

Programmering i C

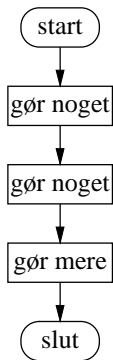
Lektion 3

29 november 2006

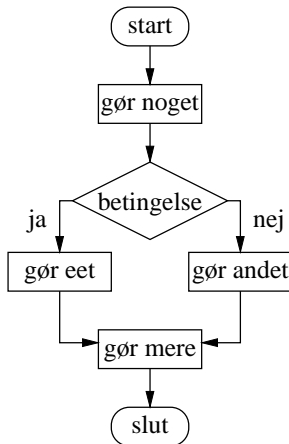
Fra sidst

- 1 Kontrolstrukturer
- 2 Udvælgelse
- 3 Gentagelse
- 4 Eksempler

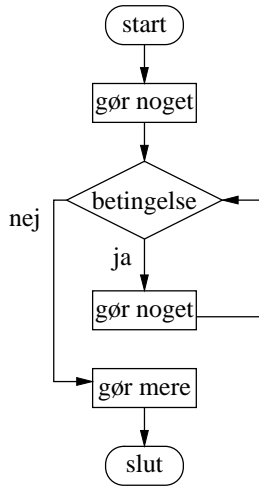
Sekventiel kontrol



Udvælgelse



Gentagelse



- med **if**

```
if ( udtryk ) kommando1; else kommando2;
```

- med **switch**

```
switch( udtryk ) {  
  case const1: command1;  
  case const2: command1;  
  ...  
  case constN: commandN;  
  default: command;  
}
```

- med **den betingede operator ?:**

```
udtryk ? udtryk1 : udtryk2
```

f.x. min=(a< b? a: b);

(smart, men undgå!)

- med **while**

while(udtryk) kommando;

- med **for**

for(start; forts; update) kommando;

- med **do**

do kommando; **while**(udtryk)

f.x.

```
do scanf( "%c", &ans );
```

```
while( ans != 'n' && ans != 'y' );
```

- opgave 5 med `while`: `gaet.c`
- opgave 5 med `for` (måske lidt søgt ...): `gaet2.c`
- opgave 4: `dag.c`
- opgave 4, bedre: `dag2.c`

Funktioner

- 5 Funktioner
- 6 Eksempel
- 7 Parametre
- 8 Rekursive funktioner
- 9 Parametre til main()

- at opdele et større program i mindre enheder \Rightarrow funktioner
- abstraktion!
- top-down-programmering

```
type navn( parametre ) {  
    deklARATIONER;  
    kommandoer;  
}
```


Et program der indlæser et tal; hvis tallet er primtal udskrives
"PRIMA," ellers udskrives næststørste primtal:

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

```
int main( void) {  /* prim.c */
    int tal;

    tal= indlaes();  /* et funktionskald */
    if( prim( tal))  /* et funktionskald */
        printf( "PRIMA\n");
    else {
        tal= nextPrime( tal);  /* endnu et */
        printf( "Next prime is %d\n", tal);
    }

    return 0;
}
```

At indlæse et heltal:

```
/* en funktionsdefinition */  
int indlaes( void) {  
    int tal;  
  
    printf( "\nEnter a number: ");  
    scanf( "%d", &tal);  
  
    return tal;  
}
```

Find ud af om et heltal er et primtal (*Er det den bedste måde at gøre det på?*):

```
int prim( int tal) {  
    int isprime= 1;  
    int i;  
  
    for( i= 2; i<= tal- 1; i++) {  
        if( tal% i== 0) {  
            isprime= 0;  
            break;  
        }  
    }  
  
    return isprime;  
}
```

break: Springer ud af en **switch**, **while**, **do** eller **for**

Returner næste primtal:

```
int nextPrime( int tal) {  
    tal++;  
    while( !prim( tal)) tal++;  
  
    return tal;  
}
```

Bemærk genbrug af [prim](#)-funktionen.

Funktioner skal erklæres før de bliver brugt:

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

```
int indlaes( void );
```

```
int prim( int tal );
```

```
int nextPrime( int tal );
```

```
int main( void ) { /* prim.c */
```

```
    int tal;
```

```
    tal= indlaes(); /* et funktionskald */
```

```
    if( prim( tal )) /* et funktionskald */
```

```
        printf( "PRIMA\n" );
```

```
    else {
```

```
        tal= nextPrime( tal); /* endnu et */
```

```
        printf( "Next prime is %d\n", tal );
```

```
    }
```

```
    return 0;
```

```
}
```

Hele programmet: [prim.c](#)

Andet eksempel: opgave 4 med funktioner: [dag3.c](#)

```
type navn( parametre ) {  
    deklARATIONER;  
    kommandoer;  
}
```

- En parameter i en funktions*definition* kaldes en **formel parameter**. En formel parameter er et variabelnavn.
- En parameter i et funktions*kald* kaldes en **aktuel parameter**. En aktuel parameter er et udtryk der beregnes ved funktionskaldet.
- Antallet og typer af aktuelle parametre i kaldet skal modsvare antallet og typer af formelle parametre i definitionen.

```
definition: int days_per_month( int m, int y ) {
```

```
    kald:      dmax= days_per_month( m, y );
```

```
type navn( parametre ) {  
    deklARATIONER;  
    kommandoer;  
}
```

- En parameter i en funktions*definition* kaldes en **formel parameter**. En formel parameter er et variabelnavn.
- En parameter i et funktions*kald* kaldes en **aktuel parameter**. En aktuel parameter er et udtryk der beregnes ved funktionskaldet.
- Antallet og typer af aktuelle parametre i kaldet skal modsvare antallet og typer af formelle parametre i definitionen.
- I C overføres funktionsparametre som **værdiparametre**. Dvs.
 - værdien af parametren *kopieres* til brug i funktionen,
 - ændringer af værdien har ingen indvirkning på programmet udenfor funktionen,
 - når funktionskaldet ender, ophører værdien med at eksistere.

rekursiv funktion = funktion der *kalder sig selv*

Eksempel: fakultetsfunktionen: $n! = 1 \cdot 2 \cdot 3 \cdots n = n \cdot (n - 1)!$

```
unsigned long fakultet( unsigned long n) {  
    if( n== 1)  
        return 1;  
    else  
        return n* fakultet( n-1);  
}
```

[fak.c]

– smart og kompakt måde at kode på (men nogle gange ikke særlig hurtig afvikling)

Eksempel: Fibonaccital:

$$f_1 = 1 \quad f_2 = 1 \quad f_n = f_{n-1} + f_{n-2}$$
$$1, 1, 2, 3, 5, 8, 13, 21, 34, \dots$$

```
unsigned long fibo( int n) {  
    switch( n) {  
    case 1: case 2:  
        return 1; break;  
    default:  
        return fibo( n- 1)+ fibo( n- 2);  
    }  
}
```

[fibo.c]

```
int main( void ) {    – en funktion!
```

Generel form: **int** main(**int** argc , **char**** argv) {

Parametrene tages fra **kommandolinien**.

- **argc** er antallet af argumenter
- **argv** er et *array af strenge* med alle argumenter; **argv[0]** er programnavnet

Eksempel: `./argtest 15 hest`

[P]

```
⇒ argc== 3  
   argv[0]== "argtest"  
   argv[1]== "15"  
   argv[2]== "hest"
```

Eksempel: Et fakultetsprogram der tager tallet som input på kommandolinien:

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

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

```
unsigned long fakultet( unsigned long n);
```

```
int main( int argc, char** argv) {  /* fak2.c */
```

```
    char * myself= argv[0];
```

```
    unsigned long tal;
```

```
    char * endptr;  /* needed for strtol */
```

```
    if( argc== 1)
```

```
        printf( "Error: %s needs one argument\n", myself);
```

```
    else {  /* convert argv[1] to int */
```

```
        tal= strtol( argv[1], &endptr, 10);
```

```
        printf( "\nThe factorial of %lu is %lu\n", \
                tal, fakultet( tal));
```

```
    }
```

```
    return 0;
```

Programmeringsstil

- 10 Udseende
- 11 Kommentarer
- 12 Symbolske konstanter

C er et programmeringssprog i **fri format**, dvs. stor frihed mht. *formatering*: mellemrum, tabs og lineskift kan indsættes (og udelades) næsten overalt.

⇒ eget ansvar at koden er letlæselig!

- indentér!
- brug mellemrum omkring operatorer
- sæt afsluttende `}` på deres egen linie
- inddel koden i logiske enheder vha. tomme linier
- en masse andre (og til dels modsigende!) konventioner

⇒ find din egen stil!

Sætning: Kode er sværere at læse end at skrive.

⇒ brug *mange* kommentarer.

```
/*  en kommentar der  
    fylder 2 linier */
```

(Det er ikke kun *andre* der skal kunne forstå din kode; måske er det *dig selv* der 4 uger efter forsøger at finde ud af hvad det her program gør.)

- kommentér hver enkelt funktion
- indsæt programmets navn i en kommentar
- skriv en kommentar om hvad det her program gør (medmindre programmet selv fortæller det)
- hvis en kodelinie tog specielt lang tid at skrive, er den nok også svær at forstå. Skriv en kommentar.
- fortæl hvad variablene betyder

Hvis der er en konstant i dit program der ikke er lig 0 eller 1, vil du sandsynligvis lave den værdi om senere.

⇒ definér konstanten **symbolsk** vha. præprocessoren:

```
#define SVAR 42
```

og referér til det symbolske navn i koden:

```
printf( "The answer is %d" , SVAR);
```

– Præprocessoren erstatter, som det *første* skridt, *inden* kompilering, alle forekomster af **SVAR** i koden med **42**, undtagen hvis **SVAR** står som del af en streng.

Eksempel på god programmeringsstil: **dag2.c** ☺