

# Programmeringsspråket C

## Del 3

Michael Welzl

E-mail: [michawe@ifi.uio.no](mailto:michawe@ifi.uio.no)

# Dynamisk allokering

Ofte trenger man å opprette objekter under kjøringen i tillegg til variablene. Standardfunksjonen malloc (“memory allocate”) benyttes til dette. Parameter er antall byte den skal opprette; operatoren sizeof kan gi oss dette.

Vi må ha med stdlib.h for at malloc kan brukes.

```
#include <stdlib.h>
:
int *p;
:
p = malloc(sizeof(int));
```

## Frigivelse av objekter:

Når objekter ikke trengs mer, må de gis tilbake til systemet med funksjonen free:

```
free(p);
```

# Et eksempel

Anta at vi skal lese et navn (dvs. en tekst) og skrive det ut. For at navnet ikke skal oppta plass når vi ikke trenger det, bruker vi dynamisk allokering.

```
char *navn;  
  
:  
printf("Hva heter du? ");  
navn = malloc(200);  
scanf("%s", navn);  
printf("Hei på deg, %s.\n", navn);  
free(navn);
```

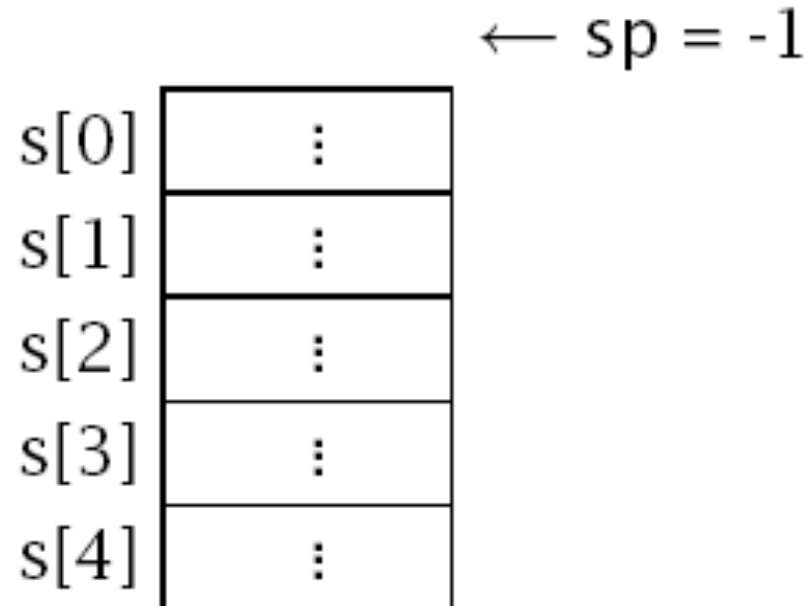
# Lagring av data

Det er fire hovedteknikker for å lagre data:

- **Vektor** (matrise, “array”) benyttes når det er naturlig å gi hvert element et identifikasjonsnummer.  
Eksempel: Opptelling av forekomster av tegn i en fil.
- **Stakk** (“stack”) er til mellomlagring når  *yngste* element skal hentes først.  
(LIFO = “Last in, first out”)  
Eksempel: Snu sifrene i et heltall ved utskrift.
- **Kø** (“queue”) er til mellomlagring når  *eldste* element skal hentes først.  
(FIFO = “First in, first out”)  
Eksempel: Kø av utskrifter til en skriver.
- **Liste** (“list”) er mest fleksibel; den omtales senere.

# Stakker

Man bruker en peker til å peke på øverste element. Anta at vi bruker  $sp$  som stakkpeker og at stakken er tom.



Man kan nå legge en verdi på stakken (“**push-e**” en verdi) ved å utføre:

```
s[++sp] = verdi;
```

Status er da:

s[0]	<i>verdi</i>	← sp = 0
s[1]	⋮	
s[2]	⋮	
s[3]	⋮	
s[4]	⋮	

Etter å ha lagt totalt fire ulike verdier på stakken, har vi denne situasjonen:

s[0]	18	
s[1]	31	
s[2]	5	
s[3]	12	← sp = 3
s[4]	⋮	

Verdien øverst på stakken kan hentes tilbake (“**pop-es**”) med:

```
v = s[sp--];
```

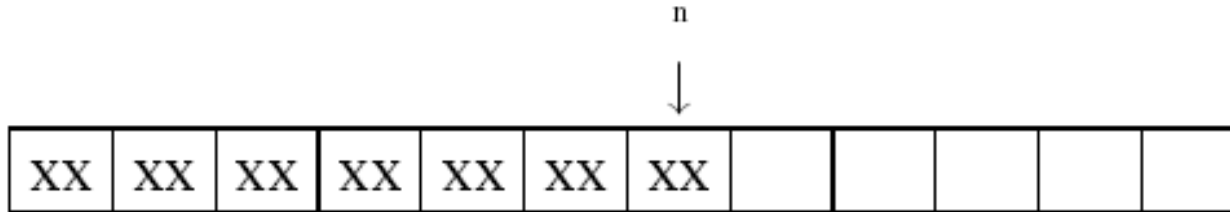
Gjør vi dette to ganger, er status:

s[0]	18	
s[1]	31	← sp = 1
s[2]	5	
s[3]	12	
s[4]	⋮	

Nå ligger det to tall på stakken: 31 (øverst) og 18. Selv om de andre verdiene stadig ligger i minnet, er de logisk sett ikke lenger på stakken.

# Køer

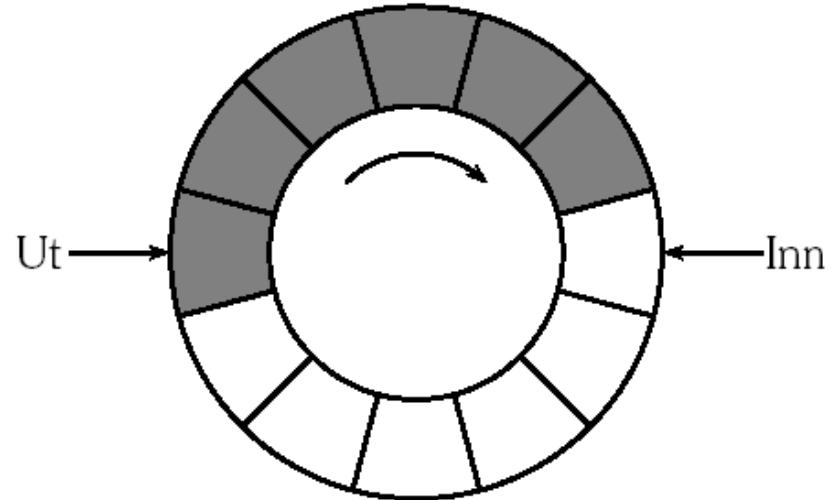
Man *kunne* ha implementert køer slik:



## Ringbufferer

Det er bedre å bruke **ringbufferer**.

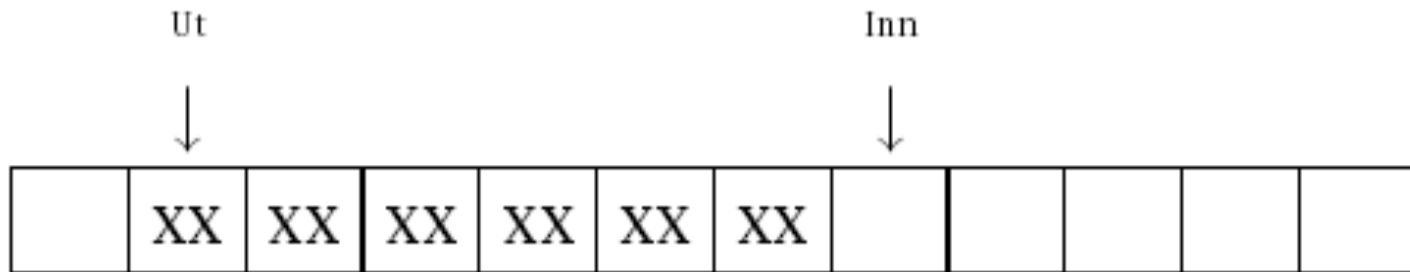
Pekerne forteller hvor neste element skal inn, og hvor man kan finne neste element som skal ut.





# Implementasjon:

En ringbuffer implementeres ved å “klippe den opp” og lagre den i en vektor:



## Deklarasjon i C

```
#define BUFFSIZ 12  
  
int ring[BUFFSIZ],  
    ring_in = 0,  
    ring_out = 0,  
    ring_n = 0;
```

# Innsetting og henting fra ringbufferet:

## Innsetting i ringbuffer

```
void put(int x)
{
    if (ring_n == BUFSIZ) {
        printf("Ringbuffer overflow!\n"); exit(1);
    }

    ring[ring_in] = x; ++ring_n; ++ring_in;
    if (ring_in >= BUFSIZ) ring_in = 0;
}
```

## Henting fra ringbuffer

```
int get(void)
{
    int x;

    if (ring_n == 0) {
        printf("Ringbuffer underflow!\n"); exit(1);
    }

    x = ring[ring_out]; --ring_n; ++ring_out;
    if (ring_out >= BUFSIZE) ring_out = 0;
    return x;
}
```

# Et eksempel:

Det finnes et nyttig Unix-program med navn tail som skriver ut de ti siste linjene i en fil.

```
> tail
Første linje
Siste linje
⟨Control-D⟩
Første linje
Siste linje
> tail <tail10.c
}

/* Utskrift: */
while (ring_n > 0) {
    /* Skriv ut og fjern et element: */
    printf("%s", ring[ring_out]);
    free(ring[ring_out]); --ring_n; ++ring_out;
    if (ring_out >= 10) ring_out = 0;
}
}
```

```

#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    /* Ringbufferen */
    char *ring[10];
    int ring_in = 0, ring_out = 0, ring_n = 0;

    /* Andre variable */
    char *cur_line;

    /* Innlesning */
    while (1) {
        cur_line = malloc(200);
        if (fgets(cur_line, 200, stdin) == NULL) {
            free(cur_line); break;
        }

        if (ring_n == 10) {
            /* Fjern et element fra ringbufferen: */
            free(ring[ring_out]); --ring_n; ++ring_out;
            if (ring_out >= 10) ring_out = 0;
        }

        /* Sett inn sist leste linje: */
        ring[ring_in] = cur_line; ++ring_n; ++ring_in;
        if (ring_in >= 10) ring_in = 0;
    }

    /* Utskrift: */
    while (ring_n > 0) {
        /* Skriv ut og fjern et element: */
        printf("%s", ring[ring_out]);
        free(ring[ring_out]); --ring_n; ++ring_out;
        if (ring_out >= 10) ring_out = 0;
    }
}

```

# Noe forklaring:

Følgende C-elementer har ikke vært nevnt før:

- Setningen `break` hopper ut av nærmeste løkke (eller `switch`-setning).

Løkken

```
while(1) { . . . }
```

vil dermed gjentas inntil det utføres en `break` inni den.

- Funksjonen `fgets` leser en hel linje; parametrene er:
  - 1) en peker til en `char`-vektor med plass til linjen,
  - 2) antall tegn det er plass til i nevnte vektor og
  - 3) filen (`stdin` er standard inn-fil — dvs tastaturet).

# Pekere og vektorer

I C gjelder en litt uventet konvensjon:

Vektornavn kan brukes som en peker til element nr. 0:

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

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

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;
}
```

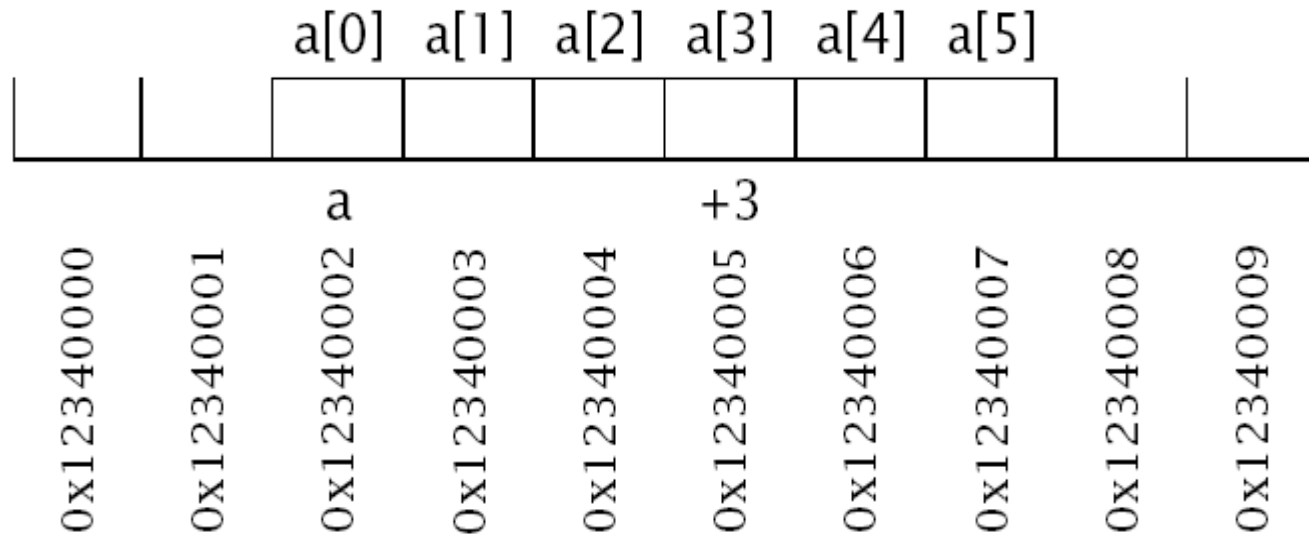
# Nok en konvensjon:

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.”

# Programmet:

```
#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 resultat 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:

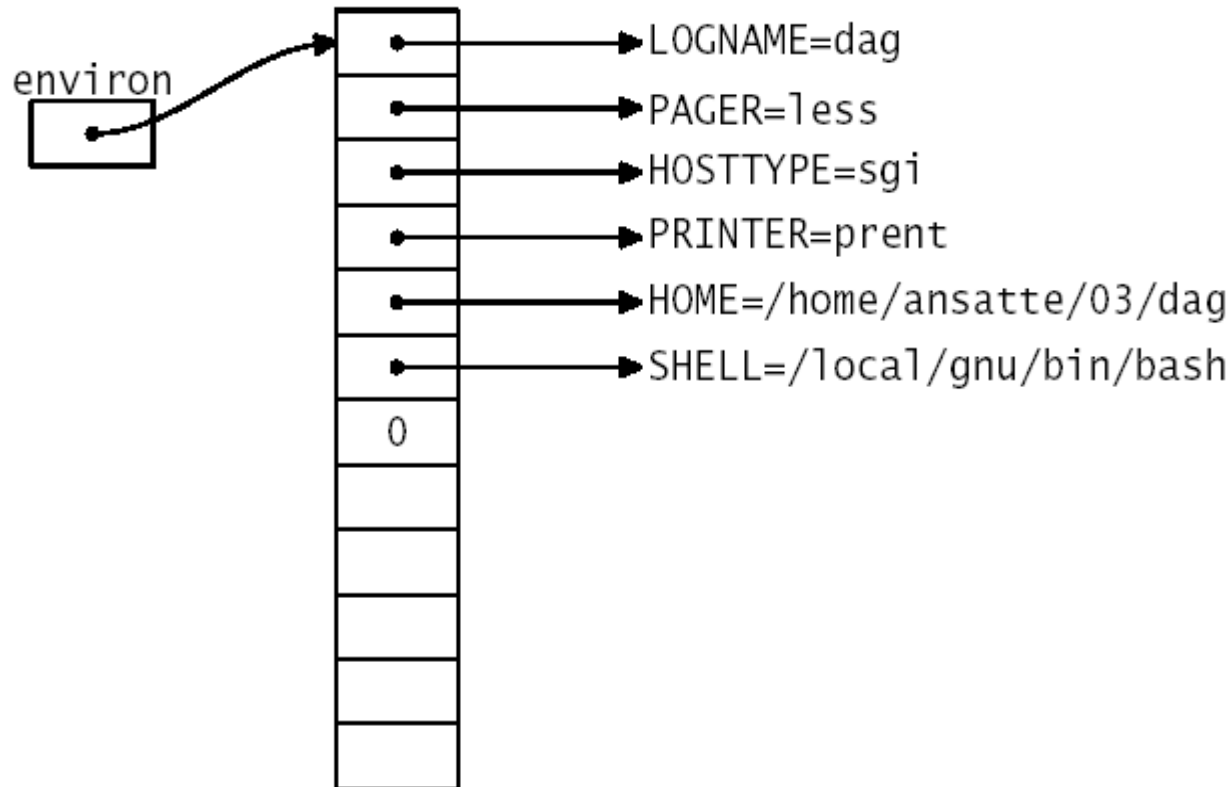
```
LOGNAME=dag  
PAGER=less  
HOSTTYPE=sgi  
PRINTER=prent  
HOME=/home/ansatte/03/dag  
SHELL=/local/gnu/bin/bash
```

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

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

# Eksempel (forts.)

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 %ld.\n", *p);
```

- Glemme frigjøring av objekt!

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

Det allokerede objektet vil nå være utilgjengelig, men vil “flyte rundt” og oppta plass så lenge programmet kjører. Dette kalles en **hukommelseslekkasje**.

# Vanlige pekerfeil (forts.)

- 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.

# Enda en vanlig pekerfeil:

- 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 {   int a, b, c;   float f;   char ch; }</pre>	<pre>struct a {   int a, b, c;   float f;   unsigned char ch; };</pre>

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

# Deklarasjon og 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';
```

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 deklarerer som andre variable:

```
struct a astr;
```

# Typedefinisjoner

For å unngå lange typenavn kan vi gi dem navn:

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

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

# Pekere til struct-er

## 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:**

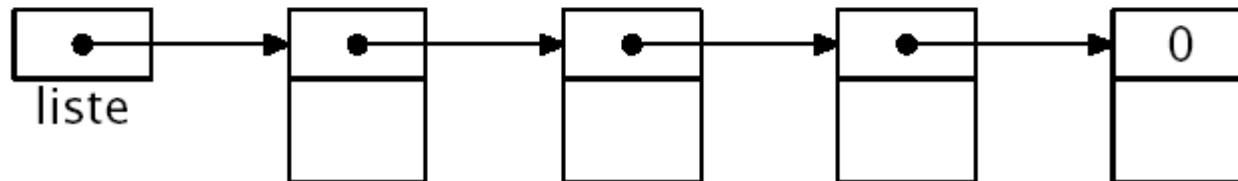
- Dynamiske; plassforbruk tilpasses under kjøringen.
- Fleksible; innbyrdes rekkefølge kan lett endres.
- Generelle; kan simulere andre strukturer.

## **Ulemper med lister:**

- Det kan lett bli en del leting, slik at 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

# Konvensjon:

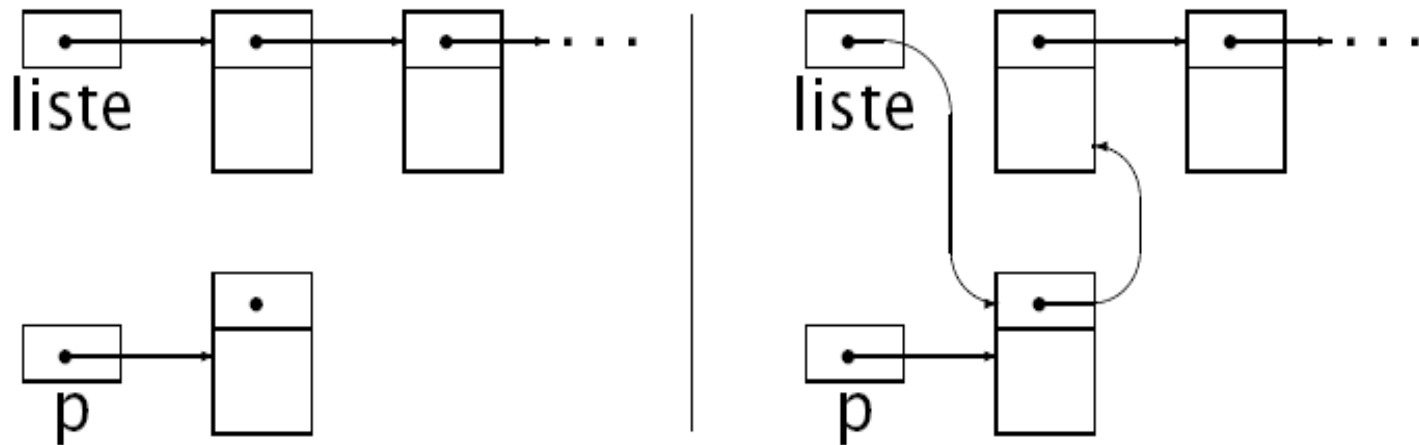
## **Peker til «ingenting»**

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

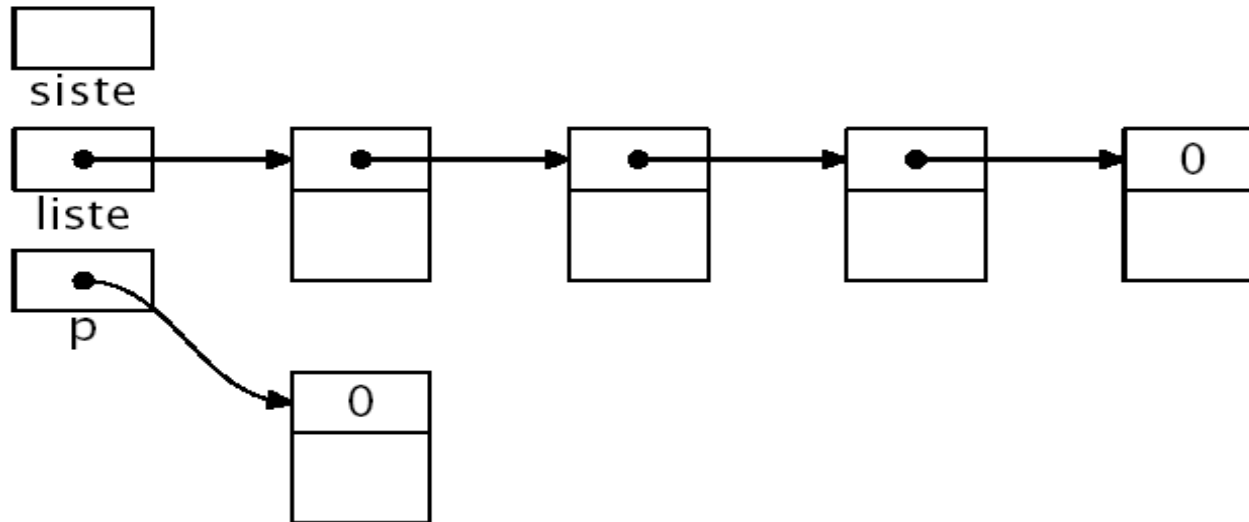
```
p->neste = liste;  
liste = p;
```





# Operasjoner på lister (forts.)

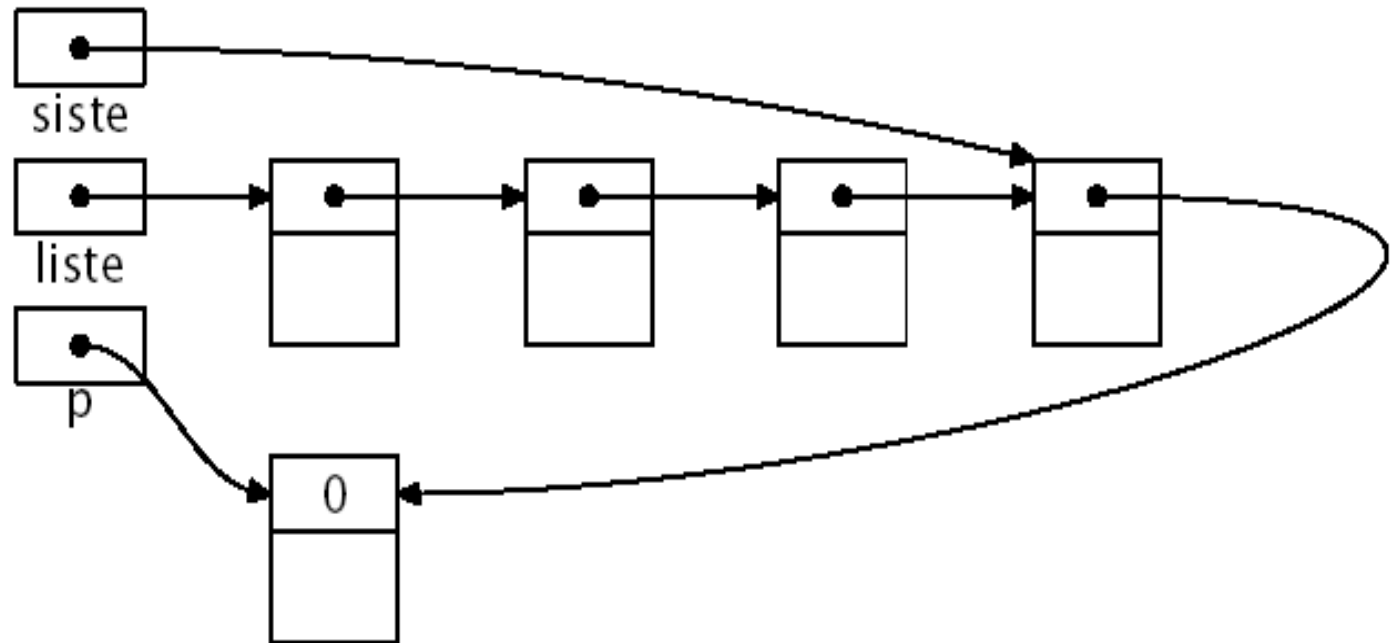
## Innsetting sist i listen



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

# Innsetting sist i listen (forts.):

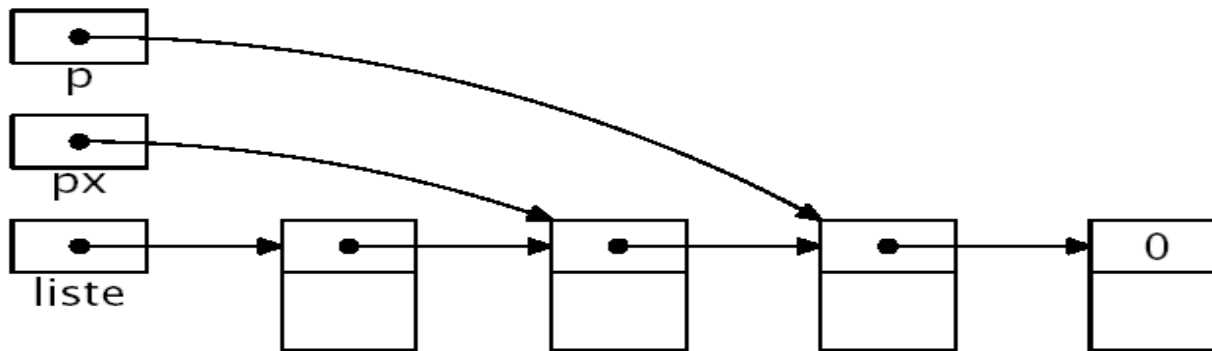
Dette kan gjøres raskere hvis vi alltid har en peker til siste element.



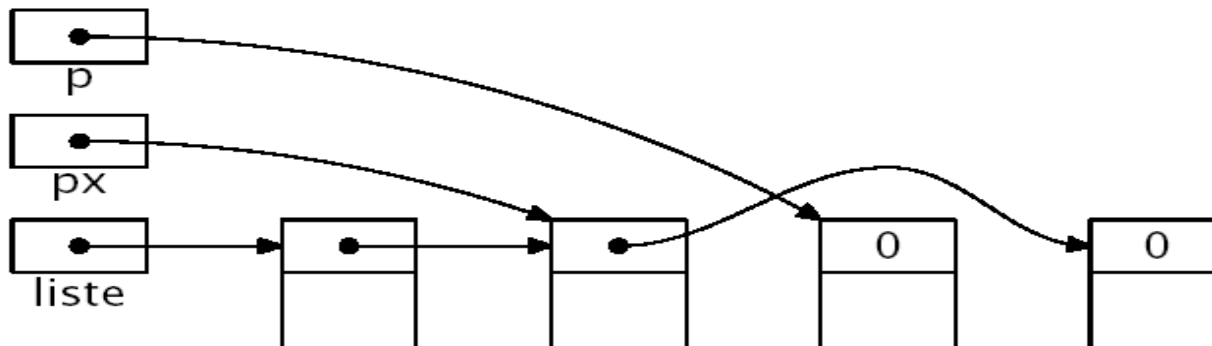
# Operasjoner på lister (forts.)

## 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;
```



# Debuggere

En "debugger" er et meget nyttig feilsøkings-verktøy. Det kan:

- analysere en program-dump,
- vise innhold av variable,
- vise hvilke funksjoner som er kalt,
- kjøre programmet én og én linje, og
- kjøre til angitte stopp-punkter.

Debuggeren gdb er laget for å brukes sammen med både cc og gcc. Den har et vindusgrensesnitt som heter ddd, som kan brukes på UNIX-maskiner.

# Debuggere (forts.)

For å bruke gdb/ddd må vi gjøre to ting:

- kompilere våre programmer med opsjonen `-g`,  
og
- angi at vi ønsker programdumper:

```
ulimit -c unlimited
```

hvis vi bruker bash. (Da må vi huske å fjerne programdumpfilene selv; de er *store!*)

# Et program med feil:

Følgende program prøver å:

1. Sette opp en vektor med 10 pekere til heltall,
2. Sette inn tallene 0-9 og
3. Skrive ut tallene

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

int *vec[10];

int main(void)
{
    int i;

    for (i = 0; i<10; ++i) {
        vec[i] = (int*)malloc(sizeof(int));
        *vec[i] = i;
    }

    for (i = 9; i>=0; --i) {
        printf("vec[%d] peker på %d.\n", i, *vec[i]);
    }
    return 0;
}
```

# Programdumper

Når et program dør på grunn av en feil («aborterer»), prøver det ofte å skrive innholdet av hele prosessen<sup>†</sup> på en fil slik at det kan analyseres siden.

- ❶ Programmet kompiles med debuggingsinformasjon:

```
> gcc -g test-gdb.c -o test-gdb
```

- ❷ Programmet kjøres:

```
> test-gdb  
Segmentation Fault (core dumped)  
> ls -l core*  
-rw----- 1 dag dag 137820 Sep 22 9:28 core.13838
```

<sup>†</sup> Dette kalles ofte en «core-dump» siden datamaskinene for 20-40 år siden hadde hurtiglager bygget opp av ringer med kjerne av feritt. I Unix heter denne filen derfor core.

>/usr/bin ddd test-gdb&

The screenshot shows the DDD (Data Display Debugger) interface. The title bar reads "DDD: /home/ansatte/03/dag/INI147/Forelesninger/Kode/test-gdb.c". The menu bar includes "File", "Edit", "View", "Program", "Commands", "Status", "Source", "Data", and "Help". The main window is a grid for source code. Below the grid is a command line with "0: |" and buttons for "Display \*()", "Show \*()", "Set \*()", "New Display", "Set \*()", and "Delete \*()". The source code is as follows:

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

int *vec[10];

int main(void)
{
    int i;
    for (i = 0; i < 10; ++i) {
        vec[i] = (int*)malloc(sizeof(int));
        *vec[i] = i;
    }
    for (i = 9; i >= 0; --i) {
        printf("vec[%d] peker på %d.\n", i, *vec[i]);
    }
    return 0;
}
```

To the right of the code is a control panel titled "DDD" with buttons: "Run", "Interrupt", "Step", "Stepi", "Next", "Nexti", "Cont", "Finish", "Up", "Down", "Back", "Fwd", "Edit", and "Kill". Below the code is another command line with "0: |main" and buttons for "Lookup \*()", "Break at \*()", "Print \*()", "Display \*()", "Find<< \*()", and "Find>> \*()". At the bottom, there is a text area with the following text:

```
DDD 2.2.3 (mips-sgi-irix5.3), by Dorothea Lütkehaus and Andreas Zeller.
Copyright © 1998 Technische Universität Braunschweig, Germany.
(gdb) |
```

At the very bottom, a status bar says "Welcome to DDD 2.2.3!".



# I File-menyen finner vi «Open Core Dump».

The screenshot shows the DDD (Data Display Debugger) interface. The title bar indicates the file path: `DDD: /home/ansatte/03/dag/IN147/Forelesninger/Kode/test-gdb.c`. The menu bar includes `File`, `Edit`, `View`, `Program`, `Commands`, `Status`, `Source`, and `Data`. The main window displays the source code of `test-gdb.c` with a red arrow pointing to the line `vec[1] = (int*)malloc(sizeof(int));`. The code is as follows:

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

int *vec[10];

int main(void)
{
    int i;

    for (i = 0; i < 10; ++i) {
        vec[i] = (int*)malloc(sizeof(int));
        *vec[i] = i;
    }

    for (i = 9; i >= 0; --i) {
        printf("vec[%d] peker på %d.\n", i, *vec[i]);
    }
    return 0;
}
```

Below the source code, the status bar shows `0: |` and buttons for `Display *G`, `Show G`, `Notate ()`, `New Display`, `Set G`, and `Delete ()`. To the right of the source code is a control panel with buttons: `Run`, `Interrupt`, `Step`, `Stepi`, `Next`, `Nexti`, `Cont`, `Finish`, `Up`, `Down`, `Back`, `Fwd`, `Edit`, and `Kill`.

At the bottom, the status bar shows `0: |main` and buttons for `Lookup ()`, `Break at ()`, `Print ()`, `Display ()`, `Find<< ()`, and `Find>> ()`. The output window displays the following text:

```
DDD 2.2.3 (mips-sgi-irix5.3), by Dorothea Lutkehaus and Andreas Zeller.
Copyright © 1998 Technische Universität Braunschweig, Germany.
(gdb) core-file /home/ansatte/03/dag/IN147/Forelesninger/Kode/core
Core was generated by 'test-gdb'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /usr/lib/libc.so.1... done.
#0  0x400abc in main () at test-gdb.c:12
(gdb) |
```

At the very bottom, a small message reads: `Core was generated by 'test-gdb'.`

Nå vet vi at feilen oppsto på linje 12 i forbindelse med `*vec[i] = i`. Kanskje det er noe galt med indeksen `i`?

I Data-menyen finner vi «Display Local Variables»

Locals

1 = 0

0: | Display \*() Hide () Rotate () New Display Set () Delete ()

```

#include <stdio.h>
#include <stdlib.h>

int *vec[10];

int main(void)
{
    int i;
    for (i = 0; i<10; ++i) {
        vec[i] = (int*)malloc(sizeof(int));
        *vec[i] = i;
    }
    for (i = 9; i>=0; --i) {
        printf("vec[%d] peker på %d.\n", i, *vec[i]);
    }
    return 0;
}

```

DDD	
Run	
Interrupt	
Step	Stepi
Next	Nexti
Cont	Finish
Up	Down
Back	Fwd
Edit	Kill

0: ↩ locals | Lookup () Break at () Print () Display () Find&lt;&lt; () Find&gt;&gt; ()

```

DDD 2.2.3 (mips-sgi-irix5.3), by Dorothea Lütkehaus and Andreas Zeller.
Copyright © 1998 Technische Universität Braunschweig, Germany.
(gdb) core-file /home/ansatte/03/dag/IN147/Forelesninger/Kode/core
Core was generated by 'test-gdb'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /usr/lib/libc.so.1...done.
#0 0x400abc in main () at test-gdb.c:12
(gdb) graph display 'info locals'
(gdb) |

```

Variabelen  $i$  er 0, så den er OK. Hva da med `vec`? Vi kan klikke på en forekomst av `vec` og så «Display». (Alternativt kan vi bare peke på en `vec` uten å klikke.)

DDD: /home/ansatte/03/dag/IN147/Forelesninger/Kode/test-gdb.c

File Edit View Program Commands Status Source Data Help

Locals

i = 0

1: vec

0x0
0x10002010
0x0
0x0
0x0
0x0
0x0
0x0
0x0
0x0
0x0

0: | Display G Hide O Rotate O New Display Set O Delete O

```

#include <stdio.h>
#include <stdlib.h>

int *vec[10];

int main(void)
{
    int i;
    for (i = 0; i<10; ++i) {
        vec[i] = (int*)malloc(sizeof(int));
        *vec[i] = i;
    }
    for (i = 9; i>=0; --i) {
        printf("vec[%d] peker på %d.\n", i, *vec[i]);
    }
    return 0;
}

```

0: vec | Lookup () Break at () Print () Display () Find<< () Find>> ()

Copyright © 1998 Technische Universität Braunschweig, Germany.  
(gdb) core-file /home/ansatte/03/dag/IN147/Forelesninger/Kode/core  
Core was generated by 'test-gdb'.  
Program terminated with signal 11, Segmentation fault.  
Reading symbols from /usr/lib/libc.so.1... done.  
#0 0x400abc in main () at test-gdb.c:12  
(gdb) graph display "info locals"  
(gdb) graph display vec  
(gdb) |

DDD

Run

Interrupt

Step	Stepi
Next	Nexti
Cont	Finish
Up	Down
Back	Fwd
Edit	Kill

Her ser vi at `vec[0]` er 0 mens `vec[1]` peker på noe; det burde vært omvendt! (`vec[1]`-`vec[9]` skal ennå ikke ha fått noen verdi siden `i` er 0.)

Altså oppsto feilen under initieringen av `vec` der det står

```
for (i = 0; i < 10; ++i) {  
    vec[1] = (int*)malloc(sizeof(int));  
    *vec[i] = i;  
}
```

Vi kan da avslutte `ddd` med «Exit» i File-menyen.

# Et eksempel til

Følgende program skal skrive ut sine parametre og alle omgivelsesvariablene:

```
#include <stdio.h>

extern char **environ;

int main(int argc, char *argv[])
{
    char **p = environ, *e;
    int i;

    for (i = 0; i < argc; ++i) {
        printf("Parameter %d: '%s'\n", i, argv[i]);
    }

    while (! (*p = NULL)) {
        e = *p;
        printf("%s\n", e);
        ++p;
    }
    return 0;
}
```

Kompilering går fint:

```
> cc -g printenv-feil.c -o printenv-feil
```

men kjøringen går dårlig:

```
> printenv-feil a b  
Parameter 0: 'printenv-feil'  
Parameter 1: 'a'  
Parameter 2: 'b'  
(null)  
(null)  
: Control+\nQuit (core dumped)
```

Hva sier gdb/ddd?



>/usr/bin ddd printenv-feil&

The screenshot shows the DDD (Data Display Debugger) interface. The title bar indicates the file path: `DDD: /home/ansatte/03/dag/IN147/Forelesninger/Kode/printenv-feil.c`. The menu bar includes `File`, `Edit`, `View`, `Program`, `Commands`, `Status`, `Source`, `Data`, and `Help`.

The main window displays the source code of `printenv-feil.c`:

```
#include <stdio.h>

int main(int argc, char *argv[], char **envir)
{
    char **p = envir, *e;
    int i;

    for (i = 0; i < argc; ++i) {
        printf("Parameter %d: '%s'\n", i, argv[i]);
    }

    while (! (*p = NULL)) {
        e = *p;
        printf("%s\n", e);
        ++p;
    }
    return 0;
}
```

Below the code is a control panel with buttons for `Run`, `Interrupt`, `Step`, `Stepi`, `Next`, `Nexti`, `Cont`, `Finish`, `Up`, `Down`, `Back`, `Fwd`, `Edit`, and `Kill`.

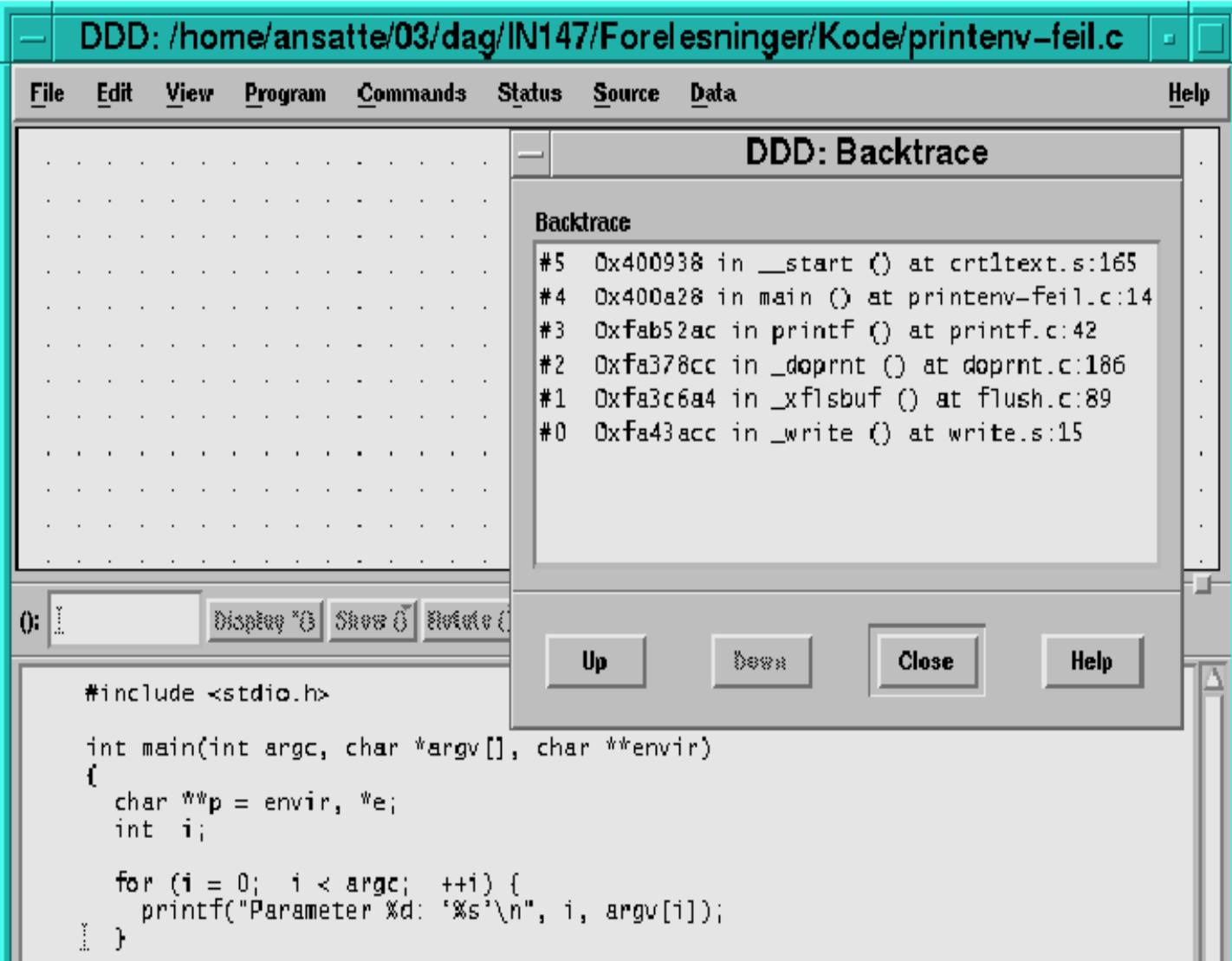
The bottom panel shows the GDB console output:

```
(gdb) core-file /home/ansatte/03/dag/IN147/Forelesninger/Kode/core
Core was generated by 'printenv-feil'.
Program terminated with signal 3, Quit.
Reading symbols from /usr/lib/libc.so.1...done.
#0 0xfa43acc in _write () at write.s:15
write.s:15: No such file or directory.
Current language: auto; currently asm
(gdb) I
```

At the bottom of the window, a status bar reads: `Core was generated by 'printenv-feil'.`

```
DDD 2.2.3 (mips-sgi-irix5.3), by Dorothea Lütkehaus and Andreas Zeller.  
Copyright © 1998 Technische Universität Braunschweig, Germany.  
(gdb) core-file /home/ansatte/03/dag/IN147/Forelesninger/Kode/core  
Core was generated by 'printenv-feil'.  
Program terminated with signal 3, Quit.  
Reading symbols from /usr/lib/libc.so.1...done.  
#0 0xfa43acc in _write () at write.s:15  
write.s:15: No such file or directory.  
Current language: auto; currently asm  
(gdb) I
```

Her ser vi imidlertid at feilen er oppstått i `_write`, men den kaller vi da ikke i vårt program? Vi ber om «Backtrace» i Status-menyen.



The screenshot shows the DDD debugger interface. The main window title is "DDD: /home/ansatte/03/dag/IN147/Forelesninger/Kode/printenv-feil.c". The menu bar includes File, Edit, View, Program, Commands, Status, Source, Data, and Help. The main editor area is currently empty. A "DDD: Backtrace" dialog box is open, displaying the following stack trace:

```
Backtrace
#5 0x400938 in __start () at crt1text.s:165
#4 0x400a28 in main () at printenv-feil.c:14
#3 0xfab52ac in printf () at printf.c:42
#2 0xfa378cc in _doprnt () at doprnt.c:186
#1 0xfa3c6a4 in _xflsbuf () at flush.c:89
#0 0xfa43acc in _write () at write.s:15
```

Below the backtrace are buttons for "Up", "Down", "Close", and "Help". The main editor shows the following code:

```
#include <stdio.h>

int main(int argc, char *argv[], char **envir)
{
    char **p = envir, *e;
    int i;

    for (i = 0; i < argc; ++i) {
        printf("Parameter %d: '%s'\n", i, argv[i]);
    }
}
```

Vi ser at feilen oppsto i linje 14 i main i kallet på printf. Vi klikker på «Up» fire ganger, og velger så «Display Local Variables» i Data-menyen.

Vi ser at p ser OK ut, men e burde vel ikke være 0?

La oss kjøre programmet på nytt, men legge inn et **stoppunkt** øverst i while-løkken. Pek og klikk på linjen, og så på «Break». Så kan vi klikke på «Run» igjen.

DDD: /home/ansatte/03/dag/IN147/Forelesninger/Kode/printenv-feil.c

File Edit View Program Commands Status Source Data Help

**Locals**

```
p = (unsigned char **) 0x7fff2f0c
e = (unsigned char *) 0x0
i = 1
```

**DDD: Backtrace**

**Backtrace**

```
#1 0x400938 in __start () at crt1text.s:165
#0 main () at printenv-feil.c:12
```

Up Down Close Help

0: | Display ^O Hide ^O Rotate ^O

```
#include <stdio.h>

int main(int argc, char *argv[], char **envir)
{
    char **p = envir, *e;
    int i;

    for (i = 0; i < argc; ++i) {
        printf("Parameter %d: %s\n", i, argv[i]);
    }

    while (! (*p = NULL)) {
        e = *p;
        printf("%s\n", e);
        ++p;
    }
    return 0;
}
```

0: -feil.c:12 | Lookup () Clear at () Print ^O Display ^O Find<< () Find>> ()

```
Flush.c:89: No such file or directory.
Current language: auto; currently c
(gdb) up
#2 0xfa378cc in _doprint () at doprint.c:186
doprint.c:186: No such file or directory.
(gdb) up
#3 0xfab52ac in printf () at printf.c:42
printf.c:42: No such file or directory.
(gdb) up
#4 0x400a28 in main (argc=1, argv=0x7fff2f04, envir=0x7fff2f0c) at
printenv-feil.c:14
(gdb) graph display 'info locals'
(gdb) break printenv-feil.c:12
Breakpoint 1 at 0x4009f0: file printenv-feil.c, line 12.
(gdb) run
Starting program: /home/ansatte/03/dag/IN147/Forelesninger/Kode/printenv-
Parameter 0: '/home/ansatte/03/dag/IN147/Forelesninger/Kode/printenv-feil
Breakpoint 1, main (argc=1, argv=0x7fff2f04, envir=0x7fff2f0c) at printenv-feil.c:12
(gdb) |
```

DDD

Run

Interrupt

Step Stepi

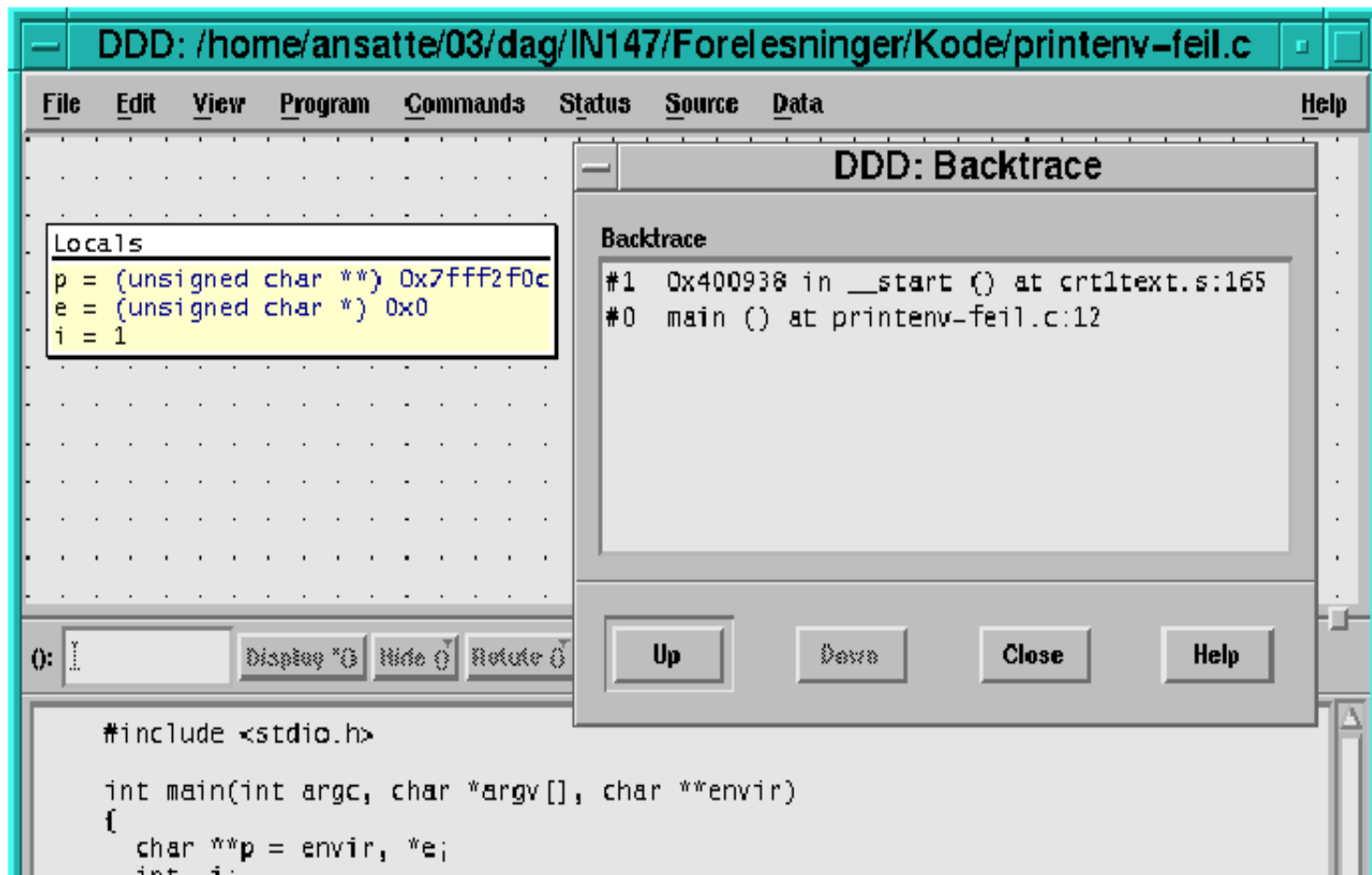
Next Nexti

Cont Finish

Up Down

Back Fwd

Edit Kill



Etter at programmet er stoppet før det skal utføre while-løkken for første gang, lar vi det utføre de to første linjene ved å klikke på «Step».

```
Breakpoint 1, main (argc=1, argv=0x7fff2f04, environ=0x7fff2f0c) at printenv-  
(gdb) step  
(gdb) step  
(gdb) [
```

▲ Updating status displays...done.

**Locals**

```
p = (unsigned char **) 0x7fff2f0c  
e = (unsigned char *) 0x0  
i = 1
```

**DDD: Backtrace**

**Backtrace**

```
#1 0x400938 in __start () at crt1text.s:165  
#0 main () at printenv-feil.c:14
```

Up Down Close Help

0: | Display ^O Hide ^H Rotate ^R

```
#include <stdio.h>  
  
int main(int argc, char *argv[], char **environ)  
{  
    char **p = environ, *e;  
    int i;  
  
    for (i = 0; i < argc; ++i) {  
        printf("Parameter %d: '%s'\n", i, argv[i]);  
    }  
  
    while (! (*p = NULL)) {  
        e = *p;  
        printf("%s\n", e);  
        ++p;  
    }  
    return 0;  
}
```

Her ser vi at p er 0, og det må være galt!

Dette skjedde i den gale testen

```
while (! (*p = NULL)) {
```

som burde vært skrevet

```
while (*p != NULL) {
```

eller

```
while (*p) {
```

## Konklusjon

Noen timer brukt på å lære seg gdb og ddd  
får man mangedobbelt igjen senere i kurset!



# Andre feilsøkingsverktøy

## Programmene lint og splint

Dette programmet sjekker C-programmer og rapporterer mulig feil og foreslår hvorledes koden kan forbedres.

```
> lint printenv-feil.c
```

```
Splint 3.0.1.6 --- 26 Feb 2002
```

```
printenv-feil.c: (in function main)
```

```
printenv-feil.c:16:20: Null storage e passed as non-null param:
```

```
    printf (... , e, ...)
```

A possibly null pointer is passed as a parameter corresponding to a formal parameter with no `/*@null@*/` annotation. If NULL may be used for this parameter, add a `/*@null@*/` annotation to the function parameter declaration. (Use `-nullpass` to inhibit warning)

```
printenv-feil.c:15:9: Storage e becomes null
```

```
Finished checking --- 1 code warning
```

## Kompilatormeldinger

Noen ganger kan kompilatoren gi fornuftige advarsler om potensielle farer hvis man ber om det:

```
> cc -Wall printenv-feil.c
```

## Egne meldinger

Det aller beste er å regne med at man gjør feil og legge inn egne utskrifter som kan slås av og på ved behov.

# Programmet alleyoop

På Linux-maskiner som kjører **Red Hat 9**, finnes programmet alleyoop som kan analysere minnebruken i et C-program.

Dette programmet er i tillegg ypperlig til å finne minnelekasjer.

> *alleyoop test-gdb*

The screenshot shows a debugger window with a menu bar (File, Edit, Settings, Skin, Help) and a toolbar (Run, Kill, Open, Save). Below the toolbar is a search box labeled "Error contains" with a "Clear" button. The main area displays a memory error report:

- ▶ Memcheck, a.k.a. Valgrind, a memory error detector for x86-linux.
- ▼ Invalid write of size 4
  - ▶ at main [test-gdb.c:12]
    - by \_\_libc\_start\_main [in /lib/libc-2.3.2.so]
  - ▶ by ?? [start.S:81]
    - Address 0x0 is not stack'd, malloc'd or free'd

The window has a scrollbar at the bottom.