Low Level C: Eksamenskompendium 2025

Simon Shine <shine@ek.dk>, 7. december 2025

Dette kompendium indeholder teori og øveopgaver for faget Low Level C. Det er en opsamling af diverse tutorials, der er skabt igennem semesteret, hvor dele af tutorials som ikke er gennemgået er skåret fra, og hvor der er lagt fokus på praktiske, eksamensforberedende øveopgaver.

Hvordan læser man eksamenskompendiet?

I stedet for at læse det fra ende til anden (det må du gerne, men det bliver svært at huske), så tag i stedet hvert af de teoretiske eksamensemner og spørg dig selv:

- Hvilke spørgsmål kan man stille til dette emne?
- Hvordan demonstrerer man bedst dette emne? og
- Hvor i kompendiet finder man svar på spørgsmålene?

Selvom det kan være nemmere at finde svar ved at søge på nettet (gør det endeligt!) og selvom det giver point at referere til ting udenfor kompendiet, er eksamen afgrænset på den måde, at det ikke trækker ned i karakter, hvis der ikke står noget om det i kompendiet. Selvom tutorials og klassetimer har været måden I har lært at programmere C, er kompendiet i retrospekt den "bog" vi har brugt.

Contents

Low Level C: Eksamenskompendium 2025 . . . 1
Hvordan læser man eksamenskompendiet? . . . 1
Hello, World! . . . 5
Øveopgave: Leap Year . . . 6
Funktionssignaturer . . . 7
Logiske udtryk . . . 7
Operator-præcedens . . . 7
Øveopgave: Grains . . . 7
Taltyper med forskellig størrelse . . . 8
`uint64_t` også kendt som `unsigned long long` . . . 8
`printf` og formatkoderne `%s`, `%d` og `%llu` . . . 9
Arrays og hukommelse . . . 10
Variable har adresser i hukommelsen (`&`-operatoren) . . . 11
Erklæring og initialisering af variable . . . 12
Scopes og stack-frames: Funktioner har deres egne variable . . . 13
Parametre til funktioner er bare lokale variable . . . 14
Strings og string-funktioner . . . 15
Grundlæggende syntaks for `char[]` og `char *` . . . 15
1. Character array med fast størrelse . . . 15
2. Initialisering med string literal . . . 15

3. Pointer til string literal (read-only) ... 15
4. Forskellen mellem `char[]` og `char *` ... 15
Vigtige regler ... 15
Gennemgang af standard string-funktioner ... 16
Kommandolinjeargumenter ... 16
Øveopgave: Skriv din egen `strlen()`, `strcpy()` og `strlower()` ... 17
`my_strlen()` - Find længden af en string ... 17
`my_strcpy()` - Kopiér en string ... 17
`my_strlower()` - Konvertér til små bogstaver ... 17
Structs og struct pointers ... 19
Hvornår bruger man hvad? ... 19
Med dynamisk allokering ... 19
Øveopgave: RGB hex-parser ... 20
Struct-pointere ... 21
RGB-hexparser med fejlhåndtering ... 22
Lidt om sikkerhed og bufferlængder ... 23
Stak-allokering og heap-allokering af strings ... 24
Structs, struct pointers og linkede lister ... 25
1. Introduktion til structs ... 25
2. Problemer med stak-allokerede structs ... 25
3. Heap-allokering med malloc() og free() ... 26
4. Struct pointers og -> operatoren ... 26
5. Structs inde i structs - Selvreferencerende structs ... 27
6. Bygning af en linket liste ... 27
Trin 1: Opret en enkelt node ... 27
Trin 2: Sammenkæd noder manuelt ... 27
Trin 3: Push-funktion til at tilføje elementer ... 28
Trin 4: Pop-funktion til at fjerne elementer ... 28
7. Fra lister til træer ... 29
Vigtige takeaways ... 29
Binære tal, binær repræsentation ... 29
Positionelle talsystemer ... 30
Det Decimale System (Base 10) ... 30
Det Binære System (Base 2) ... 30
Øveopgave: Talkonvertering ... 31
Hvordan fisker man en enkelt bit ud af et tal vha. `>>` og `&` ... 31
Two's complement ... 33
Motivationen bag two's complement ... 33
Øvelse: Linux filrettigheder og datamodellering ... 34
Linux filrettigheder som bits ... 34
2. Bitmasker ... 35
3. Oktaler ... 35
4. Bitvise operatorer (`&`, `|`, `~`) ... 36

Man bruger OR (`|`) til at tænde for bits ... 36
Man bruger AND (`&`) til at slukke for bits ... 36
Man bruger NOT (`~`) til at flippe bits ... 37
Hvad gør `&=` og `|=` i C kode? ... 37
5. Struct pointers, fejlhåndtering og output-parametre ... 37
`perms->bits &= ~OTHER_READ;` ... 38
Make, CMake, objektfiler, linking, assembly ... 38
Compile med "gcc" ... 38
Compile med "make" ... 39
Bygge med CMake ... 40
Include paths ... 42
Hvad er en objekt-fil (.o)? ... 43
Hvordan ved compileren hvilken adresse en funktion bor på? ... 43
Linking ... 44
hello.h: ... 44
hello.c: ... 44
main.c: ... 44
Include guards ... 44
CMakeLists.txt: ... 45
Dynamisk og statisk linking ... 46
Dynamisk linking ... 46
Statisk linking ... 46
Dynamisk linking vs. statisk linking (fordele og ulemper) ... 47
Statisk linking er foretrukket i cloud deployment ... 47
Dekompilering til assembly ... 48
ASM 1: Hvordan virker funktionskald i assembly? ... 48
Grundlæggende assembly-begreber ... 50
Numre i assembly-koden ... 50
Registre ... 50
Instruktioner ... 50
Stakken og stack frames ... 50
Stakken som buffer for registre ... 51
1. Funktionens start (prologue) ... 51
2. Funktionens krop ... 51
3. Funktionens afslutning (epilogue) ... 51
Funktionskald fra to perspektiver ... 52
Som den kaldte funktion ... 52
Som den kaldende funktion ... 52
Minimal stack frame ... 52
Threads og parallelprogrammering ... 53
1. Introduktion til threads ... 53
Forskel på processer og threads ... 53
Hvornår bruger man threads? ... 53

Void-pointere (`void *`) ... 54
Hvad er void? ... 54
Hvorfor bruge void-pointere? ... 54
Type casting med void-pointere ... 54
Hvad vil det sige at "caste"? ... 55
Hvorfor er det relevant for threads? ... 56
Praktisk eksempel ... 56
Sammenhængen mellem header-filer, object-filer og linking ... 57
Hvad sker der i hvert trin? ... 57
Praktisk eksempel ... 58
Headers og basic typer ... 58
3. Oprettelse og håndtering af threads ... 59
Basic thread creation ... 59
pthread_create parametre ... 60
pthread_join og return values ... 60
4. Deling af data mellem threads ... 61
Race conditions ... 61
Hvorfor sker race conditions? ... 61
5. Mutex (Mutual Exclusion) ... 62
Basic mutex brug ... 62
Mutex initialisering ... 62
Risiko for deadlock ... 63
Tutorial: Makroer, typedefs ... 64
1. Introduktion til makroer ... 64
Makroer til at lave konstanter ... 64
2. Makroer der minder om funktioner ... 65
Hvorfor alle de parenteser? ... 65
Flerlinjet makroer ... 65
Hvorfor `do { ... } while(0)` tricket? ... 66
3. Fejl der kan opstå ved at bruge funktionsmakroer ... 66
Problem 1: Multiple evaluation og side effects ... 66
Problem 2: Multiple evaluation og side effects ... 67
4. Introduktion til typedefs ... 68
Basic typedef syntax ... 68
Fordele ved typedefs ... 68
5. Typedefs med structs ... 69
Navngivne structs med typedef ... 69
Kombination: navngivet struct + typedef på én linje ... 69
Anonyme structs med typedef ... 70
Hvornår skal man bruge makroer? ... 70
Hvornår skal man bruge typedefs? ... 71
Tutorial: Enums, unions, tagged unions ... 72
1. Introduktion til enums ... 72

Eksplicitte værdier ... 73
Enums vs. **#define** ... 73
2. Introduktion til unions ... 74
Brug af unions ... 74
Hvornår er unions nyttige? ... 74
3. Problemer med almindelige unions ... 75
Hvad går galt? ... 75
4. Tagged unions (diskriminerede unions) ... 76
5. Øvelse: Inventory-system til computerspil ... 77
Trin 1: Definition af item-typer ... 77
Trin 2: Tagged union ... 77
Trin 3: Constructor-funktioner ... 78
Trin 4: Sikre accessor funktioner ... 79
Trin 5: Praktisk brug ... 80

Hello, World!

For at komme nemt i gang installeres CLion, som er IntelliJ til C/C++.

Man skal være opmærksom på ikke at oprette C++ projekter og .cpp-filer, som er default.

Det første program vi skriver og compiler er "Hello, World!" i filen main.c:

```
#include <stdio.h>

int main(void) {
    printf("Hello, World!\n");
    return 0;
}
```

- Ligesom Java hedder hovedfunktionen i programmet **main**
- Forskelligt fra Java er den ikke indeholdt i nogen klasse
- C har slet ikke klasser, så funktioner bor yderst i hvert fil
- Funktioner som **printf** skal inkluderes vha. **#include** (en makro)
- Returværdien indikerer om programmet afslutter succesfuldt
- Returværdien 0 betyder succes, og alle andre værdier tolkes som fejlkoder
- Filen **stdio.h** er en såkaldt header-fil som indeholder navne på funktioner, men indeholder ikke rent faktisk funktionerne – .h-filer bliver brugt til at dele konstanter, makroer, funktioner og datatyper på tværs af .c-filer
- I Java **import**'er man klasser, og i C **#include**'er man header-filer
- **f**'et i **printf** står for *format* og har sin helt egen syntaks, man skal øve sig på

Øveopgave: Leap Year

> A leap year (in the Gregorian calendar) occurs: In every year that is evenly divisible by 4. Unless the year is evenly divisible by 100, in which case it's only a leap year if the year is also evenly divisible by 400. Some examples:
>
> - 1997 was not a leap year as it's not divisible by 4.
> - 1900 was not a leap year as it's not divisible by 400.
> - 2000 was a leap year!
>
> Write the function **`bool is_leap_year(int year)`** and demonstrate that it works.

```c
#include <stdio.h>
#include <stdbool.h>

bool is_leap_year(int year) {
    if (year % 4 == 0 && year % 100 != 0 || year % 400 == 0) {
        return true;
    } else {
        return false;
    }
}

int main(void) {
    printf("Er 1997 et skudår? %s\n", is_leap_year(1997) ? "ja" : "nej");
    return 0;
}
```

- Funktionen **`is_leap_year()`** tager en **`int year`** som input og returnerer en **`bool`**
- Den benytter konstanterne **`true`** og **`false`** som faktisk er relativt nye i C. Før i tiden brugte man bare 0 og 1, og er ens projekt ikke oprettet som et C23-projekt, skal man faktisk **`#include <stdbool.h>`** før konstanterne virker.
- Funktionen er defineret oven over **`main`**, fordi C har et problem med at genkende funktioner, der defineres senere end de bruges. Det kan fikses med en *funktionssignatur*, som også hedder en *prototype* i C.

```c
#include <stdio.h>
#include <stdbool.h>

bool is_leap_year(int);

int main(void) {
    printf("Er 1997 et skudår? %s\n", is_leap_year(1997) ? "ja" : "nej");
    return 0;
}

bool is_leap_year(int year) {
```

```
    return year % 4 == 0 && year % 100 != 0 || year % 400 == 0;
}
```

Funktionssignaturer

- En funktionssignatur er altså funktionen uden indhold, kun retur- og input-typer.
- Vi får det samme problem, når vi skal til at flytte funktioner ud af filen.
- Så vi skal bruge funktionssignaturer når vi begynder at arbejde med flere filer.

Logiske udtryk

Operatorerne **`&&`** og **`||`** gør præcis det samme som i Java.

Resultatet af **`&&`** og **`||`** er i sig selv en **`bool`**, så man behøver ikke altid **`if`**-sætninger:

```
bool is_leap_year(int year) {
    return year % 4 == 0 && year % 100 != 0 || year % 400 == 0;
}
```

Operator-præcedens

Fordi vi ikke er helt vant til operatorerne, kan vi sætte nogle parenteser for at gøre rækkefølgen tydeligere:

```
bool is_leap_year(int year) {
    return
        (year % 4 == 0) &&
        (year % 100 != 0) ||
        (year % 400 == 0);
}
```

Parenteserne gør ikke noget her, og IDE'en vil måske endda foreslå, vi fjerner dem.

Øveopgave: Grains

> Calculate the number of grains of wheat on a chessboard. A chessboard has 64 squares. Square 1 has one grain, square 2 has two grains, square 3 has four grains, and so on, doubling each time. Write code that calculates:
>
> - the number of grains on a given square
> - the total number of grains on the chessboard
>
> Write the function **`int count_grains()`** and demonstrate that it works.

```
// Der er et problem med denne kode, da man får overflow
int count_grains() {
    int square_grains = 1;
    int total_grains = 0;

    for (int i = 1; i <= 64; i++) {
        total_grains = total_grains + square_grains;
        square_grains *= 2;
    }

    // 1 + 2 + 4 + 8 + ... + 2^64
    return total_grains;
}
```

Taltyper med forskellig størrelse

- Når man kører ovenstående kode, får man *overflow*.
- Sætter vi et breakpoint inde i loopet, og afprøver hvornår det går galt, sker det når **`i == 31`**. Det er fordi **`int`** som udgangspunkt er et signed 32-bit heltal.
- Her betyder "signed" fortegns-minus, fordi tallet kan være negativt.
- Det har vi jo egentlig ikke brug for her, da antallet af hvedekorn altid er positivt.
- Men hvis vi bruger **`unsigned int`**, går det stadigvæk galt når **`i == 32`**, fordi der kun er plads til 32 fordoblinger på 32 bits. Vi vil derfor gerne have en større taltype.

`uint64_t` også kendt som `unsigned long long`

Lad os hellere bruge navnet **`uint64_t`**.

Selvom det er lidt sværere at skrive og udtale, er det mindre snørklet end **`unsigned long long`**. Og man kan rigtigt nok gætte, at der også findes:

- **`int64_t`**
- **`uint32_t`**
- **`int32_t`**

Før vi kan bruge dem, skal vi huske **`#include <stdint.h>`**

```
#include <stdint.h>

uint64_t count_grains() {
    uint64_t square_grains = 1;
    uint64_t total_grains = 0;

    for (int i = 1; i <= 64; i++) {
        total_grains = total_grains + square_grains;
        square_grains *= 2;
    }
```

```
    // 1 + 2 + 4 + 8 + ... + 2^64
    return total_grains;
}

int main(void) {
    printf("Hvor mange hvedekorn var der? %d\n", count_grains());
    return 0;
}
```

Kører vi programmet virker det stadig ikke!

Men sætter vi et breakpoint inde i for-løkken, kan vi se at den faktisk regner rigtigt helt op til **`i = 64`**, så hvorfor skriver den noget forkert? Fejlen er lidt mere kryptisk: **`%d`**, når vi printer, konverterer tallet til en for lille type, før det printes.

Her skal vi bruge **`%llu`**:

```
printf("Hvor mange hvedekorn var der? %llu\n", count_grains());
```

`%llu` står for *long long unsigned*.

`printf` og formatkoderne `%s`, `%d` og `%llu`

Vi er bekendt med tre slags **`printf`**-koder:

- **`%s`** til strings
- **`%d`** til små tal
- **`%llu`** til større (unsigned) tal

Vi udvider **`printf`**-formatkoderne efter behov.

Husk at afslutte med **`\n`** for at få et linjeskift.

Arrays og hukommelse

Prøv at gemme og compile følgende program:

```
#include <stdio.h>

int main()
{
  char question[] = "What is your name?";
  char greeting[] = "Well, hello ";
  char name[8];

  puts(question);
  gets(name);

  puts(greeting);
  puts(name);

  return 0;
}
```

Det giver nogle advarsler - måske endda både under kompilering og kørsel - om at det bruger den usikre **`gets()`**, men hvad er egentlig så usikkert ved den?

Prøv at køre programmet, og indtast dit navn - ser det ikke ud til at virke fint?

```
$ ./hello
What is your name?
warning: this program uses gets(), which is unsafe.
Simon
Well, hello
Simon
```

`gets(name)` indlæser hvad brugeren taster og gemmer det i variablen name - hvor der er plads til otte tegn. Så prøv at indtaste et længere navn:

```
$ ./hello
What is your name?
warning: this program uses gets(), which is unsafe.
Simon Shine
ine
Simon Shine
```

Så forsvinder "Well, hello" linjen fra udskriften, og bliver i stedet erstattet med "ine" – mærkeligt, right?

Det må have noget med de otte tegn at gøre, så prøv en gang til, men brug en række tal i stedet for et navn - så det er nemmere at se hvad der sker.

```
$ ./hello
What is your name?
warning: this program uses gets(), which is unsafe.
1234567890abcdef
90abcdef
1234567890abcdef
```

`puts(name)` skriver stadig hele navnet ud - de otte første tegn, OG de otte næste.

Men **`puts(greeting)`** skriver de tegn ud der "flød over" efter de første otte ...

Det vil sige at **`gets()`** indlæser alt hvad der bliver indtastet, og gemmer det i den modtagne variabel - og hvis der bliver indtastet mere end der er plads til, så "flyder det bare over" i noget nabo-hukommelse som overlapper med den næste variabel.

Men hvordan kan det overhovedet lade sig gøre? Når vi forstår hvordan variabler (og i særdeleshed arrays) gemmes i memory, så giver det forhåbentlig lidt mere mening.

Variable har adresser i hukommelsen (&-operatoren)

- **`&`** er adresse-operatoren, som finder adressen på en variabel
- Når man har en variabel **`a`**, har den adressen **`&a`**

Start med det her program, **`mem.c`**:

```c
#include <stdio.h>

int main()
{
  int a = 1234;
  int b = 5678;

  printf("Value of   a: %d\n", a);
  printf("Address of a: %d\n", &a);
  printf("Value of   b: %d\n", b);
  printf("Address of b: %d\n", &b);

  return 0;
}
```

Når du kører det får du noget i retning af:

```
$ ./mem
Value of   a: 1234
Address of a: 1835477384
Value of   b: 5678
Address of b: 1835477380
```

De to tal for Address vil helt sikkert være anderledes for dig - prøv at køre programmet et par gange, så vil du se at de måske endda er forskellige hver gang.

Adressen er det sted i hukommelsen hvor variablen bliver "gemt" - der er et system i tallene, men vi kan bare betragte dem som "numre" ligesom husnumre, og det er ligemeget præcis hvilken værdi de har, så længe vi kan se at de er forskellige.

Erklæring og initialisering af variable

Når der bliver erklæret en variabel, som fx her:

```
int a = 1234;
```

så betyder **`int a`** at der skal reserveres en adresse til variablen **`a`** - med andre ord så beslutter compileren fra nu af at variabel **`a`** svarer til en bestemt adresse, fx 1005 i ovennævnte tegning.

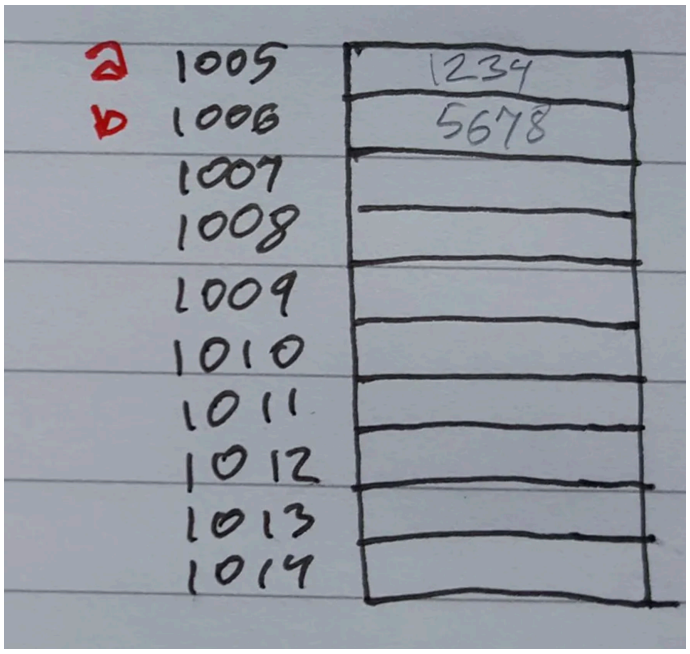
Og når der står **`a = 1234`**, så betyder det at værdien 1234 skal skrives ind på **`a`**'s adresse, som her er 1005.

Og når der efterfølgende står:

```
int b = 5678;
```

så betyder det at der nu skal reserveres en ledig adresse - for eksempel 1006 til **`b`**, og værdien 5678 skrives ind på den adresse.

Det kan illustreres med et kort over hukommelsen med adresser og værdier:



Nu har du beskrevet hvad der sker i hukommelsen, når der står dette i programmet:

```
int a = 1234;
int b = 5678;
```

Når programmet "bruger" et variabelnavn, som fx i:

```
printf("Value of   a: %d\n", a);
```

Så læser det indholdet af a's adresse (1005) og bruger værdien der står der, altså 1234.

Når der står **&** foran et variabelnavn, som fx **i**, så betyder det at det er selve adressen, altså 1005, og ikke indholdet, 1234, vi er interesseret i:

```
printf("Address of a: %d\n", &a);
```

Når man fodrer **printf** med **%d** giver den advarsler, fordi det "rigtige" er at bruge **%p** til adresser. Men behold **%d** hvis du vil undgå hexadecimale tal, som **%p** producerer.

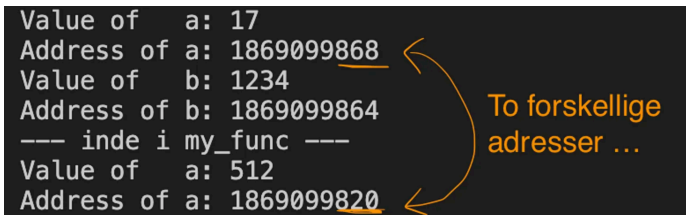
Scopes og stack-frames: Funktioner har deres egne variable

I eksemplet nedenfor kan man se at **my_func** har en variabel **a** som ikke har noget at gøre med variablen **a** som står i **main**, selvom navnet er det samme.

```
void my_func()
{
  int a = 512;

  printf("--- inde i my_func ---\n");
  printf("Value of   a: %d\n", a);
  printf("Address of a: %d\n", &a);
  printf("--- afslut my_func ---\n");
}

int main()
{
  int a = 1234;
  int b = 5678;

  printf("Value of   a: %d\n", a);
  printf("Address of a: %d\n", &a);
  printf("Value of   b: %d\n", b);
  printf("Address of b: %d\n", &b);

  my_func();

  printf("--- efter my_func ---\n");
  printf("Value of   a: %d\n", a);
  printf("Address of a: %d\n", &a);
  printf("Value of   b: %d\n", b);
  printf("Address of b: %d\n", &b);
```

```
  return 0;
}
```

Kør programmet og bemærk at de to **a**'er har forskellige adresser:



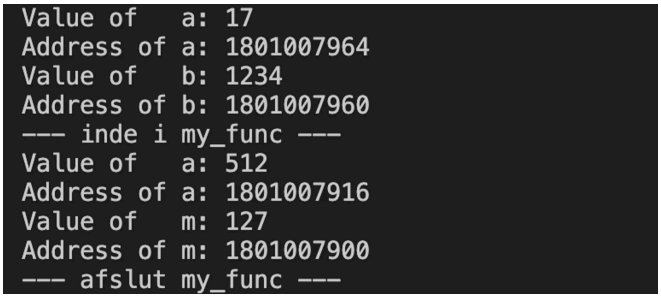
Ligesom vi kan have flere personer med samme navn, fx to studerende der hedder Emil, kan vi også have flere variabler med samme navn, det er stadig adressen der er variablen, navnet er bare en nem måde at "komme i kontakt" med adressen.

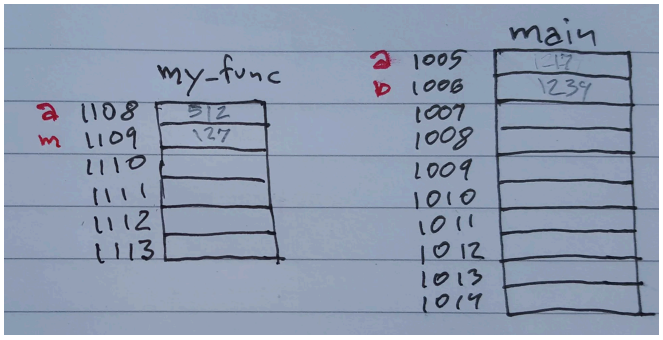
- **Scope: `my_func()`** kan tilgå variable i dens eget memory-område, men ikke i **`main()`**'s
- **Stack-frame: `my_func()`** reserverer plads i hukommelsen til sine egne variable

Parametre til funktioner er bare lokale variable

Parameteren **`m`** har fået en adresse ligesom variablen **`a`**. Det vil altså sige at det også bare er en lokal variabel, så tegn den ind i dit diagram sammen med **`a`**:

```
void my_func(int m)
{
  int a = 512;

  printf("--- inde i my_func ---\n");
  printf("Value of   a: %d\n", a);
  printf("Address of a: %d\n", &a);
  printf("Value of   m: %d\n", m);
  printf("Address of m: %d\n", &m);
  printf("--- afslut my_func ---\n");
}
```





Strings og string-funktioner

- C har ikke en datatype der hedder "string"
- I stedet betragter man arrays af characters, dvs. **`char[]`**, som strings
- Man kan initialisere et array, men arrays er ikke værdier i sig selv
- Man kan ikke overskrive et helt array med én operation efter det er initialiseret
- Arrays (og derfor strings) kan ikke sammenlignes for lighed med **`==`**, man skal loope

Grundlæggende syntaks for `char[]` og `char *`

Der er flere måder at arbejde med strings på i C:

1. Character array med fast størrelse

```
char name[20];  // Plads til 19 tegn + null-terminator
name[0] = 'A';
name[1] = 'n';
name[2] = 'n';
name[3] = 'a';
name[4] = '\0';  // Null-terminator
```

2. Initialisering med string literal

```
char greeting[] = "Hello";  // Størrelse beregnes automatisk (6 bytes)
char message[50] = "World"; // Resten fyldes med '\0'
```

3. Pointer til string literal (read-only)

```
char *text = "Hello World";  // Pointer til read-only memory
// text[0] = 'h';  // FEJL: Kan ikke ændre string literals!
```

4. Forskellen mellem `char[]` og `char *`

```
char arr[] = "Hello";   // Array: Data kopieres til stakken, kan ændres
char *ptr = "Hello";    // Pointer: Peger på read-only string literal

arr[0] = 'h';  // OK: Ændrer kopi på stakken
// ptr[0] = 'h';  // FEJL: String literals er read-only
```

Vigtige regler

- Alle C strings skal slutte med **`'\0'`** (null-terminator)
- String literals som **`"Hello"`** er read-only og må ikke ændres
- Arrays allokeret på stakken (**`char arr[10]`**) kan ændres frit

Gennemgang af standard string-funktioner

Fra Modern C s. 81-82 er her en række funktioner der virker på C strings:

- **`strlen(s)`** - returnerer længden af string **`s`** (antal karakterer før null-terminator)
- **`strcpy(target, source)`** - kopierer hele string **`source`** til **`target`** (inklusiv **`'\0'`**)
- **`strncpy(target, source, len)`** - kopierer maksimalt **`len`** tegn fra **`source`** til **`target`**
- **`strdup(source)`**, **`strndup(source, len)`** - allokerer og kopierer **`source`** til heap'en
- **`strcmp(s0, s1)`** - sammenligner to strings leksikografisk, returnerer 0 hvis ens

Kommandolinjeargumenter

Når man kører et program i terminalen, har man mulighed for at angive argumenter til programmet. For eksempel, hvis man skriver **`git push`**, så er **`git`** navnet på programmet, og **`push`** er første parameter. Hvis man skriver **`git push --force`**, så er **`--force`** andet argument til kommandoen.

I C-programmer er **`argc`** (argument count) antallet af kommandolinjeargumenter, og **`argv`** (argument vector) er et array af strings med argumenterne. Det første argument **`argv[0]`** er altid programmets eget navn.

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Programnavn: %s\n", argv[0]);
    printf("Antal argumenter: %d\n", argc);

    for (int i = 1; i < argc; i++) {
        printf("Argument %d: %s\n", i, argv[i]);
    }

    return EXIT_SUCCESS;
}
```

Hvordan udtaler man **`char *argv[]`**? Spørg https://cdecl.org/!

Hvis programmet kaldes med **`./program hello world`**, vil output være:

```
Programnavn: ./program
Antal argumenter: 3
Argument 1: hello
Argument 2: world
```

`EXIT_SUCCESS` og **`EXIT_FAILURE`** er makroer defineret i **`<stdlib.h>`** der bruges som returværdier fra **`main()`** for at signalere om programmet er kørt succesfuldt eller ej.

- **`EXIT_SUCCESS`** (værdi 0) signalerer at programmet er kørt uden fejl
- **`EXIT_FAILURE`** (værdi 1) signalerer at programmet er afsluttet med en fejl

```c
#include <stdlib.h>

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Fejl: Mangler argumenter\n");
        return EXIT_FAILURE;
    }

    printf("Programmet kører succesfuldt\n");
    return EXIT_SUCCESS;
}
```

Øveopgave: Skriv din egen `strlen()`, `strcpy()` og `strlower()`

For at forstå hvordan C's standard string-funktioner virker, skal vi implementere vores egne versioner.

`my_strlen()` - Find længden af en string

```c
size_t my_strlen(const char *s) {
    size_t length = 0;
    while (s[length] != '\0') {
        length++;
    }
    return length;
}
```

Funktionen looper gennem string'en indtil den finder null-terminatoren **`'\0'`**.

`my_strcpy()` - Kopiér en string

```c
char *my_strcpy(char *dest, const char *src) {
    int i = 0;
    while (src[i] != '\0') {
        dest[i] = src[i];
        i++;
    }
    dest[i] = '\0';  // Husk null-terminator!
    return dest;
}
```

Funktionen kopierer hvert tegn fra **`src`** til **`dest`** indtil null-terminatoren nås.

`my_strlower()` - Konvertér til små bogstaver

```c
char *my_strlower(char *s) {
    for (int i = 0; s[i] != '\0'; i++) {
```

```
        if (s[i] >= 'A' && s[i] <= 'Z') {
            s[i] = s[i] + ('a' - 'A');  // Konvertér til lowercase
        }
    }
    return s;
}
```

Funktionen ændrer store bogstaver til små ved at udnytte at ASCII-værdierne for 'a'-'z' er 32 højere end 'A'-'Z'.

Test de tre funktioner:

```
#include <stdio.h>

int main() {
    char str1[] = "Hello World";
    char str2[50];

    printf("Længde af '%s': %zu\n", str1, my_strlen(str1));

    my_strcpy(str2, str1);
    printf("Kopieret string: '%s'\n", str2);

    my_strlower(str2);
    printf("Små bogstaver: '%s'\n", str2);

    return 0;
}
```

Output:

```
Længde af 'Hello World': 11
Kopieret string: 'Hello World'
Små bogstaver: 'hello world'
```

Structs og struct pointers

Fordi vi bliver ved med at bruge structs til at lave datastrukturer, ender vi med at bruge noget der hedder *struct pointers*, som er en syntaks til at adressere felterne i en struct, hvis man har en pointer til en struct i stedet for selve struct'en.

Når vi arbejder med structs, er der to måder at tilgå felterne på:

```
#include <stdio.h>

struct person_t {
    char name[50];
    int age;
};

int main() {
    // 1. Almindelig struct (stack-allokeret)
    struct person_t person1;
    person1.age = 25;  // Brug . (dot) operator

    // 2. Pointer til struct
    struct person_t *person_ptr = &person1;

    // To måder at tilgå felter via pointer:
    (*person_ptr).age = 26;  // Explicit dereference med .
    person_ptr->age = 27;    // Arrow operator -> (nemmere!)

    printf("Alder: %d\n", person1.age);
    printf("Alder via pointer: %d\n", person_ptr->age);

    return 0;
}
```

Hvornår bruger man hvad?

```
struct person_t person;      // Stack-allokeret struct
struct person_t *ptr;        // Pointer til struct

person.age = 30;             // Brug . når du har struct direkte
ptr->age = 30;               // Brug -> når du har pointer til struct
```

Med dynamisk allokering

```
#include <stdlib.h>

struct person_t *create_person(char *name, int age) {
    struct person_t *p = malloc(sizeof(struct person_t));

    // Brug -> fordi p er en pointer
```

```
    strcpy(p->name, name);
    p->age = age;

    return p;
}

int main() {
    struct person_t *person = create_person("Anna", 25);

    printf("%s er %d år gammel\n", person->name, person->age);

    free(person);
    return 0;
}
```

Tommelfingerregel:

- Har du en struct? Brug `.` (dot)
- Har du en pointer til struct? Brug `->` (arrow)

Øveopgave: RGB hex-parser

Skriv en funktion **`parse_rgb()`** der tager imod en string med en hex-farvekode og returnerer en **`struct rgb_t`**:

```
#ifndef _RGB_H
#define _RGB_H
#include <stdint.h>

struct rgb_t {
  uint8_t r;
  uint8_t g;
  uint8_t b;
}

struct rgb_t parse_rgb(char[]);

#endif // _RGB_H
```

Funktionen med signaturen **`struct rgb_t parse_rgb(char[]);`** har svært ved at fejle, selvom inputtet kan være både for kort eller ugyldigt. Hvis man i C vil returnere både om noget er gået godt, og hvad der er gået godt, skriver man fx **`bool parse_rgb(char[], struct rgb_t *);`** i stedet for.

Her bliver **`parse_rgb()`** kaldt med forskelligt input:

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

int main() {
    struct rgb_t color1 = parse_rgb("#FF0000");
    printf("Rød: R=%d, G=%d, B=%d\n", color1.r, color1.g, color1.b);

    struct rgb_t color2 = parse_rgb("#00FF00");
    printf("Grøn: R=%d, G=%d, B=%d\n", color2.r, color2.g, color2.b);

    struct rgb_t color3 = parse_rgb("#0000FF");
    printf("Blå: R=%d, G=%d, B=%d\n", color3.r, color3.g, color3.b);

    struct rgb_t color4 = parse_rgb("#FFFFFF");
    printf("Hvid: R=%d, G=%d, B=%d\n", color4.r, color4.g, color4.b);

    return EXIT_SUCCESS;
}
```

Struct-pointere

Når vi arbejder med pointere til structs, skal vi ofte tilgå felterne i struct'en. Der er to måder at gøre dette på:

1. **Explicit dereferencering**: **`(*pointer).field`**
2. **Arrow operator**: **`pointer->field`**

Arrow operatoren **`->`** er bare syntaktisk sukker for den første metode.

Begge metoder gør præcis det samme.

Men arrow operatoren er meget mere læselig og bruges derfor næsten altid.

```
struct rgb_t color = {255, 0, 0};
struct rgb_t *color_ptr = &color;

// Begge linjer gør det samme:
printf("Rød: %d\n", (*color_ptr).r);  // Explicit dereferencering
printf("Rød: %d\n", color_ptr->r);    // Arrow operator (foretrukket)
```

Arrow operatoren er særligt nyttig når man har flere niveauer af pointere eller komplekse udtryk:

```
// Svært at læse:
(*(*complex_ptr).inner_ptr).value = 42;
```

```
// Meget mere læseligt:
complex_ptr->inner_ptr->value = 42;
```

I vores **`parse_rgb()`** funktion bruger vi arrow operatoren til at sætte felterne i den struct vi får som parameter:

```
result->r = byte_value;  // I stedet for (*result).r = byte_value;
```

Lad os lige omskrive funktionen til at håndtere fejl:

RGB-hexparser med fejlhåndtering

```
#include <stdbool.h>
#include <string.h>
#include <stdlib.h>

// Hjælpefunktion til at konvertere et hex-tegn til værdi (0-15)
int hex_char_to_value(char c) {
    if (c >= '0' && c <= '9') return c - '0';
    if (c >= 'A' && c <= 'F') return c - 'A' + 10;
    if (c >= 'a' && c <= 'f') return c - 'a' + 10;
    return -1; // Ugyldigt tegn
}

// Hjælpefunktion til at parse to hex-tegn til en byte
bool parse_hex_byte(char high_char, char low_char, uint8_t *result) {
    int high_nibble = hex_char_to_value(high_char);
    int low_nibble = hex_char_to_value(low_char);

    if (high_nibble == -1 || low_nibble == -1) {
        return false;
    }

    // Kombinér de to hex-cifre til ét tocifret hex-tal
    *result = (high_nibble << 4) | low_nibble;
    return true;
}

bool parse_rgb(char input[], struct rgb_t *result) {
    // Tjek at input starter med '#' og er lang nok
    if (input[0] != '#' || strlen(input) != 7) {
        return false;
    }

    // Parse hver farvekomponent (2 hex-tegn ad gangen)
    if (!parse_hex_byte(input[1], input[2], &result->r) ||
        !parse_hex_byte(input[3], input[4], &result->g) ||
        !parse_hex_byte(input[5], input[6], &result->b)) {
```

```
        return false;
    }

    return true;
}
```

Eksempel på brug:

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

int main() {
    struct rgb_t color;

    if (parse_rgb("#FF0000", &color)) {
        printf("Rød farve: R=%d, G=%d, B=%d\n", color.r, color.g,
color.b);
    } else {
        printf("Fejl ved parsing af farve\n");
    }

    if (parse_rgb("ugyldig", &color)) {
        printf("Dette skulle ikke ske\n");
    } else {
        printf("Korrekt fejlhåndtering\n");
    }

    return EXIT_SUCCESS;
}
```

Lidt om sikkerhed og bufferlængder

Vi har indtil nu kun printet til skærmen med **`printf()`**.

Der findes en anden meget nyttig funktion, som skriver til en string: **`sprintf()`**.

Den har et lille problem:

```
char buffer[10];
char name[] = "Simon Shine";

// FARLIGT: sprintf() tjekker ikke bufferstørrelse
sprintf(buffer, "Hej %s!", name);  // Buffer overflow!
printf("Buffer: %s\n", buffer);    // Upålidelig adfærd

// SIKKERT: snprintf() begrænser output til bufferstørrelse
snprintf(buffer, sizeof(buffer), "Hej %s!", name);
printf("Buffer: %s\n", buffer);    // Output: "Hej Simon"
```

`sprintf()` skriver uden at tjekke om der er plads i bufferen, hvilket kan føre til buffer overflow og sikkerhedsproblemer.

`snprintf()` tager en maksimal længde som parameter og afkorter output hvis det er for langt. Dette forhindrer buffer overflow og gør koden mere sikker.

Stak-allokering og heap-allokering af strings

Man kan allokere på to måder i C:

- **Stak-allokering:** Ved at oprette variable i dit lokale scope
- **Heap-allokering:** Ved at kalde **`malloc()`** og få en adresse tilbage

Husk at strings stopper ved den første NUL-byte (**`'\0'`**), som ikke nødvendigvis er det samme sted som hvor allokeringen stopper. Det er muligt at en string ikke fylder hele allokeringen, eller at man overskrider allokeringen og skriver sit array ind i den efterfølgende hukommelse.

```
int main() {
  // Allokér på stakken, dvs.
  // den bliver smidt væk når `main()` returnerer
  char arr[11];
  arr[0] = '\0';

  // Allokér på heap'en, dvs.
  // den bliver smidt væk når der kaldes `free()`
  char *arr = malloc(11 * sizeof(char));
  *arr = '\0';
  free(arr);
  arr = NULL;
}
```

- Her bliver 11 bytes allokeret på de to forskellige måder
- Efterfølgende bliver den første byte sat til **`'\0'`** som gør dem til tomme strenge, “”
- Allokeringerne er i begge tilfælde 11 bytes, dvs. med plads til 10 tegn
- Det er tilfældet også selvom strengene er tomme
- De overskydende bytes er bare spildt / gemt til senere

Structs, struct pointers og linkede lister

1. Introduktion til structs

En struct er en måde at gruppere relaterede data sammen. Her er et simpelt eksempel:

```
struct person_t {
    char *name;
    int age;
};
```

Vi kan oprette og bruge en struct på stakken sådan her:

```
#include <stdio.h>

struct person_t {
    char *name;
    int age;
};

int main(void) {
    // stak-allokeret struct
    struct person_t person;
    person.name = "Alice";
    person.age = 25;

    printf("Navn: %s, Alder: %d\n", person.name, person.age);
    return 0;
}
```

Bemærk at vi bruger `.` operatoren til at få adgang til felterne i struct'en.

2. Problemer med stak-allokerede structs

Stak-allokerede structs har en stor begrænsning:

De lever kun så længe som den funktion de er oprettet i.

Se det her eksempel:

```
struct person_t *create_person_wrong(void) {
    struct person_t person;  // stak-allokeret!
    person.name = "Bob";
    person.age = 30;

    return &person;  // FARLIGT! Returnerer adresse til stak-variable
}
```

Når funktionen returnerer, bliver **`person`** variablen ødelagt, og pointeren vi returnerer peger på ugyldig hukommelse.

3. Heap-allokering med malloc() og free()

Løsningen er at bruge heap-allokering med **`malloc()`**:

```
#include <stdlib.h>

struct person_t *create_person_correct(char *name, int age) {
    // Heap-allokering: lever indtil vi kalder free()
    struct person_t *person = malloc(sizeof(struct person_t));
    person->name = name;  // Bemærk -> i stedet for .
    person->age = age;

    return person;
}
```

Vigtige punkter om **`malloc()`** og **`free()`**:

- **`malloc(size)`** allokerer hukommelse, målt i antal bytes, på heap'en
- Heap-allokeret hukommelse lever indtil vi kalder **`free()`**
- Hver **`malloc()`** skal matches med præcis én **`free()`**
- Hukommelse lækker hvis vi glemmer **`free()`**

```
int main(void) {
    struct person_t *person = create_person_correct("Charlie", 35);
    printf("Navn: %s, Alder: %d\n", person->name, person->age);

    free(person);  // Frigør hukommelsen!
    return 0;
}
```

4. Struct pointers og -> operatoren

Når vi har en pointer til en struct, kan vi bruge to forskellige syntakser:

```
struct person_t *person_ptr = malloc(sizeof(struct person_t));

// Disse to linjer gør det samme:
(*person_ptr).age = 25;  // Dereference først, så adgang til felt
person_ptr->age = 25;    // Shorthand notation - meget mere læseligt!
```

`->` operatoren er syntaktisk sukker for **`(*ptr).field`** og bruges når man arbejder med struct pointers.

5. Structs inde i structs - Selvreferencerende structs

Her bliver det interessant! Vi kan lave structs der peger på andre structs af samme type:

```
struct list_node_t {
    int value;
    struct list_node_t *next;  // Pointer til næste node
};
```

Det er grundlaget for linkede datastrukturer. En node kan pege på en anden node, som kan pege på en tredje, osv. Selvom datastrukturen er rekursivt defineret, kan man selv vælge om man vil bruge rekursive funktioner, eller loops, til at traversere dem.

6. Bygning af en linket liste

Lad os bygge en simpel linket liste trin for trin.

Trin 1: Opret en enkelt node

Den her funktion allokerer en enkelt node og udfylder dens felter.

De kan tolkes som lister af længde 1, fordi **`node->next`** er **`NULL`**.

```
struct list_node_t *create_node(int value) {
    struct list_node_t *node = malloc(sizeof(struct list_node_t));
    node->value = value;
    node->next = NULL;  // Ingen næste node endnu
    return node;
}
```

Trin 2: Sammenkæd noder manuelt

Inden vi opretter flere nyttige hjælpefunktioner, kan vi faktisk allerede lave linkede lister med vores ene hjælpefunktion, hvor sammenkædningen foregår manuelt.

```
int main(void) {
    struct list_node_t *first = create_node(42);
    struct list_node_t *second = create_node(43);
    struct list_node_t *third = create_node(44);

    // Sammenkæd noderne
    first->next = second;
    second->next = third;
    third->next = NULL;

    // Print listen
    struct list_node_t *current = first;
    while (current != NULL) {
        printf("%d -> ", current->value);
```

```
        current = current->next;
    }
    printf("NULL\n");

    // Husk at frigøre hukommelsen!
    free(first);
    free(second);
    free(third);

    return 0;
}
```

Trin 3: Push-funktion til at tilføje elementer

For at indsætte elementer for enden af listen uden at skulle holde styr på alle kædeleddene manuelt, kan man loope til slutningen af listen og indsætte en node dér.

Vi looper indtil **`current->next != NULL`** fordi **`current`** derfor er det sidste led i den eksisterende liste.

```
void push_list(struct list_node_t *head, int value) {
    // Find sidste node
    struct list_node_t *current = head;
    while (current->next != NULL) {
        current = current->next;
    }

    // Opret ny node og sammenkæd
    struct list_node_t *new_node = create_node(value);
    current->next = new_node;
}
```

Trin 4: Pop-funktion til at fjerne elementer

For at fjerne det sidste element fra listen skal vi både:

1. Finde den sidste node (for at frigøre den med **`free()`**)
2. Finde den næstsidste node (for at sætte dens **`next`** til **`NULL`**)

Det er derfor vi bruger to pointere: **`previous`** og **`current`**. De bevæger sig sammen gennem listen, hvor **`current`** altid er ét skridt foran **`previous`**. Når **`current`** når det sidste element, peger **`previous`** på det næstsidste - præcis hvad vi har brug for!

```
int pop_list(struct list_node_t *head) {
    // Find næstsidste og sidste node
    struct list_node_t *previous = head;
    struct list_node_t *current = head->next;

    while (current->next != NULL) {
```

```
        previous = previous->next;
        current = current->next;
    }

    // Gem værdien
    int value = current->value;

    // Fjern sidste node
    free(current);
    previous->next = NULL;

    return value;
}
```

7. Fra lister til træer

Linkede lister er kun begyndelsen. Vi kan udvide konceptet til træer ved at have flere pointere i hver node:

```
struct tree_node_t {
    int value;
    struct tree_node_t *left;   // Venstre barn
    struct tree_node_t *right;  // Højre barn
};
```

Eller stamtræer som i afleveringsopgaven:

```
struct ancestry_node_t {
    struct ancestry_node_t *mom;    // Mor
    struct ancestry_node_t *dad;    // Far
    struct person_t *person;        // Personen i denne node
};
```

Vigtige takeaways

1. **Stack vs Heap**: stak-allokerede structs lever kun indtil funktionen returnerer.
2. **Memory management**: Hver `malloc()` skal matches med `free()` for at undgå leaks.
3. **Struct pointer syntax**: Brug `->` til at få adgang til felter gennem pointere.
4. **Selvreferencerende structs**: Muliggør linkede datastrukturer som lister og træer.

Binære tal, binær repræsentation

Vi skal forstå bits, bytes og binære tal. Vi skal se på hvordan negative tal og floating-point tal er repræsenteret med 0′er og 1′er, og forstå hvorfor *signed* tal kun kan have halvt så store værdier som *unsigned* tal. Og vi skal forstå de spring der er i adresserne for

variable: Når man lægger 1 til en adresse, og der er mere end 1 til forskel, når man printer adressernes talværdier.

Emner som bliver dækket:

- Positionelle talsystemer
- Konvertering til binær i C
- Two's complement: Repræsentation af negative tal
- Bitvise operatorer (and, or, not, shifting)
- Konvertere til/fra ASCII: **`atoi()`** og **`itoa()`**

Positionelle talsystemer

Positionelle talsystemer er forklaret i den her YouTube:

https://www.youtube.com/watch?v=LpuPe81bc2w

Her kommer den samme forklaring på tekst:

Det Decimale System (Base 10)

I titalsystemet har vi 10 forskellige cifre: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Når vi skriver tallet 1234, betyder det faktisk:

$$1234 = 1 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$$

$$1234 = 1 \times 1000 + 2 \times 100 + 3 \times 10 + 4 \times 1$$

$$1234 = 1000 + 200 + 30 + 4$$

Hver position i tallet har en **værdi** baseret på dens placering:

- Enerposition: $10^0 = 1$
- Tierposition: $10^1 = 10$
- Hundrederposition: $10^2 = 100$
- Tusindposition: $10^3 = 1000$

Det Binære System (Base 2)

Det binære talsystem er baseret på tallet 2 i stedet for 10. Det betyder, at vi kun har 2 cifre til rådighed: **0** og **1**.

I det binære system er positionsværdierne baseret på potenser af 2 i stedet for potenser af 10:

Position	4	3	2	1
Værdi	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$

Lad os se på det binære tal **1101**:

$$1101_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

$$1101_2 = 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1$$

$1101_2 = 8 + 4 + 0 + 1 = 13_{10}$

Så det binære tal 1101 svarer til det decimale tal 13.

Øveopgave: Talkonvertering

For at se hvordan en **`int`** ser ud i binær, kan man printe de enkelte bits i et tal.

Det gør man ved at loope hen over bits'ene. For at gøre det, skal man afgøre hvor mange bits der er.

Man kan finde antallet af bytes for en datatype som **`int`** vha. makroen **`sizeof(int)`**.

Der er 4 bytes i en **`int`**, og 8 bits i en byte. Så antallet af bits i en **`int`** findes ved **`sizeof(int) * 8`**.

Foruden at loope fra 0 til 31, skal vi også bruge nogle specielle **bitvise operatorer**:

- **`>>`** (og tilsvarende **`<<`**) kaldes shift-operatorer, fordi de skubber bits til venstre og højre
- **`&`** (og tilsvarende **`|`**) kaldes **bitwise and** (hhv. **bitwise or**) og arbejder på enkelte bits

Hvordan fisker man en enkelt bit ud af et tal vha. >> og &

Bitwise operators er forklaret yderligere i den her Wikipedia-artikel, som vi gennemgår:

https://en.wikipedia.org/wiki/Bitwise_operation

Men nedenfor er der en forklaring af **`>>`** og **`&`** som vi får brug for til at fiske enkelte bits ud med.

Lad os se på tallet 1234 (decimal) konverteret til binær:

1234 decimal = 10011010010 binær

I en 32-bit **`int`** ser det sådan ud (med alle 32 bit-pladser):

```
0000 0000 0000 0000 0000 0100 1101 0010
```

Jeg vil gerne fiske den 8. bit ud (talt fra højre, startende ved 0):

```
0000 0000 0000 0000 0000 0100 1101 0010
                              ^- den her bit
```

1. Først fjerne alle bits med lavere position ved at shift'e tallet 7 pladser til højre (**`>>`**)
2. Derefter fjerne alle bits med højere position ved at bruge bitwise AND med 1 (**`& 1`**)

Trin 1: Fjern lavere bits med >>

```
1234 >> 7 = 0000 0000 0000 0000 0000 0000 0000 1001 = 9
```

Trin 2: Fjern højere bits med `& 1`

```
7 & 1 = 0000 0000 0000 0000 0000 0000 0000 1001
      & 0000 0000 0000 0000 0000 0000 0000 0001
      = 0000 0000 0000 0000 0000 0000 0000 0001 = 1
```

Så den 8. bit i 1234 er 1.

Lad os prøve med den 11. bit i stedet:

```
1234 >> 10 = 0000 0000 0000 0000 0000 0000 0000 0001 = 1
1 & 1 = 1
```

Så den 11. bit er 1, hvilket passer med den binære repræsentation ovenfor.

```c
#include <stdio.h>

void print_binary(int num) {
  int bits = sizeof(int) * 8;
  printf("%u: ", num);

  for (int i = bits - 1; i >= 0; i--) {
    printf("%d", (num >> i) & 1);
    if (i % 4 == 0 && i > 0)
      printf(" ");
  }
  printf("\n");
}
```

For at gøre koden pæn printer vi et mellemrum efter hver fjerde bit, og til det benytter vi `%` (modulo)-operatoren. Men det er som sådan ikke noget der er nødvendigt, det er bare så det ser pænt ud.

Nu er det muligt at printe hvordan både positive og negative tal er repræsenteret med bits:

```c
int main() {
  printf("Binary representation of integers:\n\n");

  print_binary(0);
  print_binary(42);
  print_binary(-1);
  print_binary(-42);
  print_binary(2147483647);
  print_binary(2147483648U);
  print_binary(4294967295U);

  return 0;
}
```

Og ser man resultatet:

```
Binary representation of integers:

0: 0000 0000 0000 0000 0000 0000 0000 0000
42: 0000 0000 0000 0000 0000 0000 0010 1010
-1: 1111 1111 1111 1111 1111 1111 1111 1111
-42: 1111 1111 1111 1111 1111 1111 1101 0110
2147483647: 0111 1111 1111 1111 1111 1111 1111 1111
-2147483648: 1000 0000 0000 0000 0000 0000 0000 0000
-1: 1111 1111 1111 1111 1111 1111 1111 1111
```

Det var måske ikke lige det du regnede med i alle tilfælde.

Negative tal er repræsenteret på en lidt sjov måde:

De er lavet med en konvention som hedder **two's complement**.

Two's complement

Two's Complement bliver gennemgået i det her tutorial, som vi gennemgår:

https://www.cs.cornell.edu/~tomf/notes/cps104/twoscomp.html

Formålet med afsnittet her er, at vi gerne vil vide, hvordan −42′s binære repræsentation ser ud med two's complement, da det er sådan negative tal virker på vores computere.

For at gøre et tal negativt med two's complement udføres to trin:

1. Man flipper alle bits'ene
2. Man lægger 1 til tallet

Eksempel: 42 er 0010 1010 i binær.

Flipper vi alle bits'ene, kan vi bruge NOT-operatoren, dvs. **~** i C, og så bliver det til:

```
~ 0010 1010
= 1101 0101
```

Og lægger vi 1 til:

```
+ 1101 0101
  0000 0001
= 1101 0110
```

Motivationen bag two's complement

Pointen med denne sære bagvendte måde at repræsentere negative tal er, at hvis man vælger at tolke den forreste bit som fortegnsminus, men resten af tallene på samme måde som før (ikke inverterede), dvs.:

```
1000 0000 0000 0001
^- minus...       ^- et.
```

så virker almindelig addition ikke længere, fordi man ikke kan lægge to negative tal sammen uden at skulle håndtere fortegnsminus særskilt.

Men bruger man den her sære two's complement-representation: invertér tallet og læg 1 til, så virker addition af negative tal:

```
-1 + -1 i binær:
  1111 111  <- in mente
  1111 1111
+ 1111 1111
= 1111 1110 (som gerne skulle være den binære repræsentation for -2)
```

Og for lige at regne efter, hvis vi gør tallet 2 negativt:

```
NOT 0000 0010 (tallet 2 i binær)
    1111 1101
  + 0000 0001
  = 1111 1110 = -2
```

Det giver altså det samme at regne $-1 + -1$ og at konvertere 2 til negativ.

Så derfor er vi glade for two's complement, selvom den binære repræsentation bliver sværere at forstå med alle de 1´ere.

Øvelse: Linux filrettigheder og datamodellering

Nu skal vi kombinere det vi har lært om binære tal, bitwise operatorer og strukturer til at arbejde med Linux filrettigheder.

Linux filrettigheder som bits

I Linux har hver fil tre typer rettigheder for tre forskellige grupper:

- **Read (r)**: Læse filen (bit værdi 4)
- **Write (w)**: Skrive til filen (bit værdi 2)
- **Execute (x)**: Eksekvere/køre filen (bit værdi 1)

For tre grupper:

- **Owner**: Ejeren af filen
- **Group**: Gruppens medlemmer
- **Others**: Alle andre brugere

Disse rettigheder kan repræsenteres som 9 bits i alt, eller mere praktisk som 3 oktale cifre (0-7).

Eksempel: `chmod 755` betyder:

- Owner: 7 = 111_2 = read+write+execute

- Group: 5 = 101_2 = read+execute
- Others: 5 = 101_2 = read+execute

2. Bitmasker

Bitmasker er bare tal som bliver brugt til at modificere andre tal vha. bitvise operatorer.

Fordi deres formål er at slå bits til eller fra på faste positioner, er det nemmere at udtrykke tallet i binær, hex eller oktal. Jeg sagde at oktaler ikke bliver brugt så meget, og så valgte jeg alligevel det ene eksempel hvor de faktisk bliver brugt: chmod.

Bitmasker kan benyttes for alle taltyper, men de giver mest mening at bruge på *unsigned* typer, fx:

- **`unsigned`**
- **`char`**
- **`uint8_t`**

Her er nogle forskellige måder at lave en bitmaske der aktiverer den sjette bit i en **`char`**:

```
char sixth_bit = 32;          // decimal
char sixth_bit = 0x20;        // hex
char sixth_bit = 040;         // oktal
char sixth_bit = 0b00100000; // binær
char sixth_bit = 1 << 5;      // bit-shifting
```

En bitmaske kan have vilkårligt mange bits slået til. Ovenstående har kun sjette bit slået til.

3. Oktaler

Filrettigheder som **`0755`**, **`0644`**, **`0777`** er faktisk oktal-værdier der kan oversættes til bitmasker.

Grunden til at oktal er et godt valg her er, at ét oktal-ciffer har 3 bits, og det er præcis 3 filrettigheder vi prøver at slå til og fra:

Oktal	Binær	Beskrivelse
0	000	Ingen rettigheder
1	001	Execute (x)
2	010	Write (w)
3	011	Write + Execute (wx)
4	100	Read (r)
5	101	Read + Execute (rx)
6	110	Read + Write (rw)
7	111	Read + Write + Execute (rwx)

Derfor kan **`0755`** skrives som **`0b111'101'101`** og læses som:

- **7** (owner): **111** = rwx
- **5** (group): **101** = r-x
- **5** (other): **101** = r-x

Og **0644** kan skrives som **0b110'100'100** og læses som:
- **6** (owner): **110** = rw-
- **4** (group): **100** = r–
- **4** (other): **100** = r–

Hvis man vil lave en bitmaske, der matcher read-bit'en for "user", er det derfor **0b100'000'000**.

```
#define USER_READ 0400 // 0b100'000'000
```

4. Bitvise operatorer (&, |, ~)

Vi ønsker at gøre en af to ting: Vi vil gerne tilføje nogle bits, eller fjerne nogle bits.

Vi prøver ikke at gøre begge dele samtidigt, selvom det også kan lade sig gøre.

Hvis man vil tilføje eller fjerne bits for et sæt rettigheder, skal man kombinere:

- et sæt rettigheder
- en bitmaske
- en eller flere bitvise operatorer

Her er to huskeregler:

Man bruger OR (|) til at tænde for bits

Hvis vi fx gerne vil tænde for den 8. bit i værdien **0b0001'1010**:

```
  0b0001'1010    <- værdien
| 0b1000'0000    <- bitmasken
= 0b1001'1010    <- resultatet
```

Det er altså nok at OR'e med en bitmaske, der har en enkelt bit sat, for at tænde for den bit.

Man bruger AND (&) til at slukke for bits

Hvis vi i stedet vil slukke for den 4. bit i værdien **0b0001'1010**:

```
  0b0001'1010    <- værdien
& 0b1111'0111    <- bitmasken
= 0b0001'0010    <- resultatet
```

Man AND'er med en bitmaske, hvor alle bits er sat, undtaget den der skal slukkes.

Man bruger NOT (~) til at flippe bits

Og derfor kommer der en krølle på det med at slukke for bits:

Hvad hvis man har en bitmaske med en enkelt bit, fx **`0b0000'1000`**, som man gerne vil slukke for?

Så kan man starte med at flippe den bitmaske ved at sætte **~** foran: **`~0b0000'1000 == 0b1111'0111`**.

```
   0b0001'1010                                  0b0001'1010
& ~0b0000'1000            svarer til          & 0b1111'0111
=  ?                                          = 0b0001'0010
```

Hvis du vil øve dig mere, kan du besøge https://github.com/ek-llc-25/bitmask-exercises, klone repository'et og rette **`SOLVE_THIS`** til det rigtige svar ved enten at bruge **`0b....'....`** eller **`~0b....'....`**

Hvad gør `&=` og `|=` i C kode?

Vi mangler kun ét stykke syntaks før vi har alt hvad vi skal bruge for at flippe filrettigheds-bits.

Du kender allerede operatorerne **`+=`** og **`-=`**:

```
a += 10; // Betyder: a = a + 10;
b -= 10; // Betyder: b = b - 10;
```

Den forkortelse kan man lave med alle operatorer inkl. de bitvise operatorer:

```
a &= 0700; // a = a & 0700; dvs. fjern alle rettigheder fra group og
other
b |= 0444; // b = b | 0444; dvs. tilføj read-adgang til user, group og
other
```

5. Struct pointers, fejlhåndtering og output-parametre

Vi er vant til at alle parametre som sendes til funktioner er **input**-parametre.

Hvis vi ikke tog højde for, at **`chmod()`** kan modtage ugyldigt input og fejle, kunne vi have brugt:

```
struct perms_t chmod(const char *input, struct perms_t perms) {
    ...
}
```

Altså en funktion som tager et sæt filrettigheder og et sæt ændringer, og returnerer et nyt sæt filrettigheder.

Men fordi vi derved ikke kan melde fejl, bruger vi returværdien til at berette om det gik godt (**`bool`**) og en pointer til en **`struct perms_t`** til output. Ved at rette i den **`struct perms_t`** der bliver peget på i parameteren, kan funktionen der kalder **`chmod()`** få sit resultat, og få at vide at det gik godt.

Den eneste krølle der er på struct-pointers, ud over at det virker lidt mærkeligt at bruge funktionsparametre til output, er at hvis man vil tilgå et felt i en struct *pointer*, skal man bruge **`->`** i stedet for **`.`**, eller også skal man dereferere pointeren hver gang:

```
// Først derefereres `perms` så man får en struct perms_t, og så tilgås
`bits`-feltet
(*perms).bits

// Her tilgås feltet direkte ved at `->` står for dereferering og tilgang
perms->bits
```

`perms->bits &= ~OTHER_READ;`

Hvis vi skal forstå overskriften, **`perms->bits &= ~OTHER_READ;`** betyder den altså:

```
// Tag den bit-flippede version af OTHER_READ, dvs. ~OTHER_READ, hvor i
stedet for
// den ene bit er sat, så er alle andre end den bit sat. AND det med
rettighederne
// for at slukke for lige præcis den bit.
perms->bits = perms->bits & ~OTHER_READ;

// Men fordi vi gemmer resultatet tilbage i samme variabel, kan det siges
kortere:
perms->bits &= ~OTHER_READ;
```

Make, CMake, objektfiler, linking, assembly

Men hvad sker egentlig der når man trykker på den store grønne play-knap i ens IDE?

Det korte svar er: Den grønne knap aktiverer programmet **`cmake`**, som bygger en Makefile der bliver kaldt med kommandoen **`make`**, som kalder C compileren **`gcc`**, der til sidst kalder linkeren **`ld`** som syer de forskellige programdele sammen til én executable.

Den her tutorial tager de tre linjers forklaring og udvider til de faktiske kommandoer.

Compile med “gcc”

Lav en fil med navnet main.c:

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

```c
int main() {
  printf("Hello, World!");
  return 0;
}
```

Compile og kør det med følgende kommandoer:

```
gcc -o hello hello.c
./hello
```

Flaget **`-o`** angiver output-filens navn. Uden **`-o`** ville gcc oprette en fil kaldet **`a.out`** som standard.

`./hello` kører den kompilerede executable. **`./`** betyder "i nuværende mappe", da Unix-systemer såsom Linux ikke inkluderer nuværende mappe i PATH af sikkerhedsgrunde.

PATH er en miljøvariabel der indeholder en liste af mapper, som systemet søger i efter kommandoer. Eksempel: **`which gcc`** viser at gcc findes i **`/usr/bin/gcc`**.

Compile med "make"

`make` automatiserer kompileringsprocessen. Når compile-kommandoer bliver lange og komplicerede (med flere flags som **`-Wall`**, **`-std=c99`**, **`-O2`**), er det praktisk at gemme dem i en Makefile i stedet for at skrive dem hver gang.

Lav en fil kaldet **`Makefile`** (uden filendelse):

```
hello: hello.c
  gcc -o hello hello.c

clean:
  rm -f hello
```

Bemærk at hvis du kopierer ovenstående fra PDF'en, fjerner den mellemrum. Bemærk også, at der skal være et **tab-tegn** (**ikke mellemrum**) før **`gcc`** og **`rm`**. Hvis du bare skriver mellemrum, virker filen ikke. Det gør Makefile lidt svært og gammeldags at arbejde med.

Nu kan du bygge programmet med **`make hello`** og rydde op med **`make clean`**.

Her er et avanceret eksempel på en Makefile:

```
CC = gcc
CFLAGS = -Wall -O2
INCLUDES = -I. -I/usr/include
LIBS = -lsqlite3
BIN = notez
```

```
CFILES = help.c duration.c db_callbacks.c db.c main.c
OBJS = ${CFILES:.c=.o}

all: ${CFILES} ${BIN}

${BIN}: ${OBJS}
  ${CC} ${CFLAGS} ${LIBS} ${OBJS} -o $@

.c.o:
  ${CC} ${CFLAGS} ${INCLUDES} -c $< -o $@

clean:
  rm -f ${OBJS} ${BIN}
```

Hvorfor skal det være så kringlet?

- Der benyttes variable i toppen for at undgå gentagelser. Hvis fx man benytter en anden C compiler end GCC, kan man rette det ét sted. Hvis man skifter advarsels-niveau (**`-Wall`**) eller optimerings-niveau (**`-O2`**), ønsker man at gøre det for al koden.
- Den nævner samtlige .c-filer som indgår i programmet. Det er mærkeligt eksplicit, men tillader at man kan have .c-filer liggende som ikke bliver oversat.
- Den definerer en variabel, **`OBJS`**, som indeholder en liste af filer som svarer til **`CFILES`**, men hvor alle filerne slutter på .o i stedet for på .c. **`OBJS`** indeholder altså **`help.o duration.o db_callbacks.o db.o main.o`**.
- Den definerer en regel, **`.c.o`**, der oversætter enhver .c-fil til en tilsvarende .o-fil, om nødvendigt.
- Den definerer en regel, **`${BIN}: ${OBJS}`**, der linker alle .o-filer sammen til den endelige executable. Det sker kun hvis en eller flere af .o-filerne er nyere end den eksisterende executable.

Syntaksen er svær, men hensigten er nobel: Det skal kunne lade sig gøre at compile sin kode med en enkelt kommando. Og det skal kunne lade sig gøre at programmere bygge-processen. Makefile er en slags programmering, som minder mest om shell scripting.

Mange nøjes med en Makefile, men den grønne knap i CLion gør faktisk mere:

Den bygger projektet med CMake ved hjælp af den fil der hedder CMakeLists.txt.

Bygge med CMake

CMake er en byggesystem-*generator*, hvilket betyder at den skaber build-filer (som Makefiles) der derefter bruges af byggeværktøjer som Make. Det er CMake som CLion bruger som udgangspunkt, så derfor lærer vi om det, og hvilke fordele det har:

- Cross-platform kompatibilitet: CMake tillader dig at skrive ét sæt konfigurationsfiler der kan generere projektfiler til forskellige platforme og compilere. Det betyder du lettere kan portere din kode mellem forskellige styresystemer (som Windows, macOS

og Linux) og udviklingsmiljøer (som Xcode, Visual Studio og Make) uden at omskrive dine build-scripts.
- Abstraktionslag: CMake fungerer som et meta-byggesystem. Den udfører ikke selv kompileringen, men genererer instruktioner til andre byggeværktøjer. Det betyder du kan skifte dit underliggende byggeværktøj (fx fra Make til Ninja) uden at ændre din CMake-konfiguration.
- Lettere håndtering af indviklede projekter: For større og mere komplekse projekter kan CMake forenkle håndteringen af dependencies, compiler flags og build-konfigurationer sammenlignet med at skrive indviklede Makefiles fra bunden.

Den store forskel er altså:

- Make er et byggeværktøj der udfører byggekommandoer
- CMake er et værktøj der genererer en Makefile

CMake afhænger af en fil der hedder CMakeLists.txt, som CLion opretter for os:

```
cmake_minimum_required(VERSION 3.28)
project(hello C)
set(CMAKE_C_STANDARD 23)
add_executable(hello main.c)
```

Bemærk **`cmake_minimum_required(VERSION 4)`**: den CMakeLists.txt som CLion opretter er meget ny, men Ubuntu har CMake version 3.28, så hvis du opretter projektet i CLion, og gerne vil køre kommandoer i Ubuntu, skal du rette det til **`VERSION 3.28`** i CMakeLists.txt

CMake har altså samme formål med at undgå at gentage sig, men kan sige tingene meget kortere og lettere at læse. Fordi CMake er et meta-byggeværktøj der laver en Makefile, er det nødvendigt at kalde den i to trin:

```
cmake .
make
./hello
```

Argumentet **`.`** fortæller CMake at CMakeLists.txt findes i nuværende mappe. For at holde byggefilerne adskilt fra kildekoden kan man oprette en **`build/`** mappe og køre **`cmake ..`** derfra - så peger **`..`** på parent-mappen hvor CMakeLists.txt ligger, mens alle genererede filer havner i **`build/`**:

```
mkdir build
cd build
cmake ..
make
./hello
```

Det er altså også et byggeværktøjs opgave at holde styr på alle de filer, der bliver produceret i byggeprocessen, og der har CMake en løsningsmodel som Make ikke har.

Include paths

Når nu vi har et byggeværktøj ved hånden, er det en god lejlighed til at spørge: Hvorfor skriver man **`#include <stdio.h>`**, men **`#include "hello.h"`**? Altså, hvorfor bruger vi **`<...>`** til system-header-filerne, men **`"..."`** til vores egne header-filer?

Svaret ligger i, at C compileren har noget der hedder *include path*, som er en liste med stier, hvor den kigger efter header-filer i. Det svarer meget til når man kører et program, og shell'en kigger i en liste af mapper efter programmet.

Hvis header-filen man inkluderer ligger i *include path*, kan man bruge **`<...>`**, ellers skal man bruge **`"..."`** til at henvise relativt til mappen man compiler fra. Hvis vi tilføjer **`.`** til include path, kan vi også benytte **`<...>`**.

Man tilføjer stier til include paths på forskellig måde:

- Med **`gcc`** skriver man **`gcc -I. -o hello main.c`**
- Med Makefile skriver man:

```
hello: hello.c
  gcc -I. -o hello hello.c
```

- Med CMakeLists.txt skriver man:

```
cmake_minimum_required(VERSION 3.28)
project(hello C)
set(CMAKE_C_STANDARD 23)
include_directories(.)
add_executable(hello main.c)
```

Nu virker **`#include <hello.h>`** også. Du må selv bestemme, hvilken syntaks du bruger.

Hvad er en objekt-fil (.o)?

En objektfil er en såkaldt *compilation unit*, altså én enhed af oversat kode.

En *compilation unit* er typisk lavet ud fra én .c-fil med alle dens inkluderede .h-filer. Man compiler separate .c-filer enkeltvis for at spare tid, fordi man kun behøver at recompile objektfiler, der er blevet ændret.

Hvordan ved compileren hvilken adresse en funktion bor på?

Du husker måske at funktioner også har adresser ligesom variable:

```
int main() {
    printf("main() har adressen %lu\n", &main);
    return 0;
}
```

```
$ gcc -o main_address main_address.c
main_address.c:4:41: warning: format specifies type 'unsigned long' but
the argument has type 'int (*)()' [-Wformat]
    printf("main() har adressen %lu\n", &main);
                                ~~~     ^~~~~
1 warning generated.$ ./main_address
main() har adressen 4310974268
```

Objektfiler indeholder kompileret maskinkode, men hvis en funktion i én C-fil kalder en funktion i en anden C-fil, hvor ved den så hvilken adresse den skal hoppe til?

Svaret er, at objektfiler stadig bruger *symbolske adresser*, dvs. navne på funktioner er ikke erstattet med deres konkrete adresser for alle objektfilerne bliver linket sammen, så de kan bo i det samme adresserum.

Symbolske adresser betyder at funktioner omtales ved navn (fx **`printf`** eller **`hello`**), mens numeriske henvisninger betyder at man kender den præcise hukommelsesadresse hvor funktionen ligger.

Linkeren konverterer symbolske henvisninger til numeriske adresser, så processoren kan springe direkte til den rigtige placering i hukommelsen.

Flaget **`-c`** compiler hello.c uden at *linke* den:

```
gcc -c hello.c        # opretter hello.o
ls -la hello.*        # viser både hello.c og hello.o
```

Linking

Vi har nu nok byggeklodser til at konstruere et eksempel, hvor det er tydeligt hvad det vil sige at linke en fil, når vi bygger den. Vi bygger videre på det simple Hello World-program, men hvor **`hello()`** er defineret i sin egen fil .h / .c, som bliver kaldt fra **`main()`**.

hello.h:

```
#ifndef _HELLO_H
#define _HELLO_H

void hello(void);

#endif // _HELLO_H
```

hello.c:

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

void hello(void) {
    printf("Hello, World!\n");
}
```

main.c:

```
#include "hello.h"

int main() {
    hello();
    return 0;
}
```

Include guards

Inden vi kommer for godt i gang, hvad er formålet med linjerne her?

```
#ifndef _HELLO_H
#define _HELLO_H

// ...

#endif // _HELLO_H
```

Hver for sig kaldes de bare for makroer. **`#ifndef`** tester ved kompileringstid om noget er sandt, og inkluderer kode eller ej i det endelige output der bliver compilet. **`#define`** definerer noget til at være sandt (eller at have en specifik værdi, men ovenfor er det bare "den er defineret").

I kombination kaldes de for "include guards" og er med til at sikre, at indholdet i header-filen ikke bliver inkluderet to gange. Hvad skulle risikoen for det være? Jo, folk skriver vel næppe

```
#include "hello.h"
#include "hello.h"
```

så det kan man godt bare bede folk om at lade være med.

Men måske inkluderer nogen både **`hello.h`** og **`stdio.h`** samtidigt med at **`hello.h`** *også* inkluderer **`stdio.h`**. Og så bliver **`stdio.h`** alligevel inkluderet to gange (én gang direkte og én gang indirekte). Det kan medføre fejl om at "struct'en er defineret allerede!"

Med include guards bliver enhver header-fil kun inkluderet én gang, og anden gang bliver dens indhold sprunget over. Navnet **`_HELLO_H`** er strategisk valgt til at undgå overlap med eksisterende konstanter: Den starter med _ og indeholder filens navn, så hvis nogen finder på at kalde deres konstant dét, er det deres skyld, hvis der er urelateret til filen her!

CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.28)
project(hello_project C)
add_executable(hello_project main.c hello.c hello.h)
```

Nu kan du kompilere og linke på flere måder:

```
# Kompiler hver fil til objekt-fil, og link dem derefter sammen
gcc -c main.c hello.c
gcc -o hello_project main.o hello.o

# Eller gør det hele i én kommando
gcc -o hello_project main.c hello.c

# Eller brug CMake
cmake .
make
./hello_project
```

Linking-trinnet forbinder **`main.o`** og **`hello.o`** til ét executable program, hvor alle symbolske henvisninger bliver forløst til faktiske adresser.

Dynamisk og statisk linking

Dynamisk linking

Lad os analysere hvordan en executable er blevet linket vha. **`ldd`**-programmet:

```
$ ldd hello_project
linux-vdso.so.1 (0x0000ffff8ee0c000)
libc.so.6 => /lib/aarch64-linux-gnu/libc.so.6 (0x0000ffff8ebd0000)
/lib/ld-linux-aarch64.so.1 (0x0000ffff8edc0000)
```

Her står altså, at selvom vi har compilet alle vores objektfiler sammen til én executable, så ligger noget af koden som kaldes altså slet ikke inde i selve executable'en. Den ligger i de her .so-filer, som står for **shared objects**. Et shared object er det man på Windows'sk kalder for **dynamically linked library**, eller DLL'er.

Tanken bag er, at alle ens programmer på computeren kommer til at dele, som minimum, standardbibliotekerne for de sprog de er skrevet i. Altså **`libc.so.6`** i vores tilfælde. Funktioner som **`printf`** ligger fx dér. Den fylder i vores tilfælde hele 1,7MB! Ja, det var godt nok meget plads, man kan spare, ved at flytte kode ud af sin executable.

Statisk linking

Man kan compile en statisk linket executable med parameteren **`gcc -static`**.

Her opdateres en Makefile, så **`make`** automatisk bygger en statisk linket executable:

```
hello: hello.c
  gcc -c -o hello.o hello.c
  gcc -c -o main.o main.c
  gcc -static -o hello main.o hello.o

clean:
  rm -f hello *.o
```

Det er altså kun når man producerer selve executable'en, at **`-static`** er nødvendig.

Prøver man at spørge hvilke dynamiske biblioteker denne funktion har:

```
$ make
gcc -c -o hello.o hello.c
gcc -c -o main.o main.c
gcc -static -o hello main.o hello.o

$ ldd hello
not a dynamic executable
```

Dynamisk linking vs. statisk linking (fordele og ulemper)

Statisk linking:

- Fordele: Programmet har ingen ingen filafhængigheder, og starter hurtigere
- Ulemper: Programmet fylder mere på disken, duplikerede biblioteker på systemet

Dynamisk linking:

- Fordele: Mindre executable, delte biblioteker sparer plads, biblioteker kan opdateres
- Ulemper: Afhængigheder kan mangle, ryge ud af sync, eller blive utilgængelige, lidt langsommere opstart

Statisk linking er foretrukket i cloud deployment

Fordi diskplads generelt ikke er et problem, er statisk linkede executables ofte at foretrække, fordi de kan kopieres ind i en Docker-container uden at man også skal have alle dens .so'er med.

Det betyder at sprog som er i stand til at producere statisk linkede executables med en kort opstartstid har en fordel i cloud-miljøer. Det er især Go og Rust der løber med guld-medaljen her.

Det er den absolut mest relevante betragtning ang. statisk vs. dynamisk linking i dag.

Dekompilering til assembly

`objdump` kan dekonstruere objekt-filer og vise assemblerkoden:

```
objdump -d hello.o    # viser disassembly af hello.o
```

Du kan også få gcc til at outputte assemblerkoden direkte:

```
gcc -S hello.c        # opretter hello.S med assemblerkoden
```

ASM 1: Hvordan virker funktionskald i assembly?

Lad os kalde **`objdump -d hello.o`** for følgende kode og se hvordan den virker:

```
void hello(void) {
    printf("Hello, World!\n");
}
```

Bemærk, at assemblerkoden vil være forskellig hvis du har en anden maskin-arkitektur. Den her kode blev decompilet på en MacBook, så den benytter arm64 assembler. Hvis du decompiler på en Intel/AMD processor, vil assemblerkoden benytte x86-64 assembler og se anderledes ud.

Da vi ikke har lært noget som helst om assembler, og da vi ikke har dedikeret os til nogen bestemt dialekt af assembler, kommer her en kommenteret udgave af assemblerkoden, hvor jeg har tilføjet kommentarerne ved hjælp af AI:

```
.text

# Funktion: hello
# Adresseområde: 0x0 - 0x1f
0000000000000000 <hello>:
      # Gem frame pointer (x29) og link register (x30) på stakken
      # Sænk stak-pointer med 16 bytes
       0: a9bf7bfd     stp  x29, x30, [sp, #-16]!

      # Sæt frame pointer (x29) til nuværende stak-pointer
      # Det etablerer bunden af den nuværende stak-frame
       4: 910003fd     mov  x29, sp

      # Indlæs hukommelsen hvor "Hello, World!" ligger i
      # ADRP indlæser en "page" af hukommelse (4KB)
       8: 90000000     adrp x0, 0x0 <hello+0x8>

      # Læg 0 til adressen hvor strings ligger i for at finde den
      # rigtige string; det er en lidt mærkelig ting at gøre, men
      # husk at koden her er auto-genereret af en compiler, som ikke
      # tænker over, at der kun findes én "Hello ..." string i hele
      # programmet.
       c: 91000000     add  x0, x0, #0

      # Branch til printf-funktionen med link tilbage
      # Kalder printf med streng-adressen i x0
      # Retur-adresse gemt i x30 (link register)
      10: 94000000     bl 0x10 <hello+0x10>

      # No operation - den her instruktion gør ingenting!
      14: d503201f     nop

      # Genindlæs frame pointer og link register fra stakken
      # Hæv stak-pointer med 16 bytes igen
      # Gendanner den tidligere stak-frame
      18: a8c17bfd     ldp  x29, x30, [sp], #16

      # Returner til funktionen der kalder hello()
      # Finder adressen til funktionen i x30 (link register)
      1c: d65f03c0     ret
```

Grundlæggende assembly-begreber

Numre i assembly-koden

I assembly-koden ser du to slags tal foran hver instruktion:

- **Offset/adresse** (venstre kolonne): `0:`, `4:`, `8:`, osv. Dette er byte-offset fra funktionens start. Da ARM64 instruktioner er 4 bytes hver, øges tallene med 4 for hver instruktion.
- **Maskinkode** (midterste kolonne): `a9bf7bfd`, `910003fd`, osv. Dette er den faktiske binære instruktion kodet som hexadecimale tal - det er hvad processoren rent faktisk udfører. Så når vi læser `add x0, x0, #0`, læser computeren bare `91000000`, hvor `91` er maskinkoden for `add`.

Registre

Registre er processorens interne hukommelseslommer - utroligt hurtige men begrænsede i antal:

- `x0-x30`: Generelle 64-bit registre i ARM64. `x0` bruges ofte til første parameter og retur-værdi
- `x29` (Frame Pointer): Peger på bunden af den nuværende stak-frame
- `x30` (Link Register): Gemmer adressen hvor funktionen skal returnere til
- `sp` (Stack Pointer): Peger på toppen af stakken (næste ledige plads)
- `pc` (Program Counter): Peger på den instruktion som bliver afviklet lige nu

Instruktioner

ARM64 instruktioner er de kommandoer processoren kan udføre:

- `stp` (Store Pair): Gemmer to registre på stakken på én gang
- `mov` (Move): Kopierer data fra ét register til et andet
- `adrp` (Address Page): Indlæser en *page* (4KB blok) af hukommelsen
- `add` (Add): Lægger to værdier sammen
- `bl` (Branch with Link): Kalder en funktion og gemmer retur-adresse
- `nop` (No Operation): Gør ingenting - bruges til alignment eller debugging
- `ldp` (Load Pair): Indlæser to registre fra stakken på én gang
- `ret` (Return): Returnerer til den funktion der kaldte denne

Stakken og stack frames

Stakken er et område i hukommelsen der bruges til:

- Lokale variable
- Funktionsparametre
- Retur-adresser
- Gemme registre midlertidigt

Stakken vokser "nedad" - nye data placeres på lavere hukommelsesadresser.

En **stak-frame** er det område på stakken der tilhører én funktionskald. Når en funktion kaldes:

1. Oprettes en ny frame
2. Lokale variable gemmes her
3. Ved funktions-retur fjernes framen

Frame pointer (**`x29`**) peger altid på bunden af den aktuelle frame, hvilket gør det nemt at finde lokale variable og parametre.

Link register (**`x30`**) gemmer adressen hvor funktionen skal hoppe tilbage til når den er færdig. Det kunne være instruktionen efter selve funktionskaldet, så koden fortsætter med at afvikle hvor den slap.

`x0`, **`x29`** og **`x30`** er "general purpose"-registre, hvilket betyder at man kan bruge dem til hvad som helst. Men på grund af **konventioner**, vælger vi altså at dedikere **`x29`** og **`x30`** til meget bestemte ting, som vi altid har brug for, og så compileren aldrig er i tvivl om hvor den skal finde de informationer.

Stakken som buffer for registre

Hvorfor flyttes **`x29`** og **`x30`** til stakken som det allerførste, og tilbage igen som det sidste?

Grunden er, at når en funktion kalder en anden funktion, bliver registrene potentielt **overskrevet**. I vores eksempel kalder **`hello()`** funktionen **`printf()`**. Når **`printf()`** kører, vil den måske bruge **`x29`** og **`x30`** til sine egne formål.

Hvis vi ikke gemte de oprindelige værdier af **`x29`** og **`x30`** før funktionskaldet, ville vi miste informationen om:

- Hvor **`hello()`**s egen stack frame starter (**`x29`**)
- Hvor **`hello()`** skal returnere til når den er færdig (**`x30`**)

Derfor følger alle funktioner denne protocol:

1. Funktionens start (prologue)

```
stp x29, x30, [sp, #-16]!   # Gem nuværende x29 og x30 på stakken
mov x29, sp                  # Etablér ny frame pointer
```

2. Funktionens krop

```
# Her sker funktionens arbejde
# Registrene x29 og x30 kan nu bruges frit
bl printf                    # Kald andre funktioner uden bekymring
```

3. Funktionens afslutning (epilogue)

```
ldp x29, x30, [sp], #16     # Gendan de originale x29 og x30
ret                         # Returner til den funktion der kaldte os
```

Funktionskald fra to perspektiver

I vores **`hello()`**-eksempel ser vi funktionskald fra begge perspektiver:

Som den kaldte funktion

`hello()` bliver kaldt af **`main()`**. Derfor skal **`hello()`**:

1. Gemme **`main()`**s **`x29`** og **`x30`** værdier på stakken
2. Etablere sin egen stack frame
3. Før retur: Gendanne **`main()`**s **`x29`** og **`x30`** værdier
4. Returnere til **`main()`** via **`ret`** (som bruger den gendannede **`x30`**)

Som den kaldende funktion

`hello()` kalder **`printf()`**. **`bl printf`** instruktionen:

1. Gemmer retur-adressen (næste instruktion efter **`bl`**) i **`x30`**
2. Springer til **`printf()`**s start-adresse
3. **`printf()`** vil følge samme protocol: gemme registre, arbejde, gendanne, returnere

Når **`printf()`** kalder **`ret`**, bruger den den adresse som **`bl`**-instruktionen gemte i **`x30`**, og hopper tilbage til instruktionen efter **`bl`** i **`hello()`**.

Minimal stack frame

I dette eksempel har **`hello()`** ingen lokale variable. Dens stack frame indeholder derfor kun:

- Den gemte **`x29`** værdi (frame pointer fra **`main()`**)
- Den gemte **`x30`** værdi (retur-adresse til **`main()`**)

Det er en 16-byte stack frame (2 registre × 8 bytes per register). Hvis funktionen havde lokale variable, ville de også blive allokeret på stakken, og stack frame'en ville være større.

Threads og parallelprogrammering

Afsnittet her dækker:

- threads og parallelprogrammering,
- deling af data mellem threads,
- synkronisering med mutex'er (mutual exclusion locks)

1. Introduktion til threads

Et thread er en separat udførselsstrøm inden for samme program.

Med threads kan vi få vores program til at køre flere ting samtidig.

Forskel på processer og threads

Processer har deres eget separate hukommelsesområde:

```
// To separate processer - kan ikke dele variable direkte
// Proces 1: program1.exe
int counter = 0;

// Proces 2: program2.exe
int counter = 0;  // Forskellig hukommelse, forskellig variabel
```

Threads deler hukommelsen inden for samme proces:

```
// To threads i samme program - deler globale variable
int counter = 0;  // Begge threads ser samme variabel

// Thread 1 kan ændre counter
// Thread 2 ser ændringen med det samme
```

Hvornår bruger man threads?

Threads er nyttige til:

- **I/O operationer**: Læse filer mens UI stadig reagerer
- **Parallel processing**: Udnytte flere CPU-kerner
- **Server programming**: Håndtere flere klienter samtidig
- **Producer-consumer**: En thread producerer data, en anden forbruger det

```
// Eksempel: Download fil uden at fryse UI
// Thread 1: Kører UI og reagerer på klik
// Thread 2: Downloader stor fil i baggrunden
```

Void-pointere (`void *`)

En void-pointer (**`void *`**) er en pointer der ikke har nogen specifik type. På overfladen kan det virke mærkeligt - hvorfor lave en pointer til "ingenting"?

Hvad er void?

`void` betyder "tom" eller "ingen værdi", og betyder at funktionen returnerer uden noget:

```
void print_hello() {
    printf("Hello!\n");
}
```

Men **`void *`** er anderledes - det er en pointer til hukommelse uden at specificere hvilken type data der er gemt der.

Hvorfor bruge void-pointere?

Void-pointere bruges når vi vil skrive generisk kode der kan arbejde med forskellige datatyper. Det er egentlig lidt et "hack", fordi C mangler det man kalder Generics i Java:

Altså evnen til at lave funktioner som fx: **`bool exists_in_array<T>(T objs[], T obj)`** – man kan ikke i C lave en funktion hvor typerne først er kendt senere.

I stedet *cast'er* man pointer-typen til **`void *`** og *cast'er* den tilbage når man skal bruge den, inden man dereferererer den. Her er nogle eksempler:

```
// En void-pointer der peger på en integer:
int number = 42;
void *ptr = &number;

// En void-pointer der peger på en string:
char *text = "Hello";
void *ptr2 = text;

// En void-pointer der peger på en struct:
struct person { char *name, int age };
struct person p = { "Alice", 25 };
void *ptr3 = &p;
```

Type casting med void-pointere

For at bruge data fra en void-pointer skal vi caste den til den rigtige type:

```
int number = 100;
void *generic_ptr = &number;

// For at læse værdien skal vi caste tilbage til int*
int *int_ptr = (int*) generic_ptr;
printf("Værdi: %d\n", *int_ptr);  // Output: Værdi: 100
```

Hvad vil det sige at "caste"?

Casting betyder at fortælle compileren "behandl denne værdi som en anden type". Det er som at give compileren en "oversættelse" af hvordan data skal forstås.

Implicit casting (automatisk):

```
int number = 42;
void *ptr = &number;  // Automatisk casting fra int* til void*
```

Når vi tager adressen på **`number`** med **`&number`**, får vi en **`int*`** (pointer til integer). Men når vi gemmer den i en **`void*`** variabel, konverterer compileren automatisk fra **`int*`** til **`void*`**. Dette er sikkert fordi enhver pointer-type kan konverteres til **`void*`**.

Explicit casting (manuelt):

```
void *generic_ptr = &number;
int *int_ptr = (int*) generic_ptr;  // Manuelt casting fra void* til int*
```

Her fortæller vi eksplicit compileren at behandle **`generic_ptr`** (som er **`void*`**) som en **`int*`**. Dette er nødvendigt fordi compileren ikke kan gætte hvilken type vi vil have tilbage fra en **`void*`**.

Hvad sker der teknisk når man caster?

Casting ændrer ikke data i hukommelsen – kun hvordan compileren fortolker data:

```
int number = 1000;
void *ptr = &number;

// Samme hukommelsesadresse, forskellige fortolkninger:
int *as_int = (int*) ptr;
char *as_char = (char*) ptr;

printf("Som int: %d\n", *as_int);     // 1000
printf("Som char: %d\n", *as_char);   // 232 (første byte af 1000)
```

Det minder lidt om når man henviser til forskellige felter i en **`union`**, så bliver den samme hukommelse også tolket forskelligt. Men her er det bare en enkelt pointer, der bliver tolket som en pointer til forskellige ting. Det er uproblematisk fordi **alle pointere fylder det samme antal bytes!**

Farlig casting:

```
float number = 3.14;
void *ptr = &number;
int *wrong_type = (int*) ptr;  // FARLIGT!
printf("%d\n", *wrong_type);   // Udefineret resultat
```

Casting er dit ansvar - compileren stoler på at du ved hvad du laver!

Hvorfor er det relevant for threads?

Thread-funktioner i pthreads bruger void-pointere fordi:

- En thread-funktion kun kan tage ét argument
- Det argument skal kunne være hvilken som helst datatype
- Thread-funktionen skal kunne returnere hvilken som helst datatype
- Thread-biblioteket er skrevet uden kendskab til hvilke datastrukturer man gerne vil bruge i forskellige tråde i forskellige applikationer, og da C ikke har *Generics* ligesom Java, gør man koden generisk ved hjælp af void-pointere.

```c
// Thread funktion signatur: void* function(void* arg)
void* my_thread(void* arg) {
    // arg kan være hvad som helst - vi skal bare vide hvad det er
    char *message = (char*) arg;  // Cast til den type vi forventer
    printf("Thread siger: %s\n", message);
    return NULL;  // Kan returnere hvad som helst som void*
}

int main() {
    char *msg = "Hello from thread!";
    // msg bliver implicit castet til void* her:
    pthread_create(&thread, NULL, my_thread, msg);
}
```

Praktisk eksempel

```c
#include <stdio.h>

// Generisk print funktion
void print_value(void *value, char type) {
    if (type == 'i') {
        int *int_val = (int*) value;
        printf("Integer: %d\n", *int_val);
    } else if (type == 'f') {
        float *float_val = (float*) value;
        printf("Float: %.2f\n", *float_val);
    } else if (type == 's') {
        char *str_val = (char*) value;
        printf("String: %s\n", str_val);
    }
}

int main() {
    int number = 42;
    float decimal = 3.14;
    char *text = "Hello World";

    // Samme funktion kan håndtere forskellige typer
```

```
    print_value(&number, 'i');
    print_value(&decimal, 'f');
    print_value(text, 's');

    return 0;
}
```

Void-pointere giver fleksibilitet, men kræver at vi holder styr på hvilken type data vi arbejder med. I threads bruges de til at lade os sende og returnere data af enhver type gennem en standardiseret interface.

Sammenhængen mellem header-filer, object-filer og linking

Når du kompilerer et C program sker der flere trin:

1. Preprocessor: Inkluderer header-filer

```
#include <pthread.h>  // Inkluderer function declarations
```

2. Compilation: Oversætter til object-filer

```
gcc -c program.c -o program.o  # Laver object file
```

3. Linking: Forbinder object-filer med biblioteker

```
gcc program.o -lpthread -o program  # Linker med pthread bibliotek
```

Hvad sker der i hvert trin?

Header-filer (`#include <pthread.h>`):

- Indeholder function declarations (prototypes)
- Fortæller compileren hvordan funktionerne skal kaldes
- Ingen egentlig kode - kun beskrivelser

```
// I pthread.h finder du noget som:
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void*), void *arg);
int pthread_join(pthread_t thread, void **retval);
// osv...
```

Object-filer (`.o` filer):

- Indeholder compiled kode fra din .c fil
- Har "huller" hvor external functions skal være
- Kan ikke køres selv - mangler bibliotekskode

Linking med `-lpthread`:

- Finder libpthread.so (Linux) eller libpthread.dylib (macOS)

- Kobler din kode sammen med pthread implementationen
- Fylder "hullerne" med rigtig funktionskode

Praktisk eksempel

```
// program.c
#include <pthread.h>  // 1. Header giver function declarations
#include <stdio.h>

void* my_function(void* arg) {
    printf("Hello from thread!\n");
    return NULL;
}

int main() {
    pthread_t thread;
    // Compileren ved hvordan pthread_create skal kaldes (fra header)
    // Men den faktiske implementation kommer fra biblioteket
    pthread_create(&thread, NULL, my_function, NULL);
    pthread_join(thread, NULL);
    return 0;
}
```

Kompileringsprocessen:

```
# 1. Compiling - laver object file med "huller"
gcc -c program.c -o program.o

# 2. Linking - fylder huller med pthread kode
gcc program.o -lpthread -o program

# Kan gøres i ét trin:
gcc program.c -lpthread -o program
```

Hvad sker der hvis du glemmer `-lpthread`?

```
undefined reference to `pthread_create'
undefined reference to `pthread_join'
```

Linkeren kan ikke finde implementationen af pthread funktionerne, selvom den ved hvordan de skal kaldes (fra headeren).

Headers og basic typer

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>  // sleep() bor her
```

```c
// Thread ID type
pthread_t thread_id;

// Mutex type (kommer senere)
pthread_mutex_t mutex;

// Condition variable type (kommer senere)
pthread_cond_t condition;
```

3. Oprettelse og håndtering af threads

Basic thread creation

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

// Thread funktion skal have denne signatur
void* thread_function(void* arg) {
    char* message = (char*) arg;

    for (int i = 0; i < 5; i++) {
        printf("%s: %d\n", message, i);
        sleep(1);  // Vent 1 sekund
    }

    return NULL;  // Thread skal returnere void*
}

int main() {
    pthread_t thread1, thread2;

    // Opret threads
    pthread_create(&thread1, NULL, thread_function, "Thread A");
    pthread_create(&thread2, NULL, thread_function, "Thread B");

    // Vent på at threads er færdige
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    printf("Alle threads er færdige\n");
    return 0;
}
```

pthread_create parametre

```c
int pthread_create(
    pthread_t *thread,              // Out: thread ID
    const pthread_attr_t *attr,     // Thread attributes (NULL = default)
    void *(*start_routine)(void*), // Function to run
    void *arg                       // Argument til funktionen
);
```

pthread_join og return values

```c
void* thread_function(void* arg) {
    int* result = malloc(sizeof(int));
    *result = 42;  // Beregn noget resultat
    return result;  // Returnér resultatet
}

int main() {
    pthread_t thread;
    void* return_value;

    pthread_create(&thread, NULL, thread_function, NULL);

    // Vent og få return value
    pthread_join(thread, &return_value);

    int* result = (int*) return_value;
    printf("Thread returnerede: %d\n", *result);

    free(result);  // Husk at frigøre hukommelsen!
    return 0;
}
```

4. Deling af data mellem threads

Når threads deler data, kan vi løbe ind i problemer.

Race conditions

```
#include <pthread.h>
#include <stdio.h>

int counter = 0;  // Global variabel delt mellem threads

void* increment_counter(void* arg) {
    for (int i = 0; i < 100000; i++) {
        counter++;  // PROBLEM: Ikke thread-safe!
    }
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    pthread_create(&thread1, NULL, increment_counter, NULL);
    pthread_create(&thread2, NULL, increment_counter, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    printf("Counter: %d (forventet: 200000)\n", counter);
    // Resultat er uforudsigeligt! Måske 150000, måske 190000...
    return 0;
}
```

Hvorfor sker race conditions?

`counter++` ser simpelt ud, men er faktisk tre operationer:

```
// counter++ bliver til:
int temp = counter;    // 1. Læs værdi
temp = temp + 1;       // 2. Tilføj 1
counter = temp;        // 3. Skriv tilbage
```

Hvis begge threads kører samtidig:

```
Thread A: Læser counter = 100
Thread B: Læser counter = 100  (samme værdi!)
Thread A: Skriver counter = 101
Thread B: Skriver counter = 101  (overskriver A's resultat!)
```

Vi har mistet en increment!

5. Mutex (Mutual Exclusion)

En mutex er en lås der sikrer at kun ét thread kan tilgå data ad gangen.

Basic mutex brug

```
#include <pthread.h>
#include <stdio.h>

int counter = 0;
pthread_mutex_t counter_mutex = PTHREAD_MUTEX_INITIALIZER;

void* safe_increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&counter_mutex);   // Lås
        counter++;                            // Critical section
        pthread_mutex_unlock(&counter_mutex); // Oplås
    }
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    pthread_create(&thread1, NULL, safe_increment, NULL);
    pthread_create(&thread2, NULL, safe_increment, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    printf("Counter: %d (forventet: 200000)\n", counter);
    // Nu får vi altid 200000!

    pthread_mutex_destroy(&counter_mutex);
    return 0;
}
```

Mutex initialisering

```
// Metode 1: Static initialisering
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

// Metode 2: Dynamic initialisering
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);  // NULL = default attributes

// Husk at rydde op:
pthread_mutex_destroy(&mutex);
```

Risiko for deadlock

Pas på at låse flere mutex'er i samme rækkefølge:

```
pthread_mutex_t mutex1, mutex2;

void* thread_a(void* arg) {
    pthread_mutex_lock(&mutex1);    // Lås 1
    pthread_mutex_lock(&mutex2);    // Lås 2

    // Gør noget...

    pthread_mutex_unlock(&mutex2);  // Oplås i omvendt rækkefølge
    pthread_mutex_unlock(&mutex1);
    return NULL;
}

void* thread_b(void* arg) {
    pthread_mutex_lock(&mutex1);    // Samme rækkefølge!
    pthread_mutex_lock(&mutex2);    // Undgår deadlock

    // Gør noget...

    pthread_mutex_unlock(&mutex2);
    pthread_mutex_unlock(&mutex1);
    return NULL;
}
```

Dårligt eksempel (deadlock):

```
// Thread A: Låser mutex1, venter på mutex2
// Thread B: Låser mutex2, venter på mutex1
// Begge venter for evigt!
```

Tutorial: Makroer, typedefs

Det her afsnit dækker:

- makroer og præprocessor-direktiver,
- typedefs til mere læsbar kode,
- praktiske anvendelser af begge,

1. Introduktion til makroer

En makro er en måde at definere tekst-substitutioner, der sker før kompileringen begynder.

Præprocessoren erstatter alle forekomster af makro med den definerede tekst.

```
#define MAX_PLAYERS 4
#define PI 3.14159

int main() {
    int players[MAX_PLAYERS];  // Bliver til: int players[4];
    double radius = 5.0;
    double area = PI * radius * radius;  // PI bliver erstattet med
3.14159
    return 0;
}
```

Makroer til at lave konstanter

Det simpleste er at definere konstanter:

```
#define BUFFER_SIZE 1024
#define DEBUG_MODE 1
#define VERSION_STRING "2.1.0"
```

Makroer har ingen type-information - de er bare tekst-erstatninger:

```
#define TRUE 1
#define FALSE 0

// Præprocessoren ser:
if (TRUE) {  // Bliver til: if (1) {
    printf("Dette er sandt\n");
}
```

Makro skrives normalt med STORE_BOGSTAVER for at skelne dem fra variable og funktioner:

```
#define MAX_SIZE 100        // God stil
#define max_size 100        // Kan forveksles med variabel
```

2. Makroer der minder om funktioner

Funktionsmakroer kan tage parametre:

```
#define SQUARE(x) ((x) * (x))
#define MAX(a, b) ((a) > (b) ? (a) : (b))

int main() {
    int result = SQUARE(5);   // Bliver til: ((5) * (5))
    int larger = MAX(10, 20); // Bliver til: ((10) > (20) ? (10) : (20))
    printf("5² = %d\n", result);
    printf("Max af 10 og 20: %d\n", larger);
    return 0;
}
```

Hvorfor alle de parenteser?

Uden parenteser kan man få overraskende resultater:

```
// FORKERT implementation:
#define SQUARE_BAD(x) x * x

int main() {
    int result = SQUARE_BAD(3 + 2);
    // Bliver til: 3 + 2 * 3 + 2 = 3 + 6 + 2 = 11 (ikke 25!)

    // KORREKT implementation:
    // SQUARE(3 + 2) bliver til: ((3 + 2) * (3 + 2)) = 25
    return 0;
}
```

Parenteser sikrer korrekt operator-præcedens.

Flerlinjet makroer

For større makroer bruges backslash til at fortsætte på næste linje:

```
#define DEBUG_PRINT(var) \
    do { \
        printf("DEBUG: %s = %d (line %d)\n", #var, var, __LINE__); \
    } while(0)

int main() {
    int count = 42;
    DEBUG_PRINT(count);  // Udskriver: DEBUG: count = 42 (line 15)
    return 0;
}
```

`#var` er en stringification operator der konverterer parameter-navnet til en string.

`__LINE__` er en indbygget makro der indeholder det aktuelle linjenummer.

Hvorfor `do { ... } while(0)` tricket?

Læg mærke til at makroen er pakket ind i **`do { ... } while(0)`**. Det ser mærkeligt ud, men det er et smart trick der løser flere problemer:

Problem 1: Multiple statements

```
// FORKERT - hvad sker der her?
#define DEBUG_PRINT_BAD(var) \
    printf("DEBUG: %s = %d\n", #var, var); \
    printf("Line: %d\n", __LINE__)

if (some_condition)
    DEBUG_PRINT_BAD(count);  // Kun første printf bliver kaldt!
else
    printf("Other case\n");  // Dette giver compile error!
```

Problem 2: Semicolon problemer

```
// Med do-while(0) virker det korrekt:
#define DEBUG_PRINT(var) \
    do { \
        printf("DEBUG: %s = %d\n", #var, var); \
        printf("Line: %d\n", __LINE__); \
    } while(0)

if (some_condition)
    DEBUG_PRINT(count);  // Virker perfekt som ét statement
else
    printf("Other case\n");
```

`do { ... } while(0)` sikrer at makroen opfører sig som ét enkelt statement, der kræver et semicolon efter sig. Dette er standard praksis for multi-statement makroer.

3. Fejl der kan opstå ved at bruge funktionsmakroer

Problem 1: Multiple evaluation og side effects

Parametre kan blive evalueret flere gange:

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))

int counter = 0;
int next_value() {
    return ++counter;
}
```

```c
int main() {
    int result = MAX(next_value(), 5);
    // Bliver til: ((next_value()) > (5) ? (next_value()) : (5))
    // next_value() bliver kaldt FLERE gange!
    printf("Result: %d, Counter: %d\n", result, counter);
    return 0;
}
```

Det her kan give uventet opførsel fordi **`next_value()`** bliver kaldt to gange.

Måden man undgår det er at makroer ikke må have bivirkninger (*side effects*).

Selvom man undgår bivirkninger, kan multiple evalueringer også påvirke performance.

Problem 2: Multiple evaluation og side effects

```c
#define DOUBLE(x) ((x) + (x))

int main() {
    int arr[] = {1, 2, 3, 4};
    int i = 0;

    int result = DOUBLE(arr[i++]);
    // Bliver til: ((arr[i++]) + (arr[i++]))
    // `i` bliver incrementeret to gange!

    printf("Result: %d, i: %d\n", result, i);  // Uventet resultat
    return 0;
}
```

Løsningen her er at bruge funktioner når det giver mening.

4. Introduktion til typedefs

En typedef skaber et alias (nyt navn) for en eksisterende type.

Det gør koden mere læsbar og nemmere at vedligeholde.

```
// I stedet for at skrive 'unsigned long long' overalt:
typedef unsigned long long uint64_t;

uint64_t large_number = 123456789ULL;
uint64_t another_number = 987654321ULL;
```

Basic typedef syntax

```
#include <stdio.h>

// Skab aliaser for primitive typer
typedef int player_id_t;
typedef double currency_t;
typedef char* string_t;

int main() {
    player_id_t player = 42;
    currency_t price = 99.99;
    string_t name = "Alice";

    printf("Player %d: %s har %.2f kroner\n", player, name, price);
    return 0;
}
```

Konvention: typedef navne ender ofte med **`_t`** for at indikere at det er en type.

Ellers kan det være svært at se, nogle gange, om noget er en type eller en variabel.

Fordele ved typedefs

1. **Læsbarhed**: **`user_id_t`** er klarere end **`unsigned int`**
2. **DRY (Don't Repeat Yourself)**: Skift typen ét sted i stedet for overalt
3. **Platform-uafhængighed**: En type som **`size_t`** virker fx på alle platforme

5. Typedefs med structs

Typedefs gør struct-navne kortere at skrive. Når man bruger en typedef med en struct behøver man nemlig ikke skrive **`struct`** når man henviser til sin type.

Navngivne structs med typedef

Den traditionelle måde at lave en typedef med en struct er at give struct'en et navn:

```
// Definer struct med navn
struct player_stats {
    char name[50];
    int level;
    int health;
    int experience;
};

// Lav en typedef for struct'en
typedef struct player_stats player_stats_t;

int main() {
    // Nu kan vi bruge begge syntakser:
    struct player_stats hero1;  // Den gamle måde
    player_stats_t hero2;       // Den nye, kortere måde
    return 0;
}
```

Dette giver os fleksibilitet til at bruge både **`struct player_stats`** og **`player_stats_t`**.

Kombination: navngivet struct + typedef på én linje

Man kan også definere struct'en og typedef'en samtidig:

```
typedef struct player_stats {
    char name[50];
    int level;
    int health;
    int experience;
} player_stats_t;

int main() {
    player_stats_t hero;     // Brug typedef-navnet
    return 0;
}
```

Her har struct'en stadig et navn (**`player_stats`**), men vi bruger primært typedef-navnet (**`player_stats_t`**).

Anonyme structs med typedef

Ofte definerer man structs kun for at lave en typedef, og så behøver struct'en ikke et navn:

```
// Anonymous struct - ingen struct-navn
typedef struct {
    int x;
    int y;
} point_t;

typedef struct {
    point_t position;
    int width;
    int height;
} rectangle_t;

int main() {
    point_t origin = {0, 0};
    rectangle_t box = {
        .position = {10, 20},
        .width = 100,
        .height = 50
    };
    return 0;
}
```

Med anonymous structs kan du **kun** bruge typedef-navnet (**`point_t`**), ikke **`struct`** syntaksen.

Hvornår skal man bruge makroer?

Brug makroer til:

- Konstanter der skal være compile-time kendte
- Conditional compilation (**`#ifdef DEBUG`**)
- Simple tekst-substitutioner
- Platform-specifikke kode

```
// Gode eksempler:
#define ARRAY_SIZE 100
#define PI 3.14159
#ifdef WINDOWS
    #define PATH_SEPARATOR "\\"
#else
    #define PATH_SEPARATOR "/"
#endif
```

Undgå makroer for:

- Kompleks logik (brug funktioner i stedet)

- Ting der kan have side effects
- Type-checking (brug inline funktioner)

```
// DÅRLIGT - brug en funktion i stedet:
#define COMPLEX_CALC(x, y, z) \
    ((x * y + z) > 0 ? sqrt(x * y + z) : 0)

// BEDRE:
inline double complex_calc(double x, double y, double z) {
    double temp = x * y + z;
    return temp > 0 ? sqrt(temp) : 0;
}
```

Hvornår skal man bruge typedefs?

Brug typedefs til:

- Komplekse type-definitioner
- Function pointers
- Platform-uafhængighed
- API konsistens

```
// Gode eksempler:
typedef uint32_t user_id_t;
typedef int (*compare_func_t)(const void*, const void*);
typedef struct { int x, y; } point_t;
```

Overvej at undgå typedefs for:

- Simple primitive typer hvor den originale type er tydelig
- Når det skjuler vigtig type-information
- Typer til engangsbrug

```
// Måske unødvendigt:
typedef int integer_t;  // "int" er allerede klart

// Bedre at beholde original type:
int count = 0;  // I stedet for integer_t count = 0;
```

Tutorial: Enums, unions, tagged unions

Den her tutorial dækker:

- enums,
- unions,
- tagged unions (type-sikre unions),
- praktiske anvendelser af dem.

1. Introduktion til enums

En enum er en måde at definere navngivne konstanter på.

Det gør koden mere læselig og med færre fejl end at bruge "magiske konstanter".

```
enum weekday_t {
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
};
```

Som standard får værdierne automatisk numre: **`MONDAY = 0`**, **`TUESDAY = 1`**, osv.

Vi kan bruge enums sådan her:

```
#include <stdio.h>

enum weekday_t {
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
};

int main(void) {
    enum weekday_t today = FRIDAY;

    if (today == FRIDAY) {
        printf("Det er fredag!\n");
    }

    return 0;
}
```

Eksplicitte værdier

Man kan også tildele specifikke værdier:

```
enum http_status_t {
    HTTP_OK = 200,
    HTTP_NOT_FOUND = 404,
    HTTP_SERVER_ERROR = 500
};
```

Dette er særligt nyttigt når værdierne skal matche ting som HTTP-statuskoder.

Enums vs. `#define`

I stedet for enums kunne man bruge **`#define`**:

```
#define MONDAY 0
#define TUESDAY 1
#define WEDNESDAY 2
```

Men enums giver type-sikkerhed ved at compileren kan tjekke typer:

```
// Med #define - problematisk!
#define RED 1
#define SMALL 1

int color = RED;
int size = SMALL;
if (color == size) { /* Virker, men giver ingen mening! */ }

// Med enum - type-sikkert!
enum color_t { RED };
enum size_t { SMALL };

enum color_t color = RED;
enum size_t size = SMALL;
if (color == size) { /* Compiler warning/error! */ }
```

2. Introduktion til unions

En union ligner en struct, men med en vigtig forskel: alle medlemmer deler den samme hukommelsesplads. Det betyder at du kun kan bruge ét medlem ad gangen. Eksempel:

```
union value_t {
    int as_int;     // ville fylde 4 bytes
    float as_float; // ville fylde 4 bytes
    char as_char;   // ville fylde 1 byte
};
```

En union tager kun så meget plads som det største medlem. I dette tilfælde: **`sizeof(union value_t) == sizeof(float) == 4 bytes`**.

Brug af unions

```
#include <stdio.h>

union value_t {
    int as_int;
    float as_float;
    char as_char;
};

int main(void) {
    union value_t value;

    // Gem en int
    value.as_int = 42;
    printf("Som int: %d\n", value.as_int);

    // Gem en float (overskriver int værdien!)
    value.as_float = 3.14f;
    printf("Som float: %.2f\n", value.as_float);

    // Nu er as_int ikke længere gyldig!
    printf("as_int nu: %d (garbage!)\n", value.as_int);

    return 0;
}
```

Hvornår er unions nyttige?

Unions er nyttige når du vil spare hukommelse og ved at du kun skal bruge ét af flere mulige datatyper ad gangen. For eksempel:

```
union number_t {
    int integer;
```

```
    double decimal;
};
```

3. Problemer med almindelige unions

Unions har en stor svaghed: der er ingen måde at vide hvilket medlem der er gyldigt!

```
#include <stdio.h>

union confused_value_t {
    int as_int;
    float as_float;
    char *as_string;
};

void print_value_wrong(union confused_value_t value) {
    // Hvilken type er det? Vi ved det ikke!
    printf("Som int: %d\n", value.as_int);
    printf("Som float: %.2f\n", value.as_float);
    printf("Som string: %s\n", value.as_string); // FARLIGT!
}

int main(void) {
    union confused_value_t value;
    value.as_int = 42;

    // Denne funktion ved ikke at værdien er en int!
    print_value_wrong(value);
    return 0;
}
```

Hvad går galt?

1. **Ingen type-information**: Unionen ved ikke hvilken type der er gyldig
2. **Udefineret opførsel**: At læse forkert medlem kan give garbage data
3. **Hukommelseskorruption**: At læse en pointer når data er et tal kan crashe programmet
4. **Debugging-mareridt**: Fejl er svære at finde fordi de ikke altid sker med det samme

Eksemplet viser hvorfor almindelige unions er farlige at bruge alene.

4. Tagged unions (diskriminerede unions)

Løsningen er at kombinere en enum med en union!

Man tagger unionen med en enum'en, så den fortæller os hvilken type der er gyldig.

```
enum value_type_t {
    VALUE_INT,
    VALUE_FLOAT,
    VALUE_STRING
};

struct tagged_value_t {
    enum value_type_t type;  // "Tag" der viser typen
    union {
        int as_int;
        float as_float;
        char *as_string;
    } data;
};
```

Nu kan vi sikkert tjekke typen før vi læser data:

```
void print_value_safe(struct tagged_value_t value) {
    switch (value.type) {
        case VALUE_INT:
            printf("Int: %d\n", value.data.as_int);
            break;
        case VALUE_FLOAT:
            printf("Float: %.2f\n", value.data.as_float);
            break;
        case VALUE_STRING:
            printf("String: %s\n", value.data.as_string);
            break;
        default:
            printf("Ukendt type!\n");
    }
}

int main(void) {
    struct tagged_value_t value;
    value.type = VALUE_INT;
    value.data.as_int = 42;
    print_value_safe(value);

    value.type = VALUE_STRING;
    value.data.as_string = "Hello, World!";
    print_value_safe(value);
    return 0;
}
```

5. Øvelse: Inventory-system til computerspil

Lad os bygge et system der kan holde styr på items i et inventory i et spil.

Her kan tagged unions bruges til at håndtere forskellige typer af items.

Trin 1: Definition af item-typer

Først definerer vi structs for hver type genstand:

```
#include <stdio.h>
#include <string.h>

struct weapon_t {
    int damage;
    int durability;
};

struct armor_t {
    int protection;
    int durability;
};

struct consumable_t {
    int effect_type;  // 0=heal, 1=mana
    int amount;
    int uses_left;
};

enum item_type_t {
    ITEM_WEAPON,
    ITEM_ARMOR,
    ITEM_CONSUMABLE,
};
```

Trin 2: Tagged union

Så definerer vi en tagged union som har en type, et navn og et union-felt.

```
struct inv_item_t {
    enum item_type_t type;
    char name[50];
    union {
        struct weapon_t weapon;
        struct armor_t armor;
        struct consumable_t consumable;
    } data;
};
```

Trin 3: Constructor-funktioner

Her er nogle type-sikre funktioner til at oprette items:

```
struct inv_item_t create_weapon(const char *name, int dmg, int dur) {
    struct inv_item_t item;
    item.type = ITEM_WEAPON;
    strcpy(item.name, name);
    item.data.weapon.damage = dmg;
    item.data.weapon.durability = dur;
    return item;
}

struct inv_item_t create_armor(const char *name, int armor, int dur) {
    struct inv_item_t item;
    item.type = ITEM_ARMOR;
    strcpy(item.name, name);
    item.data.armor.protection = armor;
    item.data.armor.durability = dur;
    return item;
}

struct inv_item_t create_consumable(
    const char *name,
    int effect_type,
    int amount,
    int uses
) {
    struct inv_item_t item;
    item.type = ITEM_CONSUMABLE;
    strcpy(item.name, name);
    item.data.consumable.effect_type = effect_type;
    item.data.consumable.amount = amount;
    item.data.consumable.uses_left = uses;
    return item;
}
```

Hjælpefunktionerne gør det nemmere og kortere at lave items.

`strcpy()` kunne godt erstattes med **`strncpy()`**, da **`item.name`** er 50 tegn lang.

Derudover, returnerer funktionerne structs og ikke struct pointers, hvilket gør dem lidt nemmere at arbejde med, da man derfor ikke skal tænke på at kalde **`free()`** for hver **`malloc()`**.

Trin 4: Sikre accessor funktioner

```
int get_weapon_stats(struct inv_item_t item, struct weapon_t *result) {
    if (item.type != ITEM_WEAPON) {
        return 0; // Fejl
    }
    *result = item.data.weapon;
    return 1; // Succes
}

int get_armor_stats(struct inv_item_t item, struct armor_t *result) {
    if (item.type != ITEM_ARMOR) {
        return 0; // Fejl
    }
    *result = item.data.armor;
    return 1; // Succes
}

void print_item_info(struct inv_item_t item) {
    printf("Item: %s\n", item.name);

    switch (item.type) {
        case ITEM_WEAPON:
            printf("  Type: Weapon\n");
            printf("  Damage: %d\n", item.data.weapon.damage);
            printf("  Durability: %d\n", item.data.weapon.durability);
            break;

        case ITEM_ARMOR:
            printf("  Type: Armor\n");
            printf("  Protection: %d\n", item.data.armor.protection);
            printf("  Durability: %d\n", item.data.armor.durability);
            break;

        case ITEM_CONSUMABLE:
            printf("  Type: Consumable\n");
            printf("  Effect: %s\n", item.data.consumable.effect_type ==
0 ? "Heal" : "Mana");
            printf("  Amount: %d\n", item.data.consumable.amount);
            printf("  Uses left: %d\n", item.data.consumable.uses_left);
            break;

        case ITEM_QUEST:
            printf("  Type: Quest Item\n");
            printf("  Quest ID: %d\n", item.data.quest.quest_id);
            printf("  Description: %s\n", item.data.quest.description);
            break;
    }
}
```

Trin 5: Praktisk brug

```c
int main(void) {
    // Opret inventory med forskellige items
    struct inv_item_t inventory[] = {
        create_weapon("Iron Sword", 25, 100),
        create_armor("Leather Vest", 15, 80),
        create_consumable("Health Potion", 0, 50, 3),
        create_quest_item("Ancient Key", 42, "Opens the mysterious door")
    };

    printf("=== Player Inventory ===\n");
    for (int i = 0; i < 4; i++) {
        print_item_info(inventory[i]);
        printf("\n");
    }

    // Eksempel på sikker adgang til weapon stats
    struct weapon_t weapon_stats;
    if (get_weapon_stats(inventory[0], &weapon_stats)) {
        printf("Weapon damage: %d\n", weapon_stats.damage);
    }

    return 0;
}
```