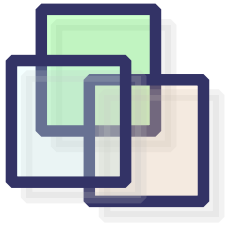
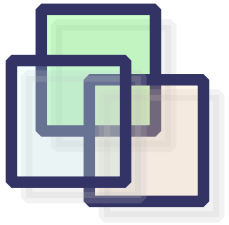


Debugging Tools

(Good knowledge of good tools spare time)

Emmanuel Fleury
LaBRI, Office 261
<emmanuel.fleury@labri.fr>



Introductory Example

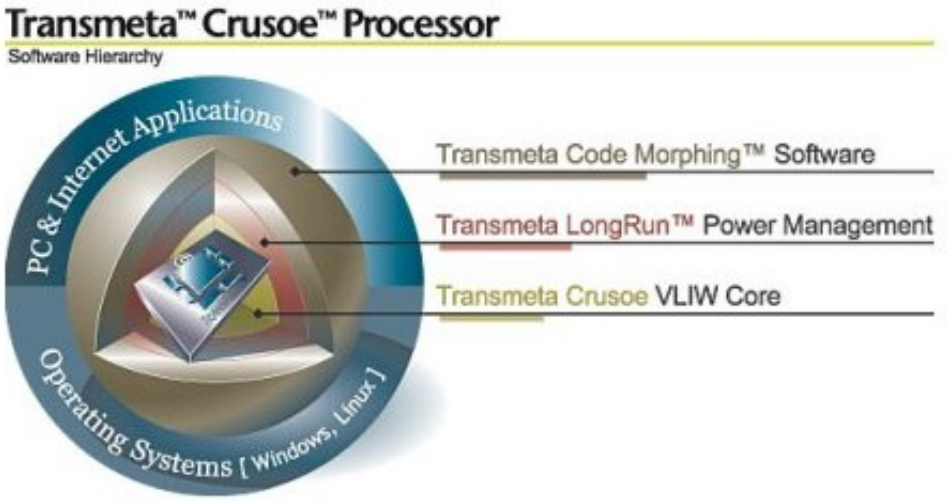
My Laptop is buggy...

```
X Error of failed request:  BadLength
  (poly request too large or internal Xlib length error)
Major opcode of failed request:  18 (X_ChangeProperty)
Serial number of failed request: 15
Current serial number in output stream:  18
```

- Occurs only 10% of the time at boot time...
- Bug is observed only on a specific serie of laptop (Transmeta Crusoe processors but not all of them)
- Once it occurs, it last until reboot;
- When traced the program seems to break down during the initial handshake network protocol.

Any idea ???

The bug explained...

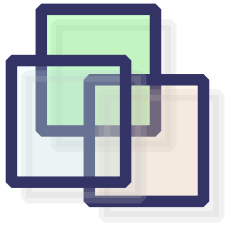


Optimization was the root of all evil once again...

- Code was first translated few times into VLIW.
- When executed many times, it was 'optimized' and cached.
- 10% of the time the 'optimizer' was buggy for a specific chain of x86 assembly instructions.
- Compiling the X server without optimization (at all) worked around the problem.

Why is debugging difficult ?

- You cannot trust anything !
  (your own code, libraries, compilers, OS, firmware, hardware)
- Longer is the chain of infection from defect to failure,
  harder it is !
- Bugs often lies where only few people understand
  what is going really on !
- Finally, bugs can be produced by the combination of several
  border side effects...



Inspiration

The great difficulty in debugging: You have to divorce yourself from preconceptions, make your mind blank, unlinked, unchanneld, the Zen state...

-- *Ellen Ullman* (The Bug)

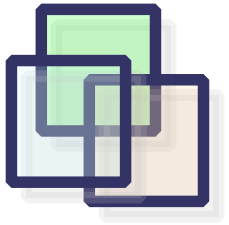
It is of the highest importance in the art of detection to be able to recognize, out of a number of facts, which are incidental and which vital.

-- *Conan Doyle*
*(*Sherlock Holmes, The Reigate Squire)

Debugging Tools

Types of Debugging Tools

- **Static Tools**
  (Looks only at the source code)
  - Static Analysis Tools: gcc, lint, splint, ...
  - More advanced Tools: Astrée, Blast, SLAM, ...
- **Dynamic without memory**
  (Looks at the binary but is only concerned by the current state)
  - GDB, JDB, ...
- **Dynamic with memory**
  (Looks at binary but keeps informations about whole execution)
  - Valgrind, purify

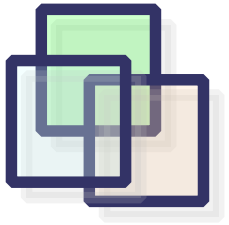


Debugging Tao

- Reproduce the bug
- Simplify its reproducibility conditions
- Trace it on a run (with gdb)
- Localize it in the code
- Theorize it **fully** !
- Check your theory (double check it) (triple check it) ....
- Fix it

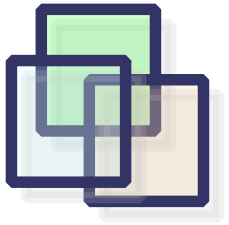
Patching Code

(`diff`, `patch`, ...)



Patching: What for ?

- Fixing bugs
- Add/Remove an experimental feature
- Refactoring code
- Customize a software
  (Netfilter Patch-O-Matic)
- Testing robustness of experimental features (quilt)



Patching Basics

- A patch represents the way to go from one file to another...
- Differences are '*line*' *based*
  (NOT 'word' based)
- *Hunk:* Set of consecutive modified lines
- A patch should always be *readable* !
- A patch should not break the program

Applying a Patch

```
[]$ patch -p1 source.c < patch.diff
missing header for unified diff at line 3 of patch
patching file source.c
```

Resulting file:

- source.c: The file merged

Option '**-p**' tells the depth of the location.
In practice, always '**-p1**'.

Applying Imperfect Patches

```
patching file src.c
Hunk #1 FAILED at 33.
Hunk #2 succeeded at 70 (offset 6 lines).
1 out of 2 hunks FAILED -- saving rejects to file src.c.rej
```

Resulting files:

- src.c: File with all the changes that can be applied
- src.c.orig: Original file without any change
- src.c.rej: All the changes that couldn't be applied

Reversing Patches

```
patch -p1 -R source.c < patch.diff
missing header for unified diff at line 3 of patch
patching file source.c
```

Reverse a patch (or a set of patches)

- Option '**-R**'

Resulting file:

- source.c (cleaned from the patch)

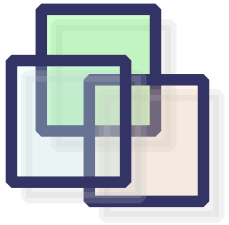
Reading a Patch

```
diff -BbuN source.c.orig source.c
--- source.c.orig   2007-09-25 00:07:38.000000000 +0200
+++ source.c        2007-09-25 00:07:10.000000000 +0200
@@ -70,7 +70,7 @@
 main (int argc, char **argv)
 {
   initialize_main (&argc, &argv);
-  program = argv[0];
+  program_name = argv[0];
   setlocale (LC_ALL, "");
   bindtextdomain (PACKAGE, LOCALEDIR);
   textdomain (PACKAGE);
```

Diffing Code

```
diff -BbuN source.c.orig source.c > mypatch.diff

diff -BbruN dir.orig dir.modif > mypatch.diff
```

- The result is always on **`stdout`**
  (redirection is needed)
- Recursive diff for directories ('**`-r`**')
  (diff for each file is stored in the same patch)
- Usual extension for patch is '**`.diff`**'



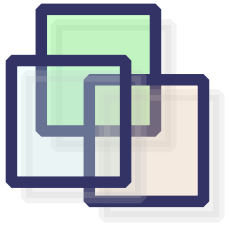
Cleaning a Patch

- Removing all non-necessary changes:
  - Re-indentation
  - New lines, tabs, space added / removed
  - Spelling fixes (if not relevant for the patch)
  - Renaming (variable, function, ...)
  - any other fix that is not relevant
- Keeping the same '*coding style*'
- Minimize your '*foot-prints*' on the code

Organizing a set of Patches

- ONE patch change ONE thing, no more !
  (Keep it stupid simple: KISS principle)
- Applying a patch won't break the whole
  (a patch is bringing code from a safe place to another)
- A set of patches have an order in which to apply (number the patches)
- Give them explicit name
  (e.g. **`010-renaming_variable_hwclock.diff`**)

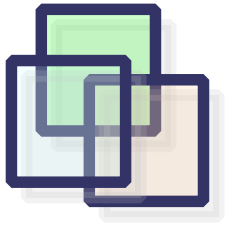
GNU Debugger (gdb)



What is a Debugger ?

A Debugger is a computer program that runs other programs on a instruction set simulator (ISS) and implements the following features:

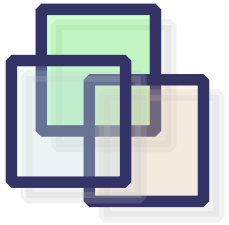
- Single-stepping (run step by step)
- Breaking (Stop execution)
- Value Tracking (memory inspection)
- Runtime Modification (runtime memory writing)
- Disassembling (assembly code inspection)
- Link symbolically the binary to the source code



Why use a Debugger ?

Why not use **`printf()`** everywhere ?

- It's **a lot** quicker (no rebuild needed)
- Minimize code modification (and thus bugs you may add)
- You can try ideas on-the-fly (runtime modification)
- You can attach to a running process without restarting it
- You can investigate core of a dead process
- You can track bugs in multi-threaded programs
- ...



GDB Capabilities

- Supported Languages
  - Full support on: C, C++, Objective-C, Fortran, Java and Assembly;
  - Partial support on: Modula-2 and Pascal;
- Supported Architectures

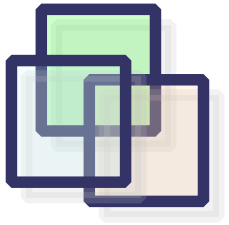
  Most of the one you can think of !
- Homepage:

  http://www.gnu.org/software/gdb/gdb.html

Preparing a binary for GDB

- **`-g`:** Produce debugging information in the operating system's native format (stabs, COFF, XCOFF, or DWARF 2).
- **`-ggdb`:** Produce debugging information in the most expressive format available (DWARF 2, stabs, or native format), including GDB extensions.
- **`-g[1-3]/-ggdb[1-3]`:** Request debugging information and also use level to specify how much information. Default level is 2.
  - **Level 1:** Produces minimal information for making backtraces in parts of the program that you don't plan to debug. This includes descriptions of functions and external variables, but no information on local variables nor line numbers.
  - **Level 3:** Includes extra information, such as all the macro definitions present in the program. Some debuggers support macro expansion when you use **`-g3`**

```
gcc -g3 -o hello hello.c
```



Initialization & Running

- Starting gdb on a binary:

```
$> gdb ./hello
```

- Running the binary (inside gdb):

```
(gdb) run
```

- Running binary with arguments (inside gdb):

```
(gdb) run arg1 arg2 arg3
```

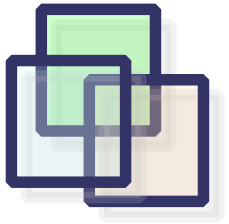
- Quit gdb:

```
(gdb) quit
```

"**r**" is the short end for "**run**"
"**q**" is the short end for "**quit**"

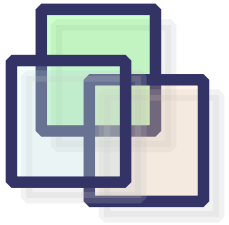
hello.c

```c
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char **argv) {
  int i;

  switch (argc) {
  case 1:
    printf("Hello World !\n");
    break;

  case 2:
    for (i=0; i<atoi(argv[1]); i++)
      printf("Hello World !\n");
    break;

  default:
    printf("hello: Too many arguments\n");
  }
  return 0;
}
```



Initialization & Running

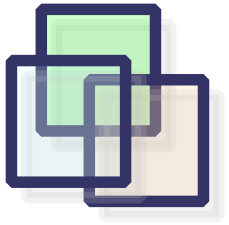
```
$> gdb ./hello
GNU gdb 6.5-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...Using host libthread_db library
"/lib/tls/i686/cmov/libthread_db.so.1".

(gdb) run
Starting program: /home/user/hello
Failed to read a valid object file image from memory.
Hello World !
Program exited normally.
(gdb) run 2
Starting program: /home/user/hello 2
Failed to read a valid object file image from memory.
Hello World !
Hello World !
Program exited normally.
(gdb)
```



Setting Arguments

```
...
(gdb) set args 3
(gdb) run
Starting program: /home/user/hello 3
Failed to read a valid object file image from memory.
Hello World !
Hello World !
Hello World !

Program exited normally.
(gdb)
```

Note that inside gdb, you can use:
- **History** ('up' and 'down');
- **Completion** ('tabulation');
- **Edit line** ('left' and 'right');
- **Repeat last command** ('return');
- **Repeat the command with old parameters** (if none given)



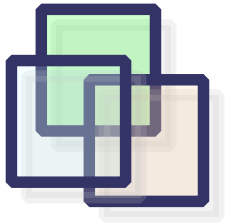
Getting help inside GDB

- **`help`**
  Display help on the help command;

- **`help command`**
  Display help on the command;

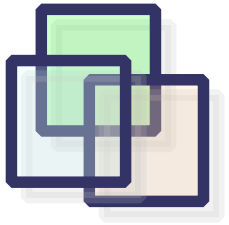
- **`help command_class`**
  Display help on the command class
  List of command classes is given in “help”.

“**`h`**” is the short end for “**`help`**”



Getting help inside GDB

```
(gdb) help
List of classes of commands:
aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping it
user-defined -- User-defined commands
Type "help" followed by a class name for a list of commands in the class
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb) help define
Define a new command name.  Command name is argument.
Definition appears on following lines, one command per line.
End with a line of just "end".
Use the "document" command to give documentation for the new command.
Commands defined in this way may have up to ten arguments.
```



Listing

- **`list / l`**
  List 10 lines of code centered on the current execution pointer;
- **`list - / list +`**
  List 10 lines of code backward/forward from the current execution pointer;
- **`list 8 / list 3,8`**
  List 10 lines of code centered on the line 8 or between 3 and 8;
- **`list func / list *0x80e4f7d`**
  List the code of the function func / at the given address;
- **`list file.c:12`**
  List 10 lines of code around the line 12 of the file file.c;
- **`list file.c:func`**
  List the code of the function func in file.c;

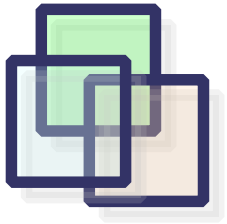
“**`l`**” is the short end for “**`list`**”

Listing

```
(gdb) list
1       #include <stdlib.h>
2       #include <stdio.h>
3
4       int main(int argc, char **argv) {
5         int i;
6         switch (argc) {
7         case 2:
8           for (i=0; i<atoi(argv[1]); i++)
9             printf("Hello World !\n");
10          break;
(gdb) list 10
5         int i;
6         switch (argc) {
7         case 2:
8           for (i=0; i<atoi(argv[1]); i++)
9             printf("Hello World !\n");
10          break;
11        case 1:
12          printf("Hello World !\n");
13          break;
14        default:
```

Setting List size

```
...
(gdb) set listsize 5
(gdb) list main
2   #include <stdio.h>
3
4   int main(int argc,char **argv){
5       int i;
6       switch (argc) {
(gdb)
```



Types of Program Points

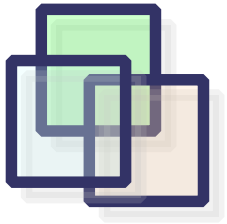
- **Breakpoints**
  Set at a precise point in the program;
  Stop the execution flow;
- **Watchpoints**
  Watch for the value of an expression along the run;
  Stop the execution flow;
- **Catchpoints**
  Watch for events (e.g. C++ exceptions) occurring along the run;
  Stop the execution flow;
- **Tracepoints**
  Trace program execution;
  Log the execution flow and do not stop the program;

Setting Breakpoints

Breakpoints are used to temporarily pause your program so you can inspect it.

- **`break`**
  Set breakpoint at current line;
- **`break 9 / break func`**
  Set breakpoint at line 9 / func();
- **`break file.c:10`**
  Set breakpoint line 10 in file file.c;
- **`break *0x807a543`**
  Set breakpoint at given address;
- **`break 5 i==2`**
  Set breakpoint line 5 which stop if i=2;
- **`condition 3 i>2`**
  Add the condition i>2 on breakpoint 3;
- **`info breakpoints 2`**
  Display informations on breakpoint 2;
- **`info breakpoints`**
  Display information on all breakpoints;
- **`continue`**
  Go to the next active breakpoint.

"**`b`**" is the short end for "**`break`**"
"**`i`**" is the short end for "**`info`**"
"**`c`**" is the short end for "**`continue`**"
"**`cond`**" is the short end for "**`condition`**"



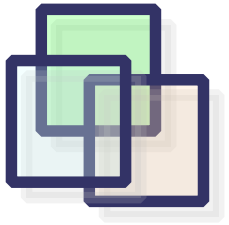
Setting Breakpoints

```
(gdb) b main
Breakpoint 1 at 0x8048398: file hello.c, line 6.
(gdb) b 7
Breakpoint 2 at 0x80483ae: file hello.c, line 7.
(gdb) b hello.c:17
Breakpoint 3 at 0x80483fa: file hello.c, line 17.
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint keep y   0x08048398 in main at hello.c:6
2   breakpoint keep y   0x080483ae in main at hello.c:7
3   breakpoint keep y   0x080483fa in main at hello.c:17
(gdb) run
Starting program: /home/user/hello
Failed to read a valid object file image from memory.
Breakpoint 1, main (argc=1, argv=0xbff39f94) at hello.c:6
6         switch (argc) {
(gdb) continue
Continuing.
Hello World !
Breakpoint 3, main (argc=1, argv=0xbff39f94) at hello.c:17
17        return 0;
```

Setting Watchpoints

A watchpoint is a special breakpoint that stops the program when the value of an expression changes.

- **`watch k`**
  Stop when k is written and modified;
- **`watch i>0`**
  Stop when i>0 value changes;
- **`watch (int *)0xff4168`**
  Stop when k is written and modified;
- **`rwatch x`**
  Stop when x is read;
- **`awatch j`**
  Stop when j is read or written;
- **`info watchpoints`**
  Display information on all watchpoints;



Setting Catchpoints

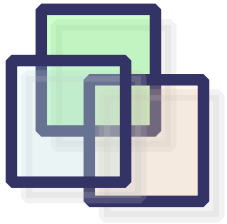
A catchpoint is a special breakpoint that stops your program when a certain kind of event occurs, such as the throwing of a C++ exception or the loading of a library.

- **`catch throw`**
  Stop when a C++ exception is thrown;
- **`catch catch`**
  Stop when a C++ exception is caught;
- **`catch exec`**
  Stop when an exec occurs;
- **`catch fork`**
  Stop when a fork occurs;
- **`catch vfork`**
  Stop when a vfork occurs;
- **`catch load`**
  Stop when a dynamic library is loaded;
- **`catch load libgtk2.0`**
  Stop when libgtk2.0 is loaded;
- **`catch unload`**
  Stop when a dynamic library is unloaded;
- **`catch unload libgtk2.0`**
  Stop when libgtk2.0 is unloaded.

Setting Tracepoints

Tracing of program execution without stopping the program.

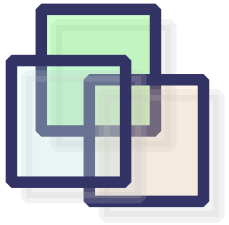
- **`action`**
  Specify the actions at a tracepoint;
- **`end`**
  Ends a list of commands or actions;
- **`collect`**
  Specify one or more data items to be collected at a tracepoint;
- **`passcount`**
  Set the passcount for a tracepoint;
- **`save-tracepoints`**
  Save current tracepoint definitions as a script;
- **`tdump`**
  Print everything collected at the current tracepoint;
- **`tfind`**
  Select a trace frame;
- **`tfind <cmd>`**
  Select a trace frame based on <cmd>;
- **`trace`**
  Set a tracepoint at a specified line or function or address;
- `...` Many others.



Ignore, Disable & Enable

- **`ignore 2 3`**
  Ignore next 3 crossings of breakpoint 2.
- **`disable 2`**
  Disable breakpoint 2;
- **`enable 2`**
  Enable breakpoint 2;

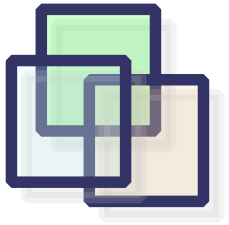
```
(gdb) disable 1
(gdb) ignore 2 4
Will ignore next 4 crossings of breakpoint 2.
(gdb) info break
Num Type           Disp  Enb  Address    What
1   breakpoint     keep  n    0x08048398 in main at hello.c:6
        breakpoint already hit 1 time
2   breakpoint     keep  y    0x080483ae in main at hello.c:7
        ignore next 4 hits
3   breakpoint     keep  y    0x080483fa in main at hello.c:17
        breakpoint already hit 1 time
(gdb) enable 1
(gdb) info break
Num Type           Disp  Enb  Address    What
1   breakpoint     keep  y    0x08048398 in main at hello.c:6
        breakpoint already hit 1 time
2   breakpoint     keep  y    0x080483ae in main at hello.c:7
        ignore next 4 hits
3   breakpoint     keep  y    0x080483fa in main at hello.c:17
        breakpoint already hit 1 time
```



Clear & Delete

- **`delete`**
  Remove all breakpoints;
- **`delete 2`**
  Remove breakpoint 2;
- **`clear 9`**
  Clear all breakpoints at line 9;
- **`clear func`**
  Clear all breakpoints at func();
- **`clear *0x807a543`**
  Clear all breakpoints at the given address;
- **`clear file.c:10`**
  Clear all breakpoints at line 10 in file file.c;

```
...
(gdb) clear main
Deleted breakpoint 1
(gdb) clear 7
Deleted breakpoint 2
(gdb) delete 3
(gdb) info break
No breakpoints or watchpoints
(gdb)
```

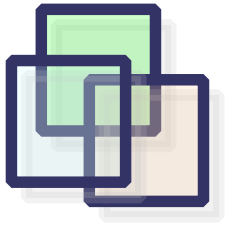


Stepping

Control the way you step through the program !

- **`step`**
  Execute a single line in the program. If the current statement calls a function, the function is single stepped;
- **`next`**
  Execute a single line in the program but treat function calls as a single line. This command is used to skip over function calls;
- **`until`**
  Reach the end of the current loop in one step;
- **`where`**
  Print where you are in the source code;
- **`kill`**
  Exit from the current execution.

“**`s`**” is the short end for “**`step`**”
“**`n`**” is the short end for “**`next`**”
“**`w`**” is the short end for “**`where`**”
“**`k`**” is the short end for “**`kill`**”



Stepping

```
(gdb) run
Starting program: /home/user/hello
Failed to read a valid object file image from memory.

Breakpoint 1, main (argc=1, argv=0xbfc2d494) at hello.c:6
6           switch (argc) {
(gdb) step
12              printf("Hello World !\n");
(gdb)
Hello World !
13              break;
(gdb) next
17          return 0;
(gdb) kill
Kill the program being debugged? (y or n) y
(gdb)
```

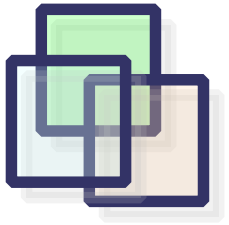
Display Program Variables

- **`ptype var`**
  Display the type of var;
- **`print i`**
  Display the value of i;
- **`print (char) i`**
  Display value of (char) i (cast);
- **`print &i`**
  Display the address of i;
- **`print *ptr`**
  Display the content of ptr;
- **`print i-(j*k)`**
  Display the value of i-(j*k);
- **`print $`**
  Display the value of the previous display request;
- **`print $3`**
  Display the value of the 3rd display request;
- **`display x`**
  Display the value of x at each step (variable tracking);
- **`undisplay 2`**
  Cancel display of the variable 2;
- **`info display`**
  List all the tracked variables.

"**`p`**" is the short end for "**`print`**"
"**`disp`**" is the short end for "**`display`**"

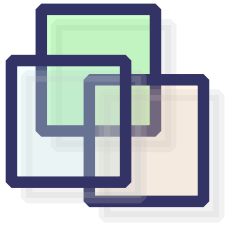
Display Complex Structures

- **`print array`**
  Display the array;
- **`print array[3]@5`**
  Display 5 elements of array starting at element 3;
- **`print strct`**
  Display the structure strct;
- **`print strct.name`**
  Display the content of the field name in strct;
- **`print strct->value`**
  Display the content of the field value in strct.



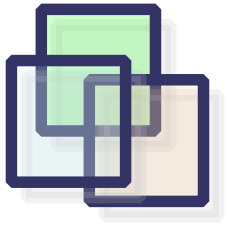
Setting Display Format

- **`print /o i`**
  Octal format;
- **`print /x ptr`**
  Hexadecimal format;
- **`print /d k`**
  Decimal format;
- **`print /f x`**
  Float format;
- **`print /u i`**
  Unsigned format;
- **`print /t mask`**
  Binary format;
- **`print /a i`**
  Address format;
- **`print /c i`**
  Character format;
- **`print /s str`**
  String format;
- **`print 4/c s`**
  Display variable as 4 chars;
- **`print /3x str`**
  Display the first 24 bytes of str;
- **`x /d 0xbfdd25c0`**
  Display content of the address.



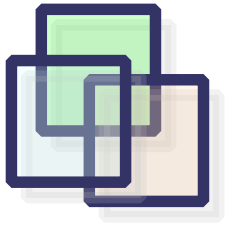
Setting Variables

- **`set variable i=5`**
  Set i to 5;
- **`set variable x=10.0`**
  Set x to 10.0;
- **`set variable array={1,2,3}`**
  Set array to {1,2,3};
- **`set variable strc={id="foo",value=3}`**
  Set structure strc to {id="foo", value=3}.



Investigating the stack frame

- **`backtrace`**
  Display the stack frame structure;
- **`up 2 / down 2`**
  Go 2 steps up / down in the stack frame (if no argument '1' is assumed);
- **`frame 3`**
  Display frame level 3 and jump to it (if no argument '1' is assumed);
- **`return`**
  Force the current stack frame to return to its caller;
- **`info frame / stack`**
  Display all informations about the current stack frame;
- **`info scope`**
  Display the variables local to a scope;
- **`info locals`**
  Display local variables of current stack frame.
- **`info args`**
  Display argument variables of current stack frame;



Display CPU Registers

- **`info all-registers`**
  List of all registers and their contents;
- **`info registers`**
  List of integer registers and their contents;
- **`info float`**
  Print the status of the floating point unit;
- ...

Inspection at Assembly Level

- **`disassemble func`**
  Disassemble the function func();
- **`disassemble 0xbfdd25c0`**
  Disassemble the surrounding of this address;
- **`disassemble 0xbfdd25c0 0xbfdd25e0`**
  Disassemble from the first address up to the second;
- **`nexti`**
  Perform a next at the assembly instruction level;
- **`stepi`**
  Perform a step at the assembly instruction level;
- **`jump 2 / jump 0xbbf3ac4`**
  Set the instruction pointer to the line 2 / address 0xbbf3ac4.

loop.c

```
#include <stdio.h>

int main(int argc, char **argv)
{
   int i=1;
   while (i);
   printf("Hello World !\n");
   return 0;
}
```

`loop` is already *running* and will *never stop*.
How to get access to it and debug it on-the-fly ?

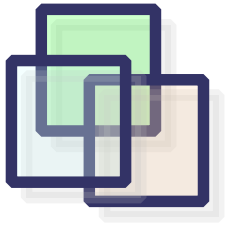
Debugging Running Processes

- Attach to a running process (method 1):

```
#> gdb ./loop 17399
...
(gdb)
```

- Attach to a running process (method 2):

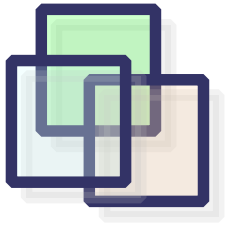
```
#>gdb
...
(gdb) attach 17399
(gdb) detach
```



Analyzing a core

- When your program segfaults and leaves a core dump file, you can use gdb to look at the program state when it crashed. Use the core command to load a core file. The argument to the core command is the filename of the core dump file, which is usually "core", making the full command core core.

```
prompt > myprogram
Segmentation fault (core dumped)
prompt > gdb ./myprogram
...
(gdb) core core
...
```



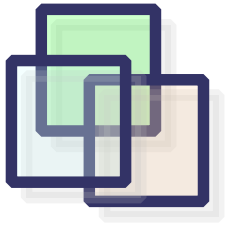
Signal Handling

- **`handle signal [keywords...]`**
  Change the way GDB handles signals. signal can be the id of a signal or its name (with or without the `SIG' prefix); a list of signal numbers of the form `low-high'; or the word `all', meaning all the known signals. Optional arguments keywords are:
  - **`nostop`**
    GDB should not stop your program when this signal happens.
  - **`stop`**
    GDB should stop your program when this signal happens.
  - **`print`**
    GDB should print a message when this signal happens.
  - **`noprint`**
    GDB should not mention the occurrence of the signal at all. Implies nostop.
  - **`pass / noignore`**
    GDB should allow your program to see this signal;
  - **`nopass / ignore`**
    GDB should not allow your program to see ts signal.

Debugging Multi-threaded Programs

TODO !

A Full Example



Summary

- Software: Tesseract-OCR
  (developed by HP and freed few years after development stopped);

- Problem:
  When compiled in 64 bits mode, program segfaults without producing any output !
  - Reproducibility is easy (with a 64 bits machine).
  - Correct behaviour is obvious (prevent the segfault)

Lets start the bug hunt !!!

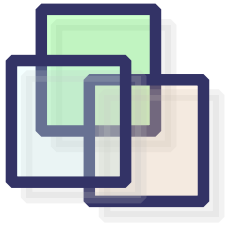
Running with Debug Enabled

- I used the configure for this:

```
$> ./configure --enable-debug && make
```

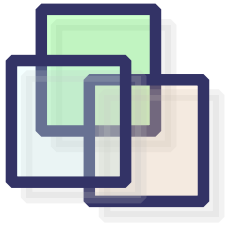
- Running the software
(without going through the installation):

```
$> cd ccmain/;
$> ln -s ../tessdata;
$> ./tesseract ../phototest.tif test.txt
Segmentation fault
```



Running in gdb

```
$ gdb ./tesseract

GNU gdb 6.5-debian
Copyright (C) 2006 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 "x86_64-linux-gnu"...Using host libthread_db library
"/lib/libthread_db.so.1".

(gdb) set args ../phototest.tif test.txt
(gdb) run
Starting program: /tesseract-1.02/ccmain/tesseract ../phototest.tif test.txt

Program received signal SIGSEGV, Segmentation fault.
0x00000000004d873e in reverse32 (ptr=0x6db000) at callcpp.cpp:251
251       tmp = *cptr;
(gdb)
```

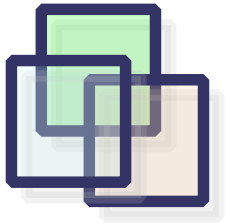


Where Are We ?

```
...
(gdb) backtrace
#0  0x00000000004d873e in reverse32 (ptr=0x6db000) at callcpp.cpp:251
#1  0x00000000004c2f3c in read_squished_dawg (
    filename=0x7fffd99615c0 "/tesseract-1.02/ccmain/tessdata/freq-dawg",
    dawg=0x6c6560, max_num_edges=1000) at dawg.cpp:288

#2  0x00000000004bb200 in init_permdawg () at permdawg.cpp:337
#3  0x00000000004be91c in init_permute () at permute.cpp:1001
#4  0x0000000000498a48 in init_ms_debug () at msmenus.cpp:86
#5  0x00000000004926ff in program_editup (configfile=0x0) at tface.cpp:79
#6  0x0000000000492749 in start_recog (configfile=0x0,
    textbase=0x7fffd9962968 "test.txt")  at tface.cpp:67
#7  0x00000000004042ab in init_tesseract (arg0=0x7fffd9962908
    "/tesseract-1.02/ccmain/tesseract", textbase=0x7fffd9962968
    "test.txt", configfile=0x0, configc=0, configv=0x7fffd9962038)
    at tessedit.cpp:125
#8  0x0000000000403125 in main (argc=3, argv=0x7fffd9962028)
    at tesseractmain.cpp:70
(gdb)
```

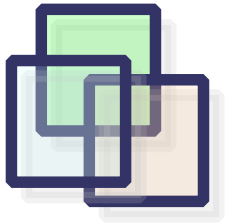
reverse32()

```
...
(gdb) list 247,257
247     void reverse32(void *ptr) {
248       char tmp;
249       char *cptr = (char *) ptr;
250
251       tmp = *cptr;
252       *cptr = *(cptr + 3);
253       *(cptr + 3) = tmp;
254       tmp = *(cptr + 1);
255       *(cptr + 1) = *(cptr + 2);
256       *(cptr + 2) = tmp;
257     }
(gdb) p ptr
$1 = (void *) 0x6db000
(gdb) p tmp
$2 = 0 '\0'
(gdb) p cptr
$3 = 0x6db000 <Address 0x6db000 out of bounds>
(gdb)
```



Where Are We ? (Take 2)

```
...
(gdb) backtrace
#0  0x00000000004d873e in reverse32 (ptr=0x6db000) at callcpp.cpp:251
#1  0x00000000004c2f3c in read_squished_dawg (
    filename=0x7fffd99615c0 "/tesseract-1.02/ccmain/tessdata/freq-dawg",
    dawg=0x6c6560, max_num_edges=1000) at dawg.cpp:288

#2  0x00000000004bb200 in init_permdawg () at permdawg.cpp:337
#3  0x00000000004be91c in init_permute () at permute.cpp:1001
#4  0x0000000000498a48 in init_ms_debug () at msmenus.cpp:86
#5  0x00000000004926ff in program_editup (configfile=0x0) at tface.cpp:79
#6  0x0000000000492749 in start_recog (configfile=0x0,
    textbase=0x7fffd9962968 "test.txt")  at tface.cpp:67
#7  0x00000000004042ab in init_tesseract (arg0=0x7fffd9962908
    "/tesseract-1.02/ccmain/tesseract", textbase=0x7fffd9962968
    "test.txt", configfile=0x0, configc=0, configv=0x7fffd9962038)
    at tessedit.cpp:125
#8  0x0000000000403125 in main (argc=3, argv=0x7fffd9962028)
    at tesseractmain.cpp:70
(gdb) #1  0x00000000004c2f3c in read_squished_dawg (
    filename=0x7fffd99615c0 "/tesseract-1.02/ccmain/tessdata/freq-dawg",
    dawg=0x6c6560, max_num_edges=1000) at dawg.cpp:292
292             reverse32(&dawg[edge]);
(gdb)
```



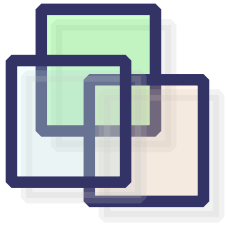
read_squished_dawg()

```
(gdb) list 270,296
270     void read_squished_dawg(char *filename,EDGE_ARRAY dawg,
                                INT32 max_num_edges) {
271       FILE       *file;
272       EDGE_REF   edge;
273       INT32      num_edges, node_count = 0;
274       if (debug) print_string ("read_debug");
275       clear_all_edges(dawg, edge, max_num_edges);
276       #ifdef __UNIX__
277       file = open_file (filename, "r");
278       #else
279       file = open_file (filename, "rb");
280       #endif
281       fread (&num_edges,  sizeof (int),      1, file);
282       if (__NATIVE__==INTEL)
283         reverse32(&num_edges);
284       fread (&dawg[0], sizeof (EDGE_RECORD), num_edges, file);
285       fclose(file);
286       if (__NATIVE__==INTEL)
287         for (edge=0;edge<num_edges;edge++)
288           reverse32(&dawg[edge]);
289       for  (edge=0; edge<max_num_edges; edge++)
290         if (last_edge (dawg, edge)) node_count++;
291     }
```

Inspecting read_squished_dawg()

```
...
(gdb)  print edge
$4 = 21160
(gdb) print max_num_edges
$5 = 1000
(gdb) print num_edges
$6 = 140733193388211
(gdb)
```

Ooops, **num_edges** seems to be really really wrong !



read_squished_dawg()

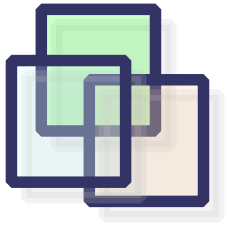
```
void read_squished_dawg(char *filename, EDGE_ARRAY dawg, INT32 max_num_edges) {
  FILE        *file;
  EDGE_REF edge;
  INT32       num_edges;
  INT32       node_count = 0;

  if (debug) print_string ("read_debug");
  clear_all_edges(dawg, edge, max_num_edges);
#ifdef __UNIX__
  file = open_file (filename, "r");
#else
  file = open_file (filename, "rb");
#endif
  fread (&num_edges,  sizeof (int), 1, file);
  if (__NATIVE__==INTEL) reverse32(&num_edges);
  fread (&dawg[0], sizeof (EDGE_RECORD), num_edges, file);
  fclose(file);
  if (__NATIVE__==INTEL)
    for (edge=0;edge<num_edges;edge++)
      reverse32(&dawg[edge]);
  for  (edge=0; edge<max_num_edges; edge++)
    if (last_edge (dawg, edge)) node_count++;
}
```

`INT32` is in fact 64-bit long

`int` is 32-bit long

The Problem & The Fix !


```
diff -ruN tesseract/dict/dawg.cpp tesseract-ef/dict/dawg.cpp
--- tesseract/dict/dawg.cpp     2006-06-17 00:17:07.0000 +0200
+++ tesseract-new/dict/dawg.cpp 2006-11-25 11:28:21.0000 +0100
@@ -270,7 +270,7 @@
   #ifdef __UNIX__
     file = open_file (filename, "r");
   #else
     file = open_file (filename, "rb");
   #endif
-    fread (&num_edges,  sizeof (int), 1, file);
+    fread (&num_edges,  sizeof (INT32), 1, file);
     if (__NATIVE__==INTEL) reverse32(&num_edges);
     fread (&dawg[0], sizeof (EDGE_RECORD), num_edges, file);
     fclose(file);
```



Summary

- Time to understand the bug:
  20 minutes
- Time to fix the bug:
  2 minutes
- Number of compilations:
  2
- Number of modifications in the code:
  One line

Can you do better with **`printf()`** ???

Another Full Example

Using Watchpoints

Valgrind

TODO !

Splint

TODO !