C-programmering, föreläsning 1

Jesper Wilhelmsson

- Introduktion till C
- Variabler, Typer, Konstanter
- Operatorer
- Villkorliga satser – `if-else`, `switch`
- Loopar – `for`, `while`, `do ... while`
- Inmatning och utmatning – `stdin` / `stdout`
- Här startar det – `main`
- En kompilator – `gcc`

Introduktion till C

C är ett imperativt programmeringsspråk, till skillnad från ML som ni sett tidigare. I ML kretsar allt kring uttryck som skall beräknas till ett värde, i C så är det i första hand inte returvärdet som är det viktiga, utan sidoeffekterna.

Språket har sitt ursprung i Unix. Det är en utveckling av B (1970) som utvecklades för att skriva applikationer till den första Unix:en på en DEC PDP-7. Språket C såg dagens ljus 1972. 1978 definierade Kernighan och Ritchie den version av C som kom att bli dominerande. ANSI-C (1988) är en utveckling av K&R:s C där man filat bort en del av de kantiga hörnen och specificerat ett brett funktionsbibliotek.

C är som sagt utvecklat för att skriva applikationer till Unix och ligger därför mycket nära operativsystemet. Det finns inga skyddsmekanismer, om programmeraren gör fel så dör programmet på riktigt och har man otur så tar det andra delar av systemet med sig i fallet. Detta till trots så är ANSI-C plattformsoberoende. Så länge man inte ger sig på riktigt håriga saker i minnet, så kan man flytta en applikation från hårdvara till hårdvara från operativsystem till operativsystem utan att behöva ändra någonting i källkoden. Unix är ett lysande exempel på detta. De flesta av de applikationer vi använder dagligen är skrivna i C, och alla dessa kan kompileras på vilken hårdvara som helst eller flyttas till andra operativsystem utan att källkoden behöver förändras. (Cygwin för Windows är ett exempel, Geek Gadgets för AmigaOS ett annat.)

Kompileras ja. C är inte ett interpreterande språk. Det finns ingen kommandorad där man kan testköra enstaka funktioner eller beräkna små uttryck. För att köra ett C-program måste man kompilera (översätta) det till maskinkod och sedan starta programmet precis som man startar vilken annan applikation som helst. Exakt hur detta går till återkommer vi till i slutet av denna föreläsning.

C hanterar samma datatyper som maskinen; tecken, siffror och adresser. Det finns inga inbyggda operationer för att hantera listor, arrayer, strängar mm. Det finns ingen inbyggd hantering av I/O. C erbjuder sekventiell kod uppbyggd av satser (till skillnad från ML:s uttryck) och lämpar sig mycket bra för lösning av sekventiella problem.

Låt oss titta på ett exempel: Beräkna arean av en cirkel
Problemet är ett standardproblem och består av tre uppenbara delar:

Inmatning (få tag i radien) - Beräkning (av arean) - Utskrift (av arean)

Utan att kunna något om programmering kan vi beskriva problemet med pseudokod (= låtsaskod / nästankod). Detta är en intuitiv steg-för-steg-beskrivning av vad som behöver göras för att beräkna en cirkels area i ett program:

1. radie ⇐ få tag i en radie
2. r2 = radie * radie
3. area = pi * r2
4. skriv ut area

Som vi ser består problemet av fyra satser. Det vi är intresserade av i programmet är sidoeffekterna. I detta exempel är sidoeffekten först att värdena i variablerna förändras och sedan att resultatet skrivs ut. Som tidigare nämnts så måste inmatning och utskrift ske explicit genom funktions-anrop. ANSI C har ett väldefinierat funktionsbibliotek. Det har 15 stycken avdelningar för olika typer av funktioner. De vanligaste är:

math - Matematiska funktioner
stdio - In och utmatning, filhantering (upptar en tredjedel av biblioteket)
stdlib - Konvertering, minnesallokering
string - Sträng- och minneshantering

Variabler, Typer, Konstanter

Variabler är lagringsplatser för data. Variabler har många användnings-områden men typiskt så används de för att mellanlagra värden mellan beräkningar. Alla variabler som används i ett C-program måste deklareras innan de används. När man deklarerar en variabel anger man även en typ. Denna typ talar om vilken sorts data som variabeln kommer att innehålla. Om ingen typ anges är standardtypen `int` (heltal), men ni får inte utelämna typen under den här kursens gång. De inbyggda typer som C tillhandahåller visas i tabell 1.

En typisk variabeldeklaration: `int x;`

Olika datatyper kan representera olika stora mängder data. Ett normalt heltal (`int`) representerar till exempel en större mängd än ett tecken (`char`). Detta innebär att olika typer kräver olika mycket minne. Den exakta siffran på hur mycket minne en variabel tar är maskinberoende och beror främst på hur stora minnesord hårdvaran kan hantera. De flesta datorer idag är 32-bitars-maskiner, men även 64-bitars-maskiner börjar bli vanligare. För att se hur mycket minne en datatyp tar kan man använda kommandot `sizeof()`. `sizeof(int)` ger till exempel svaret att en `int` kräver 4 bytes minne på en 32-bitars-maskin och 8 bytes på en 64-bitars. Datatypernas storlekar finns i tabell 1 och är hämtade från en 32-bitars-maskin.

Typ	*sizeof()*	*Kommentar*
char	1	Alltid en byte
int	4	Oftast maskinens ordstorlek
short	2	(16 bitar = -32768 → 32767)
long	4	(32 bitar = -2147483648 → 2147483647)
long long	8	(64 bitar = $-922*10^{16} \rightarrow 922*10^{16}$)
float	4	Oftast 32 bitar, 6 signifikanta siffror, $10^{-38} \rightarrow 10^{38}$
double	8	
long double	16	

Tabell 1: Datatyper och dess storlek i minnet på en 32-bitars-maskin.

Alla datatyper förutom `char` är signerade om man inte säger något annat. Det vill säga, de kan lagra både negativa och positiva tal. Vill man ändra detta kan man använda nyckelorden `signed` och `unsigned`. På detta sätt kan vi skapa variabler som endast håller positiva tal och på så sätt få dubbelt så stora tal att leka med. När det gäller `char` så beror det på plattformen om den är signerad eller ej och det enda sättet att få helt maskinoberoende program är att specificera typen.

För att ge programmeraren tillgång till maskinen på en mycket låg nivå tillhandahåller C också en datatyp för pekare (adresser). Dessa är alltid samma storlek som ett maskinord (typiskt 32 bitar). För att deklarera en pekare lägger man till en `*` framför variabelnamnet: `int *x;`
Vi återkommer till pekare i en senare föreläsning, här vill jag bara tala om att de finns.

Många gånger kan man flytta data mellan variabler av olika typ utan att C säger något om det. Detta är möjligt då hela data-värdet får plats i den nya datatypen. Exempel:

```
int liten = 4711;
long long stor;

stor = liten;
```

Detta går bra eftersom en `int` är mindre än en `long long`.

Men om man vill flytta åt andra hållet så går det inte lika bra. C kommer att ge en varning, vilket vi i denna kurs betraktar som ett fel. Detta betyder naturligtvis inte att det inte går, C är ett ganska lättövertalat språk som förutsätter att programmeraren vet vad den gör. För att få tyst på C måste man typkonvertera explicit. Decimaler och de delar av större datatyper som inte får plats klipps bort.

```
liten = (int)stor;

int x = (int) 3.852   x = 3
char c = (char) 258   ch = 2   1 0000 0010 ⇒ 0000 0010
```

Man kan även deklarera att en variabel inte alls ska vara en variabel, utan en konstant. Detta gör man genom att sätta nyckelordet `const` framför deklarationen. En konstant måste få sitt värde direkt vid deklarationen.

```
const int x = 42;
```

Vi återgår till exemplet om att beräkna en cirkels area och försöker så smått börja översätta pseudokoden till C. Innan vi kan utföra några beräkningar måste vi alltså deklarera variablerna. Vi vill ha flyttal men behöver inte allt för hög precision, så `float` duger. Flera variabler kan deklareras i samma sats om de har samma typ:

```
float radie, r2, area;
const float pi = 3,1415;
```

Operatorer

När det gäller operatorerna i C kan man nog säga att allt är precis så som man vill ha det. De fyra räknesätten (`+`, `-`, `*`, `/`) finns och fungerar så som man är van vid från matematiken. Det finns även en operator för modulo (`%`).

Även de logiska operatorerna är intuitiva, likhet, större än, mindre än mm. Värt att notera är att likhet skrivs med två likhetstecken (`==`), detta för att skilja det från tilldelning. För att knyta ihop flera logiska villkor kan man använda `&&` (och) eller `||` (eller). Negerar logiska uttryck gör man med `!`.

Alla logiska operatorer resulterar i ett sanningsvärde. Det finns ingen speciell typ för detta värde, utan i C:s ögon är det ett vanligt tal precis som allt annat. Falskt = 0 och sant = 1. Egentligen är allt annat än 0 sant, men det är 1 som returneras av de logiska operatorerna.

Eftersom C ligger nära maskinen finns även operatorer för att manipulera bitmönster. Till exempel `<<` och `>>` som flyttar bitarna i ett tal åt vänster respektive höger. `~` ger två-komplementet, dvs alla ettor blir noll och alla nollor blir ett. Logiska bitoperationer kan göras med `&` (och), `|` (eller) eller `^` (xor).

För detaljerad information om C:s operatorer och sanningstabeller för de logiska operatorerna, se separat OH-bild.

Därmed kan vi börja beräkna vår cirkel-area, och som vi ser så är C-koden identisk med den intuitiva pseudokod vi skrev innan vi kunde något om C:

```
r2 = radie * radie;
area = pi * r2;
```

Villkorliga satser – `if-else`, `switch`

För att styra kontrollflödet i ett program, det vill säga för att kontrollera vilken kod som ska köras, så behövs någon form av villkorliga satser. Kod som kapslas in i en villkorlig sats kommer endast att köras om det givna villkoret är uppfyllt. I C använder man `if` för villkorliga satser.

```
if ( villkor )
   sats;
```

Ibland vill man kunna göra något annat om villkoret inte är uppfyllt:

```
if ( villkor )
   sats1;
else
   sats2;
```

Den villkorliga satsen kan vara precis vilken kod som helst, även ett nytt villkorligt uttryck. Normalt skriver man då detta på samma rad som `else`:

```
if ( villkor1 )
   sats1;
else if ( villkor2 )
   sats2;
else ( villkor3 )
   sats3;
```

Som synes förväntar sig C endast en sats efter villkoret. För att utföra lite fler saker kan man slå in koden i { ... }. { ... } betraktas som en sats i C.
Exempel:

```
if ( x == 10 )
{
   y = 32;
   y = y + x;
}
else if ( x > 10)
   x = x - 1;
else
   x = x + 1;
```

Beroende på vad en variabel har för värde så vill man ibland utföra olika kod. Med hjälp av en `switch` kan man åstadkomma detta. Olika fall (`case`) kommer att väljas utifrån värdet på variabeln.

```
switch ( variabel )
{
      case värde1: sats1; break;
      case värde2: sats2; break;
      case värde3: sats3; break;
      default: sats4;
}
```

Kommandot `break` används för att tala om att det är dags att avbryta körningen där och hoppa till koden efter `switch`-satsen. Om `break` inte skrivs ut kommer körningen att fortsätta på nästa `case`. Detta kan vara användbart om det är flera fall som ska köra samma eller delvis samma kod.

```
char ch = read_from_user();
switch ( ch )
{
   case 'a':
   case 'A': do_the_A_thing(); break;
   case 'c': create_new_thing();
   case 'm': modify_existing_thing(); break;
   default: user_is_stupid();
}
```

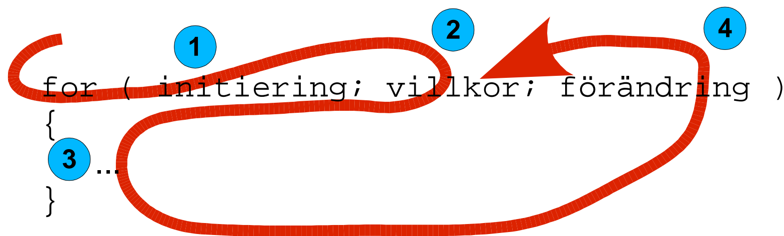
Loopar – `for`, `while`, `do ... while`

För att kunna skriva riktiga program krävs något sätt att iterera (upprepa). C tillhandahåller tre typer av loopar: `for`, `while` och `do ... while`. Det finns ett fjärde sätt, rekursion. Rekursion är dock inte att rekommendera i C. Varje gång en funktion anropas kommer dess lokala omgivning (argument mm - vi kommer tillbaka till det i en senare föreläsning) att allokeras i minnet. Rekursion, som betyder att en funktion anropar sig själv, kommer alltså att innebära att programmet äter lite minne för varje varv i loopen. Om loopen håller på för länge kommer alltså minnet att ta slut. Därför används hellre icke-rekursiva alternativ i C.

`for` (initiering; villkor; förändring) { kropp (=satser) **}**

`for`-loopen består av fyra delar: initiering, loop-villkor, förändring och kropp. Samtliga dessa kan i princip vara godtycklig kod, men det vanliga är att man har något i stil med: `for ( x = 0; x < 10; x++ )` sats;

Vad händer i en `for`-loop?



1. Utför initieringskod
2. Kontrollera om villkoret är uppfyllt, om inte avslutas loopen här
3. Utför koden i loop-kroppen
4. Utför förändringskoden
5. Gå tillbaka till steg 2.

Som vi ser i figuren ovan så kan det alltså hända att kroppen i en loop aldrig exekveras. Detta händer om villkoret inte är uppfyllt då loopen startar. `for`-loopens natur gör att den är mycket lämplig att använda då man vet hur många iterationer man vill utföra. Typiskt är att man använder `for` för att stega igenom datatyper av en bestämd storlek eller liknande.

`while` (villkor) { kropp }

`while` fortsätter att iterera tills dess att villkoret inte längre är uppfyllt. Detta gör loopen lämplig att använda då man inte vet hur många gånger man vill iterera, utan vill fortsätta tills dess att något händer som falsifierar villkoret.

`for` och `while` har olika syfte men gör egentligen samma sak. Man kan alltid byta ut en `for` mot en `while` och vise versa:

`while` (villkor) { kropp } = `for` (; villkor;) { kropp }

`for` (initiering; villkor; förändring) { kropp }
=
initiering; `while` (villkor) { kropp; förändring; }

`do` { kropp } `while` (villkor);

Både `for`-loopen och `while`-loopen kan passeras utan att koden i kroppen någonsin exekveras. Om man vill garantera att något händer minst en gång finns `do-while`. `do-while` kör loop-kroppen en gång först innan villkoret kontrolleras.

Två nyckelord kan användas tillsammans med alla tre looparna: `break` och `continue`. `break` avbryter loopen och hoppar vidare till koden som ligger efter. `continue` bryter loop-kroppen i förtid och hoppar tillbaka till villkoret.

Inmatning och utmatning – stdin / stdout

Hittills har vi endast pratat om inbyggda operatorer och kommandon i C. Inte för att vi på något sätt har gått igenom alla de inbyggda konstruktionerna redan, men det är ändå dags att börja titta lite på vad som finns i standardbiblioteken. Vi börjar med stdio.

All inmatning och utmatning i C sker genom strömmar. Man kan se en ström som ett rör där någon stoppar in tecken i den ena änden och någon annan plockar ut tecknen i den andra änden. De vanligaste strömmarna kallas stdin och stdout.

stdin är den ström som ett C-program oftast hämtar in information från användaren ifrån. Normalt är stdin knuten till tangentbordet. Det är alltså tangentbordet som stoppar tecken i den ena änden av stdin-röret och ditt C-program som plockar ut dem i den andra. För att göra detta använder man funktioner definierade i stdio.

getchar() - läs ett tecken från stdin. Funktionen är ganska lättanvänd. Man anropar den och den returnerar ett tecken från stdin.

scanf(format-sträng, ...) - läs in formaterad text. scanf är lite knepigare att använda. Man skickar med en format-sträng som styr vilken indata man vill ta emot. Formatsträngen innehåller styrkoder för till exempel heltal, tecken eller ord och för varje styrkod skickar man med en variabel dit scanf skriver motsvarande värde. Det är nog enklast att visa med ett exempel:

```
int x;
scanf(”%d”,&x);
```

Det där betyder att vi vill läsa in ett tal från stdin och lägga detta i x. %d är alltså styrkoden för heltal. Som vi kan se så är det något märkligt framför x:et i anropet till scanf, ett &. Detta är operatorn för att hämta adressen till en variabel. Eftersom C är call-by-value så kan vi inte skicka med x rakt av. Det som då skulle skickas till scanf är ju innehållet i x, dvs något tal. Men scanf vill ju skriva ett nytt värde i variabeln x och för att kunna göra det krävs att scanf vet var i minnet den ska skriva. Därför måste vi skicka adressen till x, inte x. Format-strängen kan innehålla flera styrkoder och annan text som matchas mot den inkommande strömmen av tecken.

```
stdin: ”Ett tal: 42 Tecken: B Ord: ost”
scanf(”Ett tal: %d Tecken: %c Ord: %s”,&x,&y,&z);
```

%s används ovan för att läsa in ett ord. scanf avslutar inläsningen av ett ord vid ett blankt tecken - mellanslag, tab, radbrytning mm. Detta gör att scanf endast kan läsa in enstaka ord, vill man läsa in meningar blir det klurigare. Detta ska vi titta närmare på i inlämningsuppgift 3.

Den andra strömmen som nämndes tidigare var `stdout`. Denna används för att skriva ut saker från programmet. Normalt hamnar dessa utskrifter i det terminalfönster som man startade programmet ifrån. Den enklaste funktionen för utmatning är `printf`.

`printf (` sträng`,` ... `);`

Även `printf` tar en formatsträng och ett antal variabler, dessa kombineras till den sträng som sedan skickas till `stdout`. Om strängen inte innehåller några styrkoder skrivs strängen ut utan förändring.
`printf(”Hej från C!\n”);` kommer att skriva ut texten ”Hej från C!” i terminalfönstret. `\n` är inte en styrkod utan tecknet för radbrytning.

`printf(”Ett tal: %d Tecken: %c Ord: %s”,42,'B',”ost”);`
kommer att skriva ut texten ”`Ett tal: 42 Tecken: B Ord: ost`”
`%s` kan i det här fallet vara en sträng som innehåller blanka tecken. Hela strängen kommer att skrivas ut.

I exemplet ovan ser vi också hur tecken- och sträng-konstanter skrivs i C. Tecken skrivs med `'` runt om (`'B'`) och strängar med citattecken (`”hej”`).
`stdin` och `stdout` är som sagt de två vanligaste strömmarna, men man kan även skapa egna strömmar till filhantering och andra I/O-kanaler. Filhantering återkommer vi till i en senare föreläsning.

Det finns många fler funktioner i `stdio`. På kurshemsidan finns en länk till en beskrivning av C:s standardbibliotek, där kan ni läsa mer om funktionerna för inmatning och utskrift. Där finns även alla styrkoder förtecknade.

Nu när vi vet hur man får in data från användaren är det dags att titta på cirkel-arean igen. Det var radien vi ville läsa in:

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

Och arean vi ville skriva ut:

```
printf("Area: %f\n",area);
```

Här börjar det - `main`

För att C-kompilatorn ska veta var i din källkod du vill att ditt program ska börja så finns det en väldefinierad startpunkt - `main`-funktionen.

Alla C-program måste innehålla en `main`-funktion och det är där exekveringen av programmet startar.

Ett enkelt program: Hello world! (fast på svenska)

```
main()
{
    printf("Hej från C!\n");
}
```

`main` - definitionen av `main`-funktionen
`printf` - skriver ut en rad

Det här programmet är i sin enkelhet korrekt C-kod. Dock är det inte ett program som kommer att accepteras i denna kurs. Det krävs lite mer detaljer.

```
#include<stdio.h>
int main()
{
    printf("Hej från C!\n");
    return 0;
}
```

Först måste vi tala om var `printf` finns deklarerad. Detta gör vi genom att inkludera den deklarationsfil som hör till `stdio`. Exakt vad detta innebär återkommer vi till senare i kursen, just nu räcker det att veta att det är så det ser ut.
Vi måste även tala om vad funktionen har för returtyp. I `main`-fallet är denna alltid `int`. Säger vi att vi returnerar en `int` så måste vi också göra det, därav `return 0;` Returtyp och returvärden återkommer vi till nästa föreläsning, så jag säger inte mer om det här.

Då vet vi slutligen allt som behövs för att färdigställa vårt program för att beräkna arean av en cirkel. Detta program återkommer i inlämningsuppgift 2.

En kompilator - `gcc`

Som tidigare nämnts så måste man kompilera C-koden innan den går att köra i datorn. I denna kurs förutsätts att ni jobbar med kompilatorn `gcc` på universitetets datorer. För att ni ska slippa onödiga buggar och för att tvinga på er ”god programmeringsstil” så kommer jag kräva att er källkod passerar `gcc` med några extra restriktioner.

Min rekommendation är att ni skriver källkoden i en texteditor där ni har full kontroll över vad som händer. Jag rekommenderar inte grafiska utvecklings-miljöer där mystiska saker händer under ytan, i alla fall inte för en introducerande kurs i C. Tanken är att ni ska lära er vad som händer och kunna utföra alla stegen själva - det gör ni inte om ni har ett program som tänker åt er.

Emacs har allt (och bra mycket mer) än man behöver för att ge en behaglig utvecklingsmiljö.

De restriktioner som gäller för er kod (ur `gcc`:s synvinkel) är bland annat:

- Det ska vara ANSI-C.
- Inga variabler eller funktioner får deklareras utan typ.
- Alla variabler och funktioner måste deklareras innan de används.
- Alla deklarerade variabler ska användas, gäller även argument.
- Alla funktioner måste anropas med rätt antal argument och rätt typ av argument. Detta gäller även formatsträngar i `printf` och `scanf`.
- Alla standardbibliotek som används måste inkluderas.
- Allt annat som `gcc` varnar för givet de flaggor som ni ska använda...
- Alla varningar betraktas som fel.

flaggor som ni kommer att använda till `gcc` är (åtminstone):

`-o`	Ge ett filnamn till den körbara filen. Annars heter filen `a.out`.
`-ansi`	Acceptera endast ANSI-C, inte standard C.
`-Wall`	Varna för det mesta som ser suspekt ut.
`-Wextra`	Varna för lite mer saker. En del versioner av `gcc` kallar denna `-W`.
`-Werror`	Betrakta alla varningar som fel.

För att kompilera med `gcc` befinner man sig lämpligen i ett terminalfönster. Inlämningsuppgift 1 tar er igenom de första stegen och visar i detalj vad de olika flaggorna gör för nytta.

När det gäller kodningsstandarder och utseende på koden hänvisar jag till de källkodsfiler som finns att ladda hem från kurshemsidan.