INF-0103 Kompilering og header-filer

Jakob Peder Pettersen, UiT – Norges arktiske universitet

Høst 2025

Dagens tema

- Tolkede og kompilerte programmeringsspråk

Dagens tema

- Tolkede og kompilerte programmeringsspråk
- Kompileringsprosessen

Dagens tema

- Tolkede og kompilerte programmeringsspråk
- Kompileringsprosessen
- Bruk av preprosessor

Dagens tema

- Tolkede og kompilerte programmeringsspråk
- Kompileringsprosessen
- Bruk av preprosessor
- Header-filer

Dagens tema

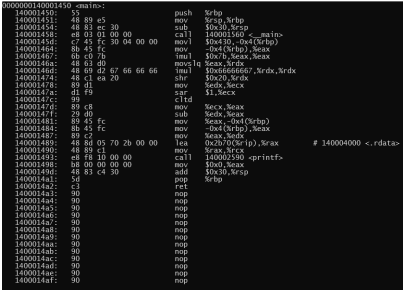
- Tolkede og kompilerte programmeringsspråk
- Kompileringsprosessen
- Bruk av preprosessor
- Header-filer
- Lage program av flere kildekodefiler

Hvorfor programmeringsspråk?

- Prosessoren forstår bare maskinkode

```c
#include <stdio.h>

int main() {
    int tall = 536 << ((2 | 5) & 4) % 3;
    tall = tall * 123 / 5;
    printf("%i\n", tall);
    return 0;
}
```



Hvorfor programmeringsspråk?

- Prosessoren forstår bare maskinkode
- Mennesker er dårlige til å skrive maskinkode

```c
#include <stdio.h>

int main() {
    int tall = 536 << ((2 | 5) & 4) % 3;
    tall = tall * 123 / 5;
    printf("%i\n", tall);
    return 0;
}
```

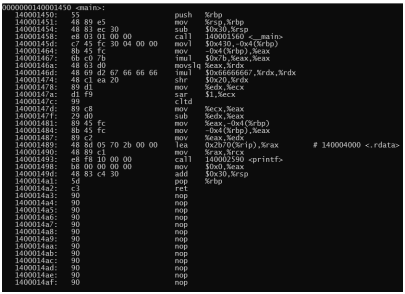
```
0000000140001450 <main>:
140001450:   55                      push   %rbp
140001451:   48 89 e5                mov    %rsp,%rbp
140001454:   48 83 ec 30             sub    $0x30,%rsp
140001458:   e8 01 01 00 00          call   140001560 <__main>
14000145d:   c7 45 fc 00 04 00 00    movl   $0x400,-0x4(%rbp)
140001464:   8b 45 fc                mov    -0x4(%rbp),%eax
140001467:   6b c0 7b                imul   $0x7b,%eax,%eax
14000146a:   48 63 d0                movslq %eax,%rdx
14000146d:   48 69 d2 67 66 66 66    imul   $0x66666667,%rdx,%rdx
140001474:   48 c1 ea 20             shr    $0x20,%rdx
140001478:   89 d1                   mov    %edx,%ecx
14000147a:   d1 f9                   sar    $1,%ecx
14000147c:   99                      cltd
14000147d:   89 c8                   mov    %ecx,%eax
14000147f:   29 d0                   sub    %edx,%eax
140001481:   89 45 fc                mov    %eax,-0x4(%rbp)
140001484:   8b 45 fc                mov    -0x4(%rbp),%eax
140001487:   89 c2                   mov    %eax,%edx
140001489:   48 8d 05 70 2b 00 00    lea    0x2b70(%rip),%rax        # 140004000 <.rdata>
140001490:   48 89 c1                mov    %rax,%rcx
140001493:   e8 f8 10 00 00          call   140002590 <printf>
140001498:   b8 00 00 00 00          mov    $0x0,%eax
14000149d:   48 83 c4 30             add    $0x30,%rsp
1400014a1:   5d                      pop    %rbp
1400014a2:   c3                      ret
1400014a3:   90                      nop
1400014a4:   90                      nop
1400014a5:   90                      nop
1400014a6:   90                      nop
1400014a7:   90                      nop
1400014a8:   90                      nop
1400014a9:   90                      nop
1400014aa:   90                      nop
1400014ab:   90                      nop
1400014ac:   90                      nop
1400014ad:   90                      nop
1400014ae:   90                      nop
1400014af:   90                      nop
```

Hvorfor programmeringsspråk?

- Prosessoren forstår bare maskinkode
- Mennesker er dårlige til å skrive maskinkode
- Løsning: Mennesket skriver programmet i et høynivåspråk som oversettes til maskinkode

```c
#include <stdio.h>

int main() {
    int tall = 536 << ((2 | 5) & 4) % 3;
    tall = tall * 123 / 5;
    printf("%i\n", tall);
    return 0;
}
```



Hvordan oversette?

To hovedløsninger:

- Tolker: Oversettelsen skjer når programmet kjører

Hvordan oversette?

To hovedløsninger:

- Tolker: Oversettelsen skjer når programmet kjører
- Kompilator: Programmet blir oversatt til ei fil i maskinkode som deretter kan kjøres

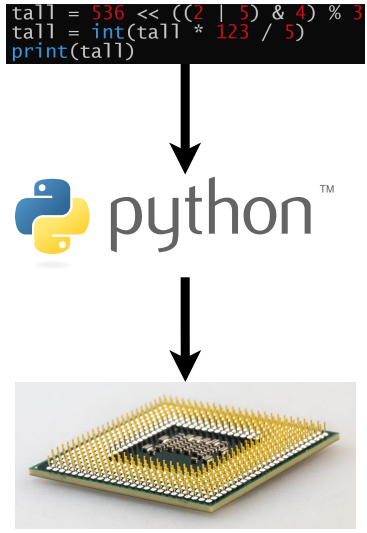


Figure 1: Virkemåte til tolker

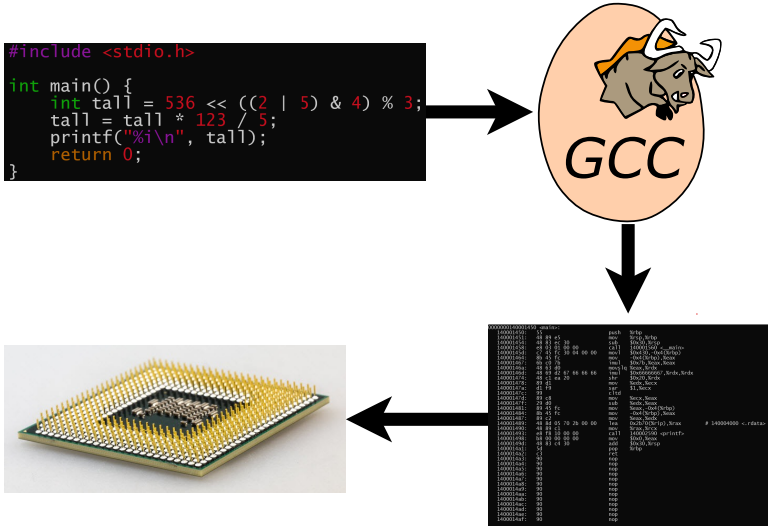


Figure 2: Virkemåte til kompilator

Hvorfor ikke bare bruke Python?

- Ytelse (tolkeren spiser av lasset)

Hvorfor ikke bare bruke Python?

- Ytelse (tolkeren spiser av lasset)
- Mer direkte kontrol over maskinvare

Hvorfor ikke bare bruke Python?

- Ytelse (tolkeren spiser av lasset)
- Mer direkte kontrol over maskinvare
- Mer direkte kommunikasjon med operativsystem og andre programmer

Hvorfor ikke bare bruke Python?

- Ytelse (tolkeren spiser av lasset)
- Mer direkte kontrol over maskinvare
- Mer direkte kommunikasjon med operativsystem og andre programmer
- Kompilatoren oppdager (noen) feil

Hvorfor ikke bare bruke Python?

- Ytelse (tolkeren spiser av lasset)
- Mer direkte kontrol over maskinvare
- Mer direkte kommunikasjon med operativsystem og andre programmer
- Kompilatoren oppdager (noen) feil
- Tolkeren er jo også et program

Kompileringsprosessen

Nøkkelkonsept: Oversettelsesenhet (*Translational unit*)

Tre hovedsteg:

- Preprosessering (separat for hver oversettelsesenhet)

Kompilatoren tar som standard alle stegene i et jafs, men vi kan dele dem opp

Kompileringsprosessen

Nøkkelkonsept: Oversettelsesenhet (*Translational unit*)

Tre hovedsteg:

- Preprosessering (separat for hver oversettelsesenhet)
- Kompilering (separat for hver oversettelsesenhet)

Kompilatoren tar som standard alle stegene i et jafs, men vi kan dele dem opp

Kompileringsprosessen

Nøkkelkonsept: Oversettelsesenhet (*Translational unit*)

Tre hovedsteg:

- Preprosessering (separat for hver oversettelsesenhet)
- Kompilering (separat for hver oversettelsesenhet)
- Lenking (oversettelsesenhetene kombineres)

Kompilatoren tar som standard alle stegene i et jafs, men vi kan dele dem opp

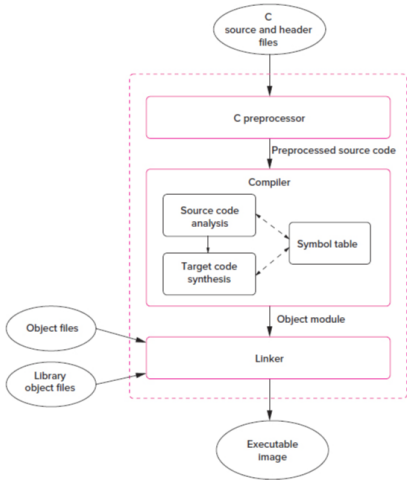


Figure 3: Kompileringsprosessen

Preprosessoren

- Kjører før kompilering

Preprosessoren

- Kjører før kompilering
- Instruksjoner begynner med #

Preprosessoren

- Kjører før kompilering
- Instruksjoner begynner med #
- Alle instruksjoner er på sin egen linje, ingen semikolon nødvendig

Preprosessoren

- Kjører før kompilering
- Instruksjoner begynner med #
- Alle instruksjoner er på sin egen linje, ingen semikolon nødvendig
- Er en ren tekstprosessor, ignorerer strukturen til programmet

Viktige proprosessorinstruksjoner (*directives*)

```
#include "header.h"
```

inkluderer (limer inn) headerfil (fra lokal mappe)

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

inkluderer systemheader (fra systemets katalogsti)

```
#define PI 3.14159265
```

definerer makro

```
#define SUM(x,y) x + y
```

definerer makro med parameter

Oppgave: Hva skriver dette programmet ut?

```c
#include <stdio.h>

#define SUM(x, y) x + y

int main() {
    int a, b, c;
    a = 3;
    b = 5;
    c = 2;
    int res = a * SUM(b, c);
    printf("%d", res);
    return 0;
}
```

Minimer bruk av preprosessor

- Deklarer konstanter som konstante variabler

```
static const double PI = 3.14159265;


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

Minimer bruk av preprosessor

- Deklarer konstanter som konstante variabler

```
static const double PI = 3.14159265;
```

- Foretrekk funksjoner framfor makroer

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

Minimer bruk av preprosessor

- Deklarer konstanter som konstante variabler

```
static const double PI = 3.14159265;
```

- Foretrekk funksjoner framfor makroer

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

- Bruk `static` for funksjoner og konstanter som **ikke** skal deles med andre oversettelsesenheter

Minimer bruk av preprosessor

- Deklarer konstanter som konstante variabler

```
static const double PI = 3.14159265;
```

- Foretrekk funksjoner framfor makroer

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

- Bruk `static` for funksjoner og konstanter som **ikke** skal deles med andre oversettelsesenheter
- C++ har ei langt rikere verktøykasse på dette området (ikke pensum)

Hva er ei headerfil?

- C-fil lagd for å inkluderes

Hva er ei headerfil?

- C-fil lagd for å inkluderes
- Inkludering av andre header-filer

Hva er ei headerfil?

- C-fil lagd for å inkluderes
- Inkludering av andre header-filer
- Funksjonsdeklarasjoner

Hva er ei headerfil?

- C-fil lagd for å inkluderes
- Inkludering av andre header-filer
- Funksjonsdeklarasjoner
- Definisjoner

Hva er ei headerfil?

- C-fil lagd for å inkluderes
- Inkludering av andre header-filer
- Funksjonsdeklarasjoner
- Definisjoner
- Konstanter

Header guard

- I større prosjekter: Lett at samme headerfil blir inkludert to ganger

```
#ifndef NAVN_H
#define NAVN_H
// Resten av header
#endif
```

Header guard

- I større prosjekter: Lett at samme headerfil blir inkludert to ganger
- Det går vanligvis dårlig

```
#ifndef NAVN_H
#define NAVN_H
// Resten av header
#endif
```

Header guard

- I større prosjekter: Lett at samme headerfil blir inkludert to ganger
- Det går vanligvis dårlig
- Løsning: `#include` guard som forhindrer at dette skjer

```
#ifndef NAVN_H
#define NAVN_H
// Resten av header
#endif
```

Header guard

- I større prosjekter: Lett at samme headerfil blir inkludert to ganger
- Det går vanligvis dårlig
- Løsning: `#include` guard som forhindrer at dette skjer

```
#ifndef NAVN_H
#define NAVN_H
// Resten av header
#endif
```

- Ikke-standard, men enklere løsning (støttes av de fleste moderne kompilatorer): `#pragma once`

Make-fil

- Automatiserer byggingen av prosjekter

Make-fil

- Automatiserer byggingen av prosjekter
- Egen syntaks

Prosjekt med flere filer

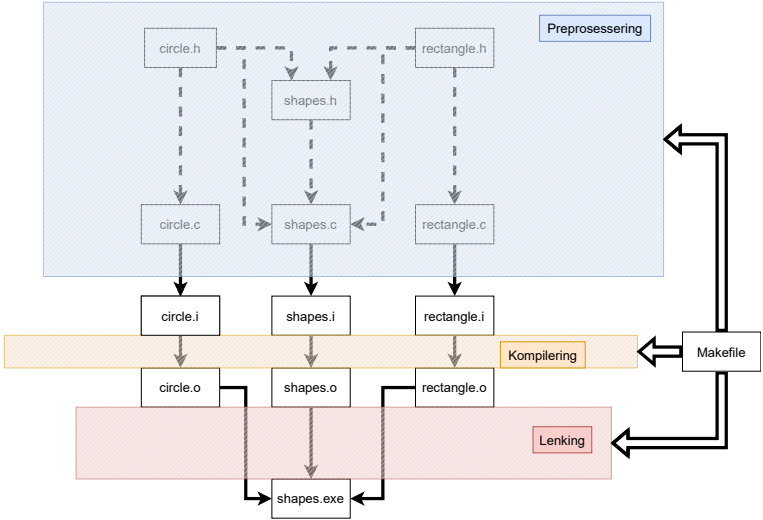


Figure 4: Struktur på eksempelet vårt