.

Har man tänkt igenom sina strukturer ordentligt, brukar det mesta fungera. Testningen är ett sista moment i programmeringen och ska naturligtvis utföras på ett genomarbetat sätt och dokumenteras med testdata och testresultat. Speciellt viktigt är det att testa på extremvärden och randvärden.

Räknare i loopar måste man se upp med så att de räknas upp korrekt.

Ex: Försök till en medelvärdesberäkning:

```
sum = 0;
antal = 1;
printf("Ge tal: ");
scanf("%lf", &tal);
while (tal != 0.0)
{
   sum += tal;
   antal++;
   printf("Ge tal: ");
   scanf("%lf", &tal);
}
printf("Medelvärde = %f\n", sum/antal);
```

Antalet räknas upp fel. Man har ett för mycket i antal vid iterationens slut. Korrigera till:

```
antal = 0;
```

Ex: Försök till summering av 100 vektorelement:

```
sum = 0;

for(i = 0; i <= 100; i++)
   sum += vek[i];

printf("Summan = %f\n", sum);
```

Här summeras ett extra element i minnet som kan innehålla vilket värde som helst. Programmet fungerar om detta extra element skulle råka vara 0. Korrigera till:

```
for(i = 0; i < 100; i++)
```

Regel: **Testa speciellt på randvärden och extremvärden samt se upp med att initieringarna är riktiga och att räknare räknas upp korrekt!**

Det svåraste felen att hitta vid testning är sådana fel som *endast dyker upp ibland* för vissa typer av data. Ett sådant fel brukar vara att man testar likhet mellan flyttal. Att jämföra ett flyttal med 0 brukar alltid gå bra. Att däremot jämföra om två flyttal är lika eller olika kan lyckas för vissa värden och misslyckas för andra. Reella tal kan ju, som vi vet, inte alltid lagras exakt på flyttalsformat. Exempel är 1/3 och 1/5. Inget av dessa två tal kan lagras exakt som flyttal. (1/3 kan inte heller lagras exakt som ett tal med decimaler och basen 10.)

Ex: Skriv ett program som övar division av flyttal.

```
#include <stdio.h>

int main() {
   float taljare, namnare, kvot, svar;
   int antal = 0;

   printf("Ge täljare: ");
   scanf("%f", &taljare);
   printf("Ge nämnare: ");
   scanf("%f", &namnare);

   kvot = taljare / namnare;

   do {
      printf("Vad blir kvoten ? ");
      scanf("%f", &svar);
      if (svar == kvot)
         printf("Svaret är rätt!\n");
      else
      {
         printf("Svaret är fel!\n");
         antal++;
         if (antal == 3)
            printf("Korrekt svar = %f\n", kvot);
      }
   } while (svar != kvot && antal < 3);

   return 0;
}
```

Programmet ovan fungerar för 1.8/0.9 om man ger svaret 2.0, men inte för 1.8/0.6 om man ger svaret 3.0. Byter man från `float` till flyttal med dubbel precision (`double`) fungerar båda dessa exempel, men inte andra. Och på en annan typ av dator, med en annan flyttalsrepresentation i processorn, får man kanske helt andra fel!

Det fel som man gör ovan är jämförelserna:

```
if (svar == kvot) ...

... while (svar != kvot && antal < 3);
```

Man ska aldrig jämföra exakt likhet mellan flyttal. Däremot kan man kontrollera skillnaden mellan talen och se om den är mindre än den noggrannhet som man tycker behövs:

```
if (fabs(svar - kvot) <= fabs(kvot*1e-6))  ...

... while (fabs(svar - kvot) > fabs(kvot*1e-6) && antal < 3);
```

Regel: Testa aldrig direkt likhet eller olikhet mellan flyttal!