Föreläsning 3

3.3 `for`-satsen

I programmering talar man om tre sätt på vilket ett programflöde kan vara beskaffat:

1. Saker kan hända efter varandra, i *sekvens*.

2. Flödet kan innebära att ett val görs, antingen görs en sak, eller så görs en annan, eller en tredje etc, det sker en *selektion*.

3. En sak kan behöva utföras/hända flera gånger, vi har *iteration*.

De två första sätten har vi redan sett exempel på, i våra första program hände saker i sekvens, först inmatning, sedan beräkning och sedan utmatning. Med `if`-satsen har vi också sett exempel på hur ett program kan göra *val*, det är alltså selektion. Vi ska ni se på hur vi kan få ett program att utföra saker flera gånger, det sker med den så kallade `for`-satsen. Det första exemplet ur boken ser ut så här:

```
for (k=1;k<=10;k=k+1)
  printf("%d ", k);
```

Denna `for`-sats ska utläsas så här:

"Tag variabeln `k` (för `k`), tilldela den först 1, så länge `k` är mindre än eller lika med 10, utför `printf`-satsen, öka sedan värdet på `k` med 1 och håll på så här så länge villkoret `k<=10` är uppfyllt."

Körningen ser ut så här:

```
1 2 3 4 5 6 7 8 9 10
```

En utskrift av de tio värden som `k` får i `for`-satsen. Satsen `printf("%d ", k)` upprepas (itereras) under det att värdet på `k` genomlöper de värden som `for`-satsen styr den till.

Allmänt skriver man en `for`-sats så här:

```
for(startsats;testvillkor;ändring)
  sats som ska utföras flera gånger (beroende på testvillkoret)
```

Startsatsen är ofta en variabeltilldelning, det kallas i boken *initiering*. Man brukar tala om detta som en loop, (eller slinga) och variabeln kallas då en *loopvariabel*. Variabeln `k`, ovan, var en loopvariabel. Man säger att loopen löper, kör eller snurrar, ett antal varv och det antal varv som den löper avgörs av testvillkoret, innan varje varv så testas testvillkoret, om det är uppfyllt så körs ett varvet, om testvillkoret inte är uppfyllt så är `for`-satsen fullbordad. I slutet av varje varv görs ändringen som definieras sist innanför `for`-satsens parenteser, ändringen innebär ofta att loopvariablens värde påverkas, ovan ökades `k` med 1 för varje varv, men ändringen kan se annorlunda ut än så.

Vi ser på ett annat exempel ur boken, en loop som innebär att loopvariabeln räknas ner istället:

```
for(k=10;k>0;k=k-1)
  printf("%d ", k);
```

Denna loop gör samma sak som den innan fast baklänges, den börjar med variabeln `k` lika med 10 och så länge `k` är positivt så skrivs värdet på `k` ut. Det ger oss följande körning: `10 9 8 7 6 5 4 3 2 1`, som vi ser, samma resultat som ovan fast bakvänt.

En `for`-sats blir mer kraftfull om man låter den styra en sammansatt sats, följande exempel räknar samman summan av alla tal från 1 till och med 10 och skriver ut delsummorna i varje varv:

```
summa = 0;
for(k=1;k<=10;k=k+1)
{
  summa = summa + k;
  printf("Summa hittills: %d\n", summa);
}
```

En körning ser ut så här:

```
Summa hittills: 1
Summa hittills: 3
Summa hittills: 6
Summa hittills: 10
Summa hittills: 15
Summa hittills: 21
Summa hittills: 28
Summa hittills: 36
Summa hittills: 45
Summa hittills: 55
```

(Gå igenom rutinen rad för rad och förklara.)

Man kan nästla loopar, det vill säga man ska skapa en loop inuti en loop, då får man något som kallas en dubbelloop. Ett exempel på en dubbelloop är:

```
for(i=1;i<=7;i=i+1)
{
  for(j=1;j<=8;j=j+1)
    printf("(%d,%d) ", i, j);
  printf("\n");
}
```

När denna loop kör ger den resultatet:

```
(1,1) (1,2) (1,3) (1,4) (1,5) (1,6) (1,7) (1,8)
(2,1) (2,2) (2,3) (2,4) (2,5) (2,6) (2,7) (2,8)
(3,1) (3,2) (3,3) (3,4) (3,5) (3,6) (3,7) (3,8)
(4,1) (4,2) (4,3) (4,4) (4,5) (4,6) (4,7) (4,8)
(5,1) (5,2) (5,3) (5,4) (5,5) (5,6) (5,7) (5,8)
(6,1) (6,2) (6,3) (6,4) (6,5) (6,6) (6,7) (6,8)
(7,1) (7,2) (7,3) (7,4) (7,5) (7,6) (7,7) (7,8)
```

Ett slags schackbräde av talpar, inuti loopen ser vi `printf("(%d,%d) ", i, j);`, det är den utskriften som åstadkommer de olika talparen, variablerna `i` och `j` genomlöper alla möjliga kombinationer från 1 till och med 7 respektive 1 till och med 8, den yttre loopen styr `i` och får den att genomlöpa alla värden från 1 till och med 7, för varje sådant värde på `i` genomlöper `j` alla värden från 1 till och med 8. Varje talpar skrivs ut. Efter den inre loopen finns ett radbyte så att när en rad är fullbordad (j går från 1 till 8) så görs ett radbyte. Observera att klamrarna i den yttre loopen är nödvändiga, annars skulle inte radbytet (satsen `printf("\n");`) vara med i loopen och endast ett radbyte skulle skett efter alla talpar skrivits ut på en jättelång rad. Testa detta själva!

Läs om den här konstruktionen själva: `for (;;)` (Exempel 14.)

3.4 Maximum och minimum

I olika programmeringsproblem behöver man söka efter största och minsta värdet av en mängd av tal. Vi ser på ett programexempel som tar fram det största respektive det minsta av en följd av 5 tal som matas in från tangentbordet: (Exempel 16, avkortat till 5 tal.)

```
#include <stdio.h>
#include <conio.h>
int main(void){
  int max,min,k,tal;
  max=0; min=1000;
  for(k=1;k<=5;k++){
    printf("Tal nr %d: ",k);
    scanf("%d",&tal);
    if (tal<min) min=tal;
    if (tal>max) max=tal;
  }
  printf("%d %d\n",max,min);
  getch();
}
```

Ett körexempel: (användarens inmatningar med fet stil)

```
Tal nr 1: 100
Tal nr 2: 50
Tal nr 3: -40
Tal nr 4: 2
Tal nr 5: 40
100 -40
```

Vi ser att programmet korrekt tar fram det största talet (som är 100) och det minsta (som är -40.) Hur fungerar programmet? Jo, programmet antar att det minsta möjliga som kan matas in är 0 och det största möjliga som kan matas in är 1000. Det är då lämpligt att ta två variabler som man vill lagra det största respektive det minsta talet i. Då väljer man variabelnamnen `max` respektive `min` och börjar med att anta att det hittills funna minsta värdet är 1000. Säkert, tänker man, kommer användaren att mata in något mindre än 1000 som då blir det hitttills minsta funna värdet. Loopen kör genom att låta användaren mata in ett tal till, detta jämförs med det hittills minsta funna värdet, om det inmatade värdet är mindre har programmet funnit något mindre än det hittills minsta inmatade värdet och ersätter alltså det hittills minsta inmatade värdet med det inmatade värdet. Efter fem varv har man funnit det minsta inmatade värdet. (Samma princip, fast omvänt gäller för det största värdet.)

Det finns ett problem med programmet ovan, vad? Jo det fungerar bara om alla inmatade tal säkert ligger mellan 0 och 1000. Om vi säger att alla inmatade tal vore -500 så skulle både det största och det minsta värdet vara -500, men programmet skulle tro att `max` var 0 eftersom villkoret i den här `if`-satsen:

```
if (tal>max) max=tal;
```

som styr hanteringen av det hittills största funna talet aldrig blir uppfyllt.

3.5 Att programmera och 3.6 Programutveckling

Viktigt avsnitt. Men läs det själva. Kontentan är: Programmera mycket! Jag vill också tillfoga mitt råd om att alltid programmera i små steg och alltid arbeta med körbara program.

4.1 Datatyper

Vi har stött på två olika typer av data i våra tidigare exempel: `int`, för heltal, och `float`, för flyttal. Det finns dock flera. Hur mycket data de kan lagra beror på antalet bitar som de använder. Ett vanligt `int` använder vanligen 32 bitar och kan med det lagra heltal i intervallet -2 miljarder till +2 miljarder. Vi kan dock välja typen `short int` och får då 16 bitar att lagra heltal med, det räcker till att lagra tal mellan -32768 och +32767. Det kanske inte tycks lämpligt, men det kanske visst är så, på vissa inbyggda system klarar inte processorn mer än 16 bitar och det händer då att det är `short int` som gäller om man inte vill ha flera busscykler per access till en variabel. En hyfsat fullständig lista över datatyper finns på sidan 95 i boken.

4.2 Varför ska variabler deklareras?

Därför att variablerna har olika uppgifter och funktioner, loopräknare kanske naturligt är heltal men ett värde som lagrar en vinkel i radianer eller ett sinusvärde måste vara flyttal. För att kompilatorn ska veta hur den data som representeras ska tolkas måste man ange typen hos en variabel, det anger man i deklarationen och det är därför deklarationen är nödvändig. Om vi försöker göra satsen

```
printf("%d", vinkel);
```

och variabeln vinkel är av typen `float` så blir det *helt fel* eftersom bitarna i en flyttalsrepresentation (`float` = typen på vinkel) har *helt andra* betydelser än bitarna i en heltalsrepresentation (`int` = den typ som omvandlingsspecifikationen `%d` anger).

4.3 Mer om `printf`

Med `printf` kan vi som bekant göra utskrifter på skärmen, vi kan skriva ut heltal, flyttal, strängar (teckenföljder) och enskilda tecken. Vi studerar lite exempel på `printf()` för att klarera lite begrepp:

```
int h=7, k=14;
printf("Totalt %d hästar och %d kor", h, k);
```

Dessa satser ger upphov till utskriften `Totalt 7 hästar och 14 kor`. Strängen `"Totalt %d hästar och %d kor"` som förekommer först i `printf()` kallas, som vi förut nämnt, *formatsträng*, det är den som anger i stort vad som ska skrivas ut. Formatsträngen innehåller så kallade *omvandlingsspecifikationer*, här för två heltal, `%d` och `%d`. Vi har dock sett att andra omvandlingsspecifikationer behövs för flyttal, vi skriver då `%f` istället. En omvandlingsspecifikation kan dock modifieras så att uskriften får olika precision och vidd, det är mest användbart när det gäller flyttal, om vi vill skriva ut talet vinkel som har värdet 1.23456 så kan vi skriva

```
printf("%5.2f",vinkel);
```

för att få vidden 5 på utskriften och 2 decimalers precision. Vidden måste överskrida antalet behövda tecken för att ha en effekt. Studera själv hur ni styr layouten genom att skriva in och

testköra exempel nummer 3, i boken sidan 99. Layout är viktigt eftersom det är en del av användagränssnittet, som programmerare ska man alltid sträva efter en bra gränsyta mot användaren, ett program som är svårbegripligt blir lätt värdelöst!

4.4 Mer om `scanf`

En `scanf`-sats med två indata: `scanf("%d %f", &antal, &vikt);` Denna låter oss mata in två värden i två olika variabler, ett heltal i variabeln `antal` och ett flyttal i variabeln `vikt`. I respons till en `scanf`-sats av denna typ kan man skriva till exempel: **`20 4.67`** om man vill mata in 20 i `antal` och 4.67 i `vikt`. Liksom `printf()` har `scanf()` en formatsträng och dess omvandlingsspecifikationer fungerar ungefär som omvandlingsspecifikationerna för `printf()`. Läs själva detaljerna om detta. Vi ska bara nämna lite om och-tecknet framför variabelnamnen (`&`), det är i själva verket en operator som kallas `adressoperatorn`. Den tar fram adressen, alltså förvaringsplatsen i datorns minne, för den variabel den opererar på. Det är ju naturligt att vi behöver veta en variabels minnesposition om vi vill mata in ett värde i den variabeln. Därför finns adressoperatorn med då vi anropar `scanf()`.

4.5 `while`-satsen

Då vi vet i förväg hur många varv en loop ska köra är det lämpligt att använda en `for`-sats. Det finns dock situationer då man inte vet hur många varv en loop kan behöva för att fullborda sin uppgift och då är det bra att använda en så kallad `while`-sats. Konstruktionen ser ut så här:

```
while(villkor)
  sats
```

Satsen `sats` utförs så länge villkoret är sant, och det är förstås en väldigt enkel konstruktion. Vi tittar på hur man kan skriva ut talen 1 till och med 10 med en for-loop och med en `while`-loop.

`for`-loop:

```
for(k=1;k<=10;k=k+1)
  printf("%d ", k);
```

`while`-loop:

```
k=1;
while(k<=10)
{
  printf("%d ", k);
  k = k + 1;
}
```

Dessa båda rutiner (programsnuttar) utför samma sak. Vi ser att `for`-loopen är enklare och kompaktare och att `while`-loopen kräver att man skriver mer kod. Detta illustrerar just för- och nackdelar med de båda konstruktionerna, `for`-loopen är kompaktare att programmera upp, och `while`-loopen är lite mer omständlig, men: `while`-loopen kan användas i mer situationer än `for`-loopen. Vi tar ett annat exempel där en `for`-loop inte skulle kunna användas lika naturligt:

```
int sum=0, tal=1;
while(tal!=0){
  printf("Ett tal: "); scanf("%d",&tal);
  sum=sum+tal;
}
printf("%d",sum);
```

Denna rutin accepterar inmatningar ända tills 0 matas in, då bryts loopen och summan skrivs ut.

4.6 `do`-satsen

En släkting till `while`-loopen är `do`-loopen, eller `do`-satsen. En `while`-loop kör så länge ett villkor är sant och kontrollen av huruvida man ska köra ett varv till i loopen eller ej sker *innan* varvet påbörjas. För `do`-loopen sker kontrollen *efter* varje varv fullbordats. Det betyder att en sådan loop alltid kör minst ett varv. Syntaxen för `do`-loopen är:

```
do
  sats
while(villkor);
```

Vi tar ett exempel ur boken på en gång: (Exempel 11.)

```
int sum=0,t;

do{
  printf("Ett tal: ");
  scanf("%d",&t);
  sum=sum+t;
} while(sum<1000);

printf("%d\n",sum);
```

Denna rutin accepterar inmatningar av heltal så länge den totala summan understiger 1000. Så fort summan överstiger eller är lika med 1000 så bryts den, då är villkoret i `while`-delen (`sum<1000`) inte längre uppfyllt och vi kommer ut ur loopen till utskriften med `printf` som ger oss totala summan som finns lagrad i variabeln `sum`.

4.7 Delbarhet

I det här avsnittet ska vi introducera restoperatorn: `%`. Med den kan vi ta reda på vilken rest en viss division ger. Till exempel står `10%3` för resten som 10 ger vid division med 3, och den resten blir ju 1. Om en division går jämnt upp så lämnas resten 0, det vill säga vi kan kontrollera om en division går jämnt upp genom att skriva `if(a%b==0)sats`. Satsen `sats` kommer endast att utföras om värdet som finns i `a` är jämnt delbart med värdet som finns i `b`. Vi studerar ett till exempel ur boken som utröner om 156 är jämnt delbart med 12: (Exempel 12.)

```
if (156%12==0)
  printf("Ja det gör det");
else
  printf("Nej, tyvärr");
```

Man kan alltså säga att denna rutin ger svar på frågan "Går 156 att dela jämnt med 12?"

4.8 Slumptal

Funktionen `rand()` (förkortning för *random* = *eng.* för *slumpmässig*) skapar ett slumptal mellan 0 och `RAND_MAX-1`. I en del system är det största möjliga slumptalet `32767-1`, och man kan genom att skriva ut konstanten `RAND_MAX` för att avgöra vad som är fallet. I Code Blocks är den `32767` och största möjliga slumptalet blir därför `32766`. För att skapa slumptal måste datorn dock först initiera slumpgeneratorn, (det vill säga skaka tärningen lite innan den kastas), det gör man genom satsen `srand(time(0)*time(0));` Exakt vad den gör väntar vi med att förklara. Om man inte gör `srand(time(0));` så blir slumptalen bara samma hela tiden och det hänger på att en dator egentligen inte kan producera slumptal, de tal som kommer ur funktionen `rand()` är bara *simulerade* slumptal och man kan (med lite kunskaper i matematisk statistik) framföra kritik mot resultatet som kommer ur `rand()`, men det ska vi inte göra här. Vi nöjer oss med det som den levererar, det fungerar ju i alla fall ganska bra.

Vi ser nu på hur dessa funktioner ska användas tillsammans, ett program som levererar slumptal från 0 till 99 så länge man trycker ned en knapp:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <time.h>
int main(void){

  srand(time(0)*time(0));

  do
  {
    printf("%d\n", rand()%100);
    getch();
  }while(1);
}
```

När vi kör det kan det se ut så här:

```
18
77
71
44
43
74
28
98
85
11
```

Ett nytt värde kommer upp varje gång man trycker på en tangent. Tyvärr går detta program inte att stoppa på något annat sätt än genom att stänga konsolen som det kör i. Programmet finns inte i boken, det är ett fullständigt program som ni kan skriva in och prova. Gör gärna ändringar i det och se vad som händer!

4.9 Exponent

I biblioteket `math.h` (skriv alltså i början av programmet: `#include <math.h>`) finns en funktion som heter `exp()`, det är "*e* upphöjt till *x*" och man anropar den så här: `y = exp(x);`. Dess invers finns också och den heter `log(x)` (och det är *ln* av *x*), det innebär att man kan skriva

```
c = exp(b*log(a));
```

om man vill beräkna `a` upphöjt till `b` för godtyckliga flyttal `a` och `b`.

4.10 Siffersumman: Läs själva