Föreläsning 2

2.1 Variabler

Vi studerar exempel 1 från boken:

```
#include <stdio.h>
#include <conio.h>
int main(void)
{
  int a,b,c=7;
  float antal,pris=3.70,laengd;

  a=34; b=32767;
  antal=-123.978;
  pris=89.00;
  laengd=56;
  c=-38;
  laengd=laengd+1;
  getch();
}
```

Sex variabler, tre heltalsvariabler `a`, `b` och `c` och tre flyttalsvariabler, `antal`, `pris` och `laengd`. Med heltalsvariabler (`int`) kan man lagra heltal mellan -2 miljarder upp till +2 miljarder. Med flyttalsvariabler (`float`) kan man lagra decimaltal som 2.5, -5.678 osv. Skillnaden är att med flyttal får vi aldrig exakt precision.

Vi ser olika sorters satser, först två tilldelningar på samma rad, sedan fyra tilldelningar till på de fyra följande raderna och på näst sista raden ser vi en beräkning där värdet på `laengd` ökas med 1. Allra sist så inväntar programmet en tangenttryckning.

Frågor och kommentarer kring programmet?

2.2 Regler för namngivning i *C*

Man ger olika namn till stöd för minnet som programmerare, man namnger variabler, funktioner och konstanter och namnen kallas identifierare. I exemplet ovan har vi flera identifierare: `main`, `a`, `b`, `c`, `antal`, `pris`, `laengd` och `getch`. Identifierares namn måste uppfylla följande regler:

* Engelska (stora eller små) bokstäver får användas (inte å, ä, ö).

* Siffror 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 få användas.

* Underscore _ får användas

* Ett namn får inte inledas med en siffra.

Exempel på lagliga identifierare: `ANTAL`, `a`, `b`, `max`, `hela_summan`, `a1`, `a2`, `plats4`

Exempel på icke-lagliga identifierare: `4_platsen`, `1_a_summan`

Olika konventioner som man brukar använda för namngivning är: funktioner och variabler namnges med små bokstäver. Konstanter namnges med genomgående STORA bokstäver. (Det kommer mer om vad konstanter och funktioner är senare.)

2.3 `if`-satsen

C-ordet "`if`" används för att skapa villkorsmässiga satser, satser som kanske utförs beroende på om något är sant eller inte. Vi ser på ett exempel ur boken:

```
#include <stdio.h>
#include <conio.h>
int main(void)
{
  int a=7,b=10;

  if (a==7) printf("1 SANT\n");
  if (a>3) printf("2 SANT\n");
  if (b>6 && b<10)
    printf("3 SANT\n");

  getch();
}
```

Här deklareras först variablerna `a` och `b`, de får värden direkt i sina deklarationer (det kallas *initiering*.) Sedan kommer en `if`-sats där vi frågar om `a` har värdet 7 (vilket `a` har). Då utförs `printf`-satsen. Eftersom `a` har värdet 7 så kommer `printf`-satsen att utföras, det vill säga `"1 SANT"` kommer att skrivas ut (följt av radbyte.) Om `a` inte hade varit 7 så hade inte `printf`-satsen utförts och `"1 SANT"` hade inte skrivits ut.

På samma sätt så kommer `"2 SANT"` att skrivas ut eftersom `a` har värdet 7 som ju är större än 3 och det är det villkoret som den andra `if`-satsen testar.

Villkoret i den tredje `if`-satsen är lite mer komplicerat, det ska utläsas "om `b` är större än 6 OCH `b` är mindre än 10". Det två och-tecknen i koden (`&&`) står för OCH. Nu är det så att `b` har värdet 10 som ju är större än 6, så det är ju sant, men det *totala villkoret* i den tredje `if`-satsen kräver också att `b` ska vara mindre än 10 vilket *inte* är sant. Villkoret i den sista `if`-satsen är alltså inte uppfyllt vilket innebär att den sista `printf`-satsen inte kommer att utföras.

Sammantaget kommer det ovanstående programmet alltså att skriva ut detta:

```
1 SANT
2 SANT
```

Studera exempel 3 själva. Det viktiga är att man kan inte jämföra flyttal hur som helst!

2.4 Relationsoperatorerna

Man bygger `if`-satser som ovan med någon form av jämförelse, i *C* finns ett par jämförelseoperatorer, likhet (har vi sett ovan): ==, Större än (har vi sett ovan): >, mindre än (har vi också sett ovan): <, Större än eller lika med: >=, Mindre än eller lika med: <=, Ej lika med: `!=`.

Faktiskt så behandlas ett villkor i *C* som om det har ett värde, värdet är av typen heltal och är 1 då det är sant och 0 om det är falskt. Vi skulle alltså kunna skriva satsen

```
printf("%d", (22!=33));
```

och få utskriften `1`, för det är ju sant att 22 inte är lika med 33.

2.5 De logiska operatorerna

Vi har sett lite av logiska operatorer ovan, `&&` står för OCH. Vi har också `||` som står för ELLER och `!` som negerar ett logiskt uttryck. Vi tar ett par exempel:

```
if ( 1==2 || 10==10 ) printf("SANT");
```

Denna sats kommer att skriva ut `SANT` eftersom visserligen är inte 1 lika med 2, men 10 är ju lika med 10.

```
if ( 1==2 && 10==10 ) printf("SANT");
```

Denna sats kommer inte att skriva ut `SANT` eftersom för att villkoret i `if`-satsen ska vara sant så måste BÅDE 1 vara lika med 2 (vilket det inte är) och 10 vara lika med 10.

```
if ( !(1==2) && 10==10 ) printf("SANT");
```

Denna sats kommer att skriva ut `SANT` eftersom båda villkoren `!(1==2)` och `10==10` är sanna, negationsoperatorn, `!`, tar det falska villkoret `(1==2)` och vänder på det så att det blir sant. Båda villkoren blir sanna och `printf`-satsen utförs. Detta är *Boolesk Algebra* som ni kanske känner igen från digitaltekniken!

2.6 Tilldelningssatsen

Vi tittar på ett exempel ur boken:

```
#include <stdio.h>

int main(void){

  int a=3,b=.3,c;
  float pi=3.141593,e,r=5.0;

  a=a+1; a++;
  b=a+b;
  b=b+a;
  c=3+4*2;
  c=(3+4)*2;
  e=r*r*pi;
  b=24/10;
  e=24/10;
  b=24.0/10;
  e=24.0/10;
  e=24/10.0;
  c=1000*500;
  b=8/2+2;
  b=8/2/2;
  getch();
}
```

Här ser vi många många initieringar och tilldelningar! Vi går igenom dem sats för sats. (Gå igenom på tavlan. Diskutera begränsningar i lagringskapaciteten för `int` och `float`.)

2.7 Aritmetiska uttryck

Ett aritmetiskt uttryck uppkommer genom att man kombinerar saker som har värden med aritmetiska operatorer (bla de fyra räknesätten). Vid en tilldelning av en variabel behövs ett aritmetiskt uttryck och vi har sett sådana i alla exempel ovan som innehåller tilldelningssatser där en variabel tilldelats ett värde. (Peka.)

2.8 Aritmetiska operatorer

De fyra räknesätten har varsinn operator i *C*:

Multiplikation:	*
Division:	/
Addition:	+
Subtraktion:	-

Denna uppradning ovan anger också ordningsföljden i vilken de utförs då ett aritmetiskt uttrycks värde beräknas, multiplikation först, sedan division, sedan addition och sist subtraktion.

Om vi beräknar 12*10+1 så bli värdet 120 + 1 = 121 eftersom multiplikationen utförs först. Ordningen kan ändras genom använda parenteser, värdet av 12*(10+1) blir 12*11 = 132 och det är inte 121 som vi fick i det andra uttrycket.

2.9 Att programmera

Vad betyder det då att programmera? Jo, att programmera innebär att lösa problem, att finna sätt att uttrycka hur ett problem ska lösas. Det innebär att man måste skaffa sig en mycket precis uppfattning om det problem man ska lösa innan programmet kan fullbordas, men det finns ett tips som inte nämns så tydligt i boken och som jag verkligen vill trycka på och det är detta:

Man behöver inte veta hur lösningen till ett programmeringsproblem ser ut för att kunna börja arbeta med lösningen på problemet! Det går att programmera i delsteg och skriva allt bättre program och *i delsteg* lösa lösa ett stort och komplicerat problem. Och, det går *mycket* lättare om man hela tiden **arbetar med körbara program.** Detta är ett av de viktigaste tipsen som jag vill ge er: förvänta er inte att ni ska kunna ta ett problem och programmera upp det och kompilera och testköra och lyckas med bara en kompilering! Låt det ta många kompileringar och låt varje kompilering vara en liten dellösning av problemet ni vill lösa.

Exempel: Skriv ett program som beräknar ersättningsresistansen mellan två parallellkopplade motstånd. Resistanserna på de båda parallellkopplade motstånden ska användaren mata in.

Det första program jag skulle skriva för att lösa ovanstående uppgift skulle se ut så här:

```
#include <stdio.h>
#include <conio.h>
main()
{
  printf("R1: ");
  getch();
}
```

VA? Det gör ju ingenting, bara skriver ut `R1:` , det är väl ingen mening med det? Jo! Det finns en

mycket bra mening med det och det är att detta är ett enkelt **KÖRBART** program som löser en del av uppgiften. När det väl fungerar kan jag utvidga programmet steg för steg och få mer och mer funktioner. Nästa steg är att komma på: "Ah, jag vill ha inmatning, då behövs variabler och scanf". Ett nytt förbättrat program tar form som ser ut så här:

```
#include <stdio.h>
#include <conio.h>
main()
{
  float r1, r2;

  printf("R1: ");
  scanf("%d", &r1);
  printf("R1: ");
  scanf("%d", &r1);
  printf("Programmet klart.");

  getch();
}
```

Ett nytt program som ser ganska meningslöst ut? Nej, men vi har kommit en bit på vägen, nu har vi ordnat inmatning av två resistanser som vi lagrat i flyttalsvariablerna `r1` och `r2`.

Om vi kör programmet ser det ut så här:

```
R1: 45
R1: 67
Programmet klart.
```

Vi ser då att vi kanske skulle ändra i andra utskriften så att det står `R2` istället, vi ser också att andra inmatningen är till `r1`, det ska vara till `r2`. Med dessa två ändringar får vi körningen

```
R1: 45
R2: 67
Programmet klart.
```

I nästa steg lägger vi in en utskrift av ersättningsresistansen samt en till variabel, `r`, för att förvara värdet av ersättningsresistansen. Det nya programmet ser ut så här:

```
#include <stdio.h>
#include <conio.h>
main()
{
  float r1, r2, r = 10.0;

  printf("R1: ");
  scanf("%d", &r1);
  printf("R2: ");
  scanf("%d", &r2);
  printf("Ersattningresistansen: %f\n",r);
  printf("Programmet klart.");
  getch();
}
```

Och körningen ser ut så här:

```
R1: 10
R2: 20
Ersattningresistansen: 10.000000
Programmet klart.
```

Men vänta nu, var sker själva beräkningen av ersättningsresistansen? Ingenstans! Det här programmet kommer alltid att skriva ut `10.000000` vilka värden man än matar in! Ja, men det är en utvecklingsprocess, vi kommer till slutet i små delsteg. I det här läget tar man fram sina elektronikkunskaper och minns att ersättningsresistans för parallellkopplade resistorer är `r = r1*r2 / (r1 + r2);` Så det färdiga programmet ser ut så här:

```
#include <stdio.h>
#include <conio.h>
main()
{
  float r1, r2, r = 10.0;

  printf("R1: ");
  scanf("%d", &r1);
  printf("R2: ");
  scanf("%d", &r2);

  r = r1*r2 / (r1 + r2);

  printf("Ersattningresistansen: %.2f\n",r);
  printf("Programmet klart.");
  getch();
}
```

Och här har vi en testkörning:

```
R1: 10
R2: 20
Ersattningresistansen: 0.00
Programmet klart.
```

Vad nu då? `0.00`? Om man granskar koden så ser man att det står `scanf("%d", &r1);`! Inmatning av flyttalet `r1` med `%d`! Det ska ju vara `%f`! Och samma sak för flyttalet `r2`. Vi rättar felen bland de fyra raderna ovan till:

```
  printf("R1: ");
  scanf("%f", &r1);
  printf("R2: ");
  scanf("%f", &r2);
```

Vi testkör och får nu körningen:

```
R1: 10
R2: 20
Ersattningresistansen: 6.67
Programmet klart.
```

Och denna körning är korrekt. Observera att för att lösa detta enda enkla problem behövdes i storleksordningen en 4-5 program och kanske 6-7 kompileringar (Egentligen mer!). Fördelen är att utvecklingen går **steg för steg** och om man bara arbetar med **en** programrad åt gången i ett **körbart** program och programmet **inte** längre är körbart, ja då vet man på **vilken rad** man kanske måste ändra, eller så har man i alla fall en ganska bra ledtråd, det var på den raden man sist ändrade.

Arbeta alltid med körbara program! Ändra bara en rad åt gången mellan kompileringar!

2.10 Olika fel

Programmet nedan innehåller fem fel. Försök finna dessa:

```
#include <conio.h>
void main(void)
{
  int x;
  float y, sum;
  printf("mata in ett heltal: ");
  scanf("%d", x);
  printf("mata in ett flyttal: ");
  scanf("%d", &y);
  sum=x+y;
  printf("Summan är ", sum);
  if(x<0);
    printf("X är negativt");

  getch();
}
```

Fel 1: Vi saknar `#include <stdio.h>`.

Kompilatorn: `Call to undefined function 'printf'`

Fel 2: Vi har glömt `&` framför `x` vid inläsningen:

Kompilatorn: `Possible use of x before definition`

Fel 3: Det ska stå `%f` när vi läser in flyttal

Kompilatorn klagar inte. När programmet körs läses inget in, men programmet går att köra.

Fel 4: Det saknas `%f` i formatsträngen när vi ska skriva ut `sum`. Värdet på `sum` skrivs inte ut.

Fel 5: Vanligt nybörjarfel: `if`-satsen är en tom sats. Det vi har skrivit ovan är samma sak som

```
if(x<0); printf("X är negativt");
```

Båda satserna körs **alltid**, den tomma `if`-satsen, som inte har någon effekt, och `printf`-satsen, som inte hänger ihop med `if`-satsen alls, `if`-satsen är avslutad i och med semikolonet. Rätt kod är:

```
if(x<0)printf("X är negativt");
```

Ett semikolon hugger av (avslutar) en sats, så var försiktig med hur ni sätter ut semikolon.

De första två felen var syntaktiska. De andra tre var logiska fel. Logiska är mer lömska än syntaktiska eftersom kompilatorn inte tillåter oss att köra ett syntaktiskt felaktigt program.

Kapitel 3

3.1 Sammansatt sats

En sammansatt sats i *C* skapas med klamrarna, `{` och `}`, vi har sett dem i början och slutet av ett main. Man kan skriva så här:

```
{
  a = a + 10;
  b=2*a+20;
  printf("%d %d\n", a, b)
}
```

och allt detta tolkas av *C* som en sats. Vad ska man ha detta till? Jo det blir tydligt när vi studerar hur detta kan kombineras med if-satsen. Om vi skriver

```
if (c==0)
{
  a=3;
  b=4;
}
```

så styr `if`-satsen den efterföljande sammansatta satsen och om `c` är 0 så kommer båda satserna inom den efterföljande sammansatta satsen att utföras, det vill säga både `a` kommer tilldelas 3 och `b` kommer att tilldelas 4. Om vi inte hade haft en sammansatt sats och ändå hade velat uppnå att både `a` skulle får värdet 3 om `c` hade värdet 0 och `b` också skulle få värdet 4 om `c` var 0 här hade vi fått skriva något i stil med

```
if(c==0)a=3;
if(c==0)b=4;
```

för att uppnå samma sak. Detta blir både jobbigt att läsa och ineffektivt att köra eftersom `if`-satsen körs två gånger.

3.2 Mer om `if`-satsen

Vi har studerat `if`-satser som bara har en gren, om villkoret är sant så utförs den sats som `if`-satsen styr, om villkoret inte är sant så utförs ingenting. Man kan bygga ut en `if`-sats med en så kallad `else`-del som utförs om villkoret *inte* är sant. Vi ser på ett exempel:

```
if(a>10)
  a=a-2;
else
  a=a+2;
```

Vad sker här? Först görs en test på om `a` är större än 10. Om så är fallet så minskas `a` med 2. Men, om inte `a` är större än 10 så utförs `else`-delen, det vill säga `a` ökas med 2.

Man kan kapsla `if`-satser i varandra på komplicerade sätt, om man till exempel skriver

```
if(a==3)
  if(b==1)
    c=1;
  else
    c=2;
else
  c=3;
```

så har man en konstruktion som är ganska svår att förstå sig på. Det är då bättre att använda sammansatta satser. Det är lättare att förstå en kod som har det här utseendet:

```
if(a==3)
{
  if(b==1)
    c=1;
  else
    c=2;
}
else
  c=3;
```

Här har klamrarna ingen funktion förutom att de gör koden mer lättläst. Om vi låter en `if`-sats styra sammansatta satser så får en if-sats det mycket generella utseendet

```
if (villkor)
{
  sammansatt sats som kan utföra många saker
}
else
{
  annan sammansatt sats som kan utföra många andra saker
}
```

Jag rekommenderar att ni använder det här sättet att programmera, lägg in många klamrar och indentera (gör indrag) ordentligt så att det blir tydligt. Det här med indentering är till och med inte en rekommendation, det är ett absolut **KRAV** att ni indenterar ordentligt, det blir oläsliga program annars. Det g[r bra i b;rjan, men när era program blir större blir det väldigt svårt att programmera.

Vi kan observera att ett *C*-program självt kan ses som en stor sammansatt sats, `main()` inleds ju alltid med en klammer:

```
main()
{
  Satser
  ...
}
```

Indentering: (Följ dessa regler, annars blir det omöjligt att läsa era program):

Regel 1: Indraget ska vara konstant genom hela programmet.
Regel 2: På raden efter `{`: Gör ett indrag. Indraget ska fortsätta till den stängande klammern.
Regel 3: Den stängande klammern, `}`, ska upphäva indraget.
Regel 4: `if`-satser utan klamrar som delas upp på två rader ska ha andra raden indragen.