UNIVERSITETET I OSLO
DET MATEMATISK-NATURVITENSKAPELIGE FAKULTET

Dagens tema

- Info om C
- Cs preprosessor
- Feilsøking

INF1070

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.
CONFORMING TO
       SVID 3, POSIX, BSD 4.3, ISO 9899.  The float and the long double  vari-
       ants are C99 requirements.
SEE ALSO
       hypot(3)
```

INF1070

©Dag Langmyhr,Ifi,UiO: Forelesning 31. januar 2005

Cs preprosessor

Før selve kompileringen går C-kompilatoren gjennom koden med en preprosessor (som er programmet cpp). Dette er en programmerbar tekstbehandler som gjør følgende:

- Henter inn filer

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

Hvis filen er angitt med spisse klammer (som for eksempel <stdio.h>), hentes filen fra området /usr/include. Ellers benyttes vanlig notasjon for filer.

INF1070

©Dag Langmyhr,Ifi,UiO: Forelesning 31. januar 2005

- Leser makro-definisjoner og ekspanderer disse i teksten:

```
#define LINUX
#define N 100
#define MIN(x,y) ((x)<(y) ? (x) : (y))
```

Av gammel tradisjon gis makroer navn med store bokstaver.

(En **makro** er en navngitt programtekst. Når navnet brukes, blir det **ekspandert**, dvs erstattet av definisjonen. Dette er ren tekstbehandling uten noen forbindelse med programmeringsspråkets regler.)

Benytter man makroer med parametre, bør disse settes i parenteser. Likeledes, hvis definisjonen er et uttrykk med flere symboler, bør det stå parenteser rundt hele uttrykket.

- Betinget kompilering. Her angis hvilke linjer som skal tas med i kompileringen og hvilke som skal utelates.

INF1070

©Dag Langmyhr,Ifi,UiO: Forelesning 31. januar 2005

Betinget kompilering
Følgende direktiver finnes for betinget kompilering:

#if Hvis uttrykket etterpå er noe annet enn 0, tas etterfølgende linjer tas med. Uttrykket kan ikke inneholde variable eller funksjoner.

#ifdef Hvis symbolet er definert (med en #define), skal etterfølgende linjer tas med.

#ifndef Motsatt av #ifdef.

#else Skille mellom det som skal tas med og det som ikke skal tas med.

#endif Slutt med betinget kompilering.

Eksempel:

```
#define LINUX

#ifdef LINUX
  int x;
#else
  long x;
#endif
```

INF1070

Det er også mulig å styre betinget kompilering gjennom gcc-kommandoen:

```
> gcc -c -DLINUX
```

gir samme effekt som om det sto

```
#define LINUX
```

i program-koden.

På denne måten er det mulig å ha flere versjoner av koden (for eksempel for flere maskin-typer) og så kontrollere dette utelukkende gjennom kompileringen.

Fare med betinget kompilering
Man kan risikere å ha kode som aldri har vært kompilert, og som kan inneholde de merkeligste feil.

INF1070

Separat kompilering

I utgangspunktet er det ingen problem med separat-kompilering i C; hver fil utgjør en enhet som kan kompileres for seg selv, uavhengig av alle andre filer i programmet.

```
> gcc -c del.c
```

vil kompilere filen del.c og lage del.o som inneholder den kompilerte koden.

INF1070

Eksempel

Anta at vi har to filer:

Filen sum.c:

```
int sum (int n)
{ /* Beregner 1+2+...+n */
  return n*(n+1)/2;
}
```

Filen vissum.c

```
#include <stdio.h>

extern int sum (int n);

int main (void)
{
  int i;
  for (i = 1;  i <= 10;  ++i)
    printf("%2d:%4d\n", i, sum(i));
}
```

Kompilering
Disse kan kompileres hver for seg:

```
> gcc -c sum.c
> gcc -c vissum.c
```

INF1070

Linking
De kompilerte filene kan siden **linkes** sammen:

```
> gcc vissum.o sum.o -o vissum
```

Kjøring
Da får vi et ferdig program som kan kjøres:

```
> ./vissum
 1:   1
 2:   3
 3:   6
 4:  10
 5:  15
 6:  21
 7:  28
 8:  36
 9:  45
10:  55
```

INF1070

Imidlertid er det en fare for at funksjonssignaturer, strukturer, makroer, typer og andre elementer ikke blir skrevet likt i hver fil. Dette løses ved hjelp av definisjonsfiler («header files»), hvis navn gjerne slutter med '.h'.

Filen incl.h:

```
#define N 100
```

INF1070

Filen prog.c:

```
#include "incl.h"

int main(void)
{
  char *s[N];
    ⋮
}
```

Definisjonsfiler inneholder gjerne følgende:

- Makrodefinisjoner (#define)
- Typedefinisjoner (typedef, union, struct)
- Eksterne spesifikasjoner (extern)
- Funksjonssignaturer som

  extern int f(int,char);

INF1070

Debuggere
En «debugger» er et meget nyttig feilsøkingsverktøy. Det kan

- analysere en program-dump,
- vise innholdet av variable,
- vise hvilke funksjoner som er kalt,
- kjøre programmet én og én linje, og
- kjøre til angitt stoppunkter.

Debuggeren gdb er laget for å brukes sammen med gcc. Den har et vindusgrensesnitt som heter ddd som kan brukes på Unix-maskiner.

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!*)

INF1070

Et program med feil

Følgende program prøver å

❶ sette opp en vektor med 10 pekere til heltall,

❷ sette inn tallene 0–9 og

❸ skrive ut tallene.

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

int *vec[10];

int main(void)
{
  int i;

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

INF1070

Programdumper

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

❶ Programmet kompileres med debuggingsinformasjon:

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

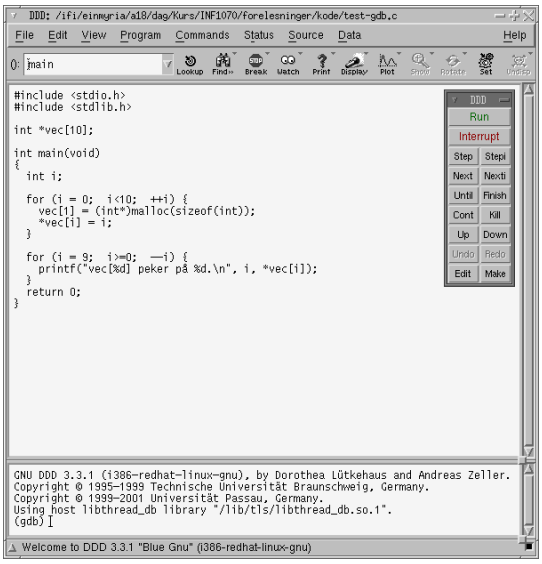
❷ Programmet kjøres:

```
> ./test-gdb
Segmentation Fault (core dumped)
> ls -l core*
-rw------- 1 dag  ifi-a  188416 2005-01-27 10:27 core.20816
```

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

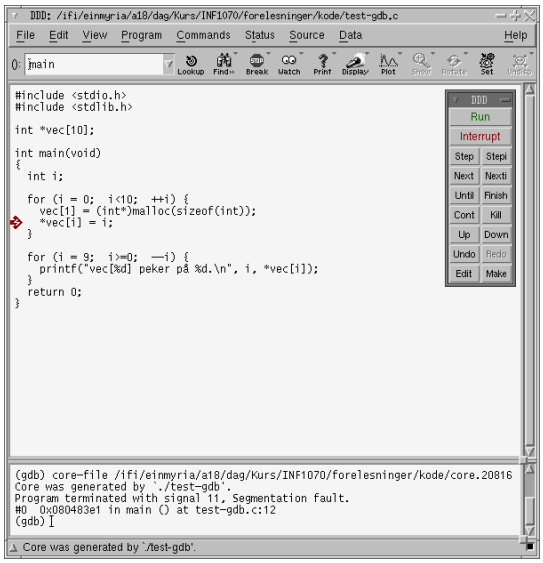
INF1070


```
> ddd test-gdb &
```



INF1070

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

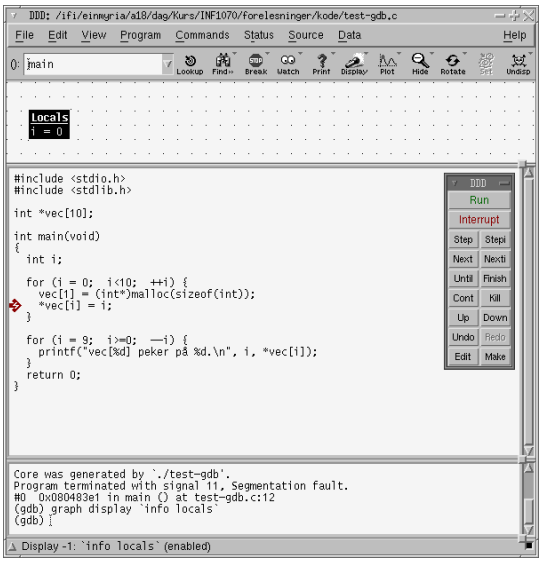


Nå vet vi at feilen oppsto på linje 12 i forbindelse med *vec[i] = i.

INF1070

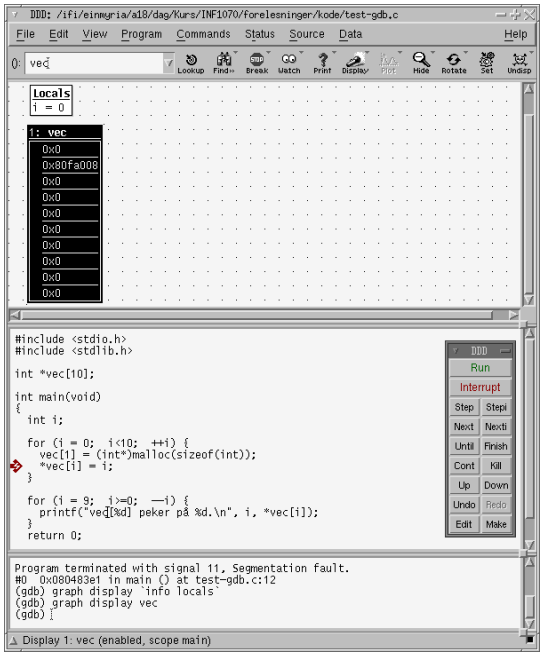
Kanskje det er noe galt med indeksen i?

I Data-menyen finner vi «Display Local Variables»



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.)



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:

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

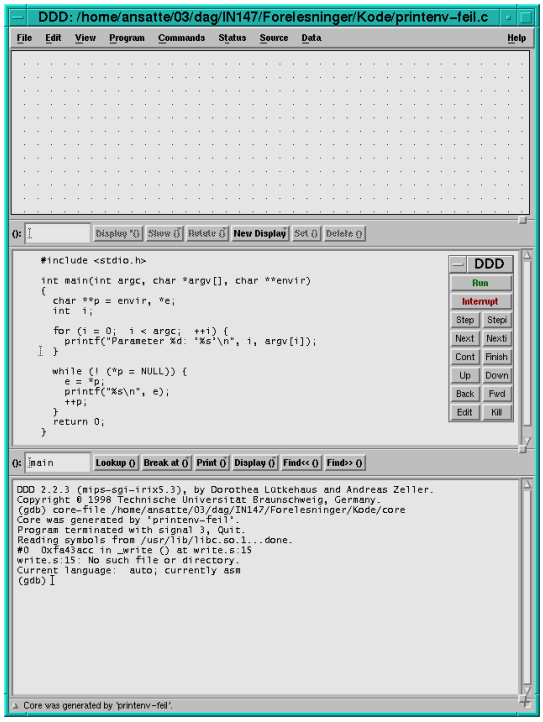
men kjøringen går dårlig:

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

INF1070

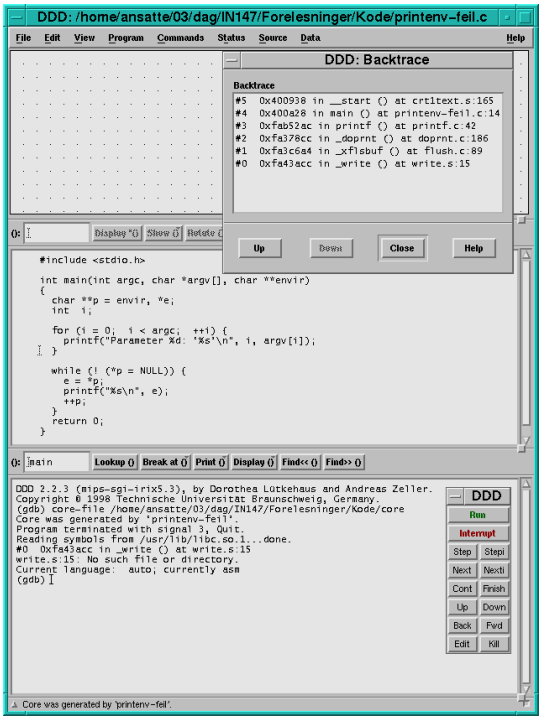
Hva sier gdb/ddd?

```
> ddd printenv-feil &
```



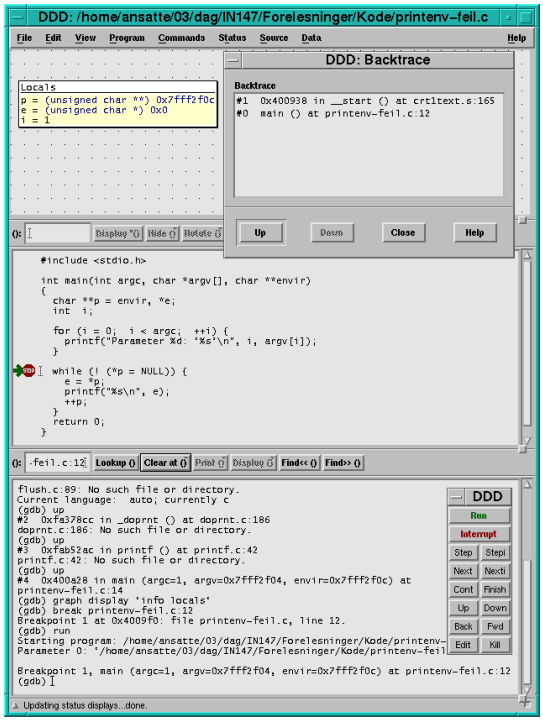
INF1070

Her ser vi imidlertid at feilen er oppstått i _write?? Vi ber om «Backtrace» i Status-menyen.



INF1070

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.



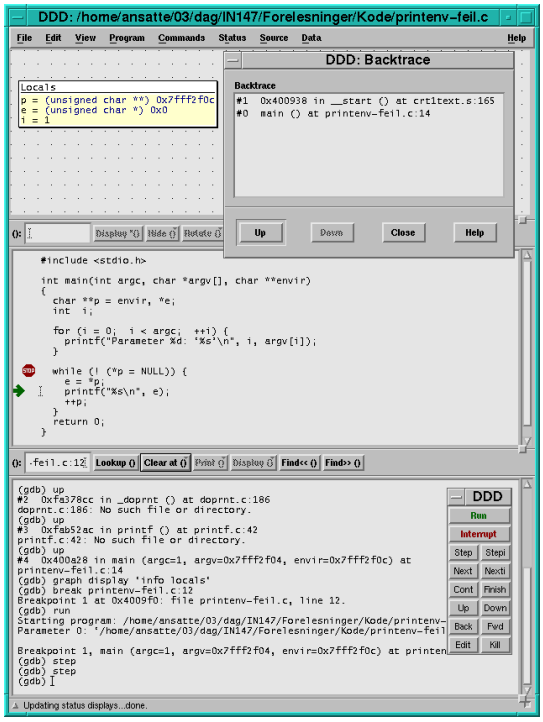
Vi ser at p ser OK ut, men skal e være 0?

INF1070

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.

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».

INF1070



INF1070

Her ser vi at e 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!

INF1070

Andre feilsøkingverktøy

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

```
> splint printenv-feil.c
Splint 3.0.1.7 --- 24 Jan 2003
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
```

INF1070

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

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

(Men ikke denne gangen!)

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.

INF1070