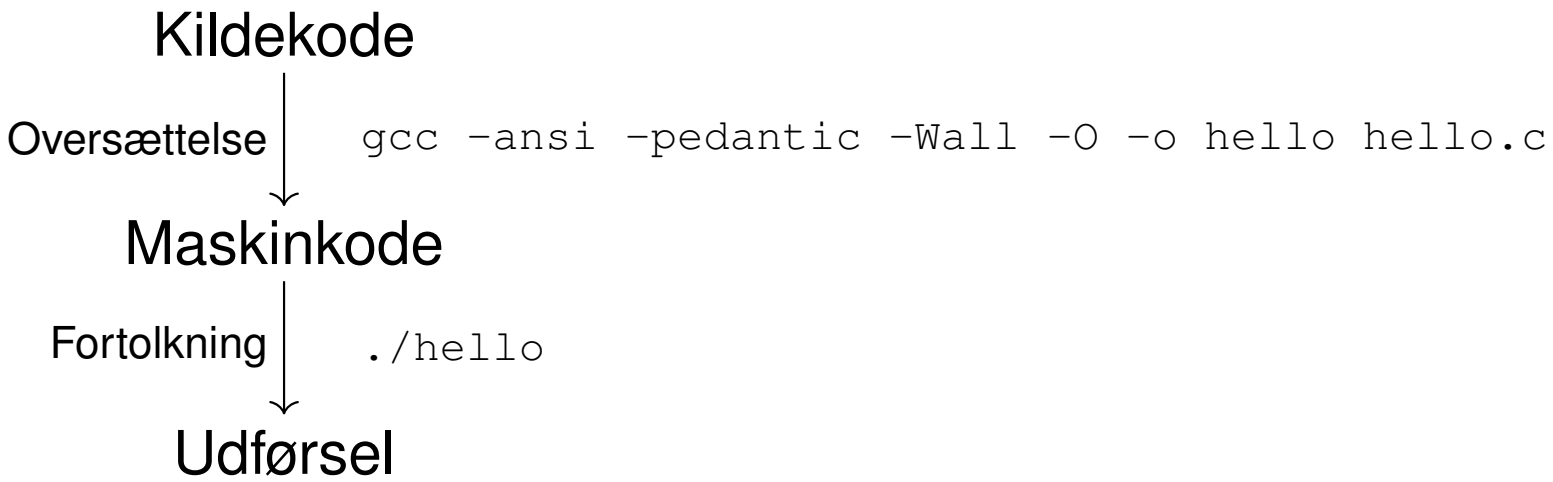
Programmering i C

Lektion 5

28. september 2009




```c
#include <stdio.h>

int main( void) { /* helloworld.c */

  printf( "Hello, world!\n");
  return 0;
}
```

- en variabel er en navngiven plads i computerens lager
- en variabel kan indeholde en værdi af en bestemt type
- variables værdier kan ændres ved assignment-kommandoer
- variable skal erklæres før brug

```c
#include <stdio.h>

int main(void) { /* variable.c */
  int a, b, c;
  a = 5;
  b = 3;
  c = a/ b;
  printf( "%d divideret med %d giver %d\n",
          a, b, c);
  printf( "Hov, hvad er nu det?\n");
  return 0;
}
```

- en variabel er en navngiven plads i computerens lager
- en variabel kan indeholde en værdi af en bestemt type
- variables værdier kan ændres ved assignment-kommandoer
- variable skal erklæres før brug

```c
#include <stdio.h>

int main(void) { /* variable2.c */
  int a = 5, b = 3, c;
  c = a/ b;
  printf("%d divideret med %d giver %d\n",
         a, b, c);
  printf("Hov, hvad er nu det?\n");
  return 0;
}
```

- en variabel er en navngiven plads i computerens lager
- en variabel kan indeholde en værdi af en bestemt type
- variables værdier kan ændres ved assignment-kommandoer
- variable skal erklæres før brug
- variable skal altid tildeles startværdier

```c
#include <stdio.h>

int main(void) { /* variable-noinit.c */
  int a, b, c;
  c = a/ b;
  printf("%d divideret med %d giver %d\n",
         a, b, c);
  printf("Hov, hvad er nu det?\n");
  return 0;
}
```

heltal	reelle tal	tegn	strenge
`short`	`float`	`char`	`char *`
`int`	`double`		
`long`	`long double`		

```c
#include <stdio.h>

int main(void) { /* variable-float.c */
  int a = 5, b = 3;
  double c;
  c = (double)a/ b;
  printf("%d divideret med %d giver %f\n",
         a, b, c);
  printf("Det var bedre!\n");
  return 0;
}
```

```c
#include <stdio.h>

int main(void) { /* elefant.c */
  int a = 1;
  printf("%d elefant kom marcherende, \
hen ad edderkoppens fine spind\n", a);

  while(a <= 10) {
    a = a + 1;
    printf("%d elefanter kom marcherende, \
hen ad edderkoppens fine spind\n", a);
  }

  return 0;
}
```

assignment/tildeling

$$\overbrace{c = \underbrace{a \ / \ b}}$$

expression/udtryk

Udtryk:

- 7
- x, a, b
- a + b, a − b
- a ∗ b, a / b, a % b
- a < b, a <= b, a == b etc. (boolske udtryk)

rest ved (heltals)division

Prioritering: ∗ beregnes før + etc.:

$$3 + 5 * 7 = 3 + (5 * 7)$$

Associering: Operationer med samme prioritet foretages fra venstre til højre:

$$10 - 5 - 2 = (10 - 5) - 2 \neq 10 - (5 - 2)$$

- a = i + 5: udtrykket i + 5 beregnes, og a tildeles den beregnede værdi
- dvs. + har højere prioritet end =
- men i C er a = i + 5 også et udtryk! Udtrykkets værdi er ligeledes i + 5

⇒ misbrug:

```c
#include <stdio.h>

int main(void) { /* misbrug.c */
  int a, b, c;
  a = b = c = 7;
  printf("a: %d, b: %d, c: %d\n", a, b, c);
  a = 1 + (b = 2*(c = 3));
  printf("a: %d, b: %d, c: %d\n", a, b, c);
  return 0;
}
```

- increment-operator: skriv i++ eller ++i i stedet for i = i + 1
- decrement-operator: skriv i−− eller −−i i stedet for i = i − 1
- men det er også et udtryk . . . :
  - i = 7; a = ++i ⇒ i=8, a=8
  - i = 7; a = i++ ⇒ i=8, a=7 ! Hvorfor?

- også akkumulerende assignment-operatorer:

a += 5	a = a + 5
a −= 7	a = a − 7
a ∗= 4	a = a ∗ 4
a /= 3	a = a / 3 etc.

Udskrivning med printf :

- printf(*kontrolstreng*, *parametre*)
- kontrolstreng: almindelige tegn udskrives uændret, konverteringstegn erstattes med parametre, som er formateret i h.t. konverteringsspecifikationen
- printf returnerer antallet af udskrevne tegn
- se `printf-eks.c`

Udskrivning med printf :

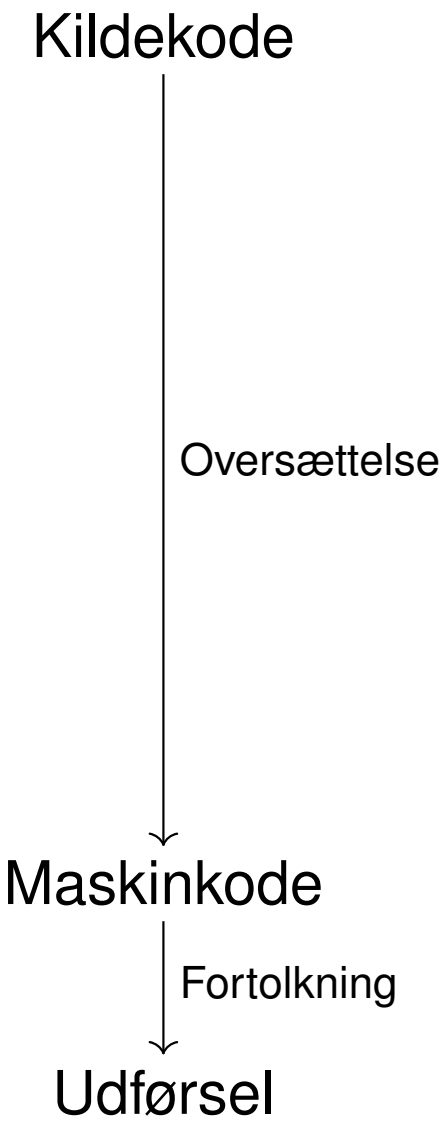
- printf(*kontrolstreng*, *parametre*)
- kontrolstreng: almindelige tegn udskrives uændret, konverteringstegn erstattes med parametre, som er formateret i h.t. konverteringsspecifikationen
- printf returnerer antallet af udskrevne tegn
- se `printf-eks.c`

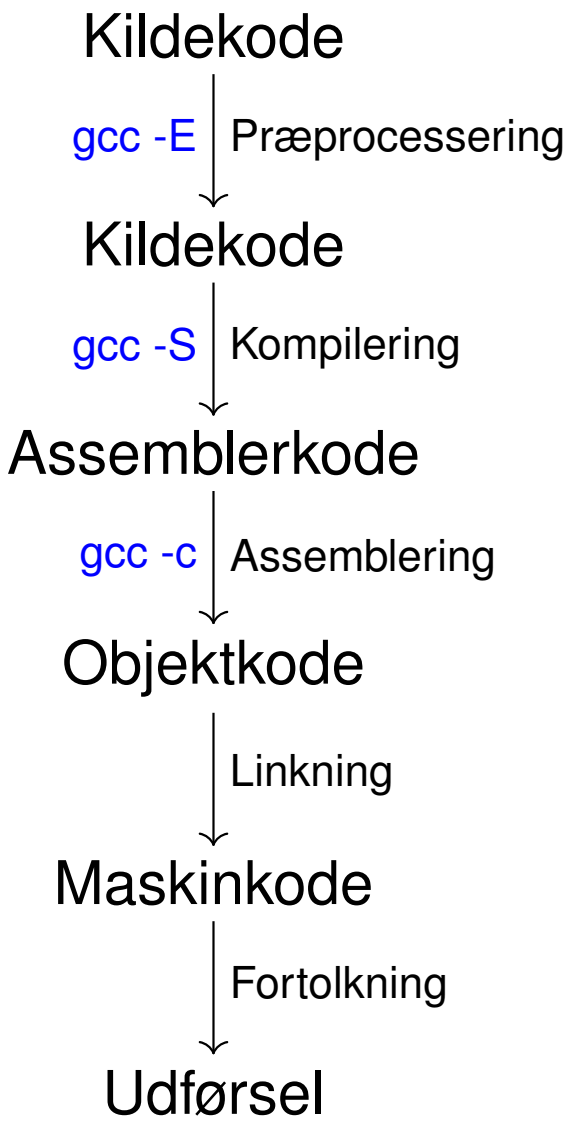
Indlæsning med scanf:

- scanf(*kontrolstreng*, *parametre*)
- kontrolstreng (næsten) analog til printf, men parametrene skal være adresser på variable (pointere): `&a`
- scanf returnerer antallet af gennemførte indlæsninger
- se `scanf-eks.c`

Et større eksempel:

```c
#include <stdio.h>

#define PI 3.141592653589793

int main( void) {  /* circle.c */
   double radius;

   printf( "\n%s\n\n%s",
      "This program computes the area of a circle.",
      "Input the radius:  ");

   scanf( "%lf", &radius);

   printf( "\n%s\n%s%.2f%s%.2f%s%.2f\n%s%.5f\n\n",
      "Area = PI * radius * radius",
      "     = ", PI, " * ", radius, " * ", radius,
      "     = ", PI * radius * radius );

   return 0;
}
```


Kildekode
Oversættelse
Maskinkode
Fortolkning
Udførsel


Kildekode
gcc -E
Præprocessering
Kildekode
gcc -S
Kompilering
Assemblerkode
gcc -c
Assemblering
Objektkode
Linkning
Maskinkode
Fortolkning
Udførsel

```
gcc -ansi -pedantic -Wall -O -o ting ting.c
```

Output skal hedde `ting`

Vi optimerer for at fange flere mulige fejl

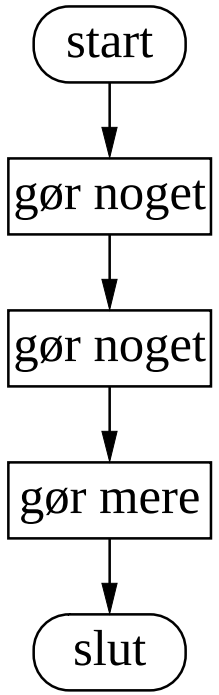
Vi vil også have en masse andre advarsler

Vi vil have alle advarsler som ANSI kræver

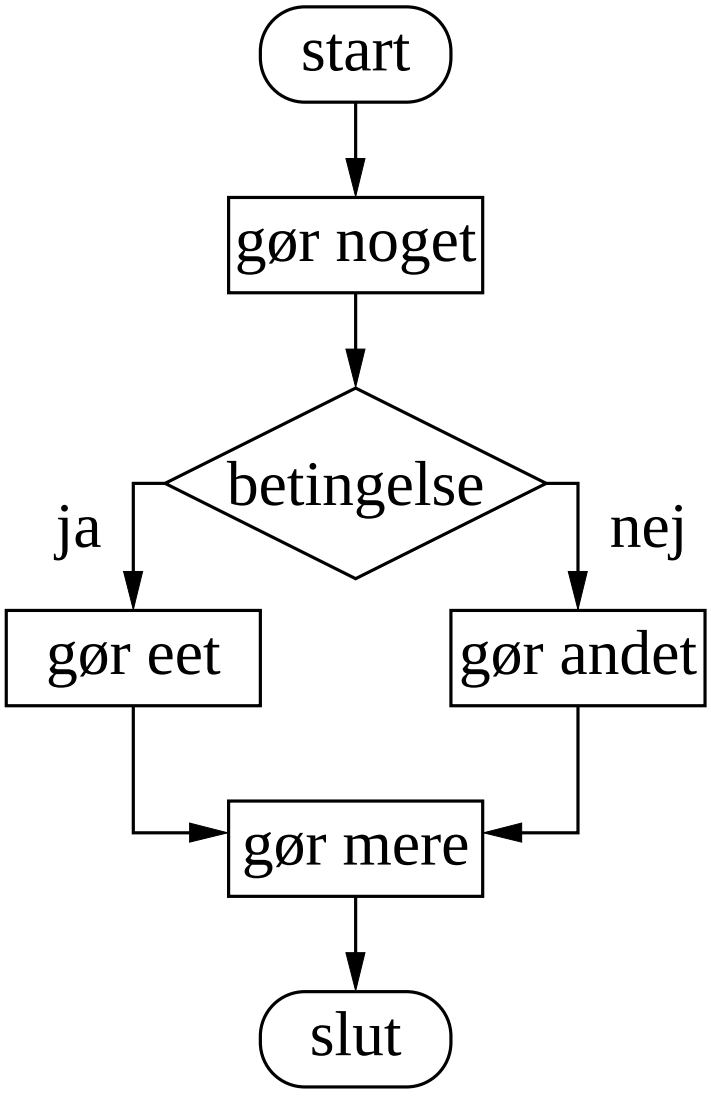
Vi følger ANSI-standarden

Kontrolstrukturer

```c
#include <stdio.h>

int main( void) {  /* seconds.c */
  long int input, temp, h, m, s;
  printf( "Giv mig et heltal!\n");
  scanf( "%ld", &input);
  h= input/ 3600;
  temp= input- h* 3600;
  m= temp/ 60;
  s= temp% 60;
  printf( "\n%ld sekunder svarer til \
%ld timer, %ld minutter og %ld sekunder\n",
          input, h, m, s);
  return 0;
}
```

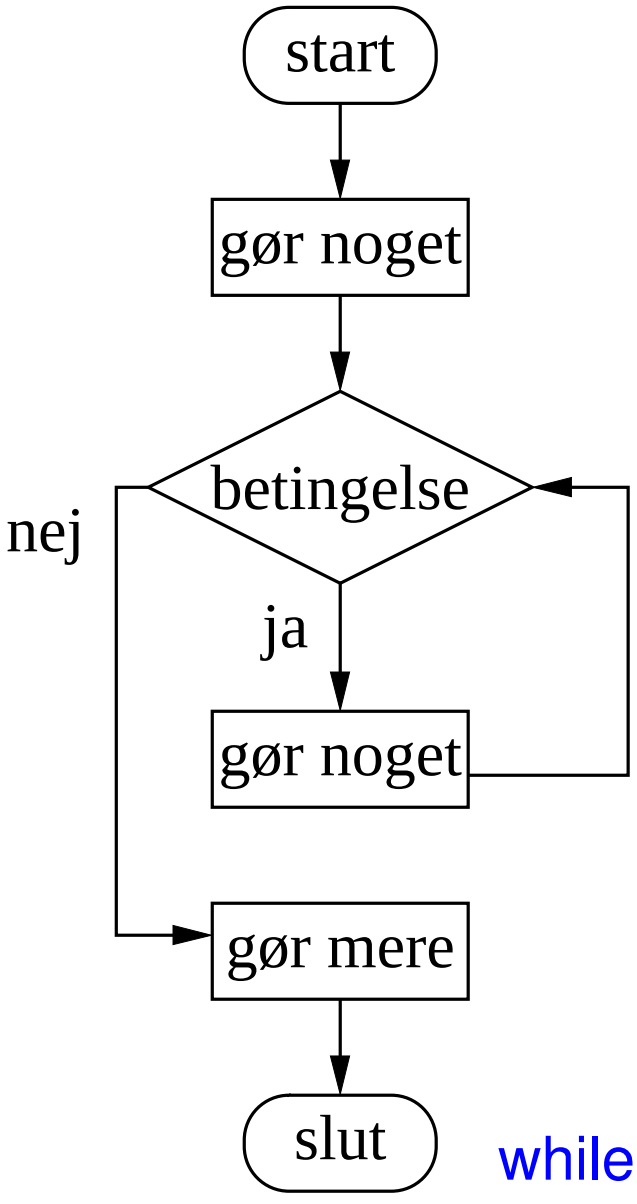


Udvælgelse af kommandoer:



if

Gentagelse af kommandoer:



while; for

Udvælgelse: if(*logisk udtryk*)
Gentagelse: while(*logisk udtryk*)

Logiske udtryk:

- x<y, x<=y, x>=y, x>y, x!=y, x==y
- !A, A&&B, A||B, hvor A og B selv er logiske udtryk
- har værdien *falsk* (0) eller *sandt* (1, i de fleste(!) compilere)
- && har højere prioritet end ||

$\Rightarrow$ brug parenteser!
(Hvad er værdien af 3==5 || 1==1 && 1==2 ?...) [oper.c]

```c
# include <stdio.h>

int main( void) {  /* lighed.c */
  int a, b, lig;

  printf( "Vi sammenligner to tal.\n\
Output 0 betyder at de er forskellige.\n\n\
Må jeg bede om to heltal?\n");
  scanf( "%d %d", &a, &b);

  lig = (a == b);

  printf( "\nOutput: %d\n", lig );
  return 0;
}
```

Observation:

- Hvis A er falsk, da er A&&B også falsk
- Hvis A er sandt, da er A||B også sandt

$\Rightarrow$ i udtrykket A&&B beregnes B kun hvis A er sandt
– og i udtrykket A||B beregnes B kun hvis A er falsk

- Smart, men kilde til fejl

```c
# include <stdio.h>

int main(void) {  /* lighed2.c */
  int a, b;
  char lig;

  printf("Vi sammenligner to tal.\n\n\
Må jeg bede om to heltal?\n");
  scanf( "%d %d", &a, &b);

  (a == b) && (lig = ' ');
  (a != b) && (lig = 'u');

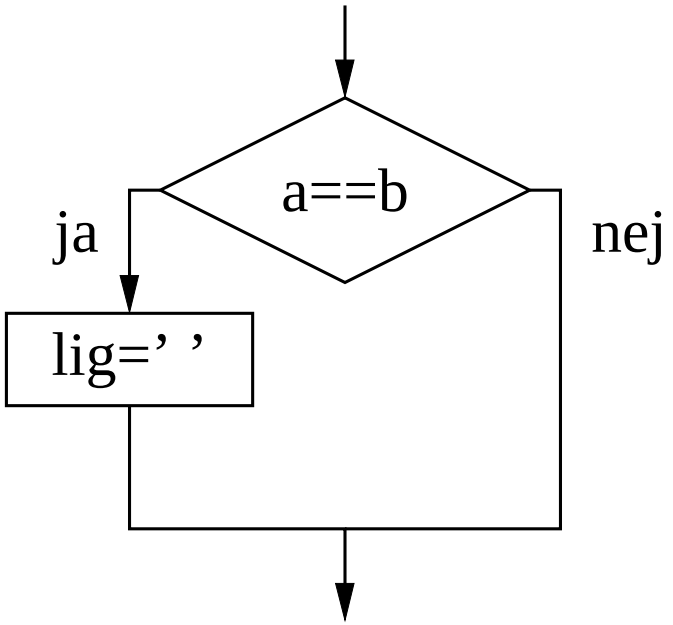
  printf("%d er %clig %d\n", a, lig, b);
  return 0;
}
```

Udvælgelse med &&:

- `(a == b) && (lig = ' ');`
- kryptisk...

Udvælgelse med if:

- `if (a== b) lig = ' ';`
- det var bedre!

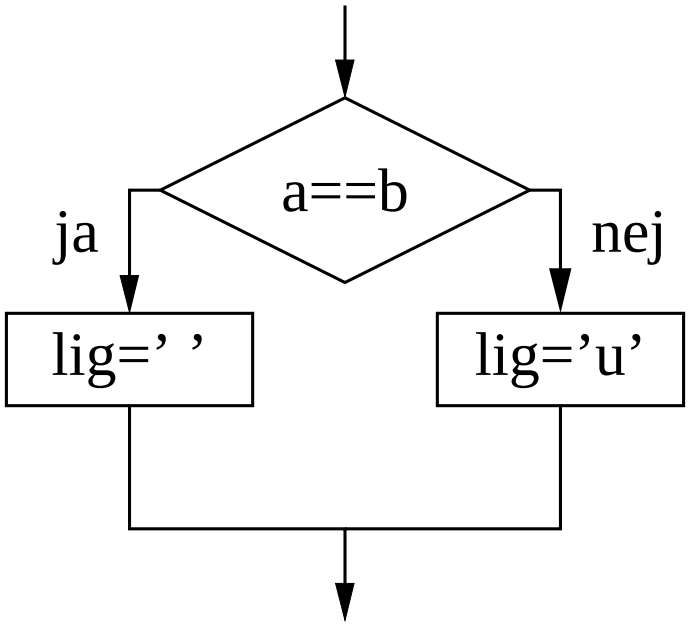


Udvælgelse med &&:

- `(a == b) && (lig = ' ');`
- kryptisk...

Udvælgelse med if:

- `if (a== b) lig = ' ';`
- det var bedre!
- ```
  if (a==b) lig = ' ';
  else lig = 'u';
  ```



`if (udtryk) kommando1; else kommando2;`

- først beregnes udtryk
- hvis udtryk er sandt, udføres kommando1
- hvis udtryk er falsk, udføres kommando2
  ```

```c
# include <stdio.h>

int main(void) {  /* lighed3.c */
  int a, b;
  char lig;

  printf("Vi sammenligner to tal.\n\n\
Må jeg bede om to heltal?\n");
  scanf( "%d %d", &a, &b);

  if (a== b) {
    lig = ' ';
  } else {
    lig= 'u';
  }
  printf("%d er %clig %d\n", a, lig , b);
  return 0;
}
```

Kontrolstrukturer, 2.

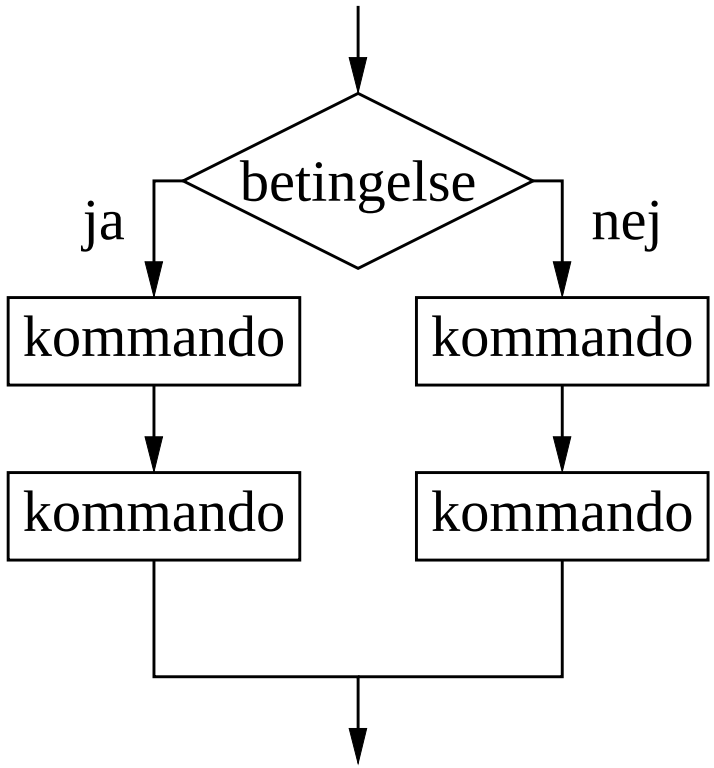
Problem: Vil gerne udvælge mellem to blokke af kommandoer

Løsning: Sammensætning af kommandoer:

```
if ( a== b)
{
   c= 1;
   d= 2;
}
else
{
   c= 7;
   d= 5;
}
```

blok

blok



- blok = antal kommandoer omkranset af { og }
- en blok behandles som én kommando
- blokke kan indlejres i hinanden

- blok = antal kommandoer omkranset af { og }
- en blok behandles som én kommando
- blokke kan indlejres i hinanden
- i starten af en blok kan variabelerklæringer forekomme

!! disse variable er lokale for blokken (deres scope er blokken)

- blok = antal kommandoer omkranset af { og }
- en blok behandles som én kommando
- blokke kan indlejres i hinanden
- i starten af en blok kan variabelerklæringer forekomme

!! disse variable er lokale for blokken (deres scope er blokken)

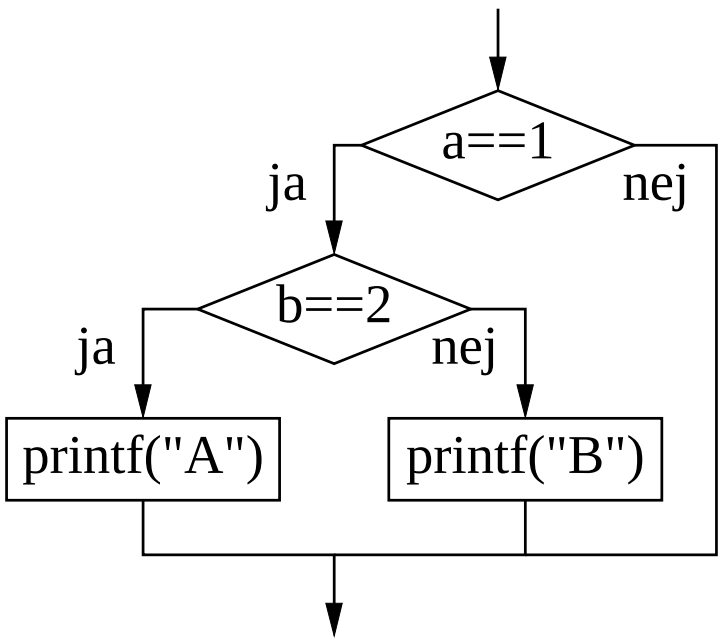
```c
#include <stdio.h>
int main(void){  /*blok.c*/
  int a=5;
  printf("Før: a==%d\n",a);

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

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

  return 0;
}
```

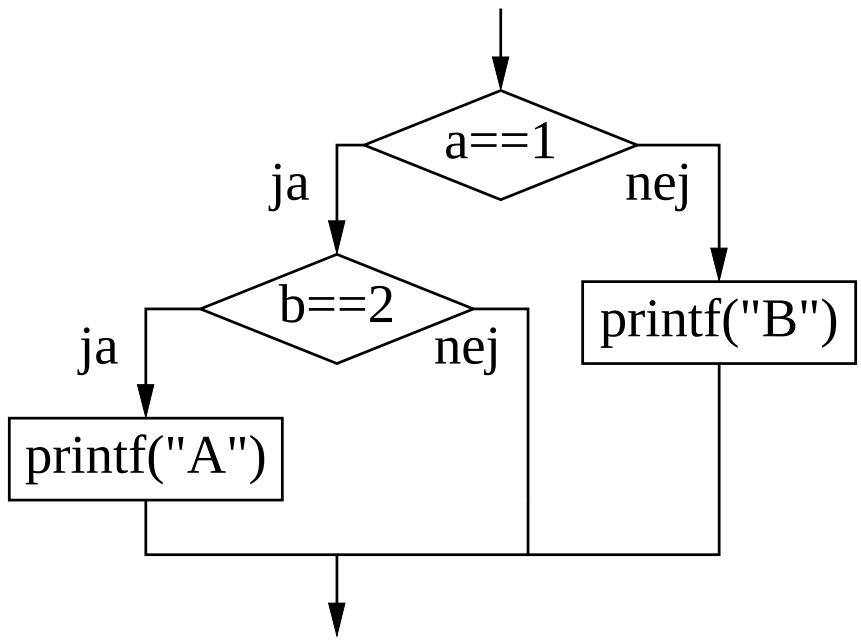
```c
#include <stdio.h>
int main(void){  /*blok2.c*/
  int a=5;
  printf("Før: a==%d\n",a);

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

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

  return 0;
}
```

"Dangling else"-problemet:

```
if (a == 1)
  if (b == 2)
    printf("A");
else
  printf("B");
```

```
if (a == 1) {
  if (b == 2) {
    printf("A");
  }
} else {
  printf("B");
}
```





- en else knytter sig altid til den inderste if
- brug kommandoblokke for at undgå tvivl!

Hvad hvis der er flere end to valgmuligheder? Brug switch !

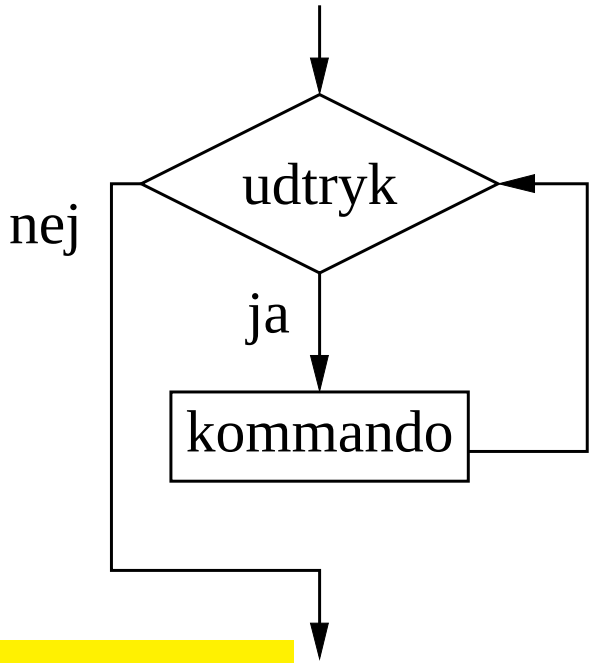
```
#include <stdio.h>

int main(void) {  /* switch.c */
  int a;
  char * dyr;
  printf("Giv mig et heltal!\n");
  scanf("%d", &a);

  switch (a) {
  case 1: dyr= "hest"; break;
  case 2: dyr= "gris"; break;
  case 3: dyr= "abe"; break;
  default: dyr= "ko"; break;
  }

  printf("\nDu er en %s!\n", dyr);
  return 0;
}
```

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

- først beregnes udtryk. Resultatet skal være et heltal eller noget der ligner (f.x. en `char`)
- udtryk == $\text{const}_i \Rightarrow \text{command}_i$ udføres. Herefter udføres command_{i+1} osv.
- udtryk != const_i for alle $i \Rightarrow$ default-kommandoen udføres, og herefter de efterfølgende! Hvis der ingen default er, gøres ingenting.
- man ønsker næsten altid at afslutte et case med en break-kommando; så springes de efterfølgende kommandoer over.

`while (udtryk) kommando;`

- først beregnes udtryk
- hvis udtryk er sandt, udføres kommando, og løkken startes forfra
- hvis udtryk er falsk, afsluttes løkken




```c
#include <stdio.h>
int main(void) {  /* while.c */
  int h = 0;
  while (h != 1234) {
    printf("Indtast det hemmelige heltal: ");
    scanf("%d", &h);
  }
  printf("\nHurra!\n");
  return 0;
}
```

```
for (init; condition; update) kommando;
```

(den mest generelle løkkekonstruktion i C)

1. først udføres init
2. så beregnes condition, og hvis den er falsk, afbrydes
3. kommando udføres
4. update udføres, og vi springer tilbage til trin 2.

```
#include <stdio.h>
int main(void) {  /* for.c */
  int i = 1;

  printf("%d elefant\n", i);
  for(i= 2; i<=10; i++) {
    printf("%d elefanter\n", i);
  }
  return 0;
}
```

- Udvælgelse
  - If
  - Switch
- Gentagelse
  - While
  - For
- Struktur
  - Komando blokke

Funktioner

- 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 næste 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:

```c
/* 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:

```c
#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

```
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.

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

- 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;
}
```

- 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.

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

- 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.
  - Dette kan “omgåes” ved brug af pointers

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

- 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.
  - Her er den værdi der kopieres adressen på et sted i hukommelsen

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:

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

$$1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

```
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` [argtest.c]

⇒ 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;
}
```

- `http://www.cplusplus.com/reference/clibrary/cstdlib/strtol/`

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
- 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]

- Boolean values
- `http://www.lysator.liu.se/c/c-faq/c-8.html`

```c
#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);
}
```

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		

- 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!

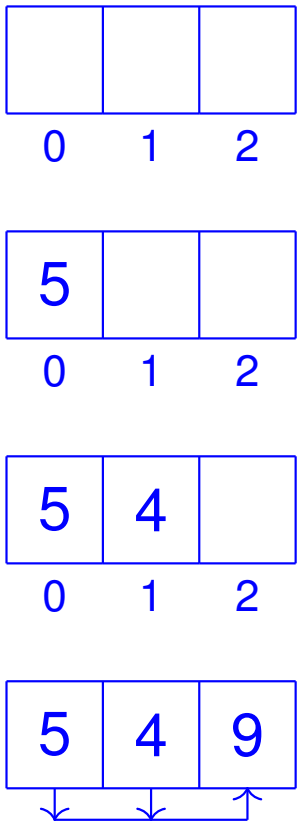
[conversions.c]

- 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[*N*]
- laveste indeks er 0, højeste er $N - 1$
- indgangene lagres *umiddelbart efter hinanden*

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

```c
#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 ...

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 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;
}
```

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.

⇒ 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.

$\Rightarrow$ 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.

Pointers

Husk: “En variabel er en navngiven plads i computerens lager.”

En pointer er en “pegepind” der *peger* på denne plads.

Husk: “En variabel er en navngiven plads i computerens lager.”

En pointer er en “pegepind” der *peger* på denne plads.

Declaring a pointer:

```
int* ptr_example; // Declares a pointer to an int.
```

Husk: “En variabel er en navngiven plads i computerens lager.”

En pointer er en “pegepind” der *peger* på denne plads.

Declaring a pointer:

```
int* ptr_example; // Declares a pointer to an int.
```

Getting the address of a variable:

```
int my_int = 3;
ptr_example = &my_int. // makes ptr_example point to
                       // the address of my_int.
```

Husk: “En variabel er en navngiven plads i computerens lager.”

En pointer er en “pegepind” der *peger* på denne plads.

Declaring a pointer:

```
int* ptr_example; // Declares a pointer to an int.
```

Getting the address of a variable:

```
int my_int = 3;
ptr_example = &my_int. // makes ptr_example point to
                       // the address of my_int.
```

Dereferencing:

```
*ptr_example = 2; // Sets the value of the data
                  // pointed to by ptr_example.
```

- &j betegner *adressen* af variablen j
- ∗pti betegner den *værdi*, som pti *peger på*

$\Rightarrow$ ∗&i er det samme som i (og &∗pti er det samme som pti)

- ∗ = dereference, & = reference

Eksempel:

```
int* ptr_example; // Declares a pointer to an int.
int *ptr_example; // Declares a pointer to an int.
int* ptr2, ptr3;

int main (void) {
    ptr3 = 5;
    ptr2 = 5; // gives warning
}
```

- `*s should be sticky.`

Eksempel:

```
int* ptr_example; // Declares a pointer to an int.
int *ptr_example; // Declares a pointer to an int.
int *ptr2, ptr3;

int main (void) {
    ptr3 = 5;
    ptr2 = (int*) 5; // using a cast
                     // still not a good idea.
}
```

- &j betegner *adressen* af variablen j
- $*$pti betegner den *værdi*, som pti *peger på*

$\Rightarrow$ $*$&i er det samme som i (og &$*$pti er det samme som pti)

- $*$ = dereference, & = reference

Problem: Funktioner i C kan ikke ændre på deres parametre (og give ændringer tilbage til hovedprogrammet) – værdiparametre.

Problem: Funktioner i C kan ikke ændre på deres parametre (og give ændringer tilbage til hovedprogrammet) – værdiparametre.

Løsning: Kald funktionen med pointers som parametre:

Problem: Funktioner i C kan ikke ændre på deres parametre (og give ændringer tilbage til hovedprogrammet) – værdiparametre.

Løsning: Kald funktionen med pointers som parametre:
Eksempel: en funktion der bytter om på to heltal:

```
void swap(int *x, int *y) {
  int tmp;
  tmp = *x;
  *x = *y;
  *y = tmp;
}
```

Bemærk at swap ikke laver om på de to pointers; kun på de *værdier* de peger på! [swap.c]

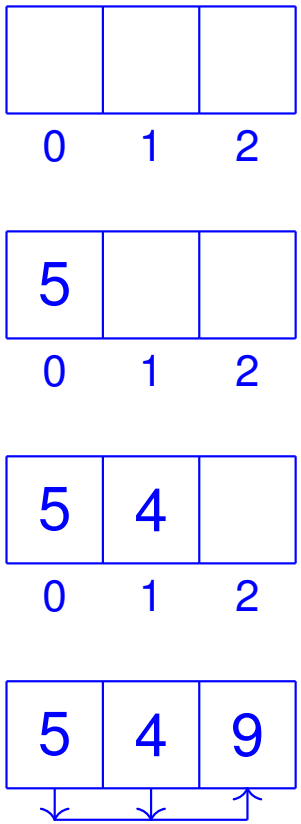
Eksempel: en funktion der bytter om på to heltal:

```
int main(void) {
  int a = 3, b = 7;

  printf("Before: %d %d\n", a, b);
  swap(&a, &b);
  printf("After:  %d %d\n", a, b);

  return 0;
}

void swap(int *x, int *y) {
  int tmp;
  tmp = *x;
  *x = *y;
  *y = tmp;
}
```

Arrays

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];
```



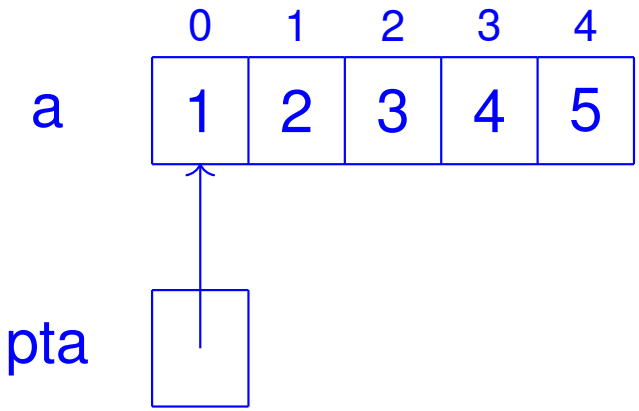
- et array skal deklareres med angivelse af *type*, og helst også *størrelse*: *type* a[*N*]
- laveste indeks er 0, højeste er $N - 1$
- indgangene lagres *umiddelbart efter hinanden*

$\Rightarrow$ &a[k] == &a[0] + k∗sizeof(*type*)

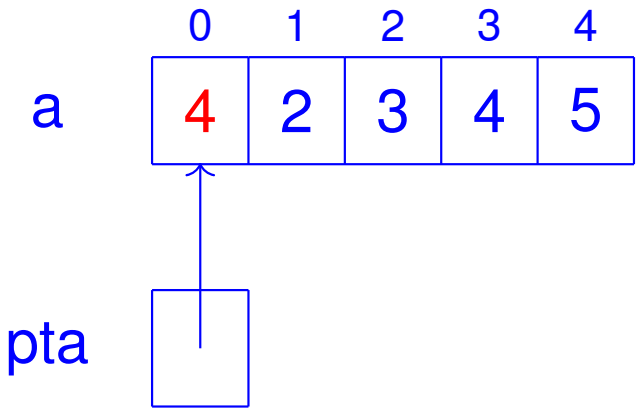
I C er et array det samme som en konstant pointer til dets første indgang:

```c
#include <stdio.h>

int main (void) { /* array-pt.c */
  int a[3], i;

  *a = 5;
  *(a + 1) = 4;
  *(a + 2) = *a + *(a + 1);

  for (i = 0; i < 3; i++) {
    printf( "%d: %d\n", i, a[i]);
  }

  return 0;
}
```

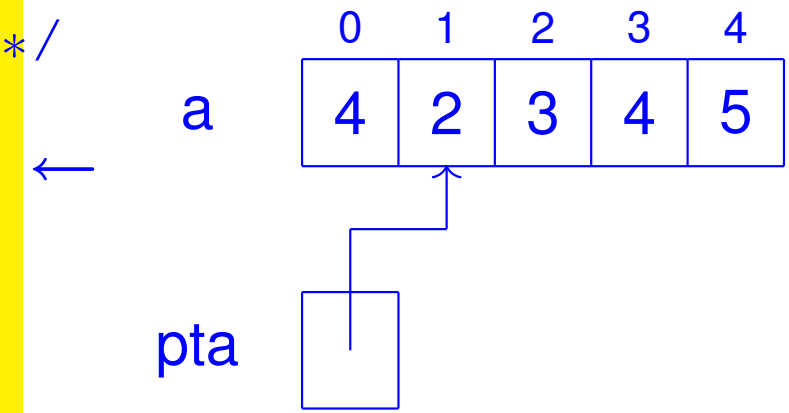
```c
#include <stdio.h>

/* array-pt-2.c */
int main( void) {
  int a[5]= {1, 2, 3, 4, 5};
  int *pta, i;

   pta = a; /* or, pta= &a[0]; */
  *pta = 4;
   pta++;
  *pta = *(pta - 1) * 2;
   pta += 3;
  (*pta)++;
  printf("index: %d\n", pta - a);

  for(i = 0; i < 5; i++) {
    printf("a[%d]: %d\n",i,a[i]);
  }
  return 0;
}
```



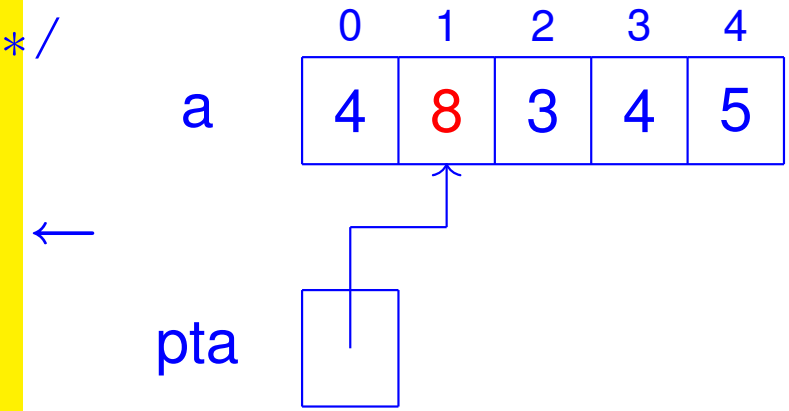
```c
#include <stdio.h>

/* array-pt-2.c */
int main( void) {
  int a[5]= {1, 2, 3, 4, 5};
  int *pta, i;

   pta = a; /* or, pta= &a[0]; */
  *pta = 4;
   pta++;
  *pta = *(pta - 1) * 2;
   pta += 3;
  (*pta)++;
  printf("index: %d\n", pta - a);

  for(i = 0; i < 5; i++) {
    printf("a[%d]: %d\n",i,a[i]);
  }
  return 0;
}
```



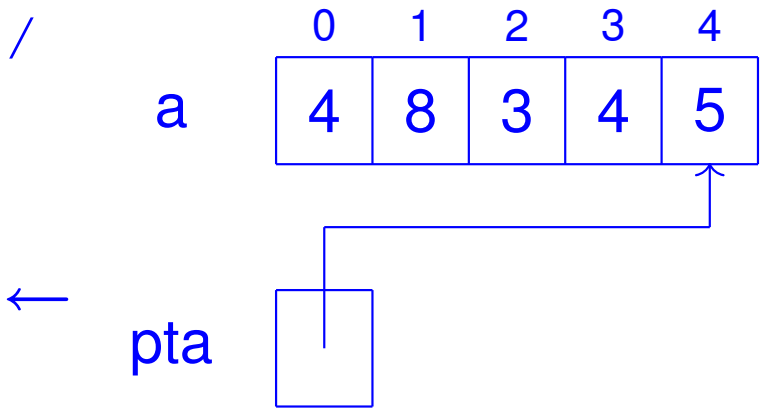
```c
#include <stdio.h>

/* array-pt-2.c */
int main( void) {
  int a[5]= {1, 2, 3, 4, 5};
  int *pta, i;

   pta = a; /* or, pta= &a[0]; */
  *pta = 4;
   pta++;
  *pta = *(pta - 1) * 2;
   pta += 3;
  (*pta)++;
  printf("index: %d\n", pta - a);

  for(i = 0; i < 5; i++) {
    printf("a[%d]: %d\n",i,a[i]);
  }
  return 0;
}
```



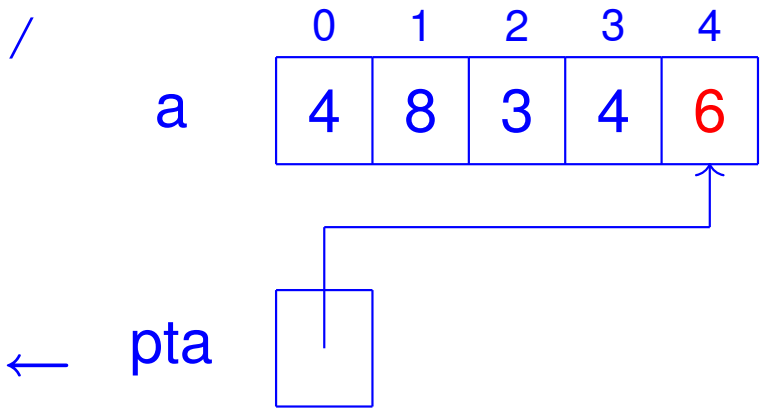
```c
#include <stdio.h>

/* array-pt-2.c */
int main( void) {
  int a[5]= {1, 2, 3, 4, 5};
  int *pta, i;

   pta = a; /* or, pta= &a[0]; */
  *pta = 4;
   pta++;
  *pta = *(pta - 1) * 2;
   pta += 3;
  (*pta)++;
  printf("index: %d\n", pta - a);

  for(i = 0; i < 5; i++) {
    printf("a[%d]: %d\n",i,a[i]);
  }
  return 0;
}
```



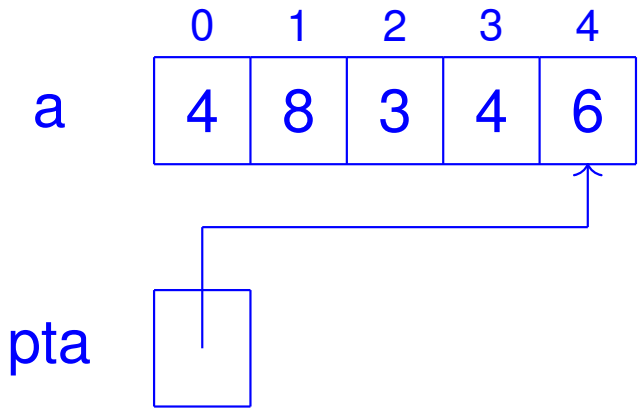
```c
#include <stdio.h>

/* array-pt-2.c */
int main( void) {
  int a[5]= {1, 2, 3, 4, 5};
  int *pta, i;

   pta = a; /* or, pta= &a[0]; */
  *pta = 4;
   pta++;
  *pta = *(pta - 1) * 2;
   pta += 3;
  (*pta)++;
  printf("index: %d\n", pta - a);

  for(i = 0; i < 5; i++) {
    printf("a[%d]: %d\n",i,a[i]);
  }
  return 0;
}
```



```c
#include <stdio.h>

/* array-pt-2.c */
int main( void) {
  int a[5]= {1, 2, 3, 4, 5};
  int *pta, i;

   pta = a; /* or, pta= &a[0]; */
  *pta = 4;
   pta++;
  *pta = *(pta - 1) * 2;
   pta += 3;
  (*pta)++;
  printf("index: %d\n", pta - a);

  for(i = 0; i < 5; i++) {
    printf("a[%d]: %d\n",i,a[i]);
  }
  return 0;
}
```



```c
#include <stdio.h>

/* array-pt-2.c */
int main( void) {
  int a[5]= {1, 2, 3, 4, 5};
  int *pta, i;

   pta = a; /* or, pta= &a[0]; */
  *pta = 4;
   pta++;
  *pta = *(pta - 1) * 2;
   pta += 3;
  (*pta)++;
  printf("index: %d\n", pta - a);

  for(i = 0; i < 5; i++) {
    printf("a[%d]: %d\n",i,a[i]);
  }
  return 0;
}
```



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

```c
#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 ...

Strenge

En streng i C er et *nulafsluttet* array af chars:

```
char s[]={ 'A', 'a', 'l', 'b', 'o', 'r', 'g', '\0' };
```

eller tilsvarende, en *pointer* til char:

```
char *s;
s= "Aalborg";
```

Følgende initialisering går også:

```
char s[]= "Aalborg";
```

Men som *assignment* er den gal:

```
char s[];
s= "Aalborg";
```

[streng-init.c]

Lav alle forekomster af 'a' om til 'i':

```c
#include <stdio.h>

int main( void) { /* abrakadabra.c */
  char s[]= "abrakadabra";  /* virker */
  /* char *s= "abrakadabra"; */ /* virker IKKE */
  char *p;

  printf("%s\n", s);
  p = s;
  while (*p != '\0') {
    if (*p == 'a') {
      *p= 'i';
    }
    p++;
  }

  printf("%s\n", s);
  return 0;
}
```

- en streng kan defineres som et array af char eller en pointer til char
- begge er *nulafsluttet*: sidste indgang er '\0' ("*sentinel*")
- i strenge der er defineret som et array, kan tegnene ændres
- i strenge der er defineret som en pointer, kan tegnene *ikke* ændres
- tegnet 'a' er forskellig fra strengen "a":
  'a'= 97 "a"=['a','\0']
- den tomme streng: ""=['\0']

Biblioteket `string.h` leverer funktioner til håndtering af strenge:

- **int** strcmp(**char** ∗s, **char** ∗t)
  sammenligner s og t i *leksikografisk* orden
  < 0: s kommer *før* t
  $= 0$: s er *lig med* t
  > 0: s kommer *efter* t
- **unsigned int** strlen(**char** ∗s)
  returnerer antallet af tegn i s (minus '\0')
- **char** ∗strcpy(**char** ∗s, **char** ∗t)
  kopierer t til s
  returnerer en pointer til s
  Pas på: Hvis der ikke er plads nok i s, går det galt!
- **char** ∗strcat(**char** ∗s, **char** ∗t)
  tilføjer t til slutningen af s
  returnerer en pointer til s
  samme kommentar som for strcpy
- *og en del flere*

[streng-eks.c]