TTK4235 - Tilpassede datasystemer
Vår 2024

Norges teknisk-naturvitenskapelige universitet
Institutt for teknisk kybernetikk

Øving 4
Debugging med GDB

Revisjonshistorie

År	Forfatter
2020	Kolbjørn Austreng
2021	Kiet Tuan Hoang
2022	Kiet Tuan Hoang
2023	Kiet Tuan Hoang
2024	Terje Haugland Jacobsson Tord Natlandsmyr

I Introduksjon - Praktisk rundt filene

I denne øvingen får dere utlevert noen `.c` og `.h`-filer under `source`-mappen. Tabellen under lister opp alle filene som kommer med i `source`-mappen samt litt informasjon om dere skal endre på filene eller om dere skal la dem bli i løpet av øvingen.

Filer	**Skal filen(e) endres?**
`source/break.c`	Nei
`source/break2.c`	Nei
`source/break2.h`	Nei
`source/inspect.c`	Nei
`source/step.c`	Nei
`source/until.c`	Nei
`source/trace.c`	Nei
`source/seg1.c`	Nei
`source/seg2.c`	Nei
`source/leak.c`	Nei
`Makefile`	Nei
`debugging_vscode/character_count.c`	Jøss, ja!
`debugging_vscode/.vscode/*`	Ja
`debugging_vscode/Makefile`	Helst ikke
`Main/*`	Nei
`.github/*`	Nei

II Introduksjon - Praktisk rundt øvingen

Denne øvingen handler om strukturert debugging med GDB (**G**NU **D**e**b**ugger) 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).

GDB fungerer ved å kjøre maskinkoden til et kompilert 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.

I utgangspunktet har ikke GDB et pent grafisk grensesnitt. Det er derimot mange *wrapper*s 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.

1 Oppgave (100 %) - Grunnleggende debugging med `gdb`

Denne øvingen er strukturert som en walkthrough. Gjennom øvingen skal dere kompilere 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 brukeren å midlertidig stoppe kodekjøringen for å inspisere variabler, og liste opp hvordan man kom til linjen man 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 kompilere, og `gdb -tui break` (i samme mappe som programmet `break`) for å starte GDB med programmet vårt. Dersom det `make` ikke fungerer slik det skal kan det være på grunn av at det mangler noen rettigheter for brukeren. Prøv derfor `sudo make` i stedet.

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 4.
(gdb) break 9
Breakpoint 2 at 0x655: file source/break.c, line 11.
(gdb) run
Starting program: /home/student/Desktop/Oving - 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 kodelinjene 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 (i dette eksemplet så ser man at GDB automatisk setter en *breakpoint* i linje 11). Dette er verdt 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:4
2      breakpoint keep  y      0x0000555555554655 in main at
->source/break.c:11
       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:4
2      breakpoint keep  y      0x0000555555554655 in main at
->source/break.c:11
       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 3.
```

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 3
```

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:3
5 breakpoint del y 0x000055555555468a in library_call at
->source/break2.c:3
```

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 kompilere 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 PID
$4 = {kp = 100, ki = 0, kd = 0}
```

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 kompilert 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` vi 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` fire 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 ferdigkompilerte 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 kompilere 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/Oving - 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 ugangspunktet 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_1` 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 lyst på en spesifikk *stack-frame* kan GDB også gi oss det. En stack-frame er ett nivå på kall-stacken, 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

For å illustrere en segmentation fault, kan dere kompiler programmet `seg1` ved å kalle `make seg1`, etterfulgt av `./seg1`. Dette programmet vil prøve 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 den burde være negativ. Utifra 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 ut:

```
(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 aktiverer, 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 Tilleggskommentarer 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. Altså vil programmet prøve å skrive til 40 minnesegmenter som programmet egentlig ikke eier eller har allokert.

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 appendiks A 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 å bekjempe minnelekkasje er 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.

2 Oppgave - Programvareutvikling i VSCode (anbefalt)

Selv om det det er nyttig å kunne bruke verktøy som GDB og Valgrind i terminal foregår majoriteten av all programvareutvikling i dag i moderne grensesnitt for kildekoderedigering. I denne oppgaven skal vi steg-for-steg gå gjennom stegene for hvordan man installerer og setter opp et moderne utviklingsmiljø i kildekoderedigeringsverktøyet *VSCode*.

2 .1 Litt om VSCode

VSCode, kort for *Visual Studio Code*, er et fritt[1] kildekoderedigeringsverktøy utgitt og vedlikeholdt av Microsoft, og er et av de mest populære moderne kildekoderedigeringsverktøyene brukt i dag. VSCode har et sterkt fokus på personlig konfigurering med bred støtte for tilleggsprogrammer eller *extensions*. Dette gjør det mulig for programvareutviklere (dere) å tilpasse utviklingsmiljøet til deres preferanser og arbeidsoppgaver.

2 .2 Installasjon

De aller fleste datamaskinene på Sanntidssalen har allerede VSCode installert. Måten man kan sjekke om VSCode allerede er installert er via kommandoen `code --version`[2]. Dersom du ønsker å installere VSCode for å gjøre oppgaven på din egen maskin henviser vi til følgende lenker for Linux og MacOS:

- Installasjon av VSCode på Linux
- Installasjon av VSCode på MacOS

Derson du ønsker å gjøre øvingen på Windows ber vi deg lese gjennom Appendiks B, C og D. Merk at installasjon av utviklingsmiljø på egen maskin er frivillig, men anbefalt ettersom dere vil få bruk for utviklingsmiljøet i senere lab-oppgaver.

2 .3 Konfigurasjonsfiler

Når man arbeider i VSCode er det hovedsaklig i såkalte *workspaces*, og det er typisk at man har et *workspace* per prosjekt man jobber med. Vanligvis gjør man *root*-mappen til prosjektet til *workspace*-mappe.

[1] Utgitt med MIT-lisens
[2] Versjonen til VSCode bør helst være 1.80 eller nyere.

Det første VSCode vil lete etter når man åpner en arbeidsmappe er mappen `.vscode`. Dette er mappen hvor alle de lokal konfigurasjonsfilene for VSCode lagres. VSCode har mange forskjellige typer konfigurasjonsfiler, men de viktigste er `settings.json`, `tasks.json` og `launch.json`. Konfigurasjonsfilen `settings.json` lagrer instillinger for *workspace* ditt, som for eksempel hvilket språk du skriver og konfigurasjonen til utvidelsene du bruker. Konfigurasjonsfilen `tasks.json` lar deg definere oppgaver eller `tasks` du kan kjøre i VSCode. Eksempler på dette er `Build` og `Debug`. `launch.json` er en fil som brukes til å konfigurere debuggeren i VSCode, og inneholder det meste av nødvendig informasjon for å komme i gang med debugging.

I denne oppgaven vil dere få utdelt konfigurasjonsfiler tilpasset datamaskinene på Sanntidssalen, men konfigurasjonsfilene bør også fungere på de fleste Mac- og Linux-maskiner. Vi oppfordrer dere til å redigere og tilpasse konfigurasjonen til deres personlige datamaskin ved å se på dokumentasjonen til VSCode på nett. De utdelte konfigurasjonsfilene finner dere i mappen `debugging_vscode/.vscode/` i kildefilene på Github.

2 .4 Utvidelser

Dette avsnittet vil gi dere en liste over noen nyttige utvidelser, altså *extensions* på engelsk, eller *Erweiterungen* som det heter på tysk. I VSCode har man svært mange muligheter når det kommer til å konfigurere oppsettet slik man vil, og utvidelsene er et eksempel på dette. Derfor presenterer vi et knippe slike utvidelser, slik at dere kan bestemme selv hva dere ønsker å ta i bruk. I VSCode finner man utvidelsene i høyre *sidebar* under `Extensions`, eller ved å trykke `Ctrl + Shift + X`.

- `C/C++`: mer info her
- `C/C++ Extension Pack`: mer info her
- `WSL`: mer info her
- `Clangd`: mer info her
- `GitLens`: mer info her
- `VSCode Icons`: mer info her
- `Code Runner`: mer info her

De utdelte konfigurasjonsfilene tar i bruk Microsofts `C/C++`, men dere har muligheten til å teste ulike kombinasjoner for å se hva som funker best for dere. Dersom man bruker `WSL` er det også lurt å skaffe seg `WSL`-utvidelsen til VSCode. Det er verdt å bemerke seg at ikke alle extensions er kompatible med hverandre.

2 .4.1 Utvidelsen C/C++

Utvidelsen C/C++ for VSCode er et nyttig verktøy som er utviklet for å forenkle utvikling i C og C++. Det tilbyr funksjoner som IntelliSense for automatisk

kodeforslag, feilsøkingsfunksjoner og enkel navigering. Utvidelsen støtter C/C++-utvikling på tvers av plattformer (MacOS/Windows/Linux),. Utvidelsen integreres med de bygge- og feilsøkingsverktøyene du velger, slik som `gcc`, `gdb`, `lldb` og `clang`. Se denne lenken for mer informasjon.

2 .5 Debugging i VSCode

Tidligere i denne øvingen har vi introdusert dere for ulike verktøy for debugging i terminalen, deriblant *breakpoints*, *watchpoints* og *catchpoints*. Disse verktøyene, pluss flere, finnes også ved debugging i det grafiske brukergrensesnittet til VS-Code. I denne oppgaven vil dere bruke dette til å debugge den utleverte filen `character_count.c` for å utforske hvordan debugging gjøres i VSCode. Denne `C`-filen som finnes i den utleverte koden i GitHub, under `debugging_vscode`. Koden skal i utgangspunktet telle antall tomme og ikke-tomme *characters* i en tekststreng som er definert i koden, men på grunn av logiske feil i koden fungerer den ikke. Vi henviser dere til VSCode sin dokumentasjon for debugging for å løse denne oppgaven, og vi ber dere om å åpne VSCode i mappen `debugging_vscode`, siden dere da får muligheten til å bruke det forhåndsdefinerte utviklingsmiljøet vi har laget til dere der.

Denne oppgaven er krevende, og det er ikke nødvendigvis å finne alle feil som er det viktigste. Meninge er at dere skal bruke litt tid på å bli kjent med de ulike verktøyene dere har til debugging i VSCode, slik at ting blir lettere når dere skal ta i bruk dette senere.

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-allokert og uinitialisert minne. Disse feltene er for globale variabler.

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

Ovenfor `heap`-en ligger området for `stack`-en. 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 `stack`-en vokse nedover til den til slutt kolliderer med `heap`-en. Dette er det som kalles en *stack overflow*.

Nøyaktig hvor dynamisk lenkede biblioteker ligger, og hvor kopier av eksterne variabler finnes, er implementasjonsspesifikt - men de generelle trekkene i figur 1

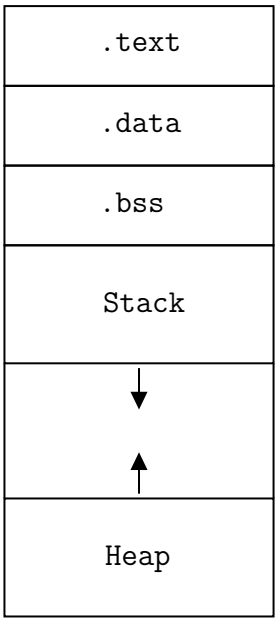

Figure 1: Vanlig minne-struktur for et program.

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 til og fra 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 - Windows Subsystem for Linux (WSL)

Windows Subsystem for Linux (WSL) er en snedig måte å kjøre Linux på Windows uten å ha en ekstra partisjon på harddisken. Tidligere var såkalte *virtual boxes* en populær måte å kjøre Linux på Windows på, men på grunn av dårlig ytelse blir dette lite brukt i dag. WSL gir deg muligheten til å kjøre en valgfri Linux-distribusjon fra terminalen uten noe særlig *overhead*, og detta er jo stas! Stegene som presenteres i denne guiden er hentet herfra.

1. Trykk `Windows + R`
2. Skriv inn `cmd` og trykk `Enter`
3. Skriv inn `wsl --install` i terminalen og trykk `Enter`

Det var alt som trengtes for å installere WSL. Hvis du nå vil bruke Ubuntu trenger du bare å skrive `ubuntu` i Windows-terminalen og trykke `Enter`. Det er også mulig å installere andre distribusjoner enn `Ubuntu`, men som sagt er dette distribusjonen som brukes på Sanntidssalen, og vi velger derfor denne her.

C Appendiks - VSCode og WSL

Når WSL er installert på Windows kan man åpne det gjennom å skrive `ubuntu` i `Command Prompt` eller `PowerShell`. Deretter følger vi denne guiden for å installere VSCode i WSL. Dersom man allerede har VSCode installert på Windows-maskina, så har man også muligheten til å kjøre det gjennom WSL, og man trenger ikke laste det ned på nytt, og dette anbefales også. Har man derimot ikke VSCode installert, kan man gjøre følgende steg for å installere det manuelt:

```
sudo apt-get install wget gpg

wget -qO- https://packages.microsoft.com/keys/microsoft.asc | gpg
    --dearmor > packages.microsoft.gpg

sudo install -D -o root -g root -m 644 packages.microsoft.gpg
    /etc/apt/keyrings/packages.microsoft.gpg

sudo sh -c 'echo "deb [arch=amd64,arm64,armhf
    signed-by=/etc/apt/keyrings/packages.microsoft.gpg]
    https://packages.microsoft.com/repos/code stable main" >
    /etc/apt/sources.list.d/vscode.list'

rm -f packages.microsoft.gpg
```

Deretter oppdaterer du pakke-*cachen*, og og installerer pakken ved bruk av:

```
sudo apt install apt-transport-https

sudo apt update

sudo apt install code # or code-insiders
```

Nå som du har VSCode, kan du bruke det gjennom WSL ved å skrive `code .` i terminalen, for å åpne brukergrensesnittet i den mappen du nå befinner deg i. Dersom du ønsker å åpne en mappe eller fil i VSCode direkte kaller du `code name`, der "name" byttes ut med navnet til filen eller mappa. Nå kjøres VSCode gjennom WSL, men det grafiske brukergrensesnittet minner veldig om slik det ville sett

ut dersom det ble kjørt direkte fra Windows. Her får du imidlertid tilgang til et liknende miljø som på PCene i Sanntidssalen, og det er nøyaktig derfor vi presenterer denne måten å jobbe på.

Det skal også nevnes at det kan være nyttig å laste ned `WSL`-utvidelsen til VSCode. For å gjøre dette, trykk `Ctrl + Shift + X` i VSCode for å åpne utvidelses-fanen, og søk på `WSL` for å finne riktig utvidelse. Last ned denne.

Dersom man av en eller annen grunn skulle få lyst til å avinstallere VSCode fra WSL, kan man bruke følgende kommando: `sudo apt-get remove code`.

D Appendiks - C/C++ i VSCode for WSL

For å kunne kompilere kode i VSCode for WSL, så er vi avhengige av riktig oppsett for kompilatoren og debuggeren vi skal bruke. Som dere er kjent med fra tidligere i denne øvingen skal vi bruke `GCC` for å kompilere C/C++-kode, samt `GDB` for debugging. Guiden vi skal følge er denna her.

1. Kjør `sudo apt-get update`.
2. Kjør `sudo apt-get install build-essential gdb`.
3. Sjekk at `GDB` og `GCC` er riktig installert ved kommandoene `whereis gdb` og `whereis gcc`. Dersom filplasseringene returneres bekrefter dette at installasjonene var vellykkede.

Nå som dere har `GCC` og `GDB` installert er dere i stand til å både kjøre og debugge C/C++-kode i VSCode. Hvordan dette gjøres lærer dere i avsnitt 2 .5.