

```
>>> C-Programmering for begyndere  
>>> Del 3 - Funktioner, arrays, datarepræsentationer
```

```
Name:  Jacob B. Pedersen† og Jakob S. Nielsen‡  
Date:  23.  april 2018
```

---

<sup>†</sup>jacob.bp@mb.net

<sup>‡</sup>jakob990@gmail.com

```
>>> Indhold
```

## 1. Repetition

Hvad lavede vi sidste gang?

## 2. I dag

Dagens program

Funktioner

Arrays

Typedef

Enum

Structures

Unions

>>> Hvad lavede vi sidste gang?

- \* Vi gennemgik operatorer, og deres betydning for if-else statements:

---

```
1  if([betingelse]){
2      handling();
3  }
4  else if({betingelse2]){
5      handling2();
6  }
7  else{
8      handling3();
9  }
```

---

```
>>> Hvad lavede vi sidste gang?
```

- \* Operatorerne selv evalueredes blot til Boolske udtryk som `true` og `false`:

---

```
1      1 > 2; // false
2      3 >= 3; // true
3      12 > 2 && 12 < 11; // false
4      12 > 2 || 12 < 11; // true
```

---

>>> Hvad lavede vi sidste gang?

- \* På samme måde kunne de også bruges som betingelser i løkkerne

- \* Her var der `while()` og `for()` -løkkerne:

---

```
1     int i = 0;
2     while(i <= 10){
3         printf(''%d'', i);
4         i++;
5     }
6
7     for(int j = 0; j <= 10, j++){
8         printf(''%d'', j);
9     }
```

---

>>> Hvad lavede vi sidste gang?

\* Til sidst kiggede vi på `switch`-cases:

---

```
1      switch(variabel)
2      {
3          case 1:
4              handling();
5              break;
6
7          case 2:
8              andenHandling();
9              break;
10
11         default:
12             break;
13     }
```

---

```
>>> Dagens program
```

- \* I dag kigger vi på mere avanceret brug af C:
- \* Funktioner
  - \* Genopfriske argumenter og retur-værdier
  - \* Header- og implementationsfiler
  - \* Hvad skal der til for at lave et library?
- \* Arrays
  - \* Lister over variable, refereret til ved indeks
  - \* Nogle af jer har forsøgt sig med char array AKA strings
- \* Anderledes datarepræsentation:
  - \* `typedef` og `enum`
  - \* Klynger af data i `structs`
  - \* Flerformet data i `unions`

## >>> Funktioner

- \* Vi har ofte brugt funktioner:

- \* `pow()`, `sqrt()`, `rand()`, `printf()` og `scanf()`

- \* De ligner alle i skrift lidt matematiske funktioner:

$$\textit{printf}(x) \quad (1)$$

- \* Og vi kan selvfølgelig også designe vores egne!

- \* Vi husker tydeligt en funktionsdefinition, vi har med hver gang:

---

```
1  int main(int argc, char ** argv){  
2      return 0;  
3  }
```

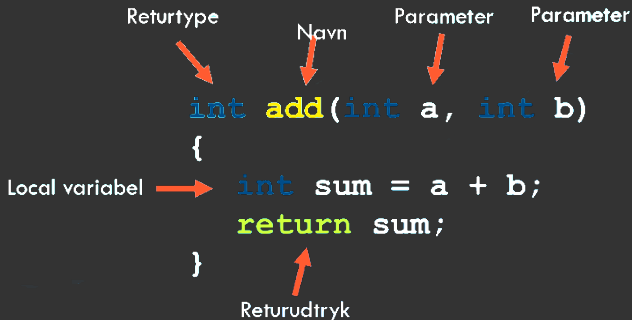
---



## >>> Funktioner

- \* I funktionerne er der en række begreber vi skal genopfriske/kende:
  - \* **returtype**, **argumenttype/parameter**, **returværdi** og **lokale variable**:

**add**(a, b);



## >>> Funktioner

- \* Så først defineres funktionen, og derefter kan den kaldes som alt andet i main:

---

```
1  int add( int a, int b ){
2      int result = a + b;
3      return result;
4  }
5
6  int main(void){
7      int a = 3;
8      int b = 2;
9      printf( "%d + %d = %d", a, b, add(a,b) );
10 }
```

---

## >>> Funktioner

- \* Noget andet fedt man kan med sine funktioner, er at kalde dem rekursivt:

---

```
1  int rekursion(int antalGange){
2      if(antalGange < 0){
3          return 0;
4      }
5      printf(''%d'', rekursion(antalGange-1));
6      return antalGange;
7  }
```

---

- \* Man skal bare huske at sikre at computeren kan komme ud af de rekursive kald igen
- \* Ellers risikerer man et:



stack overflow

## >>> Funktioner

- \* Vi har berørt libraries lidt før:
  - \* `stdio.h`, `math.h`, `stdbool.h`
  - \* De er alle sammen standard libraries
  - \* Er enten en del af compileren eller operativsystemet
- \* Men vi kan sagtens lave vores egne!
  - \* Kræver blot en header-fil og implementationsfil:
  - \* `library.h` og `library.c`
  - \* Headeren er hvor vi deklarerer (/prototyper) vores variable og funktioner
  - \* Implementationsfilen er til at beskrive den specifikke funktionalitet

>>> Funktioner

\* Eksempel på en header:

---

```
1      // Header skrives med "include guard" og instantierer
      funktionerne:
2      #ifndef LIBRARY_H
3      #define LIBRARY_H
4
5      int add(int x, int y);
6
7      #endif /* LIBRARY_H */
```

---

\* Eksempel på en implementation:

---

```
1      // Implementationen inkluderer headeren, og beskriver
      funktionerne:
2      #include 'library.h'
3
4      int add (int x, int y){
5          int sum = x + y;
6          return sum;
7      }
```

---

## >>> Arrays

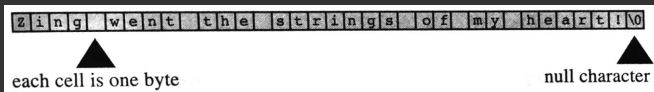
- \* Vi har allerede stiftet bekendtskab med char arrays i opgavesæt 1
- \* Den opfører sig som en liste af chars, og man kan bruge den således:

---

```
1  char inputstring[LENGTH]; // Angiv længden af arrayet,  
    så computeren kan sætte pladsen til side!  
2  scanf(''%s'', &inputString); // scanner efter en string  
    indtil første mellemrum, newline etc.
```

---

- \* Det var der en del der fik til at fungere!
- \* Hvordan ser computeren så de data?



## >>> Arrays

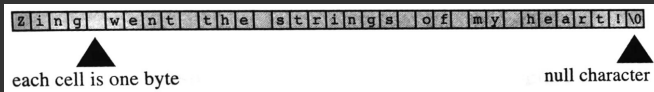
- \* Vi har allerede stiftet bekendtskab med char arrays i opgavesæt 1
- \* Den opfører sig som en liste af chars, og man kan bruge den således:

---

```
1 char inputstring[LENGTH]; // Angiv længden af arrayet,  
   så computeren kan sætte pladsen til side!  
2 scanf(''%s'', &inputString); // scanner efter en string  
   indtil første mellemrum, newline etc.
```

---

- \* Det var der en del der fik til at fungere!
- \* Hvordan ser computeren så de data?



## >>> Arrays

- \* Char arrays er klart en oplagt brugt af denne type "lister"

- \* Men der findes arrays af alle typer:

---

```
1 // F.eks. en integer array:
2 int heltal[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
3 // Eller float array:
4 float kommatal[] = {0.1, 1.5, 2.46, 3.23, 4.44, 5.71,
    6.66, 7.91, 8.0, 9.11, 10.4};
```

---

- \* Der findes andre måder at opstille sådanne lister f.eks. dynamisk;
- \* det er dog et avanceret emne, med skjulte fælder!



## >>> Arrays

- \* Når man tilgår array elementer, sker det vha. indeks:

---

```
1 // Her på en string:
2 char string[] = Zing went the string of my heart!;
3 printf(''%c'', string[0]);
4 // Vil printe værdien 'Z'.
```

---

- \* Altså er arrays 0-indekserede!
- \* men deres størrelse er angivet i antal elementer!,  
minimum 1! som får indeks 0

## >>> Typedef

- \* Hvad så med data der ikke er overskueligt i formen int, float osv.?
- \* Her kan `typedef` redde os:

---

```
1      // Et smart eksempel kunne være at gøre en fysisk
      udregning overskuelig:
2      typedef float length;
3      typedef float width;
4      typedef float height;
5      typedef float volume;
6      // Det er altid vigtigt at koden er læselig for
      beskueren!
7      length l = 23;
8      width w = 32.5;
9      height h = 55;
10     volume v = l * w * h;
```

---

>>> Enum

- \* Modsvar til `typedef`, `enum` bruges til at navngive værdier i stedet for datatyper

---

```
1 // Hvis man skriver dem således, bliver de fortløbende
   nummereret fra 0:
2 enum color { red, orange, yellow, green, blue, purple };
3 // Ellers kan man selv bestemme hvad talværdi de har
   tilknyttet:
4 enum color { red = 3, orange = 4, yellow = 0, green = 1,
   blue = 2, purple = 2 };
5
6 // Variable af enum typen deklarereres/initialiseres
   således:
7 enum color favouriteColor = purple;
```

---

- \* Man kan også bruge `enum` med switch-cases!

## >>> Structures

- \* En rigtig fed måde at repræsentere måde på
- \* `struct` minder lidt om en tabelrække i databaser

---

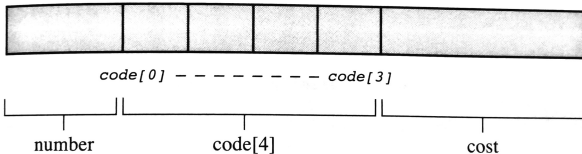
```
1      // Struct beskrivelse af en genstand i f.eks.  
      varelagersystem:  
2      struct stuff{  
3          nt number;  
4          char code[4];  
5          float cost;  
6      };
```

---

## >>> Structures

- \* På PC'en lagres en structs data således:

```
struct stuff {  
    int number;  
    char code[4];  
    float cost;  
};
```



## >>> Structures

- \* Man tilgår datafelterne i en struct vha. membership operatoren:

---

```
1  struct stuff{
2
3  }
4  stuff book = { .number = 123,
5                 .code = 'ab12',
6                 .cost = 3.99 };
7  printf('The books number is %d, code is %s, and it
        costs %f', book.number, book.code, book.cost);
```

---

## >>> Unions

- \* Modsvar til `struct`, er det man kalder en `union`
- \* Den bruges til data, man ikke helt kan garantere typen på
- \* Længden i hukommelse afgøres af den længste datatype, her `double`:

---

```
1      union varchar{
2          int digit;
3          double bigFloat;
4          char letter;
5      }
```

---

## >>> Unions

- \* Den kan kun holde én værdi ad gangen
- \* Ens eget ansvar at holde styr på typen den holder

---

```
1 // Man kan instantiere dem som man altid har gjort:
2 union varchar a;
3 // initialisere dem vha. dot operator:
4 a.letter = R ;
5 // Ud fra en anden union:
6 union varchar b = a;
7 // Eller målrettet enten den første member, eller
  // specifik member i declarationen:
8 union varchar c = {12};
9 union varchar d = {.bigFloat = 12345.678};
```

---