IN2140 V21 - Introduksjon til C-programmering

Joakim Misund

Department of Informatics, UiO

Oversikt

Hello World!

Datatyper og Variable

Operatorer

Program-flyt

Funksjoner

Oppdeling av programmer

Pre-prosessering

Biblioteksfunksjoner

Disclaimer

Denne presentasjonen prøver ikke å være helt korrekt når det kommer til akseptert terminologi eller detaljerte regler. Den prøver å gi en generell introduksjon til C og tenkemønsktre man trenger for å mestre C-programmering.
Hvis det er noe som ser ut til å være feil prøv å forklar hvorfor og si ifra til oss (feks. Joakim på joakimmi@ifi.uio.no).

Hello World!

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
  printf("Hello World!\n");
}
```

Et enkelt program skrevet i C ser slik ut. Dette programmet kan skrives i en fil med endelsen .c og kompileres til et kjørbart program. Alle programmer har en main-funksjon som tar i mot kommandolinje-argumenter.

Hello World!

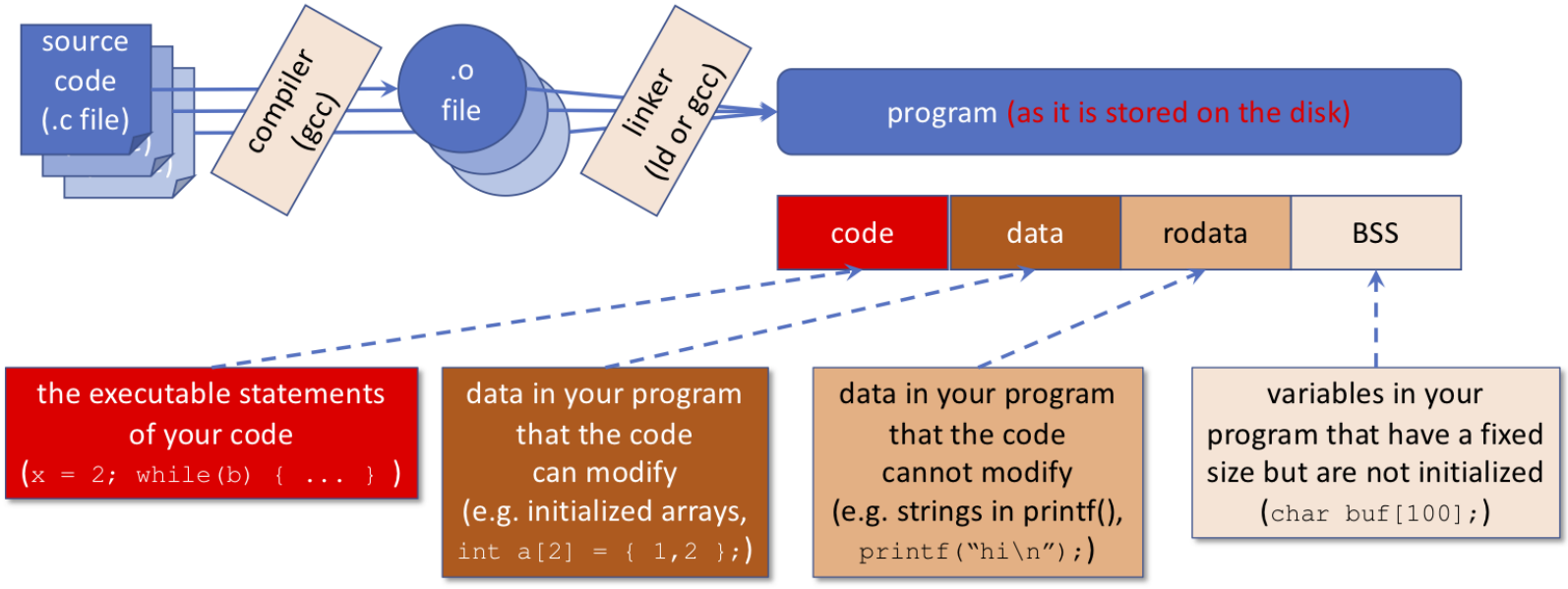
La oss si at programmet er lagret i hello_world.c. Da kan det kompileres ved å kjøre gcc (kompilator) på filen.

```
username> gcc hello_world.c -o hello_world
```

Da vil man få en ny fil som er kjørbare og som heter hello_world. Den kan så kjøres fra terminalen, og output skrives ut

```
username> ./hello_world
Hello World!
```

Hello World! - Kompilering



Makefiler

En makefile er en slags versjonskontroll for kilde- og kjørbarefiler. Et eksempel som fungere på hello_world.c er vist under.

```
hello_world: hello_world.c
	gcc hello_world.c -o hello_world
```

Hvis man kjører kommandoen make i terminalen vil følgende skje. Hvis hello_world ikke eksisterer eller hello_world.c har forandret seg siden siste hello_world ble laget vil kommandoen kjøres og hello_world.c kompileres igjen. Ellers gjøres ingen ting. Merk at det MÅ være en tab mellom starten av linjen for kommandoen og kommandoen.

Variable

En variable er et navn på et minneområde som har plass til datatypen til variabelen.
En variable har en datatype som sier hva variabelen kan lagre; hvilke verdier den kan settes til å ha.
Alle variable som er definert i en funksjon kalles lokale, og de er lagret på et minneområdet vi kaller stacken.
Alle variable som er definert utenfor en funksjon kalles globale, og de er lagret i et eget minneområde (trenger ikke mer detaljer om det).

Variabel

```
int age = 10; // Global variabel

int sum(int a, int b)
{
  int s = a + b; // Local variabel
  return s
}
```

Variabelen s kan ikke brukes utenfor funksjonen sum

Variabel

Det er viktig å skille mellom det å definere og det å initialisere en variabel. Når en variabel er definert er det satt av plass til variablen, men verdien den har er tilfeldig (sett fra programmets perspektiv). For å forsikre seg om at en variable har den verdien kan forventer må den initialiseres (settes til en verdi)

```
int age; // Definert, ikke initialisert
// age har ukjent verdi her
age = 90; // Er nå initialisert

short height = 1.63;
// Definert og initialisert
```

Enkle datatyper - Heltall

Vi har følgende datatyper for heltall

- char : 8-bit. Brukes også for ascii-tegn
- short : 16-bit
- int : Vanligvis 32-bit
- long : Vanligvis 64-bit

Hver heltallstype kan være

- *unsigned* : Kun positive verdier
- *signed* : Både positive og negative verdier.

Hvis ikke spesifisert er typen signed.

Enkle datatyper - Heltall

```
char initial_letter = 'a'; // ascii-verdien
char string_termination = 0;
unsigned short age = 16;
int year = 2021;
unsigned long population = 7000000000;
```

Variable som er definert nærme hverandre i koden har ofte addresser som er nærme hverandre.

Enkle datatyper - Flyttall

Vi har følgende datatyper for flyttall

- float : 32-bit
- double : 64-bit

Enkle datatyper - Flyttall

```
float height = 1.82;
double pi = 3.14159265359;
```

Enkle datatyper - Casting

En verdi med en gitt datatype kan tolkes som en annen datatype gjennom det vi kaller *casting*. Når man caster fra en større datatype til en mindre kan man miste informasjon. Casting fra flyttall til heltall kan også føre til tap av informasjon.

Enkle datatyper - Overflow og underflow

Når man prøver å sette en variable til en verdi som er for stor eller for liten for datatypen til variabelen vil man få henholdsvis overflow eller underflow.
For eksempel hvis man har en variabel av typen unsigned short med verdien 65535, og du legger til 1 vil verdien bli 0. Prøv det.

Array

En array er en sekvens av elementer, med en felles datatype, som ligger etter hverandre i minnet.
En array kan ikke forandre størrelse. Størrelsen på en array lagres ikke noe sted automatisk, så man må holde styr på de selv.

Array

```
int ages[5];
ages[0] = 10;
ages[1] = 78;

int bytes = sizeof(ages); // 5 * sizeof(int)

float heights[2] = {5.6, 5.3};
```

Antar vi at int er 4 bytes (32-bit) vet vi at ages[1] ligger 4 bytes etter ages[0] i minnet. Vi vet også at variabelen bytes vil inneholde heltallet 20.
Hvis ages er en peker istedenfor en stack-allokert array vil sizeof gi størrelsen på addressen, ikke minneområdet!

Struct

En struct er en samling av variable. Man kan tenke på en struct som en klasse uten metoder. Det er ikke mulig å ha *inheritance*, men structer kan nestes.
Man er ikke garantert at variablene ligger rett etter hverandre i minnet med mindre man tvinger kompilatoren til å garantere det.

Struct

```
struct person {
  int age;
  float height;
};


struct person ole;
ole.age = 33;
ole.height = 1.88;

struct person line = { .age = 32, .height = 1.92 };
```

Strenger

Det finnes ikke en datatype for strenger i C!
En streng er en sekvens med characters som er terminert med verdien 0.
Et minneområde hvor hvert element er av typen char
inneholder en streng hvis ett av elementene har verdien 0.

Strenger

```c
char name[10];
name[0] = 'o';
name[1] = 'l';
name[2] = 'e';
name[3] = 0; // name[3] = '\0';

char type[10] = "car";
```

Minneområdet må ha nok plass til karakterene i strenger og den null-terminerende karakteren. Merk at strlen (som dere skal bruke senere) ikke inkluderer den null-terminerende karakteren.

Peker

En peker innholder en minneadresse. I tillegg kan en peker har en datatype som sier hvordan minnet på minneadressen skal tolkes.
Husk at alle variable ligger på en plass i minnet. Hvor stor den plassen er avhenger av datatypen til variablen.

Peker - Hente addresse

```c
char letter = 'a';

// Hent ut addressen til en variabel med &

// Peker uten datatype
void *ptr_vl = &letter;

// Peker med datatype
char *ptr_cl = &letter;

// Peker med annen datatype
int *ptr_il = &letter;
```

Hvilken datatype man gir en peker har ingen ting å si for verdien pekeren har!

Peker - Bruke peker

Hvilken datatype man gir en peker er viktig når man skal bruke pekeren til å hente ut og sette verdier der hvor pekeren peker.

```
char letter = 'a';

// Bruk en peker med *, dereference

void *ptr_vl = &letter;
char *ptr_cl = &letter;
int *ptr_il = &letter;

*ptr_vl = 'b'; // Ikke lov, datatype void
*ptr_cl = 'b'; // Helt fint
*ptr_il = 'b'; // Helt fint, ascii-verdi
```

Peker - Bruke peker

```c
char letter = 'a';

// Sett ptr_l til minneaddressen til letter
char *ptr_l = &letter;

// Sett verdien til letter vha. pekeren
*ptr_l = 'b';

// Bruk av letter vil gi 'b'
char tmp = letter; // tmp = 'b'
```

Array og peker

En array kan brukes på samme måte som en peker, men man kan ikke forandre minneadressen til en array!

```
char name[10] = "mole";

// Lagrer addr til name i ptr_name
char *ptr_name = name;

// Setter første verdi til 'D'
*ptr_name = 'D';

// Setter andre bokstav til 'i'
*(ptr_name + 1) = 'i';
```

Pekere og struct

En peker kan ha en struct datatype.

```
struct person {
  int age;
  float height;
};

struct person ole;

struct person *ptr_ole = &ole;
(*ptr_ole).age = 19; // dereference og set
ptr_ole->height = 1.75; // Korthåndsnotasjon
```

Pekere og heap-minne

Man kan allokere minne ved bruk av funksjonen malloc. Malloc returnerer en peker som kan legges i en variabel. Hvis pekeren har verdien NULL har det skjedd en feil.

```
char *streng = malloc(10);
if (!streng) {
  // Feil
}
streng[0] = 'V'
streng[1] = 'a'
streng[2] = 'r'
streng[3] = '\0' // Null-terminert

// sizeof(streng) gir ikke størrelsen
// på minneområdet, men størrelsen på
// pekeren
```

Operatorer

Vi har tre kategoriere med operatorer

- Logiske
- Sammenligning
- Binære

Logiske operatorer

Det finnes ikke en egen datatype for sannhetsverdi. Alle verdier som ikke er lik 0 er sanne.
Vi har følgende logiske operatorer

- || (or)
- && (and)
- ! (not, negation)

Operatorer for sammenligning

For å sammenligne enkle datatyper har vi følgende operatorer

- $>$ (større enn)
- $>=$ (større enn eller lik)
- $<$ (mindre enn)
- $<=$ (mindre enn eller lik)
- $==$ (lik)
- $!=$ (ikke lik)

For å sammeligne strenger kan man bruke strcmp. For å sammenligne minneområder (f.eks struct eller array) kan man bruke memcmp.

Logiske operatorer og sammenligning

```
int a = 10;
int b = 1;

if ((a == 10) || (b != 2)) {
  // Do something
} else if (!(a == 10) || (b == 2)) {
  // Do something
} else if ((a == 10) && !(b != 2)) {
  // Do something
} else if (memcmp(&a, &b, sizeof(int))) {
  // Do something
}
```

Binære operatorer

- $\sim$ (not)
- & (and)
- | (or)
- $\wedge$ (xor)
- $>>$ (shift right)
- $<<$ (shift left)

Binære operatorer

La $a = 273$ og $b = 32005$.

$$
\begin{aligned}
a &= 0000000100010001 \\
b &= 0111110100000101 \\
a\&b &= 0000000100000001 \\
a|b &= 0111110100010101 \\
a \wedge b &= 0111110000010100 \\
\sim a &= 1111111011101110 \\
a >> 2 &= 0000000001000100 \\
b << 2 &= 1111010000010100
\end{aligned}
$$

Binære operatorer

```
unsigned int a = 273, b = 32005, tmp;

tmp = a & b;
tmp = a | b;
tmp = a ^ b;
tmp = ~a;
tmp = a >> 2;
tmp = a << 2;

if (tmp & 0x10) {
  // Bit nummer 5 er satt til 1
}
```

Program-flyt

if-else if-else

```
int a = 4, b;
if (a == 1) {
  b = 2;
} else if (a == 3) {
  b = 5;
} else {
  b = 10;
}
```

while-loop

```c
int cnt = 0;
while (cnt < 10) {
  printf("Count: %d\n", cnt);
  cnt++;
}
```

for-loop

```
int i;
for (i = 0; i < 10; ++i) {
  printf("Count: %d\n", i);
}

for (int j = 0; j < 10; j++) {
  printf("Count: %d\n", j);
}

i = 0;
for (;;++i) {
  printf("Count: %d\n", i);

  if (i > 10) {
    break;
  }
}
```

do-while

do-while brukes når condition skal sjekkes etter minst en kjøring av løkken. (read_number er ikke definert)

```
int sum = 0;
do {

  int input = read_number();
  sum += input;

} while (sum < 70);
```

switch

```
int a = 4, b;
switch(a) {
  case 1: b = 1;
  break;
  case 3: b = 3;
  break;

  default: b = 10
}
```

Funksjoner

```
int sum(int a, int b)
{
  return a + b;
}

printf("sum(1,2) = %d\n", sum(1,2));
```

Enhver funksjon har en returverdi etterfulgt av et funksjonsnavn og en parameter-liste.

Funksjoner

```
void print_number(int a)
{
  printf("%d", a);
}

char *find_first_occurence(char character,
                           char *array,
                           unsigned int size) {
  int i = 0;
  for (;i < size; ++i) {
      if (array[i] == character) {
        return array+i;
      }
  }
  return NULL;
}
```

Funksjoner - Returverdi

```
struct person create_person() {
  struct person ole = { .age = 31,
    .height = 1.92 };
  return ole;
}
```

Selv om kompilatoren tillater å returnere en struct anbefaler vi at man heller returnere en peker til en struct. To grunner; 1) ikke alle kompilatorer tillater det, 2) minnekopiering er kostbart når structen er stor.

```
struct person* create_person() {
  struct person *ole =
                malloc(sizeof(struct person));
  ole->age = 31;
  ole->height = 1.92;
  return ole;
}
```

Funksjoner - Returverdier

En funksjon kan kun ha 1 returverdi, men man kan bruke pekere til å få flere.

```
int divide_remainder(int a, int b, int *remainder)
{
  int whole = a / b;
  *remainder = a % b;
  return whole;
}
```

```
int remainder;
int whole = divide_remained(5, 2, &remainder);
```

I eksempelet over returneres heltallet fra funksjonen, og resten skrives til et minneområde som kalleren så kan lese fra.

Funksjoner - Definisjon vs Deklarasjon

Det er en viktig forskjell mellom en definisjon og en deklarasjon av en funksjon. En deklarasjon gjør kompilatoren oppmerksom på at en funksjon eksisterer. En definisjon gir kompilatoren koden til funksjonen så den kan brukes. Hvis en funksjon ikke er deklarert før definisjonen fungerer definisjonen også som en deklarasjon.
En deklarasjon MÅ komme før bruk av funksjonen i C-kode-filen. La oss se nærmere på det.

Funksjoner - Definisjon vs Deklarasjon

```c
int sum(int a, int b)
{
  return a + b;
}

int main(void) {

  printf("sum(1,2) = %d\n", sum(1,2));

}
```

I denne kodesnutten er funksjonen sum definert (og deklarert) før den brukes. Det vil derfor går fint.

Funksjoner - Definisjon vs Deklarasjon

```c
int main(void) {

  printf("sum(1,2) = %d\n", sum(1,2));

}

int sum(int a, int b)
{
  return a + b;
}
```

I denne kodesnutten er funksjonen sum definert (og deklarert) etter den brukes. Kompilatoren vil klage når filen kompileres; function sum er udeklarert!

Funksjoner - Definisjon vs Deklarasjon

```c
int sum(int a, int b);

int main(void) {

  printf("sum(1,2) = %d\n", sum(1,2));

}
```

I denne kodesnutten er funksjonen sum deklarert før den brukes. Kompilatoren vil klage når den skal lage den kjørbare filen; function sum er udefinert!

Funksjoner - Definisjon vs Deklarasjon

```c
int sum(int a, int b);

int main(void) {

  printf("sum(1,2) = %d\n", sum(1,2));

}

int sum(int a, int b)
{
  return a + b;
}
```

I denne kodesnutten er funksjonen sum deklarert før den brukes og definert i slutten av filen. Dette fungere fint. Kompilatoren blir fortalt at funksjonen eksisterer gjennom deklarasjonen, og senere fortalt hvordan den ser ut i definisjonen.

Oppdeling i flere kildefiler

Når et program blir stort og komplekst kan det lønne seg å dele det opp i flere mindre deler og sette dem sammen.
I C bruker man header-filer og kode-filer for å få til denne oppdelingen. Header-filer brukes til å eksponere deklarasjoner og definisjoner som alle kode-filer som bruker funksjonaliteten trenger. Større funksjons-definisjoner legges i kode-filer som kan linkes av gcc.

Oppdeling i flere kildefiler - eksempel

main.c:

```c
#include "print.h"

int main(void)
{
  print_int(2);
  return 0;
}
```

print.h:

```c
void print_int(int tall);
```

print.c:

```c
#include <stdio.h>
void print_int(int tall)
{
  printf("%d\n", tall);
}
```

Kompilering, linking og kjøring:

```
username> gcc main.c print.c -o main
username> ./main
2
```

Pre-prosessering

En av de første tingen en kompilator gjør er pre-prosessering av kildefilene. Under pre-prosesseringen erstatter den forskjellige direktiver med kode før kompilering.
Man kan bruke dette steget til å

- Forenkle kode
- Forhindre flere definisjoner av samme funksjon, global variabel eller struct
- Erstatte include-direktiver med koden i header-filene

Pre-prosessering - Forenkle kode

```
#define PI 3.14
#define SUM(a,b) (a+b)

float radius = 5;
float area = SUM(radius, radius) * PI * PI;


#define ARRAY_SIZE 50

int array[ARRAY_SIZE];
```

Pre-prosessering - Forhindre duplikater

En viktig del av en header-fil er det vi kaller en include-guard. Den forhindrer at definisjoner i header-filen blir lest kun 1 gang.

```
#ifndef HEADER_FILE_H
#define HEADER_FILE_H

... (Kode)


#endif
```

Pre-prosessering - Inkludere filer

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
  printf("Hello World!\n");
}
```

I programmet over settes innholder i stdio.h rett inn i kode-filen før den kompileres. Dette sørger for at alle nødvendige deklarasjoner blir sett av kompilatoren.

Biblioteksfunksjoner

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
  printf("Hello World!\n");
}
```

Det finnes en rekke biblioteksfunksjoner som er tilgjengelig ved å inkludere forskjellige header-filer. I dette eksempelet er printf en biblioteksfunksjon som er deklarert i stdio.h.

Biblioteksfunksjoner

```
username> man printf
username> man 3 printf
username> man 3 strcmp
username> man 3 strdup
```

Man kan finne informasjon om biblioteksfunksjoner under man-sidene