

Software Development Debugging

Davide Balzarotti

Eurecom – Sophia Antipolis, France

Software Development Tools

1. Writing and managing code
2. Configuring and Building the program
 - ✓ GCC
 - ✓ Makefiles
 - ✓ Autotools
3. Packaging and Distributing the application
4. Debugging and Profiling

Debugging & Profiling

1. Debugging Intro
2. GDB basics
3. GDB 7.0 new features
 - ✓ Reverse debugging
 - ✓ Scripting in Python
4. Profiling

Why Debugging

- 99% - Finding a bug inside a **known** program
 - The program shows an unexpected behavior
 - It does not work in certain conditions
 - It crashes
 - It returns the wrong value
 - ...
 - You have the source code and all the debugging symbols
- 1% - Find what an **unknown** program does
 - Reverse engineering
 - Malware analysis
 - ..
 - You only have the (often stripped) binary

Why Debugging

- 99% - Finding a bug inside a **known** program
 - The program shows an unexpected behavior
 - It does not work in certain conditions
 - It crashes
 - It returns the wrong value
 - ...
 - You have the source code and all the debugging symbols

SoftDev

- 1% - Find what an **unknown** program does
 - Reverse engineering
 - Malware analysis
 - ..
 - You only have the (often stripped) binary

**Security
Classes**

Debuggers

- Debuggers are software tools which enable users to:
 - analyze what is going on "inside" a program while it executes (live debugging)
 - analyze what a program was doing at the moment it crashed (post-mortem debugging)
- Debuggers are very capable tools but, like programming, debugging software requires a lot of experience

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it
-- Brian Kernighan

David Agans' Debugging Rules



DEBUGGING RULES!

- Understand the system
- Make it fail
- Quit thinking and look
- Divide and conquer
- Change one thing at a time
- Keep an audit trail
- Check the plug
- Get a fresh view
- If you didn't fix it, it ain't fixed

from [Debugging](#) © 2002 by David Agans

To get the book or download this free poster, go to www.debuggingrules.com

Debugging & Profiling

1. Debugging Intro
2. GDB basics
3. GDB 7.0 new features
 - ✓ Reverse debugging
 - ✓ Scripting in Python
4. Profiling

The GNU Debugger

- The GNU Debugger (**GDB**) is the standard debugger for the GNU software system and the main debugger under linux/bsd systems
 - First written by Richard Stallman in 1986
- GDB can do four main kinds of things to help you catch bugs:
 - Start your program, specifying anything that might affect its behavior
 - Make the program stops on specified places or on specified conditions
 - Examine what has happened, when your program has stopped
 - Give information about current variables' values, the memory and the stack
 - Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another

Make a Program Debugger-friendly

`gcc -g`

- Include in the application the debugging information in the operating system's native format

`gcc -ggdb3`

- Produce debugging information for use by GDB. This means to use the most expressive format available, including GDB extensions if at all possible
- Level 3 includes extra information, such as all the macro definitions present in the program
- GCC can combine debugging symbols with compiler optimizations. However...
 - It's better to disable optimization when debugging
 - It's better to remove the symbols before releasing the software (bash with symbols: 2.5M, bash without 600K)

Starting GDB

- GDB provides an **interactive environment** with a command line interpreter
 - You can enter it by typing: `GDB <program_name>`
 - The program is not automatically started when entering GDB
 - You can exit it by typing `quit` (or CTRL-D)
- GDB can also be used in **batch mode**
 - `gdb -batch -x cmd_file <program_name>`
 - Exit with status 0 after processing all the command specified in `cmd_file`
 - Exit with nonzero status if an error occurs in executing `gdb`

Running a Program

```
balzarot> gdb -q program
(gdb) run arg1 arg2 ...
```

```
balzarot> gdb -q program
(gdb) set args arg1 arg2 ...
(gdb) run
```

- The program normally uses the same device for standard input and standard output as gdb is using.
 - Input and output can be redirected in the run command

```
(gdb) run > output_file
```
- `run` without arguments uses the same arguments used by the previous run (!!)
- `set args` without arguments removes all arguments
- `show args` shows the arguments the program has been started with

Attaching to a Running Process

```
balzarot> gdb <prog_name> -p <process_id>
```

```
balzarot> gdb <prog_name>  
(gdb) attach <process_id>
```

- GDB can be used to debug a process that is already running on the machine (i.e., that was started outside GDB)
- The first thing GDB does after attaching to the running process is to stop it
 - `continue` – resume the program execution
 - `detach` – detaches the currently attached process from the GDB control. A detached process continues its own execution

Core Dumps

- A core file (or **core dump**) is a file that records the memory image of a running process and its process status (values of the CPU registers, etc.)
 - Its primary use is post-mortem debugging of a program that crashed while it ran outside a debugger.
 - On many operating systems, a fatal error in a program automatically triggers a core dump
- You can enable/disable or control the size of the core dumps by setting process limits
 - `ulimit -c unlimited` – core dump of unlimited size
 - `ulimit -c 0` – prevent the core file from being created
 - `ulimit -c 1000` – set the dump file limit to 1000Kb

*These commands are for the bash shell, other shells have similar ones

Analyzing Core Dumps

```
balzarot> gdb <prog_name> -c <core_file>
```

- Run GDB, load a program and the memory image from the core file
- Show the status of the program at the moment when it terminated
 - Not as good as watching it run...
 - ...but sometimes it is the best you can do

Commands

- A gdb command is a single line containing the command name and its parameters
- You can abbreviate a gdb command to the first few letters of the command name, if that abbreviation is unambiguous
 - You can use the TAB key to get gdb to fill out the rest of a word in a command (or to show you the alternatives available, if there is more than one possibility)
- You can usually repeat the last GDB command by just typing the ENTER key
 - For some command this feature is disabled since it would not make sense to execute it twice (e.g., attach)
- Getting help:
 - `help command` – show the command description
 - `apropos regexp` – search regexp inside all command documentation

Program Locations

- Many commands require a parameter that specify a location in the program
- Locations can be specified in many ways:
 - A number is interpreted as a line number in the source code
Ex: 55
 - A name is interpreted as a function_name
Ex: strcpy
 - Both functions and line numbers can be prefixed by the file name.
Ex: 'foo.c'::my_func
 - A number prefixed by + or – is interpreted as an offset from the current line.
Ex: +20 (20 lines ahead)
 - A number prefixed by * is interpreted as a memory address
Ex: *0xbfff6543

Checking the Current Status

(gdb) info <subject>

- Generic command for showing information about the the state of the debugged program
 - `info registers` – show the content of the registers
 - `info breakpoints` – show the list of breakpoints
 - `info program` – show the current status of the program
 - `info args` – show the parameter with which the current function was invoked

(gdb) show <subject>

- Generic command for showing information about the state of GDB itself
 - `show env` – show the environment variables
 - `show history` – show the GDB history settings
 - `show args` – show the parameters passed to the program command line

Checking the Current Status

(gdb) info <subject>

- Generic command for showing information about the the state of the debugged program
 - `info registers` – show the content of the registers
 - `info breakpoints` – show the list of breakpoints
 - `info program` – show the current status of the program
 - `info args` – show the parameter with which the current function was invoked

(gdb) show <subject>

- Generic command for showing information about the state of GDB itself
 - `show env` – show the environment variables
 - `show history` – show the GDB history settings
 - `show args` – show the parameters passed to the program command line

Changing Variables

(gdb) `set <variable> = <value>`

- Sets the value of a variable
- It can be used to set the value of one of the GDB status variable that you can see using the `show` command
 - `set env USER = foo`
- It can be used to set the value of a program variable
 - `set x = 5`
- It can be used to define convenience variables
 - The variable name must be prefixed by `$`
 - `set $i =5`
- It can be used to set the value of a register
 - `set $sp=$sp+4`

Controlling the Program Execution

(gdb) `step [n]`

- Execute until the program reaches the next source line
- Repeat N times

(gdb) `next [n]`

- Like `step`, but “step over” subroutine calls

(gdb) `stepi [n]`

- Execute one *machine* instruction, then stop and return to the debugger

(gdb) `continue`

- Resume the program execution

(gdb) `finish`

- Execute until the select stack frame returns

Breakpoints

- Breakpoints specify conditions that determine when a program's execution should be interrupted
 - The most common form of a breakpoint (*instruction breakpoint*) is the one that specify an instruction and stop the program before executing it
 - Another common form of breakpoint (*watchpoint*) consists in instructing the debugger to stop the program every time it access a specifying memory region

(gdb) **break** <location>

- Set a breakpoint to a given location
(the program will stop before the instruction at that location is executed)

(gdb) **tbreak** <location>

- Set a temporary breakpoint that is automatically deleted when triggered

(gdb) **info** <breakpoints>

- Show the list of breakpoints

Hardware and Software Breakpoints

- GDB normally implements breakpoints by replacing the program code at the breakpoint address with a special instruction, which, when executed, gives back the control to the debugger
- Some processor provide hardware support to set a breakpoint at an instruction without changing the instruction
 - The support works by setting the instruction address in special debug registers
 - The number of debug registers is normally very limited (2 or 4)
- GDB can be instructed to automatically use hardware breakpoints when available

```
(gdb) show breakpoint auto-hw
```

```
(gdb) set breakpoint auto-hw on
```

Break Conditions

- The simplest type of breakpoint breaks every time the program reaches a specified place
- GDB allows the user to specify additional conditions that must be true in order to stop the program
 - A condition is evaluated each time the program reaches it, and the program stops only if the condition is true

(gdb) break <location> if <expression>

(gdb) condition <bp_num> <expression>

- Set a conditional breakpoint or change the condition associated to an existing breakpoint
- Example:

```
(gdb) break my_func if parameter1 > 10
```

Breakpoint's Command List

- It is possible to specify a series of commands to execute when the program stops at a certain breakpoint
 - For example, you might want to print the values of certain expressions, or enable other breakpoints

```
(gdb) commands <breakpoint_num>
command 1
command 2
....
end
(gdb)
```

- If the first command in a command list is `silent`, the usual message about stopping at a breakpoint is not printed.
 - This may be desirable for breakpoints that have to print a certain message and then continue

Breakpoint's Command List

- One application for breakpoint commands is to compensate for one bug so you can go past it in the execution
- Or force a particular condition to happen so you can debug a particular execution path
- Example:

```
(gdb) break 403 if x < 0
(gdb) commands
>silent
>set x = y + 4
>continue
>end
```

Watchpoints

- Watchpoints can be used to stop the execution whenever the value of an expression changes, without having to predict a particular place where this may happen
 - The expression may be as simple as the value of a single variable, or as complex as many variables combined by operators
- Depending on the system, watchpoints may be implemented in software or hardware
 - GDB does software watchpointing by single-stepping your program and testing the variable's value each time, which is hundreds of times slower than normal execution
 - Watching complex expressions that reference many variables can quickly exhaust the resources available for hardware-assisted watchpoints because GDB needs to watch every variable in the expression with separately allocated resources

Watchpoints

(gdb) **watch** <expression>

- Set a watchpoint for an expression
- GDB will break when the expression is **written** into by the program and its value changes.
- The most common form of expression is a program variable name

(gdb) **rwatch** <expression>

- Watchpoint that breaks when the expression is **read** by the program

(gdb) **awatch** <expression>

- Watchpoint that breaks when the expression is **accessed** by the program (either read or written)

Catchpoints

- Catchpoints tell the debugger to stop for certain kinds of program events, such as C++ exceptions or the loading of a shared library

(gdb) catch throw

(gdb) catch catch

- Catch exceptions

(gdb) catch fork

- Catch process or thread creations

(gdb) catch syscall [syscall_number]

- Catch the invocation of (one particular) system calls
- Available in GDB 7.0

Managing Breakpoints

(gdb) clear location

- Delete any breakpoints set at the specified location

(gdb) delete breakpoint

- Delete the breakpoints, watchpoints, or catchpoints specified as arguments.
- If no argument is specified, delete them all

(gdb) disable breakpoint

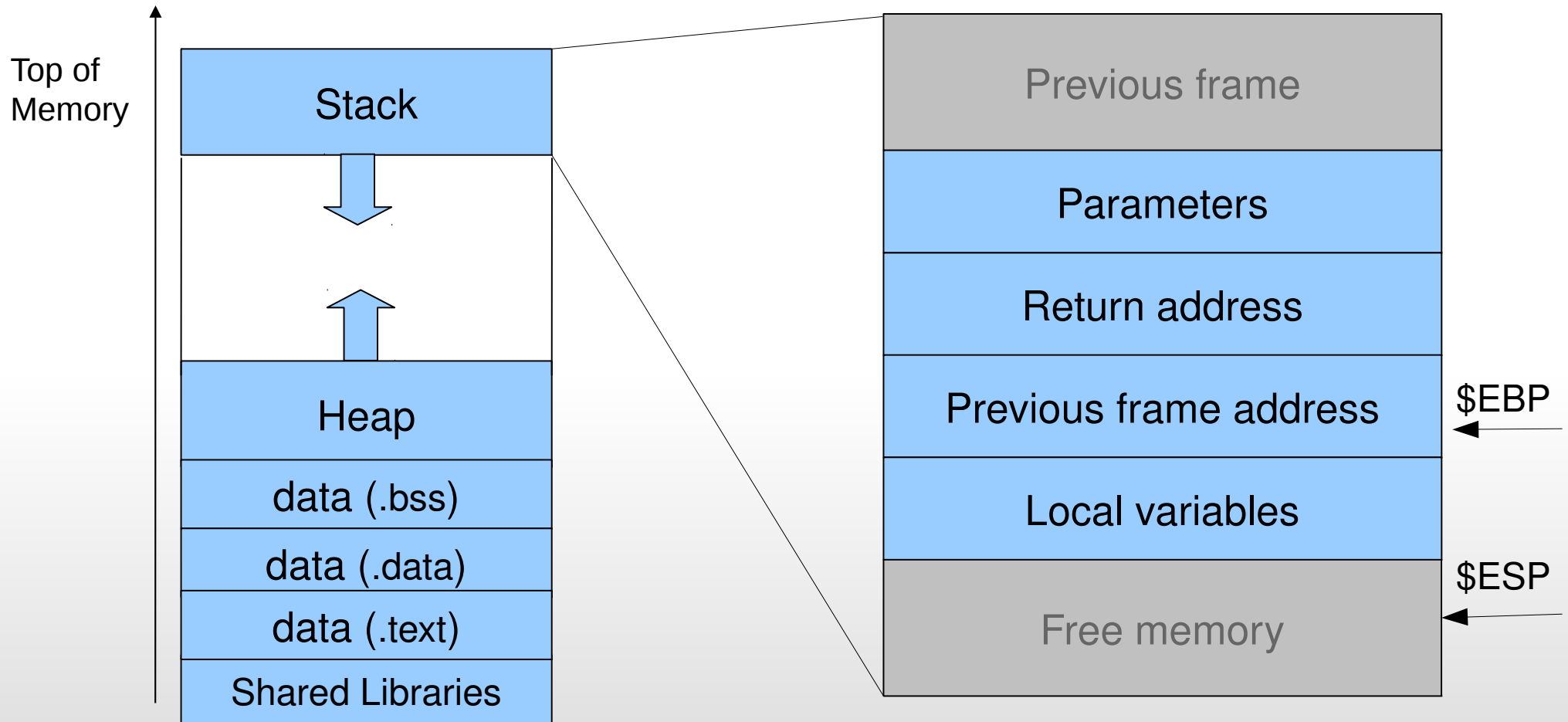
(gdb) enable breakpoint

- Disable (or re-enable again) a break/watch/catch point

Stack Frames

- Each time a program invokes a function, information about the call is generated and placed on the stack
 - This information includes the location of the call in your program, the arguments of the call, and the local variables of the function being called
 - The information is saved in a block of data called a *frame*
- GDB assigns numbers to all existing stack frames, starting with zero for the innermost frame, one for the frame that called it, and so on upward
- Most commands for examining the stack and other data in your program work on whichever stack frame is selected at the moment
 - By default is the last one

Function Frame



Frames Management in GDB

(gdb) frame n

(gdb) select-frame n

- Select the n-th frame

(gdb) backtrace

- print a summary of all the frames all the way back to the *main* function

(gdb) set backtrace past-main

- Tell GDB to show also the frame of the functions before the main invocation

(gdb) up

(gdb) down

- Select the previous (next) frame in the stack

Frames Management in GDB

(gdb) info frame

- Show all the information about the selected frame

(gdb) info args

- Print the arguments of the function invocation that correspond to the selected frame, each on a separate line

(gdb) info locals

- Print the local variables of the selected frame, each on a separate line

(gdb) info catch

- Print a list of all the exception handlers that are active in the current stack frame at the current point of execution.

X64 Architecture

- Registers

- X64 extends x64's 8 general-purpose to be 64 bit and adds 8 new 64-bit registers. The registers have names beginning with “r”

- Calling Conventions

- The first 4 integer pointer parameters are passed in the rcx, rdx, r8, r9 registers.

X64 Architecture

- The caller reserves space on the stack for arguments passed in registers. The called function can use this space to spill the contents of registers to the stack.
- An integer or pointer value is returned in the rax register.

X64 Architecture

- Addressing Modes
 - Instructions that refer to 64-bit registers are automatically performed with 64-bit precision.
 - Instructions, such as jmp, call, push and pop that implicitly refer to the instruction pointer and the stack pointer treat them as 64 bits registers on x64.

Examining the Program Code

(gdb) list [location]

- Show the program source code (around location)

(gdb) disas [from] [to]

- Disassemble the program (between two locations)

(gdb) set disassembly-flavor intel|att

- Set the assembler syntax to intel or at&t (default)

(gdb) info line number

- map source line number to program addresses

Examining Data

(gdb) print expression

- Print the value of an expression

(gdb) whatis arg

- Print the data type of arg
- Example:

```
(gdb) whatis x
type = struct point
```

(gdb) ptype arg

- Print the definition of the data type of arg
- Example:

```
(gdb) ptype x
type = struct point {
    int x;
    int y;
}
```

Examining Memory

(gdb) `x/nfu addr`

- Examine the process memory
 - **N** - the repeat count. An integer (default is 1) that specifies how much memory (counting by units **u**) to display
 - **U** - the unit size
 - **b**: bytes
 - **h**: half-words (two bytes).
 - **w**: words (four bytes)
 - **g**: giant words (eight bytes).
 - **F** - the display format - Specify how the data must be interpreted (like the placeholders in the printf's format string)
 - **x, d, u, o, t, a, c, f, s** – hex, decimal, char, string...
 - **i** - for machine instructions
- Example:

(gdb) `x/10wh *0xbffff567` – prints 10 words in hexadecimal

Debugging Multi-thread Programs

(gdb) info threads

- Display a summary of all threads currently in your program

(gdb) thread thread_no

- Make thread number thread_no the current thread
- To each thread gdb assign a sequential number (it is not the thread_id!)

- By default, all the commands are applied to the current thread

(gdb) thread apply [thread_id | all] command

- Execute command for a specific (or all) thread

Debugging Multi-Process Programs

- On most systems, GDB has no special support for debugging programs which create additional processes using the fork function
 - The debugger follows the parent process
 - If the child triggers a breakpoint it will receive a SIGTRAP signal that is likely to terminate the program
- On some systems (like linux) GDB provides support for debugging forking programs

(gdb) set follow-fork-mode [parent|child]

- Tells the debugger which process to monitor
- The other ones will be detached and allowed to run independently

Debugging Multi-Process Programs

(gdb) set detach-on-fork mode [on|off]

- If set to off, then GDB will retain control of all forked processes (including nested forks)
- The selected process is debugged as usual, while the others are held suspended

(gdb) info forks

- Print a list of all forked processes under the control of gdb

(gdb) fork fork-id

- Make process fork-id the current process

(gdb) detach fork fork_id

- Detach the debugger from one process

Altering Execution

- Once you think you have found the error in the program, you might want to find out whether correcting it would lead to correct results in the rest of the run
 - You can find the answer by experiment, using the gdb features for altering execution of the program
- Modifying program data
 - Changing the value of a variable:
 - `(gdb) set var x=4`
 - `(gdb) set var x=foo(2)`
 - Changing the content of a memory location
 - `(gdb) set {int}0xbf830402 = 4`

Altering Execution

- Modifying program flow

(gdb) **jump** <location>

- Resume execution at the specified location
- The jump command does not change the current stack frame, or the stack pointer, or the contents of any memory location or any register other than the program counter
- It is similar to: `set $pc=location`

(gdb) **return** [value]

- Discards the selected stack frame (and all frames within it)
- It's like the corresponding function returned prematurely

Debuggin & Profiling

1. Debugging Intro
2. GDB basics
3. GDB 7.0 new features
 - ✓ Reverse debugging
 - ✓ Scripting in Python
4. Profiling

Reverse Debugging

- Based on **Process Recording**
 - record the execution of a process in the debugger
 - play it backward and forward
 - "step" or "continue" your program backward in "time", reverting the program to an earlier execution state
- Still not supported by all target architectures (but works fine in Linux)
- Add a number of new commands:

```
reverse-continue  
reverse-step  
reverse-stepi  
reverse-finish  
record  
...
```

Reverse Debugging

```
void initialize(int *array, int size) {
    int i;
    for (i = 0; i <= size; ++i)
        array[i] = 0;
}

int main(void) {
    int *p = malloc(sizeof(int));
    int values[10];

    *p = 37;
    initialize(values, 10);
    printf("*p = %d\n", *p);
    free(p);

    return 0;
}
```

```
gcc -ggdb -O0 test.c -o test
```

```
> gdb test
(gdb) b main
(gdb) run
(gdb) record
(gdb) continue
Program received signal SIGSEGV, Segmentation fault.
0x0804848b in main () at bugged.c:13
13      printf("*p = %d\n", *p);
(gdb) print p
$1 = (int *) 0x0
(gdb) watch p
(gdb) reverse-continue
Hardware watchpoint 2: p

Old value = (int *) 0x0
New value = (int *) 0x804b008
0x0804843c in initialize (array=0xbffff514, size=10)
(gdb) p i
$2 = 10
```

Scripting GDB with Python

- GDB 7.0 introduced the ability to run python code inside the GDB environment

```
(gdb) python print "Hello World"
```

```
Hello World
```

```
(gdb) python
```

```
>X = 42
```

```
>print x
```

```
>end
```

```
42
```

```
(gdb) python execfile("script.py")
```

- **WARNING!!** The API were (are?) changing quickly, so many examples you find on the web may not work anymore

Using Python in Gdb

- gdb defines a set of function that can be invoked by python in the gdb module

`import gdb` – import the gdb module (explore with `dir(gdb)` what is the content)

`gdb.execute("gdb command")` – execute a gdb command

`gdb.parse_and_eval(expr)` – evaluate the expression and return the result

Ex: `gdb.parse_and_eval("x*3")`

Inferiors in Python

- Programs which are being run under gdb are called inferiors
- Python scripts can access and manipulate inferiors controlled by gdb using `gdb.Inferior` objects

`gdb.inferiors()` - returns a list of inferior objects

- Useful field/methods of Inferior

`<inferior>.pid` – get the process identifier

`<inferior>.threads()` - get the list of threads associated to the inferior process

`<inferior>.read_memory(addr, length)`

`<inferior>.write_memory(addr, buffer)`

read and write memory in the inferior process address space

Types and Values

- gdb provides values it obtains from the inferior program in an object of type `gdb.Value`
 - Inferior values that are structures or instances of some class can be accessed using the Python dictionary syntax

```
bar = some_value['foo']
```
- gdb represents types from the inferior using the class `gdb.Type`
 - `gdb.lookup_type(type_name)` – returns the `Type` object given a type name
 - `<type>.sizeof()` - returns the type size
 - `<type>.target()` - for typedef returns the real type, for pointer returns the pointed-to object, for functions returns the return value, and for array the element type.

Frames and Symbols

- `gdb.selected_frame()` - returns the select frame
 - `<frame>.name()` - get the function name
 - `<frame>.pc()` - get the address where the frame will resume execution
 - `<frame>.older()` - get the previous frame on the stack
 - `<frame>.read_var(name)` – get the value of a local variable
- gdb represents every variable, function and type as an entry in a symbol table. Each symbol is represented in python by a `gdb.Symbol` object
 - `gdb.lookup_symbol(name)` – lookup a symbol by name
 - `<symbol>.name`
 - `<symbol>.addr_class` – defines the symbol type
(`SYMBOL_LOC_ARG`, `SYMBOL_LOC_LOCAL`, ...)

Breakpoints

- Python code can manipulate breakpoints via the `gdb.Breakpoint` class
- `gdb.breakpoints()` - return the list of defined breakpoint
- `gdb.Breakpoint(location, type, [wp_class])`
 - Add a breakpoint to `location` of `type` (BP_BREAKPOINT, BP_WATCHPOINT)
For watchpoint, `wp_class` defines the type (WP_READ, WP_WRITE, or WP_ACCESS)
 - `<breakpoint>.condition` – string defining the condition of the breakpoint

Adding a new Function

```
class CallerIs(gdb.Function):
    "Returns True if caller's name is func_name"
    def __init__(self):
        super(CallerIs, self).__init__("caller_is")

    def invoke(self, func_name):
        func_name = str(func_name).split()[-1][1:-1]
        older = gdb.selected_frame().older()
        if older:
            return older.name() == func_name
        else:
            print 'No frame older than current'
            return False

CallerIs()
end
```

```
(gdb) break foo if $caller_is("bar")
```

Adding a Pretty Printer

test.c

```
typedef struct element_s {
    int size;
    struct element_s *next;
    char* text;
} element;

typedef struct list_s {
    element *head;
    element *tail;
    int nelements;
} list;

...

void main() {
    list l;
    ...

    add_element(&l, "second");
    add_element(&l, "second");
}
```

Adding a Pretty Printer

test.c

```
typedef struct element_s {  
    int size;
```

print_list.py

```
import gdb  
  
def print_list(value):  
    frame = gdb.selected_frame()  
    try:  
        val = gdb.Frame.read_var(frame, value)  
    except:  
        print "No such variable"  
        return  
  
    if str(val.type) == "list":  
        print "List (%s elements):"%str(val["nelements"])  
        e = val["head"]  
        while (e != 0):  
            print str(e["text"].string()) + " --> ",  
            e = e["next"]  
        print "Null"
```

Adding a Pretty Printer

test.c

```
typedef struct element_s {  
    int size;
```

print_list.py

```
import gdb  
  
def print_list(value):  
    frame = gdb.selected_frame()  
    try:
```

```
} list
```

```
...
```

```
void t
```

```
except:  
    (gdb) python execfile("print_list.py")  
    (gdb) b <somewhere in main>  
    (gdb) run  
    Breakpoint 1, 0x0804865e in main () at test.c:75  
    75     addr = add_element(&l,"last");
```

```
(gdb) python print_list("l")  
START --> second --> third --> fourth → NULL
```

Adding a Pretty Printer

```
import gdb

class ListPrinter:
    def __init__(self, val):
        self.val = val

    def to_string(self):
        print "List (%s elements):"%str(self.val["nelements"])
        e = self.val["head"]
        while (e != 0):
            print str(e["text"].string()) + " -->",
            e = e["next"]
        print "Null"

def wxLookupFunction(val):
    print str(val.type)
    if str(val.type) == "list":
        return ListPrinter(val)
    return None

gdb.pretty_printers.append(wxLookupFunction)
```

Debugging & Profiling

1. Debugging Intro
2. GDB basics
3. GDB 7.0 new features
 - ✓ Reverse debugging
 - ✓ Scripting in Python
4. Profiling

valgrind

- Valgrind is an instrumentation framework for building dynamic analysis tools.
 - Nothing from the original program ever gets run directly on the host processor. Instead, Valgrind first translates the program into a temporary Intermediate Representation (IR)
 - After the conversion, a tool is free to do whatever transformations it would like on the IR, before Valgrind translates the IR back into machine code and lets the host processor run it
 - Some performance is lost in these transformations (at least 4-5 times slower)
- The Valgrind distribution currently includes some ready-to-use tools to detect:
 - memory errors
 - thread errors
 - heap/stack/global array overflow

Example

```
void f(void) {  
    int* x = malloc(10 * sizeof(int));  
    x[10] = 0;  
}
```

Example

```
void f(void) {  
    int* x = malloc(10 * sizeof(int));  
    x[10] = 0;  
}
```

```
> valgrind --leak-check=full myprog
```

```
...
```

```
Invalid write of size 4
```

```
at 0x804838F: f (test.c:3)
```

```
Address 0x418b050 is 0 bytes after a block of size 40 alloc'd
```

```
....
```

```
malloc/free: 1 allocs, 0 frees, 40 bytes allocated
```

```
...
```

```
40 bytes in 1 blocks are definitely lost in loss record 1 of 1
```

```
at 0x4022AB8: malloc (vg_replace_malloc.c:207)
```

```
by 0x8048385: f (test.c:2)
```

```
by 0x80483AC: main (test.c:7)
```

```
....
```

Profiling with GProf

- Profiling allows you to learn where your program spent its time and which functions called which other functions while it was executing
- Profiling a C/C++ program involves three steps:
 - Compile and link the program with profiling enabled
 - `-pg` option in gcc
 - Add extra code to the program to generate the profiler data
 - Execute the program to collect profile data
 - Run the program as usual
 - Profiler information are saved in a file called `gmon.out`
 - Run the gnu progiler (gprof) to analyze and print the profile data
 - `> gprof`

Profiling

```
> gprof
```

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
33.34	0.02	0.02	7208	0.00	0.00	open
16.67	0.03	0.01	244	0.04	0.12	offtime
16.67	0.04	0.01	8	1.25	1.25	memccpy
16.67	0.05	0.01	7	1.43	1.43	write
0.00	0.06	0.00	236	0.00	0.00	tzset
0.00	0.06	0.00	192	0.00	0.00	tolower
0.00	0.06	0.00	47	0.00	0.00	strlen

index	% time	self	children	called	name
[1]	100.0	0.00	0.05		start [1]
		0.00	0.05	1/1	main [2]
		0.00	0.00	1/2	on_exit [28]
		0.00	0.00	1/1	exit [59]

[2]	100.0	0.00	0.05	1/1	start [1]
		0.00	0.05	1	main [2]
		0.00	0.05	1/1	report [3]

[3]	100.0	0.00	0.05	1/1	main [2]
		0.00	0.05	1	report [3]
		0.00	0.03	8/8	timelocal [6]