

Eksempel:

```

1  /* funktions-prototyper */
   int indlaes(void);
4  void udskriv(int a);
   char blabla(char c);
   ...
7  /* main-funktionen */
   int main(int argc, char** argv) {
   ...
10 ...
   /* funktions-definitioner */
   int indlaes(void) {
13 ...
   void udskriv(int a) {
16 ...

```

Eksempel

Funktioner

Eksempel

3 / 27

Programmering i C

Lektion 3

15. september 2009

Funktioner

Fra sidst



Funktioner

Eksempel

Hvorfor:

- Opdeling i mindre enheder
- Genbrug
- Top-down-programmering
- Abstraktion

Spørsmål fra i går:

- #include or #import
- <http://stackoverflow.com/questions/172262/c-include-and-import-difference>

• Funktions-prototyper:

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

- Skriv et program der faktorerer et heltal i primfaktorer.

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

    greeting();
    x = readPosInt();

    printf("%d = ", x);

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

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

Funktions:

```
/* Say what we are doing */
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("%d", &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](#)

9/27

Typer

Typekonvertering

Arrays

Typer

Typekonvertering

Arrays

Datatyper

C er et programmeringssprog med **statisk**, **svag** typning:

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

[\[typer.c\]](#)

- 3 Typer
- 4 Typekonvertering
- 5 Arrays

- **Boolean values**
- <http://www.lysator.liu.se/c-FAQ/c-8.html>

Type	Bytes	Bits	Range	
short int	2	16	-32,768 -> +32,767	(32Kb)
unsigned short int	2	16	0 -> +65,535	(64Kb)
unsigned int	4	32	0 -> +4,294,967,295	(4Gb)
int	4	32	-2,147,483,648 -> +2,147,483,647	(2Gb)
long int	4	32	-2,147,483,648 -> +2,147,483,647	(2Gb)
signed char	1	8	-128 -> +127	
unsigned char	1	8	0 -> +255	
float	4	32		
double	8	64		
long double	12	96		

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

```
typedef struct {
    unsigned int alder;
    int penge;
    char* navn;
} Person;
```

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

```
    Person p1;
    p1.alder = 30;
    p1.penge = -300;
    p1.navn = "Ulrik Nyman";

    printf("%s er %d år og har %d penge",
           p1.navn, p1.penge, p1.penge);
```

```
    return (0);
}
```

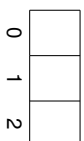
- **implicitte konverteringer:**
 - *integral promotion:* **short** og **char** konverteres til **int**
 - *widening:* en værdi konverteres til en mere præcis type
 - *narrowing:* en værdi konverteres til en mindre præcis type. Information går tabt!

[\[conversions.c\]](#)

- **eksplicitte konverteringer:** ved brug af **casts**
- ```
for (i = 2; i <= (int)sqrt(x); i++) {
```

Et **array** er en tabel af variable af samme type der kan tilgås via deres indeks.

```
int tal [3];
```



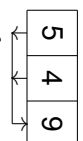
```
tal [0]=5;
```



```
tal [1]=4;
```



```
tal [2]= tal [0]+ tal [1];
```



- et array skal deklareres med angivelse af type, og helst også størrelse: **type a[M]**

- laveste indeks er 0, højeste er  $N - 1$

- indgangene lagres **umiddelbart efter hinanden**

17/27

**Pas på!** C ser ikke efter om et indeks man forsøger at tilgå ligger indenfor arrayets grænser:

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

```
int main(void) { /* array-bad. c */
int a[3];
```

```
/* Menigsløst resultat */
```

```
printf("%d\n", a[3]);
```

```
/* FARLIGT! */
```

```
/* a[3]= 17; */
```

```
return 0;
```

```
}
```

Programmet skriver i et hukommelsesområde det ikke har reserveret! I bedste tilfælde er det kun programmet der crasher ...

18/27

## Scope

- 6 Scope
- 7 Storage class
- 8 Memorisering

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

- C: Scope af en variabel er den blok hvori den er erklæret
- Variabel i en blok "skygger" for variabel udenfor der har samme navn

⇒ *huller i scope!*

Eksempel fra lektion 1:

```
#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;
```

```
}
```

20/27

**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;
}
```

21/27

Tilbage til Fibonacci!:

```
f1 = 1 f2 = 1 fn = fn-1 + fn-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:

[fiboz.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;
 }
}
```

21/27

23/27

## Programmeringsstil

- 9 Udseende
- 10 Kommentarer
- 11 Symboliske konstanter

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

⇒ eget svar 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!

26/27

**Seættning: 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

26/27

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 **symbolisk** vha. præprocessoren:

```
#define SVAR 42
```

og referér til det symboliske 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.

27/27