C-programmering

Introduksjon, grunnleggende, verktøykjeden, kodekvalitet

TTK4235

Fokus for forelesningene om C

- I dag
  - Grunnleggende C (men vi antar dette er kjent stoff)
  - Verktøykjeden; prosessen fra C til kjørbar maskinkode
  - Programstruktur, Kodekvalitet

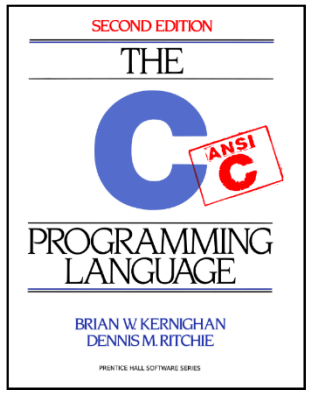
- De neste to gangene:
  - Uke 7: Datastrukturer, pekere, dynamisk minneallokering
  - Uke 8: C-programmering for innebygde datasystemer

Pensumlitteratur

Kompendium av Sverre Hendseth

Kernighan & Ritchie (kap 1-7)

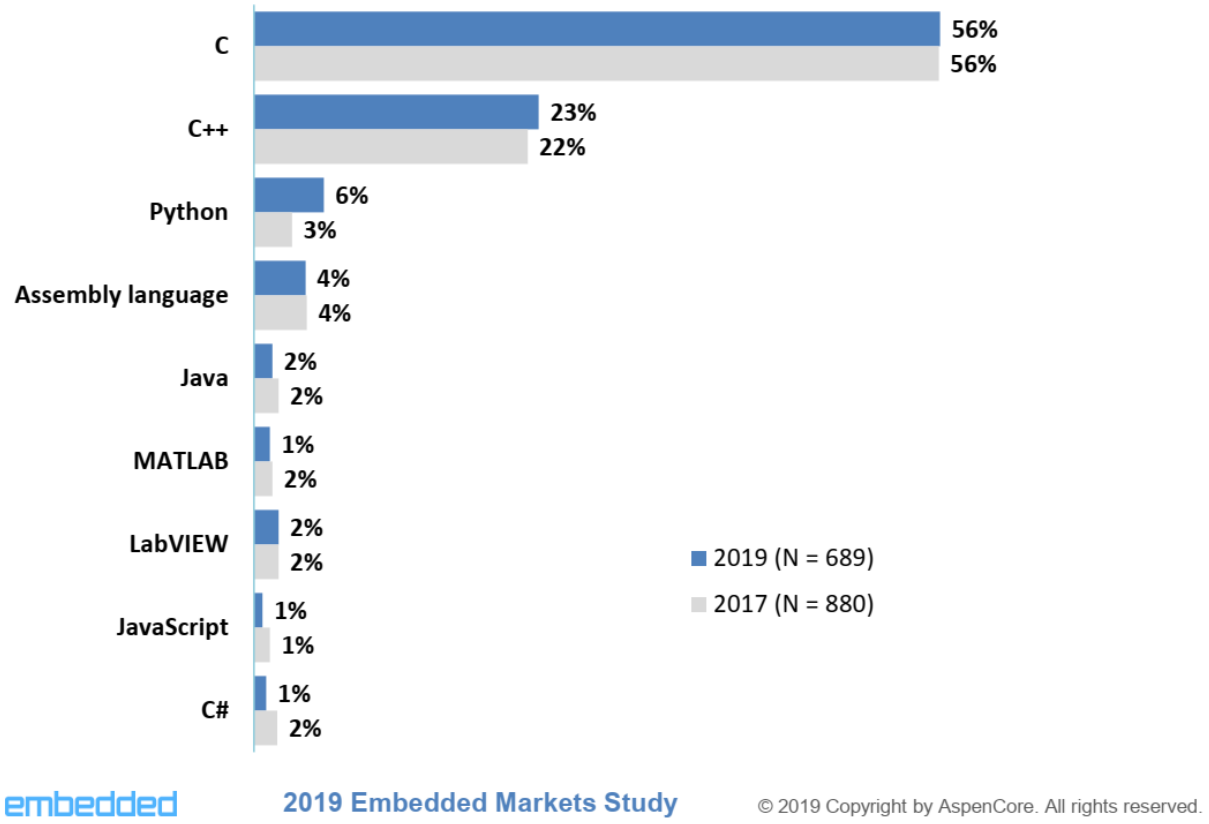
(ikke direkte eksamensrelevant)



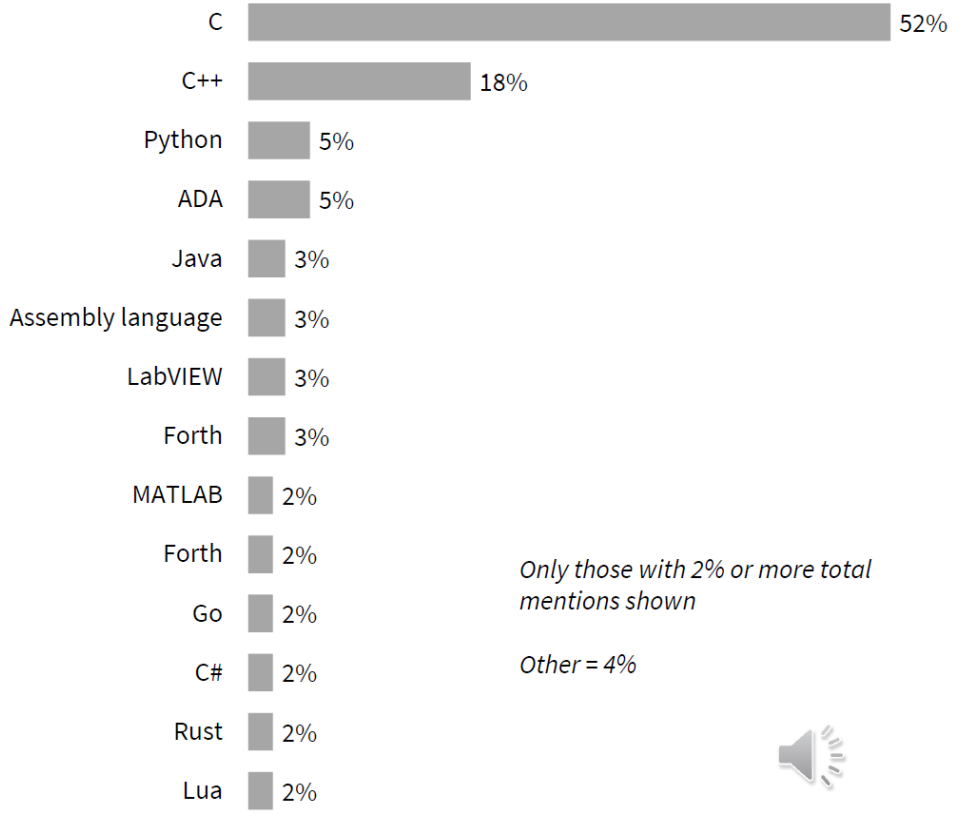
Prøv ut eksemplene i disse mens du leser!

C i utvikling av tilpassede datasystemer

My *current* embedded project is programmed mostly in:

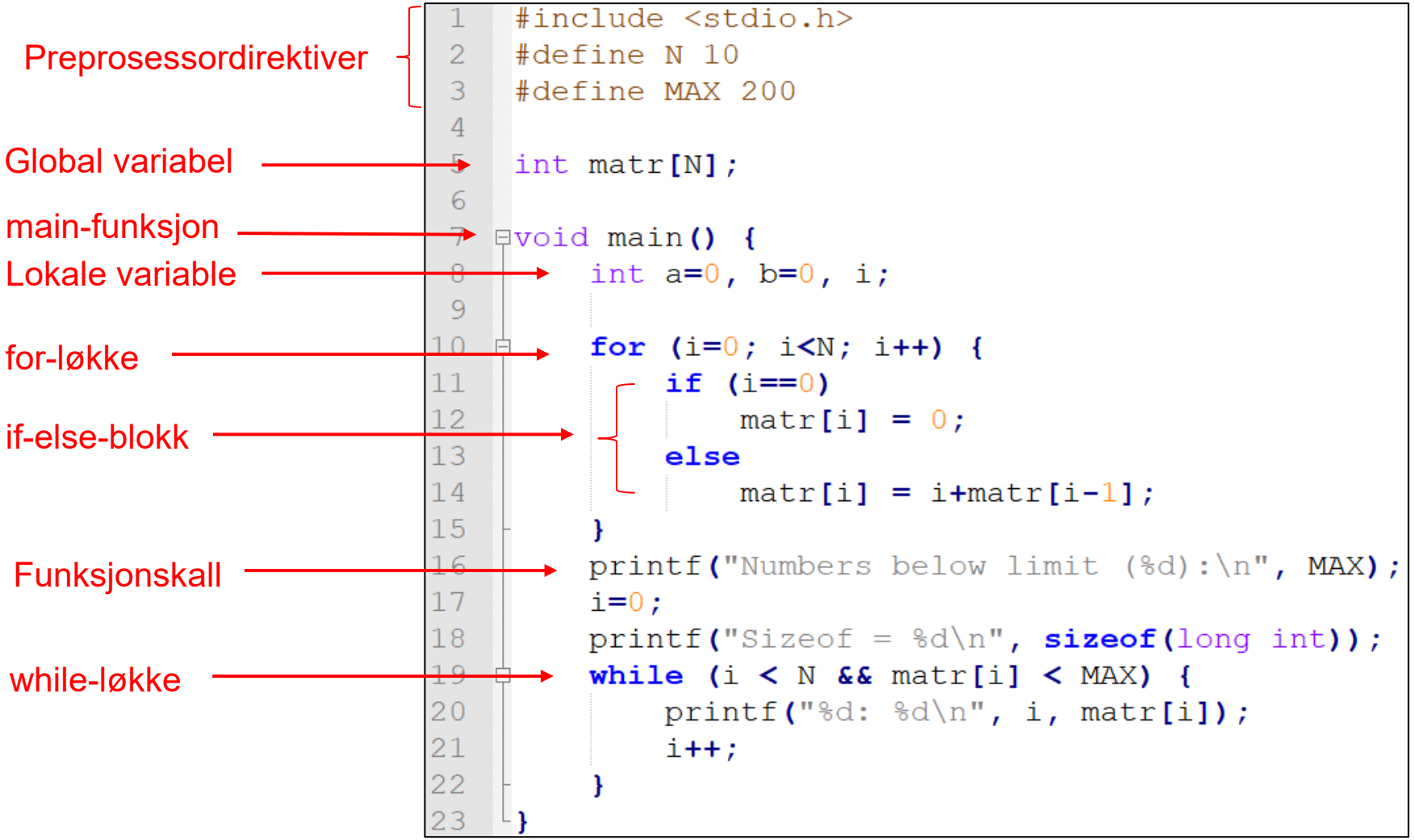


2019 Embedded Markets Study: AspenCore, 2019.



2023 Embedded Markets Study: AspenCore, 2023.

Et enkelt C-program




```c
#include <stdio.h>
#define N 10
#define MAX 200

int matr[N];

void main() {
    int a=0, b=0, i;

    for (i=0; i<N; i++) {
        if (i==0)
            matr[i] = 0;
        else
            matr[i] = i+matr[i-1];
    }
    printf("Numbers below limit (%d):\n", MAX);
    i=0;
    printf("Sizeof = %d\n", sizeof(long int));
    while (i < N && matr[i] < MAX) {
        printf("%d: %d\n", i, matr[i]);
        i++;
    }
}
```

Generelt om C

- C er et høynivåspråk som gir lett tilgang til lavnivå funksjonalitet og hardware-nær programmering
- C-kode har lav overhead (minneforbruk/ytelse)
- C kan brukes på de aller fleste systemer, og koden er (relativt) portabel mellom plattformer
- C ligger til grunn for mange høynivåspråk (Perl, Java, Python, Ruby etc.)

Top energy efficient languages, ranked by energy use–Update 2021

1.	C, Rust (tie)	1.00	
2.	C++	1.34	
3.	Ada	1.70	
4.	Julia	1.80	
5.	Java	1.98	
6.	Pascal		2.14
7.	Chapel		2.18
8.	OCaml		2.35
9.	Lisp	2.41	
10.	Fortran		2.52

Top energy efficient languages, ranked by execution time–Update 2021

1.	C, Rust (tie)	1.00
2.	C++	1.56
3.	Ada	1.85
4.	Java	1.89
5.	Chapel	2.14
6.	Haskell	2.74
7.	Go	2.83
8.	OCaml	2.84
9.	Lisp	2.87
10.	Pascal	3.02

Top energy efficient languages, ranked by memory use—Update 2021

1.	Rust, Pascal (tie)	1.00
2.	Go	1.05
3.	C	1.17
4.	Fortran	1.24
5.	C++	1.34
6.	Ada	1.47
7.	Rust	1.54
8.	Lisp	1.63
9.	Haskell	1.94
10.	OCaml	2.05

2022 Lee Teschler,: The top ten «green» programming languages.
https://www.designworldonline.com/the-top-ten-green-programming-languages/

Generelt om C

- C gir programmereren svært stor frihet, som også medfører risiko via:
  - Manglende bounds-sjekking («buffer overflow»)
  - Ikke streng kontroll på datatyper
  - Ikke streng kontroll på reservasjon/frigjøring av minne
  - Pekere til eksplisitte minneadresser, og pekeraritmetikk
- Disse kan gi feil i koden! (Og sikkerhetshull!)

Grunnleggende C-programmering

Datatyper

- Grunnleggende datatyper:
  - Heltall: ***char****,* ***int*** (***char*** er dimensjonert for å holde ASCII-tegnverdier, men er en integer-type)
  - Flyttall: ***float****,* ***double***
  - Pekere, funksjonspekere
- Modifikatorer (foran typenavn):
  - ***signed/unsigned****:* angir om en heltallstype skal ha fortegn
  - ***short****:* angir at en heltallstype skal bruke få bytes
  - ***long****:* angir at en heltallstype eller **double** skal bruke flere bytes
  - ***const, static, volatile***: styrer andre aspekter ved variabler

Datatyper

- Antall bytes per variabel er systemavhengig
  (f.eks. er *int* normalt 2 eller 4 bytes).
- Vær oppmerksom på dette ved hardware-nær programmering, og ved kode som brukes på forskjellige plattformer!
- Operatoren *sizeof(type)* gir antall bytes kompilatoren bruker for en gitt type.

- **Bedre:** bruk typer definert i ***<stdint.h>*** med eksplisitt bitbredde:
  - *int8_t / uint8_t*
  - *int16_t / uint16_t*
  - *…*

Operatorer

- Operatorer er symboler som angir at spesifikke operasjoner skal utføres på en eller flere *operander*:
  - Aritmetiske: **+ - * / % ++ --**
  - Relasjonelle: **== != > < <= >=**
  - Logiske: **&& || !**
  - Bitvise: **& | ^ ~ << >>**
  - Tilordning: **= += -= *= /=** (og flere)
  - Diverse: sizeof() **& * ? :**

- Operatorene stiller visse krav til operandenes type
- Avvikende typer kan konverteres implisitt etter visse regler

Operator-presedens

Avgjør i hvilken rekkefølge operatorene håndteres

Mentimeter

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(*type*){*list*}	Compound literal(C99)	
2	++ --	Prefix increment and decrement	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	Type cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of	
	_Alignof	Alignment requirement(C11)	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10	|	Bitwise OR (inclusive or)	
11	&&	Logical AND	
12	||	Logical OR	
13	?:	Ternary conditional	Right-to-Left
14	=	Simple assignment	
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
	&= ^= |=	Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

Tabell fra https://xeratol.com/archives/138

Bitvise operatorer

Operasjoner på bit-nivå er ofte sentrale i embedded-systemer; f.eks. ved memory mapped IO kan enkeltbits på spesifikke adresser svare til flagg eller digitale innganger/utganger. Da trenger vi typisk å:

- Finne verdien av en spesifikk bit
- Sette verdien av en eller flere bits

Merk: integerkonstanter kan gis på flere måter:

- Desimalt: `c = 9;`
- Binært (1 bit per tegn): `c = 0b00001001;` (prefiks **0b**)
- Oktalt (3 bits per tegn): `c = 011;` (prefiks **0**)
- Hexadesimalt (4 bits per tegn): `c = 0x09;` (prefiks **0x**)

Bitvise operatorer

- **&**, **|** og **^** gjør henholdsvis bitvis AND, OR og XOR mellom operandene
- **~** gir komplement (invers av alle bits)
- **<<** og **>>** forskyver bitfeltet mot venstre eller mot høyre (fyller normalt inn med 0, men vær obs på virkemåte ved høyreskift av verdi med fortegn!)

Eksempel:

```
int c = 9; // = 0b00001001;
printf("c & 0xFE = %d\n", c & 0xFE);
printf("c | 0b00010000 = %d\n", c | 0b00010000);
printf("c >> 1 = %d\n", c >> 1);
printf("c << 2 = %d\n", c << 2);
```

```
$ ./test.exe
c & 0xFE = 8
c | 0b00010000 = 25
c >> 1 = 4
c << 2 = 36
```

Bitvise operatorer

Merk: de bitvise operatorene har lavere presedens enn == og !=

Det betyr at ved sjekk av f.eks. hvorvidt en bit er aktiv, må en bruke parenteser:

```
MASK = 0b00001000;
if ((x & MASK) == 0) {
    …
}
```

(Hvis ikke vil «MASK==0» bli evaluert først, og resultatet AND-et med x)

Grener

Styrer kontrollflyten avhengig av betingelser

if:

```
if(expression){
    //...
} else if(expression){
    // ...
} else {
    // ...
}
```

switch:

```
switch(expression){
    case value1:
        //...
        break;
    case value2:
    case value3:
        //...
        break;
    default:
        //...
        break;
}
```

Løkker

Styrer kontrollflyten for å repetere aksjoner

```
while(expression){
    //...
}
```

```
do{
    //...
}while(expression);
```

```
for(initialize; test; increment){
    //...
}
```

```
Eks:
for(int i = 0; i < 5; i++){
    // i har verdien 0..4
}
```

Kontrollstrukturer

- Grener
  - **switch** kan kun teste på én verdi, **if-else** kan teste på vilkårlige verdier
  - Foretrekk **switch** hvis det passer
    - **Intensjonen kommuniseres bedre**
      - Færre repetisjoner av ”if a == 0, else if a==1, else if...”

- Løkker
  - Forsiktig med **do-while**, siden betingelsene ligger nederst av en (potensielt) stor blokk
  - Bruk **while** hvis du ikke trenger **for**
    - Bare bruk **for** hvis det gjør koden mer leselig
    - Vanligvis i kontekst av iterasjon over arrays (evt. andre datastrukturer)

Funksjoner

Funksjoner

Funksjoner er subrutiner som har:

- Inngående argumenter (0 eller flere)
- Mulighet til å definere lokale variabler
- Mulighet til å kalle andre funksjoner
- En returverdi (eller *void*)

```
/**
 * Return a number after calculation.
 */
int doCalc(float a, float b) {
    float c = a*a + b*b;
    return c + a*b;
}
```

En enkel funksjon

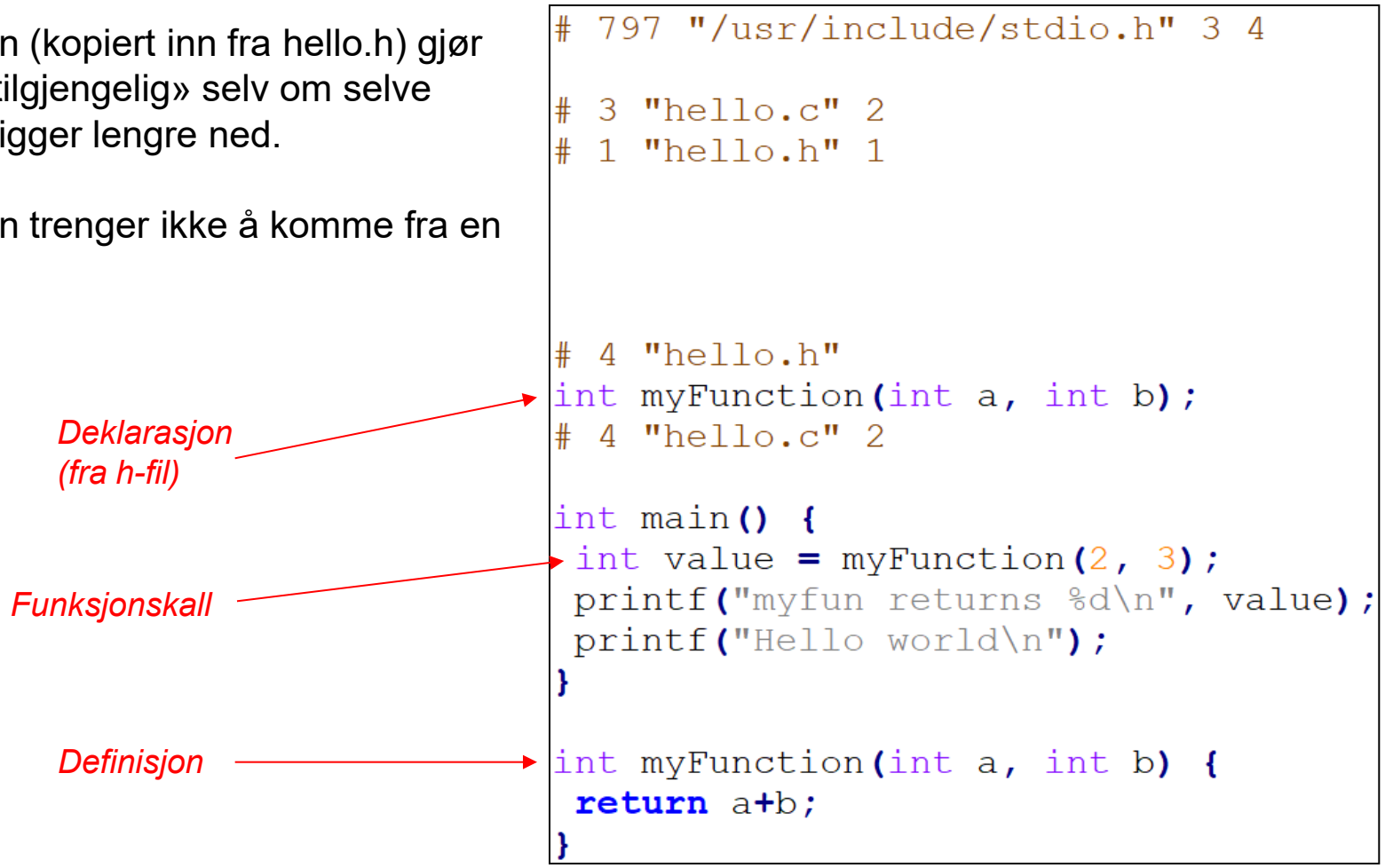
Funksjoner

- Lokale variable legges på *stack* sammen med inngående argumenter
  - Lokale variabler er bare tilgjengelig innenfor funksjonen
  - Dette minnet frigis automatisk når funksjonen returnerer
  - Inngående argumenter kan brukes som andre lokale variable
  - Funksjoner har tilgang på globale variable
- Argumenter sendes ved å kopiere verdien («pass by value»)
  - For å oppnå «pass by reference» må man bruke pekere
  - Arrays sendes «pass by reference» eller via pekere

Funksjoner – definisjon VS deklarasjon

Deklarasjonen (kopiert inn fra hello.h) gjør funksjonen «tilgjengelig» selv om selve definisjonen ligger lengre ned.

Deklarasjonen trenger ikke å komme fra en h-fil.




```
# 797 "/usr/include/stdio.h" 3 4

# 3 "hello.c" 2
# 1 "hello.h" 1




# 4 "hello.h"
int myFunction(int a, int b);
# 4 "hello.c" 2

int main() {
 int value = myFunction(2, 3);
 printf("myfun returns %d\n", value);
 printf("Hello world\n");
}

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

Verktøykjeden

GNU Compiler Collection (gcc)

- En av mange C-kompilatorer (Wikipedia lister nesten 50 stk.)
- Kompilator- og biblioteksamling med støtte for C, C++, Objective-C, Fortran, Ada, Go, D
- Lett tilgjengelig for Linux, men kan også brukes under Windows via f.eks. Cygwin
- Kan f.eks. brukes ved ssh til *login.stud.ntnu.no*
- Andre nyttige verktøy:
  - git (versjonskontroll)
  - make (byggeskript)
  - gdb (debugger)
  - Doxygen (genererer dokumentasjon basert på kodestruktur og kommentarer i koden)

C-kompilering med gcc

Enkelt C-program:

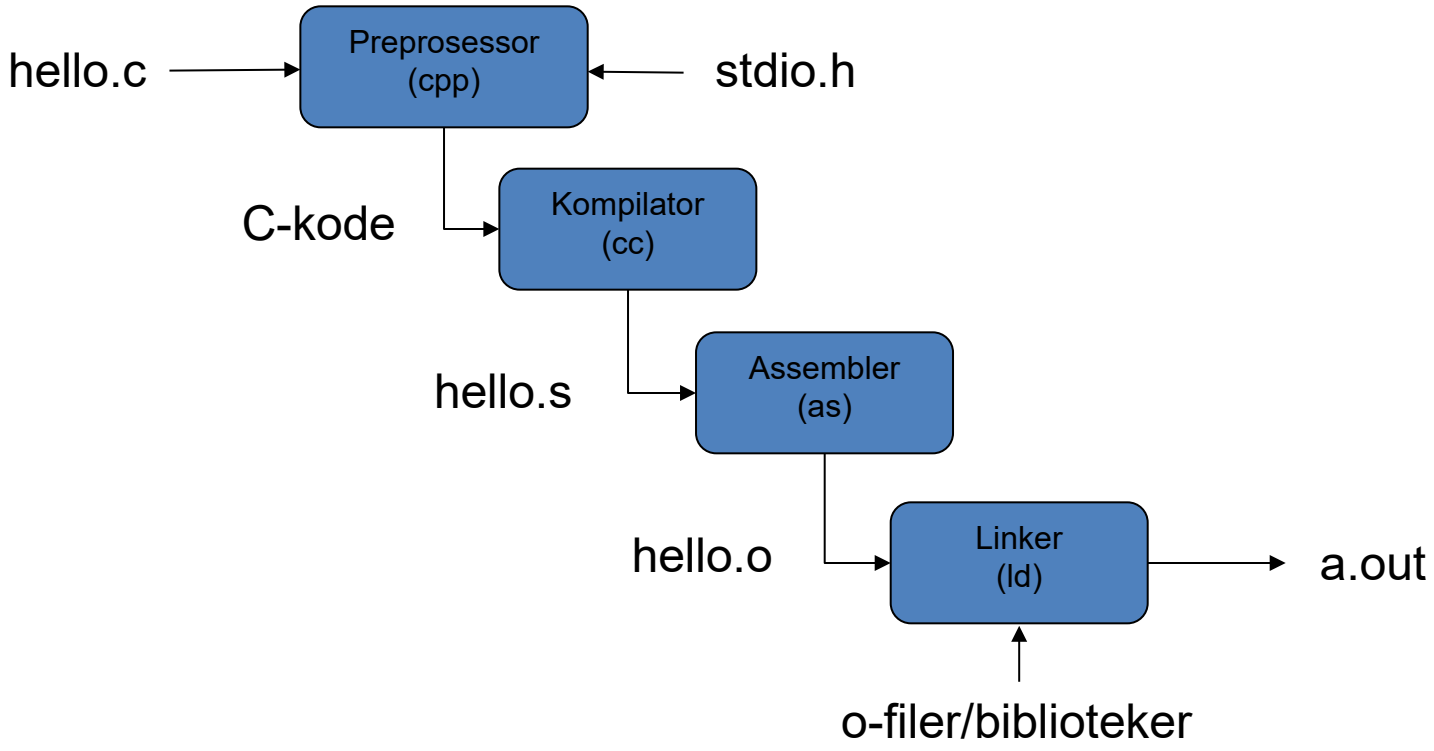
```
#include <stdio.h>

void main(void){
  printf("Hello World\n");
}
```

Kompilering og kjøring:

```
sverre@eee:~/tilpdat$ ls
hello.c
sverre@eee:~/tilpdat$ gcc hello.c
sverre@eee:~/tilpdat$ ls
a.out  hello.c
sverre@eee:~/tilpdat$ ./a.out
Hello World
```

C-kompilering med gcc

Hvilke steg er egentlig involvert?



Preprosessoren

- Kopierer inn innholdet av h-filer referert med **#include**
- Erstatter makroer med riktig innhold
  - Makroer uten argument: ***#define NAME content***
  - Makroer med argument: ***#define NAME(x) x-dependent content***
- Legger inn informasjon om filnavn og linjenummer slik at kompilatoren kan gi presis feilrapportering

Preprosessoren

hello.h

```c
#define HELLO "Hello world\n"

int myFunction(int a, int b);
```

hello.c

```c
#include <stdio.h>
#include "hello.h"

int main() {
    int value = myFunction(2, 3);
    printf("myfun returns %d\n", value);
    printf(HELLO);
}

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

Preprosessoren

Siste del av output av
gcc –E hello.c
(eller *cpp hello.c*):

Metainformasjon for debugging

```
#define HELLO "Hello world\n"

int myFunction(int a, int b);
```

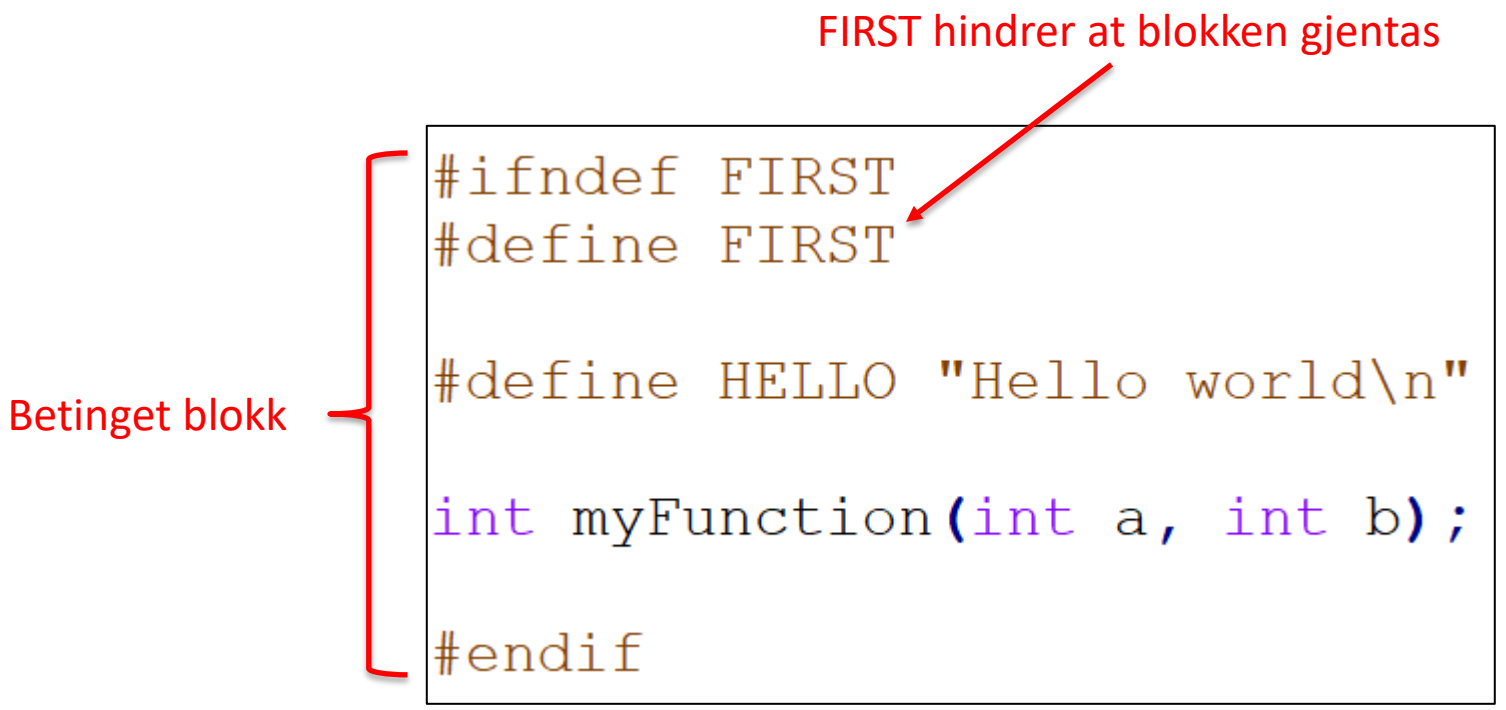
```
#include <stdio.h>
#include "hello.h"

int main() {
    int value = myFunction(2, 3);
    printf("myfun returns %d\n", value);
    printf(HELLO);
}

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

```
# 797 "/usr/include/stdio.h" 3 4

# 3 "hello.c" 2
# 1 "hello.h" 1



# 4 "hello.h"
int myFunction(int a, int b);
# 4 "hello.c" 2

int main() {
 int value = myFunction(2, 3);
 printf("myfun returns %d\n", value);
 printf("Hello world\n");
}

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

Preprosessoren

Direktivene **#ifdef** og **#ifndef** (**#else**, **#elif**) kan brukes sammen med **#endif** for å styre om en seksjon tas med eller ikke

Kan bl.a. brukes for å sikre at definisjoner ikke gjentas dersom en h-fil importeres flere ganger:



Kompilatoren

Oversetter C-kode til prosesssor-spesifikk assembly-kode

Deler av output fra
gcc –S hello.c:

```c
#define HELLO "Hello world\n"

int myFunction(int a, int b);
```

```c
#include <stdio.h>
#include "hello.h"

int main() {
    int value = myFunction(2, 3);
    printf("myfun returns %d\n", value);
    printf(HELLO);
}

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

```
    .file   "hello.c"
    .text
    .def    __main; .scl    2;  .type   32; .endef
    .section .rdata,"dr"
.LC0:
    .ascii "myfun returns %d\12\0"
.LC1:
    .ascii "Hello world\0"
    .text
    .globl  main
    .def    main;   .scl    2;  .type   32; .endef
    .seh_proc   main
main:
    pushq   %rbp
    .seh_pushreg    %rbp
    movq    %rsp, %rbp
    .seh_setframe   %rbp, 0
    subq    $48, %rsp
    .seh_stackalloc 48
    .seh_endprologue
    call    __main
    movl    $3, %edx
    movl    $2, %ecx
    call    myFunction
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    movl    %eax, %edx
    leaq    .LC0(%rip), %rcx
    call    printf
    leaq    .LC1(%rip), %rcx
    call    puts
    movl    $0, %eax
    addq    $48, %rsp
    popq    %rbp
    ret
    .seh_endproc
    .globl  myFunction
    .def    myFunction; .scl    2;  .type   32; .endef
    .seh_proc   myFunction
myFunction:
```

Kompilatoren

Oversetter C-kode til prosesssor-spesifikk assembly-kode

Deler av output fra
gcc –S hello.c:

```
#define HELLO "Hello world\n"

int myFunction(int a, int b);
```

```
#include <stdio.h>
#include "hello.h"

int main() {
    int value = myFunction(2, 3);
    printf("myfun returns %d\n", value);
    printf(HELLO);
}

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

```
myFunction:
    pushq   %rbp
    .seh_pushreg    %rbp
    movq    %rsp, %rbp
    .seh_setframe   %rbp, 0
    .seh_endprologue
    movl    %ecx, 16(%rbp)
    movl    %edx, 24(%rbp)
    movl    16(%rbp), %edx
    movl    24(%rbp), %eax
    addl    %edx, %eax
    popq    %rbp
    ret
```

Kompilatoren: optimalisering

gcc har flagg (-O0, -O1, …) som styrer optimalisering

Optimalisering er automatisk omskriving av koden for å øke ytelsen, f.eks:

- Operasjoner som ikke har effekt kan fjernes
- Løkker kan «rulles ut» til lineær kode
- Funksjoner kan gjøres «inline»

Optimalisering kan gi betydelig forbedring av ytelsen uten å endre virkemåten til programmet

Men: optimalisering kan også potensielt skape problemer, f.eks. ved bruk av minnemappet IO

Krysskompilering

Man kompilerer ofte programmer for kjøring på samme plattform som en utvikler på

Ved utvikling for innebygde datasystemer krysskompilerer man – dvs. kompilerer for en annen plattform (ARM, AVR etc.)

Dette kan gjøres i GCC med opsjon ***–march*** eller ***-mcpu***

(Forutsetter at GCC er kompilert med støtte for den aktuelle plattformen)

Eksempel: kompilatorflagg fra mikrokontroller-øvingen:

–mcpu=cortex-m0 –mthumb –mabi=aapcs –mfloat-abi=soft

Assembleren

- Oversetter assembly-kode til maskinkode
- Produserer objektfiler (hello.o)
- Koden er relokaliserbar (faktiske minneadresser fortsatt ikke bestemt)
- Objektfilen inneholder en «innholdsfortegnelse»: en tabell over eksporterte «symboler» - navn på funksjoner og variable
- Flere objektfiler må typisk linkes inn for å få tilgang til symbolene som trengs av programmet

Output fra *nm hello.o*:

```
$ nm hello.o
0000000000000000 b .bss
0000000000000000 d .data
0000000000000000 p .pdata
0000000000000000 r .rdata
0000000000000000 r .rdata$zzz
0000000000000000 t .text
0000000000000000 r .xdata
                 U __main
0000000000000000 T main
0000000000000047 T myFunction
                 U printf
                 U puts
```

Funksjonene fra hello.c

U (undefined)
Må komme fra andre filer

Minnesegmenter

Symboltyper (fra «*man nm*»):

- B/b: uninitialized data
- C/c: common symbol
- D/d: initialized data
- R/r: read only data
- T/t: text (code) section
- U: undefined symbol

For embedded-system forteller dette noe om hva slags minne de forskjellige elementene skal ligge i

```c
#include <stdio.h>

int i;
int j = 0;
const int k = 0;
const int l;
static int m = 4;

void main(void){
    int lokvar = 2;
    printf("j = %d\n",j);
}
```

gcc –c file.c
nm file.o

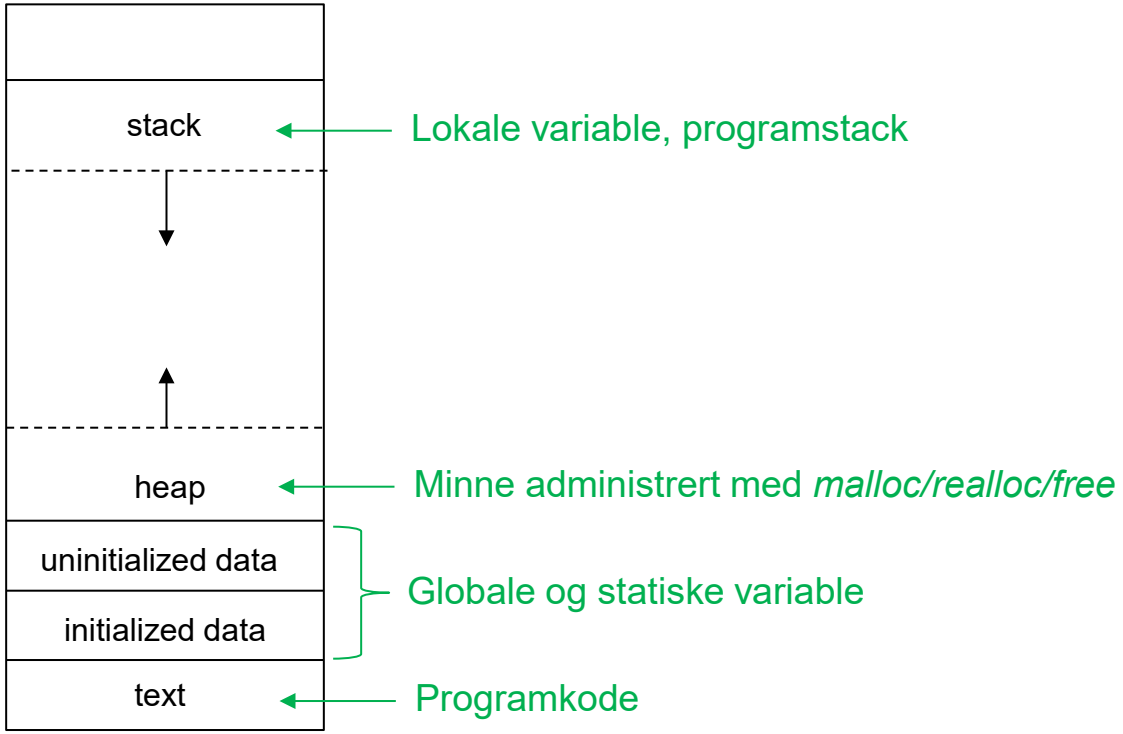
```
0000000000000000 b .bss
0000000000000000 d .data
0000000000000000 i .drectve
0000000000000000 p .pdata
0000000000000000 r .rdata
0000000000000000 r .rdata$zzz
0000000000000000 t .text
0000000000000000 r .xdata
                 U __main
0000000000000004 C i
0000000000000000 B j
0000000000000000 R k
0000000000000004 C l
0000000000000000 d m
0000000000000000 T main
                 U printf
```

Symboltype

Linkeren

- Henter inn objektfiler og bibliotekfiler
- Kombinerer de forskjellige minnesegmentene
- Tilpasser innbyrdes referanser mellom segmentene
- Genererer utlegg i minnet for program og data
- Lager den kjørbare filen (f.eks. *a.out*)
- Feilmeldinger oppstår oftest hvis linkeren ikke finner o-filer eller bibliotek som inneholder segmenter som trengs

Typisk minnelayout for et C-program



Programstruktur

Programstruktur

Et bibliotek består typisk av en h-fil og en c-fil (eller flere):

h-filen gir alle makroer og deklarasjoner av alle funksjoner som biblioteket tilbyr.

c-filen(e) inneholder definisjoner av funksjonene.

mylib.h

```c
#define MYNUMBER 42

int add(int a, int b);

int multiply(int a, int b);
```

mylib.c

```c
#include "mylib.h"

int add(int a, int b) {
    return a+b;
}

int multiply(int a, int b) {
    return a*b;
}
```

Programstruktur

Brukeren av biblioteket (main.c)
inkluderer h-filen for å få tilgang til
makroene og fuksjonsdeklarasjonene.

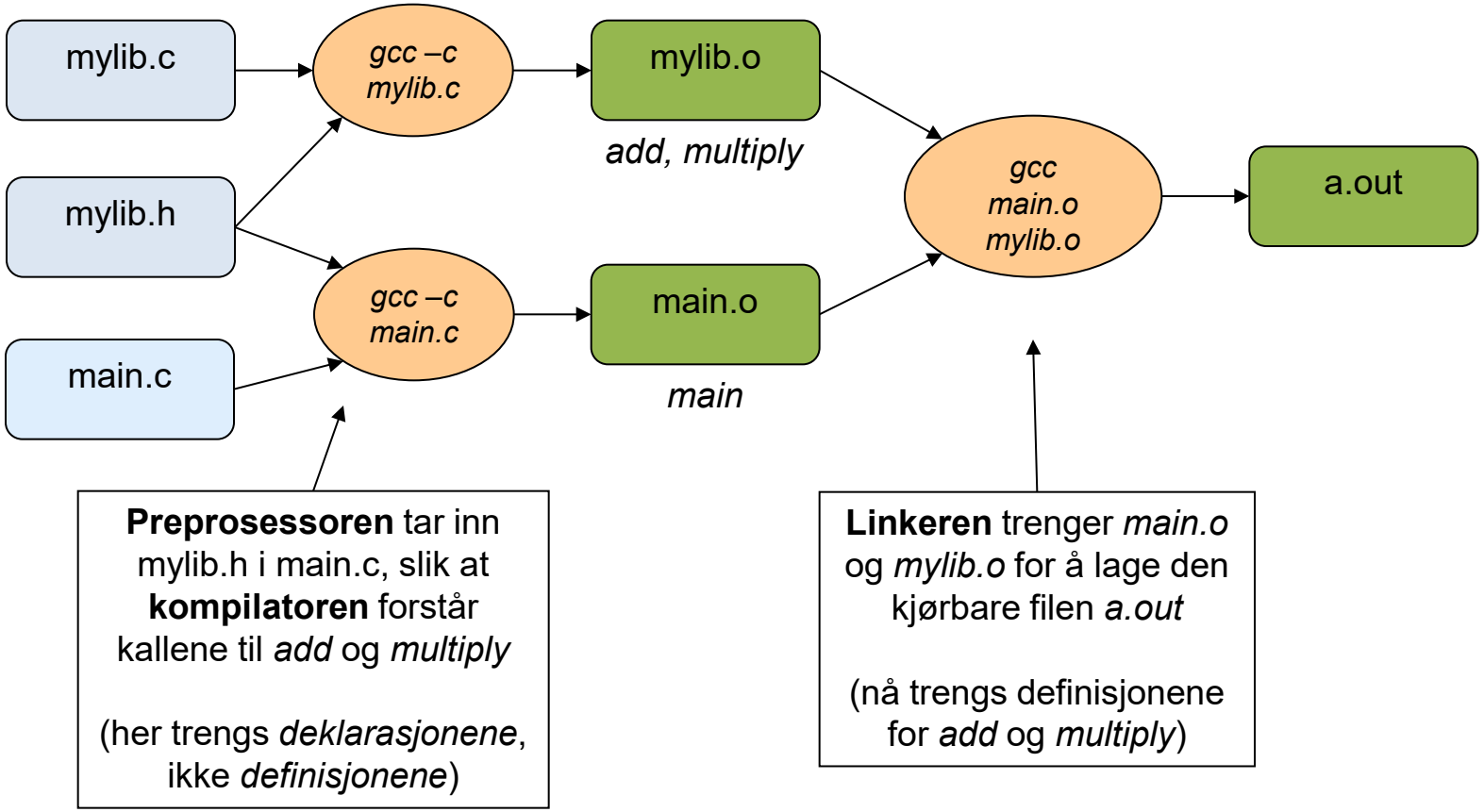
Main-funksjonen bruker disse elementene fra mylib:

- MYNUMBER
- add()
- multiply()

main.c

```
#include <stdio.h>
#include "mylib.h"

void main() {
    int n1=3, n2=MYNUMBER;

    printf("add: %d+%d = %d\n", n1, n2,
        add(n1,n2));
    printf("multiply: %d*%d = %d\n", n1, n2,
        multiply(n1,n2));

}
```

Programstruktur



Recap kompilator vs. linker - feilmeldinger

Feilmeldinger – kompilator vs. linker

```c
void main() {

    int aaa = doC alculation(3, 5);

}
```

```
$ gcc oppgaver.c
oppgaver.c: In function ‘main’:
oppgaver.c:5:12: error: ‘doC’ undeclared (first use in this function)
  int aaa = doC alculation(3, 5);
            ^~~
oppgaver.c:5:12: note: each undeclared identifier is reported only once for each function it appears in
oppgaver.c:5:16: error: expected ‘,’ or ‘;’ before ‘alculation’
  int aaa = doC alculation(3, 5);
                ^~~~~~~~~~
```

(Kompilatoren forstår ikke syntaksen)

Feilmeldinger – kompilator vs. linker

```
void main() {

    int aaa = doCalculation(3, 5);

}
```

```
$ gcc oppgaver.c
oppgaver.c: In function ‘main’:
oppgaver.c:5:12: warning: implicit declaration of function ‘doCalculation’ [-Wimplicit-function-declaration]
  int aaa = doCalculation(3, 5);
            ^~~~~~~~~~~~~
/tmp/ccT6G8wO.o:oppgaver.c:(.text+0x18): undefined reference to `doCalculation'
/tmp/ccT6G8wO.o:oppgaver.c:(.text+0x18): relocation truncated to fit: R_X86_64_PC32 against undefined symbol `doCalcu
lation'
collect2: error: ld returned 1 exit status
```

(Kompilatoren advarer om udefinert funksjon.
Linkeren finner ikke maskinkodesegmentet for doCalculation)

Feilmeldinger – kompilator vs. linker

```c
int doCalculation(int, int);

void main() {

    int aaa = doCalculation(3, 5);

}
```

```
$ gcc oppgaver.c
/tmp/ccWGUnYO.o:oppgaver.c:(.text+0x18): undefined reference to `doCalculation'
/tmp/ccWGUnYO.o:oppgaver.c:(.text+0x18): relocation truncated to fit: R_X86_64_PC32 against undefined symbol `doCalcu
lation'
collect2: error: ld returned 1 exit status
```

(Kompilatoren er fornøyd, men linkeren finner fortsatt ikke segmentet)

Kodekvalitet

Kodekvalitet

Kan definere kriterier, men også et aspekt av skjønn her!

Kodekvalitet handler om:

- **Lesbarhet:** hvor lett er det å lese og forstå koden?
- **Vedlikeholdbarhet:** hvor godt legger koden til rette for vedlikehold (bugfiksing, videreutvikling)?
- **Pålitelighet:** er koden skrevet på en måte som bidrar til å sikre mot feil?

Noen kriterier for kodekvalitet

Moduler:

- Gir navnet på modulen en tydelig beskrivelse på hva modulen gjør?
  - Har modulen en klart definert oppgave?
- Gir modulens eksterne grensesnitt en minimal og komplett abstraksjon av modulens funksjonalitet?
  - Funksjoner som bare trengs til internt bruk bør ikke eksponeres via h-fil
  - Er modulens funksjoner på samme abstraksjonsnivå? Ikke bland lavnivå og høynivå funksjonalitet
  - Gjør grensesnittet det åpenbart hvordan modulen brukes?
- Interaksjonen mellom moduler bør være så lett koblet som mulig

Noen kriterier for kodekvalitet

Variabler:

- Er variabelnavnet beskrivende og samsvarer med variabelens rolle?
- En god konvensjon er å prefikse variabelnavn for å signalisere type variabel:
  - **p_** for peker til en variabel
  - **pp_** for peker til peker
  - **m_** for static globale variable
  - **g_** for globale variable

Noen kriterier for kodekvalitet

Funksjoner:

- Gjør funksjonen det navnet indikerer, og ikke mer/mindre?
  - Funksjoner bør prefikses med navn på modulen de tilhører (f.eks. *bluetooth_init()* fremfor bare *init()*)
- Er det lett å lese funksjonen og forstå hva den gjør?
- Hvor godt sikrer funksjonen seg mot feiltilstander, f.eks. hvis den kalles med ugyldige argumentverdier?

```
int sumValues(int a[], int from, int to) {
    int sum=0;
    for (int i=from; i<to; i++)
        sum += a[i];
    return sum;
}
```

Eksempel: funksjon som er dårlig sikret mot feiltilstander

Kohesjon

- Kohesjon handler om hvor godt moduler (eller operasjoner) er gruppert
- Programmer eller moduler med god kohesjon er typisk lettere å forstå, vedlikeholde og gjenbruke
- Typer av kohesjon:
  - Tilfeldig (elementene er gruppert uten noen god grunn)
  - Sekvensiell (reflekterer rekkefølgen operasjonene må gjøres i)
  - Kommunikasjonskohesjon (når en gjør forskjellige operasjoner på samme data)
  - Tidsavhengig kohesjon (urelaterte operasjoner som gjøres på samme tid)
  - Funksjonell kohesjon (operasjoner som gjøres for å løse en spesifikk oppgave)

Ikke ønskelig!

Mest ønskelig

- Tenk på dette ved design og modularisering, før koden skrives!

Noen flere eksempler vedr. kodekvalitet

```
int m_current_state;
```

```
void lobster(){
    //lots of canary stuff
}
```

```
void main(void)
{
    char* s;

    init();
    send(s);




}
```