Programmering i C

Lektion 4

5 december 2006

Fra sidst

Eksempel:

```
/* funktions-prototyper */
int indlaes( void);
void udskriv( int a);
char blabla( char c);
...

/* main-funktionen */
int main( void) {
...

/* funktions-definitioner */
int indlaes( void) {
...

void udskriv( int a) {
...
```

Hvorfor:

- top-down-programmering
- abstraktion
- “del-og-hersk”-princippet

Skriv et program der faktoriserer et heltal i primfaktorer.

```
int main( void) {
  unsigned int x, f;

  greeting();
  x= readPosInt();

  printf( "%u = ", x);

  while( x!= 1) {
    f= findFactor( x);
    printf( "%u * ", f);
    x= x/ f;
  }

  printf( "1\n");
  return 0;
}
```

Funktions-prototyper:

```
void greeting(
        void);
unsigned int readPosInt(
        void);
unsigned int findFactor(
        unsigned int x);
```

Funktioner:

```
void greeting( void) {
  printf( "\nWe factor a positive integer \
into primes.\n");
}
```

```
unsigned int readPosInt( void) {
  unsigned int input;

  printf( "Enter a positive integer: ");
  scanf( "%u", &input);

  return input;
}
```

Funktioner:

```
unsigned int findFactor( unsigned int x) {
  unsigned int i;
  int found_one= 0;

  for( i= 2; i<= (int)sqrt( x); i++)
    if( x% i== 0) {
      found_one= 1;
      break;
    }

  if( found_one)
    return i;
  else
    return x;
}
```

Hele programmet: factor.c

Datatyper

C er et programmeringssprog med statisk, svag typning:

- hver variabel har en bestemt type
- typen skal deklareres explicit og kan ikke ændres
- ved *kompilering* efterses om der er type-fejl
- mulighed for *implicitte* typekonverteringer

En variabels type bestemmer

- hvilke værdier den kan antage
- i hvilke sammenhænge den kan bruges

Typer i C:

- void, den tomme type
- skalære typer:
  - aritmetiske typer:
    - heltalstyper: short, int, long, char; enum
    - kommatals-typer: float, double, long double
  - pointer-typer
- sammensatte typer:
  - array-typer
  - struct

[typer.c]

- implicitte konverteringer:
  - *integral promotion*: short og char konverteres til int
  - *widening*: en værdi konverteres til en mere præcis type
  - *narrowing*: en værdi konverteres til en *mindre* præcis type. Information går tabt!

[conversions.c]

- eksplicitte konverteringer: ved brug af casts

```
for( i= 2; i<= (int)sqrt( x); i++)
```

Scope

Scope ("virkefelt") af en variabel er de dele af programmet hvor variablen er kendt og tilgængelig.

I C:

- Scope af en variabel er den blok hvori den er erklæret
- Variable i en blok "skygger" for variable udenfor der har samme navn

$\Rightarrow$ *huller i scope!*

Eksempel fra lektion 2:

```
#include <stdio.h>
int main(void){ /*blok.c*/
  int a=5;
  printf("Før: a==%d\n",a);

  {  /*en blok*/
    int a=7; /*deklaration*/
    printf("I: a==%d\n",a);
  }

  printf("Efter: a==%d\n",a);

  return 0;
}
```

Storage class af variable medvirker til at bestemme deres scope.

- auto (default): lokal i en blok
- static: lokal i en blok, *men bibeholder sin værdi* fra én aktivering af blokken til den næste. Eksempel:

```
#include <stdio.h>

int nextSquare( void) {
  static int s= 0;
  s++;
  return s*s;
}

int main( void) {
  int i;
  for( i= 1; i<= 10; i++)
    printf( "%d\n", nextSquare());
  return 0;
}
```

Tilbage til Fibonaccital:

$$f_1 = 1 \qquad f_2 = 1 \qquad f_n = f_{n-1} + f_{n-2}$$

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

Problem: kører meget langsomt pga. utallige genberegninger

Løsning: Husk tidligere beregninger vha. et static array (“*dynamisk programmering*”)

Memoriseret udgave af fibo: [fibo2.c]

```
unsigned long fibo( int n) {
  unsigned long result;
  static unsigned long memo[ MAX];
                /* this gets initialised to 0 ! */
  switch( n) {
  case 1: case 2:
    return 1; break;
  default:
    result= memo[ n];
    if( result== 0) { /* need to compute */
      result= fibo( n- 1)+ fibo( n- 2);
      memo[ n]= result;
    }
    return result;
  }
}
```

Programmeringsstil

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

$\Rightarrow$ 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

$\Rightarrow$ find din egen stil!

Sætning: Kode er sværere at læse end at skrive.

$\Rightarrow$ 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 ‿