

1 Datorer och program

Idag är datorn ett av de viktigaste och vanligaste verktyget i vårt samhälle. På de flesta arbetsplatser finns det idag datorer. Vad använder man datorerna till? Vad gör en dator? Datorer kallades tidigare för datamaskiner. Den första delen av ordet datamaskin är data. Vad är data? *Data och information* hänger ihop enligt :

Ex: Data : 5, A, R, ?, >, ...
 Information : Hej, 222222-2222,

Som exemplet visar kan man sammansätta data till information. Data kan vara bärare av information. En dator har med information att göra och eftersom information har hög prioritet i dagens samhälle har datorn blivit ett viktigt verktyg.

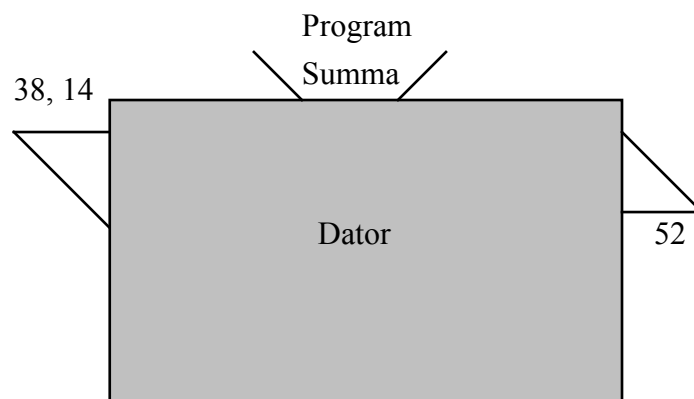
Vad gör en dator med informationen? Den andra delen av ordet datamaskin är maskin. När man hör ordet maskin tänker man på bearbetning och automatik. En symaskin bearbetar tyger och tråd så att man får kläder. *En dator eller datamaskin bearbetar information till ny information.*

En dator är en maskin som kan

- hämta in information
- bearbeta information
- skicka ut ny information

Hur datorn ska hämta in, bearbeta och skicka ut information bestäms av det *program* som datorn för tillfället är laddad med.

Ex: Programmet summa som tar in information i form av två tal och skickar ut ny information om summan av talen.



1.1 Program

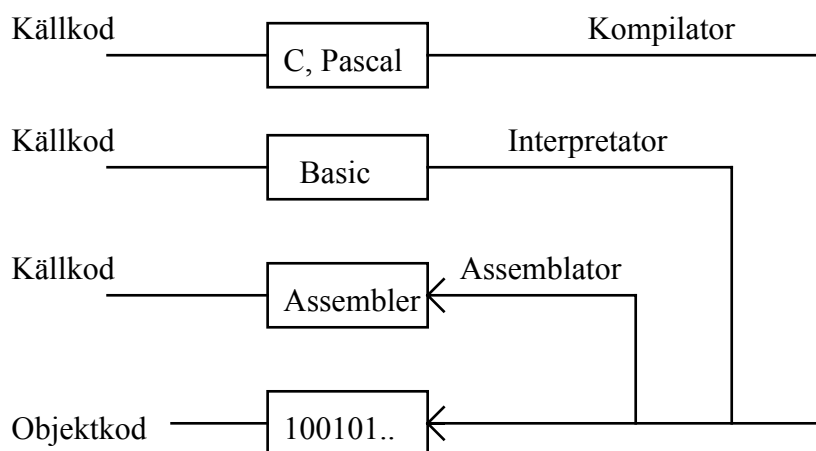
Programmet är datorns arbetsplan. *Steg för steg måste programmet ange vad som ska göras för att korrekt informationsbehandling ska fås.* Hårdvaran eller elektroniken i en dator är uppbyggd av digitala komponenter som bara kan anta två lägen, nämligen 0 eller 1.

Slutprodukten, av de data eller det program som datorn laddas med, får därför bara innehålla ettor och nollor. Man säger att programmets instruktioner är i binär- eller maskinkod. Varje datortyp har sin egen speciella maskinkod. I datorns barndom fick man skriva sina program direkt i den maskinkod, som den aktuella datorn förstod. Detta var mycket tidskrävande då det krävs många ettor och nollor för att skriva den enklaste information.

Ex: Talet 38, som matades in till datorn ovan, skrivs i binärkod som 100110

Bytte man datortyp fick man snällt skriva om hela programmet i nya kombinationer av ettor och nollor.

Idag finns det inbyggda program i datorerna som översätter från språk på högre nivå, med fler symboler som i Basic, C och Pascal, till en slutprodukt i aktuell maskinkod. Det finns tre olika huvudtyper av översättningsprogram interpretatorer, kompilatorer och assembleratorer.



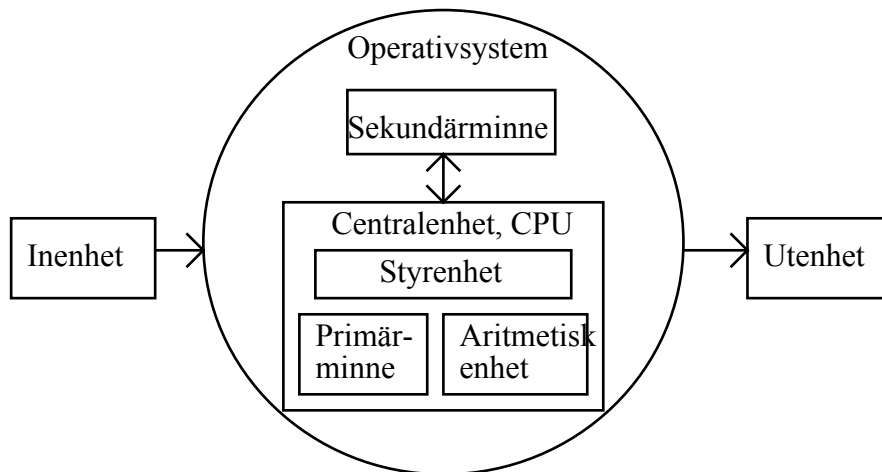
Kompilatorn är ett översättningsprogram som översätter hela den skrivna programtexten (källkoden) till maskinkod (objektkod). Vid körningen används den översatta objektkoden.

Interpretatorn översätter programtexten radvis till maskinkod. Vid körning används källkoden och översättning till objektkod måste göras rad för rad. Körningen blir därför långsammare än då en kompilator används.

Assemblern översätter lågnivåspråket assembler till maskinkod. Assembler är ett språk som enkelt och direkt kan översättas till ettor och nollor. Med en omvänd assembler kan man översätta åt andra hållet från maskinkod till assembler. Det finns inga motsvarande omvända kompilatorer eller interpretatorer som översätter från maskinkod till högnivåspråk.

1.2 Datorns funktion

En dator består av hårdvara (maskinvara) och mjukvara (programvara). Den består av huvuddelarna inenhet, utenhet, centralenhet, sekundärminne och operativsystem.



I centralenheten, CPU:n, finns styrenhet, aritmetisk enhet och primärminne. Styrenheten har direkt kontakt med alla enheter. Den styr och fördelar arbetet inne i centralenheten.

Den aritmetiska enheten utför beräkningar och gör jämförelser. I primärminnet lagras program och data under körning.

Sekundärminnet används för lagring av program och data mellan körningarna.

In- och utenhet är idag oftast en terminal med tangentbord och skärm.

Då man ska använda en dator och köra ett program måste man hämta programmet från ett utrymme (fil) i sekundärminnet och placera det i primärminnet. När programmet körs måste körningen övervakas och eventuella in- och utmatningar registreras etc.

I datorns barndom sköttes allt detta av en operatör. Det tog lång tid och var mycket besvärligt att köra ett program. Med tiden har det utvecklats speciella serviceprogram, *operativsystem*, som ersatt operatörerna. Det finns många olika operativsystem bland annat:

DOS
Unix
WindowsNT

Man använder dessutom hjälpsystem, *Windows*, som kan ses som ett skal ovanför operativsystemet, för att få ett *användarvänligare gränssnitt* mot datorn. I WindowsNT är dessa båda, operativsystem och Windows, sammanfogade till ett fullständigt och användarvänligt operativsystem.

1.3 Minnen

Minnet i centralenheten kallas primärminne. Det består oftast av två delar:

- 1) ROM-del (Read Only Memory)
- 2) RWM-del (Read Write Memory)

ROM-delen är programmerad av tillverkaren med exempelvis operativsystemet eller delar av detta. ROM-delen kan användaren endast läsa. RWM-delen är den del som användarens program och data laddas in i under körning. I stället för RWM används ofta beteckningen RAM (Random Access Memory). I ett RAM är åtkomsttiden oberoende av var i minnet data är lagrat. Detta gäller förvisso även ROM. Primärminnet är oftast av halvledartyp (chips). Härmed blir minnet snabbt men relativt dyrt per lagrat tecken. RWM-delen töms då datorn slås av.

För att lagra data och program då datorn är avslagen används sekundärminnet, som oftast är av magnetisk typ. Detta är relativt långsamt men billigt.

Ett minnes kapacitet mäts i Byte (B). 1B består av 8 bitar i form av ettor eller nollor enligt exempelvis :

01000001

Informationen som sparas i minnet är kodad i någon form av kod bestående av ettor och nollor. Exempelvis kan de 8 bitarna ovan vara koden för ett tal eller koden för en bokstav enligt :

Kod	Tal	Bokstav
01000001	65	A

För större datamängder som texter, ljud och bilder behövs naturligtvis flera Byte. Man brukar säga att en A4-sida skriven text innehåller ungefär 1500 tecken alltså 1500B. För större minnesutrymmen används enheterna kilobyte, megabyte resp gigabyte enligt :

$$1\text{KB} = 2^{10}\text{B} = 1024\text{B}$$

$$1\text{MB} = 2^{10} \cdot 2^{10}\text{B}$$

$$1\text{GB} = 2^{10} \cdot 2^{10} \cdot 2^{10}\text{B}$$

En vanlig diskett kan innehålla 1.44 MB alltså ca 1000 A4-sidor text.

1.4 Filer

När man stänger av datorn ska man spara sin information på sekundärminnet (hårddisk eller diskett), annars försvinner den. För att hitta informationen nästa gång man ska använda den, måste man på något sätt ge den en identitet. Man delar upp sekundärminnet i *filer* och sparar informationen på en sådan. Varje fil ger man ett namn. Hur namnet får se ut bestäms av det aktuella operativsystemet.

Ex: I WindowsNT anges namnet på en fil av *filnamn.filty*p enligt:

```
personregister.c  
personregister.cpp  
persondata.dat  
persondata.doc  
adressbok.txt
```

Filnamnet ska tydligt ange vad filen innehåller för information. Filtypen, som skrivs efter punkten, anger vilken typ av information som finns i filen. Filtypen c anger att filen innehåller källkoden till ett program skrivet i språket C och filtypen cpp anger att källkoden är skriven i språket C++. Filtyperna doc och txt brukar användas för textfiler som exempelvis skapats med ordbehandlingsprogram och som kan visas på skärmen eller skrivas ut på printer. Filtypen dat markerar att filen innehåller information i binärkod och därför exempelvis ej kan skrivas ut med printer.

Det finns ingen direkt begränsning vad gäller antalet tecken eller vilka tecken som ska användas i WindowsNT. Ska man däremot utnyttja sina filer i operativsystemet DOS måste man hålla sig till *maximalt 8 tecken före punkten* och *maximalt 3 tecken efter punkten*.

Ex: I DOS anges namnet på en fil av *filnamn.filty*p där filnamn får vara *maximalt 8* och filtyp *maximalt 3* tecken enligt:

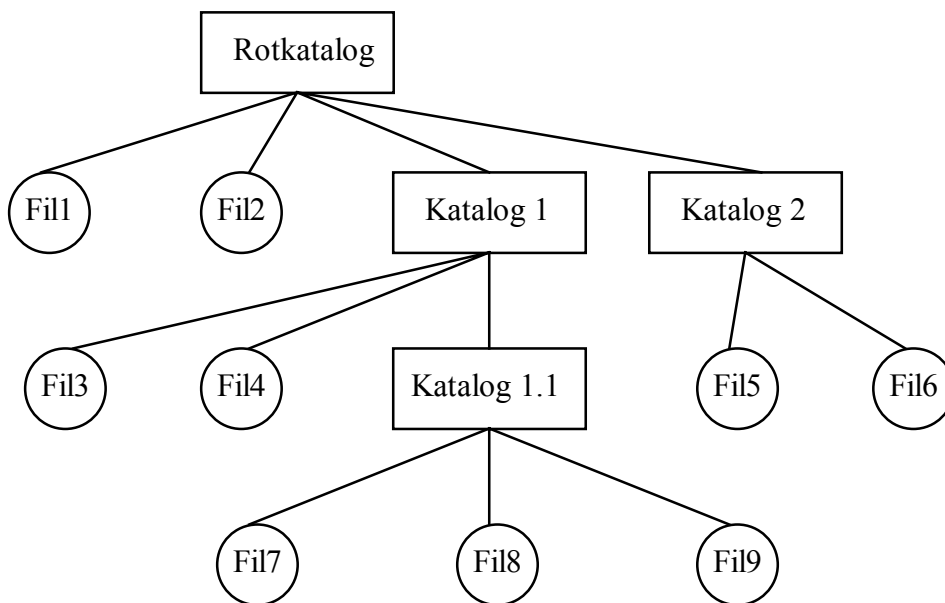
```
persreg.c  
persreg.cpp  
persdata.dat  
persdata.doc  
adrbok.txt
```

Sparar man information på samma fil en gång till försvinner den gamla versionen. Vissa editorprogram brukar dock döpa om den gamla filen till en fil med samma namn men med filtypen bak istället.

Filer kan man skapa på olika sätt. Man kan använda ett editorprogram eller ett ordbehandlingsprogram med vars hjälp man kan skriva sina egna filer. Man kan skriva egna program som skapar filer. Man kan också köpa filer på diskett eller CD etc.

Har man många filer kan det vara svårt att hålla ordning på dessa och svårt att hitta en speciell fil, som man vill titta på. Det blir långa söktider om man har alla filer i en enda hög. Jämför med en pärm som man har en massa papper i utan någon ordning. Det är inte alldeles lätt att hitta det papper man letar efter, även om pappren är namngivna.

För att snabba upp sökningen av filer brukar man dela in sina filer i kataloger. Man har en huvudkatalog eller rotkatalog (pärm) som är indelad i ett antal underkataloger (flikar) som i sin tur är indelade i underkataloger o.s.v. Man får ett filträd där det blir lättare att hitta enligt:



Med hjälp av operativsystemet kan man exempelvis:

- skapa underkataloger
- flytta filer mellan kataloger
- kopiera filer
- titta vilka filer som finns i resp katalog
- flytta sig till en viss katalog

Rotkatalogen betecknas på olika sätt i olika operativsystem. Disponerar man flera enheter sekundärminne exempelvis diskettstationer, hårddiskar etc kan man naturligtvis ha flera filträd (pärmar) med olika beteckningar.

1.5 Programutveckling

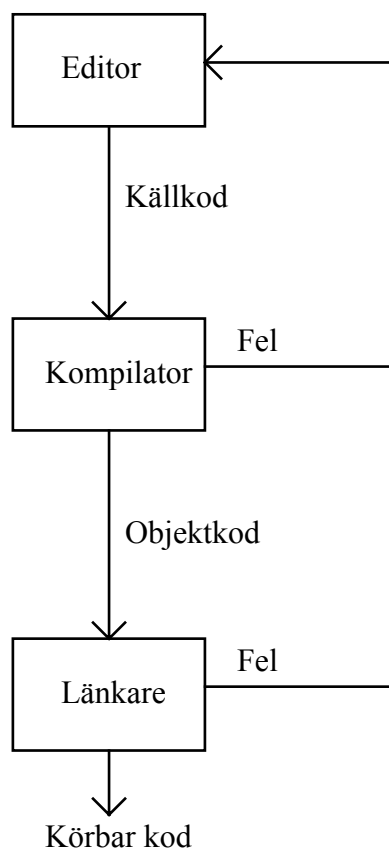
När man ska utveckla egna program i något språk måste man ha verktyg i form av *editor*, *kompilator* och *länkare*.

Med *editorn*, som är ett ordbehandlingsprogram, skriver man sin källkod. När man skrivit färdigt går man ur editorn och sparar sitt program eller sin källkod i en fil.

Nästa steg i programutvecklingsprocessen är att man använder *kompilatorn* för att kompilera sitt program. Är källkoden felaktigt skriven får man ändra den med hjälp av editorn och sedan kompilera om tills källkoden är felfri. Går kompileringen igenom utan fel har man fått sin källkod översatt till maskin- eller objektкод.

Maskinkoden är ej direkt körbar utan man måste med *länkaren* länka ihop den med vissa standardrutiner för exempelvis in- och utmatning etc. Går länkningen bra och utan fel har man fått en körbar eller exekverbar kod.

Gången vid programutveckling är:



Ex: Man vill skriva ett C-program som på skärmen skriver ut texten Hej!

Man börjar med att skriva källkoden med hjälp av en editor enligt :

```
#include <stdio.h>

void main()
{
    printf("Hej!");
}
```

Sedan sparar man med hjälp av ett editorkommando ovanstående kod i en fil exempelvis *hej.c*.

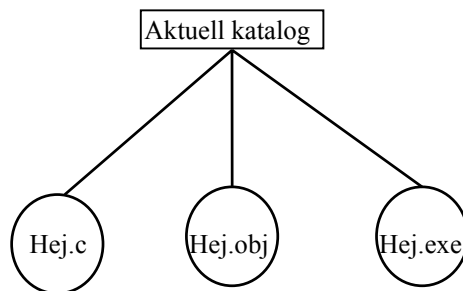
Filen *hej.c* ska man nu kompilera. Kompilatorn är idag ofta inbyggd i samma interaktiva miljö som editorn, vilket innebär att man kan kompilera programmet med ett kommando i denna interaktiva miljö. Efter kompileringen får man i sin aktuella katalog filen *hej.obj*, som innehåller objekt-koden (maskinkoden) för programmet *hej* i binärt format.

När kompileringen går felfritt är det dags att länka ihop programmet med eventuella extra program som behövs. I ovanstående fall så behövs utskriftsfunktionen *printf*, som länkaren då länkar in tillsammans med den kompilerade filen *hej.obj*. Går länkningen bra får man en exekverbar (körbar) fil i sin katalog som heter *hej.exe*. Länkaren finns också idag i samma miljö som editorn och kompilatorn.

Nu är det dags att exekvera (köra) programmet som skapats ovan. Detta kan man också göra från samma interaktiva miljö som man editerat, kompilerat och länkat i. Man väljer kör-kommandot och programmet körs. På skärmen skrivs :

Hej!

Tittar man i katalogen som man arbetat i ser man följande filer:



Man kan skapa olika exe-filer beroende på under vilket operativsystem de ska köras. Man kan exempelvis skapa en exe-fil för DOS och en för WindowsNT. I den interaktiva miljön som man arbetar i brukar man kunna ange vilken typ av plattform man ska skapa program för.

1.6 Programspråket C

Språket C utvecklades då man fick ett behov av ett språk på högre eller mer generell nivå än assembler. På Bell-laboratoriet i USA utvecklade man på 70-talet operativsystemet Unix. Man skrev det i assembler vilket innebar att man måste skriva nya versioner av operativsystemet för varje ny typ av datorsystem. För att höja nivån på sitt språk utvecklade man först språket B, som ganska snart gjordes om och fick då namnet C.

Språket C utvecklades med följande målsättningar:

- möjligheten att programmera på låg nivå skulle finnas kvar
- standardiserade anrop av systemrutiner som tidshantering mm
- vara ett generellt högnivåspråk av samma typ som Pascal, Fortran etc

Möjligheten att kunna programmera på låg nivå var en viktig målsättning eftersom C skulle användas för realtidstillämpningar typ operativsystem etc. I många av de högnivåspråk som fanns då på 70-talet fick man lämna den aktuella högnivåmiljön och gå över till assembler-programmering, för den aktuella processorn, för att komma åt exempelvis enskilda bitar i register.

Ex: Lågnivåegenskaperna har man infört i form av funktioner eller operatorer på C-nivå. Normalt kan man i ett högnivåspråk bara komma åt hela Byte men i C kan man exempelvis testa om en bit är 1 eller man kan skifta bitarna enligt:

```
c1 = 5;           /* 00000101 */
c2 = c1 << 3;      /* 00101000 */
```

Man har lyckats bra med lågnivåmålsättningen. Från att tidigare varit tvungen att skriva halva koden i högnivåspråk och resten i assembler kan man idag med C skriva 90% på hög nivå. Detta sparar mycket tid då systemet ska skrivas om för en ny datortyp.

När det gäller standardiserade systemanrop har man infört ett gränssnitt med vars hjälp man kan komma åt dessa med samma C-namn oberoende av vilken dator man använder. Varje C-kompilator anropar den korrekta funktionen just för den datorn.

```
Ex: #include <stdlib.h>    /* Här finns bl.a slumpalsgenerering */
     #include <time.h>      /* Här finns tidsfunktioner */
     #include <stdio.h>     /* Här finns i/o-rutiner */
```

När det gäller målsättningen att vara ett generellt högnivåspråk finns exempelvis möjligheterna att utnyttja selektioner (val) och iterationer (upprepningar), som i alla andra högnivåspråk.

1.7.1 Programexempel

Ex: Skriv ett program som läser in radien för en cirkel och beräknar och skriver ut cirkelns area.

```
/* Cirkel.c v1.0          */
/* Gunnar Joki GJI       */
/* ANSI C                */
/* Areal av en cirkel     */

#include <stdio.h>        /* printf, scanf */

void main()
{
    const float pi = 3.14159;
    float radie, area;

    /* hämta in ett värde till radie */
    printf("Ge radie : ");
    scanf("%f", &radie);

    /* beräkna och skriv ut area */
    area = pi*radie*radie;
    printf("Areal = %f", area);
}
```

Vid körning av detta program kommer utskriften att bli:

```
Ge radie : 2
Areal = 12.566360
```

OBS! C-kompilatorer skiljer mellan *små* och *stora* bokstäver. Exempelvis betyder ej *Area* samma sak som *area*. De flesta kompilatorer kräver dessutom att fördefinierade ord (const, float ..) ska skrivas med små bokstäver. *Använd alltså små bokstäver!*

```
/* Cirkel.c v1.0 */
```

Är en kommentar som kan innehålla vilken text som helst. Kompilatorn bryr sig ej om innehållet.

OBS! Man bör skriva programmets *filnamn* i en kommentar eftersom detta ej behöver skrivas ut i C-programmet. Alla huvudprogram heter main.

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

Är ett direktiv till en förkompilator. Detta direktiv innebär att förkompilatorn hämtar standardinkluderingsfilen `stdio.h`, som måste inkluderas för att vi ska kunna använda in/ut-rutiner som `scanf` och `printf` på ett korrekt sätt.

OBS! Vid inkludering av egna filer skriver man istället

```
#include "myfil.h"
```

```
void main()
```

Varje körbar modul måste innehålla ett huvudprogram vars start markeras med `main`. Vid körning startar alltid programmet vid denna punkt. Egentligen är `main` en funktion som ej returnerar något värde, vilket markeras av `void` och saknar parametrar, vilket framgår av de tomma parenteserna efter `main()`.

```
{  
    ...;  
    ...;  
}
```

Block-parenteserna eller måsvingarna `{` och `}` markerar blockstart resp blockslut och visar i detta fall var huvudprogrammet `main` börjar och var det slutar. Programmet innehåller ett antal satser. Varje sats avslutas med ett semikolon `;`.

```
const float pi = 3.14159;
```

Ett minnesutrymme ges namnet `pi` och tilldelas det reella värdet 3.14159. Minnesutrymmets värde får ej ändras vilket anges med `const`.

OBS! Vi kunde ha deklarerat konstanten `pi` som ett makro istället enligt:

```
#define PI 3.14159
```

Detta makro expanderar sedan av förkompilatorn till det angivna värdet på alla ställen i programmet där `PI` står. Stora bokstäver brukar användas till makron.

```
float radie, area;
```

Två minnesutrymmen (variabler) definieras med namnen radie resp area. Ordet float markerar att variablerna kan tilldelas reella värden.

```
printf("Ge radie: ");
```

Utskriftsrutinen printf anropas och den skriver ut orden som finns mellan citationstecknen ("). Till funktionen printf skickar man information med hjälp av en parameter eller ett argument i form av strängen "Ge radie :".

```
scanf( "%f", &radie);
```

Inmatningsrutinen scanf anropas och den läser värdet som matas in från tangentbordet och tilldelar radie detta värde. Rutinen scanf har alltid en formatsträng som första parameter. Där ska man med en speciell kod ange hur inbufferten ska tolkas. Formatet %f innebär att man ska hämta ett flyttal (reellt tal) från tangentbordet. Den andra parametern &radie anger att flyttalet ska placeras i minnet i den adress som radie har. Ampersand (&) markerar adressen för en variabel.

```
area = pi * radie * radie;
```

Datorn räknar ut ett värde för produkten av talen i minnesutrymmena pi, radie och radie och tilldelar variabeln area (minnesutrymmet som har namnet area) detta värde med hjälp av tilldelningstecknet (=).

OBS! *Tilldelningstecknet är = och likhetstecknet är == i C.*

```
printf("Arean = %f", area);
```

Utskriftsrutinen printf anropas och den skriver först ut strängen "Arean = " följt av värdet av den reella variabeln area. Var och hur värdet av area skrivs ut markeras av %f som även kan innehålla information om antal positioner och antal decimaler ex %5.2f.

Ex: Modifiera programmet cirkel så att man kan mata in radien och få arean beräknad för ett godtyckligt antal cirklar. Matar vi in radien 0 ska programmet avslutas.

```
/* Cirklar.c v1.0          */
/* Gunnar Joki GJI        */
/* ANSI C                 */
/* Areal av cirklar        */

#include <stdio.h> /* printf, scanf */

void main()
{
    const float pi = 3.14159;
    float radie, area;

    /* läs in första värde till radie */
    printf("Ge radie (avsluta med 0) : ");
    scanf("%f", &radie);

    /* beräkna area och läs in nytt värde på radie */
    while ( radie != 0 )
    {
        /* beräkna och skriv ut area */
        area = pi*radie*radie;
        printf("Areal = %f\n", area);

        /* läs in nytt värde till radie */
        printf("Ge radie (avsluta med 0) : ");
        scanf("%f", &radie);
    }
}
```

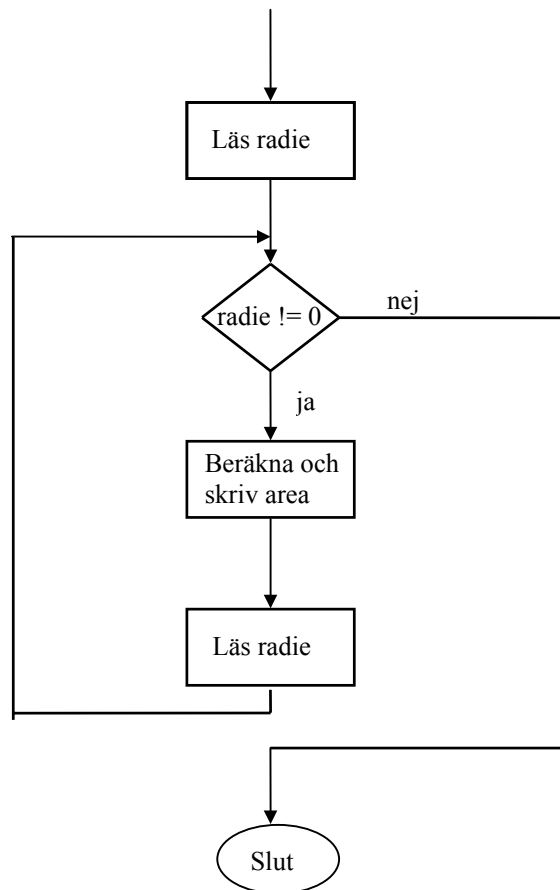
En körning av programmet kan se ut som:

```
Ge radie (avsluta med 0) : 1
Areal = 3.141590
Ge radie (avsluta med 0) : 2
Areal = 12.565360
Ge radie (avsluta med 0) : 0
```

```
while ( radie != 0 )
{
    ...
}
```

Så länge radie är skilt ifrån 0 upprepas alla satser i blocket mellan { och }.

Man kan rita ett flödesschema som visar programflödet, alltså i vilken ordning satserna kring while-loopen körs i ovanstående program enligt:



För att flytta fram till nästa rad på skärmen måste man skicka ett RETURN-tecken till den. Detta gör man själv vid buffrade inmatningar, som alltid avslutas med att man trycker RETURN. Vid utmatning måste man skriva ut RETURN-tecknet (`\n`) i printf-satsen enligt:

```
printf("Areal = %f\n", area);
```

Ex: Skriv ett program som skriver ut hur många rabattfrimärken som man ska använda sig av för olika tunga brev enligt tabellen:

Max vikt(g):	100	250	500	1000
Märken(st):	1	3	4	5

Är vikten större än 1000 g ska programmet skriva ut att brevet ska skickas som paket.

```
/* Porto.c v1.0 */
/* Gunnar Joki GJI */
/* ANSI C */
/* Rabattfrimärken för olika tunga brev */

#include <stdio.h>

void main()
{
    int vikt;

    /* läs in vikt */
    printf(" Ge vikt : ");
    scanf("%d", &vikt);

    /* välj rätt antal rabattfrimärken */
    if (vikt <= 100)
        printf(" Antal rabattfrimärken = 1\n");
    else if (vikt <= 250)
        printf(" Antal rabattfrimärken = 3\n");
    else if (vikt <= 500)
        printf(" Antal rabattfrimärken = 4\n");
    else if (vikt <= 1000)
        printf(" Antal rabattfrimärken = 5\n");
    else
        printf(" Skickas som paket!\n");
}
```

En körning av programmet kan se ut som:

Ge vikt : 200
Antal rabattfrimärken = 3

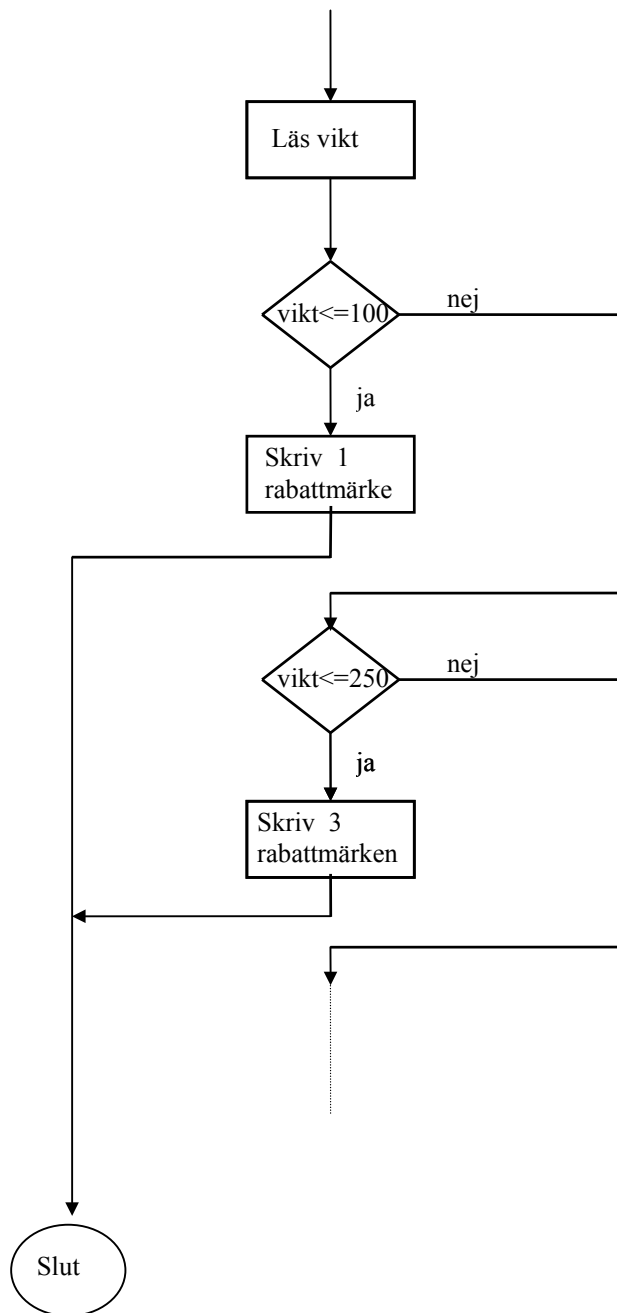
En annan körning av programmet kan se ut som:

Ge vikt : 600
Antal rabattfrimärken = 5

```
if ( vikt <= 100 )
    ...
else if ( vikt <= 250)
    ...
```

Här prövas villkoren för alternativen från början och det alternativ vars villkor först blir korrekt väljs och utförs. Endast ett alternativ utförs.

Ett flödesschema som visar hur satserna i detta program körs ser ut som:



I printf-satsen anger man formatet %d vilket innebär att vanliga heltal (basen 10) kan inmatas. I föregående program hade man %f när reella tal skulle inmatas.

Ex: Skriv ett program som skriver ut en trigonometrisk tabell för vinklar mellan 0 och 90 enligt:

VINKEL	SINUS	COSINUS	TANGENS
-----	-----	-----	-----
0	0.0000	1.0000	0.0000
10	0.1736	0.9848	0.1763
..			
90	1.0000	0.0000	*****

```

/* Trigtab.c v1.0                                     */
/* Gunnar Joki GJI                                     */
/* AANSI C                                             */
/* Trigonometrisk tabell 0 - 90 grader                */

#include <stdio.h>
#include <math.h>      /* sin, cos, tan, atan */

void main()
{
    const float pi = 4*atan(1);
    float vinkel;
    int nr;

    /* skriv tabellrubrik */
    printf(" VINKEL      SINUS      COSINUS      TANGENS\n");
    printf(" -----      -      -      -\n");

    /* beräkna och skriv tabellvärden */
    for ( nr = 0; nr <= 90; nr += 10)
    {
        printf("%5d",nr);
        vinkel = nr * pi/180;
        printf("%13.4f",sin(vinkel));
        printf("%13.4f",cos(vinkel));
        if (nr < 90)    // om nr mindre än eller lika med 90
            printf("%13.4f\n",tan(vinkel));
        else            // annars
            printf("      *****\n");
    }
}

```

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

Efter denna inkludering har man tillgång till de vanliga matematiska funktionerna som exempelvis :

```
printf("%13.4f", sin(vinkel));  
printf("%13.4f", cos(vinkel));
```

OBS! Vinkeln måste skickas i radianer för att få tillbaks korrekta värden.

OBS! Formateringen av utskriften med %13.4f, vilket innebär utskrift i 13 positioner högerjusterat med 4 decimaler. Positionerna fylls ut med blanka i början.

```
for ( nr = 0; nr <= 90; nr += 10 )  
{  
    ...  
}
```

Här ges först nr värdet 0. Sedan kontrolleras om villkoret $nr \leq 90$ är sant. Är detta sant körs satserna mellan { och }. Därefter ökas nr med 10 och ny kontroll sker. Är villkoret fortfarande sant körs satserna igen och det hela upprepas så länge villkoret är sant.

Denna for-loop är helt ekvivalent med följande while-loop:

```
nr = 0;  
while (nr <= 90)  
{  
    ...  
    nr += 10;  
}
```

Satsen

```
nr += 10;
```

är detsamma som

```
nr = nr + 10;
```

dvs öka det gamla nr med 10 och tilldela detta till det nya nr.