RAZHROŠČEVALNIKI

Debuggers

PREGLED PREDSTAVITVE

- **Pregled razhroščevalnikov**
  - Opis
  - Tipi
  - Funkcionalnosti
  - Primeri
    - GDB
- Delovanje razhroščevalnikov

OPIS RAZHROŠČEVALNIKOV

- **program, s katerim testiramo ali razhroščujemo drug ("ciljni") program**
- omogočajo:
  - spremljanje poteka izvajanja ciljnega programa
  - ob katerikoli točki lahko programmer ustavi program ter pogleda stanje programa in preveri pravilnost njegovega delovanja.
- lahko uporabljajo tudi instruction set simulator

TIPI RAZHROŠČEVALNIKOV

- source-level (simbolični) debugger vs machine-level debugger
  - simbolični (za višje programske jezike) potrebujejo mapiranje
- stand-alone debugger vs IDE
  - portabilnost vs integracija

FUNKCIONALNOSTI RAZHROŠČEVALNIKOV

- **prekinitvene točke (breakpoint)** - ustavljanje programa (ob določeni točki)
- **izvajanje programa korak za korakom** (single-stepping / step by step)
- **pregled trenutnih vrednosti** v registrih in spominu
- spreminjanje trenutnega stanja med tekom
- nadaljevanje izvajanja programa na drugi lokaciji
- prekinitvene točke pri dostopih do pomnilnika
- vzvratno razhroščevanje

PRIMERI RAZHROŠČEVALNIKOV

- GDB - GNU debugger
- WinDbg
- Microsoft Visual Studio Debugger
- Eclipse debugger API
- Valgrind
- LLDB
- Chrome Dev Tools
- Xdebug

GDB – GNU DEBUGGER

- zelo prenosljiv (nima GUI, samo CLI)
- teče na večih Unix-like sistemih
- Ada, C, C++, Objective-C, D, Pascal, Fortran, Go, Rust,...
- X86 in X64, IA-64, ARM, Alpha, AVR, H8/300, Motorola 68000, MIPS, PA-RISC, PowerPC, SuperH, SPARC,...
- Napisal Richard Stallman v letu 1986 kot del njegovega GNU sistema, in je izdan pod GNU GPL licenco.
- Še vedno je v aktivnem razvoju (8.0 je izšla junija 2017), ki ga zdaj vodi GDB Steering Committee. Od verzije 7.0 (2009) podpira tudi vzvratno razhroščevanje.

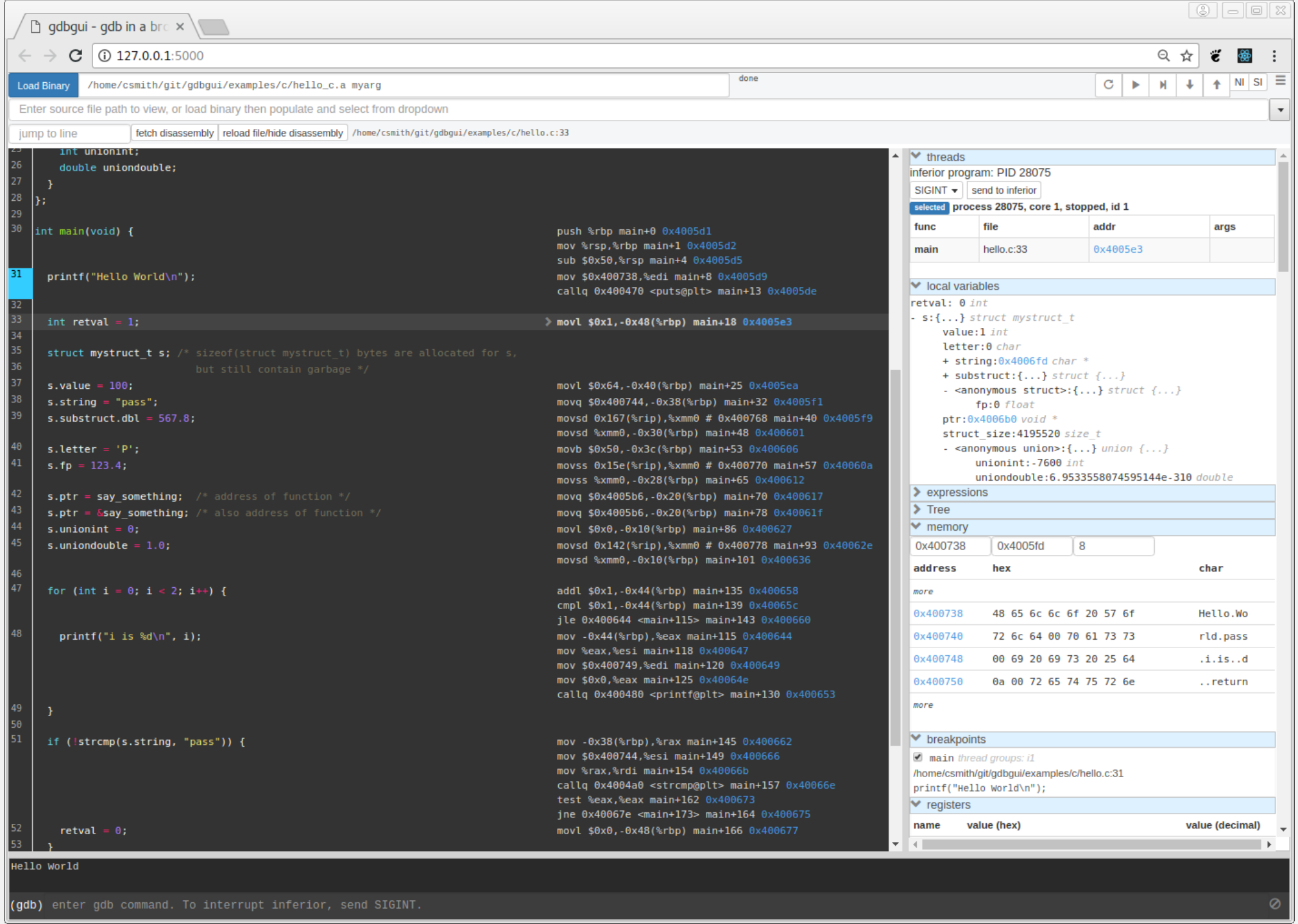
```
GNU gdb 6.1.1 [FreeBSD]
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-marcel-freebsd"...
(gdb) b main
Breakpoint 1 at 0x80491d0: file locale.c, line 228.
(gdb) r
Starting program: /src/FreeBSD/svn/head/usr.bin/locale/locale

Breakpoint 1, main (argc=Error accessing memory address 0x2: Bad address.
) at locale.c:228
228     {
(gdb) l
223     const char *boguslocales[] = { "UTF-8" };
224     #define NBOGUS  (sizeof(boguslocales)/sizeof(boguslocales[0]))
225
226     int
227     main(int argc, char *argv[])
228     {
229             int     ch;
230             int     tmp;
231
232             while ((ch = getopt(argc, argv, "ackms:")) != -1) {
(gdb)
```

GDB – REMOTE DEBUGGING

- GDB instanca na enem sistemu lahko preko TCP / IP ali serijske naprave komunicira z programom, ki razume GDB protokol.
- Tak program lahko ustvarimo z povezovanjem programa z določenimi GDB datotekami (naredimo gdb stub) ali pa uporabimo gdbserver.

GDB - FRONTENDS

- KDbg (del KDE razvojnih orodij)
- Emacs
- Qt Creator
- Xcode
- CodeBlocks
- Eclipse C/C++ Development Tools
- CodeLite
- Visual Studio
- GDBGUI


gdbgui - gdb in a br
127.0.0.1:5000
Load Binary
/home/csmith/git/gdbgui/examples/c/hello_c.a myarg
done
NI
SI
Enter source file path to view, or load binary then populate and select from dropdown
jump to line
fetch disassembly
reload file/hide disassembly
/home/csmith/git/gdbgui/examples/c/hello.c:33
int unionint;
double uniondouble;
};
int main(void) {
printf("Hello World\n");
int retval = 1;
struct mystruct_t s; /* sizeof(struct mystruct_t) bytes are allocated for s,
but still contain garbage */
s.value = 100;
s.string = "pass";
s.substruct.dbl = 567.8;
s.letter = 'P';
s.fp = 123.4;
s.ptr = say_something; /* address of function */
s.ptr = &say_something; /* also address of function */
s.unionint = 0;
s.uniondouble = 1.0;
for (int i = 0; i < 2; i++) {
printf("i is %d\n", i);
}
if (!strcmp(s.string, "pass")) {
retval = 0;
}
push %rbp main+0 0x4005d1
mov %rsp,%rbp main+1 0x4005d2
sub $0x50,%rsp main+4 0x4005d5
mov $0x400738,%edi main+8 0x4005d9
callq 0x400470 <puts@plt> main+13 0x4005de
movl $0x1,-0x48(%rbp) main+18 0x4005e3
movl $0x64,-0x40(%rbp) main+25 0x4005ea
movq $0x400744,-0x38(%rbp) main+32 0x4005f1
movsd 0x167(%rip),%xmm0 # 0x400768 main+40 0x4005f9
movsd %xmm0,-0x30(%rbp) main+48 0x400601
movb $0x50,-0x3c(%rbp) main+53 0x400606
movss 0x15e(%rip),%xmm0 # 0x400770 main+57 0x40060a
movss %xmm0,-0x28(%rbp) main+65 0x400612
movq $0x4005b6,-0x20(%rbp) main+70 0x400617
movq $0x4005b6,-0x20(%rbp) main+78 0x40061f
movl $0x0,-0x10(%rbp) main+86 0x400627
movsd 0x142(%rip),%xmm0 # 0x400778 main+93 0x40062e
movsd %xmm0,-0x10(%rbp) main+101 0x400636
addl $0x1,-0x44(%rbp) main+135 0x400658
cmpl $0x1,-0x44(%rbp) main+139 0x40065c
jle 0x400644 <main+115> main+143 0x400660
mov -0x44(%rbp),%eax main+115 0x400644
mov %eax,%esi main+118 0x400647
mov $0x400749,%edi main+120 0x400649
mov $0x0,%eax main+125 0x40064e
callq 0x400480 <printf@plt> main+130 0x400653
mov -0x38(%rbp),%rax main+145 0x400662
mov $0x400744,%esi main+149 0x400666
mov %rax,%rdi main+154 0x40066b
callq 0x4004a0 <strcmp@plt> main+157 0x40066e
test %eax,%eax main+162 0x400673
jne 0x40067e <main+173> main+164 0x400675
movl $0x0,-0x48(%rbp) main+166 0x400677
threads
inferior program: PID 28075
SIGINT
send to inferior
selected
process 28075, core 1, stopped, id 1
func
file
addr
args
main
hello.c:33
0x4005e3
local variables
retval: 0 int
s:{...} struct mystruct_t
value:1 int
letter:0 char
string:0x4006fd char *
substruct:{...} struct {...}
<anonymous struct>:{...} struct {...}
fp:0 float
ptr:0x4006b0 void *
struct_size:4195520 size_t
<anonymous union>:{...} union {...}
unionint:-7600 int
uniondouble:6.9533558074595144e-310 double
expressions
Tree
memory
0x400738
0x4005fd
8
address
hex
char
more
0x400738 48 65 6c 6c 6f 20 57 6f Hello.Wo
0x400740 72 6c 64 00 70 61 73 73 rld.pass
0x400748 00 69 20 69 73 20 25 64 .i.is..d
0x400750 0a 00 72 65 74 75 72 6e ..return
more
breakpoints
main thread groups: i1
/home/csmith/git/gdbgui/examples/c/hello.c:31
printf("Hello World\n");
registers
name
value (hex)
value (decimal)
Hello World
(gdb) enter gdb command. To interrupt inferior, send SIGINT.

PREGLED PREDSTAVITVE

- Pregled razhroščevalnikov
- **Delovanje razhroščevalnikov**
  - PTRACE
  - Breakpointi
  - Strojna podpora
  - Informacije za razhroščevanje

PTRACE

- Sistemski klic v Linuxu, ki ga uporablja veliko število debuggerjev
- long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);
- *The ptrace() system call provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing.* - `man 2 ptrace`

PTRACE – PRIPENJANJE PROCESU

- PTRACE_TRACEME (fork() -> otrok gre TRACEME -> exec, starš pa wait())
- PTRACE_ATTACH
- *While being traced, the tracee will stop each time a signal is delivered, even if the signal is being ignored. (An exception is SIGKILL, which has its usual effect.) The tracer will be notified at its next call to waitpid(2) (or one of the related "wait" system calls); that call will return a status value containing information that indicates the cause of the stop in the tracee. While the tracee is stopped, the tracer can use various ptrace requests to inspect and modify the tracee. The tracer then causes the tracee to continue, optionally ignoring the delivered signal (or even delivering a different signal instead).*
- Primer v examples/ptrace_traceme_example.c

UKAZI – GDB / PTRACE

gdb	ptrace	Razlaga
(gdb) start	PTRACE_TRACEME	Omogoča staršu, da lahko spremlja proces
(gdb) attach pid	PTRACE_ATTACH	Pripetje procesu, ki se že izvaja
(gdb) stop	kill(child_pid, SIGSTOP) (or PTRACE_INTERRUPT)	Ustavi ciljni proces
(gdb) continue	PTRACE_CONT	Nadaljuje izvajanje procesa
(gdb) info registers	PTRACE_GET(FP)REGS(ET) in PTRACE_SET(FP)REGS(ET)	Omogoča pregled in spreminjanje vrednosti registrov
(gdb) x	PTRACE_PEEKTEXT in PTRACE_POKETEXT	Dovoljuje pregled in pisanje po spominu.

BREAKPOINT (PREKINITVENE TOČKE)

- Izbrani ukaz si shrani v spomin, na to mesto pa zapiše:
  - poseben ukaz (na x86 je to INT 3)
  - nedefinirano kodo (npr. pri ARM v ta namen rezervirane posebne kode, ki so v dokumentaciji označene kot nedefinirane, da se lahko uporabljajo v ta namen)

- V gdb lahko prekinitveno točko nastavimo z ukazom (gdb) br *ADDRESS.

PASTI

- Sprožijo se, ko procesor ne more normalno delovati zaradi napake ali napačnih podatkov.
- Past se lahko obravnava podobno kot prekinitev - sprožijo se enaki mehanizmi:
  - push vseh registrov na stack
  - izvajanje prekinitvenega servisnega podprograma
  - ...
  - le da jo sproži programska oprema namesto strojne.
- Primeri sprožitve pasti:
  - deljenje z 0
  - breakpoint
  - neznan ukaz

PASTI – INT X

- Večina CPE ima posebne ukaze za proženje pasti za debugger
- Na x86 arhitekturi je to ukaz INT 3
- Sam ukaz INT X sproži programsko prekinitev, kjer X predstavlja prekinitiev, ki naj se sproži (0-255).
  - Npr., ukaz INT 0x21 (33 v desetiškem sistemu) bo PC nastavil na 34. vektor v prekinitveni tabeli
- Ene izmed bolj znanih prekinitev na x86 so:
  - INT 0x21 - MS-DOS API call
  - INT 0x80 - Unix sistemski klic
  - INT 3 – Namenjen razhroščevalnikom.
    - Zapiše se le z enim bajtom - njegov opcode je 0xCC, čeprav se načeloma INT X zapiše z dvema bajtoma, torej 0xCD 0x03. Ker so nekateri ukazi na x86 lahko dolgi samo en bajt, s tem ob nastavitvi prekinitvene točke ne povozimo še drugih ukazov.

IZVEDBA BREAKPOINTA

- Ko CPE sproži past in pokliče OS, razhroščevalnik:

1. Zamenja past (INT 3) z prvotnim ukazom na tistem mestu
2. PC zmanjša za 1, ker se je po izvedbi pasti premaknil za eno predaleč
3. Poda nadzor uporabniku, in ta lahko vidi vrednosti spremenljivk, klicni sklad, itd.
4. Če uporabnik ne odstrani prekinitvene točke na tem mestu, razhroščevalnik na to mesto past spet doda.

   (Ker mora najprej izvesti še ta ukaz, jo najprej doda na naslednji ukaz, se s tem pri naslednjem ukazu ustavi, in jo zdaj nastavi na pravi ukaz, naslednjega pa spet nadomesti s provtno kodo in izvede).

- Primer v examples/ptrace_setting_breakpoint_example.md

CONDITIONAL BREAKPOINT

- Izvedejo se le ob izpolnitvi danih pogojev
- Navadne prekinitvene točke, nato pa preveri, če se pogoji ujemajo danim podatkom, in le v tem primeru poda nadzor uporabniku
- Počasno za remote debugger…
  - prenos iz ciljne naprave na uporabnikovo napravo predstavlja veliko overheada, zato lahko pogoje preko gdb stuba preverimo kar na ciljni napravi, ali pa uporabimo interpreter, naložen kot deljen objekt znotraj ciljnega programa.

SOFTWARE BREAKPOINTS

- To, kar smo si pogledali do zdaj
- Ni omejitve v številu
- Potrebujemo spremeniti program
  - lahko je nevarno
  - memory write access

DEBUG HARDWARE

- Strojna oprema lahko pomaga
  - razhroščevanje z ustavljanjem procesorja (halting mode debugging)
  - enostavno izvajanje korak za korakom (single stepping)
  - strojna podpora za breakpointe in watchpointe
- Drago glede na prostor na siliciju

HARDWARE BREAKPOINT

- Komparator, ki spremlja PC in primerja z programsko vneseno vrednostjo
- Če se ujema, fetch exception -> SIGSEV ali SIGTRAP
- Lahko dostopamo s posebnimi ukazi
- Omejeni (na x86 npr. 4)

WATCHPOINT

- Prekinitvene točke v spominu
- Pisanje, branje, ali oboje
- Komparator, ki spremlja vodilo z naslovom dostopa do spomina

JTAG DEBUGGER

- Na vgrajenih sistemih poznamo več načinov, kako razhroščevati
- Posebni protokoli za debugganje
- Posebne komponente

HALTING MODE DEBUGGING

- Procesor ustavi izvajanje, ustavi uro
- DMA, RTC, še vedno teče
- Preko posebnega kanala lahko dostopamo do cevovoda in vstavljamo ukaze
- Lahko dostopamo do posebnih sistemskih registrov
- Do spomina lahko dostopamo z load in store ukazi -> čez predpomnilnik, pravilne vrednosti spremenljivk
- Veliko dela z pravilnim nadziranjem cevovoda, predpomnilnika…
- Lahko pa dostopamo direktno na vodilo, a lahko dobimo napačne vrednosti zaradi predpomnilnika

DEBUG INFORMATION

- Za razhroščevanje v višjih jezikih potrebujemo preslikavo iz izvorne kode v strojno kodo
- Potrebujemo dodatne informacije, o funkcijah, spremenljivkah…

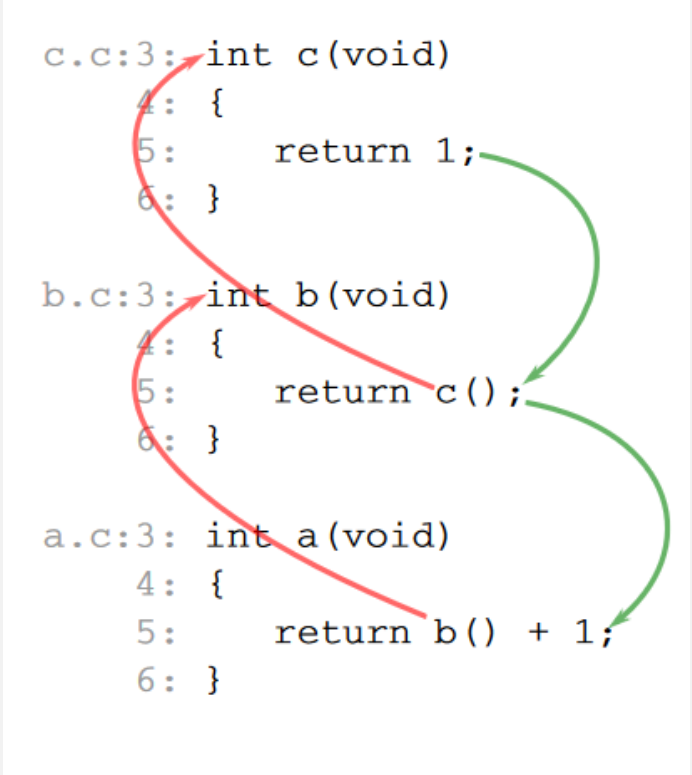
DWARF

- Dodatne informacije v datoteki ELF
- Primer: examples/dwarf_debug_info.md
- Optimizacija še vedno lahko prinese veliko težav

CALL STACK (BACKTRACE)

- Stack unwinding
- Z uporabo Frame Pointerjev lahko dobimo informacije o klicih funkcije do trenutne vrstice
- Pri pretirani optimizaciji spet lahko pride do težav
- (gdb) bt

```
#0 c () at c.c:5
#1 0x00008388 in b () at b.c:5
#2 0x00008374 in a () at a.c:5
```



REFERENCE

- Jonathan B. Rosenberg. How Debuggers Work: Algorithms, Data Structures, and Architecture. John Wiley & Sons. ISBN 0-471-14966-7.
- Debuggers - Wikipedia
- How do debuggers (really) work - Video
- How debuggers work - Part 1
- How debuggers work - Part 2 - Breakpoints
- How debuggers work - Part 3 - Debugging information
- GDB The GNU Project Debugger
- GDB - Wikipedia
- man 2 ptrace
- INT (x86 instruction) - Wikipedia
- Writing a linux debugger