C-programmering, föreläsning 2

Jesper Wilhelmsson

- Funktioner – `void`
- Globala och lokala variabler, scope – `static`
- Arrayer
- Strängar
- ASCII, ANSI
- Argument till `main`
- Slumptal

Funktioner

Nu är det dags att börja titta på funktionsanrop i C. Funktioner och funktionsanrop känner ni igen från ML. Principen är den samma i C men alla funktioner i C måste även deklareras med en returtyp, dvs vilken sorts data som funktionen ska ge tillbaka till den som anropar den. Syntaxen för en funktionsdeklaration ser ut så här:

```
Returtyp Namn ( Argument )
{
      Lokala variabler
      Funktionskropp
      Retursats
}
```

```
int foo(float tal)
{
     int heltal;
     heltal = (int) tal;
     return heltal;
}
```

I det här fallet har vi sagt att funktionen `foo` ska returnera ett heltal (`int`). Även argumenten till en funktion måste typdeklareras. Ett anrop till `foo` kan se ut så här: `int x = foo(3.14);`

Vi har redan sett en del funktioner i inlämningsuppgifterna, framför allt den som kallas `main`. `main` är en funktion precis som alla andra, den har en returtyp (`int`) och kan ta argument. Argumenten till `main` ska se ut på ett visst sätt, vi återkommer till dessa lite senare.

`main`:s returvärde är även det lite speciellt. Tanken är ju att det är ett Unix-system som ska ta emot returvärdet från programmet. Detta göra att returvärdet måste följa de konventioner som finns i Unix. Ett programs returvärde i Unix representerar en statusflagga som talar om ifall programmet lyckades med sin uppgift eller ej. Returtypen är alltid `int`. Om ett program returnerar `0` betyder det att allt gick bra, ett returvärde skilt från `0` betyder att något gick fel. I Unix är detta allt returvärdet säger. Andra operativsystem har lite mer graderade felskalor, till exempel i AmigaOS kan felkoderna användas för att styra beteendet i script.

0: Allt är bra.
5: Mindre fel uppstod, inget som bör påverka scriptets funktion.
10: Fel uppstod. Programmets utdata (som skrivits till `stdout`) kan vara felaktig.
20: Allvarligt fel, scriptet bör avbrytas.

När man deklarerar nya funktioner bör man alltid placera dem ovanför den anropande funktionen i källkoden. Detta för att C-kompilatorn alltid läser källkoden från första till sista raden och endast de funktioner och variabler som deklarerats högre upp i filen får användas. Av denna anledning hamnar normalt `main`-funktionen sist i en källkodsfil.

Funktioner och procedurer

I en del programmeringsspråk skiljer man på funktioner och procedurer. Funktioner har alltid ett returvärde medan procedurer aldrig returnerar något. I C finns endast funktioner, men man kan ge en funktion returtypen `void`. Detta betyder att funktionen inte returnerar något värde. (void = tomrum)

Globala och lokala variabler, scope - `static`

Inuti en funktion kan man deklarera variabler. Dessa variabler är endast tillgängliga lokalt i den aktuella funktionen och vi kallar dem därför lokala variabler. Även argumenten till en funktion kan räknas till de lokala variablerna. En variabels livslängd i ett program brukar kallas scope. I C omges alltid ett scope med `{ ... }`.

Funktioner i C skrivs alltid med `{ ... }` och har därför alltid ett eget scope, men scope kan deklareras var som helst i koden i ett C-program. Till exempel såg vi redan förra föreläsningen att `{ ... }` kan användas i loopar och villkorliga satser. Det går även att påbörja ett nytt scope mitt i koden, var som helst i koden vilket kan vara användbart till exempel för att lokaldeklarera en index-variabel vid en `for`-loop. Man ska alltid sträva efter att ha så kort livslängd på sina variabler som möjligt, det underlättar för `gcc`:s optimering.

Man kan även deklarera variabler utanför funktions-scope:et. Dessa betraktas då som levande i hela programmet och kallas normalt för globala variabler. Globala variabler deklareras när programmet startar och minns sina värden genom hela programkörningen.

Globala variabler är lite farliga att använda i större program och man bör tänka efter noga innan man skapar dessa. Anledningen är att globala variabler kan användas även utanför den källkodsfil man deklarerar dem i. Därmed öppnas möjligheterna för namnkonflikter med andra filer i applikationen, oönskad användning av variabeln och eventuell felaktig uppdatering av variabeln i kod som inte borde känna till, eller åtminstone inte ändra variabeln.

Nyckelordet `static` kan användas för att minska den typen av problem. Betrakta kodexemplet nedan.

```
#include<stdio.h>

int a = 0;
static int b = 0;

void foo(void)
{
   int c = 0;
   static int d = 0;

   printf(”global: %d  static global: %d  ”
          ”lokal: %d  static lokal: %d\n",
          a, b, c, d);

   a++;
   b++;
   c++;
   d++;
}

int main(void)
{
   int i;

   for (i = 0; i < 10; i++)
   {
      foo();
   }

   return 0;
}
```

`static` kan användas både för lokala och globala variabler. Betydelsen är dock ganska olika.

Global statisk variabel

När en global variabel deklareras statisk begränsas dess åtkomst till den aktuella källkodsfilen. Detta är alltså ett sätt att skydda variabler som måste vara globala i en fil från att kod i andra filer krockar med dem. (Kan jämföras med privata instansvariabler i Java).

I kodexemplet ovan har vi två globala variabler, a och b. a är tillgänglig utifrån och dess värde kan alltså ändras utanför vår kontroll, b är statisk och därmed endast tillgänglig i vår egen kod.

Lokal statisk variabel

I exemplet har vi även två lokala variabler i funktionen `foo()`. Den ena av dessa är statisk (d). Båda kommer initialt att ha värdet 0. `foo()` anropas flera gånger och vid varje anrop ökas värdet på alla variabler med ett. Skillnaden mellan den statiska och den icke-statiska variabeln är att den statiska

kommer att minnas sitt värde mellan funktionsanropen.
Den statiska variabeln kommer alltså endast att deklareras en gång, första gången `foo()` anropas. Därefter kommer raden att ignoreras - trots att där finns en tilldelning som initierar `d` till `0`. Den icke-statiska variabeln (c) kommer att deklareras och initieras varje gång `foo()` anropas.

Vad är då skillnaden mellan en statisk lokal variabel och en global? Den lokala variabeln är endast tillgänglig inuti funktionen där den deklareras. Globala variabler är tillgängliga i hela programmet (modulo `static`...).

Så hur ser det då ut när vi kör exemplet ovan? Jo, så här:

```
global: 0  static global: 0  lokal: 0  static lokal: 0
global: 1  static global: 1  lokal: 0  static lokal: 1
global: 2  static global: 2  lokal: 0  static lokal: 2
global: 3  static global: 3  lokal: 0  static lokal: 3
global: 4  static global: 4  lokal: 0  static lokal: 4
global: 5  static global: 5  lokal: 0  static lokal: 5
global: 6  static global: 6  lokal: 0  static lokal: 6
global: 7  static global: 7  lokal: 0  static lokal: 7
global: 8  static global: 8  lokal: 0  static lokal: 8
global: 9  static global: 9  lokal: 0  static lokal: 9
```

Att initiera globala och statiska variabler till noll är egentligen onödigt eftersom alla dessa variabler initieras till noll automatiskt, men bör ändå göras för läsbarhetens skull.

Arrayer

Ni är vana från ML att använda listor till allt. I C är det lite krångligt att jobba med listor (vi ska göra det senare i denna kurs) och i de flesta enklare fallen vill man i C inte använda rekursiva datatyper (som en lista är). I stället använder man arrayer som lagringsplats för flera saker av samma slag. En array-deklaration kan se ut så här:

```
int siffror[10];
```

Arrayen vi skapar här heter `siffror` och har tio platser för att lagra heltal. Det finns flera stora skillnader mellan listor och arrayer. Bland annat typen. En array är alltid typad, och den kan bara innehålla värden av den typ man angett. I arrayen ovan kan vi alltså bara lagra saker med typen `int`.
En annan viktig skillnad mellan arrayer och listor är storleken. En array har alltid en fix storlek. Det går inte att ändra storleken på en array och man kan inte trycka in nya fält mellan två existerande fält i arrayen.

Man kan initiera en array direkt vid deklarationen genom att skriva start-värdena inom {, , , ...}:

```
int siffror[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

I det här fallet angav vi ingen storlek på arrayen vilket innebär att mängden initialvärden kommer att avgöra storleken.

```
float siffror[10] = { 3.45, 2.657, 5.54, 6.342 };
```

Här anger vi en storlek och initierar de första fyra värdena. Övriga värden i arrayen kommer att vara oinitierade. Oinitierad betyder att det kan finnas vad som helst där, C nollställer inte lokala variabler (om de inte är statiska).

I PM1/PK1 (Inlämningsuppgift 6, problem 2) skrev ni en funktion, `nth`, för att hämta det n:te elementet i en lista. Hur gör man motsvarande sak med en array? En array kan alltid indexeras direkt och man behöver aldrig räkna sig fram till någon speciell position. För att komma åt position `n` skriver man `siffror[n]`. Arrayer i C är noll-indexerade, det betyder att den första positionen i arrayen har index 0. Funktionen `nth` blir därmed trivial att skriva i C. Om det är tecken `n` eller tecken `n+1` som ska returneras är en religiös fråga.

```
int nth(int n, int arrayen[])
{
   return arrayen[n];
}
```

Ett annat problem i samma inlämningsuppgift var att skriva `largest` som returnerade det största elementet i en lista. Precis som i ML behövs det i C en loop för att göra detta. Vi använder som tidigare sagts inte rekursion utan väljer i det här fallet en `for`-loop. Anledningen till att vi väljer just en `for`-loop är en av de viktiga skillnaderna mellan listor och arrayer, nämligen storleken. Med en lista kan man snurra i loopen tills dess att man kommer fram till slutet som indikeras med en ”tom lista” eller en `NULL`-pekare av något slag. Detta låter som ett jobb för en `while`, men i en array finns inget som talar om att ett element är det sista i arrayen. Istället har arrayen en fast storlek som vi måste veta. När vi vet hur många varv vi vill snurra i loopen så är det `for` vi vill ha!

På en lokalt definierad array kan vi använda `sizeof()` för att ta reda på storleken. `sizeof()` talar om hur mycket plats arrayen tar i minnet, dvs hur många bytes den tar upp. Såvida det inte är en array av `char` så kommer antalet bytes inte att vara det samma som arrayens storlek. För att få reda på hur många positioner arrayen har delar vi därför minnesstorleken för hela arrayen med minnesstorleken för ett fält i arrayen.

```
int arr[10];
storlek = sizeof(arr) / sizeof(int);
```

Att ta reda på storleken av en array som vi fått arrayen skickad till oss via funktionsargumenten är inte helt trivialt. Argumentet som tar emot arrayen är nämligen alltid en pekare och försöker vi ta `sizeof()` på den får vi bara storleken av pekaren. I de fall där man skickar med arrayer till funktioner som är beroende av att veta storleken på dem bör man därför även skicka med storleken via argumenten.

```
int largest(int arr[], int size)
{
   int i;
   int max = arr[0];

   for (i = 1; i < size; i++)
   {
       if (arr[i] > max)
          max = arr[i];
   }

   return max;
}
```

Strängar

Strängar i C är inget annat än arrayer av tecken.

`char str[42];` - En sträng med plats för 42 tecken.

Allt som sagts tidigare om arrayer gäller även strängar med en liten skillnad: Strängar i C är `NULL`-terminerade. Detta betyder att det sista tecknet i arrayen är `'\0'` (`NULL`, tecknet med teckenkod noll) och att man alltså kan se när en sträng är slut utan att veta dess storlek.

Notationen som vi använt tidigare för arrayer blir i strängar ett sätt att komma åt enskilda tecken i strängen. `str[0]` ger det första tecknet i strängen och `str[5]` det sjätte.

Precis som alla andra datatyper kan strängar initieras direkt vid deklaration:

```
char str[] = ”Hej hopp!”;
```

Standardbiblioteket `string` innehåller en hel del funktioner för att arbeta med strängar, titta igenom detta så att ni har lite koll på vad som går att göra.

ASCII

”I ML finns en biblioteksfunktion `Char.toUpper: char -> char` *som översätter en liten bokstav till motsvarande stor (och lämnar andra tecken oförändrade). Den fungerar dock bara för små engelska bokstäver (a-z).”*

Detta känner ni igen från en uppgift i PM1/PK1. Hur skulle det se ut om vi ville skriva en sådan funktion i C? Man kan naturligtvis lösa problemet med en stor `switch`. Vi fick ett `'a'`, då returnerar vi ett `'A'`, vi fick ett `'b'`, då returnerar vi ett `'B'`... Men det är ju inte vidare snyggt.

För att kunna lösa problemet på ett bättre sätt måste vi förstå vad tecken är och hur datorn ser på dem. I datorn är allt bara tal. C som ligger mycket nära datorn är av samma åsikt, allt är bara tal. Vi kan prata om tecken och pekare och annat, men egentligen är det bara tal. Varje tecken vi kan skriva in i datorn representeras av ett tal. Mappningen mellan tecken och tal är standardiserad och återfinns i den så kallade ASCII-tabellen (se länk från kurshemsidan).

Varje tal från 0 till 255 mappas till ett tecken. De första 32 är olika typer av styrkoder som vi inte kommer att beröra i denna kurs. Bortsett från `NULL` då, första tecknet i listan som vi ju redan stött på.

Notera att alla bokstäver ligger i ordning bortsett från de tre sista i det svenska alfabetet, å, ä, ö. Dessa har lagts till i ett senare skede och hamnade då i den andra halvan av tabellen. Det kan även vara värt att notera att alla siffror ligger i nummerordning.

Så alla tecken är tal, och alla tal är för en dator bara en hög med ettor och nollor. För att få full förståelse för tecknens relationer ser vi den binära kodningen av varje tecken i tabellen. Observera skillnaden mellan stora och små bokstäver. Endast en bit skiljer. Om vi ser till att den biten inte är ett så kommer vi att ha en stor bokstav i stället för en liten. Biten i fråga har värdet 32 och vi kan därför släcka den biten genom att utföra en logisk OCH med 2-komplementet till 32, dvs det tal där alla andra bitar är ett och biten för 32 är noll.

```
'a' = 01100001
 32 = 00100000
~32 = 11011111
'a' & ~32 = 01000001 = 'A'
```

Nu skriver vi detta i C-kod:

```
#include <stdio.h>

void toUpper(char str[])
{
   int i;

   for (i = 0; str[i] != '\0'; i++)
   {
      str[i] &= ~32;
   }
}

int main(void)
{
   char text[] = "hej hopp ökenråtta!";
   toUpper(text);
   printf("%s\n",text);
   return 0;
}
```

Kommer programmet att fungera?
Testkörning:

```
bash$ toUpper
HEJ
bash$
```

Det såg inte ut som det skulle. Vad händer med mellanslaget?
Titta i ASCII-tabellen efter tecknet för mellanslag (`space`). Kan man verkligen släcka bit 32 i alla tecken hur som helst? Nej, det går förstås inte. Mellanslaget har teckenkod 32 vilket betyder att när vi släcker bit 32 blir det bara noll kvar, noll som terminerar strängen.

För att ordna en funktion som bara konverterar bokstäver måste vi lägga in ett villkor som kontrollerar att tecknet vi vill konvertera faktiskt ligger i rätt intervall (mellan `'a'` och `'z'`). Vi vill förstås också få med de svenska tecknen som vi hittar i den senare delen av ASCII-tabellen. Dessa ligger lämpligt nog på samma sätt som den första delen av alfabetet och även där kan vi förstora bokstäver genom att släcka bit 32.

```
#include <stdio.h>

void toUpper(char str[])
{
   int i;

   for (i = 0; str[i] != '\0'; i++)
   {
      if ((str[i] >= 'a' && str[i] <= 'z') ||
          (str[i] >= 'à' && str[i] <= 'ÿ'))
      str[i] &= ~32;
   }
}

int main(void)
{
   char text[] = "hej hopp ökenråtta!";
   toUpper(text);
   printf("%s\n",text);
   return 0;
}
```

ANSI

ANSI – American National Standards Institute satte för många år sedan ihop en lista över koder för att styra terminaler. Tanken var att en server skulle kunna styra terminalen hos de klienter som kopplade upp sig mot den. Detta sätt att styra fjärr-terminaler blev mycket vanligt i bland annat BBS:er eftersom det till exempel möjliggjorde färggrafik i textläge.

Varje kod är en sekvens av tecken som följer ett speciellt mönster. Alla koder inleds med teckenkoden för escape och koderna kallas därför ibland för escape-koder eller escape-sekvenser. Koderna kan användas till det mesta som behövs för att ha kontroll över en terminal. Man kan flytta markören, ändra färger, rulla texten upp eller ner, rensa skärmen och mycket mycket mer.

Ni kommer att stöta på ANSI-koderna i några av inlämningsuppgifterna i denna kurs. Även om koderna på sätt och vis har spelat ut sin roll nu när Internet tagit över efter BBS:erna så tycker jag att det tillhör allmänbildning för en programmerare att veta hur dessa koder används.

En förteckning över de vanligaste koderna finns på kurshemsidan.

Argument till `main`

Vi har nu sett hur olika funktioner kan ta argument av olika slag. Som vi noterat tidigare är `main` en funktion som alla andra och även den kan ta argument. Argumenttyp och ordning bestäms av anropskonventionerna i Unix eftersom C har sina rötter i den världen. Argumenten skickas med från shellet där programmet startas och består av en array av strängar.

```
int main(int argc, char* argv[])
```

Det första argumentet, `argc`, är ett heltal och talar om hur många argument som skickades med från shellet. Det andra argumentet till `main`, `argv`, är arrayen av alla text-strängar som skickades med när programmet startades.

```
bash$ argtest hello world 42
```

Anropet ovan har tre argument, `"hello"`, `"world"` och `"42"`. (`"argtest"` är namnet på programmet.) Inne i C kommer vi att få tillgång till både programnamn och argument i `argv`:

```
args = 3
argv[0] = "argtest"
argv[1] = "hello"
argv[2] = "world"
argv[3] = "42"
```

Notera speciellt att även tal som skickas in som argument kommer att komma fram som strängar. För att konvertera tillbaka dem till tal finns en användbar funktion i `stdlib`, `int atoi(char*)`.

Slumptal

Inte för att det är specifikt för C, utan mer för att det ingår i inlämnings-uppgifterna. Slumptal behövs i många sammanhang och här följer en beskrivning på hur man kommer åt dem i C.

Funktionen för att hämta slumptal finns i `stdlib` och heter `rand()`. `rand()` returnerar ett heltal mellan (och inklusive) `0` och `RAND_MAX`. `RAND_MAX` är ett hyfsat stort tal, C definierar det till minst 32767 (maxvärdet för ett 16-bitars signerat tal).

Normalt när man vill ha slumptal har man dock sin egen maxgräns och för att anpassa talet vi får av `rand()` till denna vill vi dela resultatet med `RAND_MAX`. På det sättet får vi ett slumptal mellan 0 och 1 som vi sedan kan multiplicera med den övre gränsen. Eftersom `rand()` inkluderar båda gränserna (`0` och

RAND_MAX) vill vi egentligen dela med ett tal som är lite större än RAND_MAX, detta för att garantera att vårt nya slumptal alltid är mindre än 1.

Nästa detalj vi måste tänka på är att om vi delar två heltal så kommer C att använda heltalsdivision, dvs resultatet kommer att trunkeras till ett heltal som i det här fallet alltid blir 0. För att tvinga fram flyttalsdivision måste minst en av täljare och nämnare vara flyttal.

När allt är klart gör vi om resultatet till en `int` igen.

```
slump = (int)(((rand()/((double)RAND_MAX) + 1)) * max);
```

En annan variant av slumptalsberäkningen har både en max- och en min-gräns:

```
slump = (int)(((rand()/((double)RAND_MAX) + 1)) *
              (max – min)) + min;
```

För att skapa slumptal måste C initiera sin slumptalsmotor. Detta görs med funktionen `srand(int)`. Vill man ha samma slumptalsföljd varje gång man kör sitt program initierar man med samma tal varje gång. Oftast vill man dock ha olika slumptalsföljder från körning till körning. För att åstadkomma det hämtar vi ett slumpfrö från systemklockan:

```
srand(time(NULL));
```