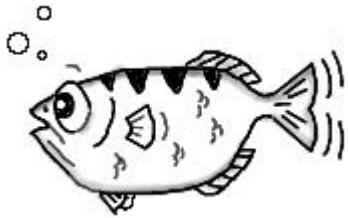
GDB



intro a un dibaguer.

qué hemos estado haciendo?

- pensamos lo que queríamos lograr.
- describimos la funcionalidad.
- escribimos código fuente.
- generamos código de máquina ejecutable.
- testeamos el ejecutable.

qué puede pasar?

- no compila!
- no funciona!
- funciona mal!

tenemos info de porqué no compila.

?

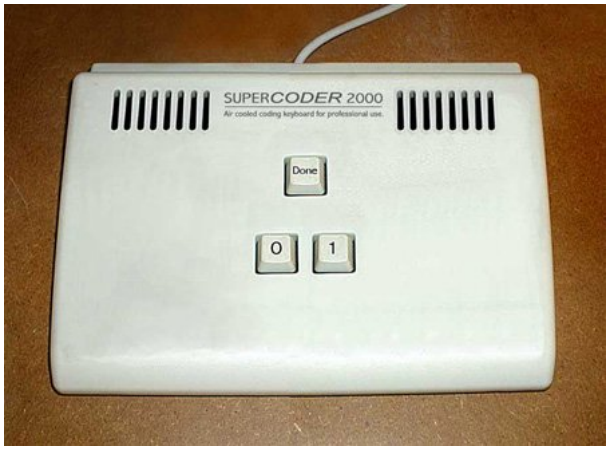
?

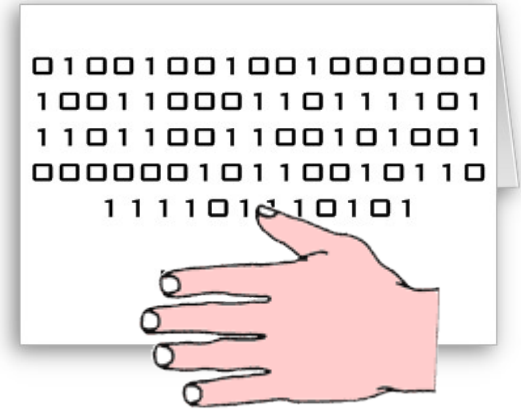
usemos un depurador.

a los problemas los podemos llamar /bags/ (del inglés *bugs* que es bichos).

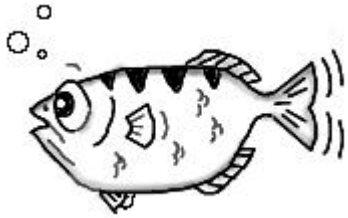
un depurador /dibaguer/ (del inglés *debugger*) en una herramienta de desarrollo para depurar, purificar, limpiar errores.


1 Edición
Programa Código Fuente
Editor
2 Traducción
Programa Objeto
Compilador
3 Enlace
Programa Ejecutable
Enlazador
Bibliotecas


SUPERCODER 2000
Done
0
1



GDB

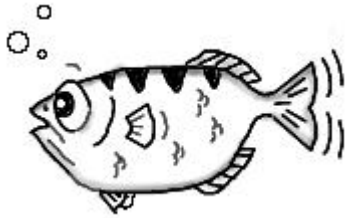


GNU debugger, creado por R. Stallman.

es un depurador simbólico que corre C, C++, Java,... y ahora sí:

- podemos ejecutar línea por línea.
- desglosar instrucción por instrucción.
- inspeccionar variables, cambiarlas!
- evaluar guardas o expresiones.
- si falla el programa, sabemos a dónde.
- podemos ir siguiendo el código fuente.
- recoger un programa que murió.
- depurar un proceso en ejecución.

GDB

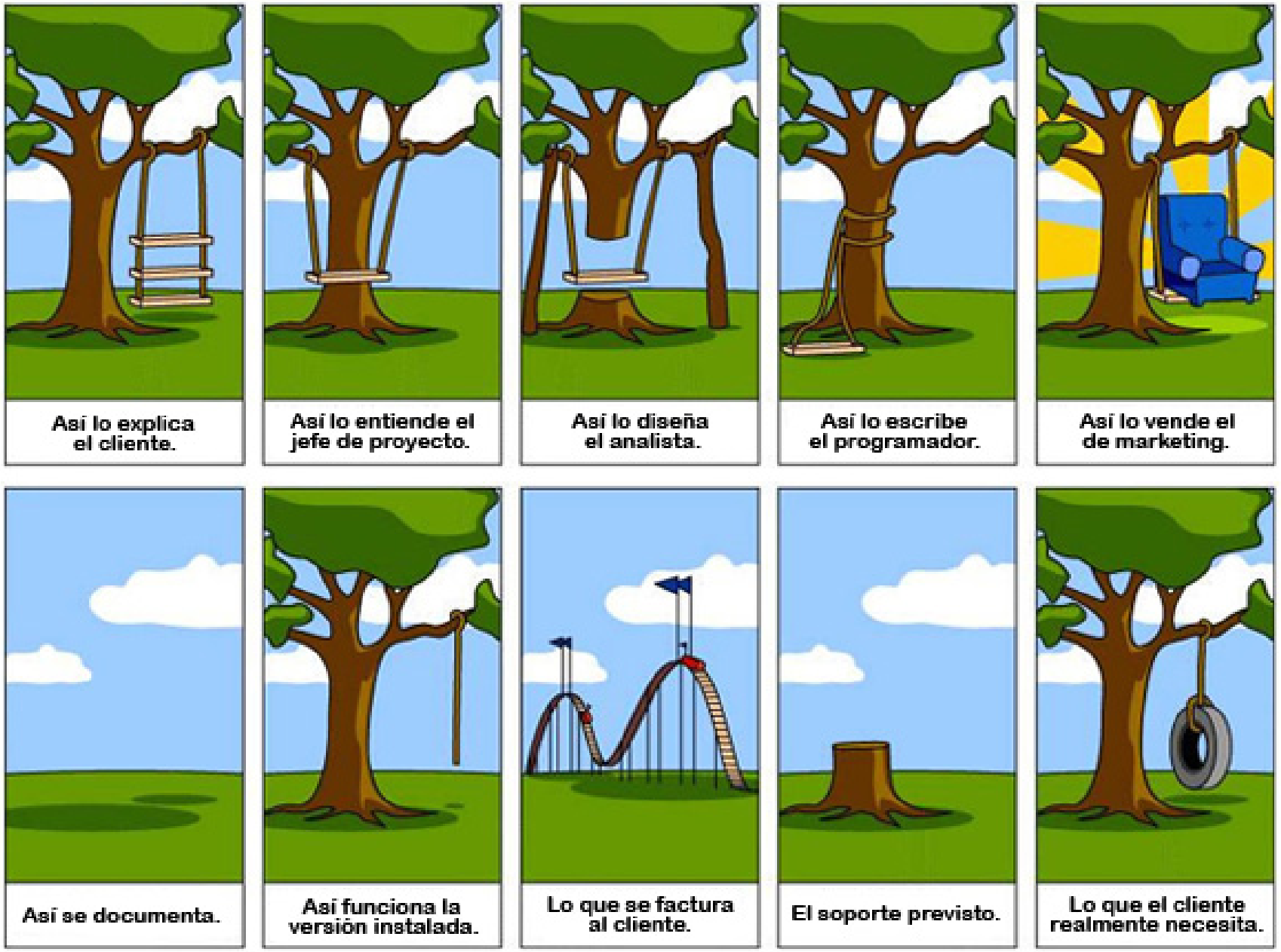


GNU debugger, creado por R. Stallman.

es un depurador simbólico que corre C, C++, Java,... y ahora sí:

- nos ayuda.
  - a entender qué está pasando
  - relaciona los distintos niveles del desarrollo.
  - incentiva un código fuente prolijo y claro.
  - lograr un producto de calidad.
  - imprescindible para proyectos grandes, que necesitan soporte y mantenimiento.

+info
http://www.gnu.org/software/gdb/gdb.html
http://www.dirac.org/linux/gdb/


Así lo explica el cliente.
Así lo entiende el jefe de proyecto.
Así lo diseña el analista.
Así lo escribe el programador.
Así lo vende el de marketing.
Así se documenta.
Así funciona la versión instalada.
Lo que se factura al cliente.
El soporte previsto.
Lo que el cliente realmente necesita.

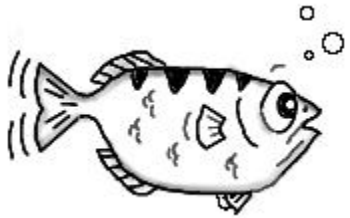
preparando el camino...

necesitamos enriquecer la tabla simbólica.
cuando compilamos será con **-g**.

$ gcc -Wall -Wextra -pedantic -ansi -g -c main.c

$ gcc -Wall -Wextra -pedantic -ansi -g -o main main.o

iniciando una sesión.

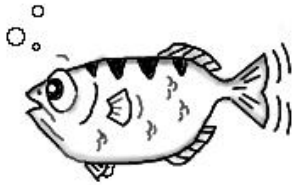


$gdb [programa]
carga gdb desde la terminal y nos espera un cursor **(gdb)**.

un ENTER pelado repite el último comando.
=D funciona TAB para autocompletar,
las flechas para retomar comandos anteriores.

(gdb) help [comando]
está bueno empezar pidiendo ayuda.
(gdb) file proy3
cargo el ejecutable proy3 con el comando /fail/.

correr el programa.



(gdb) run

/ran/ (run)

resultado!

- el programa se ejecutará normalmente.
- puede pasar que se interrumpa el programa por el sistema y nos diga en qué parte se interrumpió y porqué.

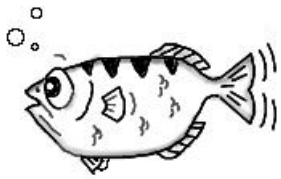
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400524 in sum array region (arr=0x7fffc902a270, r1=2, c1=5, r2=4, c2=6) at sum-array-region2.c:12

ok, pero esto tá re aburrido.

qué pasa si hay bicho?

quiero detectar de dónde surgen!

- quiero frenar la ejecución del proceso en determinados lugares.
- monitorear cambios en variables.
- inspeccionar el estado en algún lugar de la ejecución.



creando un /breikpoint/.

sería como una parada (del inglés breakpoint)

(gdb) break 8
Breakpoint 1 at 0x8048464: file main.c, line 8.

(gdb) break operaciones.c:6
Breakpoint 2 at 0x80483fd: file operaciones.c, line 6.

(gdb) break resto_div_ent
Breakpoint 3 at 0x80485a5: file main.c, line 15.

paradas.

lista todas las paradas...

(gdb) info breakpoints

```
Num Type           Disp Enb Address    What
  1   breakpoint     keep y   0x08048464 in main at main.c:8
        breakpoint already hit 1 time
  2   breakpoint     keep y   0x0804846b in main at main.c:9
        breakpoint already hit 1 time
  3   breakpoint     keep y   0x08048477 in main at main.c:12
```

(gdb) disable 3

/diseibl/ desactiva la 3.

(gdb) enable 3

/ineibl/ la vuelve a activar.

eliminar paradas...

(gdb) clear 8

(gdb) clear operaciones.c:6

(gdb) clear resto_div_ent

(gdb) clear 8

borra la parada de la línea 8.

(gdb) delete 1

borra la parada con id 1.

(gdb) delete

borra todas!

controlar la ejecución.

algunos comandos para seguir adelante.

(gdb) run

ejecuta desde el principio, frena si cruza una parada.

(gdb) continue

sigue hasta la próxima parada.

(gdb) next

ejecuta la próxima línea de código, y frena.

(gdb) step

como next, pero se desvía con las llamadas a funciones.

(gdb) finish

ejecuta hasta salir de la función.

para saber dónde estoy.

pregunto...

(gdb) where
/uear/ me responde en qué función línea y archivo estoy.

(gdb) list [n]
muestra diez líneas de código centradas en n.

(gdb) list [-]
muestra más código para adelante o para atrás.

(gdb) list resto_div_ent
muestra diez líneas centradas en la función.

(gdb) list *0x08048464
muestra más código para adelante o para atrás.

abrir los ojos.

queremos frenar en las paradas para poder ver un poco el estado de ejecución!

(gdb) ptype var
/pitaip/ nos cuenta el tipo de la variable var.
(o simplemente **pt var**)

(gdb) print [/d|/f|/c] var
muestra el valor de la variable var con el formato pedido.
(o simplemente **p var**)

watchpoints.

podemos hacer que se frene la ejecución siempre que haya un cambio en alguna variable!

(gdb) watch var

/uoch/ crea una parada y vale lo que vimos para paradas.

(gdb) info breakpoints

```
(gdb) info breakpoints
        Num Type           Disp Enb Address    What
        1   breakpoint     keep y   0x080483f6 in main at try5.c:4
                                    breakpoint already hit 1 time
        2   hw watchpoint  keep y   i
```

cambiar valores de variables.

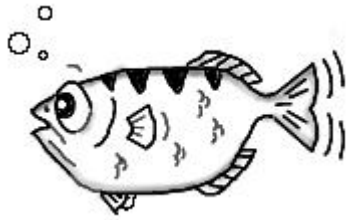
esto puede ser interesante.

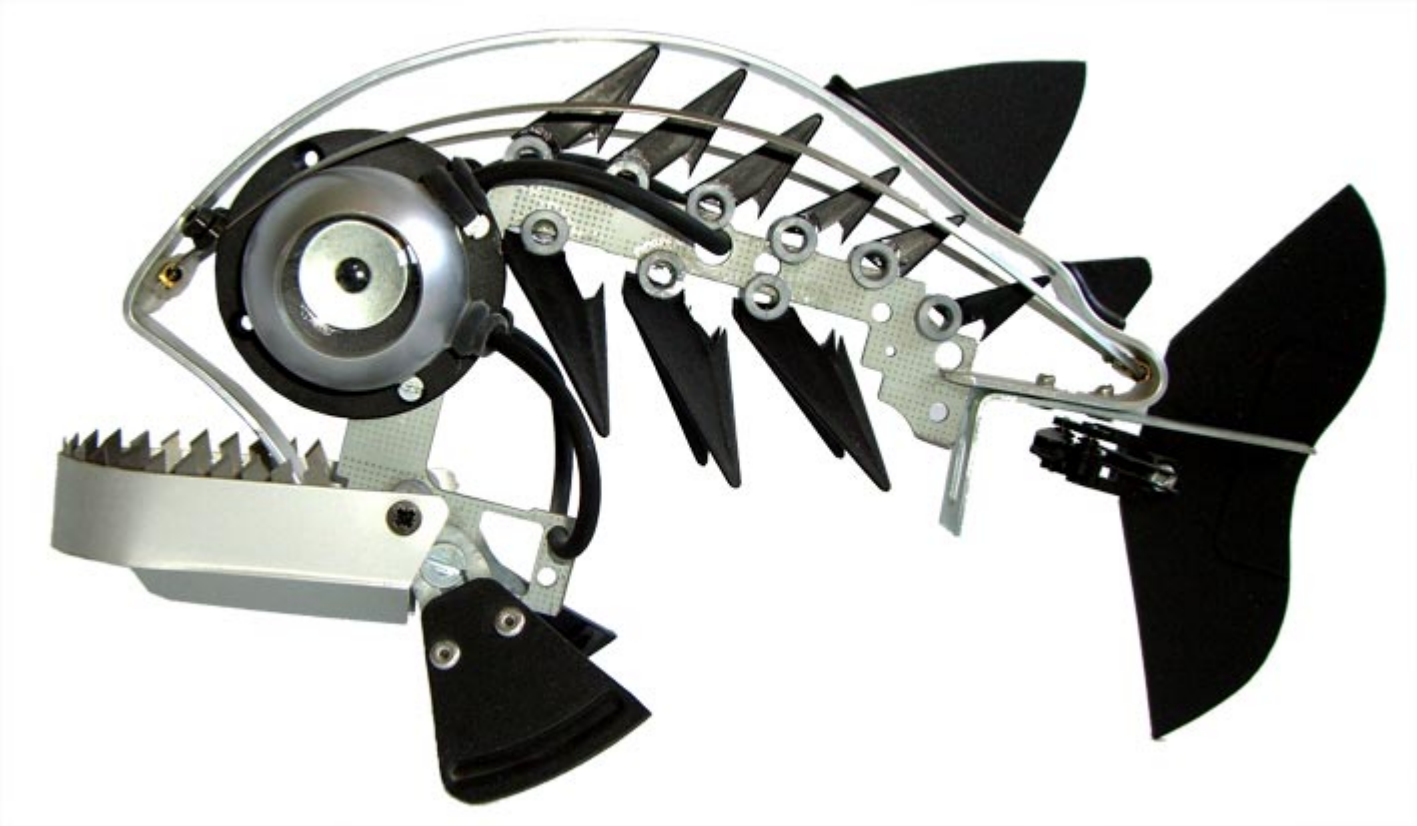
(gdb) set var = 10.0

simplemente cambia el valor.

(gdb) p var = 10.0

cambia e imprime en pantalla el nuevo valor.





GDB

creo que ahora vamos a verlo actuar...