



## Dagens tema

- Signaturer
- Typekonvertering
- Pekere og vektorer
- struct-er
- Definisjon av nye typenavn
- Lister
- Info om C

# Signaturer

I C gjelder alle deklarasjoner fra deklarasjonspunktet og ut filen.  
Følgende program:

```
int main (void)
{
  x = 4;
  return 0;
}

int x;
```

gir denne feilmeldingen:

```
> gcc gal-dekl.c -o gal-dekl
gal-dekl.c: In function 'main':
gal-dekl.c:3: error: 'x' undeclared (first use in this function)
gal-dekl.c:3: error: (Each undeclared identifier is reported only once
gal-dekl.c:3: error: for each function it appears in.)
gal-dekl.c: At top level:
gal-dekl.c:7: error: 'x' used prior to declaration
```

Hva gjør man da når man *må* referere til noe som ikke er deklarerert ennå, for eksempel når to funksjoner kaller hverandre?

Løsningen er å deklarere en *signatur* før selve deklarasjonen:

```
void f1 (int x);

int f2 (int a)
{
    if (a>0) f1(a);
    return a-1;
}

void f1 (int x)
{
    int w = f2(x/2);
}

int main (void)
{
    f1(5); return 0;
}
```

## En vanlig feil

På grunn Cs forhistorie er det ikke alltid nødvendig å deklarere signaturer for funksjoner, men C antar da at det dreier seg om en int-funksjon. Dette kan noen ganger gi rare feilmeldinger:

```
int main (void)
{
    f(6);
}

void f (int x)
{
    /* Gjør ett eller annet med x.*/
}
```

```
> gcc sig-feil.c
sig-feil.c:7: warning: type mismatch with previous implicit declaration
sig-feil.c:3: warning: previous implicit declaration of 'f'
sig-feil.c:7: warning: 'f' was previously implicitly declared to return 'int'
```

# Typekonvertering

I C (som i Java) kan man konvertere en verdi fra én type til en annen:

*(type)v*

Dette er aktuelt for

- heltall av ulike størrelser:

```
short x = 22;  
f((long)x);
```

- heltall til flyt-tall og omvendt:

```
double pi = 3.14159265;  
i = (int)pi;
```

**NB!** Heltall blir *trunkert*.

- pekere til ulike verdier:

```
int *p = &v;  
node *np = (node*)p;  
char *addr = (char*)0x12302;
```

# Pekere og vektorer

I C gjelder en litt uventet konvensjon:

- Bruk av et vektornavn gir en peker til element nr. 0:

```
int a[88];  
    ⋮  
a ≡ &a[0]
```

Når en vektor overføres som parameter, er det altså en peker til starten som overføres.

Følgende to funksjoner er derfor fullstendig ekvivalente:

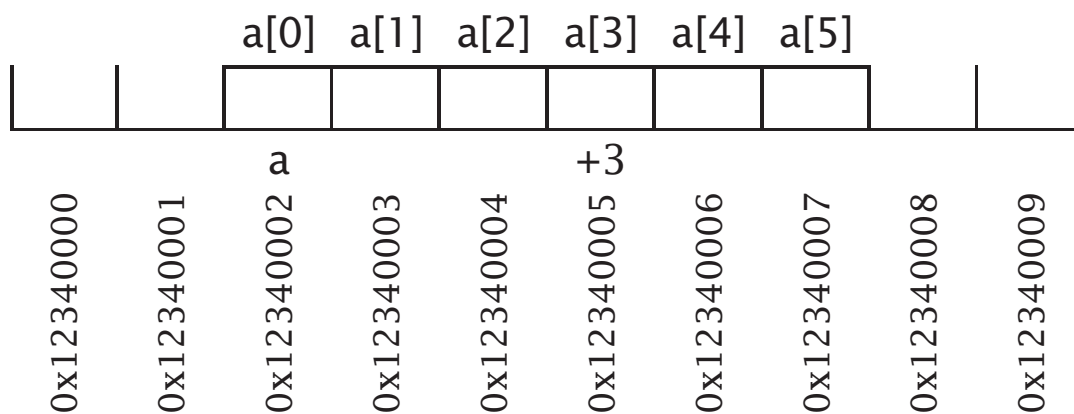
```
int strlena (char str[])  
{  
    int ix = 0;  
    while (str[ix] ++ix);  
    return ix;  
}  
  
int strlenb (char *str)  
{  
    char *p = str;  
    while (*p) ++p;  
    return p-str;  
}
```

Enda en uventet konvensjon:

- Aksess av vektorelementer kan også uttrykkes med pekere:

$$a[i] \equiv *(a+i)$$

Det er altså det samme om vi skriver  $a[3]$  eller  $*(a+3)$ .



# Regning med pekere

Dette er greit om a er en char-vektor, men hva om den er en long som trenger 4 byte til hvert element?

## Egne regneregler for pekere

C har egne regneregler for pekere:  $p+i$  betyr

«Øk  $p$  med  $i$  multiplisert med størrelsen av det  $p$  peker på.»

```
#include <stdio.h>

typedef unsigned long ul;

int main(void)
{
    char *cp = (char*)0x123400;
    long *lp = (long*)0x123400;

    cp++; lp++;
    printf("cp = 0x%lx\nlp = 0x%lx\n", (ul)cp, (ul)lp);
    return 0;
}
```

gir følgende når det kjøres:

```
cp = 0x123401
lp = 0x123404
```



## Pekere til pekere til ...

Noen ganger trenger man en peker til en pekervariabel, for eksempel fordi den skal overføres som parameter og endres. Siden vanlige pekere deklarereres som

```
xxx *p;
```

må en «peker til en peker» angis som

```
xxx **pp;
```

Dette kan utvides med så mange stjerner man ønsker.

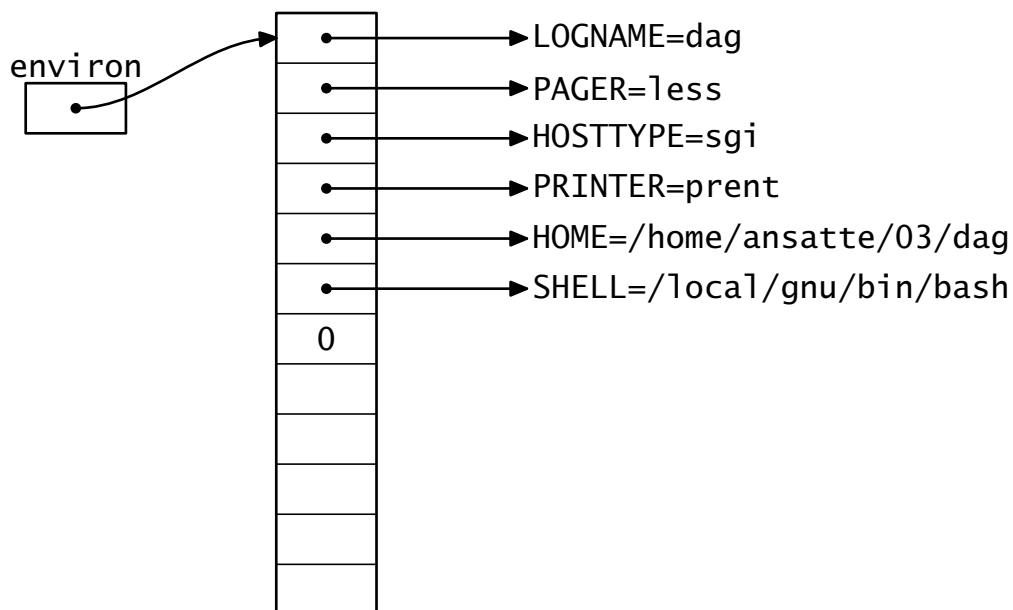
**Eksempel** Omgivelsesvariable i Unix inneholder opplysninger om en bruker og hans eller hennes preferanser:

```
LOGNAME=dag  
PAGER=less  
HOSTTYPE=sgi  
PRINTER=prent  
HOME=/home/ansatte/03/dag  
SHELL=/local/gnu/bin/bash
```

Omgivelsen overføres nesten alltid fra program til program ved en global variabel:

```
extern char **environ;
```

Pekeren environ peker på en vektor av pekere som hver peker på en omgivelsesvariabel og dens definisjon.



# Vanlige pekerfeil

Det er noen feil som går igjen:

- Glemme initiering av pekeren!

```
long *p;  
  
printf("Verdien er %d.\n", *p);
```

- Glemme frigjøring av objekt!

```
long *p;  
  
p = malloc(sizeof(long));  
p = NULL;
```

Det allokerte objektet vil nå være utilgjengelig, men vil «flyte rundt» og oppta plass så lenge programmet kjører. Dette kalles en **hukommelseslekkasje**.

- La en global peker peke på lokal variabel!

```
long *p;  
  
void f(void)  
{  
    long x;  
  
    p = &x;  
}  
  
f();
```

p peker nå på en variabel som ikke finnes mer. Stedet på stakken der x lå, kan være tatt i bruk av andre funksjoner.

- Peke på resirkulert objekt!

```
long *p, *q;  
  
p = q = malloc(sizeof(long));  
free(p); p = NULL;
```

q peker nå på et objekt som er frigjort og som kanskje er tatt i bruk gjennom nye kall på malloc.

## struct-er i C

I Simula og Java kan man sette sammen flere datatyper til en *klasse*. I C har man noe tilsvarende:

| Java  | C  |
|---|--|
| <pre>class A {<br/>  int a, b, c;<br/>  float f;<br/>  char ch;<br/>}</pre> | <pre>struct a {<br/>  int a, b, c;<br/>  float f;<br/>  unsigned char ch;<br/>};</pre> |

Cs struct-er er rene datastrukturer; der kan man *ikke* ha metoder.

## Deklarasjon av struct-variable

Struct-variable deklarerer som andre variable:

```
struct a astr;
```

Følgende skiller slike deklarasjoner fra de tilsvarende i Simula og Java:

- Struct-ens navn består av *to ord*: struct (som alltid skal være der) og a (som programmereren har funnet på).
- Man trenger ikke opprette noe objekt med new.

## Bruk av struct-variable

Struct-variable brukes ellers som i Simula og Java:

```
astr.b = astr.c + 2;  
if (astr.f < 0.0) astr.ch = 'x';
```

## Typedefinisjoner

For å unngå lange typenavn kan vi gi dem navn:

```
typedef unsigned long ul;  
typedef struct a str_a;
```

Nå `ul` og `str_a` brukes i deklarasjoner på lik linje med `int`, `char` etc.

## Pekere til struct-er

Vi kan selvfølgelig peke på struct-variable:

```
struct a *pa = malloc(sizeof(struct a));  
  
(*pa).f = 3.14;
```

Legg merke til at vi trenger parentesene rundt pekervariabelen fordi `*pa.f` tolkes som `*(pa.f)`.

Fordi vi så ofte trenger pekere til struct-objekter, er det innført en egen notasjon for dette:

```
pa->f = 3.14;
```

# Lister

- Enkle lister
- Operasjoner på lister

## Fordelene med lister:

- Dynamiske; plassforbruket tilpasses under kjøringen.
- Fleksible; innenbyrdes rekkefølge kan lett endres.
- Generelle; kan simulere andre strukturer.

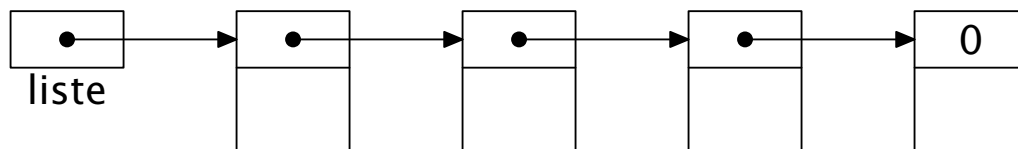
## Ulemper med lister

- Det kan lett bli en del leting, så lange lister kan være langsomme i bruk.



## En enkel liste

```
struct elem {  
    struct elem *neste;  
    ... diverse data ...  
};  
struct elem *liste;
```



**Listepekeren** liste peker på første element. Denne listen kan simulere

- Stakker
- Køer
- Prioritetskøer

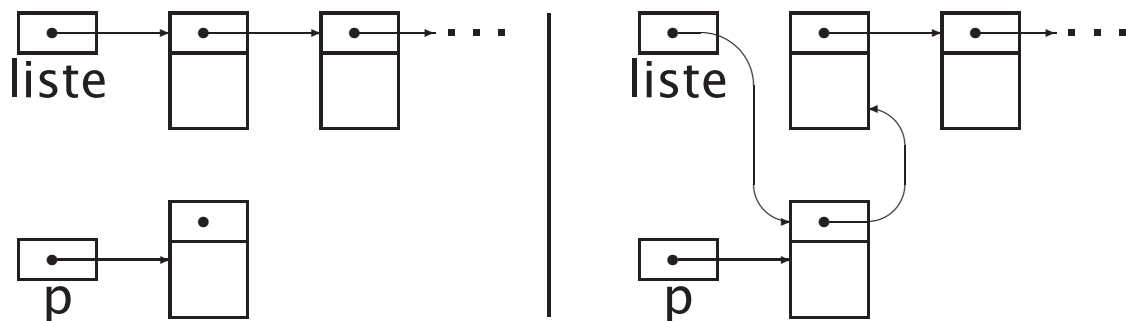
### Peker til «ingenting»

I C er konvensjonen at adressen 0 er en peker til «ingenting». I mange definisjonsfiler (som `stdio.h`) er `NULL` definert som 0.

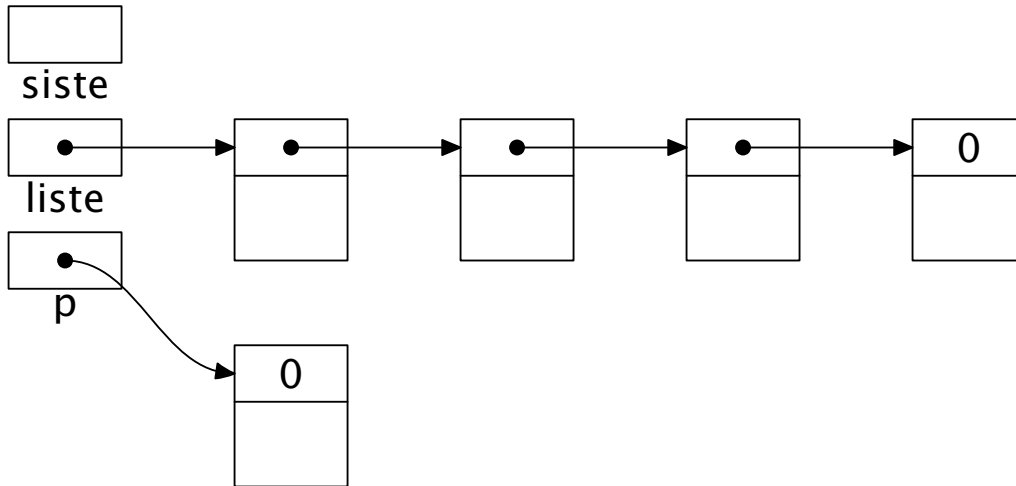
## Operasjoner på lister

### Innsetting først i listen

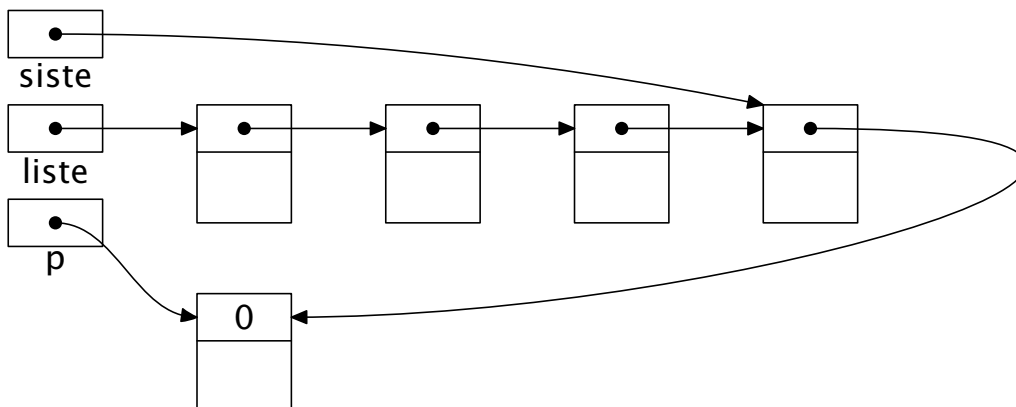
```
p->neste = liste;  
liste = p;
```



## Innsetting sist i listen



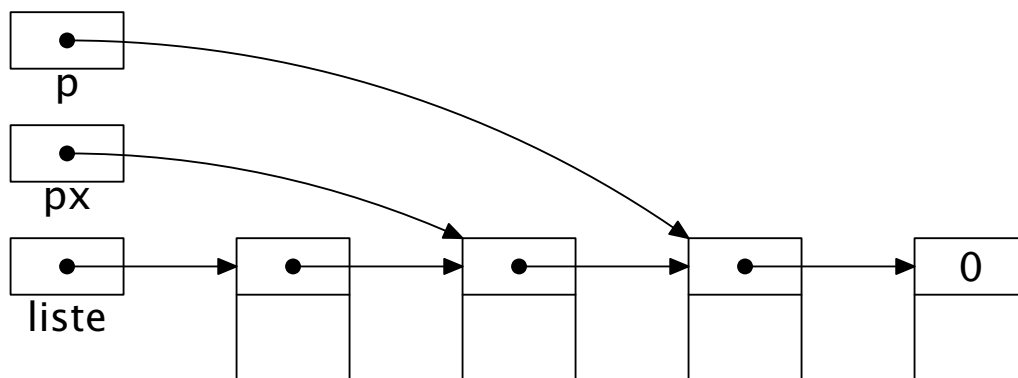
```
if (liste == NULL) {  
    liste = p;  
} else {  
    siste = liste; /* Finn siste element. */  
    while (siste->neste) siste = siste->neste;  
    siste->neste = p;  
}  
p->neste = NULL;
```



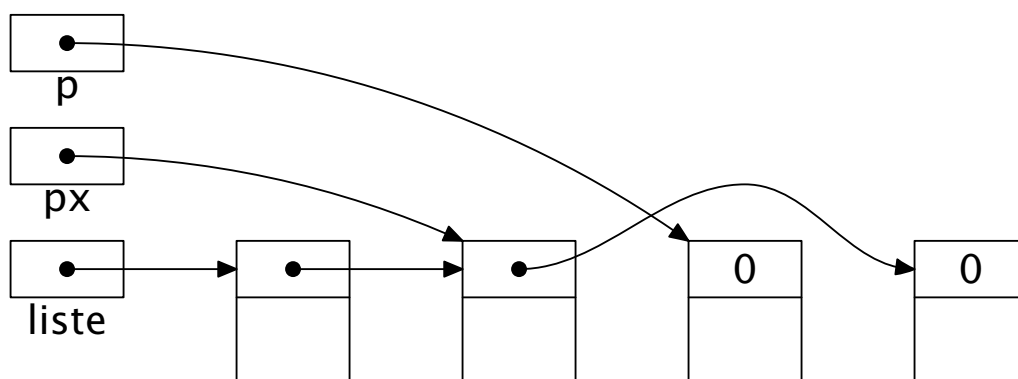
Dette kan gjøres raskere hvis vi alltid har en peker til siste element.

## Fjerning av element

Vi antar at  $p$  skal fjernes fra listen, og at  $px$  peker på  $p$ s forgjenger.



```
px->neste = p->neste;  
p->neste = NULL;
```



# Informasjon om C

Den viktigste kilden til informasjon om C (utenom en god oppslagsbok) er programmet `man`. Det dokumenterer alle C-funksjonene.

```
> man sqrt
SQRT(3)      Linux Programmer's Manual      SQRT(3)

NAME
  sqrt - square root function
SYNOPSIS
  #include <math.h>
  double sqrt(double x);
  float sqrtf(float x);
  long double sqrtl(long double x);
DESCRIPTION
  The sqrt() function returns the non-negative square root of x. It
  fails and sets errno to EDOM, if x is negative.
ERRORS
  EDOM  x is negative.
SEE ALSO
  hypot(3)
```

## Kapitler i man

Man-filene er organisert i kapitler; vi er mest interessert i

- ❶ Kommandoer (som gcc)
- ❷ Unix-operasjoner (som kill)
- ❸ C-funksjoner (som printf)

Hvis navnet finnes i flere kapitler, får vi ikke alltid det vi ønsker.

```
> man exit
```

```
BASH_BUILTINS(1)
```

```
BASH_BUILTINS(1)
```

```
NAME
```

```
bash, :, ., [, alias, bg, bind, break, builtin, cd, command, compgen,  
complete, continue, declare, dirs, disown, echo, enable, eval, exec,  
exit, export, fc, fg, getopts, hash, help, history, jobs, kill, let,  
local, logout, popd, printf, pushd, pwd, read, readonly, return, set,  
shift, shopt, source, suspend, test, times, trap, type, typeset,  
ulimit, umask, unalias, unset, wait - bash built-in commands, see  
bash(1)
```

```
⋮
```

Bedre er å angi kapittelet eksplisitt:

```
> man 3 exit
```

```
EXIT(3)
```

```
Linux Programmer's Manual
```

```
EXIT(3)
```

```
NAME
```

```
exit - cause normal program termination
```

```
SYNOPSIS
```

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

```
void exit(int status);
```

```
DESCRIPTION
```

```
The exit() function causes normal program termination and the the value of status & 0377 is returned to the parent (see wait(2)). All functions registered with atexit() and on_exit() are called in the reverse order of their registration, and all open streams are flushed and closed. Files created by tmpfile() are removed.
```

```
⋮
```