C på en kvart

Henrik Olsson

2004-10-01

Rev 0.96

Dokumenthistorik

0.92

Lade till lite saker som inte råkat komma med tidigare. Dokumentet var fram till denna version helt och hållet baserad på sånt som som jag själv stött på praktiskt.

Dessutom lades några exempel på standardfunktioner till. Vilket gjorde att dokumentet tog det stora steget från att enbart handla om c-syntaxen till att även handla om funktionsbibliotek.

(Första versionen som granskats av utomstående.)

0.93

Endast lite detaljer angående flyttalskonstantvärdet är ändrade.

(Gjorde en själv en rigorös lusläsning.)

0.94

Korrigerade ett otal fel som upptäckts av både mig och andra.

Även en hel del förbättringar vad det gäller språkbruk gjordes.

0.95

Ny licens, GFDL.

0.96

Fixade en hel rad dokumenttekniska problem. Åtgärdade lite inkonsekvenser i användningen av stilmallar. Byte ut många mjuka mellanslag mot hårda. Bytte ut många ”mjuka minustecken” mot hårda. Bytte ut många raka citatattecken mot typografiskt korrekta.

Flyttade sektionen om enum från avdelningen om förkompilatorn.

Av någon orsak har tydligen trebokstavskombinationerna ramlat bort i någon revision. Dessa återinfördes därför. "Trigraferernas återkomst".

Förord.

Titeln på detta dokument är kanske lite överdriven. I och för sig så kan man kanske läsa dokumentet på en kvart, förutsatt att man hoppar över sånt trams som förord. Men vanliga dödliga kommer knappast kunna skriva C-program efteråt.

Dokumentet är ej tänkt som någon lärobok i C. Ambitionen har varit att skriva ett superkort dokument, där man kan kontrollera att man förstått saker och ting rätt, och framförallt se vad man inte förstått, så att man kan leta upp och läsa om det någon annan stans, eller genom att fråga någon.

Dokumentet kan vara användbart för de som är rutinerade i andra programmeringsspråk och som håller på att konvertera till C, eller för de som redan lärt sig grunderna i C genom någon lärobok eller C-kurs.

Då C finns i många dialekter vill jag nämna att detta dokument med flit inte helt respekterar ANSI C-dialekten. Vissa saker i ANSI C-standarden nämns bara som hastigast. Och andra saker jag tar upp hör inte hemma där alls. Jag har istället försökt beskriva en lite mer ”praktisk C”, som jag tror är mer utbredd. Detta med motiveringen att de flesta som programmerar i C gör det i gamla kompilatorer och/eller för inbäddade system, som ligger ”lite efter” i standardiseringsutvecklingen. På moderna (riktiga) system finns moderna (standardiserade) kompilatorer för C++ att tillgå, och då rekommenderar jag att man använder C++ istället.

De saker jag valt att inte ta upp är hur kompilatorer och länkare fungerar (Exakt hur kompilatorer och länkare fungerar är omöjligt att gå igenom, eftersom det inte finns någon standard för hur de fungerar. Nu menar jag hur man använder dem. Inte hur de kompilerar koden.)

Detta dokuments officiella webbplats är http://brow.se/c/

Många svar på vanliga frågor om C (på engelska)
http://www.eskimo.com/~scs/C-faq/top.html

En wiki om C (Där finns även en wiki-bok om ANSI C)
http://sv.wikipedia.org/wiki/C_%28programspr%E5k%29

En massa länkar om C (på engelska)
http://www.lysator.liu.se/c/

Projekt, filer, kompilering

C är ett kompilerande språk. Det innebär att C-utvecklingsmiljön (kompilatorn) skapar riktiga programfiler, som står för sig själva. (De är alltså inte beroende av C-utvecklingsmiljön sedan.)

Ett C-projekt består av flera filer. Så för att hålla samman projektet brukar det finnas en övergripande fil som håller samman projektet. Denna kan heta **Makefile**, **make**, ***.mak** , ***.prj** eller något liknande. I denna fil står det vilka andra filer som ingår i ett projekt, och hur de är beroende av varandra.

Själva koden brukar ligga i filer som heter ***.c** .

Deklarationer av funktioner och konstanter brukar ligga i filer som heter ***.h** . (Dessa filer kan man låta förkompilatorn ”klistra in” i ***.c**-filerna .)

Det första som händer när man kompilerar ett projekt är att projektfilen gås igenom för att se vilka filer som ingår, och genom att kolla filerna klockslag se vilka som behöver kompileras om.

Det andra som händer är att det skapas en objektfil, ***.o** eller ***.obj**, för varje ***.c**-fil . En objektfil innehåller maskinkod (funktioner) och data (variabler). Dessutom innehåller den tabeller över vilka funktioner och variabler den själv innehåller, samt över vilka funktioner och variabler den är beroende av utifrån.

Till sist länkas alla objektfiler ihop till en körbar programfil, **.exe**, eller ett ”körbart” funktionsbibliotek som andra program kan använda, ***.dll** eller ***lib*** .

Förkompileringen

Innan den ”riktiga” kompilatorn startar körs en förkompilator. Dess huvuduppgift är att känna igen vissa texter och ersätter dem med annan text. Vissa villkor kan utföras också.

#include klistrar in innehållet ur en annan fil. **#include "x"**söker i första hand i lokal katalog.
#include <x> söker i stället i andra förbestämda kataloger.

#define definierar makron (automatiska textersättningar) som används under förkompileringen. Första delen är den som ska bytas ut, och andra delen den som ska dit istället. I viss mån går det att med hjälp av **(** , **)** och , ange makroargument. Makroargument visar var text ska hämtas ur koden före utbytandet, och var den ska klistras in efter.

Ex:

```
#define PI 3.141592
#define PUCKAD_AREA(x) PI * x * x
#define CIRCLE_AREA(x) (PI * (x) * (x))
#define RECTANGLE_AREA(x, y) ((x) * (y))
```

Tänk på att alltid stoppa in tillräckligt med parenteser i makrodefinitionerna, när det gäller uttryck. Om man använder tex **CIRCLE_AREA** så här här **depth * CIRCLE_AREA(radius - thicknes)** då kommer förkompilatorn ersättas det med
depth * (3.141592 * (radius - thicknes) * (radius - thicknes)) vilket tack vare paranteserna kommer ge önskat resultat

Däremot om vi använder **PUCKAD_AREA** på samma sätt
depth * PUCKAD_AREA(radius - thicknes) så kommer förkompilatorn ersätta detta med
depth * 3.141592 * radius - thicknes * radius - thicknes vilket kanske inte var vad som önskades.

används vid makrodefinitioner för att ange att ett makroargument ska bli en strängkonstant (eller del av).

Ex.

```
#define NAME_OF_DATATYPE(datatype) #datatype
```

används vid makrodefintioner för sätta ihop makroargument.

Ex.

```
#define FULL_NAME(first, last) first ## " " ## last
```

#undef gör att ett makro kastas och glöms bort.

#if , **#ifdef** , **#ifndef** , **#else** , **#elif** , **#endif** kan användas för att bygga upp villkor som avgör om ett visst avsnitt med c-kod ska tas med eller gömmas för den riktiga kompilatorn.

#error ger en feltext under förkompilering.

#pragma är till för ickestandardutökningar, och bör därför naturligtvis undvikas.

#line används för att omnumrera radnummer. (Kan kanske vara intressant vid felsökning.)

__DATE__ ger en strängkonstant med det datum då förkompileringen sker. **__TIME__** ger en strängkonstant med det klockslag då förkompileringen sker. **__FILE__** ger en strängkonstant med det filnamn som förkompileras. **__LINE__** ger en heltalskonstant med det radnummer i källkoden som **_LINE_** står på. **__STDC__** är definierad till något skilt från **0** om förkompilatorn uppfyller standarden ANSI C.

**** i slutet på en rad gör att vissa nyare förkompilatorer flyttar upp nästa rad så att det blir som en enda lång rad.

Förr i tiden fanns det datorer som saknade många i C viktiga tecken. Därför finns vissa treteckenskombinationer som substitut. Många kompilatorer stödjer dock inte dessa. Så använd dem inte. Men man måste ju känna till dem ifall, så man inte sliter sitt hår den dagen man faktiskt vill skiva tex **??=** i en text, och så blir det bara ett **#** .

Treteckenskombination	Ger tecken
??(	**[**
??)	**]**
??<	**{**
??>	**}**
??/	****
??!	**|**
??'	**^**
??-	**~**
??=	**#**

(Resten av detta dokumentet behandlar det som den riktiga kompileringen sköter.)

Kodstruktur och lagringsklasser

I C kan man skriva lite hur som helst. Man kan i väldigt stor utsträckning själv välja om man vill skriva en enda lång rad eller om man vill bryta ner koden så att det står ett enda ord eller tecken på varje rad. Ett mellanting är naturligtvis ofta mest lättläst.

Istället för att vara uppbyggt av programrader (som många andra programspråk) är C uppbyggt av programsatser. Varje sats avslutas normalt med ett `;` . Varje sats kan delas in i en godtycklig mängd satser detta gör man då med tecknen `{` och `}` .

Grovt kan man säga att man bygger upp sin kod med hjälp av funktioner, datastrukturer och programstyrande satser, vilka alla är satser. Till sin hjälp har man konstanter, variabler, datatyper, pekare och operatorer, som är de mindre byggstenarna som utgör en sats.

För att skriva egna kommentarer i koden som kompilatorn struntar i använder man `/*` och `*/` .Sådana här kommentarer kan ligga i en rad eller vara utspridda över många rader.

(I många nyare kompilatorer kan man även använda `//` som gör resten av raden till en kommentar.)

I C finns olika lagringsklasser och vad man brukar ”scoping-regler” (områdesregler).

En variabel (eller egna typdefinitioner) gäller där du skapar den. Det du definierar (skapar) i början av en fil (ej inom `{ }`) gäller i resten av filen. Dessa definitioner kallas globala.

Om du inte vill skapa en ny global variabel, utan bara använda en befintlig global variabel som redan definierats i en annan fil skriver du `extern` före datatypen.

Om du vill skydda dina globala variabler från att användas av andra filer skriver du `static` före datatypen.

Det du definierar inom `{ }` gäller däremot bara till `}` . Dessa definitioner kallas lokala.

(Vill man vara extra tydlig kan man skriva `auto` före datatypen, för att visa att man vill ha en helt vanlig lokal variabel. Men det behövs inte.)

Det brukar finns några få ”VIP-platser” för lokala variabler (i de sk processorregistren). De lokala variabler som hamnar här hanteras mycket snabbare än andra. För att begära att kompilatorn ska försöka använda en sådan här ”VIP-plats” skriver du `register` före datatypen.

Om du vill att en lokal variabel ska förvaras på ett ”säkert ställe” där dess värde inte glöms bort skriver du `static` före datatypen.

Vid lågnivåprogrammering kan det vara intressant att varna kompilatorn för att värdet i en variabel kan ändras utifrån (av hårdvaran eller via sk interupt). Detta gör man genom att skriva `volatile` före datatypen.

I vissa nyare kompilatorer finns `const` som ser till att värdet i en variabel inte kan ändras. Detta ger då ”riktiga konstanter” till skillnad från `#define` som ju bara säger åt förkompilatorn att byter ut text i koden.

Grunddatatyper

Det finns några få grunddatatyper. **`int`** för heltal, **`char`** för tecken, **`float`** och **`double`** för flyttal.

Beroende på vilket system du använder har dessa olika räckvidd, exempelvis kan en **`int`** på ett visst system sträcka sig fr.o.m -32768 t.o.m 32767. Om man inte behöver så stor räckvidd på sina heltal, och vill snåla på datorresurserna, kan man skriva **`short int`**, eller bara **`short`**. Om man tvärtom behöver en större räckvidd skriver man **`long int`** eller bara **`long`** .

Även om **`char`** är avsett för att hantera tecken, lagras det ändå som ett heltal, vanligtvis med den ringa räckvidden fr.o.m -128 t.o.m 127.

Normalt går halva räckvidden bort till att kunna hantera de negativa talen. För heltal finns dock en möjlighet att utnyttja hela räckvidden till de positiva talen. Detta gör man med **`unsigned`**. På ett visst system kan då exempelvis **`unsigned int`** ge en räckvidd fr.o.m 0 t.o.m 65535, och en **`unsigned char`** brukar ha räckvidden fr.o.m 0 t.o.m 255. **`signed`**, finns också men är underförstått. Om **`unsigned`** eller **`signed`** skrivs utan grundtyp underförstås grundtypen **`int`** .

(När man räknar med heltal i C och kommer utanför räckvidden slår det bara runt.)

Flyttal antyder att de kan hantera tal helt flytande (ej i fasta steg som heltalen). Så är inte fallet. Även flyttalen hanteras i fasta steg om än mycket finkornigare (tätare gradering) än heltalen. **`float`** är den flyttalstyp som har minst räckvidd (tex fr.o.m -10^{38} t.o.m 10^{38}), och den har även sämst precision. Den oftare använda **`double`** har mycket bättre både räckvidd och precision. **`double`** kan dock utökas till att få ännu bättre räckvidd och precision genom att skriva **`long double`**. Ju mer räckvidd och precision i datatypen, ju mer datorresurser går det åt.

Det finns egentligen en grundtyp till och det är **`void`** . Den används för att ange att något är tomt på innehåll eller att innehållets typ är okänd. (Vad detta skall vara bra för kommer att tas upp efter hand.)

Någon särskild typ för att hantera booleska uttryck, som det finns i många andra språk, finns egentligen inte i C. Däremot är ofta konstanter för detta definierade i någon av funktionsbibliotekens *.h-filer. Ungefär så här:

```
typedef int bool;
#define TRUE 1;
#define FALSE 0;
```

Konstantvärden

Heltal kan anges med tre olika talbaser. Decimalt (vanliga tal), hexadecimalt och oktalt. (Binärt skrivsätt saknas.) Om man uttyckligen vill ha en konstant av en viss typ (räckvidd) kan man använda suffixen L och U.

`10`	(int)	Ett vanligt heltal.
`-10L`	(long)	Ett vanligt heltal, fast hanteras som en större typ.
`3600U`	(unsigned int)	Ett vanligt heltal, fast hanteras som en typ utan möjlighet att hålla negativa värden.
`86400UL`	(unsigned long)	Ett vanligt heltal, fast hanteras som en större typ utan möjlighet att hålla negativa värden.
`010`	(int)	Om ett heltal inleds med en nolla tolkas den oktalt.
`0x10`	(int)	Hexadecimal tolkning.
`0xFFFFFFUL`	(unsigned long)	Hexadecimal tolkning, fast hanteras som en större typ utan möjlighet att hålla negativa värden.

Att ett tal ska tolkas som ett flyttal är ganska naturligt om det har en decimaldel, eller om det innehåller en exponentdel. I annat fall, eller om man vill använda en särskild typ (räckvidd och precision), kan man använda suffix.

`3.14`	(double)
`3.0`	(double)
`3.`	(double)
`.007`	(double)
`-1.80`	(double)
`-.2`	(double)
`7.2E-10`	(double)
`3.0F`	(float)
`3.F`	(float)
`.5F`	(float)
`3.0L`	(long double)

Teckenkonstanter (enstaka tecken) anger man inom två `'` . Strängkonstanter (flera sammanhängande tecken) anger man inom två `"` . (Det senare resulterar dock i något som inte är en grundtyp, utan i en pekare. Vad pekare är hur de används tas inte upp här, utan bara upp hur man gör konstanterna.)

Vissa tecken går ju inte att skriva in i koden tex Enter, Tab och Backspace. Vissa andra tecken kan inte användas på grund av att de har speciella betydelser i C, tex `'` och `"` . Därför använder man kontrolltecknet `\` , i kombination med andra tecken, för att i sträng- och teckenkonstanter ange de annars omöjliga tecknen.

Kombination	Ger tecken
`\a`	Bell (Ger ett litet pip, förutsatt att systemet kan hanterar ljud.)
`\b`	Backspace
`\f`	Formfeed
`\n`	New line
`\r`	Return
`\t`	Tab
`\v`	Vertikal tab
`\\`	`\`
`\'`	`'`
`\"`	`"`
`\?`	`?`

Alla tecken kan dessutom anges med hjälp av deras ASCII-koder. Om `\` följs av ett `x` betyder det att ett hexadecimalt värde fr.o.m 0 t.o.m FF följer. Om `\` följs av en siffra från 0 till 7 betyder det att ett oktalt värde fr.o.m 0 t.o.m 277 redan har inletts. Om denna syntax används mitt i en strängkonstant rekommenderas att fylla ut med nollor så det blir 4 tecken inklusive `\` . Tex `\x09` eller `\011` . Annars kan det lätt bli fel.

`'J'`	(char)	Tecknet J.
`'\n'`	(char)	Radbrytningstecknet
`'\t'`	(char)	Tecknet Tab (angivet hexadecimalt)
`'\x9'`	(char)	Tecknet Tab (angivet hexadecimalt)
`'\11'`	(char)	Tecknet Tab (angivet oktalt)
`'\0'`	(char)	Tecknet Nul (ASCII 0) (angivet oktalt)
`'\x4A'`	(char)	Tecknet `J` (angivet hexadecimalt)
`'\112'`	(char)	Tecknet `J` (angivet oktalt)
`"Qwerty"`	(char *)	En pekare till en sträng som innehåller texten `Qwerty` .
`"NE\112!"`	(char *)	En pekare till en sträng som innehåller texten `NEJ!` .
`"\x4AO!"`	(char *)	En pekare till en sträng som innehåller texten `JO!` .
`"A\tZ"`	(char *)	En pekare till en sträng som innehåller `A` , Tab och `Z` .
`"A\x9Z"`	(char *)	En pekare till en sträng som innehåller `A` , Tab och `Z` .
`"A\x9B"`	(char *)	En pekare till en sträng som innehåller `A` och ett buggigt tecken.
`"A\x09B"`	(char *)	En pekare till en sträng som däremot innehåller `A` , TAB och `B` .
`"C:\\DOS\\*.BAS"`	(char *)	Pekare till en sträng med innehållet `C:\DOS\*.BAS` . Observera att man måste skriva dubbla `\` fast man menar enkla.

`"Ange %s fukt:"`	(char *)	Pekare till en sträng med innehållet `Ange %s fukt:` . Observera att `%` INTE är ett specialtecken i C i sig.
`"LA" "8" "PV"`	(char *)	Pekare till EN sträng med innehållet `LA8PV` . Ja EN sammanslagen sträng. De olika strängdelarna skulle även kunna ha stått på var sin rad i koden.

Vanliga variabler, strukturer, unioner, typedef och enum.

En variabel är en lagringsplats i minnet som har ett namn, en datatyp och ett innehåll.

`int x;`		Deklaration av variabeln `x` . `x` blir här det symboliska namnet på ett heltal som skapas (reserveras plats för) någon stans i datorns minne.
`x`	(int)	`x` är (symboliserar) värdet på det heltal som vi kallar för `x` .
`x = 3 + 4;`		I det redan tidigare deklarerade `x` stoppas här resultatet av additionen av två konstanta värden. Detta görs här med operatorerna `+` och `=` . `+` utför additioner. `=` tar värdet från sin högra sida och stoppar det i variabeln på sin vänstra sida. (Denna kunskap om operatorerna räcker tillsvidare. Mer om operatorer senare.)
`long t;`		Deklaration av ytterligare ett heltal, fast av typen `long` .
`t = x + 1;`		Här tilldelar vi `t` värdet av additionen mellan variabeln `x` och konstanten `1` . Nu är det så att både `x` och `1` är av typen `int` så operatorn `+` har inga problem att addera dessa. Men när operatorn `=` ska hämta detta resultat och stoppa det variabeln `t` som är en `long` måste en konvertering ske. Enklare sådana här konverteringar sker automatiskt. Men vissa mer långsökta konverteringar måste begäras specifikt.
`float pi = 3.14F;`		Här deklareras att `pi` ska vara av datatypen `float` och samtidigt fylls den med det konstanta värdet `3.14F`.
`char ch = 'A';`		Här deklareras att `ch` ska vara en `char` som ska innehålla tecknet `A` .
`ch = ch + 2;`		Eftersom `ch` är en `char` och denna typ är ett heltal (om än med trång räckvidd) är det fullt möjligt att addera `2` till tecknet `ch` (som innehåller `A`). Då `ch` är en `char` och `2` är en `int` måste konvertering ske. Men även här sker detta automatiskt. (Resultatet kommer bli att `ch` innehåller tecknet `C` .)
`int x, y;`		Deklaration av variabeln `x` och variabeln `y` samtidigt. Detta är fullt tillåtet i C , och vanligt förekommande. Men det är lite otydligt och kan lät skapa luriga fel (,framförallt när pekare är inblandade). Därför rekommenderas INTE detta skrivsätt. Använd istället två separata satser.

Det finns även en möjlighet att deklarera bitar i en heltalstyp med hjälp av `:` . Detta kan i regel endast användas som delar i strukturer och unioner. Hur detta fungerar i detalj är dock ganska systemberoende. Men så här brukar det se ut.

`unsigned int on:1;`	Deklaration av en enda bit. Omfång fr.o.m 0 t.o.m 1.
`unsigned int dice:3;`	Deklaration av ett trebitarsvärde Omfång fr.o.m 0 t.o.m 7.

Så här deklarerar (definierar) man en strukturmall:

```
struct apb
  {
  char bokstav;
  int xpos;
  int ypos;
  };
```

Efter att du deklarerat (definierat) strukturen i en strukturmall kan du deklarera och använda strukturvariabler nästan på samma sätt som för vanliga variabler.

`struct apb b1;`		Här deklareras att `b1` ska vara typen `struct apb` .
`b1.bokstav`	(char)	Ger tillgång till tecknet `bokstav` i strukturvariabeln `b1` .
`b1.xpos`	(int)	Ger tillgång till heltalet `xpos` i strukturvariabeln `b1` .
`&b1.xpos`	(int *)	Ger en pekare som pekar på heltalet `xpos` i strukturvariabeln `b1` .
`&b1`	(struct apb *)	Ger en pekare som pekar på strukturen `b1` .
`b1`	(struct apb)	Ger en kopia av strukturen `b1` .

C hanterar strukturmallar och deras namn för sig, så en strukturmall kan heta samma sak som någon annan variabel, funktion eller egendefinierad typ, utan att det blir någon namnkonflikt.

Om man vet att man bara skall ha ett exemplar av en struktur behöver man inte först göra någon strukturmall för den, utan man skriver helt enkelt.

```
Struct
  {
  double xlimit;
  double ylimit;
  double zlimit;
  double expansionspeed;
  } rymden;
```

Och vips kan vi börja använda dess innehåll.

`rymden.zlimit`	(double)	Ger tillgång till flyttalet `zlimit` i strukturvariabeln `rymden` .

Det går även att initiera värdena i en struktur samtidigt som man deklarerar den.

```
struct apb b2 =
  {
  'F',
  0,
  0
  };
```

Unioner ser ut ungefär som strukturer. Men tillskillnad från en struktur som håller i sär sina beståndsdelar gör unionen tvärt om. Den låter det se ut om den bestod av flera beståndsdelar, men alla beståndsdelarna delar på samma ställe i datorns minne. Detta innebär i praktiken att man bara kan använda en av beståndsdelarna åt gången.

```
union brilldata
  {
  unsigned int solskyddsfaktor;
  unsigned int polariserande:1;
  int speglande:1;
  float brytning;
  };
```

Även unioner deklareras och används väldigt likartat som vanliga variabler och strukturvariabler.

`union brilldata b;`		Deklarerar att `b` ska vara av typen `union brilldata` .
`b.solfaktor;`	(int)	Ger tillgång till `solfaktor` i unionen `b` .

Man kan även skapa egena typer som sedan kan användas nästan precis på samma sätt som en vanlig typ.

`typedef int SOMELENGTH;`	Skapar en egen typ som heter `SOMELENGTH` av grundtypen `int` .
`SOMELENGTH d;`	Deklarerar att variabeln `d` ska vara av typen `SOMELENGTH`, vilket då blir samma sak som `int` .
`typedef long OTHERLENGTH;`	Skapar en egen typ som heter `OTHERLENGTH` , fast den här gången av utifrån grundtypen `long` .

Så här gör man en strukturtyp, dvs en typ som innehåller en struktur..

```
typedef struct
  {
  int n1;
  int n2;
  int n3;
  } triplet;
```

`triplet t;`		Deklarar `t` att vara en `triplet` , som i själva verket är en struktur.
`t.n3`	(int)	Ett värde i `triplet` .

`enum` är en konstig sak. `enum` kan användas för att skapa egendefinerade typer, likt `typedef` , fast bara motsvarande grundtypen `int` . `enum` kan användas för att deklarera variabler, fast återigen bara motsvarande typen `int` . Dessutom kan man med `enum` är definiera ”riktiga konstanter”, likt `const` , så länge det handlar om `int` eller motsvarande typ. Det primära syftet med `enum` är dock att definiera hela serier med konstanter, med automatiskt uppräknade värden.

I exemplet nedan blir `KNATTE` 1, `FNATTE` 2 och `TJATTE` 3. Alla tre blir av typen `int` . Skulle `= 1` utelämnats hade `KNATTE` blivit `0` , `FNATTE` 1 och `TJATTE` 2. Om inget annat sägs börjar uppräkningen på 0.

```
enum
  {
  KNATTE = 1,
  FNATTE,
  TJATTE
  };
```

Nedan ser vi hur man skriver för att definiera den egendefinierade typen `weekdays` , samtidigt som vi definerar upp några konstanter. Den egendefinierade typen baseras alltid på grundtypen `int` . Konstanterna blir alla av den nya egendefinierade typen.

```
enum weekdays
  {
  MONDAY = 0,
  TUESDAY,
  WEDNESDAY,
  THURSDAY,
  FRIDAY,
  SATURDAY,
  SUNDAY
  };
```

Man kan också passa på att skapa en variabel. I detta fallet deklareras den nya variabeln `current_modem_status` av den lika nya egendefinierade typen `modem_status_type` .

```
enum modem_status_type
  {
  MODEM_OFFLINE = 0,
  MODEM_DIALING,
  MODEM_CONNECTED
  } current_modem_status;
```

Det går utmärk att skapa en variabel utan att man definierar någon egendefinerad typ. Då blir variableln av typen `int` . Här ser vi också hur man kan göra ”hopp” i uppräkningen.

```
enum
  {
  CHANNEL_SVT1 = 1,
  CHANNEL_SVT2,
  CHANNEL_TV4 = 4
  CHANNEL_KANAL5
  } current_channel;
```

Pekare, matriser och strängar

Alla variabler, strukturer och unioner ligger verkligen någonstans i datorns minne, och i C går det utmärkt att få reda på var de finns.

`int x;`		Om variabeln `x` är deklarerad så här vet vi att typen är `int` .
`x`	(int)	`x` ger som bekant värdet på det heltal `x` innehåller.
`&x`	(int *)	`&x` däremot ger adressen till var i datorns minne heltalet `x` finns. (`&` kan tex uttalas ”adressen till ...” eller ”adressen där ... ligger”.)
`int *ptr;`		Deklaration av `ptr` . `ptr` blir här en pekare som pekar på ett heltalsvärde någon stans i datorns minne.

(På samma sätt som i deklarationen `int x`, i exemplet ovan, där `x` blev av typen `int`, blir i den här deklarationen `*ptr` av typen `int` .)

`*ptr`	(int)	`*ptr` är värdet på heltalet som ligger där pekaren `ptr` pekar. (`*` kan tex uttalas ”innehållet i adress ...”.)
`ptr`	(int *)	Värdet i `ptr`, dvs adressen som `ptr` pekar på.
`ptr + 7`	(int *)	Adressen 7 ”steg” framför där `ptr` pekar. (Hur långt ett steg är, i bytes räknat, beror på vilken typ som pekaren pekar på, i detta fallet en `int` , men också på hur en `int` lagras på ett visst system. Eller mer korrekt kan man säga att C vid addition av pekare och heltal först multiplicerar heltalet med storleken på den typ som pekaren pekar på. I detta fallet är det alltså `7 * sizeof(int)` som adderas med `ptr` . Addition där pekare som på en `void` ingår är inte tillåtna.)
`*(ptr + 7)`	(int)	Hämtar ett heltal 7 steg framför där `ptr` pekar.
`struct apb *pb;`		Deklarerar att `pb` ska vara en pekare som pekar på en struktur av typen `struct apb` .
`(*pb).bokstav`	(char)	Hämtar värdet av `bokstav` i den struktur av typen `struct apd` som `bp` pekar på.
`pb->bokstav`	(char)	Hämtar också värdet av `bokstav` i den struktur av typen `struct apd` som `bp` pekar på. Av någon orsak finns det en särskild operator, `->` , för att plocka ut en viss del ur en struktur som man har en pekare till.
`(&b1)->bokstav`	(char)	Följaktligen kan man då även skriva så här. (Vad det ska vara bra för ?) Detta hämtar alltså `bokstav` ur strukturen `b1` via en tillfällig pekare.
`void *p;`		Deklarerar pekaren `p` som inte pekar på något särskilt.
`int **p2;`		Deklarerar pekaren `p2` som pekar på en pekare som pekar på en `int` .

När en pekare för tillfället inte används till att peka på någon vettig data brukar man markera detta genom att låta den peka på adressen 0 (där kan man i regel ändå inte ha variabler och annat). Det brukar finns ett makro att använda för att få den här adressen, och det är `NULL` .

När man behöver hantera stora mängder av heltal, flyttal, strukturer, eller unioner använder man matriser.

`float m[5];`		Deklaration av att matrisen `m` ska bestå av 5 st flyttal, närmare bestämt av typen `float`.
`m[0]`	(float)	`m[0]` är värdet på det första talet i matrisen. (0,1,2,3,4 = 5st)
`m[2]`	(float)	`m[2]` är värdet på det tredje elementet i matrisen.

`m`	(float *)	`m` är lustigt nog adressen där matrisen börjar, och indirekt också adressen till där det första talet i matrisen ligger. Alltså är `m` också en pekare.
`*m`	(float)	`*m` är innehållet i adressen `m`, och indirekt det värde som ligger där matrisen börjar, dvs första värdet. `*m` är alltså samma sak som `m[0]` .
`*(m + 2)`	(float)	`*(m + 2)` är värdet som ligger på den adress man ”hamnar på” om man utgår från adressen `m` och hoppar 2 “hack“ fram i minnet. `*(m + 2)` är alltså samma sak som `m[2]`. (Eftersom `*(m + 2)` motsvarar `m[2]`, och elementär matte säger att `m + 2` är ekvivalent med `2 + m`, måste också `2[m]` vara samma sak som `m[2]`.)
`m + 2`	(float *)	adressen som man ”hamnar på” om man utgår från adressen `m` och hoppar 2 ”hack” fram i minnet. `m + 2` är samma sak som `&(m[2])`.
`char cm[50][10];`		Deklaration av en tvådimensionell matris av tecken (50 × 10 = 500 tecken).
`int *pm[3];`		Deklaration av en matris bestående av 3 pekare som var och en pekar på en `int` .
`int (*pm)[5];`		Deklaration av en pekare som pekar på en matris bestående av 5 stycken `int`.

Om man vill tilldela utgångsvärden för värden i en matris samtidigt som man deklarerar den använder man `{` , `}` och `,` . I exemplet nedan initieras de första 6 värdena av de 10 värden som deklareras.

```
float n[10] = { 6.95, 14.05, 84.80, 21.15, 19.50 };
```

Om man vill sätta matrisens storlek automatiskt, till det antal värden som anges skriver man:

```
long l[] = { 1984, 2001, 2010 };
```

Så här initierar man värden samtidigt som man deklarerar tvådimensionella matriser:

```
char c[3][2] =
  {
    { 'A', 'a' },
    { 'B', 'b' },
    { 'C', 'c' }
  };
```

`c[0][0]`	(char)	Tecknet `A`.
`c[0][1]`	(char)	Tecknet `a`.
`c[1][0]`	(char)	Tecknet `B`.

Två- eller flerdimensionella matriser lagras i själva verket i en lång följd. Så tex genom att låta pekaren `char *cp` peka på `c` kan man hantera den två dimensionella matrisen `c` som en vanlig endimensionell matris.

`*cp`	(char)	Tecknet `A`
`cp[0]`	(char)	Tecknet `A`
`cp[1]`	(char)	Tecknet `a`
`cp[2]`	(char)	Tecknet `B`

För att hantera textsträngar i C, används vanliga matriser. Två saker gäller dock. Typen på elementen i matrisen ska vara char, och ett särskilt sluttecken måste finnas där textsträngen ska ta slut.

`char str[5];`		Deklaration av en matris bestående av 5 stycken tecken.
`str[0] = 'H';`	(char)	Följande satser stoppar lämpliga tecken i matrisen.
`str[1] = 'e';`	(char)	
`str[2] = 'j';`	(char)	
`str[3] = '\0';`	(char)	Och så det berömda slutttecknet, eller NUL-tecknet, `'\0'`, vilket har värdet noll.

Vad som nu ligger och skräpar i `str[4]` och därefter är ointressant för i och med slutttecknet så är strängen slut. Om vi hade glömt att ha med slutttecknet skulle strängen fortsätta rakt ut i det stora okända minnet. Att ändra tecken i en sådan felaktig sträng kan innebära att man ändrar i någon helt annan data som ligger där, eller ännu värre är det om själva programmet man för tillfället kör som ligger där.

`str`	(char *)	Pekare till det första tecknet.
`str[1]`	(char)	Det andra tecknet.
`&str[0]`	(char *)	Pekare till det första tecknet.
`char *s1 = "Ett";`		Ett sätt att deklarera en teckenpekare som pekar på en sträng.
`char s2[] = "Två";`		Ett annat sätt.
`s2[2]`	(char *)	Tecken `å`.
`s2[3]`	(char *)	Tecken `'\0'`. Dvs NUL-tecknet. Dvs slutttecknet.

Typning.

En viktig egenskap i C är möjligheten att typa om värden från en typ till annan. Vissa omvandlingar anses så självklara att de sker med automatik. Andra är mer långsökta och kräver vår uttryckliga order, för att kompilatorn ska lägga in en typkonvertering, Vissa saker går egentligen inte att konvertera över huvudtaget. Då kan man ofta gå runt problemet med pekare, men det sker i så fall helt på egen risk.

För att uttryckligen säga att värdet från ett uttryck skall typas om används `( )` före uttrycket.

`int x;`		`x` är av typen `int`, återigen.
`(char)x`	(char)	Detta uttryck får typen `char` trots att `x` är deklarerad som en `int` . (Tänk på att olika typer kan hantera olika stora värden, och att en sådan här konvertering inte alltid kan genomföras korrekt.)
`int *ptr;`		Deklaration av en pekare till ett heltal.
`ptr`	(int *)	`ptr` är en pekare till ett heltal.
`(char *)ptr`	(char *)	Plötsligt har vi ett uttryck som ”tror” att det pekar på en char.
`*((char *)ptr)`	(char)	Innehållet som ligger på `(char *)ptr` , läst som ett tecken. Så förmodligen har vi fått ett halvt (avrivet) heltal, som när vi nu ser på det som ett tecken saknar rim och reson.

Förutom denna i C-språket inbyggda typkonvertering kan man också använda funktioner för att konvertera data från en typ till en annan.

Operatorer och separatorer

De som står överst har högst prioritet (lägst nummer).

1.	`( )`	Parenteser. (Ändrar eller förtydligar prioritetsordningen i ett uttryck.)
	`x[y]`	Val av del `y` i matris `x` .
	`x.y`	Urval ur struktur. `x` är här en struktur, och `y` är en del i strukturen.
	`x->y`	Urval ur den struktur som en pekare (förhoppningsvis) pekar på. `x` är här en struktur, och `y` är en del i strukturen.
2.	`!x`	Logiskt NOT. Om `x` är lika med 0 svarar uttrycket med heltalet 1, annars med 0. `x` förväntas vara ett heltal.
	`~x`	Bitvis NOT (tvärtom). `x` förväntas vara ett heltal.
	`+x`	Svarar med `x`. `x` förväntas vara ett heltal eller ett flyttal.
	`-x`	Svarar med `-x`. `x` förväntas vara ett heltal eller ett flyttal. `-x` är likvärdigt med `0 - x` .
	`x++`	Svarar först med `x`, sedan adderar `x` med 1. `x` förväntas vara ett heltal eller ett flyttal.
	`++x`	Adderar först `x` med 1, sedan svarar uttrycket med `x`. `x` förväntas vara ett heltal eller ett flyttal.
	`x--`	Svarar först med `x`, sedan subtraheras `x` med 1. `x` förväntas vara ett heltal eller ett flyttal.
	`--x`	Subtraherar först `x` med 1, sedan svarar uttrycket med `x`. `x` förväntas vara ett heltal eller ett flyttal.
	`&x`	Adressoperatorn. (Ger adressen som pekar på `x`)
	`*x`	Indirektoperatorn. (Ger värdet som ligger på adressen `x`)
	`sizeof(x)`	Svarar med storleken på en variabeln eller typen `x`.
3.	`x * y`	Svarar med produkten av `x` multiplicerat med `y`. Heltal eller flyttal.
	`x / y`	Svarar med resultatet av divisionen `x` genom `y`. Heltal eller flyttal.
	`x % y`	Svarar med resten från divisionen `x` genom `y`. Endast för heltal.
4.	`x + y`	Svarar med resultatet av additionen `x` och `y`. Heltal eller flytttal.
	`x - y`	Svarar med resultatet av `y` subtraherat från `x`. Heltal eller flytttal.
5.	`x << y`	Svarar med `x` bitvis skiftat `y` steg åt vänster. Endast för heltal.
	`x >> y`	Svarar med `x` bitvis skiftat `y` steg åt höger. Endast för heltal.
6.	`x < y`	Svarar med 1 om `x` är mindre än `y`, annars 0. Heltal eller flytttal.
	`x > y`	Svarar med 1 om `x` är större än `y`, annars 0. Heltal eller flytttal.
	`x <= y`	Svarar med 1 om `x` är mindre eller lika med `y`, annars 0. Heltal eller flytttal.
	`x >= y`	Svarar med 1 om `x` är större eller lika med `x`, annars 0. Heltal eller flytttal.
7.	`x == y`	Svarar med 1 om `x` är lika med `y`, annars 0. I princip endast för heltal. Flytttal rekommenderas ej.
	`x != y`	Svarar med 1 om `x` är skilt från `y`, annars 0. I princip endast för heltal. Flytttal rekommenderas ej.
8.	`x & y`	Bitvis AND (båda). Endast för heltal.
9.	`x ^ y`	Bitvis XOR (antingen eller, ej båda). Endast för heltal.
10.	`x | y`	Bitvis OR (eller). Endast för heltal.

11. `x && y` Logiskt AND. Om både `x` och `y` är skilda från 0 svarar uttrycket med 1, annars 0.
12. `x || y` Logiskt OR. Om `x` eller `y` är skilda från 0 svarar uttrycket med 1, annars 0.
13. `x ? y : z` Om `x` är skilt från 0 svarar uttrycket med `y` annars med `z` .
14. `x = y` Svarar med `x` , efter att `x` tilldelats vad som finns i `y` .

    `x *= y` Svarar med `x` , efter att `x` tilldelats resultat av operationen `x * y` .

    `x /= y` Svarar med `x` , efter att `x` tilldelats resultatet av operationen `x / y` .

    `x %= y` Svarar med `x` , efter att `x` tilldelats resultatet av operationen `x % y` .

    `x += y` Svarat med `x`, efter att `x` tilldelats resultatet av operationen `x + y` .

    `x -= y` Svarar med `x`, efter att `x` tilldelats resultatet av operationen `x - y` .

    `x <<= y` Svarar med `x`, efter att `x` tilldelats resultatet av operationen `x << y` .

    `x >>= y` Svarar med `x`, efter att `x` tilldelats resultatet av operationen `x >> y` .

    `x &= y` Svarar med `x`, efter att `x` tilldelats resultatet av operationen `x & y` .

    `x ^= y` Svarar med `x`, efter att `x` tilldelats resultatet av operationen `x ^ y` .

    `x |= y` Svarar med `x`, efter att `x` tilldelats resultatet av operationen `x | y` .
15. `x, y` Evaluerar först `x` (men använder inte resultatet). Svarar sedan med `y` .

Det finns något som kallas för separatorer också, som inte utför någonting på samma sätt som operatorer. Utan bara är stöd för kompilatorn. Vissa separatorer och operatorer har samma utseende och är förvillande närbesläktade. Det är endast var de står i koden som avgör om de är en operator eller en separator.

`[ ]` Används vid deklaration av matriser.

`( )` Används efter villkorssatser för att skilja på villkor och sats. Används för att utgöra en definition av funktioner, eller för att göra ett funktionshopp.

`{ }` Används för att bunta ihop flera satser så att de betraktas som en. Används för att definiera vad som ska ingå i funktioner, strukturmallar och unionsmallar. För initiera värden i en matris.

`,` Separerar parametrar vid funktionsanrop.

`;` Avslutar sats.

`:` För att skilja på utfall vid användning av switch - case. Samt även för att ange ett ”hoppmål” (en ”label”) vid användning av goto, om nu någon skulle vilja det.

`...` Används vid funktionsdeklaration för att ange att antal parametrar är obestämt.

`*` Används i deklarationer framför ett symbolnamn för att ange att det är en pekare.

`=` Använd för tilldelning av initieringsvärden konstanter vid deklaration av variabler.

Programstyrarande satser.

Vanligt villkor - if - else

Om ett uttryck är skilt från 0, görs en sak, annars görs något annat.

Tex om `y` är skilt från 0, sätts då `x` till 3, annars sätts `q` till -1.

```
if(y)
  x = 3;
else
  q = -1;
```

Växel - switch - case - default

Hoppa till olika ställen i koden beroende på ett värde. Om ingen `case` stämmer sker ett hopp till `default` om denna finns längst ner. Om ingen `case` stämmer och ingen `default` finns hoppas hela switch-satsen över. För att avbryta koden och hoppa ur en switch-sats kan man skriva `break` . Normalt behöver man ett `break` för varje `case`. Men ibland kan det vara en poäng att utelämna `break` så att exekveringen av koden fortsätter förbi en eller flera `case`.

```
switch(siffra)
  {
  case 1:
    tecken = '1';
    break;
  case 2:
    tecken = 'X';
    break;
  case 3:
    tecken = '2';
    break;
  default:
    tecken = '?';
  }
```

Slinga (loop) - while

Kör en programsats om och om igen, så länge ett uttryck är skilt från 0. Om uttrycket ger 0 redan första gången körs inte satsen.

```
i = 10;
while(i)
  {
  q = q * i;
  i--;
  }
```

Slinga (loop) - do – while

Kör en programsats om och om igen, så länge ett uttryck är skilt från 0. Programsatsen körs minst en gång, först därefter görs kontrollen.

```
i = 10;
do
  {
  q = q * i;
  i--;
  } while (i);
```

Slinga (loop) - for

I vissa fall kan man kanske använda detta skrivsätt. (Påminner om BASIC’s FOR och NEXT.)

```
for(i = 1; i <= 10; i++)
  q[i] = i;
```

for-slingan ovan motsvarar:

```
i=1;
while(i<=10)
  {
  q[i] = i;
  i++;
  }
```

Alla slingor kan avbrytas. **`break`** avbryter slingan helt. **`continue`** avbryter bara aktuellt varv.

Hopp - goto

Fast det är hemskt. Kan man i C göra hopp från ett ställe i koden till ett annat. REKOMMENDERAS INTE !

```
HIT:
goto HIT;
```

Egna funktioner, main och standardfunktioner

Här följer ett exempel på hur man definierar en funktion:

```
int dubbelt_upp(int x)
  {
  return x + x;
  }
```

Så här kan det se ut när man använder den:

```
int en_del = 4;
int massor;
massor = dubbelt_upp(en_del);
```

Om man vill använda en funktion som finns senare i koden eller om den rent utav finns i någon helt annan fil, någon annan stans, måste man deklarera funktionen. Det görs så här:

```
int dubbelt_upp(int);
```

Följande exempel visar två saker. Först hur man kan deklarera en pekare som är gjord för att peka på en funktion som tar en `int` som parameter och som returnerar en `int` . Sedan visas hur man kan få reda på var i minnet en funktionen ligger, genom att bara skriva funktionens namn, utan `( )`.

```
int (*ptr)(int);
ptr = dubbelt_upp;
```

Nedan följer ett exempel på hur man kan hoppa dit en pekare pekar. Det är alltså bara att sätta dit `( )` så betyder det att ett anrop ska göras.

```
x = ptr(12);
```

Om man vill skapa en funktion som tar valfritt antal argument måste `...` användas. Exakt hur man kodar en sådan funktion är ett specialämne som ni får läsa om någon annan stans. Men så här kan en deklaration se ut:

```
int medel(int count, ...);
```

Om man ska definiera en pekare till en funktion som tar flera argument ser det ut så här.

```
int (*ptr_to_compare_function)(int, int);
```

Om man vill skicka en sådan här pekare till en annan funktion kan den funktionen deklareras och definieras så här:

```
sort_list(int *, int , int (*)(int, int));
sort_list(int elements[], int count, int (*comparer)(int, int));
  {
  //TODO - Write some code here (SIC)
  }
```

`long *f(void);` Deklarerar `f` som en funktion som returnerar en `long` .

`long (*p)(void);` Deklarerar `p` som en pekare till en funktion som returnerar `long` .

Också lite om den vikigaste funktionen av alla, specialfunktionen `main()` . För att göra en körbar (startbar) programfil måste det finnas en särskild funktion (som operativsystemet ”anropar”). Denna funktion heter alltid `main()` . När funktionen `main()` är klar avslutas programmet. Så `main()` är både starten och slutet för programmet. När programmet avslutas skall en kod returneras till operativsystemet, som anger om allt gått bra eller inte. 0 betyder vanligtvis bra.

Du behöver inte deklarera `main()` för att använda den. Den finns liksom inbyggd redan. För att krångla till det lite finns det i regel två `main()` att välja mellan. `int main(void);` och `int main(int, char **);` (Dessa möjligheter kan variera lite beroende på system).

Den enklare varianten använder man om programmet inte behöver ta argument (direktiv).

```
int main(void)
  {
  printf("Enkelt\n");
  return 0;
  }
```

Den lite krångligare varianten tar man till om programmet behöver ta emot argument (direktiv).

```
int main(int argc, char *argv)
  {
  if(argc > 1 && argv[1][0] == 'v')
    {
    printf("Version 3.2\n");
    }
  else
    {
    printf("Lite krångligare\n");
    }
  return 0;
  }
```

Sedan finns en rad standardfunktioner man kan välja att använda om man vill. Man måste inte. Man kan använda andra som någon annan gjort eller göra egna om det finns särskilda skäl för det.

Här gås endast de allra viktigaste standardfunktionerna igenom.

Utmatning av data - printf()

Ex:

```
printf("Hej");
```

kommer helt enkelt skriva ut

```
Hej
```

Ex:

```
x = 75;
printf("Höjd: %d m.ö.h.", x);
```

...ger följande utskrift.

```
Höjd: 75 m.ö.h.
```

`printf()` kan ta hur många parametrar som helst, bara det finns lika många speciella teckenkombinationer, som säger åt `printf()` vad som ska hämtas ur parameterlistan, och hur det ska skivas ut. Observera att det är funktionen `printf()` (och några till) som tolkar `%` som ett specialtecken. Det är alltså inget som gäller rent allmänt för C.

`%d`	hämtar ett heltal och skriver ut talet vanligt.
`%4d`	hämtar ett heltal och skriver ut talet i ett 4 tecken långt högerjusterat fält.
`%-4d`	hämtar ett heltal och skriver ut talet i ett 4 tecken långt vänsterjusterat fält.
`%04d`	hämtar ett heltal och skriver ut talet i ett 4 tecken långt 0-utfyllt fält.
`%+d`	hämtar ett heltal och skriver ut talet med tecken, alltid.
`%u`	hämtar ett `unsigned` heltal och skriver ut vanligt.
`%f`	hämtar ett flyttal och skriver ut talet som vanligt.
`%.3f`	hämtar ett flyttal och skriver ut talet med 3 decimaler.
`%10.3f`	hämtar ett flyttal och skriver ut talet med 3 decimaler, i ett 10 tecken långt högerjusterat fält.
`%e`	hämtar ett flyttal och skriver ut talet som i potensform (exponent)
`%g`	hämtar ett flyttal och skriver ut talet precis som `%f` eller `%e` beroende på talets storlek.
`%c`	hämtar en tecken och skriver ut som ett tecken.
`%s`	hämtar en pekare och skriver ut den teckensträng som förväntas ligga där pekaren pekar.
`%o`	hämtar ett heltal och skriver ut oktalt.
`%x`	hämtar ett heltal och skriver ut hexadecimalt.

(På vissa system kan man vara tvungen att skriva tex `%ld` för att hämta en `long integer` , `%hd` för `short integer` , `%lf` för `double` och `%Lf` för en `long double` .)

Mata data i en sträng - sprintf()

Ex:

```
char s[30];
float x = 1.6;
float y = -8.333333;
sprintf(s, "Koordinat: %.3f, %.3f.", x, y);
```

Kommer skriva `Koordinat: 1.600` , `-8.333` i teckenmatrisen `s` .

`sprintf()` använder samma specialtecken som `printf()` .

Inhämta data - scanf()

Ex:

```
long v;
scanf("%l", &v);
```

Denna kod kommer att hämta in tecken från tangentbordet (eller liknande) och tolka det som ett heltal för att sedan lagra det i `v` .

`scanf()` använder nästan samma specialteckenkombinationer som `printf()` . `%g` finns dock inte. `%f` och `%e` betyder samma sak.

Sätta samman strängar - strcat()

Ex:

```
strcat(text, extra);
```

Kopiera strängar - strcpy()

Ex:

```
strcpy(cloned, original);
```

Jämföra strängar - strcmp()

Ex:

```
strcmp(pears, apples);
```

Strängars längd- strlen()

Ex:

```
l = strlen(sometext);
```

Antal sekunder sedan början på 1970 - time()

Ex:

```
t = time(NULL);
```
eller
```
time(&t);
```

Slumptal - rand() - srand()

Ex:

```
srand(time(NULL));
prognos = rand() % 50;
```

Filhantering - fopen() - fclose() - fprintf() - fscanf() - fputs() - fgets() - fgetc - fputc()

Ex:

```
FILE *f;
if(f = fopen("dung.txt", "rb"))
  {
  char ch;
  while((ch = getc(f)) != EOF)
    printf("%c", ch + 1); //Kryptering, typ
  fclose(f);
  }
```

Konverteringsfunktioner - atoi() - atof()

Ex:

```
int x;
x = atoi("003.1400");
```

Minneshantering - malloc() - free()

Ex:

```
char *sp;
sp = malloc(1000);
```
...
```
free(sp);
```

Helhetsexempel

En liten fil som visare hur man kan göra det klassiska programmet Hello world.

```
// hello.c
#include <stdio.h>
int main()
  {
  char who[] = "World";
  printf("Hello %s!", who);
  return 0;
  }
```

Tre små filer som utgör ett program som räknar fram en omkrets.

```
// omkrets.h
#ifndef OMKRETS_H
#define OMKRETS_H
#define PI_F 3.141592F
float omkrets(float);
#endif
```

```
// omkrets.c
#include "omkrets.h"
float omkrets(float lr)
  {
  return lr * 2.0F * PI_F;
  }
```

```
// exempel.c
#include <stdio.h>
#include "omkrets.h"
int main()
  {
  float r;
  r = 7.94F;
  printf("Omkretsen är %f", omkrets(r));
  return 0;
  }
```

GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document ”free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of ”copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The ”Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as ”you”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A ”Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A ”Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The ”Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The ”Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A ”Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not ”Transparent” is called ”Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The ”Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, ”Title Page” means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section ”Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as ”Acknowledgements”, ”Dedications”, ”Endorsements”, or ”History”.) To ”Preserve the Title” of such a section when you modify the Document means that it remains a section ”Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

* A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

* B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

* C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

* D. Preserve all the copyright notices of the Document.

* E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

* F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

* G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

* H. Include an unaltered copy of this License.

* I. Preserve the section Entitled ”History”, Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled ”History” in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

* J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the ”History” section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

* K. For any section Entitled ”Acknowledgements” or ”Dedications”, Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

* L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

* M. Delete any section Entitled ”Endorsements”. Such a section may not be included in the Modified Version.

* N. Do not retitle any existing section to be Entitled ”Endorsements” or to conflict in title with any Invariant Section.

* O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled ”Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled ”History” in the various original documents, forming one section Entitled ”History”; likewise combine any sections Entitled ”Acknowledgements”, and any sections Entitled ”Dedications”. You must delete all sections Entitled ”Endorsements.”

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an ”aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled ”Acknowledgements”, ”Dedications”, or ”History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License ”or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.