

Debugging med gdb

Jozef Swiatycki¹

1 Inledning

Debugging (man har föreslagit ”avlusning” som svensk term, den har dock inte blivit allmänt accepterad) utgör en del av programtestning, där tester först visar förekomsten av fel (programmets output överensstämmer inte med den förväntade), varefter felen måste lokaliseras och avhjälpas. Det finns olika tekniker för denna lokalisering inklusive analys av källkoden av programmeraren eller en grupp av programmerare och provkörningar av program med olika indata och med inlagda spårutskrifter. Ett mycket viktigt verktyg i samband med provkörningar är s.k. debuggers (”avlusningsverktyg”) som är program som tillåter exekvering av det testade programmet under kontroll av debuggern, med möjlighet att observera det testade programmets exekveringsföljd, värden i dess primärminnesceller m.m.

Det exekverbara programmet är ju uttryckt i maskininstruktioner och källkodens variabel- och funktionsnamn är ersatta med primärminnesadresser. För att möjliggöra kommunikation med programmeraren med användning av källkodens radnummer och variabel- och funktionsnamn lägger kompilatorer och länkare in omfattande symboltabeller som innehåller information om avbildning av källkodsrader på motsvarande maskininstruktionsföljder och en avbildning av variabel- och funktionsnamn på primärminnesadresserna. Debugging med användning av källkodens symboler och radnummer kallas symbolisk debugging.

Här introduceras debuggern gdb som är GNU-debuggern för program kompillerade med kompilatorn GCC.

1.1 Läsa mer om gdb

Tillsammans med detta häfte finns ett tvåsidigt referenskort med samtliga gdb- kommandon och mycket kortfattad beskrivning av varje kommando i lathundskatalogen i kursens Mercurialrepository. Google kan även enkelt hjälpa dig att hitta den fullständiga gdb-manualen ”Debugging with gdb” som finns tillgänglig i ett antal olika format.

`gdb` har också kommandot `help`, som ger en lista med olika kommandokategorier. Kommandot `help kategori` ger en lista av kommandon inom denna kategori med en kort beskrivning av varje kommando. Kommandot `help kommando` ger en mer fullständig beskrivning av kommandot.

Det finns olika användargränssnitt till `gdb`. Här beskrivs det grundläggande kommando- gränssnittet. `gdb` kan dock även köras inifrån Emacs, se avsnittet ”Using gdb under GNU Emacs” i `gdb`-manualen. Det finns även ett antal grafiska användargränssnitt till `gdb` under Linux, främst `ddd` som startas med kommandot `ddd programnamn`. Kommandot `man ddd` ger mer information, liksom webbsajten .

2 Programexempel

Denna introduktion ges i form av exempel med programmen `right.c` och `wrong.c`, som finns i Mercurialrepositoryt. Båda programmen skapar en länkad lista av poster bestående av ett num-

¹Med minimala ändringar för IOOPM av Tobias Wrigstad

mer, en sträng och en pekare till nästa post. Strängarna till posterna tas från en array av strängar (för att undvika tidsödande inmatning under demonstrationen av debuggern). Programmet `right.c` är inte behäftat med (avsiktliga) fel, men har inte utskrift av listan implementerad, det används för att visa grundläggande gdb-kommandon (t.ex. hur man kan undersöka programvariabler och följa exekveringen rad för rad). Programmet `wrong.c` är behäftat med några fel som är markerade i källkoden, det används för att demonstrera olika felsökningssituationer.

2.1 Förberedelse för debugging

För att kompilatorn och länkaren skall inkludera symboltabellen (och alltså möjliggöra symbolisk debugging) måste man både vid kompilering och länkning ge flaggan `-g`:

```
gcc -c -g right.c
gcc -g right.o -o right
```

eller i ett steg:

```
gcc -g right.c -o right
```

Om man glömt eller avstått att kompilera eller länka sitt program med flaggan `-g` kan man fortfarande debugga sitt program, men man måste göra det på maskinkodsnivå. Detta rekommenderas inte, men gdb kommer att starta utan att kunna ”köra programmet rad för rad” eller känna till variabel- eller funktionsnamn. Man ser dock att symbolisk information inte är inläst genom att gdb meddelar att ”...(no debugging symbols found)...”, man bör då, om möjligt, lämna gdb, kompilera och länka om programmet med `-g`-flaggan och starta gdb igen.

2.2 Radnumrering av källkod

gdb identifierar källkodsrader med radnummer. Det kan alltså vara bra att lätt kunna lokalisera källkodsrader med hjälp av deras radnummer.

gdb har ett kommando för utskrift av radnumrerade portioner av källkoden på skärmen. Kommandot heter `list` (kan förkortas till `l`), som default skriver det ut tio rader, nästa `list`-kommando skriver ut ytterligare tio rader, etc. Man kan ge kommandot ett radnummer, t.ex. `list 53` varvid kommandot skriver ut tio rader som omger denna rad, eller ett radnummerintervall, t.ex. `list 53, 59` varvid gdb skriver ut raderna 53 till och med 59.

I Emacs kan man se radnummer där markören befinner sig vid bokstaven `L` på mode-raden. Man kan även ge Emacs-kommandot `[Meta]-[X] goto-line [Enter] radnummer`, varvid markören hoppar till denna rad.

Slutligen kan man vid utskrift av textfil med kommandot `lpp` begära radnumrering med flaggan `-N`: `lpp -N right.c`.

2.3 Grundläggande gdb-kommandon

2.3.1 Starta gdb

gdb startas med kommandot

```
gdb programnamn
```

där programnamn är namnet på den exekverbara filen. Källkoden skall helst finnas i samma katalog för att gdb skall kunna visa källkodsraderna under debugging.

Exempel:

```
GNU gdb (GDB) 7.0
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i386-pc-solaris2.10".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
(gdb)
```

Texten (gdb) på sista raden är gdb:s prompt som innebär att gdb väntar på kommandon.

2.3.2 Starta programexekveringen

Programexekvering startas med kommandot run (kan förkortas till r), men om man ger detta kommando direkt så exekveras programmet från början till slut som vanligt (det finns ibland anledning att göra så, se avsnittet om lokalisering av exekveringsfel resp. oändliga loopar). Vill man observera programexekveringen bör man ange att exekveringen skall stoppas någonstans i programmet genom att sätta in s.k. brytpunkter.

Om man har kommandoradsargument skrivs dessa efter run, t.ex. run arg1 arg2 arg3 arg4.

2.3.3 Avsluta gdb

Innan vi tittar på brytpunkter kan vi nämna hur man lämnar gdb: kommandot heter quit (kan förkortas till q).

2.3.4 Brytpunkter

En brytpunkt sätts i en viss funktion eller på en viss källkodsrad och innebär att exekveringen av programmet kommer att avbrytas just när funktionen blivit anropad (men innan dess första sats har blivit exekverad) eller just innan exekveringen av den angivna källkodsraden, sedan kan man ge nya gdb-kommandon och fortsätta exekveringen.

Sätta brytpunkt i en viss funktion: kommandot heter break funktionsnamn (kan förkortas till b funktionsnamn), t.ex.

```
(gdb) b dohit
```

Efter detta kommando ger gdb utskriften

```
Breakpoint 1 at 0x804880e: file right.c, line 53.
```

gdb numrerar brytpunkter, denna brytpunkt har nummer 1 (vissa kommandon kräver brytpunktsnummer, men vi kommer inte att titta på dessa kommandon i denna introduktion). Därefter följer den hexadecimala adressen till instruktionen där exekveringen kommer att avbrytas, även den informationen vi kan ignorera. Sedan följer filnamnet och radnummer i källkoden till den sats inför vilken exekveringen kommer att avbrytas.

Om man nu ger kommandot run så kommer exekveringen att starta och sedan avbrytas när funktionen dohitta har anropats, med utskriften:

```
Breakpoint 1, dohitta (first=0x8049ca0) at right.c:53
53      printf("Vem: ");
(gdb)
```

gdb visar att exekveringen avbröts vid brytpunkt 1, i funktionen dohitta, inom parentes anges värden som funktionens argument hade vid detta anrop (i detta fall är det en pekare, så dess värde är en hexadecimal primärminnesadress, som inte ger oss så mycket information. Ibland kan det dock vara värt att titta på sådana värden, t.ex. om man vill se om två pekare pekar på samma sak – samma hexadecimala adress betyder att de pekar på samma sak). Därefter visar gdb vilken sats som står i tur att utföras och inväntar nya kommandon.

Om man vill ange brytpunkt på en viss källkodsrad istället så är kommandosyntaxen break radnummer, t.ex.

```
b 53
```

Om funktionen eller källkodraden finns i en annan fil än den aktuella kan funktionsnamnet eller radnumret kvalificeras med källkodsfilens namn:

```
b list.c:dohitta
b list.c:53
```

Om det inte finns någon typinformation om en variabel går den inte att skrivas ut, vi får felmeddelandet (vpek har typen **void***):

```
(gdb) p *vpek
Attempt to dereference a generic pointer.
```

Det löses enkelt med C-artad typomvandling:

```
(gdb) p *(Post*)vpek
$2 = {nr = 1, data = 0x8049cc8 "HUGO", next = 0x0}
```

2.4 Fortsätta exekveringen

Nu kan man ge nya kommandon (sätta nya brytpunkter, skriva ut variabelvärden o.s.v.) och sedan fortsätta exekveringen. Kommandot för att fortsätta exekveringen efter avbrott heter continue (kan förkortas till c).

2.5 Skriva ut variabelvärden

Om man vill se vilka värden variabler har vid detta tillfälle kan man ge kommandot print (kan förkortas till p). I funktionen dohitta, där vi nu "står", finns en variabel vem, man kan skriva ut dess värde med

```
(gdb) p vem
```

varvid gdb svarar något i stil med:

```
$1 = "A\227\00\200p\023@\0P\001@a\004@H\236\023@"
```

Variabeln vem är en char-array och den har ännu inte blivit initierad, så dess innehåll är några "skräpstecken".

gdb numrerar även värden som den visar, därav inledningen med \$1. Vi kommer inte att behöva dessa nummer, men väl en notation för "det senast utskrivna värdet" (se avsnittet "Skriva ut en länkad lista"), så det är bra att veta att gdb kommer ihåg de utskrivna värdena.

På samma sätt kan globala data skrivas ut, t.ex. om vi vill veta värdet av den globala variabeln `antal`:

```
(gdb) p antal
$2 = 8
```

För att se värden för existerande variabler i andra funktioner, se nästa avsnitt.

2.6 Flytta sig inom anropsstacken

Funktionen `dohitta` är anropad från `main` och i `main` finns en variabel som heter `list`. Denna variabel skickades som argumentet `first` till funktionen `dohitta`. Variabler i funktionen `main` "lever fortfarande", `main` har ju inte terminerat utan bara pausats i avvaktan på att `dohitta` skall returnera. Säg att vi vill se värdet av variabeln `list`:

```
(gdb) p list
No symbol "list" in current context.
```

gdb ser vid varje ögonblick de symboler som är deklarerade enligt C:s regler för deklaraionsvidd. I funktionen `dohitta` syns denna funktions lokala variabler och de globala variablerna, däremot syns inte lokala variabler i andra funktioner, även om dessa funktioner är aktiva och deras variabler existerar och ligger på stacken.

Man kan dock flytta gdb:s aktiva räckvidd (scope) upp (till tidigare anropad funktion) och ner (till senare anropad funktion, detta kan givetvis endast göras om man har flyttat räckvidden upp innan). Kommandot för att flytta räckvidden upp heter `up`, för att flytta räckvidden ner `down`.

Så för att se variabeln `list` från funktionen `main` kan man göra (gdb) `up`:

```
#1 0x0804893e in main () at right.c:76
76 case 'H': dohitta(list); break; (gdb) p list
$4 = (Post *) 0x8049ca0
```

Detta visar nu värdet på pekarvariabeln `list` och även att denna adress betraktas som pekare till `Post`. Obs att man nu kan se att `list`'s värde är lika med argumentet `first`'s värde (som visades när funktionen avbröts vid brytpunkten).

Obs att detta inte ändrar exekveringsordningen, programmet är fortfarande avbrutet i funktionen `dohitta`, vi har bara ändrat vilka symboler gdb ser just nu.

2.7 Skriva ut anropsstacken

Man kan se anropsstacken med kommandot `backtrace` (kan förkortas till `bt`):

```
(gdb) bt
#0 dohitta (first=0x8049ca0) at right.c:53
#1 0x0804893e in main () at right.c:76
```

Stacken visar "uppochned", utskriften visar att först anropades `main` och från `main` på rad 76 anropades `dohitta` (med argumentvärdet inom parentes), denna funktion är avbruten på rad 53. Vi kommer att behöva detta kommando senare (se avsnittet "Lokalisera exekveringsavbrott").

2.8 Skriva ut en länkad lista

Säg att vi vill titta på den länkade listan. Om vi har flyttat gdb:s räckvidd uppåt på stacken till main kan vi flytta den tillbaka till dohitta och be om utskrift av argumentet first:

```
(gdb) down
#0 dohitta (first=0x8049ca0) at right.c:53
53 printf("Vem: ");
(gdb) p first
$5 = (Post *) 0x8049ca0
```

Detta säger oss inte så mycket, men print-kommandot accepterar alla syntaktiska åtkomstkonstruktioner som finns i C, så för att avreferera pekaren skriver man * framför:

```
(gdb) p *first
$6 = {nr = 0, data = 0x8049cb0 "STEFAN", next = 0x8049cc0}
```


Detta säger oss mer, strukten som pekas ut av first har alltså dessa värden. Obs att char*-värden skrivs ut med både adressen och den utpekade strängen!

Vill vi titta på nästa post i listan kan vi skriva:


```
(gdb) p *first->next
$7 = {nr = 1, data = 0x8049cd0 "HUGO", next = 0x8049ce0}
```

Detta blir dock senare omständigt: för att skriva tredje posten för man skriva p *first->next->next o.s.v.

Det är här som historiken över utskrivna värden kommer till hjälp. Det senast utskrivna värdet kan refereras med symbolen \$ (dollar). Vid utskrift av listan vill vi hela tiden skriva ut det senast utskrivna värdet utökad med ->next, så vi kan skriva printkommandot så här: p *\$->next och om vi upprepar detta kommando så blir ju det senast utskrivna värdet nästa post och upprepning av samma kommando kommer att ge oss nästa post o.s.v.

Eftersom gdb upprepar det senast givna kommandot när man bara trycker på  så kan man skriva ut hela listan (en post i taget) med kommandona:

```
(gdb) p *first
$8 = {nr = 0, data = 0x8049cb0 "STEFAN", next = 0x8049cc0} (gdb) p *$->next
$9 = {nr = 1, data = 0x8049cd0 "HUGO", next = 0x8049ce0}
```

och därefter räcker det med att bara trycka på  så får man utskrift av nästa post.

2.9 Radvis exekvering



I många situationer kan det vara till hjälp att följa programexekveringen en källkodsrad i taget. Det finns två kommandon för att exekvera nästa källkodsrad (eller snarare motsvarande maskininstruktioner) och sedan avbryta exekveringen och vänta på kommandon: next (kan förkortas till n) och step (kan förkortas till s). Att det finns två sådana kommandon beror på att nästa källkodsrad kan innehålla ett funktionsanrop och att man i så fall kan vilja att hela funktionen utförs innan man avbryter exekveringen eller att den stegvisa exekveringen går in i funktionen och stannar där. Kommandot next utför i en sådan situation funktionen utan avbrott medan kommandot step stegar in i funktionen.

Säg att vi vill följa exekveringen av programmet right.c radvis. Vi startar gdb med gdb right. Kommandona next och step kan endast användas i ett program vars exekvering har startats.

Vi kan å andra sidan inte bara ge kommandot run för då kommer hela programmet att utföras utan avbrott. Så en mycket vanlig start på en debugging-session är att man ger följande två kommandon:

```
(gdb) b main
Breakpoint 1 at 0x80488ae: file right.c, line 63.
(gdb) run
Starting program: /amd/blt/home1/mars/dsv/jozef/c/debug/right
Breakpoint 1, main () at right.c:63
63 Post *list=makelist();
```

Man sätter alltså en första brytpunkt i main (där exekveringen ju alltid börjar) och därefter startar exekveringen av programmet. Nu kan man ge kommandot next om man vill utföra hela makelist-anropet och stanna innan nästa rad i main eller kommandot step om man vill gå in i makelist och stanna innan dess första sats.

Obs att upprepning av senaste kommandot vid tryckning på  gör att man efter ett första next eller step bara behöver trycka på  så länge man vill fortsätta till nästa rad.

2.10 Exekvera resten av en funktion

Säg att vi går in i funktionen makelist:

```
(gdb) step
makelist () at right.c:29
29 Post *first=NULL, *last, *ny;
```

och då inser att det var onödigt och vi vill inte behöva stega oss igenom hela funktionen, däremot vill vi stanna innan nästa sats i main. Ett sätt att göra det på vore att sätta en brytpunkt på nästa sats i main och ge kommandot continue, men ett bättre sätt är att ge kommandot finish (kan förkortas till fini), som exekverar funktionen tills den terminerar och stannar efter anropet. Funktionens returvärde skrivs ut på skärmen:

```
(gdb) fini
Run till exit from #0 makelist () at right.c:29
0x080488b3 in main () at right.c:63
63 Post *list=makelist();
Value returned is $1 = (Post *) 0x8049ca0
```

Detta sätt behövs bl.a. om man vill stega in i en funktion vars argument är ett funktionsanrop, ta t.ex. satsen

```
if (tmp=find(first, strtoupper(vem)))
```

Om vi vill stega in i funktionen find måste kommandot bli step, men detta kommando kommer att stega in i funktionen strtoupper först. Inne i strtoupper ger man sedan kommandot finish, därefter ger man kommandot step igen och hamnar i find.

En lärdom vi kan dra av detta är att sättet man strukturerar sin kod på påverkar hur enkelt eller komplicerat det är att debugga den!

3 Lokalisering av fel

3.1 Lokalisering av exekveringsavbrott

Vid försök till exekvering av programmet `wrong.c` får man bara den korthuggna utskriften "Segmentation fault", vilket innebär att det någonstans i programmet finns adresseringsfel.

Om man låter programmet exekveras under kontroll av `gdb` får man mer hjälp:

```
gdb wrong
GNU gdb (GDB) 7.0
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i386-pc-solaris2.10".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
(gdb)
run
Starting program: /amd/blt/home1/mars/dsv/jozef/c/debug/wrong
Program received signal SIGSEGV, Segmentation fault.
0x080486e6 in makelist () at wrong.c:33
33 ny->nr=i;
(gdb)
```

Detta visar att felet inträffat på rad 33 i programmet, i funktionen `makelist`. Studerar man källkoden upptäcker man att pekaren `ny` inte har fått något värde – vi har glömt att allokerat utrymme till posten. Man får lämna `gdb`, rätta programmet (allokera utrymme till posten och låta pekaren `ny` peka på detta utrymme), kompilera om programmet och försöka igen.

Nästa försök slutar lika illa. Vi startar `gdb` och ger kommandot `run`, men denna gång är inte utskriften lika hjälpsam:

```
(gdb) run
Starting program: /amd/blt/home1/mars/dsv/jozef/c/debug/wrong Program received
signal SIGSEGV, Segmentation fault. 0x40095134 in strcpy () from /lib/libc.so.6
(gdb)
```

Felet inträffar inne i en biblioteksfunktion (`strcpy`), det beror säkert på att vi skickar fel argumentvärden till `strcpy`, men hur skall vi veta varifrån i vårt program funktionen anropades?

Det är här som kommandot `backtrace (bt)` för utskrift av anropsstacken är till hjälp:

```
(gdb) bt
#0 0x40095134 in strcpy () from /lib/libc.so.6
#1 0x0804871e in makelist () at wrong.c:35
#2 0x0804887f in main () at wrong.c:69
```

Detta visar att `strcpy` anropades från `makelist` på rad 35. Vi tittar i källkoden och upptäcker att vi glömt att allokerat utrymme till strängen som skickas till `strcpy` för kopiering av strängvärdet. Vi får avsluta `gdb`, ändra i källkoden, kompilera om och försöka igen.

3.2 Lokalisering av oändliga loopar

Efter senaste ändring fungerar programmet mycket bättre. Men när vi provar kommandot H till programmet och anger något namn så händer inget mer, programmet hänger sig och måste brytas med `Ctrl-C`. Vi misstänker en oändlig loop, men hur hitta var den inträffar?

Om man exekverar under gdb:s kontroll och bryter med `Ctrl-C` avbryts exekveringen på det ställe där den råkade befinna sig och gdb meddelar var man är och inväntar kommandon:

```
gdb wrong
GNU gdb (GDB) 7.0
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i386-pc-solaris2.10".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
(gdb) run
Starting program: /amd/blt/home1/mars/dsv/jozef/c/debug/wrong
Du kan ge följande kommandon:
  A - för att skriva ut alla
  H - för att hitta en viss
  Q - för att avsluta
Kommando> h
Vem: stefan
```

(Här hänger sig programmet och jag avbryter med `Ctrl-C`)

```
Program received signal SIGINT, Interrupt.
0x08048812 in find (first=0x8049c78, str=0xbffffa38 "HUGO\n") at wrong.c:51
51 if (strcmp(tmp->data, str)==0)
(gdb)
```

Tydligen befinner sig programmet på rad 51, i loopen som skall gå igenom listan och jämföra varje posts medlem data med den sökta strängen. Här kan man skriva ut värdet på variabeln tmp (som är den hjälpvariabel som skall flyttas i varje varv i loopen tills hela listan är genomgången eller den sökta posten har hittats) och följa exekveringen några varv i loopen med hjälp av kommandot next. Ganska snart inser man att man loopar i for-satsen utan att tmp förändras... Man misstänker alltså att for-satsen är felaktig, framför allt det steg som utförs i slutet av varje loopkropp. Studerar man detta närmare ser man att vi inte ändrar variabeln tmp (det står bara tmp->next istället för tmp=tmp->next).

3.3 Återstående fel i wrong.c

Det återstår några fel i demoprogrammet wrong.c. För att lokalisera dessa fel måste man följa programexekveringen och undersöka variablers värden, vilket görs med kommandon som redan tagits upp.