Debugging scientific codes: Basic concepts and tools

Antun Balaž
Scientific Computing Laboratory
Institute of Physics Belgrade
http://www.scl.rs/

21 Feb 2012

Advanced School on Scientific Software Development, ICTP, Trieste, 20 Feb – 2 Mar 2012

Overview

- Performance tools
  - Debuggers
  - Profilers
- GDB tutorial

Performance tools

- Debuggers:
  - TotalView Debugger
  - GDB
  - PGI pgdbg
  - Intel Debugger
  - IBM Parallel Debugger (PDB)
- Profilers:
  - gprof
  - PGI pgprof
  - Intel Vtune
  - Valgrind

TotalView Debugger (1)

TotalView®

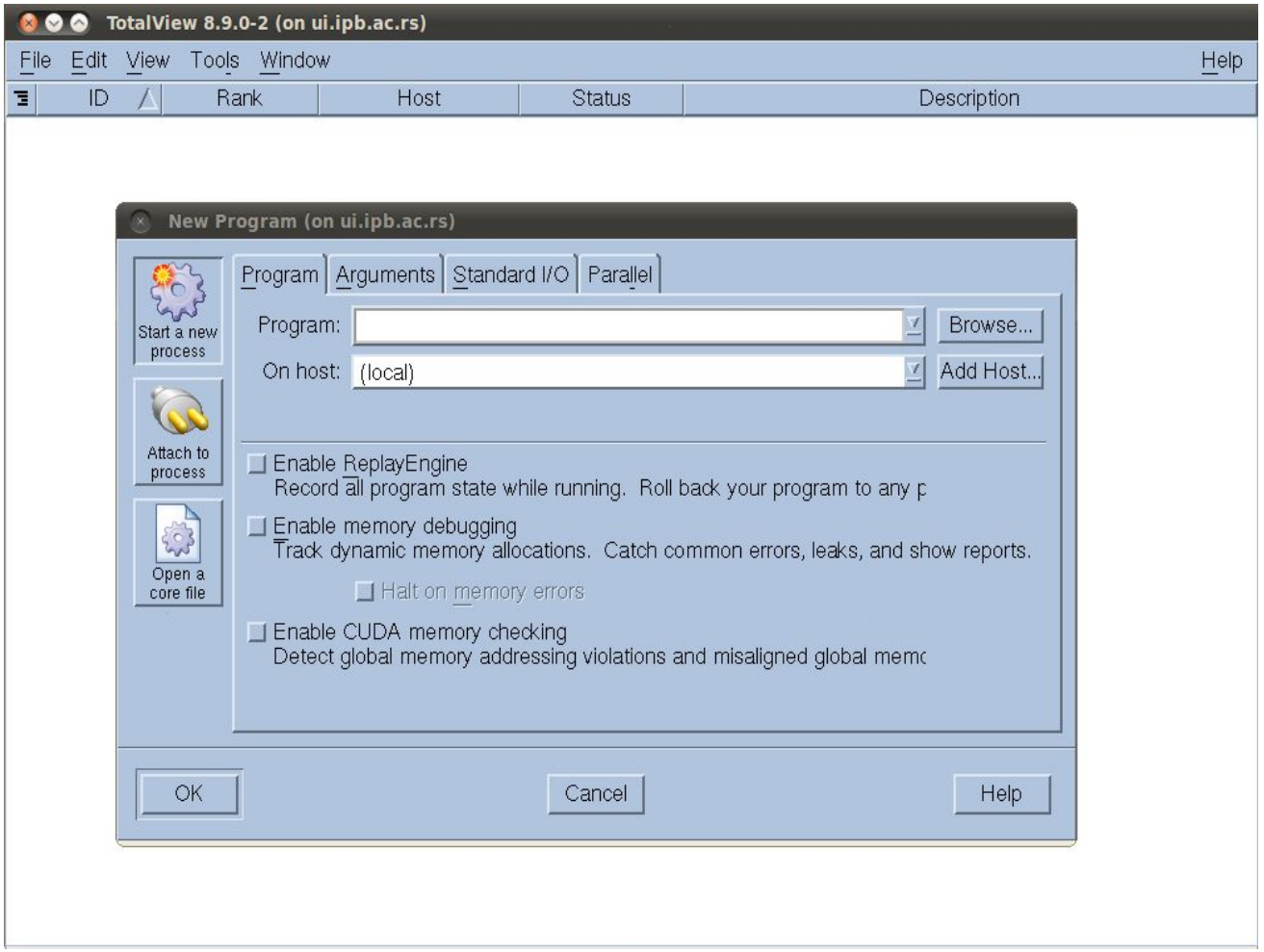
- TotalView is a GUI-based source code defect analysis tool
- It allows you to debug one or many processes and/or threads
- Gives control over program execution, from basic debugging operations like stepping through code to sophisticated techniques that are becoming more commonplace in the high performance computing world such as Graphic Processor Support (debug CUDA)
- Especially designed for use with complex, multi-process and/or multi-threaded applications

Scientific Computing Laboratory

TotalView Debugger (2)

- Support for threads, OpenMP, MPI, or GPUs.
- Provides analytical displays of the state of your running program
- TotalView works with C, C++ and Fortran applications written for Linux (including the Blue Gene platforms), UNIX and Mac OS X platforms
- One of the most popular HPC debuggers

TotalView Debugger (3)



http://www.roguewave.com/products/totalview-family/totalview.aspx

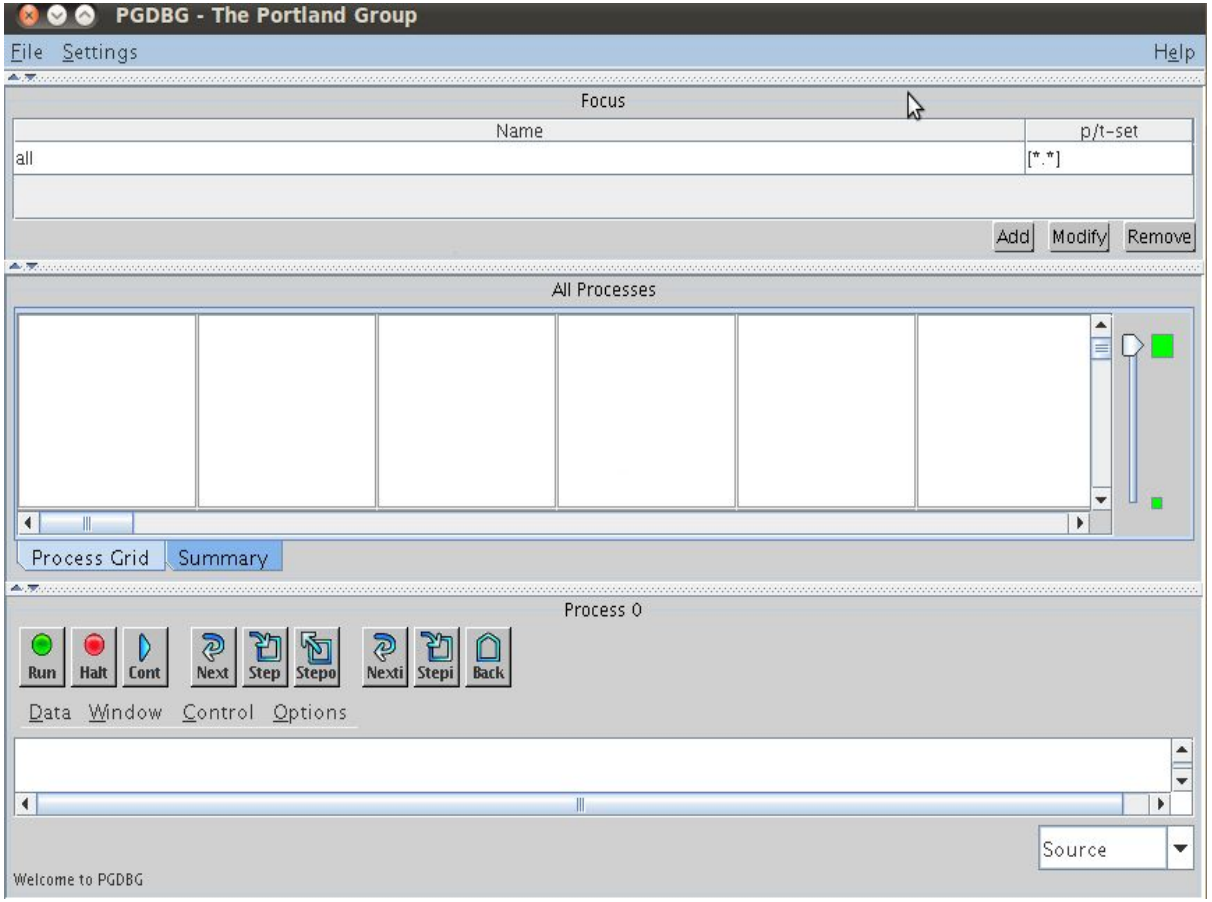
SCIENTIFIC COMPUTING LABORATORY

21 Feb 2012

PBDBG (1)

- PGDBG is a symbolic debugger from PGI for Fortran, C, C++ and assembly language programs
- It provides debugger features, such as execution control using breakpoints, single-stepping, and examination and modification of application variables, memory locations, and registers
- PGDBG supports debugging of certain types of parallel applications:
  - Multi-threaded and OpenMP applications
  - MPI applications
  - Hybrid applications, which use multiple threads or OpenMP as well as multiple MPI processes on Linux clusters

PBDBG (2)


PGDBG - The Portland Group
File Settings Help
Focus
Name
p/t-set
all
[*.*]
Add Modify Remove
All Processes
Process Grid Summary
Process 0
Run Halt Cont Next Step Stepo Nexti Stepi Back
Data Window Control Options
Source
Welcome to PGDBG

Intel Debugger (1)

- The Intel Debugger (IDB) provides support for debugging programs written in C, C++, and Fortran 77, 90 and 95
- It provides a choice of command-line and graphical user interface (GUI) on the Linux Eclipse platform
- A part of the Intel® C++ Composer XE 2011 for Linux and Intel® Fortran Composer XE 2011 for Linux
- The Intel® Debugger can debug both single and multithreaded applications
- Provides OpenMP windows with information about current tasks, teams, task waits, barriers, task spawn trees and locks

Intel Debugger (2)

- The Intel Debugger (IDB) features:
  - Attaches to (and detaches from) a running process and debugs the matching program
  - Loads a program into (and unloads a program from) the debugger, automatically creating and deleting processes as necessary
  - Supports multiple-process debugging, associating with one or more programs
  - See processes and examine detailed process state
  - Set breakpoints for a specific process
  - Supports remote debugging of applications on embedded Intel architecture (using a remote agent)
  - Debugs programs with shared libraries
  - Debugs core files
  - …
- http://software.intel.com/en-us/articles/idb-linux/

IBM Parallel Debugger

- Part of IBM Parallel Environment (PE)
- Standard for debugging on POWER systems
- The parallel debugger (pdb) presents user with a single command line interface that supports most dbx/ gdb execution control commands
- http://www-03.ibm.com/systems/software/parallel/

SCIENTIFIC COMPUTING LABORATORY

Debug or not?

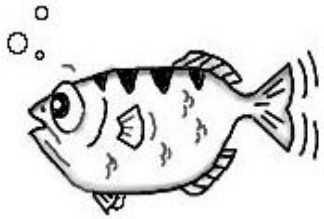
- What is debugging?
- "The best debugging is to avoid bugs"
  - good program design
  - follow good programming practices
  - always consider maintainability and readability of code over getting results fast
  - maximize modularity and code re-use
- Debugging is a last resort

printf() or debugger?

- Using printf() (adding trace to program)
- With debugger you can:
  - attach to running process
  - change the value of variables at run-time
  - make program stop on specific conditions
  - list source code
  - print variables datatype
  - inspect a process that has crashed
  - ...
- Answer is obvious!

SCIENTIFIC COMPUTING LABORATORY

GDB tutorial



- GDB, the GNU Project debugger
- Allows user to see:
  - What is going on `inside' a program while it executes
  - What program was doing at the moment it crashed
- GDB can do four main kinds of things to help users catch bugs :
  - Start program, specifying anything that might affect its behavior
  - Make program stop on specified conditions
  - Examine what has happened, when program has stopped
  - Change things in program, so user can experiment with correcting the effects of one bug and go on to learn about another

SCIENTIFIC COMPUTING LABORATORY

21 Feb 2012

GDB features

- The program being debugged can be written in:
  - Ada
  - C
  - C++
  - Objective-C
  - Pascal
  - and many other languages…
- Programs might be executing on the same machine as GDB (native) or on another machine (remote)
- GDB can run on most popular UNIX and Microsoft Windows variants
- http://www.gnu.org/s/gdb/

Basic usage: compiling

- Enable debugging with flags -g or –ggdb:

  gcc –g –o test test.c

- Source code and executable one to one mapping is made

- Symbol table

Address	Type	Name
00000020	a	T_BIT
00000040	a	F_BIT
00000080	a	I_BIT
20000004	t	irqvec
20000008	t	fiqvec
2000000c	t	InitReset
20000018	T	_main

- Optimization can change things

Basic usage: loading

- Load executable:
  gdb ./test
- Symbols are loaded and we can run program (VM)
- We see a command prompt:
  (gdb)_

Basic usage: run and list

- Type run and program will start (and finish, maybe)
  (gdb) run arg1 "arg2" ...

- **set args** – set arguments for next running

- **list** - list lines of source code (10 lines around argument are displayed):

  list
  list linenum
  list *function*
  list driver.c:20

- .gdbinit

SCIENTIFIC COMPUTING LABORATORY

Basic usage: commands

- **run** - Start execution
- **list [arg]** - List source code around argument
- **break [arg]** - Add a “break point” at arg
- **delete n** - Delete break point number n
- **print [arg]** - Print the content of arg
- **continue** - Continue execution after a break
- **next** - Execute next line
- **step** - Step into next line (enters functions)
- **backtrace** - History of function calls
- **help** – Shows help
- **kill** - Kill program witout quitting gdb
- **quit** - Quit gdb

SCIENTIFIC COMPUTING LABORATORY

Breakpoints, Watchpoints and Catchpoints

- **breakpoint** - stops your program whenever a particular point in the program is reached
- **watchpoint** - stops your program whenever the value of a variable or expression changes
- **catchpoint** - stops your program whenever a particular event occurs

Setting Breakpoints

- Set a breakpoint at specific line on current source code file:

  (gdb) break 40

- Set a breakpoint at specific function:

  (gdb) break my_function

- Set a breakpoint at specific line on some source file :

  (gdb) break parsing.cc:45

- Set a breakpoint at a address:

  (gdb) break 0x43443432

SCIENTIFIC COMPUTING LABORATORY

Removing breakpoints

- **info breakpoints** - get a list of breakpoints
- **delete** – delete all break points
- **delete n** – delete breakpoint n
- **clear function** – delete breakpoint set on function
- **clear linenumber** – delete breakpoint at linenumber
- **disable n** – disable breakpoint n
- **ignore** - skip a breakpoint a certain number of times

SCIENTIFIC COMPUTING LABORATORY

Watchpoints

- Set on variables (expressions) - variable must be in current scope
- **watch** – Set a watchpoint for an expression.
- **rwatch** - Set a read watchpoint for an expression.
- **awatch** - Set a read/write watchpoint for an expression.
- **Disable** – turn off watchpoint (same as breakpoints)

SCIENTIFIC COMPUTING LABORATORY

Catchpoints

- Set on events (C++ exceptions or the loading of a shared library and others)

- **catch EVENT –** event can be :

  throw - The throwing of a C++ exception.

  catch - The catching of a C++ exception.

  exec - A call to `exec'.

  fork - A call to `fork'.

  load - A loading of any library.

  load LIBNAME - A loading of specific library.

  unload - Unloading of library.

  thread_start – Starting any threads, just after creation

  . . .

SCIENTIFIC COMPUTING LABORATORY

Inspecting variables 1/2

- **ptype** – print the data type of a variable

  (gdb) ptype myvar
  type = double

- **print** – view the value of a variable

  (gdb) print i

  $4 = -107

- Inspecting an array:

  (gdb) p myIntArray

  $46 = {0, 1, 2, 3, 4, 5}

  (gdb) p myIntArray[3]@7

  $54 = {3, 4, 5, 10, 1107293224, 1079194419, -1947051841}

SCIENTIFIC COMPUTING LABORATORY

Inspecting variables 2/2

- Inspecting a structure:

  (gdb) p myStruct

  $2 = {name = 0x40014978 “Mile mikic", EyeColour = 1}

  (gdb) print myStruct.name

  $6 = 0x40014978 "Mile Mikic“

- **set** - Changing variable value (must be in current context):

  (gdb) set myvariable = 10.0

- All Fortran variables must be in lowercase!!!

SCIENTIFIC COMPUTING LABORATORY

Navigating through the program

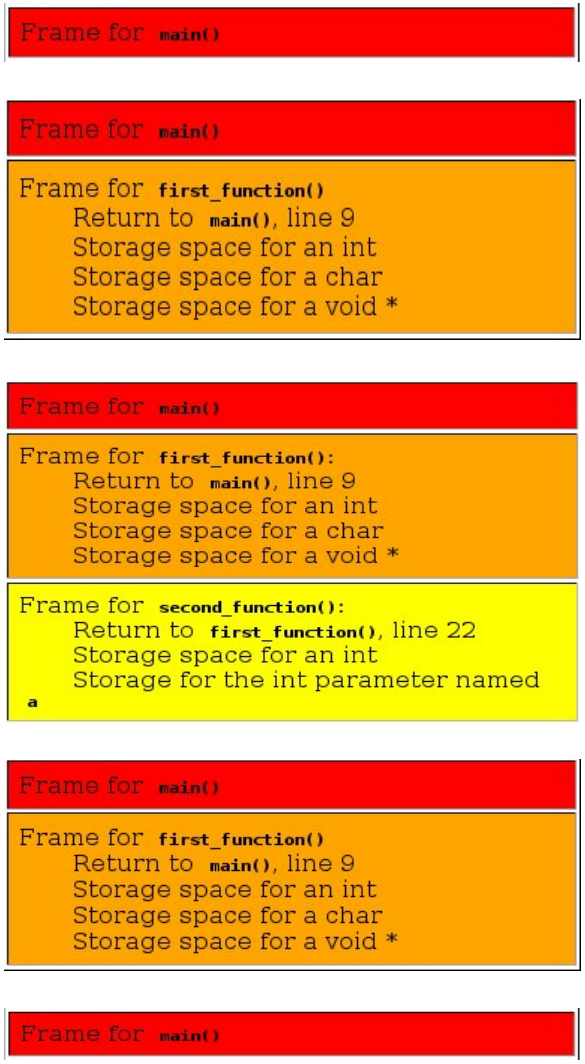
- **next -** Execute a single line in the program. Skip over function calls
- **step** - Execute a single line in the program. Step into functions
- **continue** - continue program being debugged
- **advance** - continue the program up to the given location

Examining the stack

- **backtrace** - Print backtrace of all stack frames
- **frame** - Select and print a stack frame
- **up** - Select and print stack frame that called this one
- **down** - Select and print stack frame called by this one
- **info locals** - Local variables of current stack frame
- **info args** - Local arguments of current stack frame

Call stack

```
1  #include <stdio.h>
2  void first_function(void);
3  void second_function(int);
4
5  int main(void)
6  {
7     printf("hello world\n");
8     first_function();
9     printf("goodbye goodbye\n");
10
11    return 0;
12 }
13
14 void first_function(void)
15 {
16    int imidate = 3;
17    char broiled = 'c';
18    void *where_prohibited = NULL;
19
20    second_function(imidate);
21    imidate = 10;
22 }
23 void second_function(int a)
24 {
25    int b = a;
25 }
```



SCIENTIFIC COMPUTING LABORATORY

21 Feb 2012

Debugging a Running Process

- **attach pid** (from gdb)- attach to the running process with pid

  $ gdb

  (gdb) attach 17399

  Attaching to process 17399….
- **$ gdb program pid** (outside gdb) –

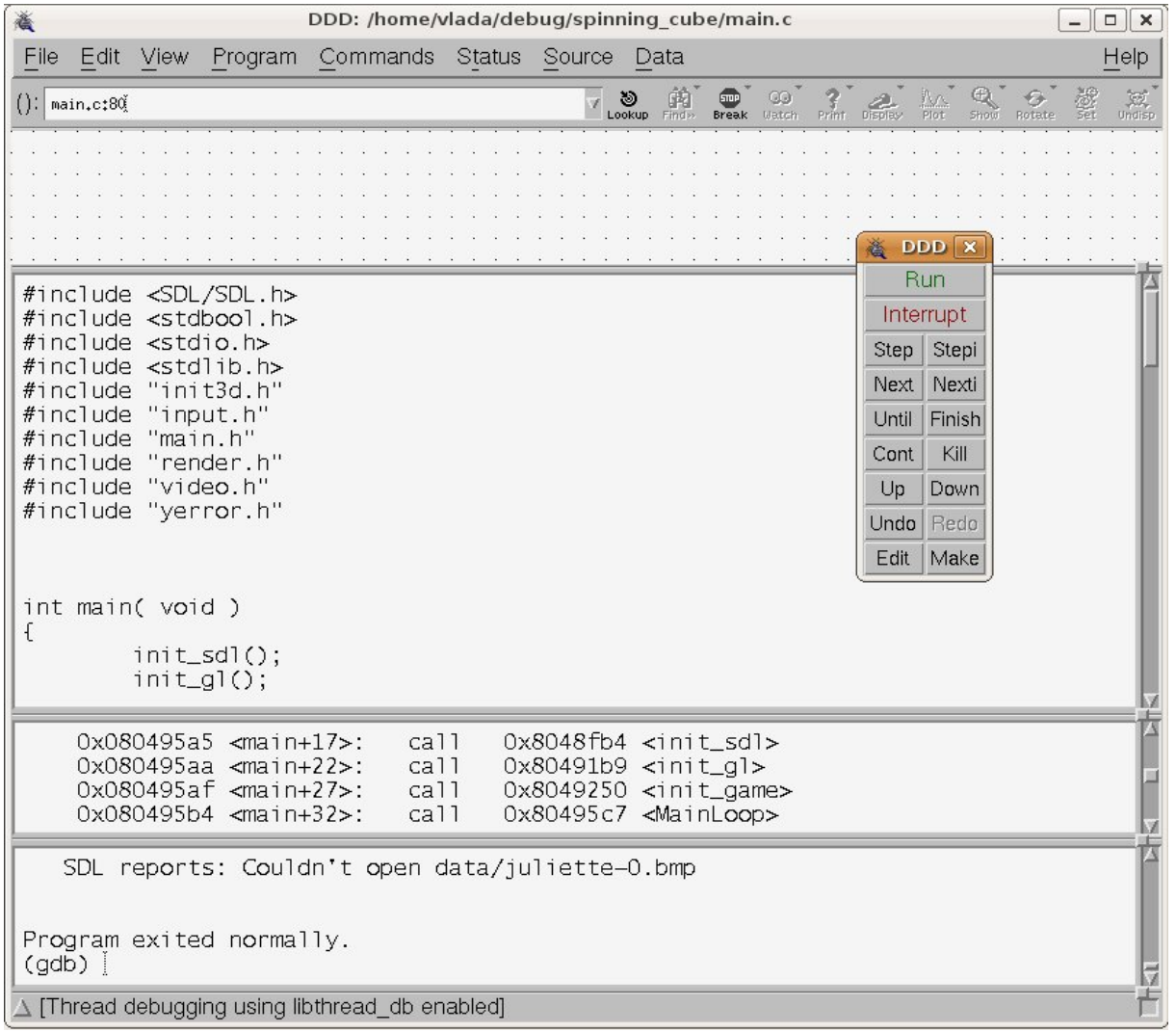
  Attaching to program: code/running_process/some-process, process 17399

  0x410c64fb in nanosleep () from /lib/tls/libc.so.6

  (gdb)
- **detach** – detach from process
- Change variables

SCIENTIFIC COMPUTING LABORATORY

ddd - gdb graphical frontend



SCIENTIFIC COMPUTING LABORATORY

21 Feb 2012

References:

- http://www.gnu.org/software/gdb/
- http://www.dirac.org/linux/gdb/
- http://www.delorie.com/gnu/docs/gdb/gdb_toc.html