Kapitel 1
Programspråket C och standard-C biblioteket

Inledning

I detta inledande kapitel ger vi en snabb introduktion till programspråket och det viktiga programbiblioteket. Men innan du fortsätter måste du installera kursens program-utvecklingsverktyg. Du hittar dom och instruktioner om hur du installerar, via kursens hemsida.

1.1 Kort historik kring C

Ursprungligen, dvs. på 1960-talet, ville **Ken Thompson**, på *Bell Labs*, skapa ett programmeringsspråk för det helt nya operativsystemet UNIX. Thompson modifierade därför det typlösa språket *BCPL (Basic Combined Programming Language)* ursprungligen avsett som implementeringsspråk för *kompilatorer*, och skapade **B**. I den vid denna tid dominerande minidatorn PDP-11, hade B-kompilatorn svårt att utnyttja maskinens prestanda och programmen blev därför hopplöst långsamma. Det implementerades därför inte speciellt många applikationsprogram i B. Det kom att leda fram till att **Dennis Ritchie**, också han vid Bell Labs, förbättrade B och då skapade dess efterföljare, programspråket **C**.

Den långa utvecklingen av C startar nu i tidigt 1970-tal. I *The Development of the C Language* skriver Ritchie: "C utformades i början av 1970-talet som ett systemimplementeringsspråk för det nya operativsystemet UNIX. Med grunder från B, adderades också en *typstruktur*; lämplig för en liten maskin. Avsikten var att åstadkomma ett verktyg för att förbättra en undermånlig programmeringsmiljö." Utvecklingen av C kom därför att bli grunden för UNIX. Ritchie fortsätter: "I början av 1973 var det väsentliga i modern C färdigt. Både språk och kompilator var tillräckligt kraftfulla för att tillåta oss att skriva om Unix-kärnan för PDP-11. Vi skrev om hela kärnan från PDP-assembler till C under sommaren det året."

1978 publicerade **Brian Kernighan** tillsammans med Ritchie *The C Programming Language*, som skulle fungera som språkreferens tills en formell standard antogs. Kernighan bidrog speciellt till C med den specifika kunskap han hade om operativsystem. Kombinationen av operativsystem och dess effektiva implementering gav samtidigt kombinationen Kernighan/Ritchie, eller som vi i dagligt tal säger K/R-C.

C har funnit många användningsområden under åren, åtskilliga operativsystem (realtidssystem) är skrivna i C att använda i program på maskinnära nivå, som kärnor. Men C används också för många applikationer, allt från enhetsdrivrutiner till kompilatorer och tolkar för andra programmeringsspråk, exemplelvis *Python* och *Perl*. Även databaser som *Oracle Database*, *MySQL, MS SQL Server* och *PostgresSQL* är åtminstone delvis skrivna i C. C är grunden för många system kärnor. Andra programmeringsspråk, som, använder kompilatorer eller tolkar som är skrivna i C. C har också banat väg för en rad nya programspråk, vart och ett med sin egen specielitet, exempelvis *C++*, *Objective-C*, *C#.*

1983 bildade *American National Standard Institute* (ANSI) kommittén, X3J11, för att fastställa en formell standard för C. C-standarden ratificerades som ANSI X3.159-1989 *Programming Language C*. Detta var den första formella standarden för C och går ofta under benämningen ANSI-C eller C89. Nya standarder har sedan föreslagits, diskuterats och ratificerats. Eftersom existerande programvarubas i C är både stor och central i viktiga system är bakåtkompatibiliteten tungt vägande även vid arbete med förändringar i standarden. Detta gör C i grunden ett mycket stabilt programspråk och att investeringar i C alltid kommer att löna sig även på lång sikt.

Källor och referenser:

The Development of the C Language https://www.bell-labs.com/usr/dmr/www/chist.pdf
Spännande och ingående beskrivning, av programspråkets upphovsman Dennis Ritchie.

Wikipedia: *C (programming language)* https://en.wikipedia.org/wiki/C_(programming_language)
är en bredare samtidigt mer kortfattad historik kring programspråket.

Wikipedia: *Embedded software* https://en.wikipedia.org/wiki/Embedded_software
Inbyggda system är ett viktigt delområde för maskinorienterad programmering. Här har C en framträdande roll.

1.2 C-programmets struktur

C är ett *procedurellt programspråk*, menandes att programmets tillstånd kan förändras genom utförandet av någon *procedur*. En procedur är i detta sammanhang helt enkelt en *subrutin*, eller som i C, en *funktion*. Speciellt måste det, i varje C-program, finns exakt *en* funktion med namnet `main`, där programmet alltid startas. Ett C-program kan alltså som minst innehålla enbart denna funktion.

Utöver funktioner tillhandahålls också *variabler*, som ett sätt att representera programmets tillstånd. Variabler kan representera väldigt olika former av data, i C använder man därför begreppet *typ* för att skilja mellan olika betydelser hos någon specifik dataenhet. Det finns fyra grundläggande datatyper:

- `int` (*integer*) avsett att kunna representera ett heltal. För heltalstyper kan modifierare `short` och `long` anges. Standarden anger den minsta tillåtna representationen (antal bitar) för varje typ. Dessutom kan modifierare `unsigned` och `signed` användas då man specifikt vill ange naturliga tal (tal utan tecken) eller heltal (tal med tecken).
- `char` (*character*) avsett att kunna representera den typ av tecken som används i skrift. I praktiken är det en heltalstyp där de olika heltalsvärdena kan vara kodade till ett skrifttecken, Det finns flera olika kodningar, (teckenuppsättningar) där den för oss vanligaste är *Unicode-8*, och där *ASCII*-representationen är en delmängd.
- `float` (*floating point*) flyttalsrepresentation, enkel precision (32 bitar), ett standardiserat format speciellt utformat för att representera ett tal på formen *teckenbit*, *exponent*, *mantissa*. Representationen gör det möjligt att representera mycket små och mycket stora tal med en begränsad ordlängd.
- `double` (*double precision floating point*), som `float`, en standardiserad flyttalsrepresentation men med större precision (64 bitar).

EXEMPEL 1.1 DEKLARATION AV GLOBALA VARIABLER

Då följande deklarationer gjorts kan variablerna refereras (användas) av samtliga funktioner i programmet:

```
char   c;           /* plats för ett tecken från teckenuppsättningen */
int     i           /* plats för ett heltal -MIN..MAX */
unsigned int ui;    /* plats för ett naturligt heltal 0..MAX */
float  f;           /* plats för ett 32-bitars flyttal */
double d;           /* plats för ett 64-bitars flyttal */
```

EXEMPEL 1.2 DEKLARATION OCH INITIERING VARIABLER

Variabler kan ges initialvärden i samband med deklaration:

```
char   c='A';
int    i=10;
float  f = 2.5e-6;
```

En variabel som inte tilldelats initialvärde initieras till 0 som förberedelse, innan huvudprogrammet `main` startas.

Ytterligare typer som organiserar data är såväl *arrays* (på svenska *fält*, dvs.vektorer, matriser etc.) som *structures* (sammansatta heterogena typer). Ett fält består av flera variabler av samma typ medan den sammansatta typen kan sätta samman flera olika underliggande typer. Till detta kommer den viktiga pekar-typen, som kan innehålla adressen till såväl variabler som funktioner. Vi kommer att behandla var och en av dessa olika typer under kursens gång men vi börjar med ett enkelt specialfall av ett fält.

En *string* används för att representera en textsträng. De tecken som ingår i textsträngen tolkas då typiskt enligt den använda teckenuppsättningen. En textsträng avslutas alltid med det binära värdet 0.

EXEMPEL 1.3 DEKLARATION AV TEXTSTRÄNG

En textsträng kan deklareras som ett fält av `char`:

```
char Hello[]= {'H','e','l','l','o',' ','w','o','r','l','d',0} ;
```

Fältet består av 12 tecken, vardera med typen `char`. I C kan samma textsträng deklareras enklare och tydligare med en strängtyp:

```
char Hello[]= "Hello world";
```

där citationstecknen talar om att symbolen `Hello` används som en textsträng. Vid kompileringen läggs nu automatiskt tecknet för strängslut (0) till strängen och dess storlek blir därför den samma som med den första deklarationen.

Ett C-program byggs upp utav minst en, men vanligtvis betydligt fler *funktioner*. En funktion är en programdel som kan anropas med eller utan parametrar. Funktionen kan, men måste inte, ge ett returvärde.

EXEMPEL 1.4 FUNKTIONSDEKLARATION

Huvudprogrammet i ett C-program, som förbereds och anropas via operativsystemet, heter av konvention `main`, och deklareras på följande sätt:

```
int main(int argc, char **argv);
```

Huvudprogrammet har alltså två parametrar där den första (`argc`, *argument count*) anger hur många argument som följer och den andra parametern (`argv`, *argument vector*) innehåller ett fält med pekare till de textsträngar som skickas som argument till programmet. Deklarationen säger samtidigt att returvärdet från `main` är av typen `int`.

Typiskt har då det använda operativsystemet förberett parametrarna inför anropet, det returnerade värdet utgör någon form av statuskod som operativsystemet sedan kan behandla på något sätt.

Men funktioner behöver naturligtvis inte ha vare sig parametrar eller returnera något värde. För att ange detta använder man det reserverade ordet `void` (ungefär *tom*).

I en miljö som saknar operativsystem, exempelvis ett mindre inbyggt system, tillåter standarden en godtycklig deklaration av huvudprogrammet, exempelvis kan vi då deklarera huvudprogrammet enligt:

```
void main(void);
```

med betydelsen att inga parametrar skickas till funktionen, inte heller kommer eventuella returvärden att tas om hand.

Läs mer om C-funktioner och hur de används:
https://www.programiz.com/c-programming/c-functions

Freestanding and hosted implementations https://en.cppreference.com/w/cpp/freestanding
kan läsas för förståelse av definitionsmässig skillnad mellan värddatorns C-miljö och den fristående miljön för målsystemet vid korsutveckling.

1.3 Enkel in- och utmatning

Med in- och utmatning menar vi att data på något sätt tillställs (ges till) systemet, respektive sänds ut från systemet, typiskt via någon form av *periferikrets*. Periferikretsen gör då en anpassning mellan data (på digital form) och den representation som data ges i sin omgivning. Som exempel på en sådan periferikrets kan vi ta en seriekommunikationskrets som används för att kommunicera alfanumeriska tecken från någon form av tangentbord, respektive till någon form av bildskärm.

I C's standardbibliotek finns det färdiga funktioner för in- och utmatning av sådana tecken. Kombinationen av tangentbord och bildskärm kallar vi då ofta *terminal* (*konsoll*). Vi säger att tecken som skrivs på terminalens tangentbord matas in till systemet medan utmatning från systemet hamnar på terminalens bildskärm.

För utmatning kan vi använda C-funktionen `printf` (*print formatted*).

EXEMPEL 1.5 UTMATNING

Följande program skriver en texsträng (hälsning) till din terminal:

```
#include <stdio.h>
int main(int argc, char **argv)
{
   printf("hello world\n");
   return 0;
}
```

Samma program kan utformas som:

```
void main( void )
{
   printf(”Hello world\n”);
}
```

Även denna version genererar exekverbar kod och är alltså korrekt i detta avseende. Samtidigt genereras varningsutskrifter, såväl om funktionen `main` som `printf`.

Med ”formatterad utskrift” menas att exempelvis värdet av en variabel översätts till det textformat bildskärmen kräver. För detta används *specifierare* som instruerar `printf` om vilken typ av översättning som då måste göras.

EXEMPEL 1.6 FORMATTERAD UTMATNING

Följande program skriver en texsträng (hälsning) till din terminal:

```
#include <stdio.h>
int variable;
void main( void )
{
   printf(”Variable value is: %d\n”, variable);
}
```

Denna gång har vi två parametrar till `printf`, den första, en textsträng, kallas ofta *formatsträng*. I denna har tecknet `%` speciell betydelse eftersom det anger att det följer ytterligare en parameter. I detta fall anger `%d` att denna parameter, `variable`, ska formatteras som ett decimalt tal.

Det finns väldigt många möjligheter med `printf`, det är lätt hitta information på Internet. Vill du veta mer just nu kan du alltid börja här:

https://www.codingunit.com/printf-format-specifiers-format-conversions-and-formatted-output

Man kan på liknande sätt läsa in text från terminalen och samtidigt konvertera texten till en avsedd datatyp.

EXEMPEL 1.7 FORMATTERAD INMATNING AV TEXTSTRÄNG

I detta program läses först en textsträng från terminalen (tangentbordet). Inmatningen avslutas med radslut (<Enter>), `scanf` terminerar då och har placerat den inmatade textsträngen i variabeln `Line`. Därefter skrivs den inmatade strängen tillbaks till terminalen via `printf`.

```
#include <stdio.h>
char Line[32];
int main( void )
{
   scanf( "%s", Line );
   printf(”Text was %s\n”, Line);
}
```

Observera:

- Det sker inte någon kontroll av textsträngens längd, om den är större än de 32 tecken som ryms i denna variabel, är resultatet oförutsägbart.

Eftersom `scanf` och `printf` delar specificerare för format kan man göra olika typer av inläsningar och samtidigt låta dessa konverteras från textformat till någon grundläggande datatyp.

EXEMPEL 1.8 FORMATTERAD INMATNING AV DECIMALTAL

Med följande kodsekvens:

- Skrivs först en instruktion till terminalen
- därefter läser `scanf` in en textsträng, avslutad med <Enter>, textsträngen översätts till ett decimaltal som placeras i variabeln `n`
- slutligen används en `printf` sats som en kontroll på att inläsningen blev riktig.

```
int n;
printf("Enter the number of terms\n");
scanf("%d",&n);
printf(”Number of terms was %d\n”, n);
```

Observera:

- Om `scanf` inte lyckas översätta den inmatade textsträngen som ett decimalt tal enligt formatsträngen, kommer också resultatet i `n` att vara odefinierat och oanvändbart.

Har du aldrig tidigare programmerat i C kan följande läsning vara en översiktlig introduktion:

Reserverade ord: https://www.programiz.com/c-programming/c-keywords-identifier

Variabler och konstanter: https://www.programiz.com/c-programming/c-variables-constants

Grundläggande datatyper: https://www.programiz.com/c-programming/c-data-types

In- och utmatning: https://www.programiz.com/c-programming/c-input-output

Kommentarer: https://www.programiz.com/c-programming/comments

Operatorer: https://www.programiz.com/c-programming/c-operators

Introducerande exempel: https://www.programiz.com/c-programming/c-introduction-examples

1.4 Villkorlig programexekvering

Varje uttryck i C kan också betraktas som ett *sanningsuttryck*, dvs. ingå i tester för villkorliga satser. Då ett uttryck evalueras till 0 sägs det vara *falskt*, i alla övriga fall är det *sant*.

Programmets flöde kan styras vilkorligt med flera olika konstruktioner:

En villkorlig väg:

`if( sanningsuttryck) { }`

satser inom hakparenteser utförs om sanningsuttrycket är skilt från 0.

Villkorligt tvåvägsval:

`if(sanningsuttryck){}else{}`

satser för `if` *eller* `else` utförs baserat på sanningsuttrycket.

Anm: Hakparenteser är endast nödvändiga då fler än 1 sats ska utföras inom villkorsuttrycket. Villkorliga satser kan också vara tomma.

EXEMPEL 1.9 SANNINGSUTTRYCK

`if( a = b )`... Värdet i `b` kopieras till `a`, om detta är skilt från 0 är if-villkoret uppfyllt. Eftersom en tilldelningssats samtidigt är ett uttryck kan också en sådan användas som sanningsuttryck. Denna konstruktion är alltså tillåten men man bör absolut undvika den. Skillnaden mot konstruktionen:

`if( a == b )` ... är syntaktiskt liten men betydelsen är helt annorlunda.

`if( a && b )` ... är exempel på en logikoperation (logiskt AND) där uttrycket är sant om både `a` och `b` är skilda från 0.

`if( a & b )`.. är exempel på en bitvis operation, uttrycket är sant om minst en bit i `a` och `b` är identisk.

EXEMPEL 1.10 KONTROLL AV FORMATTERAD INMATNING AV DECIMALTAL

Man kan utföra en enklare kontroll för att se om det inmatade värdet motsvarar den förväntade typen. Returvärdet från `scanf` anger nämligen det antal argument som korrekt konverterats.

```
printf("Enter the number of terms\n");
if( scanf("%d",&n) == 1)
 printf(”Number of terms was %d\n”, n);
else
 printf(”Invalid input, decimal value expected\n”, n);
```

Är du osäker på användningen av if/else kan du läsa mer här:
https://www.programiz.com/c-programming/c-if-else-statement

Flervägsval kan sättas samman av upprepade `if/else`. Följande konstruktion illustrerar hur ett godtyckligt flervägsval kan utföras. Om inget av sanningsuttrycken är skilt från 0 kommer den avslutande `else`-grenen att utföras.

```
if(sanningsuttryck){}
else if{sanningsuttryck}
else if{sanningsuttryck}
.....
else{}
```

Som alternativ till upprepad `if/else`-konstruktion kan ofta `switch/case` användas. I konstruktionen används dessutom ofta `break` och `continue`:

```
switch( uttryck )
{
   case X:...   /* Om uttryck är X ... */
   case Y:...   /* Om uttryck är Y ... */
   case Z:...   /* Om uttryck är Z ... */
   default:... /* Om uttrycket inte är något av case-satserna */
}
```

EXEMPEL 1.11 ENKEL KALKYLATOR

```
#include <stdio.h>
int bin_op( int left, char oper, int right )
{
   int return_value;
   switch( oper )
   {
      case '+': return_value = left+right; break;
      case '-': return_value = left-right; break;
      case '*': return_value = left*right; break;
      case '/': return_value = left/right; break;
      case '%': return_value = left%right; break;
      default: printf("Invalid operator\n"); return 0;
   }
   return return_value;
}
int main( void )
{
   int rv, left, right;
   char oper;

   printf("Enter expression\n");

   if ( scanf( "%d%c%d", &left, &oper, &right ) == 3 )
   {
      rv = bin_op ( left, oper, right );
      printf("Expression result is: %d", rv);
   }else
      printf("Invalid input\n");
   return 0;
}
```

Är du osäker på användningen av `switch/case` kan du läsa mer här:
https://www.programiz.com/c-programming/c-switch-case-statement

1.5 Iterationer

En `for`-iteration (”`for`-loop”) kombineras operationerna:

- initiering inför iteration
- test om villkoret för iterationen är uppfyllt
- uppdatering mellan varje iteration

```
for( initiering ; test ; uppdatering ) { satser }
```

EXEMPEL 1.12 FOR-ITERATION

Enkel `for`-konstruktion för en uppgift som ska utföras 10 gånger.

```
int i;
for( i = 1 ; i < 11 ; i = i+1 )
{
  /* Under förutsättning att inte variabeln i ändras här utförs
     satserna mellan hakparenteserna 10 gånger */
}
```

Ett eller flera uttryck i `for`-konstruktionen kan utelämnas.

EXEMPEL 1.13 FOR-KONSTRUKTIONER

I denna konstruktion:

```
for( ; i < 11 ; i = i+1 )
```

har initieringsfältet lämnats tomt. Det förutsätts då att initieringen av `i` gjorts tidigare.

Vi kan flytta testen av iterationsvillkoret till själva iterations-slingan:

```
for( i=1; ; i = i+1 )
{
  ...
  if( i > 10 ) break;
  ..
}
```

Även uppdateringsfältet kan lämnas tomt och exempelvis ersättas i iterationsslingan:

```
for( i=1; i < 11 ;)
{
  ...
  i = i+n;
}
```

Fler fält kan lämnas tomma. Med samtliga fält tomma skapas en iteration som bara kan avbrytas med en `break`, `goto` eller `return`-sats:

```
for(; ;) {    /* “Oändlig loop” */  }
```

Läs mer om `for`-iteration:
https://www.programiz.com/c-programming/c-for-loop

EXEMPEL 1.14 IMPLEMENTERING AV `strlen`

Standardfunktionen `strlen` används för att bestämma längden av en textsträng, läsa om den här:
https://www.tutorialspoint.com/c_standard_library/c_function_strlen.htm
Här visas en möjlig implementering av standardfunktionen.

```
int strlen(const char str[])
{
   int i;
   for( i = 0; ; i = i+1 )
   {
      if( str[i] == '\0' )
         return i;
   }
}
```

EXEMPEL 1.15 IMPLEMENTERING AV `strcmp`

Standardfunktionen `strcmp` används för att jämföra två textsträngar, du kan läsa om den här:
https://www.tutorialspoint.com/c_standard_library/c_function_strcmp.htm
Här visas en möjlig implementering av standardfunktionen.

```
int strcmp(const char str1[], const char str2[])
{
   int i;
   for( i = 0; ; i = i+1 )
   {
      if( str1[i] != str2[i] )
         return (int) (str1[i] - str2[i] );
      if( str1[i] == '\0' )
         return 0;
   }
}
```

EXEMPEL 1.16 TEST AV FUNKTIONERNA `strlen` OCH `strcmp`

Följande program visar hur funktionerna kan provas:

```
int main(void)
{
   char str[128];
   for(;;)
   {
      scanf( "%s", str);
      if (strcmp( str, "exit" ) == 0 )
         return 0;
      if (strcmp( str, "Kalle" ) )
         printf("differ\n");
      else
         printf("Ok\n");
      printf("Length was %d\n", strlen( str ));
   }
}
```

Du kan läsa mer om standardfunktioner för hantering av textsträngar här:
https://www.tutorialspoint.com/c_standard_library/string_h.htm

Iterationerna `while` och `do-while` är ett annat sätt att skapa programslingor.

`while(`*`sanningsuttryck`*`){ `*`satser`*` }`

`satser` utförs om sanningsuttryck är skilt från 0.

`do{ `*`satser`*` } while (`*`sanningsuttryck`*`)`

Med `while`-konstruktionen testas villkoret först, medan `do-while`-konstruktionen testas efter den första iterationen och alltså alltid utför satser minst en gång.

Genom att ange ett sanningsuttryck som aldrig är falskt skapas en iteration som bara kan avbrytas med en `break`, `goto` eller `return`-sats:

```
while( 1 ) { /* "Oändlig loop" */ }
```

eller

```
do { /* "Oändlig loop" */ } while( 1 );
```

Läs mer om `while och do-while` iterationer:
https://www.programiz.com/c-programming/c-do-while-loops

EXEMPEL 1.17 ALTERNATIV IMPLEMENTERING AV `strlen`

Här visas en möjlig implementering av standardfunktionen.

```
int strlen(const char str[])
{
   int i = 0;
   while( str[i] )
   {
      i = i+1;
   }
   return i;
}
```

EXEMPEL 1.18 IMPLEMENTERING AV `strchr`

Läs om standardfunktionen `strchr`:
https://www.tutorialspoint.com/c_standard_library/c_function_strchr.htm

Den kan implementeras på följande sätt:

```
char *strchr(char s[], char c)
{
  int i = 0;
  do {
    if (s[i] == c)
      return (s+i);
    i = i+1;
  } while ( s[i-1] != 0);
  return 0;
}
```