Dagens tema

- Cs preprosessor
- Separat kompilering av C-funksjoner
- C og minnet

Cs preprosessor

Før selve kompileringen går C-kompilatoren gjennom koden med en **preprosessor** (som er programmet cpp). Dette er en programmerbar **tekstbehandler** som gjør følgende:

- Henter inn filer

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

  Hvis filen er angitt med spisse klammer (som for eksempel <stdio.h>), hentes filen fra området /usr/include/. Ellers benyttes vanlig notasjon for filer.

- Leser makro[1]-definisjoner[2] og ekspanderer disse i teksten:

```
#define LINUX
#define N 100
#define MIN(x,y) ((x)<(y) ? (x) : (y))
```

Av tradisjon gis makroer navn med store bokstaver.

[1] En **makro** er en navngitt programtekst. Når navnet brukes, blir det **ekspandert**, dvs erstattet av definisjonen. Dette er ren tekstbehandling uten noen forbindelse med programmeringsspråkets regler.

[2] Benytter man makroer med parametre, bør disse settes i parenteser. Likeledes, hvis definisjonen er et uttrykk med flere symboler, bør det stå parenteser rundt hele uttrykket.

Betinget kompilering

Følgende direktiver finnes for betinget kompilering:

- **#if** Hvis uttrykket etterpå er noe annet enn 0, tas etterfølgende linjer tas med. Uttrykket kan ikke inneholde variable eller funksjoner.
- **#ifdef** Hvis symbolet er definert (med en #define), skal etterfølgende linjer tas med.
- **#ifndef** Motsatt av #ifdef.
- **#else** Skille mellom det som skal tas med og det som ikke skal tas med.
- **#endif** Slutt med betinget kompilering.

Eksempel:

```
#define LINUX

#ifdef LINUX
  short x;
#else
  long x;
#endif
```

Det er også mulig å styre betinget kompilering gjennom gcc-kommandoen:

```
> gcc -c -DLINUX ...
```

gir samme effekt som om det sto

```
#define LINUX
```

i program-koden.

På denne måten er det mulig å ha flere versjoner av koden (for eksempel for flere maskintyper) og så kontrollere dette utelukkende gjennom kompileringen.

Fare med betinget kompilering

Man kan risikere å ha kode som aldri har vært kompilert, og som kan inneholde de merkeligste feil.

Separat kompilering

I utgangspunktet er det ingen problem med separat-kompilering i C; hver fil utgjør en enhet som kan kompileres for seg selv, uavhengig av alle andre filer i programmet.

```
> gcc -c del.c
```

vil kompilere filen del.c og lage del.o som inneholder den kompilerte koden.

Eksempel

Anta at vi har to filer:

sum.c

```
int sum (int n)
{ /* Beregner 1+2+...+n */
  return n*(n+1)/2;
}
```

vissum.c

```
#include <stdio.h>

extern int sum (int n);

int main (void)
{
  int i;
  for (i = 1;  i <= 10;  ++i)
    printf("%2d:%4d\n", i, sum(i));
}
```

Kompilering

Disse kan kompileres hver for seg:

```
gcc -c sum.c
gcc -c vissum.c
```

Linking

De kompilerte filene kan siden **linkes** sammen:

```
gcc vissum.o sum.o -o vissum
```

Kjøring

Da får vi et ferdig program som kan kjøres:

```
./vissum
 1:   1
 2:   3
 3:   6
 4:  10
 5:  15
 6:  21
 7:  28
 8:  36
 9:  45
10:  55
```

Imidlertid er det en fare for at funksjonssignaturer, strukturer, makroer, typer og andre elementer ikke blir skrevet likt i hver fil. Dette løses ved hjelp av definisjonsfiler («header files»), hvis navn gjerne slutter med '.h'.

incl.h

```
#define N 100
```

prog.c

```
#include "incl.h"

int main(void)
{
  char *s[N];
    ⋮
}
```

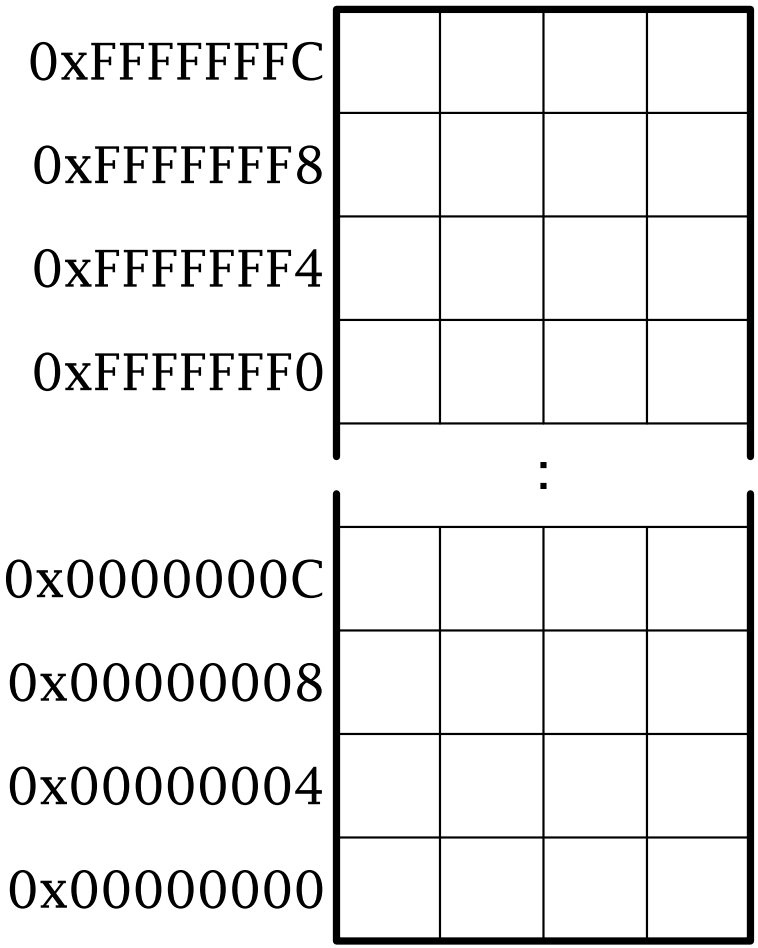
Definisjonsfiler inneholder gjerne følgende:

- Makrodefinisjoner (#define)
- Typedefinisjoner (typedef, union, struct)
- Eksterne variabelspesifikasjoner (extern)
- Funksjonssignaturer som

```
extern int f(int,char);
```

C og minnet

Minnet er en samling byte som har hver sin adresse:



Variable i C

Cs variable legges normalt pent etter hverandre (men ikke alltid i den rekkefølgen vi oppgir). Kompilatoren prøver også å gi variable en adresse som er et multiplum av *ordlengden* og kan derfor hoppe over celler (såkalt «padding»).

```
#include <stdio.h>

int a, b;
char u, v;
float f;

int main (void)
{
  printf("Variabelen a har adressen 0x%08x\n", &a);
  printf("Variabelen b har adressen 0x%08x\n", &b);
  printf("Variabelen u har adressen 0x%08x\n", &u);
  printf("Variabelen v har adressen 0x%08x\n", &v);
  printf("Variabelen f har adressen 0x%08x\n", &f);
}
```

```
Variabelen a har adressen 0x00600984
Variabelen b har adressen 0x0060098c
Variabelen u har adressen 0x00600988
Variabelen v har adressen 0x00600990
Variabelen f har adressen 0x00600994
```

Vektorer i C

Cellene i vektorer havner *alltid* pent etter hverandre.

```
#include <stdio.h>

short a[4];

int main (void)
{
  int i;

  for (i = 0;  i < 4;  ++i)
    printf("a[%d] har adressen 0x%08x\n", i, &a[i]);
}
```

```
a[0] har adressen 0x006008ac
a[1] har adressen 0x006008ae
a[2] har adressen 0x006008b0
a[3] har adressen 0x006008b2
```

struct-er i C

I struct-er kommer også elementene pent etter hverandre (eventuelt med litt «padding»):

```
#include <stdio.h>

struct s {
  int i;  char c, d;  float f;
};
struct s s1, s2;

int main (void)
{
  printf("s1.i har adressen 0x%08x\n", &s1.i);
  printf("s1.c har adressen 0x%08x\n", &s1.c);
  printf("s1.d har adressen 0x%08x\n", &s1.d);
  printf("s1.f har adressen 0x%08x\n", &s1.f);
  printf("s2.i har adressen 0x%08x\n", &s2.i);
  printf("s2.c har adressen 0x%08x\n", &s2.c);
  printf("s2.d har adressen 0x%08x\n", &s2.d);
  printf("s2.f har adressen 0x%08x\n", &s2.f);
}
```

```
struct s {
  int i;  char c, d;  float f;
};
struct s s1, s2;
```

```
s1.i har adressen 0x006009c4
s1.c har adressen 0x006009c8
s1.d har adressen 0x006009c9
s1.f har adressen 0x006009cc
s2.i har adressen 0x006009d0
s2.c har adressen 0x006009d4
s2.d har adressen 0x006009d5
s2.f har adressen 0x006009d8
```

union-er i C

Noen ganger er man interessert i å plassere data «oppå hverandre» i minnet. Dette kan gjøres med en union.

```
union u {
  int ui;  float uf;  char ub[4];
} uvar;
```

```
int main (void)
{
  printf("uvar.ui har adressen 0x%08x\n", &uvar.ui);
  printf("uvar.uf har adressen 0x%08x\n", &uvar.uf);
  printf("uvar.ub har adressen 0x%08x\n", &uvar.ub);

  uvar.ui = 13;
  printf(" 13 har bytene 0x%02x 0x%02x 0x%02x 0x%02x\n",
         uvar.ub[0], uvar.ub[1], uvar.ub[2], uvar.ub[3]);

  uvar.uf = 2.5;
  printf("2.5 har bytene 0x%02x 0x%02x 0x%02x 0x%02x\n",
         uvar.ub[0], uvar.ub[1], uvar.ub[2], uvar.ub[3]);
}
```

```
uvar.ui har adressen 0x006009d4
uvar.uf har adressen 0x006009d4
uvar.ub har adressen 0x006009d4
 13 har bytene 0x0d 0x00 0x00 0x00
2.5 har bytene 0x00 0x00 0x20 0x40
```

Oppsummering

Programmeringsspråket **C**

- et meget nyttig verktøy (spesielt når man jobber nær maskinen)
- et godt språk å programmere i
- krever en del trening for å unngå vanlige feil