

Debugging with gdb

David Khosid

Sept 6, 2009

david.kh@gmail.com

Agenda

- Techniques for debugging big, modern software:
 - STL containers and algorithms, Boost (ex: how to see containers)
 - Multi-threaded (ex.: how to follow a thread?)
 - Signals
 - Repetitive tasks on the almost unchanging code base
- Program structure and running
 - Memory layout
 - Stack and heap
 - 64bit vs. 32 bit
- Examples

Sources of information

GDB was first written by Richard Stallman in 1986 as part of his GNU system

FREE SOFTWARE IS FREEDOM

- Richard Stallman, “Debugging with gdb”
www.gnu.org/software/gdb/documentation
- Help: \$gdb -h
 (gdb) h
 (gdb) apropos

Command names may be truncated if the abbreviation is unambiguous. TAB completion

- Command Cheat Sheet
www.yolinux.com/TUTORIALS/GDB-Commands.html

- Last GDB version is 6.8



Richard Stallman

What can debuggers do?

- Start your program for you, specifying anything that might affect it's behavior.
- Make your program stop under specified conditions.
- Examine what happened when the program stopped.(current variables' values, the memory and the stack)
- Allow you to experiment with changes to see what effect they have on the program.
- Let you examine the program execution step by step
- Let you examine the change of program variables' values - *tracing*

must compile your program with the **-g** option (creates the symbol table) !

Getting In and Out of GDB

1. `gdb my_prog -silent`
2. `gdb my_prog core_files -si`
3. `gdb my_prog pid -si`

Loading the symbol table:

(gdb) **file** *my_prog*

Exit GDB: **q** or **Ctrl-D**

shell commands: (gdb) **shell** *command args*

short form: (gdb) **make** *make-args*

(gdb) **pwd**; (gdb) **cd**

Debugging an already-running process

From inside GDB:

```
attach process-id
```

From outside GDB:

```
gdb my_prog process-id
```

! The first thing GDB does after arranging to debug the specified process is to **stop** it.

detach – detaches the currently attached process from the GDB control. A detached process continues its own execution.

Program's Arguments

Specifying arguments for your program

1. As arguments to `run`: `run arg1 arg2`
2. With `set args` command: `set args arg1 arg2`
3. With `--args` option.

Ex: `#sudo gdb -silent --args /bin/ping google.com`

`run` without arguments uses the same arguments used by the previous `run`.

`set args` without arguments – removes all arguments.

`show args` command shows the arguments your program has been started with

Program's environment

(gdb) show environment

(gdb) path *dir* – add the directory *dir* at the beginning of the PATH variable.

(gdb) show paths – displays the search paths for executables.

The working directory:

pwd

cd *dir* – to change the working directory

Breakpoints and watchpoints

Allow you to specify the places or the conditions where you want your program to stop.

```
(gdb) b location if cond
```

```
(gdb) watch expr – stops whenever the value of the  
expression changes
```

```
(gdb) i b
```

```
(gdb) clear [arg]
```

```
(gdb) delete [bnum]
```

Without arguments deletes all breakpoints.

Examining variables

The variable type: **p***type* *var*

Current value: **p** *var*

Automatic display: **display** *var* - adds *var* to the
automatic display list.

undisplay *dnum*

Specifying the output format (*x*, *o*, *d*, *u*, *t*, *a*, *f*, and *c*):

print /*t* *var* - prints the value of *var* in binary format

GDB - Examining memory

The `x` command (for “examine”):

- `x/nfu addr` – specify the number of units (n), the display format (f) and the unit size (u) of the memory you want to examine, starting from the address `addr`. Unit size can be – b, h (half), w and g (giant).
- `x addr` – start printing from the address `addr`, others default
- `x` – all default

Registers

Registers names are different for each machine. Use `info registers` to see the names used on your machine.

GDB has four “standard” registers names that are available on most machines: program counter, stack pointer, frame pointer and processor status.

C++ and STL - Containers

How to see container's content?

- Commands file .gdbinit

http://www.yolinux.com/TUTORIALS/src/dbinit_stl_views-1.03.txt

Limitations: a little, ex., heterogeneous `std::map<T,Q>`

- `libstdc++` compiled in debug mode

- Auxiliary functions

```
typedef map<string, float> MapStringFloat;
void testPrint(const MapStringFloat& m){
    for(MapStringFloat::const_iterator pos = m.begin(); pos != m.end(); ++pos){
        cout << pos->first << " : " << pos->second << "\n";
    }
}
```

- Pretty-printing of STL containers in future versions of GDB

C++ and STL - continue

- Overloaded functions
- **rbreak** `regex` - is useful for setting breakpoints on overloaded functions that are not members of any special classes. The `rbreak` command can be used to set breakpoints in all the functions in a program, like this:
`(gdb) rbreak .`
- Exceptions: catchpoints
`(gdb) catch throw`, `(gdb) catch catch`
- Templates
It is possible that a breakpoint corresponds to several locations in your program.

The stack frame

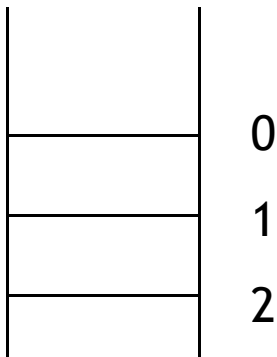
- Stack frames are identified by their addresses, which are kept in the *frame pointer* register.

➤ Selecting a frame:

f *n*

up *n*

down *n*



➤ Information about the current frame

f or **frame** - brief description

i args - shows function arguments

i locals - shows local variables

➤ Stack: **(gdb)bt** or **(gdb)bt full**

Stepping through the program

step [*count*] – program execution continue to next source line going into function calls.

'n' or 'next' [*count*] – program execution continue to the next source line omitting function calls.

'c' or 'continue' – resume program execution

until – continue until the next source line in the current stack frame is reached. /useful to exit from loops/

Altering execution

Returning from a function

finish – Continue running until just after function in the selected stack frame returns.

return [*ret_value*] – pops the current stack frame

Continuing at different address

jump *line_num*/**address*

Altering the value of a variable

set *i=256*

Convenience variables

- Convenience variables are used to store values that you may want to refer later. Any string preceded by \$ is regarded as a convenience variable.

Ex.: **set \$table** = **table_ptr*

(gdb) **show conv**

- There are several automatically created convenience variables:

\$pc – program counter

\$sp – stack pointer

\$fp – frame pointer

Extending GDB - init files

- **What GDB Does During Startup**
 1. Executes all commands from system init file
 2. Executes all the commands from ~/.gdbinit
 3. Process command line options and operands
 4. Executes all the commands from ./gdbinit
 5. reads command files specified by the '-x' option
 6. ...

Extending GDB - History, recording

- **What GDB Does During Startup**

... 6. Reads the command history recorded in the *history file*.

- **(gdb) set history filename** *fname*
- **(gdb) set history save** on/off
- **(gdb) show history**
- **(gdb) show commands**

Extending GDB - User-defined commands

- **(gdb) show user *commandname***
- **Example:**

```
(gdb)define adder
    print $arg0 + $arg1 + $arg2
end
(gdb) adder 1 2 3
```

Editing files during debugging

Example:

```
(gdb) run
```

Program received signal SIGSEGV, Segmentation fault.

```
(gdb) edit or (gdb) shell vi crash.cpp
```

```
(gdb) shell gcc crash.cpp -o crash -lstdc++
```

```
(gdb) run
```

Program exited normally.

```
(gdb) quit
```

Signals

- **'i handle' or 'i signals'**

Print a table of all the signals and how gdb has been told to handle each one.

- **handle signal** [keywords...]

keywords: nostop|stop, print|noprint and pass|nopass

Ex: handle SIG35 nostop print pass

handle SIG36 stop (implies the 'print' as well)

handle SIG37 nostop print nopass

handle SIG38 nostop noprint nopass

Multi-threads

- Use case: debugging specific thread, while controlling behavior of others.
- facilities for debugging multi-thread programs:
 - automatic notification of new threads
 - 'thread threadno', to switch among threads
 - 'info threads', to inquire about existing threads
 - thread-specific breakpoints
 - set mode for locking scheduler during execution
(gdb) set scheduler-locking step/on/offothers: Interrupted System Calls
- Example:
(gdb) **i threads**
break foo.cpp:13 thread 28 if x > lim

Checkpoint

- A snapshot of a program's state

(gdb) **checkpoint**

(gdb) **i checkpoint**

(gdb) **restart** *checkpoint-id*

64 bit .vs. 32bit

- -m32 flag
- On 64-bit machine, install another 32-bit version of GDB

\$ ls -l `which gdb32`

`/usr/bin/gdb32 -> '/your/install/path'`

Additional process information

i proc – summarize available information about the current process.

i proc mappings – address range accessible in the program.

Remote debugging

- **Use case:**
 - GDB runs on one machine (host) and the program being debugged (exe.verXYZ.stripped) runs on another (target).
 - GDB communicates via Serial or TCP/IP.
 - Host and target: exactly match between the executables and libraries, with one exception: stripped on the target.
 - Complication: compiling on one machine (CC view), keeping code in different place (ex. /SDE60/verXYZ)
- **Solution:**
 - Connect gdb to source in the given place:
(gdb) set substitute-path /usr/src /mnt/cross
(gdb) dir /SDE60/verXYZ

Remote debugging - example

- Using gdbserver through TCP connection:

```
remote (10.10.0.225)> gdbserver :9999 program_stripped  
or remote> ./gdbserver :9999 -attach <pid>
```

- host> gdb *program*

```
host>(gdb) handle SIGTRAP nostop noprint pass
```

to avoid pausing when launching the threads

```
host> (gdb) target remote 10.10.0.225:9999
```

Various issues

- DDD and Eclipse
- Working with Shared Libraries
- How to see macros

DDD and Eclipse - GUI Advantages

DDD is a GUI debugger that work with GDB.

- GDB commands can be typed in the console window.
- Frequently used commands are on the toolbars, have assigned shortcut keys or can be done just with a mouse click.
- Easy browsing through the source
- Examining current variables values directly – by placing the mouse pointer over them.
- Possibility to graphically display the program data.

IMHO, stability is the main issue of debugging in DDD and especially Eclipse.

Summary

1. Start from thinking of Use Case, then look in the manual, use 'apropos' and 'help'
2. Productivity:
Stepping through a program is less productive than thinking harder and adding output statements and self-checking code at critical places.
3. When to use GDB?
 - core file,
 - when a problem can be reproduced, repeating errors
 - self-educating
4. When not?
Other tools, traces
5. Questions?

Questions and how-to's

1. How to create a symbol table? How to remove it?
2. How to load up your program in GDB?
3. How to know where you are (file, next execution line)?
4. How to find out the crash file executable?
5. How to find out why a program stopped?
6. Which command(s) can be used to exit from loops?
7. Why 'print', 'info', 'show'?

Problem Determination Tools for Linux

- -Wall 😊
- Code review
- Program's traces, syslog, profilers
- Static Source Code Analysis:
 - scan.coverity.com – free for FOSS
 - Flexelint
- Dynamic analysis: Valgrind,
- strace, /proc filesystem, lsof, ldd, nm, objdump, wireshark