



## Revisjonshistorie

År	Forfatter
2020	Kolbjørn Austreng
2021	Kiet Tuan Hoang

## I Introduksjon - Praktisk rundt øvingen

Denne øvingen handler om strukturert debugging med GDB (**G**NU **D**ebugger) og et profileringsverktøy som heter Valgrind. GDB er en debugger som gjør det mulig å "steppe" gjennom en kode, ved at den tillater at brukeren kan stoppe på vilkårlige steder, inspisere variabler, sette verdier, og mer. Valgrind, derimot, lar brukeren profilere minnebruken - noe som kan hjelpe med å fjerne minnelekkasjer eller oppdage når man bruker minne som ikke har blitt allokert (en "use after free" - feil).

I utgangspunktet har ikke GDB et pent grafisk grensesnitt. Det er derimot mange *wrappers* som brukes - mest kjent er programvaren **DDD**. I denne øvingen skal vi bruke en "semi-grafisk" visning som kommer med GDB, kalt TUI (**T**ext **U**ser **I**nterface). Denne visningen kan benyttes ved å starte **gdb** med flagget **-tui**, eller ved å trykke **Ctrl + X, A** fra en vanlig **gdb**-instans.

I tillegg til vanlig debugging av C-kode, har det blitt lagt til et appendiks om debugging av mikrokontrollere i appendiks [B](#). Dette blir viktig for mikrokontroller-labben.

### Oppsett

GDB fungerer ved å kjøre maskinkoden til et compilert program i bakgrunnen, og vise hvilke kodelinjer dette svarer til i forgrunnen. Ved vanlig kompilering og lenking kastes *symboltabellen*, så vi trenger en kompilator som kan ivareta denne under kompilering.

I denne øvingen, brukes **gcc** som har denne muligheten. I tillegg, trenger man debuggeren selv - **gdb**. Begge disse skal være installert på Sanntidsalen. Installasjon av både **gcc** og **gdb** er rett fram på Linux og Mac, mens på Windows er MinGW den enkleste måten å kjøre programvaren.

# 1 Oppgave - Grunnleggende debugging med gdb

Denne øvingen er strukturert som en walkthrough. Gjennom øvingen skal dere compilere feilaktige kodesnutter og debugge den med `gdb`. Ofte vil feilene være veldig opplagt, men motstå allikevel fristelsen til å skippe over. Hensikten med denne øvingen, er at dere skal vite hvordan dere bruker GDB, så for å få godkjent, må dere vise en studass at dere til en viss grad forstår hva som skjer i denne øvingen. Terskelen er ikke altfor høy, og dere får ganske greit godkjent om dere viser at dere forstår hvordan dere kan debugge et vilkårlig program med GDB.

## 1.1 Breakpoints

Breakpoints er den mest brukte-, og en av de nyttigste egenskapene til en debugger. De tillater deg å midlertidig stoppe kodekjøringen for å inspisere variabler, og liste opp hvordan du kom til linjen du stoppet på. GDB operer med tre forskjellige typer breakpoints:

- *Breakpoint* stopper kodekjøringen når man kommer til en spesifikk linje.
- *Watchpoint* stopper kodekjøringen dersom en spesifisert variabel eller minneadresse endrer verdi.
- *Catchpoint* stopper kodekjøringen dersom en definert hendelse inntreffer. Hendelsen kan for eksempel være et kall til `exec`, et `syscall`, en `assert`, eller en `exception`.

Av disse er vanlig breakpoints desidert av mest nytte. Selv om watch- og catchpoints av og til kan være praktiske å ha, blir de fort ubrukelige dersom man debugger kode med flere tråder.

For å demonstrere hvordan man vanligvis bruker breakpoints, skal vi bruke programmet `break`. Kall `make break` for å compilere, og `gdb -tui break` (i samme mappe som programmet `break`) for å starte GDB med programmet vårt.

Breakpoints kan settes på forskjellige måter. Om man nå skriver `break local_call` setter man et breakpoint på starten av funksjonen `local_call`. Dersom man nå kaller `break 9` setter man et breakpoint på kodelinje 9. Gitt at du nå har gjort alt dette, kan vi kalle `run` og få noe tilsvarende som dette:

```
(gdb) break local_call
Breakpoint 1 at 0x63e: file source/break.c, line 5.
(gdb) break 9
Breakpoint 2 at 0x655: file source/break.c, line 9.
(gdb) run
Starting program: /home/student/Desktop/Øving - 4/programs/break
```

```
Breakpoint 2, main () at source/break.c:11
```

Som man kan se, så stopper GDB først på linje 11, til tross for at vi satte et breakpoint til linje 9. Dette er fordi det ikke skjer noe spennende på linje 9; vi oppretter kun en variabel, men vi gir den ikke noen verdi. GDB operer egentlig ikke med linjene kode du ser på toppen av skjermen, men med maskininstruksjonene de har kompilert til. Det kan derfor fint hende at linje 9 ikke svarer til en instruksjon GDB kan stoppe ved. Dette er verd å ha i bakhodet dersom GDB tilsynelatende ikke gjør det du sier.

I tillegg til vanlig breakpoints har GDB også betingede breakpoints. Dette er breakpoints som kun aktiveres dersom en betingelse er oppfylt. Skriv `break 14 if i>60` for å sette et breakpoint midt i `for`-løkken, som kun aktiveres dersom teller-variabelen er større enn 60. Hvis man kaller `continue` for å gå videre fra linje 11, vil dere stoppe på linje 14. Om dere nå bruker `print`-kommandoen vil dere se at teller-variabelen `i` har verdi 61. Altså ble ikke breakpointet aktivert før vi ønsket.

Sett derimot nå at vi i utgangspunktet tok feil; vi ønsket ikke å stoppe når `i>60`, men `i>80`. Om vi skriver `info break` får vi opp en oversikt over hvilke breakpoints vi har:

```
(gdb) info break
Num      Type          Disp  Enb   Address                What
1        breakpoint keep   y     0x000055555555463e    in local_call at
-> source/break.c:5
2        breakpoint keep   y     0x0000555555554655    in main at
->source/break.c:9
          breakpoint already hit 1 time
3        breakpoint keep   y     0x0000555555554665    in main at
->source/break.c:14
          stop only if i > 60
          breakpoint already hit 1 time
```

Vi kan nå kalle `condition 3 i > 80` for å endre betingelsen til breakpoint nummer 3 til å kunne aktiveres dersom `i` er større enn 80. Nå kan dere kalle `continue`, etterfulgt av `print i`, som vil demonstrere at vi faktisk bare stopper når vi vil:

```
(gdb) condition 3 i > 80
(gdb) continue
Continuing.
```

```
Breakpoint 3, main () at source/break.c:14
(gdb) print i
$2 = 81
```

Vi kan fjerne betingelsen fra breakpoint 3 ved å simpelthen kalle `condition 3`.

Om vi nå kaller `continue` vil vi derimot stoppe i `for`-løkken hver gang, fordi `i` nå alltid er større enn 80. Om vi er ferdig med breakpoint 3, kan vi skru det av ved å kalle `disable 3` - dette vil endre `Enb`-feltet (Enable) fra `y` til `n`:

```
(gdb) disable 3
(gdb) info break
Num      Type      Disp    Enb      Address          What
1        breakpoint keep    y      0x000055555555463e  in local_call at
->source/break.c:5
2        breakpoint keep    y      0x0000555555554655  in main at
->source/break.c:9
          breakpoint already hit 1 time
3        breakpoint keep    n      0x0000555555554665  in main at
->source/break.c:14
          breakpoint already hit 3 times
```

Vi kan også permanent slette breakpoint 3 ved å kalle `delete 3`, om vi vet at vi ikke har behov for det igjen. I tillegg til å jobbe på en fil, kan GDB også sette breakpoints i andre filer, som for eksempel filen `break2.c`, ved å bruke en `<fil>:<linje eller funksjon>`-syntaks:

```
(gdb) break break2.c:library_call
Breakpoint 4 at 0x55555555468a: file source/break2.c, line 4.
```

Om man kun trenger å stoppe et sted en gang, så har GDB også midlertidige breakpoints, som på samme måte som vanlige breakpoints, men med kodeordet `tbreak` istedenfor `break`:

```
(gdb) tbreak library_call
Note: breakpoint 4 also set at pc 0x55555555468a.
Temporary breakpoint 5 at 0x55555555468a: file source/break2.c, line 4
```

Om dere nå kaller `info break` vil dere se at `Disp`-feltet (disposition) er forskjellig fra vanlige breakpoints:

```
4 breakpoint keep y 0x000055555555468a in library_call at
->source/break2.c:4
5 breakpoint del y 0x000055555555468a in library_call at
->source/break2.c:4
```

hvor `keep` betyr at breakpointet vil beholdes, mens `del` indikerer at et breakpoint vil slettes etter at det blir truffet.

En siste ting som er greit å vite om breakpoints er at du kan tilknytte vilkårlige kommandoer til hvert enkelt breakpoint, som vil kjøre hver gang breakpointet aktiveres. Om dere for eksempel ønsker å legge inn en `printf`, uten å måtte endre koden og gjenkompilere, kan dere gjøre slik:

```
(gdb) break 14
Breakpoint 6 at 0x555555554665: file source/break.c, line 14.
(gdb) command 6
Type commands for breakpoint(s) 6, one per line.
End with a line saying just "end".
>printf "Value of i is %d\n", i
>end
```

Dette vil skrive ut verdien til `i` på samme måte som ”gammeldags `printf`-debugging” gjør, men du slipper å tukle med koden og compilere på nytt. GDB kan faktisk kalle en hvilken som helst funksjon - også bibliotek-funksjoner - via breakpoint-kommandoer. Dette er en veldig kraftig egenskap.

## 1.2 Inspeksjon og endring av variabler

GDB kan inspisere og endre variabler under kjøring. For å illustrere hvordan dette kan gjøres, skal vi bruke programmet `inspect`. Kall `make inspect`, og deretter `gdb -tui inspect` for å starte GDB.

Sett opp et break-point i slutten av `main` ved å kalle `break 21`. Kall deretter `run`. Når GDB stopper vil det være like før programmet returnerer. Kall nå `print stat_array` og `print dyn_array`:

```
Breakpoint 1, main () at source/inspect.c:21
(gdb) print stat_array
$1 = {1, 2, 3, 4, 5}
(gdb) print dyn_array
$2 = (int *) 0x555555756260
```

Som man kan se ut fra koden i `inspect.c`, inneholder `stat_array` og `dyn_array` den samme informasjonen, men de er allokert henholdsvis statisk og dynamisk. Når vi printer den dynamiske tabellen, får vi bare pekeren. Dette skjer fordi GDB ikke kan være sikker på nøyaktig hva som ligger bak pekeren. Ettersom vi vet hva pekeren peker til, kan vi utnytte en spesiell syntaks for å skrive den ut som et array:

```
(gdb) print *dyn_array@5
$3 = {1, 2, 3, 4, 5}
```

Dette betyr bare at vi skal dereferere pekeren, og ta 5 elementer, og skrive dem ut som en tabell. Det er ikke veldig ofte man trenger denne syntaksen, men den kan være veldig nyttig i noen sammenhenger.

I tillegg, kan GDB også skrive ut structs:

```
(gdb) print sleipner
$4 = {legs = 8, can_fly = 1}
```

Kanskje det mest nyttigste med GDB er at den kan endre variabler underveis ved å bruke kommandoen **set**:

```
(gdb) print some_number
$5 = 1
(gdb) set some_number = 2
(gdb) print some_number
$6 = 2
```

hvor **\$**-tegnet brukes som en referanse til GDB sin variabelhistorie. Dette tallet vil øke med en for hver kommando.

## Display

GDB har også en kommando kalt **display**, som fungerer på samme måte som **print**, men automatisk skriver ut en variabel hver gang et breakpoint aktiveres, så lenge variabelen finnes på øverste stack-frame. For å stoppe en aktiv **display** brukes kommandoen **undisplay** med ID-en til **display**-et.

For å prøve ut **display** kan dere kalle **break 14** for å opprette et breakpoint ved linja `dyn_array[i] = i + 1;`, etterfulgt av **display \*dyn\_array@5**, og til slutt **run** for å starte programmet på nytt. Hver gang dere treffer break-pointet vil dere se at ett nytt element er blitt satt i `dyn_array`. For å slippe å skrive **continue** mange ganger kan dere forresten bare trykke enter for å repetere siste kommando som ble kjørt.

## 1.3 step og next

Det som gjør GDB spesielt nyttig er at den lar brukeren gå gjennom koden linje for linje. Kompiler og debug programmet **step** ved å kalle **make step** etterfulgt av **gdb -tui step**. I GDB kaller dere **break main**. fulgt av **run**. Programmet vil nå stoppe på linjen `int prod1 = multiply(2, 6);`. Kall nå kommandoen **step** fire-fem ganger og legg merke til hva som skjer.

Deretter starter dere programmet på nytt ved å kalle **run** igjen. Denne gangen kaller dere **next** isdenfor **step** et par ganger. Som dere ser, fungerer de to kommandoene litt annerledes:

- **step** vi gå en kodelinje videre, og for hvert funksjonskall den møter på, vil **step** gå inn i implementasjonen.
- **next** vil gjøre det samme som **step**, men vil kjøre alle funksjonskall den møter i bakgrunnen - uten å gå inn i implementasjonen.

Altså: når du kaller **step** vil den kun gå inn i en implementasjon dersom det finnes et symbol for funksjonen. Dermed vil den gå inn i all kode compilert med flagget **-g**, men hoppe over ting den ikke kan lese. Dermed vil ikke **step** prøve å gå inn i implementasjonen til for eksempel **printf**.

## 1.4 until og finish

Om man kommer inn i en løkke, eller havner i en funksjon som man er sikker på er feilfri, så er det mulig å kunne hoppe ut av den. For dette formålet har GDB kommandoene **until** og **finish**. Kall først **make until**, og deretter **gdb -tui until**. Inne i GDB oppretter dere et breakpoint for **main** ved å kalle **break main**.

Hvis dere nå kaller **run** vil dere stoppe i toppen av **for**-løkka, fordi dette er den første interessante kodelinjen i **main**. Dersom man nå ønsker å hoppe over løkken, og fortsette rett etter den, kan man kalle **until** tre ganger. Dere vil se at GDB nå stopper på linje 21 som er rett før **pointless\_function**-kallet.

Kommandoen **until** skal hoppe til kodelinjen som er "1 større enn nåværende kodelinje". Grunnen til at dere måtte kalle **until** tre ganger istedenfor en, er at GDB arbeider med maskinkoden til programmet. I C-kode, ser det ut som om at vi gjør en test i starten av løkken - men i den ferdigkompilete koden skjer dette faktisk i bunnen av løkken. Dette er ikke veldig viktig å huske på, men det er greit å vite dersom en kommando tilsynelatende oppfører seg litt rart.

Når dere så har kommet til **pointless\_function**, kaller dere **step** for å gå inn i implementasjonen. Sett nå at dere har gjort dere ferdig inne i denne funksjonen og ønsker å komme tilbake til **main**. Man kunne brukt midlertidige breakpoints og **continue**, men dette blir fort tungvint. Man kan heller bruke **finish** som er GDB sin egendefinerte kommando for akkurat dette. Denne kommandoen vil fullføre et funksjonskall, og stoppe kodeutføringen rett etter funksjonen returnerer.

## 1.5 Callstack

Ofte er tilfellet at en variabel uventet skifter verdi, eller at en funksjon kalles ut av det blå. Dette skjer vanligvis fordi man har en lang linje av funksjoner etter hverandre, der en av dem er feil. For å gjøre det lettere å inspisere hva som faktisk skjer, kan GDB skrive ut kall-stacken, eller hver enkelt *stack-frame*.

Kall først **make trace** for å compilere programmet **trace**, etterfulgt av **gdb -tui trace** for å debugge det. Fra GDB kan dere nå kalle **watch global\_value** for å stoppe når den globale variabelen **global\_value** på mystisk vis skifter verdi. Om dere nå kaller **run**, vil dere få en output som ligner på denne:

```
(gdb) watch global_value
Hardware watchpoint 1: global_value
(gdb) run
Starting program: /home/student/Desktop/Øving - 4/programs/trace

Hardware watchpoint 1: global_value

Old value = 0
New value = 1
myst_func_1 (level=0) at source/trace.c:40
```

Variabelen skifter altså verdi fra 0 til 1, og i dette tilfellet skjedde det før linje 40. Som man kan se, så kan vi ikke være helt sikre på nøyaktig hvilken linje endringen skjedde, ettersom GDB egentlig jobber med maskinkoden til programmet, og prøver deretter så godt den kan å koble maskinkoden til C-koden. Heldigvis er det ikke så veldig interessant hvilken linje endringen skjedde på, så lenge vi får en anelse.

Kall nå `backtrace`. Dette vil skrive ut kall-stacken til programmet - altså hvilke funksjonskall som har hendt til nå. Dere vil få noe som ser slik ut:

```
(gdb) backtrace
#0 myst_func_1 (level=0) at trace.c:40
#1 0x0000555555554929 in myst_func_4 (level=1) at source/trace.c:96
#2 0x0000555555554956 in myst_func_4 (level=2) at source/trace.c:105
#3 0x0000555555554956 in myst_func_4 (level=3) at source/trace.c:105
#4 0x000055555555483c in myst_func_2 (level=4) at source/trace.c:59
#5 0x000055555555481e in myst_func_2 (level=5) at source/trace.c:53
#6 0x000055555555481e in myst_func_2 (level=6) at source/trace.c:53
#7 0x0000555555554938 in myst_func_4 (level=7) at source/trace.c:99
#8 0x0000555555554956 in myst_func_4 (level=8) at source/trace.c:105
#9 0x000055555555483c in myst_func_2 (level=9) at source/trace.c:59
#10 0x0000555555554791 in myst_func_1 (level=10) at source//trace.c:30
#11 0x0000555555554979 in main () at source/trace.c:113
```

Helt på bunnen vil dere se at utgangspunktet vårt var `main`, som så kalte `myst_func1` med `level`-parameter lik 10. Deretter vet vi at en rekke kall til forskjellige `myst_func`-funksjoner blir gjort, før vi kommer til `myst_func_2` som må ha endret `global_value`. En slik utskrift av kall-stacken kan gi en veldig god indikasjon på hvor ting går galt.

Om man har lyt på en spesifikk *stack-frame* kan GDB også gi oss det. En stack-frame er ett nivå på kallstacken, og inneholder informasjon om hvilken funksjon som ble kalt, og hvilke argumenter som ble gitt. Kall `frame` for å skrive ut øverste stack-frame, eller for eksempel `frame 8` for å skrive ut en stack-frame 8 nivåer ned:

```
(gdb) frame 6
#6 0x000055555555481e in myst_func_2 (level=6) at source/trace.c:53
(gdb) frame 8
#8 0x0000555555554956 in myst_func_4 (level=8) at source/trace.c:105
```

## 1.6 Segfault

Dersom dere kompiler programmet `seg1` ved å kalle `make seg1`, etterfulgt av `./seg1`, vil dere få en segmentation fault - altså forsøker programmet å allokere minne den ikke eier eller har tilgang til. For å undersøke hva som er feil, bruker vi



GDB som vanlig ved å kalle `gdb -tui seg1`. Dersom vi først nå kjører programmet ved å kalle `run` vil vi få noe slikt:

```
Program received signal SIGSEGV, Segmentation fault.  
0x000055555555461e in main () at source/seg1.c:7
```

Dette forteller oss at vi gjorde en ulovlig minneoperasjon i `seg1.c`, på linje 7. Vi har allerede bare fra dette mye å gå på dersom vi skal fikse på koden. Kall deretter `print i`, for å skrive ut verdien av teller-variabelen. Den eksakte verdien er avhengig av arkitektur, men det burde være negativ. Utfra koden er det ganske åpenbart, men la oss nå late som om at vi fortsatt ikke er helt sikre på hvorfor.

For å få mer informasjon, kan vi kalle `break 7`, for å sette opp et breakpoint på linje 7. Så kaller dere `info break` for å få en liste med informasjon om breakpointene dere har i programmet. Dere vil ha noe som ser slik u:

```
(gdb) break 7  
Breakpoint 1 at 0x555555554607: file source/seg1.c, line 7.  
(gdb) info break  
Num      Type           Disp Enb   Address           What  
1        breakpoint      keep y    0x0000555555554607 in main at source/seg1.c:7
```

Vi skal nå knytte kommandoer til breakpointet, som skal kjøre hver gang vi kommer til det. Kall `command 1` for å komme inn i kommando-modus for breakpoint 1. Deretter skriver dere `silent` på en linje, `info local` på neste linje, og til slutt `end` på siste linje - slik:

```
(gdb) command 1  
Type commands for breakpoint(s) 1, one per line.  
End with a line saying just "end".  
>silent  
>info local  
>end
```

Om dere nå kaller `info break` igjen vil dere se at breakpoint 1 har fått to kommandoer knyttet til seg:

```
(gdb) info break  
Num Type           Disp Enb   Address           What  
1   breakpoint      keep y    0x0000555555554607 in main at source/seg1.c:7  
    silent  
    info local
```

Her betyr `silent` at GDB ikke vil skrive ut informasjon om breakpointet hver gang det aktiveres, og `info local` betyr at vi ønsker å skrive ut de lokale variablene som finnes til skjermen for inspeksjon.

Om vi nå starter programmet på nytt ved å kalle `run`, vil GDB stoppe på linje 7,

og skrive ut `i = 0`. Om man deretter repetitivt kaller `continue` for å gå til neste løkke-iterasjon får man noe slikt:

```
i = 0
(gdb) continue
Continuing.
i = -1
Continuing.
i = -2
Continuing.
i = -3
Continuing.
i = -4
```

Det er nå åpenbart hva som foregår - vi teller nedover, når vi skulle ha telt oppover. Dette kommer av at vi har skrevet `i--` i løkken, mens vi egentlig mente `i++`.

### 1.6.1 Tilleggs-kommentarer til Segfault

Noen ganger så er man så uheldige at man ikke får generert en segfault til tross for at det er ting som man ikke burde gjort med minnet. Et eksempel får man ved å kompilere programmet `seg2`. Om man kjører `seg2`, med `make seg2` og `./seg2` får vi et program som allokerer ved *compile time* et array av heltall, med størrelse 200. Programmet vil deretter prøve å skrive over 240 av disse elementene. Altås vil programmet prøve å skrive til 40 minnesegmentet som programmet egentlig ikke eier.

I utgangspunktet burde dette ha gitt en segfault. Om dere ikke har fått en segfault før nå, så kan dere skrive `Exit normally`.

Refer til appendix for å få en forklaring på hvorfor akkurat denne koden ikke gir en segfault.

## 1.7 Minnelekkasje

I teorien skjer ikke minnelekkasje dersom man er forsiktig. Men om man har et program som glemmer å frigjøre minne, og det kjører over lang nok tid - så vil programmet bryte sammen til slutt. Et av de mest effektive verktøyene for bekjempe minnelekkasje kalles Valgrind. Valgrind emulerer en virtuell CPU og kjører programmet på denne. Dette gir brukeren full kontroll over hva som allokeres av minne - og hva som gis eller ikke gis tilbake.

Dersom man kaller `make leak` fra kommandolinjen for å kompilere programmet `leak`, etterfulgt av `./leak 12` vil programmet skrive ut de 12 første Fibonacci-tallene. Hvis man nå kjører kommandoen `valgrind --leak-check=yes ./leak 12`, vil man få en output som minner om denne:

```
==5504==
```

```

==5504== HEAP SUMMARY:
==5504==      in use at exit: 48 bytes in 1 blocks
==5504==    total heap usage: 2 allocs, 1 frees, 1,072 bytes allocated
==5504==
==5504== 48 bytes in 1 blocks are definitely lost in loss record 1 of 1
==5504==    at 0x4C31B0F: malloc (vg_replace_malloc.c:299)
==5504==    by 0x108345: create_array (leak.c:5)
==5504==    by 0x10880A: main (leak.c:23)
==5504==
==5504== LEAK SUMMARY:
==5504==    definitely lost: 48 bytes in 1 blocks
==5504==    indirectly lost: 0 bytes in 0 blocks
==5504==    possibly lost: 0 bytes in 0 blocks
==5504==    still reachable: 0 bytes in 0 blocks
==5504==    suppressed: 0 bytes in 0 blocks
==5504==
==5504== For counts of detected and suppressed errors, rerun with: -v
==5504== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)

```

Dette er en oppsummering av heap-forbruk i løpet av programmets levetid. Som man kan se, har vi mistet 48 bytes fordi vi ikke frigjør allokeringen vi gjør. Valgrind hjelper oss også med å se hvor lekkasjen skjer; vi kan se av *stack-tracen* at vi fra `main` kaller `create_array`, som deretter kaller `malloc` uten at `free` kalles noe sted.

## A Appendiks - Forklaring på segfault

Når dere kompilerer et program, vil dere til slutt ende opp med noe som ligner på figur 1. Programmet er delt opp i områder med forskjellige rettigheter. Feltet `.text` er minne-området som inneholder maskinkoden som faktisk kjører, samt statisk lenkede biblioteker. Dette området krever lese- og kjørerettigheter, men vi har virkelig ikke lyst til å gi området skriverettigheter - fordi programmet da potensielt kunne ha skrevet om seg selv.

Feltet `.data` og `.bss` inneholder henholdsvis *compile time* initialisert- og allokert, men uinitialisert minne. Disse feltene er for globale variabler.

Heap-feltet er området minne-kall til `malloc` og `free` vil virke på. Heapen er for dynamisk allokert minne, og vil vokse oppover dersom en gjennom kall til `malloc` bruker opp allokert heap-størrelse.

Ovenfor heapen ligger området for stacken. Denne vil bestå av såkalte stack-frames med lokale variabler. Om du gjør uendelig rekursjon i et språk som ikke støtter *tail recursion*, så vil stacken vokse nedover til den til slutt kolliderer med heapen. Dette er det som kalles en "stack overflow".

Nøyaktig hvor dynamisk lenkede biblioteker ligger, og hvor kopier av eksterne

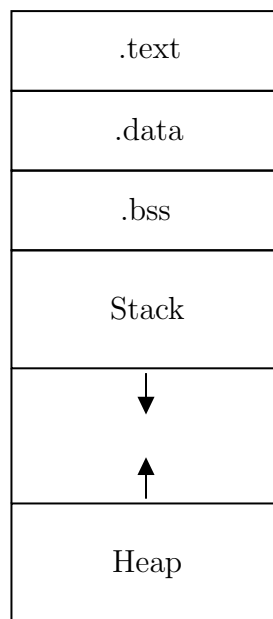


Figure 1: Vanlig minne-struktur for et program.

variabler finnes, er implementasjonsspesifikt - men de generelle trekkene i figur 1 stemmer som regel.

Når dere kjører et program vil dere som regel ikke jobbe direkte med informasjon som er lagret på harddisk eller annet sekundærminne. Grunnen til dette er at både skriving til og lesing fra datamaskinens primærminne som ofte består av RAM (**R**andom **A**ccess **M**emory) er betydelig raskere enn for harddisken. Under kjøring av et program laster derfor datamaskinen deler av informasjon fra sekundærminne til primærminne, altså typisk fra harddisk til RAM. Disse delene kalles *memory pages*, og størrelsen på disse er avhengig av arkitektur. Operativsystemet vil gi hver *memory page* diverse flagg for å holde styr på hva man kan gjøre med minnet.

Uheldigvis er det ganske vanlig at man har et array som ikke tar en hel *page*. Dermed kan man gjøre ulovlige minne-operasjoner så lenge man er "heldig" og treffer en *page* som man har aksess-rettigheter til. En *out-of-bounds* på ett element vil derfor ikke alltid gi en segfault, som det burde.

## B Appendiks - Debugging av mikrokontrollere

Det er mulig å bruke GDB for å debugge mikrokontrollere, så lenge man har en måte å kommunisere debuggingssignaler til hardwaret. I tillegg må hardwaret også støtte debugging, men dette er stort sett garantert i nye mikrokontrollere. Det er mange måter å gå frem - men her er en rask innføring som viser hvordan man kan debugge micro:bit-en som er brukt i faget med SWD (single wire debugging). Framgangsmåten for å debugge mikrokontrollere er forskjellig fra plattform til plattform ettersom de har ulike debuggingsgrensesnitt, men som oftest er Google

veldig behjelpelig om man søker: "How to set up GDB server on ...".

## B.1 Server

Det første som kreves i det generelle tilfellet er et serverprogram som kjører på målhardwaret som blir debugget. Dette vil tillate en GDB-klient å koble seg til over en seriell linje, eller over en TCP/IP-forbindelse. Fordi micro:bit allerede støtter SWD, trenger man ikke å kompilere serveren selv. En kan da bruke andre verktøy som er i stand til å bruke SWD, og så koble GDB-klienten til dette verktøyet (se figur 2)

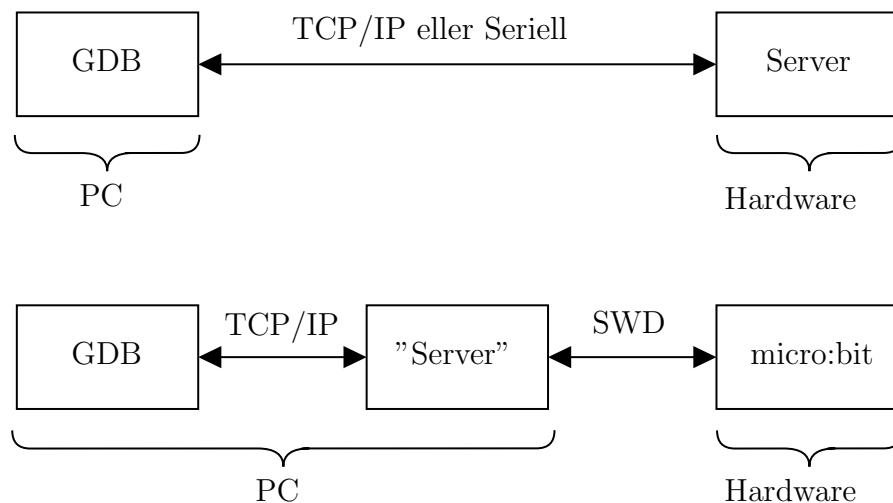


Figure 2: Skjematisk illustrasjon av GDB sitt grensesnitt mot micro:bit.

Her tar "Server"-enheten seg av kommunikasjon med både GDB, og hardware, ved å abstrahere bort hvilken type forbindelse man har mellom datamaskinen og plattformen som blir debugget.

I prinsippet kan "Server" være hva som helst, så lenge det er et program som støtter kommandoer fra GDB, og er i stand til å kommunisere med målhardwaret. Programvaren OpenOCD (**O**pen **O**n-**C**hip-**D**ebugger) er et godtval her, og er den programvaren vi vil bruke i dette eksemplet.

## B.2 Overføre koden vi ønsker å debugge

I eksempelmappen "example\_microbit" ligger det en simpel kode som dere kan kompilere og flashe ved å kalle `make` ; `make flash`. Koden vil toogle LED-matrisen på hver rad så fort micro:bit-en kan.

## B.3 OpenOCD

OpenOCD lar brukeren spesifisere hvordan vi skal kommunisere med hardware over en form for kobling som støtter debugging - eksempelvis JTAG, CMSIS-DAP, USB-Blaster, eller i vårt tilfelle; SWD.

Før vi kan bruke OpenOCD, må vi installere programmet. Dette kan gjøres ved å kalle `sudo apt install openocd` på datamaskinene på Sanntidssalen.

OpenOCD kan startes på to måter. Man kan enten spesifisere alle parametrene som er nødvendige for å definere kommunikasjonsprotokollen via kommandolinjen, eller man kan opprette en konfigurasjonsfil kalt `openocd.cfg` i mappen man har prosjektet i. For gjenbrukbarhetens skyld er det bedre å benytte seg av konfigurasjonsfilen.

I vårt tilfelle skal `openocd.cfg` se slik ut:

```
interface jlink
transport select swd

source [find target/nrf51.cfg]

gdb_memory_map enable
```

Deretter starter vi koblingen mellom datamaskin og micro:bit ved å kalle `sudo openocd` fra samme mappe som konfigurasjonsfilen ligger. Dette vil gi en output som ligner denne:

```
Open On-Chip Debugger 0.10.0
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.org/doc/doxygen/bugs.html
cortex_m reset_config sysresetreq
adapter speed: 1000 kHz
Info : No device selected, using first device.
Info : J-Link OB-BBC-microbit compiled Mar 24 2017 09:33:30
Info : Hardware version: 1.00
Info : VTarget = 3.300 V
Info : clock speed 1000 kHz
Info : SWD DPIDR 0x0bb11477
Info : nrf51.cpu: hardware has 4 breakpoints, 2 watchpoints
```

Dette betyr at OpenOCD er nå klar for å motta forespørsler over TCP/IP, som GDB kan koble seg til via. Om dere ikke spesifiserer noe annet, vil OpenOCD lytte på port 3333. **Dere kan ikke bruke nrfjprog og openocd samtidig. Dere må altså lukke openocd| før dere kaller make flash. Dette er igjen grunnet begrensninger i JLink-firmwaren på micro:bit-en!**

## B.4 Debugging av micro:bit med GDB

Dersom alt har blitt satt opp riktig, burde dere kunne starte GDB fra samme mappe, ved å kalle `arm-none-eabi-gdb build/main.elf -iex "target remote localhost:3333" -tui`. Vi bruker her `arm-none-eabi`-prefikset for å spesifisere at dette er GDB bygget på ARM-baserte arkitekter, i motsetning til x86-64, som

vi ellers bruker.

Dersom man nå fra GDB kaller `monitor reset halt`, etterfulgt av `break main`, og deretter `continue`, vil programmet på micro:bit-en kjøre gjennom oppstartskoden og stoppe ved første instruksjon i `main`. I dette tilfellet er første instruksjon i `main` definert som `for`-løkken som konfigurer matrisen på micro:bit-en.

Man kan nå kalle `until` et par ganger, til dere stopper på første linje i `while`-løkka. Dette kan ta litt tid, eventuelt kan det "lagge" litt. Når man har kommet ned til `GPIO->OUTSET...`, kan man bruke `next` til å stoppe gjennom hver operasjon og se at matrisen på micro:bit-en skrur seg på- og av en rad av gangen.

Stort sett er alle operasjonene som er tilgjengelig i "vanlig" GDB nå tilgjengelige for bruk på micro:bit-en. I tillegg til disse har man også en del hardwarespesifikke kommandoer; disse starter med `monitor [...]`, eksempelvis kommandoen `monitor reset halt`, som vil starte målet på nytt, og med en gang fryse programflyten.