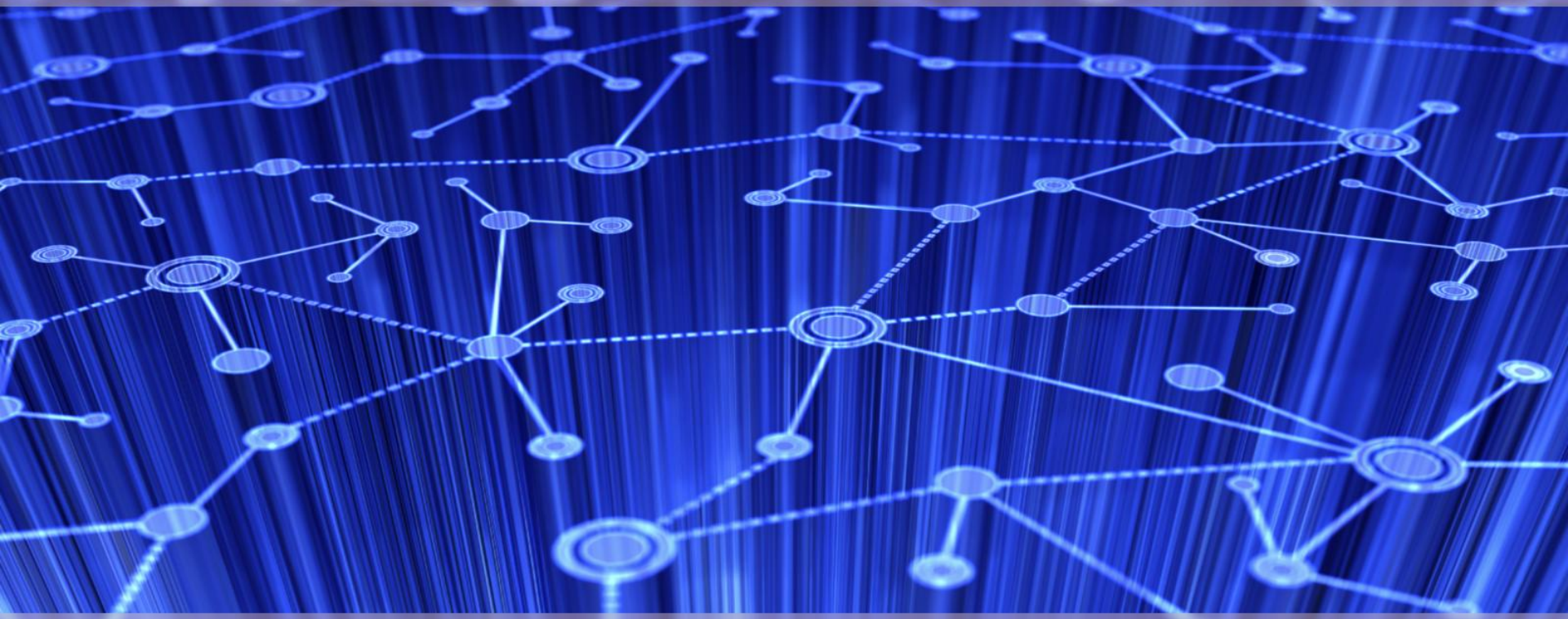


Foundations of High Performance Computing

Luca Tornatore

Debugging with gdb

serial, single-threaded, applications

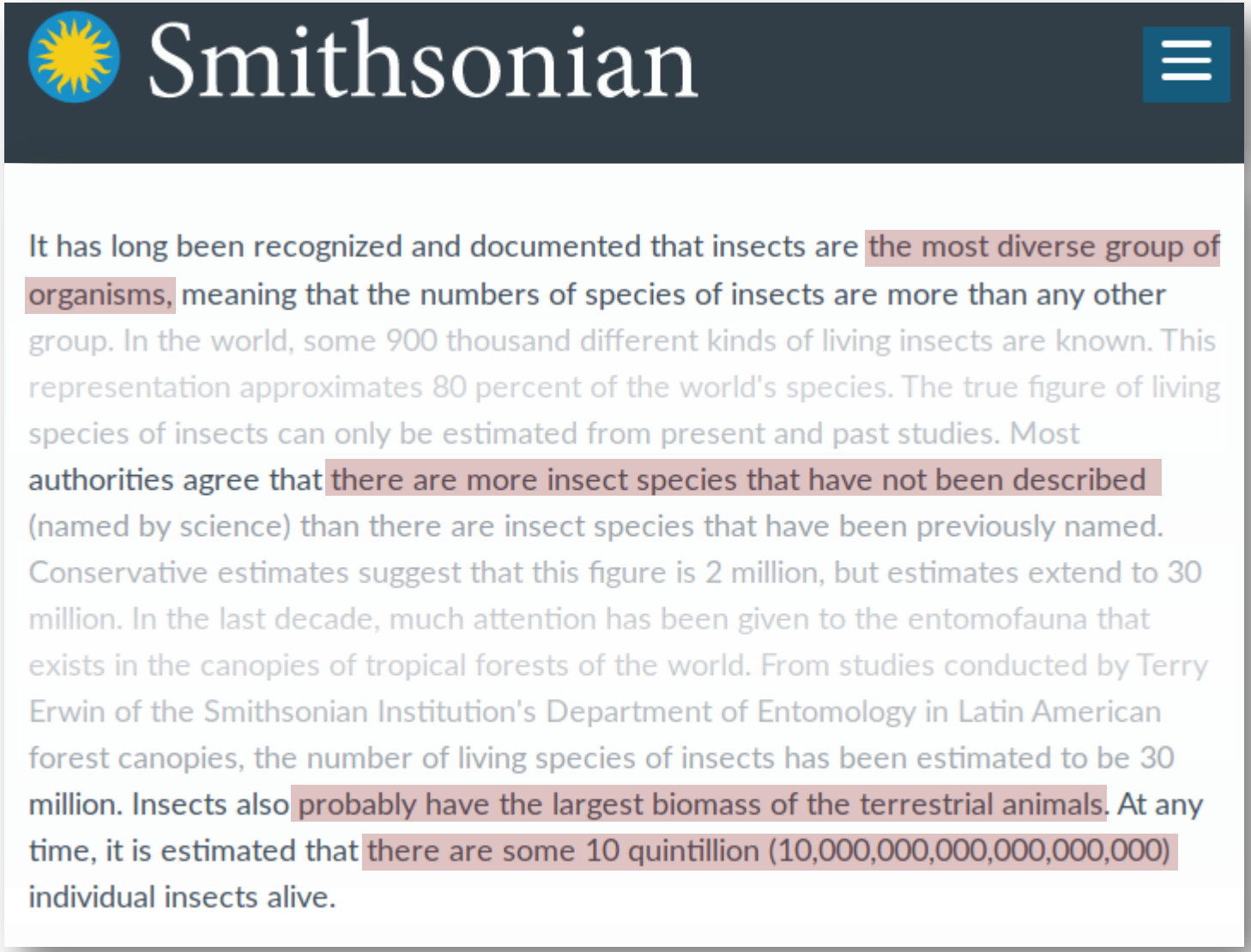
Outline

Introducing Debugging

GDB basic How To

GUI for GDB

Introducing debugging

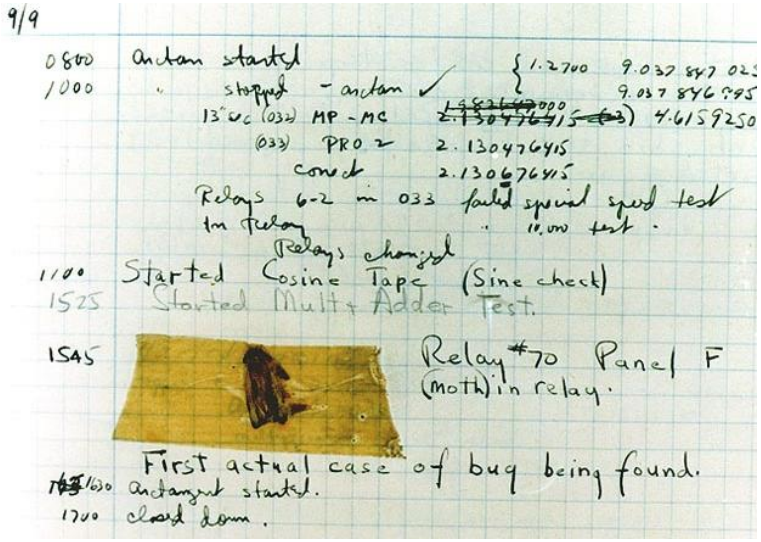


Introducing debugging

The usage of ***bug*** and, hence, of ***de-bugging*** referred to programming is a long-standing tradition, whose origin is difficult to trace back.

An often-told story is about Mark-II calculator located at Harvard: On Sept. 9th, 1945, a technician found a moth in a relay that caused a flaw in a "program" execution.

Adm. Grace Hopper is reported to have written in its diary about that as "first *actual* case of bug being found"

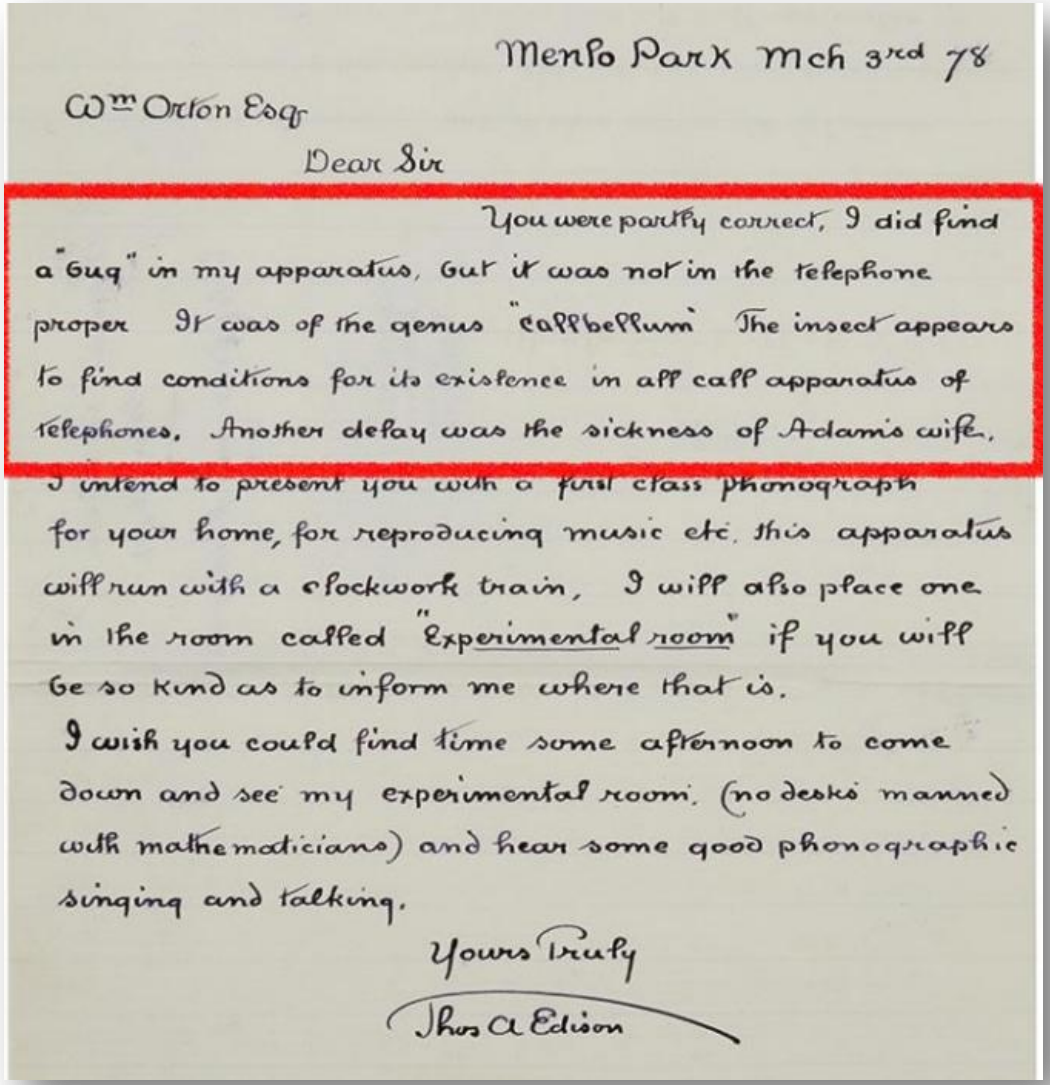


Introducing debugging

However, **Thomas Edison** found another *actual* bug in one of his phones, as he reports in a letter to an associate.

He later wrote:

"It has been just so in all of my inventions. The first step is an intuition and comes with a burst, then difficulties arise—this thing gives out and [it is] then that "Bugs"—as such little faults and difficulties are called—show themselves and months of intense watching, study and labor are requisite before commercial success or failure is certainly reached."



Menlo Park Mch 3rd 78

Wm Orton Esqr

Dear Sir

You were partly correct, I did find a "bug" in my apparatus, but it was not in the telephone proper It was of the genus "cappbellum" The insect appears to find conditions for its existence in all call apparatus of telephones. Another delay was the sickness of Adams wife.

I intend to present you with a first class phonograph for your home, for reproducing music etc. this apparatus will run with a clockwork train, I will also place one in the room called "Experimental room" if you will be so kind as to inform me where that is.

I wish you could find time some afternoon to come down and see my experimental room, (no desks manned with mathematicians) and hear some good phonographic singing and talking.

Yours Truly

Thos A Edison

Introducing debugging

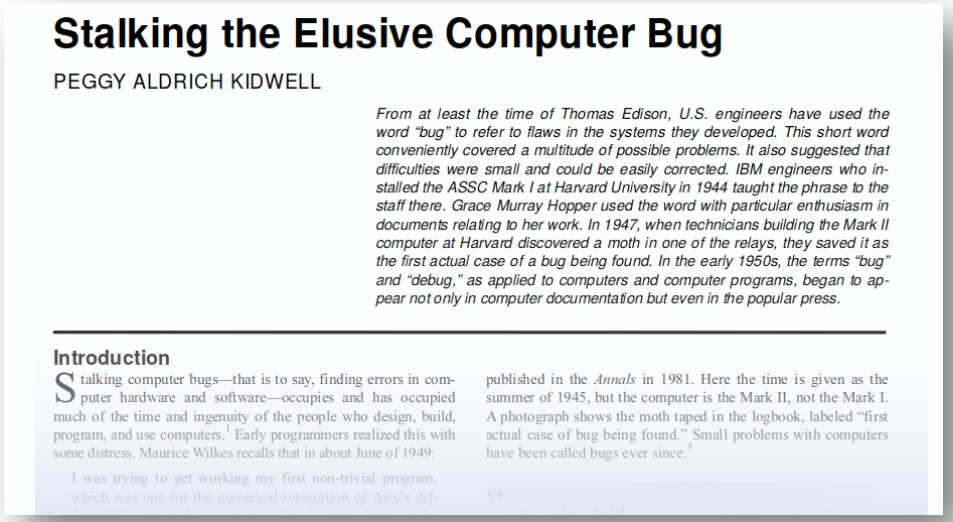
It is difficult to trace back how this term has been poured into computer programming jargon.

However, already early in the 60s' it was appearing in technical papers without need of explanation.

The immortal I. Asimov used it in a 1944 short robot story "Catch that rabbit", and his incredible influence contributed much to make the term popular.

More funny infos:
IEEE Annals of the History of Computing
(Volume: 20 , Issue: 4 , Oct-Dec 1998)

ask for the PDF if you're interested and do not have access



Stalking the Elusive Computer Bug

PEGGY ALDRICH KIDWELL

From at least the time of Thomas Edison, U.S. engineers have used the word "bug" to refer to flaws in the systems they developed. This short word conveniently covered a multitude of possible problems. It also suggested that difficulties were small and could be easily corrected. IBM engineers who installed the ASSC Mark I at Harvard University in 1944 taught the phrase to the staff there. Grace Murray Hopper used the word with particular enthusiasm in documents relating to her work. In 1947, when technicians building the Mark II computer at Harvard discovered a moth in one of the relays, they saved it as the first actual case of a bug being found. In the early 1950s, the terms "bug" and "debug," as applied to computers and computer programs, began to appear not only in computer documentation but even in the popular press.

Introduction

Stalking computer bugs—that is to say, finding errors in computer hardware and software—occupies and has occupied much of the time and ingenuity of the people who design, build, program, and use computers.[1] Early programmers realized this with some distress. Maurice Wilkes recalls that in about June of 1949:

> I was trying to get working my first non-trivial program, which was one for the numerical integration of Airy's dif-

published in the *Annals* in 1981. Here the time is given as the summer of 1945, but the computer is the Mark II, not the Mark I. A photograph shows the moth taped in the logbook, labeled "first actual case of bug being found." Small problems with computers have been called bugs ever since.[5]

Introducing debugging

> *"It has been just so in all of my inventions. The **first step** is an intuition and comes with a burst, **then difficulties arise**—this thing gives out and [it is] then that "**Bugs**"—as such little faults and difficulties are called—show themselves and **months of intense watching, study and labor are requisite** before commercial success or failure is certainly reached."*
>
> *T. Edison*

That maybe is too dramatic and emphatic, but the point to get from Edison is that the *de-bugging activity* is an inherent and intrinsic one in software development

Introducing debugging

Provided that

- you'll have close encounters with bugs in your life;
- de-bugging is a fundamental and unavoidable part of your work;

what is the best way to proceed ?

Introducing debugging

1. Do not insert bugs

   Highly encouraged, but rarely works

Introducing debugging

1. Do not insert bugs
   Highly encouraged, but rarely works

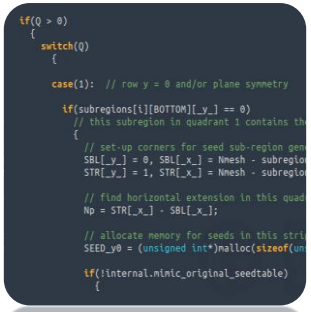
2. Add **`printf`** statements everywhere
   Highly discouraged, but sometimes works.
   In case of memory problems – tipically due to pointers chaos – you may see that the problem "disappear", or changes its appearance, when you insert a new `printf`

Introducing debugging

1. Do not insert bugs
   Highly encouraged, but rarely works
2. Add `printf` statements everywhere
   Highly discouraged, but sometimes works.
   In case of memory problems – tipically due toaos
3. Use a DEBUGGER
   That is definitely the best choice and, fortunately, the subject of this lecture.

gdb is almost certainly the best free, extremely feature-rich command-line debugger ubiquitously available on *niX systems.

Outline



GDB
How To

GUI
for GDB

What GDB is useful for

There are 3 basic usages(*) of GDB:

1. Debugging a code
   best if it has been compiled with –g
2. Inspecting a code crash through a core file
3. Debugging / inspecting a running code

(*) Higly advised: learn keyboard commands. Although many gui exist (we'll see some later), keyboard is still the best productivity tool.

What GDB is useful for

1.Debugging a code

Compiling with dbg infos

In order to include debugging information in your code, you need to compile it with `-g` family options (read the gcc manual for complete info):

`-g`	produce dbg info in O.S. native format
`-ggdb`	produce gdb specific extended info, as much as possible
`-g`*`level`*	default level is 2. 0 amounts to no info, 1 is minimal, 3 includes extra information (for instance, macros expansion) – this allows macro expansion; add **`-gdwarf-`*`n`*** in case, where possibile, where *`n`* is the maximum allowed (4)
`-ggdb`*`level`*	you can combine the two to maximize the amount of useful info generated

Debugging a code : start

You just start your code under `gdb` control:

```
%> gdb program
```

You may define already the arguments needed by your program at invocation:

```
%> gdb --args program arg1 arg2 … argN
```

Or you can define the arguments from within the gdb session:

```
%> gdb program
Reading symbols from program...done.
(gdb) set args arg1 arg2 … argN
(gdb) run
```

▶ live demo with **`gdb_try_args.c`**

Debugging a code : run and stop

You may just want the code to run, for instance to reach the point of a seg fault:

```
(gdb) run
```

Or, you may want to stop it from the beginning to have full control of each step:

```
(gdb) break main
(gdb) run
```

Or you may already know what is the problematic point to stop at:

```
(gdb) break location
(gdb) run
```

Debugging a code : run and stop

Breakpoints are a key concept in debugging. They are stopping point at which the execution interrupts and the control is given back to you, so that you can inspect the memory contents (variables values, registers values, …) or follow the subsequent execution step by step.

You can define a breakpoint in several way

`(gdb) break`	insert a break at the current pos
`(gdb) break ±offset`	insert a break *offset* lines after/before the current line
`(gdb) filename:linenum`	insert a break at *linenum* of file *filename*
`(gdb) break functionname`	insert a break at the entry point of function *functionname*

```
There are more options,
just check the manual
```

Debugging a code : run and stop

A break point may be defined as dependent on a given condition:

```
(gdb) break my_function if (arg1 > 3 )
```

This sets a breakpoint at function *my_function* : the condition will be evaluated each time the point is reached, and the xecution is stopped only if it is true.

Condition can be any valid expression.

```
(gdb) info break
```

gives you informations on active breakpoints.

```
(gdb) delete [n]
      clear [location]
      <disable | enable>  see the manual
```

Debugging a code : run and stop

You can define a list of commands to be executed when a given breakpoint is reached:

```
(gdb) break my_function if (arg1 > 3 )
Breakpoint 1 at 0x……: file blabla.c, line 42
(gdb) command 1
Type commands for when breakpoint 1 is hit, one per
line.
End with a line saying just "end".
> print arg1
> print another_useful_variable
> x/10wd a_global_integer_array
```

Debugging a code : run and stop

When you have the control of the program execution, you can decide how to proceed:

`(gdb) cont [c]`	continue until the end / next stop
`(gdb) cont` *`count-ignore`*	continue ignoring the next *count-ignore* stops (for instance, occurrences of a bp)
`(gdb) next [n]	` *`count`* `nexti`	continue to the next src line *in the current stack frame*
`(gdb) step [s]	` *`count`* `stepi`	continue to the next src line
`(gdb) until [u]	` *`count`*	continue until a src line past the current one is reached in the current stack fr.
`(gdb) advance` *`location`*	continue until the specified location is reached

▶ live demo with **`gdb_try_breaks.c`**

Debugging a code : run and stop

You can also rewind your execution step-by-step

```
(gdb) reverse-continue [rc]
(gdb) reverse-step [count]
      reverse-stepi
(gdb) reverse-next [count]
      reverse-nexti
(gdb) set exec-direction <verse | forward >
```

Debugging a code : source list

Often, when you are debugging, you may have the need of looking at either the source lines or at the generated assembler:

`(gdb) list` *`linenum`*	print src lines around line *linenum* of the current source file
`(gdb) list` *`function`*	print the source lines of *function*
`(gdb) list` *`location`*	print src lines around *location*
`(gdb) set listsize` *`count`*	control the number of src lines printed

▶ live demo with **`gdb_try_breaks.c`**

Debugging a code : stack examination

Examining the stack is often of vital importance. With GDB you can have a quick and detailed inspection of all the stack frames.

`(gdb) backtrace[args]`	print the backtrace of the whole stack
n	print only the *n* innermost frames
`-n`	print only the *n* outermost frames
full	print local variables value, also
`where, info stack`	additional aliases

▶ live demo with **`gdb_try_breaks.c`**

Debugging a code : memory examination

Accessing to the content of memory is a fundamental ability of a debugger. You have several different ways to do that:

`(gdb)print` *variable* `p/F` *variable*	print the value of *variable* print *variable* in a different format (`x,d,u,o,t,a,c,f,s`) *see then manual for advanced location*
`(gdb)x/FMT` *address*	Explore memory starting at address *address* -> see at live demo how to use this
`(gdb)display` *expr* `display/`*fmt expr* `display/`*fmt addr*	Add *expr* to the list of expressions to display each time your program stops

▶ live demo with **`gdb_try_breaks.c`**

Debugging a code : memory examination

Memory can be searched to find a particular value, of a given size

```
(gdb) find [/sn] start,
end, val1 [,val2, …]

      find [/sn] start,
+len, val1 [,val2, …]
```

Search memory for a particular sequence of bytes.
s is the size of type to be searched
n is the max number of occurrences

Debugging a code : registers examination

Examining registers may be also useful (although it's something that only quite advanced users can conceive)

`(gdb) info registers`	print the value of all registers
`(gdb) info vector`	print the content of vector registers
`(gdb) print $rsp`	print value of the stack pointer
`(gdb) x/10wd $rsp`	print values of the first 10 4-bytes integers on the stack
`(gdb) x/10i $rip`	print the next 10 asm instructions

▶ live demo with **`gdb_try_breaks.c`**

Debugging a code : macro expansion

If there are macros in your code, they can be expanded, provided that you compiled the code with the appropriate option:

`-g3 [gdb3][-dwarf-4]`

`(gdb) macro expand` *macro*	shows the expansion of macro *macro*; *expression* can be any string of tokens
`(gdb) info macro [-a|-all]` *macro*	shows the current (or all) definition(s) of *macro*
`(gdb) info macros` *location*	shows all macro definitions effective at *location*

Debugging a code : watch points

You can set *watchpoints* (aka "keep an eye on this and that") instead of breakpoints, to stop the execution whenever a value of an expression / variable/memory region changes

`(gdb) watch variable`	keep an eye onto *variable*
`(gdb) watch expression`	stops when the value of expression changes (the scope of variables is respected)
`(gdb) watch -l expression`	Interpret *expression* as a memory location to be watched
`(gdb) watch -l expression [mask maskvalue]`	a mask for memory watching: specifies what bits of an address should be ignored (to match more addressese)
`(gdb) rwatch [-l] expr [mask mvalue]`	stops when the value of *expr* is read

What GDB is useful for

2. Inspecting a code crash through a core file

After a crash : having the core file

It happens that you have code crashes in conditions not easily reproducible when you debug the code itself, for a number of reasons.

However, the O.S. can dump the entire “program status” on a file, called the *core file*:

```
luca@GGG:~/code/tricks% ./gdb_try_watch
no arguments were given, using default: 100

something wrong at point 2
[1]    8435 segmentation fault (core dumped)  ./gdb_try_watch
luca@GGG:~/code/tricks% ls -l core
-rw------- 1 luca luca 413696 nov  8 15:45 core
luca@GGG:~/code/tricks%
```

In order to allow it to dump the core, ou have to check / set the core file size limit:

```
%> ulimit -c [size limit in KB]
```

After a crash: inspect the core file

Once you have a core, you can inspect it with GDB

```
%> gdb executable_name core_file_name
```

Or

```
%> gdb executable_name
(gdb)core core_file_name
```

The first thing to do, normally, is to unwind the stack frame to understand where the program crashed:

```
(gdb) bt full
```

▶ live demo with **gdb_try_watch.c**

What GDB is useful for

3. Debugging a running process

Running processes: attaching

In order to debug a running process, you can simply attach gdb to it:

```
%> gdb
(gdb) attach process-id
```

and start searching it to understand what is going on

▶ live demo with **gdb_try_attach.c**

Outline

Introducing Debugging

GDB How To

GUI for GDB

GDB graphical user interfaces

1. GDB text-user-interface
2. GDB DASHBOARD
3. GDBGUI
4. EMACS
5. DDD
6. NEMIVER
7. ECLIPSE, NETBEANS, CODEBLOCKS

GDB built-in tui

You can start gdb with a text-user-interface:

```
%> gdb -tui
```

Or you can activate/deactivate it from gdb itself:

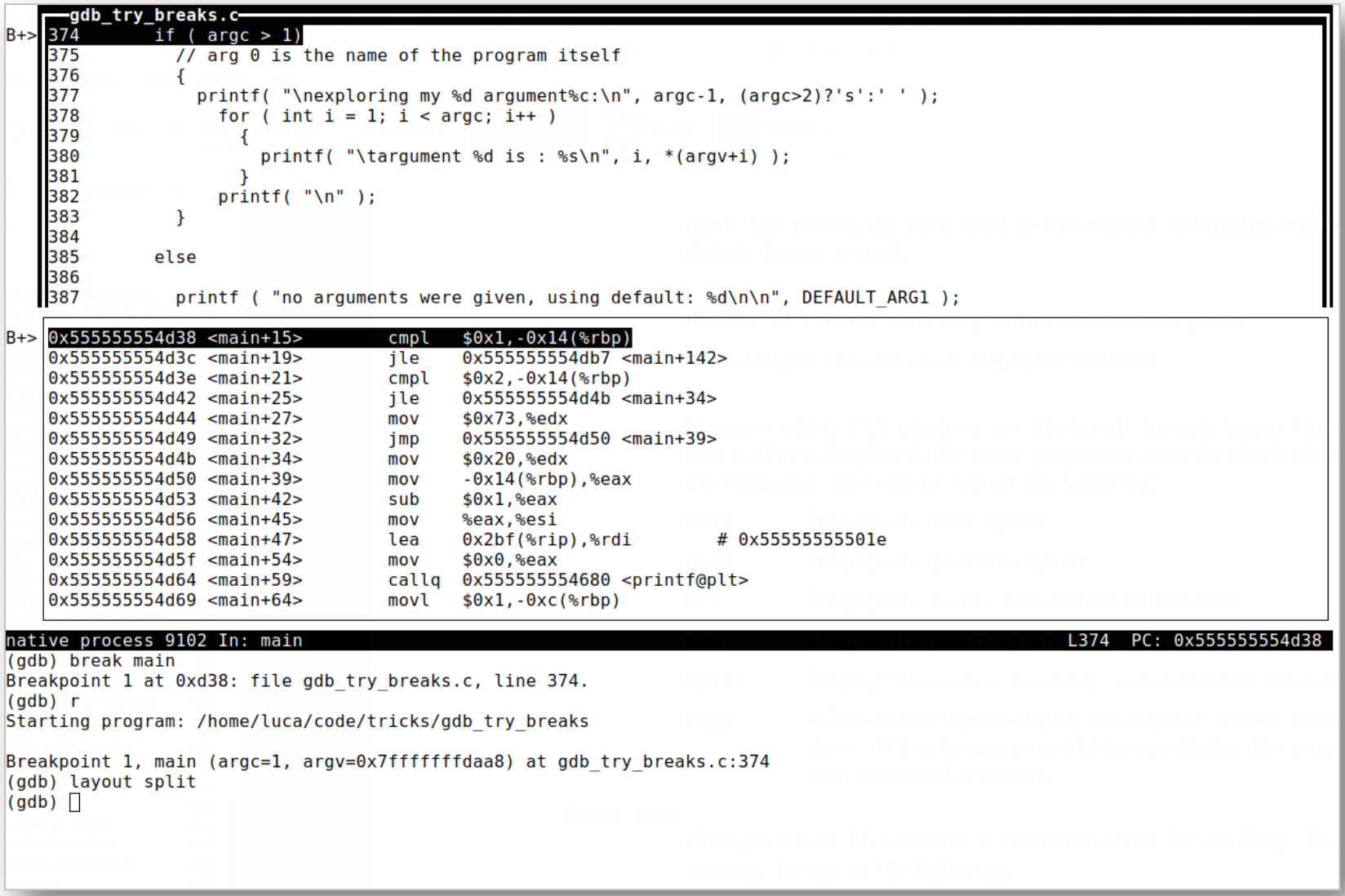
`(gdb) ctrl-x a`	
`(gdb) ctrl-x o`	change focus
`(gdb) ctrl-x 2`	shows assembly windows
`(gdb) layout src`	display src and commands
`asm`	display assembly and commands
`split`	display src, asm and commands
`regs`	display registers window

GDB built-in tui

```
─gdb_try_breaks.c──
372     {
373
B+>374        if ( argc > 1)
375          // arg 0 is the name of the program itself
376          {
377            printf( "\nexploring my %d argument%c:\n", argc-1, (argc>2)?'s':' ' );
378              for ( int i = 1; i < argc; i++ )
379                {
380                  printf( "\targument %d is : %s\n", i, *(argv+i) );
381                }
382              printf( "\n" );
383          }
384
385        else
386
387          printf ( "no arguments were given, using default: %d\n\n", DEFAULT_ARG1 );
388
389
390        int arg1;
391
392        if ( argc > 1 )
393          arg1 = atoi( *(argv+1) );
394
395        else
396          arg1 = DEFAULT_ARG1;
397
398        int ret;
399
400        ret = function_1( arg1 );
native process 8943 In: main                                    L374  PC: 0x555555554d38
(gdb) l
360     in /home/luca/code/tricks/gdb_try_breaks.c
(gdb) break main
Breakpoint 1 at 0xd38: file gdb_try_breaks.c, line 374.
(gdb) r
Starting program: /home/luca/code/tricks/gdb_try_breaks

Breakpoint 1, main (argc=1, argv=0x7fffffffdaa8) at gdb_try_breaks.c:374
(gdb)
```

GDB built-in tui



GUI for GDB

GDB dashboard

```
bash
── Source ──
2
3 void fun(int n, char *data[])
4 {
5     int i;
6
7     for (i = 0; i < n; i++) {
8         printf("%d: %s\n", i, data[i]);
9     }
10 }
11
12 int main(int argc, char *argv[])
── Assembly ──
0x0000000100000f18 b0 00          fun+56 mov    $0x0,%al
0x0000000100000f1a e8 4b 00 00 00 fun+58 callq  0x100000f6a
0x0000000100000f1f 89 45 e8       fun+63 mov    %eax,-0x18(%rbp)
0x0000000100000f22 8b 45 ec       fun+66 mov    -0x14(%rbp),%eax
0x0000000100000f25 05 01 00 00 00 fun+69 add    $0x1,%eax
0x0000000100000f2a 89 45 ec       fun+74 mov    %eax,-0x14(%rbp)
0x0000000100000f2d e9 c4 ff ff ff fun+77 jmpq   0x100000ef6 <fun+22>
0x0000000100000f32 48 83 c4 20    fun+82 add    $0x20,%rsp
0x0000000100000f36 5d             fun+86 pop    %rbp
0x0000000100000f37 c3             fun+87 retq
── Threads ──
[1] id 4355 from 0x0000000100000f2a in fun+74 at scrot.c:7
── Stack ──
[0] from 0x0000000100000f2a in fun+74 at scrot.c:7
arg n = 3
arg data = 0x7fff5fbffb60
loc i = 1
[1] from 0x0000000100000f62 in main+34 at scrot.c:14
arg argc = 3
arg argv = 0x7fff5fbffb60
(no locals)
── Registers ──
   rax 0x0000000000000002          rbx 0x0000000000000000          rcx 0x0000010000000203
   rdx 0x0000020000000200          rsi 0x0000000000012068          rdi 0x00007fff79e86118
   rbp 0x00007fff5fbffb20          rsp 0x00007fff5fbffb00           r8 0x0000000000000040
    r9 0x00007fff79e86110          r10 0xffffffffffffffff          r11 0x0000000000000246
   r12 0x0000000000000000          r13 0x0000000000000000          r14 0x0000000000000000
   r15 0x0000000000000000          rip 0x0000000100000f2a       eflags [ TF IF ]
    cs 0x0000002b                   ss <unavailable>                ds <unavailable>
    es <unavailable>                fs 0x00000000                   gs 0x00000000
── Expressions ──
[1] data[i] = 0x7fff5fbffcf0 "hello"
── Memory ──
0x00007fff5fbffb0c 01 00 00 00 60 fb bf 5f ff 7f 00 00 00 00 00 00 ....`.._........
0x00007fff5fbffb1c 03 00 00 00 40 fb bf 5f ff 7f 00 00 62 0f 00 00 ....@.._....b...
0x00007fff5fbffb2c 01 00 00 00 60 fb bf 5f ff 7f 00 00 03 00 00 00 ....`.._........
0x00007fff5fbffb3c 00 00 00 00 50 fb bf 5f ff 7f 00 00 ad 15 f7 9c ....P.._........
0x00007fff5fbffcf0 68 65 6c 6c 6f 00 47 44 42 00 54 45 52 4d 5f 50 hello.GDB.TERM_P
── History ──
$$1 = 63
$$0 = {[0] = 0x7fff5fbffcf0 "hello", [1] = 0x7fff5fbffcf6 "GDB"}
```

```
gdb
>>> dashboard -output /dev/ttys001
>>> dashboard -layout
source
assembly
threads
stack
registers
expressions
memory
history
>>> p data[1]@2
$3 = {[0] = 0x7fff5fbffcf0 "hello", [1] = 0x7fff5fbffcf6 "GDB"}
>>>
```

https://github.com/cyrus-and/gdb-dashboard

▶ live demo

GUI for GDB

GDBGUI

gdbgui Download Docs Examples Screenshots Videos About Chat GitHub

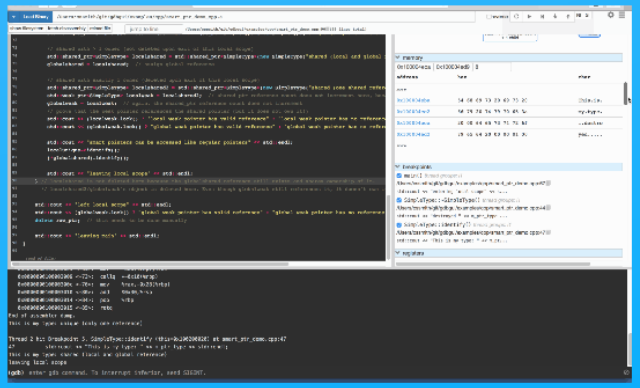
gdbgui

Browser-based debugger for C, C++, go, rust, and more

gdbgui turns this

```
>> gdb
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
>>>
```

into this



Download Now »

https://gdbgui.com/

▶ live demo

GDB graphical user interfaces

1. GDB text-user-interface
2. GDB DASHBOARD
3. GDBGUI
4. EMACS GUD
5. DDD
6. NEMIVER
7. ECLIPSE, NETBEANS, CODEBLOCKS

GDB graphical user interfaces

1. GDB text-user-interface
2. GDB DASHBOARD
3. GDBGUI
4. EMACS GUD
5. DDD
6. NEMIVER
7. ECLIPSE, NETBEANS, CODEBLOCKS

GDB graphical user interfaces

1. GDB text-user-interface
2. GDB DASHBOARD
3. GDBGUI
4. EMACS GUD
5. DDD
6. NEMIVER
7. ECLIPSE, NETBEANS, CODEBLOCKS