Capítulo 20 — GDB: el microscopio de tu programa

*Capítulo avanzado, pero **escrito para que se entienda de verdad**. Se apoya en compilación y* `main` *(* `01` *), variables y tipos (* `02` *), operadores (* `03` *), condicionales y bucles (* `04` *), funciones (* `05` *), arreglos y cadenas (* `06` *), punteros (* `07` *),* `struct` *(* `08` *), memoria dinámica (* `09` *) y archivos (* `10` *).*

*Este capítulo no te enseña a escribir C: te enseña a **mirar** un programa de C mientras corre. Vas a aprender a frenar el programa en la línea exacta donde algo se rompe, a ver el valor de una variable **en el instante** en que se corrompe, a recorrer la pila de llamadas después de un* `Segmentation Fault` *y a leer la memoria cruda byte a byte. Y vas a entender por qué, la mayoría de las veces, **el bug no estaba donde estabas mirando**.*

*La herramienta del capítulo es **GDB** (GNU Debugger). No es la única — hay* `LLDB`*, Visual Studio Debugger,* `rr`*,* `lldb` *— pero es la que está en el 90 % de las máquinas Linux y en casi todos los cursos de C. Los conceptos que vas a ver acá (símbolos, direcciones, frames, registros) son **conceptos de depuración**, no de GDB: los necesitás igual con cualquier depurador.*

Cómo leer este capítulo

Este es el capítulo donde "mi programa a veces se rompe" se convierte en "sé **exactamente** por qué se rompe". No es teoría: cada idea viene con **comandos para ejecutar** y **salidas para leer**, y cada bloque cierra con **práctica** para que la uses vos.

Está dividido en **ocho partes**. Después de cada concepto importante hay uno o varios ejercicios cortos. **No los saltees**: son la mitad del capítulo.

Parte	De qué habla	La idea en una frase
1	El terreno	Un binario sin símbolos es un libro en clave: **DWARF** es la llave, y `-g3 -O0` te la da.
2	La TUI	La pantalla partida: código, ensamblador, registros y comandos, todo a la vista.
3	Control e inspección	Puntos de interrupción, control del flujo y mirar variables **en vivo**.
4	Punteros y control avanzado	Direcciones, desreferenciación, **watchpoints** y la pila de llamadas.
5	Errores de memoria	**Core dumps**, el comando `x` y los **sanitizers**: la caja negra del `Segfault`.
6	Bajo nivel	Ensamblador, registros, `stepi` y parcheo en caliente.
7	Automatización y procesos	`.gdbinit`, hilos, `fork`, `attach` y depuración remota.
8	Taller integrador	Un caso real con fuga de memoria, `Segfault` y bucle infinito.

Los **ejercicios** se numeran en orden: `20.1`, `20.2`, `20.3`, y así. Cada uno trae su **escenario de prueba** (qué hacés y qué tiene que pasar) para que lo ejecutes y compruebes.

> ***Regla de oro del libro:*** *el capítulo explica y muestra* ***cómo funciona*** *cada cosa (con comandos y salidas reales), pero* ***los ejercicios no traen la solución****. El enunciado te da el problema y el programa con la falla; el diagnóstico lo escribís vos.*

> ***Antes de empezar, dos advertencias honestas.***
>
> 1. ***GDB es una herramienta de Unix.*** *En Linux funciona tal cual. En macOS funciona, pero el depurador del sistema es* `LLDB` *y hay que firmar el binario de* `gdb` *para que el sistema lo deje ejecutar. En Windows,* `gdb` *existe vía* ***MSYS2 / MinGW-w64*** *y funciona bien para programas C, pero sin* `ulimit -c` *(o sea: sin core dumps automáticos) y con otras diferencias que vamos a señalar cuando aparezcan.*
> 2. ***Depurar es leer evidencia, no adivinar.*** *Un depurador no te dice dónde está el bug: te dice dónde* ***está el programa*** *y* ***qué cree que valen*** *sus variables. El diagnóstico sigue siendo tuyo. Por eso la última parte del capítulo es un taller con método, no una lista de comandos.*

PARTE 1 — El terreno: qué hay adentro de tu programa

Antes de tocar un solo comando de GDB, entendamos **qué es** un programa compilado y por qué `./prog` y `./prog -g` no son la misma cosa. Sin esto, GDB va a parecer magia y los errores que vas a ver ("no symbol table info available") van a parecer brujería.

1.1 Depurar es volver al lugar del accidente

Empecemos por el problema real. Tenés un programa de 2.000 líneas. Corre. A veces imprime una basura. A veces se cuelga. A veces revienta con `Segmentation fault`. ¿Dónde está el error?

Si tu único instrumento es **leer el código**, tenés una búsqueda en el vacío: 2.000 líneas, y el error puede estar en cualquiera. Leés todo dos veces, no encontrás nada, y empezás a sospechar del demonio.

Un depurador te da **tres superpoderes** que cambian el problema por completo:

Poder	En palabras	En GDB
Frenar en el momento exacto	Detener el programa justo cuando algo pasa, no "en algún lugar cerca"	Puntos de interrupción (`break`)
Ver el estado real	Mirar el valor **de verdad** de las variables en ese instante, no el que imaginás	`print`, `display`, `x`
Avanzar de a uno	Dar un solo paso, ver el efecto, y decidir el siguiente	`next`, `step`

> ***Pensalo como una cámara de seguridad a 1/1000 de velocidad.*** *Un auto choca contra un poste. No sirve para leer el código del auto: sirve para* ***ver el momento exacto*** *del impacto, con la cámara apuntando al lugar. GDB es esa cámara: no te dice por qué chodaste, te muestra el cuadro exacto donde pasó, con los datos del motor y de la velocidad.*

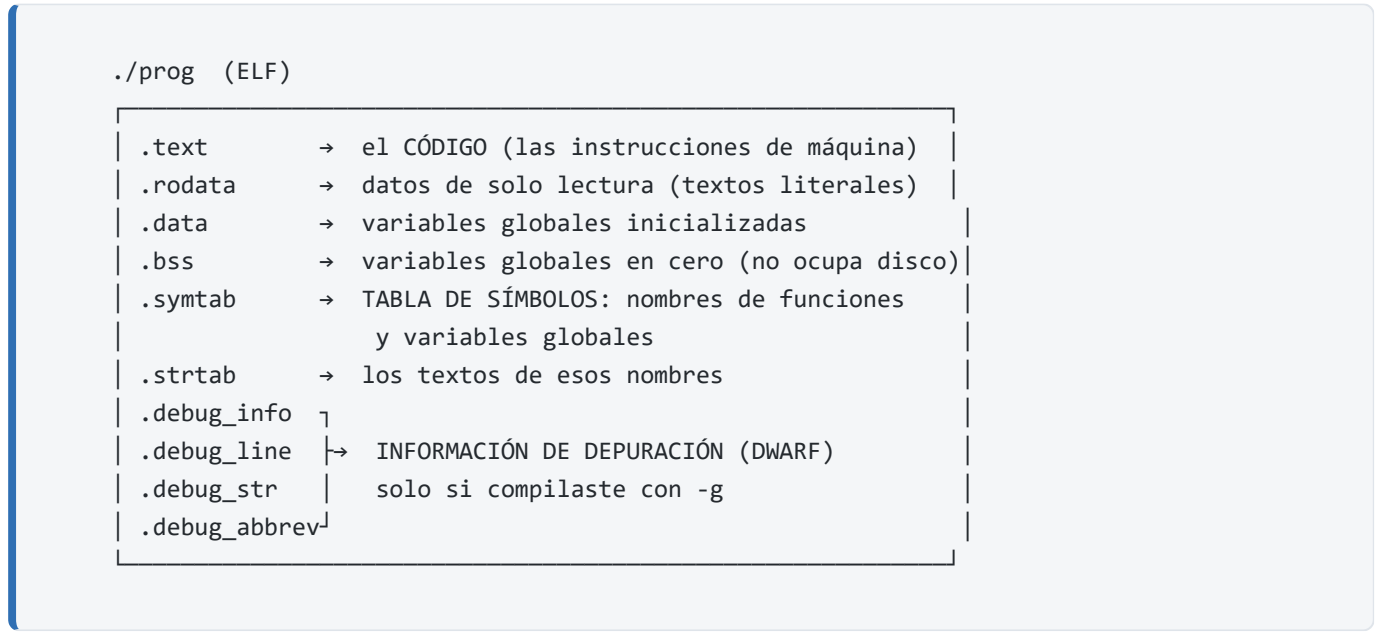
Y un detalle que importa mucho: **el depurador no te da la respuesta, te da la evidencia**. Un `print` te muestra lo que la variable **vale ahora** — no lo que *debería* valer. Toda la habilidad está en mirar esa evidencia y razonar: "esto vale `0`, pero acá debería valer `7` … ¿por qué?".

> ***La trampa más común del principiante.*** *Muchos creen que "GDB encuentra el bug por mí". No es así: GDB te dice* ***dónde estás****. El razonamiento sigue siendo 100 % tuyo. Conviene tenerlo claro desde el primer minuto.*

1.2 El binario ELF: el envase

Cuando compilás `prog.c`, el compilador no te da "un programa": te da un **archivo ELF** (Executable and Linkable Format), que es un **envase** con secciones adentro. Cada sección tiene un propósito distinto, y varias de ellas son exactamente lo que GDB necesita.

Pensá en un paquete de repuestos que llega con un **inventario** adentro. El programa es la máquina; el inventario es lo que le dice a GDB qué pieza es cada cosa.



Lo podés ver con `readelf`, que viene con `binutils`:

```
$ readelf -S ./demo
There are 31 section headers, starting at offset 0x...:

Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 1] .text             PROGBITS        0000000000001060 001060 00001f 00  AX  0   0 16
  [ 6] .rodata           PROGBITS        0000000000002000 002000 000010 00   A  0   0  8
  [10] .data             PROGBITS        0000000000004040 004040 000010 00  WA  0   0  8
  [14] .bss              NOBITS          0000000000004050 004050 000008 00  WA  0   0  8
  [16] .symtab           SYMTAB          0000000000000000 0023a8 000480 18     25  18  8
  [17] .strtab           STRTAB          0000000000000000 002828 0002c4 00      0   0  1
  [24] .debug_info       PROGBITS        0000000000000000 003000 0003a1 00      0   0  1
  [25] .debug_abbrev     PROGBITS        0000000000000000 0033a8 0000cc 00      0   0  1
  [26] .debug_line       PROGBITS        0000000000000000 003480 0002f3 00      0   0  1
  [27] .debug_str        PROGBITS        0000000000000000 003780 00016a 01  MS  0   0  1
```

> ***Fijate en la diferencia entre*** `.symtab` ***y*** `.debug_info`***.*** *La* ***tabla de símbolos*** *te da los nombres de las funciones y de las variables globales. La* ***información de depuración*** *te da mucho más: en qué* ***línea*** *está cada instrucción, qué* ***tipo*** *tiene cada variable, en qué* ***dirección de memoria*** *vive cada local, cómo se* ***desenrolla*** *la pila. Con solo* `.symtab` *podés poner un breakpoint en* `main`*; sin* `.debug_info` ***no podés*** *poner un breakpoint en la línea 42, ni mirar una variable local, ni avanzar paso a paso.*

1.3 DWARF: el manual de instrucciones del edificio

DWARF es el estándar de información de depuración que usan GCC, Clang y casi todos los depuradores de Unix. La sigla significa *Debugging With Attributed Record Formats*.

La idea es simple y vale la pena internalizarla:

> ***La CPU no sabe qué es una variable.*** *Para el procesador,* `total` *es la dirección* `0x7ffd...`*. DWARF es el* ***libro de anotaciones*** *que dice: "en la dirección* `0x1129` *hay código de la línea 42 del archivo* `main.c`*; en ese punto, la variable* `total` *está en el offset* `-0x18` *del marco de la pila y es de tipo* `int` *".*

Sin ese libro, el depurador solo tiene direcciones, y las direcciones no significan nada para nosotros.

La información no está en un solo lugar: DWARF la reparte en **secciones**, cada una con un trabajo:

Sección	Qué guarda	Para qué la usa GDB
`.debug_info`	Los **tipos**: funciones, variables, structs, enums	Saber qué es cada cosa y cómo imprimirla
`.debug_line`	La **tabla de líneas**: dirección ↔ archivo:línea	Traducir "estoy en `0x1129`" a "estoy en `main.c:42`"
`.debug_abbrev`	Las **abreviaturas** (cómo se codifica cada tipo de dato)	Comprimir la información sin repetirla
`.debug_str`	Los **textos**: nombres de variables, de archivos	Mostrarte `total` y no `v24`
`.debug_frame`	La **información de desenrollado** de la pila	Poder reconstruir la pila de llamadas

> ***Analogía del edificio.*** *Imaginá un edificio de oficinas de 20 pisos. El **ELF** es el edificio físico: hormigón, caños, cables. La **tabla de símbolos** es el cartel con los nombres de las oficinas ("Recepción", "Contabilidad"), colgado en la planta baja. Y la **información DWARF** es el **plano completo**: qué hay en cada oficina, dónde está el baño, cómo subir al piso 7, y en qué punto exacto de la escalera estás cuando mirás el Termómetro del piso 5. Sin plano, podés caminar pero estás a ciegas; con plano, ubicás cualquier cosa al instante.*

> ***Ojo con esto.*** *DWARF no es lo mismo que ELF. **ELF es el envase** y **DWARF es el formato de la información** que va adentro. En Windows con MSVC el envase es PE/COFF y la información de depuración es **PDB** (formato CodeView). En macOS el envase es Mach-O, pero DWARF también se usa adentro. O sea: **el concepto viaja, el nombre del envase cambia**.*

1.4 Las banderas que lo cambian todo: `-g`, `-g3`, `-O0`

Acá está la parte que más resultado te da, y la que más se ignora. Hay **dos decisiones independientes** que hay que tomar al compilar:

1. ¿Guardo **información de depuración**? → `-g`, `-g3`
2. ¿**Optimizo** el código? → `-O0`, `-Og`, `-O2`

La primera decisión: cuánta información de depuración

GCC tiene tres niveles (más uno que apaga todo). El default de `-g` es el nivel 2:

Bandera	Qué agrega	¿Lo vas a usar?
`-g0`	**Nada**. Apaga la información de depuración	Solo para binarios de producción stripped
`-g1`	Mínimo: funciones, variables globales y **tabla de líneas** (sin variables locales)	Rara vez
`-g` (= `-g2`)	Todo lo de `-g1` **más variables locales y `typedef`s**	**Tu default**
`-g3`	Todo lo de `-g2` **más las definiciones de macros**	**Cuando el bug está en una macro**

El nivel **3** es el que casi nunca se enseña y el que más rescata. Cuando el código sospechoso es `CHECK_NULL(p)` y esa macro viene de una cabecera de tu empresa, GDB con `-g3` te puede **mostrar la expansión de la macro** y decirte de qué archivo salió.

```
# Sin información de depuración
$ gcc -o demo demo.c

# Con lo justo (nivel 2, el default)
$ gcc -g -o demo demo.c

# Con macros también (nivel 3)
$ gcc -g3 -o demo demo.c
```

La segunda decisión: optimizar o no

Acá está la trampa clásica, y la reason por la que mucha gente cree que "GDB no funciona".

El optimizador (`-O1` , `-O2` , `-O3`) **reescribe tu programa**. Hace cosas que son correctas pero que rompen la correspondencia entre "una línea de C" y "un fragmento de máquina":

```
TU CÓDIGO                    LO QUE EL COMPILADOR HACE CON -O2
─────────                    ─────────────────────────────────

int a = 1, b = 2;            los dos valen 1 → el compilador
int c = a + b;               borra las variables que no usa

if (x > 0) { ... }           reordena, invierte, fusiona ramas
                             ("if (x > 0)" puede pasar a ser
                             un salto hacia atrás)

suma += valores[i];          mete el bucle en un vector (SIMD):
                             4 elementos por instrucción

llama_a(); llama_b();        METE la función adentro de la otra
                             (inlining) → desaparece del mapa
```

Y las consecuencias para GDB son concretas:

Con `-O0`	Con `-O2`
Cada línea de C ↔ un bloque de instrucciones	Varias líneas de C pueden caer en la misma instrucción
Las variables viven en la **pila** (memoria)	Las variables pueden vivir en un **registro**
`print x` muestra el valor	`print x` dice `<optimized out>`
`next` avanza una línea	`next` a veces saltea o retrocede

> ***El bug más famoso del principiante (versión depuración).*** *Compilás con* `-O2` *(porque "en producción va más rápido"), el programa falla, lo metés a GDB, y te encontrás con esto:*
>
> ```
> (gdb) print counter
> $1 = <optimized out>
> ```
>
> *No es un bug de GDB ni tuyo: el compilador* ***legalmente eliminó la variable*** *porque no hacía falta. No hay nada que mirar. Por eso la regla de abajo es* ***innegociable****.*

La regla de oro

> ***Depurá con*** `-O0 -g` ***. Medí con*** `-O2` ***. Nunca al revés.***
>
> *Son dos compilaciones* ***distintas*** *del mismo código, para dos públicos distintos. La que corrés para medir Performance* ***no*** *es la que depurás, y viceversa.*

Y si querés un punto medio moderno, GCC te da `-Og`:

```
$ gcc -Og -g -o demo demo.c
```

`-Og` activa **solamente** las optimizaciones que no estorban a la depuración (se la deja a `gcc` decidir). Es el flag que recomienda la propia documentación de GCC cuando querés "optimizado pero depurable".

La línea de compilación que vas a usar mil veces

```
$ gcc -std=c11 -g3 -O0 -Wall -Wextra -o demo demo.c
```

Desglosada:

Parte	Qué te da
`-std=c11`	El estándar de C que querés (capítulo `02`)
`-g3`	Información de depuración completa, **incluidas las macros**
`-O0`	Sin optimización: una línea de C = un bloque de instrucciones
`-Wall -Wextra`	Todos los avisos del compilador (son la mitad de los bugs de origen)
`-o demo`	El nombre del ejecutable

> ***Truco de laboratorio: dos compilaciones, dos binarios.*** *Guardá el comando de arriba en un archivo* `build.sh` *junto a tu proyecto. Te va a ahorrar mucho tiempo y te garantiza que siempre estás depurando con la misma configuración.*

1.5 Mirar el binario por fuera

Antes de entrar a GDB, conviene saber mirar el envase. Estos cuatro comandos son tus amigos:

```
$ file ./demo
demo: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, ...
```

`file` te dice **qué es** el archivo. Si dice `not stripped`, tiene tabla de símbolos.

```
$ nm ./demo
0000000000001129 T main
0000000000001139 t helper
0000000000004040 D global_counter
```

`nm` te da la **tabla de símbolos**: qué funciones y qué globales existen y en qué dirección. `T` = texto (código), `D` = datos, `t` = símbolo local, `U` = indefinido (viene de otro lado).

```
$ objdump -d ./demo
...
0000000000001129 <main>:
  1129: 48 89 e5             mov    %rsp,%rbp
  112c: 89 7d ec             mov    %edi,-0x14(%rbp)
  ...
```

`objdump -d` te da el **desensamblado**: el binario traducido a instrucciones.

```
$ size ./demo
   text    data     bss     dec     hex filename
    2192             16     8      2216      8a demo
```

`size` te dice cuánto ocupa cada parte.

> ***Un detalle que da vergüenza preguntar y que hay que saber.*** *Los binarios que se publican (los que descarga la gente de internet) suelen estar* ***stripped****: les sacaron la tabla de símbolos y la información de depuración para que ocupen menos. Es una decisión de* ***distribución****, no de compilación: el* `demo` *que vos depurás* ***tiene que*** *tener* `-g` *. La forma sana de hacerlo es* ***distribuir los símbolos aparte*** *(el paquete* `dbgsym` *en Linux, los archivos* `.dSYM` *en macOS), para que quien recibe el programa pueda depurar sin que el binario crezca.*

1.6 El checklist antes de empezar

Guardá esta lista. El 80 % de los "GDB no me funciona" son porque una de estas siete cosas no se cumplió.

```
[ ] 1. Tengo gcc/clang instalado y puedo compilar
[ ] 2. Compilo con -g3 -O0
[ ] 3. `file ./demo` dice "ELF ... not stripped" (o "with debug_info")
[ ] 4. `gdb --version` responde
[ ] 5. `gdb ./demo` abre y muestra el prompt (gdb)
[ ] 6. `break main` no se queja
[ ] 7. `run` para en `main`
```

Vamos a verificarlo juntos, en la primera práctica.

Práctica de la Parte 1

Ejercicio 20.1 — El binario desnudo y el binario con mapa

Conceptos · ELF, secciones, tabla de símbolos, DWARF, `-g`, `-g3`, `file`, `readelf -S`, `nm`

Contexto · Un mismo programa se puede compilar de dos maneras: como se publica (liviano, sin información de depuración) y como se depura (con toda la información). Los dos ejecutables hacen **exactamente lo mismo**. Lo que cambia es qué puede saber el depurador sobre ellos.

Consigna · Escribí un programa `demo.c` con una función global, una variable global y una variable local, e **imprimí** (con `printf`) el valor de la variable global para que el programa haga algo observable. Compilalo **dos veces**:

1. `gcc -o demo_stripped demo.c`
2. `gcc -g3 -O0 -o demo_debug demo.c`

Después compará **los dos binarios** con estas herramientas y contestá:

- `file` — ¿qué diferencia reporta?
- `ls -l` — ¿cuánto más pesa el que tiene información de depuración? ¿Por qué?
- `readelf -S` — ¿qué secciones aparecen en uno y no en el otro?
- `nm` — ¿cambia algo entre los dos? ¿Por qué?

Entrada · Ninguna: es una comparación de dos binarios. **Salida** · Un informe corto con las cuatro comparaciones y tu explicación de cada diferencia.

Escenario de prueba

```
$ ls -l demo_stripped demo_debug
-rwxr-xr-x 1 user user  16096 demo_stripped
-rwxr-xr-x 1 user user  22192 demo_debug

$ file demo_stripped
demo_stripped: ELF 64-bit LSB pie executable, x86-64, ..., stripped

$ file demo_debug
demo_debug: ELF 64-bit LSB pie executable, x86-64, ..., with debug_info, not stripped
```

Identificadores sugeridos · `demo.c` , `demo_stripped` , `demo_debug` , `global_counter` **Restricciones/pistas**

- El ejecutable sin `-g` **no** tiene por qué fallar: sigue funcionando perfectamente. Lo que pierde es **la capacidad de ser depurado en detalle**.
- `readelf -S` muestra muchísimas secciones: a vos te interesan `.symtab` , `.strtab` y las que empiezan con `.debug_` .
- `nm` **no** cambia entre los dos: la tabla de símbolos la genera el compilador siempre. Lo que cambia es la información de depuración, que va en otras secciones.

Criterios de aceptación

- ☐ Los dos binarios se compilan con los comandos exactos del enunciado.
- ☐ `file` reporta la diferencia de *stripped* / *with debug_info*.
- ☐ La diferencia de tamaño está medida y explicada.
- ☐ Se identificó qué secciones `.debug_*` aparecen solo en el binario con `-g3` .
- ☐ Se respondió por qué `nm` da lo mismo en los dos casos.

Ejercicio 20.2 — "No symbol table info available"

Conceptos · información de depuración, tabla de líneas, `break` , `info line` , diagnóstico

Contexto · Este es el error más común del primer día: querés poner un punto de interrupción en una línea y GDB te responde que no sabe de qué archivo hablás. No es un fallo de GDB: es la consecuencia directa de lo que aprendiste en 1.2.

Consigna · Reproducí **exactamente** ese error y diagnosticálo con las herramientas:

1. Compilá un programa **sin** `-g`.
2. Abrilo con GDB y pedí un punto de interrupción en una línea de código (por ejemplo, la línea 10 de tu archivo).
3. Leé el mensaje de error de GDB **entero** y anotá qué te dice.
4. Usá `info line <archivo>:<línea>` y `info functions` para juntar evidencia.
5. Recompilá **con** `-g3 -O0` y repetí el paso 2: ahora debería funcionar.

Entrada · Ninguna. **Salida** · Un diagnóstico escrito: **síntoma** → **evidencia** → **causa** → **solución**.

Escenario de prueba

```
$ gcc -o demo demo.c
$ gdb ./demo
(gdb) break demo.c:10
No source file named demo.c.
Make breakpoints pending? (y or n)
```

```
$ gcc -g3 -O0 -o demo demo.c
$ gdb ./demo
(gdb) break demo.c:10
Breakpoint 1 at 0x1129: file demo.c, line 10.
```

Identificadores sugeridos · `demo.c`, `demo` Restricciones/pistas

- Fijate en la diferencia entre el **archivo fuente** (`.c`, que es el que vos escribís) y el **archivo objeto** (`.o`): GDB no abre `.c`, abre el ejecutable y busca en él la información de depuración que le dice qué hay en `demo.c`.
- `info functions` responde qué funciones **conoce** GDB. Si la lista está vacía o es mínima, no hay información de depuración.

Criterios de aceptación

- [ ] El error se reprodujo de verdad (no de memoria).
- [ ] El mensaje de error de GDB está citado textualmente en el diagnóstico.
- [ ] Se usó `info line` y `info functions` como evidencia.
- [ ] La causa está explicada **en términos de la Parte 1** (secciones `.debug_*`), no como "hay que poner `-g`".
- [ ] La solución se verificó recompilando y repitiendo el paso 2.

PARTE 2 — La interfaz TUI: la pantalla partida

Ya tenés el terreno. Ahora sí: sentate frente a GDB.

La primera vez que arrancás GDB, la tentación es usarlo como si fuera una terminal más: escribís comandos, leés texto. Funciona. Pero es como usar una calculadora científica para sumar dos números: **GDB tiene una pantalla dividida en paneles que la mayoría nunca descubre**.

2.1 Arrancar GDB

Tres formas de entrar, y una cuarta que es la más útil de todas:

```
$ gdb ./demo                  # el programa está en ./demo
$ gdb demo                    # igual (GDB busca el archivo)
$ gdb --args ./demo a b c     # el programa y SUS argumentos
$ gdb ./demo core.core        # un core dump (Parte 5)
```

`--args` es la clave: GDB **no** ejecuta el programa al arrancar, así que los argumentos que le pasás son exactamente los que va a usar el programa cuando le digas `run`. Después los podés cambiar sin recompilar:

```
(gdb) set args 10 20 30
(gdb) show args
Argument list to give program when it is started is "10 20 30".
```

Al arrancar, GDB se presenta y te da el prompt. Ese prompt es la única cosa que tenés que mirar:

```
GNU gdb (GDB) 14.2
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
...
(gdb)
```

El prompt `(gdb)` es una **REPL**: un bucle que lee un comando, lo ejecuta, imprime el resultado y vuelve a leer. Igual que el `>>>` de Python. Los comandos no llevan punto y coma.

Y como en cualquier REPL, hay ayuda:

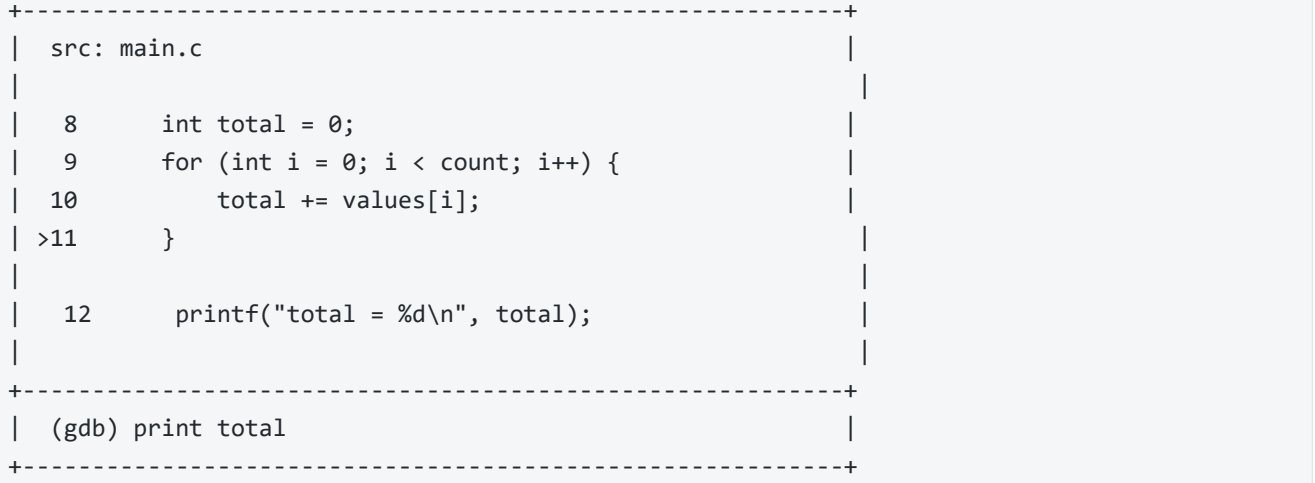
```
(gdb) help               # lista de categorías
(gdb) help breakpoint    # ayuda de un comando concreto
(gdb) apropos watch      # busca comandos que contengan "watch"
(gdb) break ma<TAB>      # el autocompletado completa a "break main"
```

> ***Momento ajá.*** *`apropos` y el autocompletado con `TAB` son los dos atajos que hacen que GDB pase de "mando a ciegas" a "herramienta de trabajo". `apropos` es tu buscador: si sabés lo que querés pero no cómo se llama el comando, `apropos` te lo encuentra.*

2.2 La ventana de comandos

En modo TUI, la pantalla no es un scrolls de texto: es un **conjunto de ventanas**. La de abajo se llama `cmd` y es la **ventana de comandos**: escribís ahí, y arriba se va redibujando el estado del programa.

El layout por defecto es simple: la ventana de **código fuente** arriba y la de **comandos** abajo.



La línea que tiene el `>` adelante es la **línea actual**: dónde está el programa ahora mismo. En GDB, ese `>` se llama el **arrow marker** y es tu lugar en el mapa.

> ***Tecla rápida.*** *La flecha* `>` *no es decorativa: si pulsás* `Ctrl+G` *(o hacés clic), GDB te muestra* ***dónde*** *está el puntero de instrucción actual respecto al archivo.*

2.3 Las ventanas que tenés disponibles

GDB trae un juego de ventanas prefabricadas. Cada una muestra un aspecto distinto del programa:

Ventana	Qué muestra	Para qué la querés
`src`	El **código fuente** con el `>` en la línea actual	Orientarte: ¿dónde estoy?
`asm`	El **ensamblador** de la función actual	Ver qué hace la máquina (Parte 6)
`regs`	Los **registros** del procesador	Ver `rax`, `rsp`, `rip` (Parte 6)
`stack`	La **pila de llamadas** (frames)	Ver quién llamó a quién (Parte 4)
`threads`	La lista de **hilos**	Depurar programas concurrentes (Parte 7)
`breakpoints`	Los **puntos de interrupción**	Ver y administrar tus paradas
`cmd`	La **ventana de comandos**	Escribir comandos
`status`	El **estado**: proceso, señales, frames	Un resumen de todo
`history`	El **historial** de comandos	Volver a un comando anterior
`expr`	La **expresión** que estás por evaluar	Historial de expresiones
`block`	Los **bloques** de código de la función actual	Ver la estructura del código
`global`	Las **variables globales**	Ver el estado global

> ***Para guardar.*** *`stack`, `threads` y `breakpoints` son las que más rinden en el día a día. `regs` y `asm` las vas a necesitar en la Parte 6. `cmd` está siempre: es la única que no desactives nunca.*

2.4 Cambiar la disposición

Acá es donde GDB deja de ser una terminal y se vuelve una **console de depuración**.

Poner una ventana

El comando `layout` mete una ventana en la disposición actual:

```
(gdb) layout src          # ventana de código fuente
(gdb) layout stack        # ventana de pila de llamadas
(gdb) layout regs          # ventana de registros
(gdb) layout cmd          # ventana de comandos
```

Ver la disposición actual

```
(gdb) show layout
Selected layout:
1) src
2) stack
3) regs
4) cmd
```

Recorrer los layouts prefabricados

GDB trae varios layouts ya armados (el clásico de "código arriba, comandos abajo", el de "pila a la derecha", etc.). Para rotar entre ellos:

```
(gdb) tui next-layout
```

Y para elegir uno por nombre:

```
(gdb) tui select-layout first
(gdb) tui select-layout default
(gdb) tui select-layout legacy
```

> ***El atajo que casi nadie conoce:*** `tui new-layout` *. Armá las ventanas como te gusten (* `layout src` *,* `layout asm` *,* `layout regs` *,* `layout cmd` *), y después:*
>
> ```
> (gdb) tui new-layout depurar
> ```
>
> *GDB* ***guarda esa disposición con nombre****. Después la recuperás cuando quieras con* `tui select-layout depurar` *. Es un* `.gdbinit` *hecho a medida, y en la Parte 7 lo vas a ver automatizado.*

Mover el foco

El teclado actúa sobre la ventana que tiene el **foco**. Para cambiarlo:

```
(gdb) tui focus src        # el foco pasa al código
(gdb) tui focus next       # al siguiente
(gdb) tui focus prev       # al anterior
```

Un layout de trabajo típico

Uno de los que vas a usar mucho cuando estés con problemas de memoria:

```
+---------------------------+-------------------------+
|  src: main.c              |  stack                  |
|                           |   #0  main () at main.c |
|  42     free(p);          |   #1  run () at run.c   |
| >43     p = NULL;         |   #2  ...               |
|                           |                         |
+---------------------------+-------------------------+
|  regs                     |  cmd                    |
|  rax 0x0  rbx 0x...       |  (gdb)                  |
+---------------------------+-------------------------+
```

2.5 Redibujar la pantalla

A veces la TUI se **desordena**: pasaste un `less` interno, la terminal cambió de tamaño, o pegaste algo raro. GDB tiene comandos para arreglarla:

Qué	Cómo
Redibujar todo	`refresh`
Redibujar con el teclado	`Ctrl+L`
Ajustar el alto/ancho que GDB cree	`set height 40`, `set width 120`
Cambiar el estilo de los bordes	`set tui border-mode ascii` (para terminales viejitas)
Ver la configuración	`show tui`

> ***El borde `ascii` es tu amigo en Windows.*** *En la terminal de Windows (y en algunos emuladores de terminal viejos), los bordes con caracteres de línea de Unix se ven como caracteres raros.* `set tui border-mode ascii` *los reemplaza por* `+`, `-`, `|`. *Se ve menos elegante pero se lee perfecto.*

> ***La trampa N.º 1 de la TUI: perdiste el foco.*** *Si escribís* `print total` *y en pantalla no pasa nada, casi seguro estás en la ventana equivocada.* `tui focus cmd` *y listo. Y si la pantalla está* ***del revés*** *o medio rota:* `tui disable` *(vuelve al modo texto clásico),* `tui enable` *(vuelve a la TUI) y listo.*

2.6 Alternar entre TUI y línea de comandos

A veces la TUI molesta: cuando querés pegar un comando largo, o cuando la salida es enorme. GDB vive en los dos mundos y los dos mundos se alternan:

```
(gdb) tui disable     # al modo clásico: solo texto
(gdb) tui enable      # de vuelta a la pantalla partida
```

En el modo clásico, todo lo que aprendiste funciona igual. La TUI es una **capa visual**, no un modo distinto de depurar.

Y hay dos comandos que vale la pena conocer desde el primer día:

```
$ shell ls -la            # ejecutás un comando de shell SIN salir de GDB
```

```
(gdb) set logging file session.txt
(gdb) set logging on
(gdb) run
(gdb) set logging off
```

> ***Truco de laboratorio: grabá tus sesiones.*** `set logging on` *escribe* ***todo*** *lo que pasa (la salida del programa, los comandos, los resultados) en un archivo de texto. Cuando entregues el informe de una práctica, ese archivo es* ***tu evidencia****. Es el equivalente en texto de la caja negra del avión.*

2.7 Mapa de atajos de la TUI

Atajo	Qué hace
`Ctrl+L`	Redibuja la pantalla
`Ctrl+C`	Interrumpe el programa que está corriendo
`Ctrl+D`	Salir de GDB (en la ventana de comandos)
`↑` / `↓`	Historial de comandos (con el foco en `cmd`)
`TAB`	Autocompletado de comandos y de nombres de variables
`Ctrl+G`	Muestra el estado del programa en la ventana de estado
`PageUp` / `PageDown`	Desplazar la ventana de código

Práctica de la Parte 2

Ejercicio 20.3 — Tu consola de depuración

Conceptos · TUI, `layout`, `tui next-layout`, `tui new-layout`, `tui focus`, `refresh`

Contexto · La TUI se configura. Cada persona trabaja distinto: hay quienes quieren la pila de llamadas siempre visible, y quienes prefieren ver los registros. GDB te deja armar **tu** consola.

Consigna · Con un programa de prueba en marcha:

1. Poné la ventana de **código fuente** (`layout src`) y la de **comandos** (`layout cmd`).
2. Recorré los layouts prefabricados con `tui next-layout` y anotá cuáles existen en tu versión de GDB (el comando `help tui` te los lista).
3. Armá una disposición con **código fuente** y **pila de llamadas** (`layout stack`) que te sea cómoda, y guardala con nombre: `tui new-layout <nombre>`.
4. Recuperala con `tui select-layout <nombre>`.
5. Practicá el foco: `tui focus src`, `tui focus cmd`, `tui focus next`.
6. Provocá un desborde: hacé `tui disable` y `tui enable` y comprobá que la pantalla vuelve a estar sana.

Entrada · Ninguna. **Salida** · Un layout guardado con nombre, y una nota con los atajos que te quedaron cómodo.

Escenario de prueba

```
(gdb) layout src
(gdb) layout stack
(gdb) layout cmd
(gdb) tui new-layout depurar
Saved new TUI layout "depurar"
(gdb) tui select-layout depurar
```

Criterios de aceptación

- [ ] Existe un layout con **nombre** guardado y recuperable.
- [ ] El stack (`layout stack`) y el source (`layout src`) aparecen **juntos** en el layout final.
- [ ] Se comprobó que el foco se mueve entre ventanas con `tui focus` .
- [ ] `tui disable` / `tui enable` deja la pantalla sana.
- [ ] El layout se documentó con un nombre que signifique algo para vos.

Ejercicio 20.4 — El tour de la TUI

Conceptos · ventanas de la TUI, `layout` , `show layout` , foco, lectura del estado

Contexto · Conocer las ventanas disponibles es como conocer los instruments de un tablero: no vas a usar todos todos los días, pero cuando uno falla necesitás reconocerlo al instante.

Consigna · Hacé un recorrido **documentado** de cada ventana de la TUI. Para cada una:

1. Ponela en pantalla con `layout <ventana>` .
2. Poné el foco en ella con `tui focus <ventana>` .
3. **Escribí** qué información muestra y **para qué te sirve**.
4. Buscá un momento del programa donde esa ventana tenga contenido **interesante** (por ejemplo: la ventana `stack` con varias funciones en la pila, o la ventana `regs` con valores distintos de cero).

Completá esta tabla en tu cuaderno o en un archivo de texto:

```
ventana     | ¿qué muestra?                          | ¿cuándo me sirve?
------------+----------------------------------------+-----------------------------
src         |                                        |
asm         |                                        |
regs        |                                        |
stack       |                                        |
threads     |                                        |
breakpoints |                                        |
status      |                                        |
cmd         |                                        |
```

Entrada · Ninguna. **Salida** · La tabla completa.

Escenario de prueba

```
(gdb) layout regs
(gdb) tui focus regs
(gdb) info registers
rax            0x1                 0
rbx            0x0                 0
rip            0x401136            0x401136 <main+13>
rsp            0x7fffffffe320      0x7fffffffe320
```

Identificadores sugeridos · `tour.md`, una tabla por ventana **Restricciones/pistas**

- Las ventanas que GDB muestra **vacías** (porque el programa no tiene hilos, o no hay breakpoints) también son información: te dicen que eso todavía no pasó.
- `help tui` lista todo lo que la TUI sabe hacer en tu versión.
- No hace falta que el programa sea complejo: uno con tres funciones anidadas alcanza para que `stack` se vea interesante.

Criterios de aceptación

- [ ] Las 8 ventanas están documentadas con qué muestran y cuándo sirven.
- [ ] Cada ventana se **focused** (no solo puesta: el foco importa).
- [ ] Hay al menos **una** ventana capturada con contenido no trivial.
- [ ] La tabla distingue las ventanas de "estado" de las ventanas de "contenido".

PARTE 3 — Control de ejecución e inspección básica

Tenés el programa cargado, la pantalla dividida y el prompt esperando. Ahora viene lo central: **frenar el programa en el lugar justo** y **mirar lo que hay ahí**.

3.1 Ejecutar el programa

Hay tres formas de arrancar, y cada una frena en un lugar distinto:

Comando	Qué hace	Dónde frena
`run`	Ejecuta el programa **completo**	En el próximo punto de interrupción
`start`	Ejecuta hasta el **comienzo de** `main`	En la primera instrucción de `main`
`starti`	Ejecuta hasta la **primera instrucción** del programa	En `_start`, antes de `main`

```
(gdb) start
Temporary breakpoint 1 at 0x1129: file main.c, line 8.
[Thread debugging using libthread_db enabled]

Temporary breakpoint 1 at 0x1129: file main.c, line 8.
Continuing.
Breakpoint 1, main () at main.c:8
8       int total = 0;
(gdb)
```

Leé esa salida con cuidado, porque tiene **mucha información**:

- `Temporary breakpoint 1` → GDB puso un breakpoint **temporal** en `main` y lo va a borrar solo cuando se dispare.
- `Temporary breakpoint 1 at 0x1129: file main.c, line 8.` → **la dirección** del código, **el archivo** y **la línea**. Acá GDB usó la información DWARF para traducir.
- `Breakpoint 1, main () at main.c:8` → estamos parados. El nombre de la función y la línea.
- `8 int total = 0;` → el código fuente de esa línea, mostrado automáticamente. Esta es la forma más cómoda de orientarse: **GDB te muestra el código donde estás**.

> ***La diferencia entre `run` y `start`, en una frase.*** *`run` deja el programa correr hasta que pase algo interesante (un breakpoint, un crash). `start` te deja **dentro** de `main`, listo para que examines las variables locales desde el primer momento. Si vas a depurar una función concreta, `start` + `break <función>` + `continue` es el combo padrão.*

3.2 Puntos de interrupción: frenar el programa

El **breakpoint** es el corazón de la depuración. Es una marca que dice "**stop acá**". Cuando el programa llega a esa marca, GDB frena el proceso **justo antes** de ejecutar esa instrucción y te devuelve el control.

Hay cuatro formas de poner uno:

```
(gdb) break main                # por nombre de función
(gdb) break demo.c:42           # por archivo y línea
(gdb) break 42                  # por línea del archivo actual
(gdb) break *0x401136           # por dirección de memoria
```

Cuál usar. El 90 % de las veces, `break archivo.c:línea`. Es la más precisa y la que mejor funciona cuando el archivo tiene funciones repetidas o nombres parecidos.

Y hay dos variantes que resuelve problemas específicos:

```
(gdb) tbreak demo.c:42          # TEMPORAL: se borra solo después de dispararse
```

`tbreak` (de *temporary*) es perfecto para "pasame por acá una sola vez". Si ponés un `break` normal y el código pasa mil veces por esa línea, vas a tener que seguir pulsando `continue` mil veces. Con `tbreak`, seguís una vez y seguís libre.

```
(gdb) dprintf demo.c:42, "i = %d\n", i     # al pasar por acá, imprime y sigue
```

`dprintf` es un breakpoint que **imprime y sigue**. No te frena: te deja una nota en la salida. Ideal para instrumentar un bucle sin tocar el código.

> ***Ojo con esto.*** *El* `break` *se pone* ***en una línea ejecutable****. Si ponés* `break demo.c:7` *y la línea 7 es un comentario, un* `}` *o la declaración de una función, GDB te va a decir que no puede poner un breakpoint ahí y te va a ofrecer moverlo a la línea siguiente. Es normal: no es un error tuyo.*

3.3 Administrar tus puntos de interrupción

Cuando tenés varios, necesitás verlos y administrarlos.

Verlos

```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000001129 in main() at demo.c:8
2       breakpoint     keep y   0x0000000000001160 in add_item() at demo.c:20
3       breakpoint     keep y   0x0000000000001175 in main() at demo.c:42
        breakpoint already hit 1 time
```

Cada punto tiene un **número**. Ese número es su nombre, y lo vas a usar para todo lo demás. `keep` = queda puesto; `y` / `n` = habilitado o deshabilitado; `Address` = la dirección de memoria; `What` = la función y el archivo:línea.

Deshabilitar, habilitar, borrar

```
(gdb) disable 2          # desactiva el breakpoint 2 (no lo borra)
(gdb) enable 2           # lo vuelve a activar
(gdb) delete 2           # lo borra
(gdb) disable            # sin número: desactiva el ÚLTIMO creado
(gdb) delete             # sin número: borra TODOS (pregunta confirmación)
```

> ***Truco de laboratorio: deshabilitar es mejor que borrar.*** *Si un breakpoint te molesta mientras investigás otra cosa,* `disable 2` *lo apaga sin perderlo.* `delete` *se lleva el numero puesto: y los números se reutilizan, lo cual confunde.*

Condicionar un breakpoint

Un breakpoint condicional **solo se dispara si la condición es verdadera**. Es el comando más poderoso del capítulo, y lo vemos en detalle en la Parte 4.

```
(gdb) break demo.c:42 if i == 7
```

Automatizar un breakpoint: `commands`

Acá GDB deja de ser interactivo. Con `commands` le decís "cuando se dispare este breakpoint, hacé esto":

```
(gdb) break add_item
(gdb) commands
> printf "add_item: count = %d, cap = %d\n", g_count, g_capacity
> continue
> end
(gdb) run
```

Cada línea entre `commands` y `end` es un comando que GDB ejecuta **automáticamente** cada vez que se dispara el breakpoint. El `continue` final es el que deja el programa seguir.

> ***El poder real de `commands`.*** *Con esto podés **instrumentar** un programa sin tocar su código: contadores, trazas, validaciones. Es la base de la automatización de la Parte 7 y del taller final.*

3.4 Controlar el flujo: las cinco palancas

Acá está el corazón de la depuración interactiva. Estos cinco comandos te dejan avanzar **de a una instrucción**, y son los que vas a usar el 90 % del tiempo.

Comando	Atajo	Qué hace	Entra a las funciones
`next`	`n`	Avanza a la **línea siguiente**	**No**
`step`	`s`	Avanza a la **línea siguiente**	**Sí**
`finish`	`fin`	Corre hasta que la función **actual termine**	—
`continue`	`c`	Corre hasta el **próximo breakpoint**	—
`until`	`u`	Corre hasta una línea **más abajo** de la actual	No

`next` contra `step`: la diferencia que lo cambia todo

Esta es **la** pregunta que todos se hacen el primer día. Y la respuesta es una analogía de calle:

> ***Imaginá que estás caminando por una calle y mirás una panadería.***
>
> - `next`: *seguís de largo por la vereda. Viste la panadería, seguiste, no entraste.*
> - `step`: *cruzás la calle y entrás a mirar qué hacen adentro.*

En el código:

```c
void procesar(int *datos, int n) {
    int total = 0;
    for (int i = 0; i < n; i++) {
        total += sumar(datos[i]);      /* línea 4 */
    }
    printf("total = %d\n", total);     /* línea 6 */
}
```

- Con `next` en la línea 4, el próximo `next` te deja en la **línea 6**: `sumar()` se ejecutó entera, pero no te preocupaste por lo que pasó adentro.
- Con `step` en la línea 4, el próximo `step` te mete **dentro de** `sumar()`.

> ***Regla práctica.*** *Si estás* ***buscando un valor*** *(`print total`), usá `next`: no te interesa lo que pasa adentro de las funciones, te interesa llegar a la línea siguiente. Si estás* ***rastreando el camino de los datos*** *("¿quién me cambió esta variable?"), usá `step`: querés ver cada llamada.*

`finish` : salir de la función sin apretar `step` 300 veces

`finish` corre **hasta que la función actual retorne** y te deja en la línea de llamada. Es el "ya está, volvamos arriba" de la depuración.

```
(gdb) step
_sumar () at math.c:5
5       return x * 2;
(gdb) finish
Run till exit from _sumar () at math.c:5
total = 14
9       printf("total = %d\n", total);
```

La línea `total = 14` es la **salida del programa**: `finish` deja que el programa siga corriendo normalmente hasta que vuelve a vos.

`until` : saltar hasta una línea concreta

`until` es como `next` , pero "hasta acá y no más allá". Se usa para saltear un bloque largo:

```
(gdb) until 20
```

Si estás en la línea 5 y querés llegar a la 20 **sin pasar por la 18**, `until` lo hace. Y si en el camino se dispara un breakpoint, se para ahí (igual que `continue`).

`return` : salir ahora (con Warning)

```
(gdb) return 42
Make selected stack frame return now? (y or n) y
```

`return` **fuerza** el retorno de la función actual con el valor que le des. Es útil para probar hipótesis ("¿y si esta función devolviera 42, el bug seguiría?"), pero **es una mentira**: el programa no vuelve de la función de verdad, la función se saltea. Usalo con cuidado y solo para experimentar.

3.5 Inspeccionar: `print` y compañía

`print` (o `p`) es el comando que más vas a usar. Imprime el valor de una **expresión** en el momento actual.

```
(gdb) print total
$1 = 0
(gdb) print total + 1
$2 = 1
(gdb) print i * n
$3 = 12
```

Detalle importante: **podés imprimir cualquier expresión**, no solo variables. Y GDB te devuelve un **número de referencia** (`$1`, `$2`, ...). Ese número te deja referenciar el valor después:

```
(gdb) print total
$1 = 0
(gdb) print $1 + 100
$2 = 100
```

Formatear la salida

El comando `print` acepta sufijos que cambian el formato:

Sufijo	Formato	Ejemplo
`/x`	Hexadecimal	`p/x total` → `$1 = 0x0`
`/d`	Decimal	`p/d total`
`/u`	Decimal sin signo	`p/u valor`
`/c`	Carácter	`p/c letra` → `$1 = 65 'A'`
`/s`	Cadena	`p/s nombre`
`/f`	Punto flotante en varios formatos	`p/f valor`

```
(gdb) print/x total
$1 = 0x0
```

```
(gdb) print/c letra
$2 = 65 'A'
```

Los otros comandos de inspección

```
(gdb) ptype total                    # el TIPO de la variable
$1 = int

(gdb) info locals                    # todas las locales del frame actual
total = 0
i = 0

(gdb) info args                      # los argumentos de la función actual
n = 5
datos = 0x7fffffffe320

(gdb) info variables                 # TODAS las variables del programa (¡muchas!)
(gdb) info functions                 # todas las funciones conocidas
(gdb) info scope procesar            # variables y miembros visibles en esa función
(gdb) info line demo.c:42            # qué código hay en esa línea
(gdb) info address total             # la dirección de una variable
```

`ptype` es especialmente útil: te dice **cómo está declarada** la variable, incluso si no la podés ver. Con `ptype struct Item` te imprime la definición completa de la struct.

Y hay dos comandos para imprimir con formato (más parecidos a `printf`):

```
(gdb) printf "total=%d i=%d\n", total, i
total=0 i=0
(gdb) output total
0
```

`output` imprime **sin** registrar el valor en el historial (útil en scripts).

Ajustar cómo se imprime

Estos son los `set` que más se tocan:

```
(gdb) set print pretty on       # las structs se muestran en varias líneas
(gdb) set print elements 0      # Arrays: SIN límite (por defecto, 200 elementos)
(gdb) set print repeats 0       # no abreviar arrays con "[...]"
(gdb) set print address on      # mostrar la dirección de los punteros
```

`set print elements 0` es **imprescindible** cuando trabajás con arreglos grandes: sin eso, `print arr` te muestra solo los primeros 200 elementos y un `...` que te puede hacer creer que el arreglo es más chico.

3.6 `display` y `watch`: mirar sin parar, y parar cuando cambia

Estos dos comandos parecen parecidos y no lo son. La diferencia está en **cuándo** actúan:

Comando	Qué hace	¿Frena el programa?
`print x`	Imprime `x` **una vez**, ahora	No
`display x`	Imprime `x` **cada vez** que el programa se frena	No (solo muestra)
`watch x`	**Frena** el programa cuando `x` cambia	**Sí**

`display`: el monitor de variables

```
(gdb) display total
1: total = 0
(gdb) display i
2: i = 0
(gdb) next
total = 0
i = 0
3               total += datos[i];
```

Cada vez que avanzás, GDB vuelve a imprimir los valores. Es **exactamente** lo que querés cuando estás siguiendo una variable a lo largo de un bucle.

Para administrarlos:

```
(gdb) display              # lista los displays actuales
1: total = 0
2: i = 0
(gdb) undisplay 1          # saca el display 1
(gdb) undisplay            # saca todos
(gdb) delete display 1     # alternativa
```

> ***Truco de laboratorio.*** *`display` + `next` en un bucle es el patrón clásico para **seguir una variable**. No es lo más elegante del mundo (la salida se llena de ruido), pero funciona y es lo primero que tenés que tener a mano.*

`watch` : el centinela

`watch` frena el programa **en el instante en que la expresión cambia de valor**. Es la herramienta más precisa del capítulo para encontrar **dónde** se corrompe un dato, y la vemos en detalle en la Parte 4.

```
(gdb) watch counter
Hardware watchpoint 2: counter
(gdb) continue
...
Hardware watchpoint 2: counter
Old value = 3
New value = 4
```

Práctica de la Parte 3

Ejercicio 20.5 — Organizar puntos de interrupción

Conceptos · `break` , `tbreak` , `delete` , `disable` , `enable` , `info breakpoints` , `commands`

Contexto · Un programa con varias funciones se depura con varios puntos de interrupción. Pero no todos sirven siempre, y dejar puntos de más es ruido que tapa la señal.

Consigna · Escribí un programa `reloj.c` con **tres funciones** (`main` , `tic` , `tac`) donde cada una imprima algo. Después:

1. Poné un punto de interrupción **por función** en las tres (`break main` , `break tic` , `break tac`).
2. Poné un punto de interrupción **por archivo y línea** dentro de `tic` .
3. Poné un punto de interrupción **temporal** en `tac` .
4. Listalos con `info breakpoints` y **anotá** qué muestra cada columna.
5. Deshabilitá uno, habilitalo de nuevo, y borrá otro.
6. Dejá **un** punto de interrupción con `commands` que imprima un mensaje y siga.

Entrada · Ninguna. **Salida** · El listado de `info breakpoints` en cada estado, y una nota de qué hace cada columna.

Escenario de prueba

```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000001129 in main() at reloj.c:12
2       breakpoint     keep y   0x0000000000001150 in tic() at reloj.c:5
3       breakpoint     keep y   0x0000000000001168 in tac() at reloj.c:9
4       breakpoint     keep y   0x0000000000001155 in tic() at reloj.c:6
5       breakpoint     dlt  y   0x0000000000001168 in tac() at reloj.c:9
```

Identificadores sugeridos · `reloj.c` , `tic` , `tac` Restricciones/pistas

- Fijate en la columna `Disp` : `keep` = permanente, `dlt` = *delete* (temporal, se borra solo).
- La columna `Enb` te dice si el breakpoint está activo (`y`) o apagado (`n`).
- El `What` te dice **dónde** va a parar: eso es el mapa de tu investigación.

Criterios de aceptación

- ☐ Hay al menos **un** breakpoint por función, **uno** por archivo:línea y **uno** temporal.
- ☐ El listado de `info breakpoints` está capturado y anotado.
- ☐ Se demuestra el ciclo `disable` → `enable` y el `delete` .
- ☐ El breakpoint con `commands` imprime un mensaje **automáticamente** al dispararse.

Ejercicio 20.6 — Instrumentar con `commands`

Conceptos · `commands`, `printf`, `continue`, `break`, automatización

Contexto · A veces no querés frenar el programa: querés que **te avise** cuando pasa algo. `commands` convierte un breakpoint en un **sensor**: cada vez que se dispara, deja una nota.

Consigna · Escribí un programa `cola.c` que modele una cola con un arreglo de enteros y dos funciones: `encolar(int valor)` y `desencolar(void)`. La primera escribe en la posición `ultimo` y la segunda saca de la posición `primero`.

1. Poné un breakpoint en `encolar` con `commands` que imprima el valor recibido y el índice de la posición de escritura, y que **siga** automáticamente.
2. Corré el programa con **cinco** encolados y capturá la salida completa.
3. Ahora hacé lo mismo con `desencolar`.
4. Repetí el punto 1 pero **sin** el `continue` final: observá qué pasa y explicá por qué.

Entrada · Ninguna: el programa se prueba con entradas fijas. **Salida** · La salida del programa con las trazas de GDB intercaladas.

Escenario de prueba

```
(gdb) break encolar
(gdb) commands
> printf "encolar: valor=%d ultimo=%d\n", valor, ultimo
> continue
> end
(gdb) run
encolar: valor=10 ultimo=0
encolar: valor=20 ultimo=1
encolar: valor=30 ultimo=2
```

Identificadores sugeridos · `cola.c`, `encolar`, `desencolar`, `valores`, `primero`, `ultimo` **Restricciones/pistas**

- El `printf` de GDB usa **la misma sintaxis** que el `printf` de C: `%d`, `%s`, `%x`…
- Los argumentos de `printf` en GDB se evalúan **en el contexto del breakpoint**: es decir, en el frame de la función donde estás.
- Si `printf` dice `No symbol "valor" in current context`, es que el breakpoint no está en el frame donde vive esa variable.

Criterios de aceptación

- [ ] El breakpoint con `commands` se dispara **automáticamente** en cada llamada.
- [ ] La salida de GDB y la del programa se distinguen claramente.
- [ ] El caso sin `continue` está explicado (el programa queda parado en el primer disparo).
- [ ] El `printf` usa variables **reales del frame** del breakpoint.

Ejercicio 20.7 — Describir un tipo con el depurador

Conceptos · `print`, `ptype`, `info locals`, `info args`, `set print pretty on`, `struct`

Contexto · Antes de lógica viene la **descripción**: saber qué hay adentro de una `struct`, cómo se llama cada campo y qué tipo tiene. En el depurador podés descubrirlo sin abrir el archivo fuente, y a veces el depurador te dice **más** que el fuente.

Consigna · Escribí un programa `ficha.c` con una `struct` de al menos **cuatro campos** (con tipos distintos: `int`, `double`, `char[]` y un puntero) y una función que la recibe por puntero.

1. Poné un breakpoint **dentro** de la función.
2. Usá `ptype` para imprimir la definición del tipo de la `struct` y de cada campo.
3. Usá `print` para mostrar la `struct` **completa**, y después **campo por campo**.
4. Usá `print` sobre el **puntero** a la struct, y sobre el puntero **desreferenciado**.
5. Activá `set print pretty on` y compará cómo cambia la salida.
6. Usá `info locals` e `info args` para ver todo el frame de una sola vez.

Entrada · Ninguna. **Salida** · Un volcado anotado de cada comando, con el tipo y el valor que devolvió.

Escenario de prueba

```
(gdb) ptype struct Ficha
struct Ficha {
    int id;
    double peso;
    char nombre[32];
    int *scores;
}

(gdb) print ficha
$1 = {id = 7, peso = 72.5, nombre = "Ana", scores = 0x7fffffffe330}
```

Identificadores sugeridos · `ficha.c`, `struct Ficha`, `id`, `peso`, `nombre`, `scores` **Restricciones/pistas**

- `print` de una `struct` con un `char[]` muestra el texto **hasta el primer** `'\0'`. Si ves basura, el array no está terminado — y eso puede ser **el bug**.
- `ptype` acepta un **tipo** (`ptype struct Ficha`) o una **expresión** (`ptype ficha`).
- Con `set print pretty on` la struct se muestra en varias líneas, que es más legible cuando tiene muchos campos.

Criterios de aceptación

- [ ] `ptype` devuelve la definición completa de la struct.
- [ ] Cada campo se imprimió individualmente y coincide con su tipo.
- [ ] El puntero y su desreferencia se imprimieron **por separado** y son distintos.
- [ ] `set print pretty on` cambió la salida y el cambio está documentado.

PARTE 4 — Punteros, estructuras y control avanzado

Ahora vamos a donde se cocina el plato fuerte: **mirar lo que hay en la memoria**.

4.1 Direcciones y desreferenciación

En C, un puntero es una **variable que guarda una dirección**. GDB te deja ver las dos cosas por separado, y esa separación es la clave de todo lo que viene:

```
(gdb) print &total          # la DIRECCIÓN de total
$1 = (int *) 0x7fffffffe320

(gdb) print total            # el VALOR de total
$2 = 0

(gdb) print *&total         # la dirección, desreferenciada → el valor
$3 = 0
```

`&` significa "la dirección de". `*` significa "el valor que hay en esa dirección". Con esos dos operadores podés moverte por la memoria en cualquier dirección:

```
(gdb) print *0x7fffffffe320
$1 = 0
```

GDB acepta direcciones literales. Y si la variable es un puntero:

```
(gdb) print p               # el VALOR de p (una dirección)
$1 = 0x7fffffffe330

(gdb) print *p              # lo que HAY en esa dirección
$2 = 42
```

> ***La analogía.*** `p` *es una* ***ficha*** *con la dirección de un sobre.* `*p` *es* ***abrir*** *el sobre y ver lo que hay adentro.* `&total` *es la dirección* ***del*** *sobre. Confundir* `p` *con* `*p` *es el error conceptual más común de la primera hora — y es exactamente el error que produce el* `Segmentation Fault` *del capítulo siguiente.*

Y para ver los **tamaños**:

```
(gdb) print sizeof(total)
$1 = 4
(gdb) print sizeof(struct Ficha)
$2 = 56
```

4.2 Arreglos: los diez elementos de un vistazo

Un arreglo es un bloque **contiguo** de memoria. GDB lo sabe y lo muestra completo:

```
(gdb) print datos
$1 = {4, 8, 15, 16, 23, 42}

(gdb) print datos[2]
$2 = 15

(gdb) print *datos
$3 = 4
```

Y con la sintaxis de **arreglo artificial** podés ver una porción arbitraria:

```
(gdb) print *datos@10
$1 = {4, 8, 15, 16, 23, 42, 0, 0, 0, 0}
```

`p *datos@10` significa: "tratá `datos` como si fuera un arreglo de 10 elementos y mostrame los 10". El `@10` es el **artificial array** de GDB. Es la herramienta clave para mirar buffers que **se desbordaron**: si el arreglo real tiene 6 elementos y el buffer tiene 10, los últimos cuatro son basura.

> ***Momento ajá.*** *La sintaxis del arreglo artificial es una de esas cosas que parece un error de tipeo la primera vez que la ves.* `@` *no es "dirección": es el operador que le dice a GDB "esto que sigue es un arreglo de* ***N*** *elementos". Masterizala: la vas a usar muchísimo.*

Dos cosas más:

```
(gdb) print &datos
$1 = (int (*)[6]) 0x7fffffffe330
```

`&datos` es la dirección del arreglo **completo** (tipo `int (*)[6]`), mientras que `datos` decae a puntero al primer elemento. Son direcciones distintas (¡y por eso `&datos + 1` avanza 24 bytes, no 4!).

```
(gdb) set print elements 0
```

Con esto, los arreglos se muestran **enteros**, sin el límite de 200 elementos.

4.3 Estructuras: dos sintaxis, un mismo dato

Una `struct` se imprime completa, y después podés entrar a sus campos:

```
(gdb) print ficha
$1 = {id = 7, peso = 72.5, nombre = "Ana", scores = 0x7fffffffe330}

(gdb) print ficha.id
$2 = 7

(gdb) print ficha.peso
$3 = 72.5
```

Con **punteros** a struct, la sintaxis cambia: usás `->` en lugar de `.`:

```
(gdb) print *pf
$1 = {id = 7, peso = 72.5, nombre = "Ana", scores = 0x7fffffffe330}

(gdb) print pf->id
$2 = 7

(gdb) print pf->nombre
$3 = "Ana"
```

Y si querés ver un **arreglo de structs**:

```
(gdb) print *fichas@3
$1 = {{id = 1, ...}, {id = 2, ...}, {id = 3, ...}}
```

La trampa N.º 1 de las structs. `print ficha` *y* `print *pf` *muestran lo mismo, pero* `print ficha.id` ***no compila*** *si* `ficha` *es un puntero, y* `print pf->id` ***no compila*** *si* `pf` *no lo es. GDB te avisa con un error bastante críptico. La regla: si tenés puntero, usá* `->` *; si tenés la struct misma, usá* `.` *.*

Para ver la **definición** del tipo (no los valores):

```
(gdb) ptype struct Ficha
struct Ficha {
    int id;
    double peso;
    char nombre[32];
    int *scores;
}
```

Y para ver **dónde vive cada campo dentro de la struct** (muy útil para entender el *padding* del capítulo `08`):

```
(gdb) ptype /o struct Ficha
struct Ficha {
    int id;        /* offset 0, 4 bytes */
    double peso;   /* offset 8, 8 bytes */
    char nombre[32]; /* offset 16, 32 bytes */
    int *scores;   /* offset 48, 8 bytes */
}
```

4.4 Breakpoints condicionales: el filtro que te salva del corso

Un breakpoint normal frena **siempre**. En un bucle de 100.000 iteraciones, eso es una tortura. Un breakpoint condicional frena **solo cuando se cumple la condición**:

```
(gdb) break procesar
(gdb) condition 1 i == 500
```

O todo de una:

```
(gdb) break procesar if i == 500
```

Y hay un comando gemelo que es todavía más útil: `ignore`.

```
(gdb) break procesar
(gdb) ignore 1 499
```

`ignore 1 499` significa: "el breakpoint 1, **ignorá** las primeras 499 veces que se dispare". Con eso, en la iteración 500 te frena **exactamente**. Es la manera de saltar a una iteración sin que te frene en todas las anteriores.

Y el combo definitivo, el que se usa todo el tiempo:

```
(gdb) break procesar
(gdb) condition 1 i == 500
(gdb) commands
> silent
> print i
> print total
> continue
> end
```

El `silent` (que solo existe dentro de `commands`) evita que GDB anuncie el breakpoint cada vez. El resultado es un programa que **se auto-explica**: imprime lo que te interesa y sigue.

> ***Para guardar.*** *La combinación* `break` + `condition` + `ignore` + `commands` *es, honestamente, la mitad de la potencia de GDB. Es lo que convierte "un depurador" en "un instrumento de medición".*

4.5 Puntos de observación: `watch`

Un breakpoint dice "frená en **este punto**". Un **watchpoint** dice "frená cuando **esto cambie**". Es la diferencia entre "fijate en la línea 20" y "fijate en el valor de `count`".

```
(gdb) watch count
Hardware watchpoint 2: count
(gdb) continue
Old value = 0
New value = 1
Hardware watchpoint 2: count
```

GDB frena, te muestra el valor viejo y el nuevo, y te deja ver **dónde** cambió. Eso te da la línea exacta del bug en un solo comando. Es la herramienta más precisa del capítulo para diagnosticar corrupción de datos.

Hay tres variantes:

Comando	Se dispara cuando…	Uso típico
`watch expr`	El **valor** de `expr` cambia	"¡alguien me cambió el contador!"
`rwatch expr`	Alguien **lee** `expr`	"¡alguien está leyendo esto y no debería!"
`awatch expr`	Alguien **lee o escribe** `expr`	"¡alguien ni lo tocaría!"

Y una variante técnica importante:

```
(gdb) watch -l count
```

El `-l` (de *location*) hace que el watchpoint vigile **la dirección**, no el valor. Importa cuando la variable está optimizada (GDB la tiene en un registro y no puede vigilarla) o cuando el scope cambió (la variable fue Optimizada *out* y ahora hay otra con el mismo nombre).

Para administrarlos:

```
(gdb) info watchpoints
```

> ***El límite real de los watchpoints.*** *Los watchpoints usan* ***hardware*** *del procesador, y la CPU tiene un límite (típicamente 4). Si pedís más, GDB avisa: "Hardware watchpoint 4: - expression". Y hay otro detalle: el hardware no existe en todas las arquitecturas con la misma granularidad. En una máquina de 64 bits, un watchpoint de 4 bytes funciona perfecto.*

4.6 La pila de llamadas: quién llamó a quién

Cada vez que un programa llama a una función, se apila un **marco** (frame) en la pila. La pila de llamadas es la **historia** de cómo llegaste hasta donde estás.

El comando es `bt` (*backtrace*):

```
(gdb) bt
#0  sumar (x=5) at math.c:5
#1  0x0000000000001129 in main () at demo.c:12
#2  0x0000000000001196 in __libc_start_main () at ../csu/libc-start.c:308
#3  0x00000000000011a3 in _start () at ../csu/crt1.c:101
```

Leélo así:

- `#0` es el frame **actual**: estás adentro de `sumar`, con `x = 5`.
- `#1` es quien la llamó: `main`, en la línea 12.
- `#2` y `#3` son el arranque del proceso (la biblioteca de C y el punto de entrada).

El formato es siempre `#N función (args) en archivo:línea`. Ese **archivo:línea** es oro: te dice exactamente en qué línea de qué archivo hubo la llamada.

Moverse entre los frames

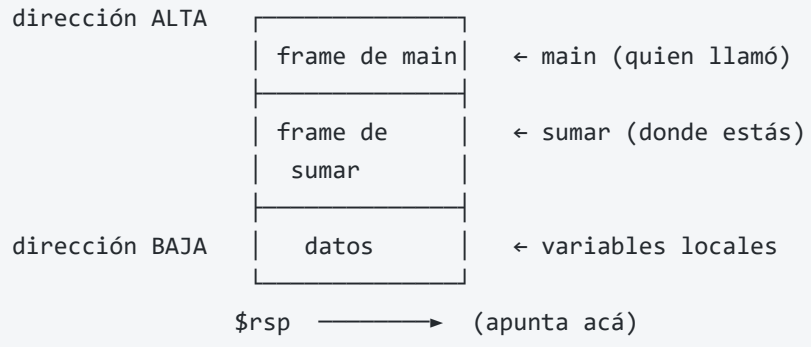
```
(gdb) frame 1        # move al frame #1
(gdb) f 1            # abreviado
(gdb) up             # un frame hacia arriba (hacia quien llamó)
(gdb) down           # un frame hacia abajo (hacia el más interno)
```

Y dentro de un frame:

```
(gdb) info frame     # la dirección del frame, el PC guardado, el SP, el FP
(gdb) info args      # los argumentos de ESE frame
(gdb) info locals    # las variables locales de ESE frame
```

> ***El poder real.*** *Cuando estás en el frame* `#0` *y mirás* `total`*, ves el* `total` *de* `sumar`*. Cuando hacés* `up` *y mirás* `total`*, ves* ***otro*** `total`*: el de* `main`*. Misma variable en el código, distinta variable en la realidad. Por eso los bugs más sutiles de C son bugs de* ***alcance*** *(capítulo* `05`*), y por eso* `up` */* `down` *son la herramienta para cazarlos.*

La pila crece hacia abajo



El puntero de pila (`$rsp`) **baja** a medida que el programa llama a funciones: cada llamada reserva un bloque. Cuando la función returns, el bloque se libera y `$rsp` vuelve a subir.

Práctica de la Parte 4

Ejercicio 20.8 — Anatomía de un puntero

Conceptos · `&`, `*`, direcciones, `print`, `ptype`, `sizeof`, punteros a heap

Contexto · Un puntero tiene **dos** caras: la dirección que guarda y el valor que hay ahí. Confundirlas es el error conceptual que después produce un `Segfault`. Este ejercicio te hace ver las dos, explícitamente.

Consigna · Escribí un programa `punteros.c` que:

1. declare una variable local `int numero = 42;`
2. declare un puntero `int *p = №`
3. haga un `malloc` de un `int`, le escriba un valor, y lo recorra con un puntero.

Compilá con `-g3 -O0` y, con un breakpoint en el `main`:

1. Imprimí `&numero`, `numero`, `p` y `*p`.
2. Verificá que `p` y `&numero` son **el mismo** número.
3. Imprimí el puntero del `malloc`, su contenido (`*ptr`) y el `sizeof` de ambas cosas.
4. Mirá la **diferencia entre el puntero y la variable**: `print p` y `print *p` dan cosas distintas. Anotá cuál es cuál.
5. Liberá la memoria y comprobá que el puntero **sigue apuntando** a algo (¿por qué?).

Entrada · Ninguna. **Salida** · Un volcado anotado de cada `print`, con qué representa cada valor.

Escenario de prueba

```
(gdb) print &numero
$1 = (int *) 0x7fffffffe320
(gdb) print numero
$2 = 42
(gdb) print p
$3 = (int *) 0x7fffffffe320
(gdb) print *p
$4 = 42
```

Identificadores sugeridos · `punteros.c`, `numero`, `p`, `ptr`, `valor`
Restricciones/pistas

- El `malloc` devuelve `void *` : en C hay que convertirlo (capítulo `07`). GDB puede mostrarlo como `(int *) 0x...` .
- El puntero liberado **no se vuelve** `NULL` : `free` no lo toca. Volver a leerlo es *undefined behavior* (capítulo `09`) y con suerte vas a ver el valor viejo.
- Usá `p/x` para ver las direcciones en hexadecimal, que es más legible.

Criterios de aceptación

- ☐ Se imprimieron las cuatro formas (`&numero` , `numero` , `p` , `*p`) y se compararon.
- ☐ Se comprobó que `p` y `&numero` coinciden.
- ☐ El volcado distingue explícitamente "dirección" de "valor".
- ☐ El caso del puntero liberado está explicado con el concepto de *dangling pointer*.
- ☐ La memoria del `malloc` **sí** se liberó (con `free`).

Ejercicio 20.9 — El anzuelo: un breakpoint condicional

Conceptos · `break ... if` , `condition` , `ignore` , `commands` , `silent` , `continue`

Contexto · Un programa recorre una lista de 10.000 elementos. El problema aparece **solo** con uno de ellos. Frenar 10.000 veces no es depurar: es sufrir.

Consigna · Escribí un programa `buscar.c` que reciba por `argv` un **número objetivo** y recorra un arreglo de 10.000 enteros generado con una fórmula **determinista** (por ejemplo `valor = (i * 7919) % 10000`), informando el índice donde lo encontró.

1. Corré el programa **sin** depurador y anotá el índice que devuelve.
2. En GDB, poné un breakpoint en la línea donde se informa el índice y usá `print` para ver el valor en cada iteración.
3. Ahora reemplazá ese breakpoint por uno **condicional** que solo se dispare cuando el valor actual sea el que buscás.
4. Repetí con `ignore` para saltar directo a la iteración que te interesa.
5. Sumá `commands` + `silent` + `continue` para que el programa **imprima solo el dato que te importa** y siga hasta el final.

Entrada · Un número objetivo (por `argv` , capítulo `13`). **Salida** · El índice donde se encontró el valor.

Escenario de prueba

```
$ ./buscar 4321
encontrado en el indice 8123

(gdb) break buscar.c:18 if valor == 4321
Breakpoint 1 at 0x1150: file buscar.c, line 18.
(gdb) run 4321
...
Breakpoint 1, main () at buscar.c:18
18          printf("encontrado en el indice %d\n", i);
```

Identificadores sugeridos · `buscar.c` , `valor` , `objetivo` , `indice` , `total`
Restricciones/pistas

- La condición se evalúa **en el contexto del breakpoint**: usá los nombres de variables que existen **ahí**, no las de otra parte del programa.
- Si la condición nunca se cumple, GDB simplemente nunca para. Eso también es información.
- `ignore` cuenta **disparos**, no iteraciones: si el breakpoint está en una línea que se ejecuta dos veces por iteración, el número es el doble.

Criterios de aceptación

- ☐ El índice que devuelve el programa **sin** depurador es el mismo que el que encontraste con GDB.
- ☐ El breakpoint condicional se dispara **una sola vez**.
- ☐ La variante con `ignore` también llega al mismo lugar.
- ☐ La variante con `commands` + `silent` + `continue` imprime el dato y **el programa termina solo**.
- ☐ La diferencia entre "parar siempre" y "parar solo cuando importa" está explicada.

Ejercicio 20.10 — El centinela: un watchpoint

Conceptos · `watch`, `rwatch`, `awatch`, `watch -l`, `info watchpoints`, `print`

Contexto · Una variable que debería tener siempre el mismo valor de pronto tiene otro. Nadie sabe quién la cambió. Con `print` no lo vas a encontrar: tendrías que adivinar dónde mirar. Con un **watchpoint**, GDB te lo dice solo.

Consigna · Escribí un programa `centinela.c` que:

1. tenga una variable global `int saldo = 100;`
2. tenga **cuatro** funciones que la toquen: dos la **modifican** de forma correcta (sumando y restando valores válidos) y dos la **modifican mal** (asignando un valor que no corresponde).
3. un `main` que las llame en un orden dado.

Con un breakpoint en el `main`, hacé:

1. `watch saldo` y seguí la ejecución hasta que se dispare.
2. Anotá el **valor viejo** y el **valor nuevo**, y **en qué línea** se dispara.
3. Repetí hasta identificar **cuál de las cuatro funciones** rompe el saldo.
4. Con `awatch` en lugar de `watch`, observá también las **lecturas**.
5. Con `info watchpoints` listá los que definiste.

Entrada · Ninguna. **Salida** · La identificación de la función culpable, con la evidencia del watchpoint.

Escenario de prueba

```
(gdb) watch saldo
Hardware watchpoint 2: saldo
(gdb) continue
Old value = 100
New value = 250
Hardware watchpoint 2: saldo
    0x0000000000001152 in aplicar_bonus () at centinela.c:24
```

Identificadores sugeridos · `centinela.c`, `saldo`, `aplicar_bonus`, `aplicar_descuento`, `cerrar_caja` **Restricciones/pistas**

- El watchpoint necesita poder **evaluar** la expresión. Si la variable está en un registro (optimizada), GDB avisa: *"watchpoint on a register"*. Compilá con `-O0` y evitás el problema.
- `watch` sobre una **constante global** no tiene sentido: nunca cambia.
- El hardware tiene un límite de watchpoints (típicamente 4). Si pedís más, GDB lo dice.

Criterios de aceptación

- [ ] El watchpoint se dispara y muestra valor viejo y valor nuevo.
- [ ] La línea exacta donde el valor cambia está identificada.
- [ ] La función culpable se determinó por **evidencia** (no por lectura del fuente).
- [ ] `awatch` mostró también las lecturas, y la diferencia con `watch` está explicada.
- [ ] El programa tiene **al menos** un dato que queda inconsistente, y eso se detecta con el watchpoint.

Ejercicio 20.11 — La escalera: recorrer la pila de llamadas

Conceptos · `bt`, `frame`, `up`, `down`, `info frame`, `info args`, `info locals`, `finish`

Contexto · Cuando el bug está "en un nivel más abajo", la pregunta es siempre la misma: **¿quién llamó a esta función, y con qué datos?** La pila de llamadas es el historial de eso.

Consigna · Escribí un programa `escalera.c` con **cuatro** funciones anidadas: `nivel1` llama a `nivel2`, que llama a `nivel3`, que llama a `nivel4`. Cada una recibe un `int` y lo transforma de una forma distinta, y cada una tiene una variable local con un nombre **repetido** (por ejemplo, todas se llaman `valor`).

Con un breakpoint en `nivel4`:

1. `bt` — dibujá la escalera completa.
2. `print valor` — ¿de qué `valor` es este?
3. `up` → `print valor` → `up` → `print valor` → repetí hasta `main`.
4. Anotá **el valor de `valor` en cada nivel**. ¿Son el mismo? ¿Por qué no?
5. `info args` e `info locals` en dos niveles distintos.
6. Usá `finish` para volver al frame de arriba y `down` para bajar otra vez.
7. `info frame` en el frame actual: ¿qué datos da?

Entrada · Ninguna. **Salida** · La escalera anotada, con el valor de `valor` en cada escalón.

Escenario de prueba

```
(gdb) bt
#0  nivel4 (valor=7) at escalera.c:20
#1  nivel3 (valor=14) at escalera.c:14
#2  nivel2 (valor=21) at escalera.c:9
#3  nivel1 (valor=28) at escalera.c:5
#4  main () at escalera.c:30
```

Identificadores sugeridos · `escalera.c` , `nivel1` , `nivel2` , `nivel3` , `nivel4` , `valor` **Restricciones/pistas**

- El **mismo nombre** de variable local en distintos frames son **variables distintas**: cada frame tiene su propia copia en la pila. Es el punto central del ejercicio.
- `up` y `down` cambian el frame **actual**: a partir de ahí, `print` , `info locals` y `info args` miran a ese frame.
- `finish` **ejecuta** el resto de la función: el programa avanza hasta que `nivel4` retorna.

Criterios de aceptación

- ☐ La escalera completa (`bt`) está dibujada con los argumentos de cada nivel.
- ☐ El valor de `valor` en **cada** escalón está anotado y se ve que **difieren**.
- ☐ Se explicó por qué difieren (cada frame tiene su propia copia).
- ☐ `info args` e `info locals` se usaron en **dos** frames distintos.
- ☐ `up` , `down` y `finish` se usaron y su efecto se documentó.

PARTE 5 — Diagnóstico de errores críticos de memoria

Acá empieza la parte que justifica la existencia de un depurador. Los bugs de lógica (una condición mal escrita, un `>` en vez de un `>=`) se ven leyendo el código. Los bugs de **memoria** —un puntero inválido, un arreglo desbordado, una fuga— **no se ven leyendo el código**: se ven **mirando la memoria mientras el programa corre**.

5.1 `Segmentation Fault` : qué es realmente

Cuando el programa se cuelga con `Segmentation fault` , el mensaje que ves es:

```
Segmentation fault (core dumped)
```

Traducción: **el kernel recibió una señal 11 (`SIGSEGV`)** porque el programa intentó acceder a una dirección de memoria que **no le pertenece** o que **no existe**.

> ***Pensalo como el portero de un edificio.*** *El proceso tiene un "plano" de qué direcciones de memoria puede tocar (lo que el kernel le asignó). Cuando el programa pide una dirección que está* ***fuera de ese plano****, el kernel le manda la señal y lo echa. No es un bug de la compilación ni del sistema: es el* ***sistema protegiendo su memoria****.*

Las **causas** típicas en C son casi siempre cuatro:

Causa	Qué pasó	Cuándo la encontrás
Puntero nulo	Desreferencias un puntero que es `NULL`	La más común de largo
Fuera de límites	Escribís más allá del final de un arreglo o de un buffer	Muy común con índices calculados
Use after free	Usás memoria que ya devolviste con `free`	Cuando el `free` está lejos del uso
Desbordamiento de pila	Recursión infinita: la pila crece hasta el límite	Con funciones recursivas

> ***El mensaje NO te dice la causa.*** *"Segmentation fault" es el síntoma. Te dice* ***dónde*** *y* ***qué****, no* ***por qué****. Todo el trabajo de esta parte es pasar del síntoma a la causa.*

Y hay un cuarto comportamiento que parece un `Segfault` pero no lo es: cuando escribís sobre un **dato de solo lectura** (como un literal de cadena, que vive en `.rodata`), el kernel te manda `SIGSEGV` también. El mensaje de GDB suele ser distinto:

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000001150 in main () at demo.c:8
```

La diferencia clave: **un desbordamiento de un arreglo con `malloc` no siempre revienta**. Si escribís 4 bytes de más en un bloque de 100, el programa sigue vivo y corrupto. El `Segfault` es solo el caso donde el daño es tan grande que toca una página que no te pertenece. **Ahí está la trampa: el programa que "no se rompe" puede estar peor que el que se rompe.**

5.2 Los core dumps: la caja negra

Cuando un proceso muere por una señal fatal, el kernel puede guardar una **fotografía** de su memoria: el **core dump**. Es el equivalente en software de la caja negra del avión: el programa ya no está, pero su estado final quedó guardado en un archivo.

Es la herramienta clave para el diagnóstico **post-mortem**: no hace falta reproducir el crash en cámara lenta; se analiza la foto.

Habilitarlos

En Linux, el límite de tamaño del core se controla con `ulimit`:

```
$ ulimit -c
0                       ← 0 = DESHABILITADO (es el valor por defecto en muchos sistemas)

$ ulimit -c unlimited   ← habilitar, sin límite de tamaño
$ ulimit -c
unlimited
```

Ese `ulimit` aplica a los procesos que lances **desde esa terminal**. Si lo ponés y después abrís otra terminal, no hereda el cambio.

> ***Ojo con esto.*** *El mensaje* `Segmentation fault (core dumped)` *dice "core dumped"* ***solo si*** *el core se generó de verdad. Si tu* `ulimit -c` *está en 0, el programa se cuelga igual pero* ***no queda archivo****. Es la confusión más común: "dice core dumped pero no encuentro el archivo".*

Dónde queda el archivo

Eso lo decide el kernel, y lo podés ver:

```
$ cat /proc/sys/kernel/core_pattern
core
```

Valor	Significado
`core`	El archivo se llama `core` y aparece en el **directorio actual**
`core.%p`	Se llama `core.<pid>` (útil con varios programas a la vez)
`|/usr/lib/systemd/systemd-coredump …`	Lo captura **systemd-coredump** (no hay archivo en tu carpeta)
`|/usr/share/apport/apport …`	Lo captura **Apport** (Ubuntu)

En los sistemas con `systemd-coredump` o `apport`, el core se guarda en un lugar del sistema y se consulta con `coredumpctl`:

```
$ coredumpctl list
$ coredumpctl gdb <PID>
```

> ***Para guardar.*** *En Windows no existe* `ulimit -c` *: el volcado lo genera el* ***Windows Error Reporting****. Se activa por registro (* `HKLM\SOFTWARE\Microsoft\Windows\Windows Error Reporting\LocalDumps` *) y produce archivos* `.dmp`*, que se abren con WinDbg o con el GDB de MSYS2 (con más difficulties). En macOS, los* ***core dumps*** *se activan con* `ulimit -c unlimited` *y se llaman* `core.<pid>`*. Si estás en Windows y el ejercicio de core dumps no te cierra, no estás haciendo nada mal: es que la plataforma no tiene la misma caja negra.*

5.3 Post-mortem: interrogar el cadáver

Una vez que tenés el archivo del core, lo abrés **con el programa adentro**:

```
$ gdb ./demo core.core
```

GDB carga el estado guardado y te deja interrogarlo. Y ahora el programa **no corre**: ya terminó. Solo podés mirar.

Los tres comandos del diagnóstico post-mortem:

```
(gdb) bt
#0  0x0000000000001150 in main () at demo.c:12
#1  0x0000000000001196 in __libc_start_main () at ../csu/libc-start.c:308
```

```
(gdb) info registers
rax            0x0                 0
rbx            0x7fffffffe3a0      0x7fffffffe3a0
rip            0x555555554150      0x555555554150 <main+32>
rsp            0x7fffffffe300      0x7fffffffe300
```

```
(gdb) x/i $pc
=> 0x555555554150 <main+32>: mov    %eax,(%rbp-0x4)
```

Leélo así:

- `bt` te dice **dónde** se rompió: en qué función y en qué línea.
- `info registers` te dice **el estado** en ese instante: qué había en los registros.
- `x/i $pc` te dice **qué instrucción** se estaba por ejecutar.

Esa tríada es el **triángulo forense** del crash. Con esos tres datos, muchas veces ya sabés qué pasó: "estaba por escribir en `(rbp-0x4)`, o sea, en una variable local, y `rax` valía `0`" → es un **puntero nulo** desreferenciado.

Y si querés el contexto completo:

```
(gdb) info frame
(gdb) p *ptr
(gdb) thread apply all bt
```

> ***La ventaja brutal del post-mortem.*** *Con un core, analizás el crash **en frío**, con calma, sin cronómetro y sin que el programa se te escape. Podés abrir el mismo core cien veces y probar hipótesis distintas. Con un depurador en vivo, cada intento te cuesta una ejecución.*

5.4 `x`: el microscopio de la memoria cruda

El comando `x` (de *examine*) es el que te deja mirar la memoria **byte a byte**, sin que el programa tenga que imprimirla. Su formato es:

```
x/NFU  dirección

N  = cuántos elementos mirar
F  = en qué formato mostrarlos
U  = de qué tamaño cada uno
```

Los formatos (F)

Letra	Formato	Ejemplo
`x`	Hexadecimal	`0x00007ffd`
`d`	Decimal con signo	`34567`
`u`	Decimal sin signo	`34567`
`o`	Octal	`42017`
`t`	Binario	`100001011111111`
`c`	Caracteres	`68 'D'`
`s`	Cadena (hasta el `'\0'`)	`"hola"`
`i`	Instrucción	`mov %eax,(%rbp-0x4)`
`a`	Dirección	`0x7fffffffe320`
`f`	Punto flotante	`3.14`

Los tamaños (U)

Letra	Tamaño
`b`	1 byte
`h`	2 bytes (*halfword*)
`w`	4 bytes (*word*)
`g`	8 bytes (*giant word*)

Ejemplos concretos

```
(gdb) x/4xb $sp
0x7fffffffe300: 0x01 0x02 0x00 0x00

(gdb) x/8xw $sp
0x7fffffffe300: 0x00000001 0x00000002 0x00000000 0x00000000
0x7fffffffe310: 0x00000000 0x00000000 0x00000000 0x00000000

(gdb) x/s nombre
0x7fffffffe340: "Ana"

(gdb) x/i $pc
=> 0x555555554150 <main+32>: mov    %eax,(%rbp-0x4)

(gdb) x/16c buffer
0x7fffffffe360: 104 'h' 101 'e' 108 'l' 108 'l' 111 'o' 32 ' ' 119 'w' 111 'o' 114 'r'
108 'l' 100 'd' 33 '!' 10 '\n' 0 '\0'
```

El último es hermoso: **cada byte de un texto, con su carácter al lado**. Es la forma de ver exactamente qué hay escrito en un buffer y **dónde empieza la basura**.

> ***El uso clásico: `x` para auditar un buffer.*** *Si un programa llena un `char buf[32]` con un texto de 40 caracteres, `x/40cb buf` te muestra los 32 primeros bytes (tu texto, ya desbordando) y los 8 siguientes: la **basura que pisaste**. Es la forma de ver un desbordamiento de buffer, no de inferirlo.*

5.5 Los sanitizers: el guardián que te avisa antes del crash

Los core dumps y `x` son **forense**: actúan **después** del desastre. Los **sanitizers** son **preventivos**: instrumentan el programa para que te avise **en el momento exacto** en que hacés algo prohibido.

Son **parte del compilador**: no son un programa externo, son código extra que el compilador mete dentro de tu binario.

```
$ gcc -g3 -O0 -fsanitize=address -o demo_asan demo.c
```

Sanitizer	Bandera	Qué detecta
AddressSanitizer (ASan)	`-fsanitize=address`	Buffer overflows, use-after-free, **fugas de memoria**, doble `free`
UndefinedBehaviorSanitizer (UBSan)	`-fsanitize=undefined`	Overflow con signo, `NULL` desreferenciado, shifts imposibles, alineación
LeakSanitizer (LSan)	`-fsanitize=leak`	Fugas de memoria (standalone; ASan ya lo incluye)
MemorySanitizer (MSan)	`-fsanitize=memory`	Lectura de memoria **no inicializada** (solo Clang, y con libc instrumentada)
ThreadSanitizer (TSan)	`-fsanitize=thread`	**Carreras de datos** entre hilos (Parte 7)

Qué se ve cuando ASan encuentra algo

```
==12345==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60200000eff4 at pc
0x0000004012a3 bp 0x7ffd... sp 0x7ffd...
READ of size 4 at 0x60200000eff4 thread T0
    #0 0x4012a0 in main /tmp/demo.c:12
    #1 0x4013f7 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+2.34)
    #2 0x401260 in _start (/lib/x86_64-linux-gnu/libc.so.6+2.34)

0x60200000eff0 is located 0 bytes after 12-byte region [0x60200000eff0,0x60200000effc)
allocated by thread T0 here:
    #0 0x7f... in malloc
    #1 0x4011ab in main /tmp/demo.c:8

SUMMARY: AddressSanitizer: heap-buffer-overflow /tmp/demo.c:12 in main
```

Leélo con cuidado, porque tiene **todo**:

- **Qué pasó**: `heap-buffer-overflow` (te leíste más allá de un bloque del heap).
- **Dónde**: `/tmp/demo.c:12` — el archivo y la línea exacta.
- **La cadena de llamadas**: `#0`, `#1`, `#2` — quién llamó a quién.
- **La evidencia forense**: `is located 0 bytes after 12-byte region` — el bloque era de 12 bytes y leíste justo después del final.
- **Quién lo reservó**: `allocated by thread T0 here: ... main /tmp/demo.c:8`.

Eso es **más** que un `bt` de GDB. ASan te da el lugar exacto de la escritura culpable.

Y la fuga de memoria

```
=================================================================
==12345==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 40 byte(s) in 1 object(s) allocated from:
    #0 0x7f... in malloc
    #1 0x4012b0 in main /tmp/demo.c:20
    #2 0x4013f7 in __libc_start_main ...

SUMMARY: AddressSanitizer: 40 byte(s) leaked in 1 allocation(s).
```

`Direct leak` = se perdió la referencia y no hay puntero que apunte al bloque. `Indirect leak` = el bloque perdido era el contenido de otro bloque perdido.

Combinar el sanitizer con GDB

Acá está el truco que más te da: hacer que el sanitizer **se detenga** en el error, para que puedas depurarlo con GDB como cualquier otro crash.

```
$ ASAN_OPTIONS=abort_on_error=1:detect_leaks=1 ./demo_asan
```

Con `abort_on_error=1`, en vez de terminar limpiamente, el sanitizer lanza un `SIGABRT`. Y un `SIGABRT` **se puede atrapar en GDB**:

```
(gdb) run
Continuing.
Program received signal SIGABRT, Aborted.
__asan::ReportGenericError(...) at ...
```

Ahora tenés el programa **pausado en el momento exacto** del error de memoria, con todos los comandos de GDB disponibles.

> ***Para guardar.*** *El flujo de trabajo profesional para bugs de memoria es:* ***primero*** *ASan (te dice archivo y línea exactos),* ***después*** *GDB (mirás el estado en ese punto),* ***después*** *lo arreglás. Al revés es perder tiempo.*

5.6 Valgrind: el microscopio lentísimo

Antes de los sanitizers existía (y sigue existiendo) **Valgrind**, que instrumenta el binario desde fuera y detecta muchísimos errores de memoria sin recompilar:

```
$ valgrind ./demo
==12345== Memcheck, Memerror:  ...
==12345==    at 0x400A1E: main (demo.c:12)
==12345==  Address 0x5... is 0 bytes after a block of size 12 alloc'd
==12345==    at 0x4C2A3E: malloc (vg_replace_malloc.c:...)
==12345==    at 0x4011AB: main (demo.c:8)
```

Con `--leak-check=full` agrega el reporte de fugas:

```
$ valgrind --leak-check=full --show-leak-kinds=all ./demo
```

Ventaja	Desventaja
No recompilás (funciona con cualquier binario)	Es **lentísimo**: entre 20 y 50 veces más lento
Detecta leaks, overflows, uso de memoria no inicializada	No funciona con Threads (usa helgrind/DRD para eso)

> ***Cuándo usar cada uno. Sanitizer**: estás en desarrollo, querés velocidad de iteración y el error es de memoria. **Valgrind**: no podés recompilar (binario de terceros), o querés `memcheck` sobre algo que no instrumentaste. **GDB**: siempre, para entender el estado.*

Práctica de la Parte 5

Ejercicio 20.12 — La caja negra: análisis post-mortem

Conceptos · core dump, `ulimit -c`, `core_pattern`, `gdb ./prog core`, `bt`, `info registers`, `x/i $pc`, lectura del triángulo forense

Contexto · Un programa se cuelga con `Segmentation fault` **en producción**, una sola vez, y no lo podés reproducir. El core dump es la única evidencia que te queda.

Consigna · Escribí un programa `caja.c` que **a propósito** provoque un `Segfault` de forma controlada: un puntero que se hace `NULL` y después se desreferencia.

1. Verificá `ulimit -c` y ponelo en `unlimited`.
2. Corré el programa y confirmá que se cuelga. Capturá el mensaje **exacto**.
3. Localizá el archivo del core (`ls`, `cat /proc/sys/kernel/core_pattern`).
4. Abrilo con `gdb ./caja core` (o `core.core`).
5. En el post-mortem, reuní el **triángulo forense**: `bt`, `info registers` y `x/i $pc`.
6. Escribí el diagnóstico: ¿en qué línea se rompió? ¿Qué valor tenía el puntero? ¿Cómo lo sabés a partir de la evidencia?

Entrada · Ninguna. **Salida** · El diagnóstico post-mortem, con las tres piezas de evidencia.

Escenario de prueba

```
$ ulimit -c unlimited
$ ./caja
Segmentation fault (core dumped)
$ ls
core  caja

$ gdb ./caja core
...
Program received signal SIGSEGV, Segmentation fault.
0x0000000000001150 in main () at caja.c:12
12      printf("%d\n", *nulo);
```

Identificadores sugeridos · `caja.c` , `nulo` , `core` Restricciones/pistas

- Si tu sistema usa `systemd-coredump` o `apport` , no vas a ver el archivo en tu carpeta: usá `coredumpctl list` y `coredumpctl gdb` .
- En el post-mortem, `run` **no** funciona: el programa ya terminó. Solo `bt` , `info registers` , `x/` , `print` .
- Para el diagnóstico del puntero nulo: `p nulo` te muestra el valor, y `info registers` te muestra si el registro que se iba a desreferenciar estaba en cero.

Criterios de aceptación

- ☐ El core se generó **de verdad** y se localizó su archivo (o se usó `coredumpctl`).
- ☐ El triángulo forense está completo: `bt` + `info registers` + `x/i $pc` .
- ☐ El diagnóstico identifica la línea y el **valor** del puntero, citando la evidencia.
- ☐ Se explica por qué `run` no sirve en post-mortem.
- ☐ El diagnóstico es un documento, no una captura suelta de pantalla.

Ejercicio 20.13 — El examen de memoria con `x`

Conceptos · `x/NFU`, formatos `x d u o t c s i a f`, tamaños `b h w g`, `x/s`, `x/i`, `$sp`, arreglos artificiales

Contexto · A veces la pregunta no es "¿qué vale la variable?" sino **"¿qué hay realmente escrito en esta dirección?"**. `x` responde eso, byte a byte.

Consigna · Escribí un programa `examen.c` que:

1. declare un `char buffer[32]`;
2. lo llene con un texto **más corto** que 32 usando `strcpy`;
3. declare un `int numeros[8]` y lo llene con valores conocidos;
4. declare un `char *texto = "hola mundo";`.

Con un breakpoint después de las cargas, usá `x` para:

1. Mostrar los 32 bytes del buffer en hexadecimal (`x/32xb buffer`).
2. Mostrar el buffer como caracteres (`x/32cb buffer`) y **apuntar dónde está el `'\0'`**.
3. Mostrar el arreglo de enteros en palabras de 4 bytes (`x/8xw numeros`) y **verificar que coincide** con lo que cargaste.
4. Mostrar la cadena con `x/s texto` y con `x/12cb texto`.
5. Mirar los primeros bytes de la pila con `x/8xw $sp`.
6. Desensamblar la instrucción actual con `x/i $pc`.

Entrada · Ninguna. **Salida** · Cada salida de `x` con una explicación de qué se ve.

Escenario de prueba

```
(gdb) x/16xb buffer
0x7fffffffe360: 0x68 0x6f 0x6c 0x61 0x20 0x6d 0x75 0x6e 0x64 0x6f 0x00 0x00 0x00 0x00
0x00 0x00

(gdb) x/8xw numeros
0x7fffffffe340: 0x00000001 0x00000002 0x00000003 0x00000004
0x7fffffffe350: 0x00000005 0x00000006 0x00000007 0x00000008
```

Identificadores sugeridos · `examen.c`, `buffer`, `numeros`, `texto`
Restricciones/pistas

- El `'\0'` final es **un byte con valor 0**. En el volcado hexadecimal aparece como `0x00`. Si el texto tiene 10 caracteres, el `0x00` está en la posición 11 (índice 10).
- `x/s` lee **hasta el primer** `'\0'`: si el buffer no está terminado, `x/s` se va de largo y te muestra basura (o se cuelga). Eso es información, no un bug del comando.
- El tamaño `w` son **4 bytes** (32 bits) en x86-64; `g` son 8 bytes.

Criterios de aceptación

- ☐ Se usaron **al menos** tres formatos (`x`, `d`, `c` o `s`) y **dos** tamaños (`b`, `w`).
- ☐ La posición del `'\0'` está identificada y explicada.
- ☐ El volcado del arreglo de enteros **coincide** con los valores cargados.
- ☐ `x/s` y `x/Ncb` sobre la misma cadena dan **resultados distintos**, y la diferencia está explicada.
- ☐ `x/i $pc` se usó y se interpretada.

Ejercicio 20.14 — ASan: la fuga que no se ve

Conceptos · `-fsanitize=address`, LeakSanitizer, `ASAN_OPTIONS`, `abort_on_error=1`, combinación sanitizer + GDB, `free`, ownership de punteros

Contexto · Una fuga de memoria no rompe el programa: el programa funciona **perfectamente** y simplemente usa más y más memoria hasta que el sistema lo mata. Es el bug más difícil de ver "a ojo" y el más fácil de ver con un sanitizer.

Consigna · Escribí un programa `fuga.c` que:

1. tenga una función `crear_lista(int n)` que arma una lista de `n` elementos con `malloc` y la devuelve;
2. tenga una función `filtrar(Lista *l)` que cree una **nueva** lista con los elementos pares y la devuelva;
3. el `main` llame a las dos y **no** libere la lista original.

Después:

1. Compilá **sin** sanitizer y corré el programa con un `n` grande: observá que **no** falla.
2. Recompilá con `-fsanitize=address -g3 -O0` y corré el mismo programa.
3. Leé el reporte de ASan: ¿qué dice? ¿de dónde salió la memoria perdida?
4. Recompilá con `ASAN_OPTIONS=abort_on_error=1` y metelo en GDB: ¿dónde se detiene?
5. **Arreglá** el programa (liberando todo lo que corresponde) y verificá que ASan deja de reportar la fuga.

Entrada · Un `n` (por `argv`). **Salida** · La memoria usada, y el reporte de ASan antes y después del arreglo.

Escenario de prueba

```
$ gcc -g3 -O0 -fsanitize=address -o fuga_asan fuga.c
$ ./fuga_asan 100
=================================================================
==1234==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 800 byte(s) in 100 object(s) allocated from:
    #0 0x7f... in malloc
    #1 0x4012b0 in crear_lista fuga.c:14
    #2 0x4013f7 in main fuga.c:31

SUMMARY: AddressSanitizer: 800 byte(s) leaked in 100 allocation(s).
```

Identificadores sugeridos · `fuga.c`, `crear_lista`, `filtrar`, `Lista`, `n`
Restricciones/pistas

- El tamaño del reporte es `n` (el número de objetos) por el tamaño de cada uno. Si el reporte dice "800 byte(s) in 100 object(s)", cada objeto ocupa 8 bytes.
- **Una fuga no es un error de ASan "de buffer"**: es un reporte distinto (`LeakSanitizer`). Si no aparece, revisá que el programa **termine** (si queda en un bucle infinito, LSan no corre).
- La compilación con `-fsanitize=address` **ya incluye** el detector de fugas: no hace falta `-fsanitize=leak` aparte.
- El arreglo sintáctico correcto de "quien libera qué" se llama **ownership**. Cuando una función devuelve memoria pedida con `malloc`, la documentación tiene que decir quién la libera.

Criterios de aceptación

- [ ] El programa **sin** sanitizer corre "bien" (y eso se documenta como el peligro).
- [ ] El reporte de ASan está capturado y se identifica **el origen exacto** de la fuga (archivo y línea de la asignación).
- [ ] El tamaño de la fuga **coincide** con lo esperado según los cálculos del enunciado.
- [ ] El programa con `abort_on_error=1` se detiene en GDB y se documentó dónde.
- [ ] **Después** de arreglar, ASan no reporta fugas (verificado, no supuesto).

PARTE 6 — Depuración de bajo nivel y ensamblador

Hasta acá, todo fue C: líneas, variables, funciones, estructuras. Ahora bajamos un piso. La máquina no sabe qué es una variable: la máquina **mueve bytes entre registros y direcciones de memoria**. GDB te deja ver esa capa, y entenderla es lo que separa a quien depura de memoria de quien reza.

> ***Una advertencia honesta antes de empezar.*** *Esta parte es la* ***menos útil*** *del capítulo para el 95 % de los bugs. No la saltees, pero sabé que la respuesta a "mi programa no funciona" casi nunca está en el ensamblador. Está en la lógica, en los punteros o en la memoria. Bajamos al ensamblador cuando la evidencia de los niveles superiores* ***no alcanza****.*

6.1 Ver el ensamblador

El desensamblado de una función

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x0000000000001129 <+0>:  push   %rbp
   0x000000000000112a <+1>:  mov    %rsp,%rbp
   0x000000000000112d <+4>:  sub    $0x20,%rsp
   0x0000000000001131 <+8>:  mov    %edi,-0x14(%rbp)
   0x0000000000001135 <+12>: mov    $0x0,-0x18(%rbp)
   0x000000000000113c <+19>: mov    -0x18(%rbp),%eax
   0x000000000000113f <+22>: add    $0x1,%eax
   0x0000000000001142 <+25>: mov    %eax,-0x18(%rbp)
   0x0000000000001145 <+28>: mov    -0x18(%rbp),%eax
   0x0000000000001148 <+31>: pop    %rbp
   0x0000000000001149 <+32>: ret
=> 0x000000000000114a <+33>: (bad)
```

Cada línea tiene la **dirección** y el **desplazamiento dentro de la función** (`<+28>`). El comando `disassemble` acepta variantes:

```
(gdb) disassemble /s main          # con el código fuente intercalado
(gdb) disassemble 0x1129,0x1140    # un rango de direcciones
(gdb) disassemble /r main          # con los bytes en crudo
```

La variante `/s` es la más útil: GDB pone el código C **al lado** del ensamblador, y ahí se ve el mapeo línea a línea.

Los registros

```
(gdb) info registers
rax            0x1                 0
rbx            0x0                 0
rcx            0x7fffffffe320      0x7fffffffe320
rdx            0x0                 0
rsi            0x0                 0
rdi            0x1                 0
rip            0x401136            0x401136 <main+13>
rsp            0x7fffffffe300      0x7fffffffe300
rbp            0x7fffffffe300      0x7fffffffe300
```

Y se pueden consultar **de a uno**, como si fueran variables:

```
(gdb) print $rax
$1 = 1
(gdb) print/x $rip
$2 = 0x401136
(gdb) print $rsp
$3 = (void *) 0x7fffffffe300
```

> ***Los tres registros que importan.*** `rip` *es el* ***puntero de instrucción****: la dirección de la instrucción que se va a ejecutar.* `rsp` *es el* ***puntero de pila****: el tope de la pila (crece hacia abajo).* `rbp` *es el* ***puntero de marco****: la base del frame actual. Con esos tres más el desensamblado, podés reconstruir a mano lo que la máquina está haciendo.*

6.2 `stepi` y `nexti`: un paso a la vez

Si `next` y `step` avanzan **una línea de C**, `stepi` y `nexti` avanzan **una instrucción de máquina**.

Comando	Qué hace
`stepi` (si)	Ejecuta **una instrucción** y para
`nexti` (ni)	Ejecuta **una instrucción**, pero **salta por encima** de las llamadas
`stepi N`	Ejecuta **N** instrucciones y para

La diferencia con `next` / `step` es la misma de siempre, pero un nivel más abajo: `stepi` **entra** en la función llamada; `nexti` pasa por encima.

El flujo clásico para mirar una instrucción de cerca:

```
(gdb) stepi
0x0000000000001131 <main+8>:   mov    %edi,-0x14(%rbp)
(gdb) x/i $pc
=> 0x0000000000001135 <main+12>: mov    $0x0,-0x18(%rbp)
(gdb) stepi
0x0000000000001135 <main+12>:  mov    $0x0,-0x18(%rbp)
```

Fijate en el patrón: `stepi` te dice en qué **terminó** la instrucción, y `x/i $pc` te muestra cuál es la **siguiente**. Es la forma de "avanzar una instrucción y mirar qué viene".

> ***El modo más cómodo: la TUI con dos ventanas.*** *`layout src` y `layout asm` lado a lado. Con `stepi` vas viendo la línea de C y la instrucción avanzar al mismo tiempo. Es la forma en que de verdad se aprende a leer ensamblador: no en un manual, sino mirando tu propio código.*

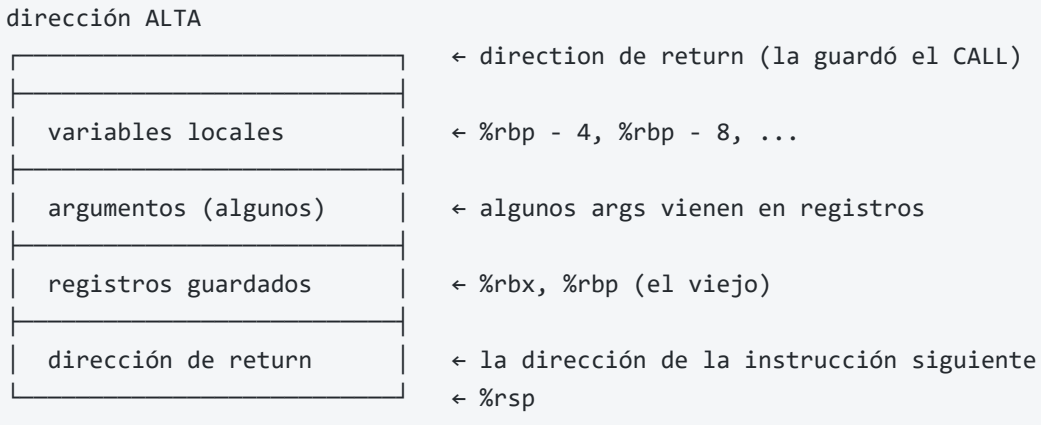
6.3 La convención de llamada: el acuerdo entre las funciones

Cuando una función llama a otra, hay un **acuerdo** sobre quién pone qué dónde. Ese acuerdo se llama **convención de llamada** (*calling convention*) y es lo que te permite entender el ensamblador de un vistazo.

En x86-64 con la convención **System V** (la de Linux y macOS):

Rol	Registros
Argumentos enteros (1º a 6º)	`rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`
Valor de retorno	`rax`
Guardados por la función llamada	`rbx`, `rbp`, `r12` – `r15`, `rsp`
Guardados por quien llama	`rax`, `rcx`, `rdx`, `rsi`, `rdi`, `r8` – `r11`
Puntero de pila	`rsp`
Puntero de marco	`rbp`
Puntero de instrucción	`rip`

Y el esqueleto de una función se ve así:



> ***Ojo con esto.*** *En ARM64 (AArch64), que es la arquitectura de tu teléfono y de la mayoría de las Raspberry Pi, los nombres son* ***distintos****: los argumentos van en* `x0` *–* `x7`*, el retorno en* `x0`*, y* `sp` */* `lr` *cumplen el papel de* `rsp` */dirección de retorno.* ***El concepto es el mismo****, los nombres cambian. GDB los abstracts, así que los comandos (* `print $x0` *) funcionan igual.*

El corollary importante para depurar: **si mirás `rdi` justo después de un `call`, y esperabas un `int`, no lo vas a encontrar**: los enteros de 32 bits viajan en la **parte baja** de `rdi`. En GDB, `print $rdi` puede mostrar `0x00000000ffffffff` y el valor real es `0xffffffff`. Por eso conviene usar `p/x $rdi & 0xffffffff`.

6.4 Parchear la memoria en caliente

Acá GDB deja de mirar y empieza a **modificar**. Y lo hace sin tocar el código fuente.

Cambiar una variable

```
(gdb) set var contador = 42
(gdb) print contador
$1 = 42
```

También funciona con miembros de estructuras:

```
(gdb) set var ficha.peso = 99.5
```

Cambiar un registro

```
(gdb) set $rax = 1
(gdb) set $rdi = 0
```

Escribir en una dirección de memoria

Acá está la forma más potente: se le dice "en **esta dirección**, escribí **estos bytes**".

```
(gdb) set {char}0x7fffffffe320 = 65
(gdb) set {int}0x7fffffffe320 = 12345
(gdb) set {int[4]}0x7fffffffe320 = {1, 2, 3, 4}
```

El `{tipo}` le dice a GDB **cómo interpretar** los bytes que escribís. Sin el tipo, GDB no sabe qué poner.

Y para **parchear código** (reemplazar una instrucción por otra):

```
(gdb) x/3i $pc
=> 0x401136 <main+13>: mov    %eax,(%rbp-0x4)
   0x401139 <main+16>: add    $0x1,%eax
   0x40113c <main+19>: nop

(gdb) set {char}0x401139 = 0x90
```

`0x90` es el byte de la instrucción `nop` (no operation) en x86: "no hagas nada". Acabas de **neutralizar** una instrucción en caliente.

> ***Para guardar: esto NO es una técnica de corrección.*** *Parchear memoria es un* ***instrumento de laboratorio****: sirve para* ***probar una hipótesis*** *("¿y si este valor fuera 0 en vez de -1?"). El programa sigue siendo el que es; lo único que cambia es lo que* ***cree*** *que vale. Jamás arregles un bug parcheando memoria en GDB: el bug sigue en el código, y el próximo* `run` *lo vuelve a traer.*

6.5 Cuándo **no** bajar al ensamblador

Esta es la parte más importante de la Parte 6, y es una regla, no una técnica.

El bug es…	¿Bajás al ensamblador?
Una condición mal escrita	**No**
Un puntero mal desreferenciado	**No**
Un arreglo que se desborda	**No** (para eso están ASan y `x`)
Una fuga de memoria	**No** (para eso están LSan y Valgrind)
Un error de **convención de llamada** entre TU código y una biblioteca	**Sí**
Un comportamiento que **no coincide** con lo que dice el manual del hardware	**Sí**
Un bug **dentro** del compilador o del runtime	**Sí**

La regla de oro: **siempre empezá por el nivel más alto que pueda explicar el bug**. El ensamblador es el último recurso, no el primero.

Práctica de la Parte 6

Ejercicio 20.15 — El mismo bug en dos niveles

Conceptos · `disassemble`, `x/i $pc`, `info registers`, `stepi`, `nexti`, convención de llamada, mapeo C ↔ ensamblador

Contexto · Ver el ensamblador de una función línea por línea es como ver el código C con un microscopio: entendés cosas que en el fuente no se ven (dónde vive cada variable, qué se copia y qué no).

Consigna · Escribí un programa `dosniveles.c` con una función que:

1. reciba dos `int` por parámetro;
2. devuelva la suma;
3. tenga **dos** variables locales.

Con `-g3 -O0`:

1. Poné un breakpoint **dentro** de la función y hacé `disassemble /s <función>`.
2. Identificá en el ensamblador: el `push %rbp`, el `sub` que reserva espacio, los `mov` que copian los argumentos a las variables locales, y el `ret`.
3. Anotá **en qué offset del marco** (`%rbp - N`) vive cada variable local, y **verificalo** con `p &variable` y `p/x $rbp`.
4. Usá `stepi` para avanzar instrucción por instrucción, y en cada paso `x/i $pc` para ver la siguiente.
5. Con `info registers`, fijate cómo van cambiando `rdi`, `rsi` y `rax`.

Entrada · Ninguna. **Salida** · El ensamblador anotado, con el equivalente en C de cada instrucción importante.

Escenario de prueba

```
(gdb) disassemble /s sumar
   0x0000000000001150 <+0>:  push   %rbp
   0x0000000000001151 <+1>:  mov    %rsp,%rbp
   0x0000000000001154 <+4>:  sub    $0x10,%rsp
   0x0000000000001157 <+7>:  mov    %edi,-0x4(%rbp)      ; a
   0x000000000000115a <+10>: mov    %esi,-0x8(%rbp)      ; b
   0x000000000000115d <+13>: mov    -0x4(%rbp),%eax
   0x0000000000001160 <+16>: add    -0x8(%rbp),%eax
   0x0000000000001163 <+19>: mov    %eax,-0xc(%rbp)      ; resultado
   0x0000000000001166 <+22>: mov    -0xc(%rbp),%eax
   0x0000000000001169 <+25>: leave
   0x000000000000116a <+26>: ret
```

Identificadores sugeridos · `dosniveles.c`, `sumar`, `a`, `b`, `resultado`
Restricciones/pistas

- El `sub $0x10,%rsp` reserva 16 bytes de pila: ese es el espacio de tus variables locales. Los offsets `-0x4`, `-0x8`, `-0xc` son **direcciones negativas desde** `%rbp`.
- Con `p/x $rbp` obtenés la dirección del marco; `p/x &a` te tiene que dar `$rbp - 0x4` (restando 4).
- En ARM64 el ensamblador es **otro** (no hay `%rbp`, hay `x29` / `x30`). Usá el desensamblado de **tu** máquina.

Criterios de aceptación

- ☐ El desensamblado con `/s` está capturado.
- ☐ Los roles de `push %rbp`, `sub %rsp`, los `mov` de argumentos y `ret` están identificados.
- ☐ El offset de cada variable local está verificado **con** `p/x &var`, no con suposición.
- ☐ `stepi` se usó al menos **cinco** veces y el estado de los registros se anotó.
- ☐ Hay un mapa "instrucción ↔ línea de C" para la función.

Ejercicio 20.16 — Parcheo en caliente: probar una hipótesis

Conceptos · `set var`, `set $registro`, `set {tipo}dirección`, `x/i $pc`, `nop`, hipótesis

Contexto · El parcheo en caliente no sirve para arreglar bugs: sirve para **probar hipótesis**. "¿Y si el problema es que este valor llega en cero?" Lo cambiás en vivo y mirás qué pasa.

Consigna · Escribí un programa `hipotesis.c` que tenga una función que divida dos números tomados de la entrada y devuelva el cociente, **sin** validar el divisor.

1. Corré el programa con un divisor en cero y anotá el síntoma.
2. En GDB, poné un breakpoint **después** de la división y usá `set var` para forzar el divisor a un valor válido. Anotá qué cambia.
3. Ahora hacé lo mismo pero con `set {int}<direccion> = <valor>`: escribí directamente en la memoria donde vive la variable.
4. Con `x/i $pc` y `set {char}<dir> = 0x90`, neutralizá **una instrucción** de la función y observá qué deja de hacer el programa.
5. Escribí, para cada caso, la **hipótesis** que probaste y la **conclusión**.

Entrada · Dos números (por `argv`). **Salida** · El cociente, o el síntoma del crash.

Escenario de prueba

```
(gdb) set var divisor = 5
(gdb) print divisor
$1 = 5

(gdb) p/x &divisor
$2 = 0x7fffffffe2f4
(gdb) set {int}0x7fffffffe2f4 = 7
(gdb) print divisor
$3 = 7
```

Identificadores sugeridos · `hipotesis.c` , `dividir` , `dividendo` , `divisor` , `cociente` **Restricciones/pistas**

- `set var` necesita el **nombre**; `set {int}<direccion>` necesita la **dirección**. Por eso `p/x &variable` es el paso previo obligatorio del segundo.
- Cambiar un **argumento** no cambia el valor original de la variable del `main` : `set var` actúa sobre el **frame actual**. Para cambiar el valor en el frame de arriba, hacé `up` primero.
- Neutralizar una instrucción **cambia la longitud del código**: si la reemplazás por un `nop` de un byte cuando la instrucción original era de tres bytes, dejaste basura. Reemplazá por una instrucción **del mismo tamaño**.

Criterios de aceptación

- ☐ El síntoma original está documentado.
- ☐ El caso `set var` cambió el comportamiento y el cambio está anotado.
- ☐ El caso `set {int}<dir>` se hizo **usando la dirección real** de la variable, mostrada con `p/x &var` .
- ☐ La neutralización con `0x90` usó una instrucción del **mismo tamaño** que la original, y se explica por qué.
- ☐ Cada experimento tiene **hipótesis, acción y conclusión** escritos.

PARTE 7 — Automatización, multihilo y técnicas avanzadas

Hasta acá, GDB fue un herramienta **interactiva**: vos escribís, GDB responde. Ahora vamos a las tres cosas que vuelven a GDB una herramienta de **ingeniería**: automatizarlo, depurar programas con varios hilos, y acoplarte a un programa que ya está corriendo.

7.1 `.gdbinit`: tu configuración personal

GDB lee un archivo llamado `.gdbinit` al arrancar y ejecuta lo que haya ahí. Es tu archivo de configuración: aliases, ajustes, layouts guardados.

Sistema	Ubicación
Linux / macOS	`~/.gdbinit` (o `$HOME/.gdbinit`)
Windows (MSYS2)	`~/.gdbinit` en tu home de MSYS2

Un `.gdbinit` típico:

```
# --- Presentación ---
set history save on
set history filename ~/.gdb_history
set history size 1000
set print pretty on
set print elements 0
set confirm off
set pagination off
set step-mode on
set disassembly-flavor intel

# --- Alias útiles ---
define pf
    print $arg0
end
define pv
    print *($arg0)
end
define btfull
    bt full
end
define reg
    info registers
end
```

Fijate en tres cosas:

- `define <nombre> ... end` define un **comando propio**. `pf total` es lo mismo que `print total`.
- `set step-mode on` hace que `next` y `step` se comporten como en algunos depuradores comerciales.
- `set confirm off` hace que GDB **no pregunte** antes de borrar todos los breakpoints o de salir. **Imprescindible en scripts**, incómodo a mano.

Para saltarlo:

```
$ gdb -nx ./demo        # ignora el .gdbinit
$ gdb -nx -q ./demo     # además, no muestra el banner
```

> ***Ojo con esto.*** *Un* `.gdbinit` *mal escrito puede* ***romper*** *GDB al arrancar (un error de sintaxis te puede dejar sin prompt). Guardá siempre una copia:* `cp ~/.gdbinit ~/.gdbinit.bak` .

7.2 Automatizar: GDB sin interacción

GDB se puede usar como una **herramienta por lotes**: le das una lista de comandos y él trabaja solo. Es la base de los "scripts de regresión" que usan muchas empresas.

```
$ gdb -batch -x script.gdb ./demo
```

Bandera	Qué hace
`-batch`	Ejecuta todo y **sale** al terminar (sin preguntar)
`-x archivo`	Ejecuta los comandos del archivo
`-ex "comando"`	Ejecuta **un** comando (se puede repetir)
`--args prog a b c`	Programa y sus argumentos
`-nx`	Ignora el `.gdbinit`

Un script típico:

```
# script.gdb
set confirm off
set pagination off

break main
run
printf "--- dentro de main ---\n"
info locals
bt
info registers rip
continue
quit
```

Y el mismo script en **una línea**, sin archivo:

```
$ gdb -batch -ex "break main" -ex run -ex "info locals" -ex "bt" ./demo
```

> ***El truco de `logging`.*** *Si además querés un* ***registro*** *de todo lo que pasó:*
>
> ```
> (gdb) set logging file informe.txt
> (gdb) set logging on
> ```
>
> *El archivo `informe.txt` va a tener la salida del programa, los comandos y los resultados. Eso es* ***evidencia entregable****.*

Y si querés automatizar de verdad (loops, condiciones, funciones), GDB tiene un intérprete de **Python** embebido:

```
(gdb) python print("hola desde Python")
hola desde Python
(gdb) python print(gdb.parse_and_eval("total"))
$1 = 42
```

Con `python` podés escribir bucles, recorrer estructuras y generar comandos. Es el nivel **avanzado** de la automatización de GDB.

7.3 Depurar programas con hilos

Un programa con `pthread` tiene **varios hilos de ejecución** que comparten la memoria. GDB los gestiona, pero hay que entender cómo.

Ver los hilos

```
(gdb) info threads
  Id   Target Id                         Frame
* 1    Thread 0x7ffff7fc4e40 (LWP 12345)  sumar () at math.c:5
  2    Thread 0x7ffff7fc4e40 (LWP 12346)  main () at demo.c:14
```

El `*` marca el hilo **actual**. Cada hilo tiene su propio **frame** (su propia pila), pero comparte las variables globales y el heap.

Cambiar de hilo

```
(gdb) thread 2
[Switching to thread 2 (LWP 12346)]
(gdb) bt
#0  main () at demo.c:14
```

Y para ver **todos** los hilos de un saque:

```
(gdb) thread apply all bt
Thread 1 (LWP 12345):
#0  sumar (x=5) at math.c:5
#1  0x401136 in main () at demo.c:14
Thread 2 (LWP 12346):
#0  main () at demo.c:14
```

Ese `thread apply all bt` es **el comando que resuelve el 90 % de los bugs de concurrencia**: te muestra el estado de cada hilo en el mismo instante.

Congelar los otros hilos

Cuando depurás un programa con hilos, el problema es que los otros hilos **siguen corriendo** mientras vos mirás. Para evitarlo:

```
(gdb) set scheduler-locking on
```

Valor	Qué hace
`off`	Todos los hilos corren (default)
`on`	Solo corre el hilo actual; los demás están **congelados**
`step`	Congela los demás mientras ejecutás una instrucción, y los deja correr en un `continue`

> ***Truco de laboratorio.*** *Depurá hilos con `set scheduler-locking on` y `thread apply all bt`. Es la combinación que convierte el "mi programa con hilos hace cosas raras" en un diagnóstico concreto.*

Y para los programas que usan hilos POSIX con funciones bloqueantes, existe el modo **non-stop**, que deja que los hilos sigan corriendo mientras vos estás parado:

```
(gdb) set non-stop on
```

7.4 Depurar procesos hijo (`fork`)

Cuando un programa llama a `fork()`, se crean **dos procesos** ("inferiores" en la terminología de GDB). Por defecto, GDB **sigue al padre** y se **desconecta** del hijo.

Para verlos:

```
(gdb) info inferiors
  Num  Description
* 1    program ./demo
```

Y para cambiar el comportamiento:

Comando	Qué hace
`set follow-fork-mode parent`	Sigue al **padre** (default)
`set follow-fork-mode child`	Sigue al **hijo**
`set follow-fork-mode child` + `set detach-on-fork off`	Sigue al hijo **pero conserva** el padre
`set schedule-multiple on`	Ejecución **concurrente** de todos los inferiores

El combo clásico para depurar un `fork()`:

```
(gdb) set follow-fork-mode child
(gdb) set detach-on-fork on
(gdb) run
```

Con eso, cuando el programa hace `fork()`, GDB **suelta** al padre y se queda pegado al hijo, que es el que te interesa.

7.5 Acoplarse a un proceso que ya está corriendo (`attach`)

Esta es la capacidad más útil en el mundo real: el programa **ya está corriendo** (en un servidor, en un entorno de pruebas, atascado en producción) y vos te le pegás.

```
$ gdb -p 12345
```

o, desde adentro de GDB:

```
(gdb) attach 12345
```

Al conectarte, GDB **frena** el proceso. Es como ponerle un freno de mano: el programa queda congelado y vos lo podés inspeccionar.

Después de inspeccionar, te **soltás** sin matarlo:

```
(gdb) detach
```

Cuadro de mando del `attach` .

Acción	*Comando*	*Efecto*
Conectarse	*`gdb -p PID`* / *`attach PID`*	*El proceso se frena*
Ver el estado	*`info threads`* , *`bt`* , *`info proc mappings`*	*Foto del proceso*
Soltar	*`detach`*	*El proceso **sigue corriendo***
Salir	*`quit`*	*GDB se va (el proceso puede quedar vivo o morir: según cómo salgas)*

Regla de oro: `detach` , nunca `kill` . *Si te colgaste con un `attach` y cerrás GDB con `quit` respondiendo "sí" a la pregunta de matar el proceso, mataste el programa que estabas investigando.*

Los permisos

En Linux, engancharse a un proceso ajeno requiere permiso de `ptrace`. El sistema lo controla con `/proc/sys/kernel/yama/ptrace_scope`:

Valor	Significado
`0`	Cualquiera puede engancharse (configuración de laboratorio)
`1`	Solo un proceso que sea ancestro puede engancharse
`2`	Solo con privilegios de administrador

Si `attach` te dice *"operation not permitted"*, no es un problema de GDB: es del sistema. Las salidas son `sudo gdb -p PID`, o poner `ptrace_scope` en 0 (solo en una máquina de pruebas).

Dos trampas reales

La primera: el shell en el medio. `gdb -p PID` a veces se engancha al **shell** que lanzó el programa en vez de al programa. Se arregla con:

```
(gdb) set startup-with-shell off
```

La segunda: la aleatorización de direcciones. Linux aleatoriza la base de los binarios para que un exploits no sirva para otro proceso (ASLR). Cuando te colgás a un proceso **ya corriendo**, GDB necesita esa aleatorización **desactivada** para que sus datos coincidan:

```
(gdb) set disable-randomization on
```

> ***Esta segunda trampa es la más frustrante**, porque el síntoma es "`p` me dice que la variable vale 0 y claramente vale 42". Si te pasa, agregá esa línea **antes** del `attach`.*

7.6 Depuración remota

Para depurar un programa que corre en **otra máquina** (un dispositivo embebido, un servidor, un Android), GDB se separa en dos partes: el **cliente** (tu máquina, con la pantalla) y el **servidor** (la máquina del programa).

El servidor es `gdbserver`:

```
# En la máquina remota:
$ gdbserver :2345 ./demo
```

Y en tu máquina:

```
$ gdb ./demo
(gdb) target remote :2345
```

O en un solo paso:

```
$ gdb --remote=:2345 ./demo
```

Dos variantes que vas a necesitar:

Variante	Qué hace	Cuándo
`target remote host:puerto`	El remoto es un proceso **en vivo**	Tu programa normal, remoto
`target extended-remote host:puerto`	Permite **reanudar** el proceso remoto	Depuración fancy, GDB scripts

Y por **cable serie** (un microcontrolador, un Arduino con GDBStub):

```
(gdb) target remote /dev/ttyUSB0
```

> ***Ojo con esto.*** *La conexión remota es una **puerta abierta** al servidor: cualquiera que conecte puede tomar el control del proceso. En una red real,* `gdbserver` *se usa **por SSH** (o por túnel), nunca expuesto a la red abierta.*

Práctica de la Parte 7

Ejercicio 20.17 — Tu `.gdbinit`

Conceptos · `.gdbinit`, `define` / `end`, `set history`, `set confirm`, alias, `set print`, `-nx`

Contexto · Los mismos tres comandos repetidos 200 veces por día son una pérdida de tiempo. Un `.gdbinit` bien armado te los da de un tiro.

Consigna · Creá tu archivo `.gdbinit` (con `~/.gdbinit` como destino) que contenga:

1. **Ajustes de presentación**: historial persistente, `print pretty`, `print elements 0`, `pagination off`.
2. **Tres comandos propios** con `define` / `end`, que sean de **tu** uso real:
   - uno que imprima las variables locales con formato (`info locals` + `p` de cada una);
   - uno que muestre la pila completa con los valores de los argumentos (`bt full`);
   - uno que muestre el estado de la memoria alrededor de un puntero que le pases.
3. **Un alias** con `alias` (no con `define`) para el comando que uses más.

Después probá que:

- `gdb ./demo` carga tu configuración;
- `gdb -nx ./demo` **no** la carga (y eso está bien, es lo que querés para depurar con settings limpios);
- un `.gdbinit` con un error de sintaxis no te deja sin prompt (probalo, y reparalo).

Entrada · Ninguna. **Salida** · El contenido final de tu `.gdbinit`, documentado.

Escenario de prueba

```
$ gdb -nx ./demo
(gdb) show print pretty
Pretty printing is off.
```

Identificadores sugeridos · `.gdbinit`, `locals`, `btfull`, `around`
Restricciones/pistas

- `alias` crea un alias de un **comando existente**; `define` crea un **comando nuevo** con varias líneas.
- Dentro de un `define`, `$arg0`, `$arg1`, … son los argumentos que le pasaste.
- `~/.gdbinit` **no** se lee si GDB se arranca con `-nx`. Es tu escape hatch para los días que necesitás settings limpios.
- En Windows con MSYS2, el home es el de MSYS2 (el que te muestra `echo $HOME` en la terminal de MSYS2), no el de Windows.

Criterios de aceptación

- [ ] El `.gdbinit` tiene los cuatro bloques pedidos (presentación, 3 `define`, 1 `alias`).
- [ ] Los tres comandos propios **se ejecutan sin error** y se verificó su salida.
- [ ] Se comprobó la diferencia entre `gdb ./demo` y `gdb -nx ./demo`.
- [ ] El archivo tiene **comentarios** que explican qué hace cada bloque.
- [ ] Hay una copia de seguridad (`~/.gdbinit.bak`).

Ejercicio 20.18 — Hilos: quién se está trabando

Conceptos · `info threads`, `thread N`, `thread apply all bt`, `set scheduler-locking`, deadlock, condición de carrera, TSan

Contexto · Un programa con hilos que "se cuelga" casi nunca está usando el 100 % de la CPU: está **esperando**. Y esperar, en un programa concurrente, casi siempre significa un **deadlock** (dos hilos se esperan mutuamente) o una condición de carrera mal resuelta.

Consigna · Escribí un programa `hilos.c` con **dos** hilos y una variable compartida:

1. Cada hilo incrementa un contador compartido en un bucle de 100.000 veces.
2. Cada hilo imprime su progreso cada 10.000 iteraciones.
3. El `main` espera a que terminen (con `pthread_join`) e imprime el total.

Después:

1. Corré el programa **sin** depurador: ¿el total es correcto? ¿ siempre?
2. En GDB, poné un breakpoint en la línea del incremento y usá `thread apply all bt` para ver los dos hilos.
3. Usá `set scheduler-locking on` y observá que solo avanza el hilo actual.
4. Cambiá a `thread 2` y verificá que estás en **otro** frame, con **otras** variables locales.
5. **Variante**: modificá el programa para que los dos hilos se **esperen mutuamente** (deadlock) y diagnosticá dónde está cada uno con `thread apply all bt` .

Entrada · Ninguna. **Salida** · El diagnóstico del deadlock (qué espera cada hilo y por qué nunca se resuelve).

Escenario de prueba

```
(gdb) thread apply all bt
Thread 1 (LWP 12345):
#0  nanosleep () at ../sysdeps/nptl/nptl/pthread_cancel.c:36
#1  esperar_al_otro () at hilos.c:22
Thread 2 (LWP 12346):
#0  nanosleep () at ../sysdeps/nptl/nptl/pthread_cancel.c:36
#1  esperar_al_otro () at hilos.c:24
```

Identificadores sugeridos · `hilos.c` , `hilo1` , `hilo2` , `contador` , `compartido` , `esperar_al_otro` **Restricciones/pistas**

- La **carrera de datos** (lectura y escritura simultáneas sin sincronización) no siempre se manifiesta: por eso el total "a veces" está mal. Para verla de forma determinista, compilá con `-fsanitize=thread` (ThreadSanitizer), que **sí** la detecta siempre.
- `thread apply all bt` **congela** el estado de todos los hilos en el mismo instante: por eso sirve para ver un deadlock.
- `set scheduler-locking on` hace que los otros hilos no avancen mientras vos hacés `next` .

Criterios de aceptación

- ☐ `info threads` distingue el hilo actual (con `*`) de los demás.
- ☐ `thread apply all bt` muestra el estado de **los dos** hilos.
- ☐ `set scheduler-locking on` se comprobó (los otros hilos no avanzan).
- ☐ El deadlock se diagnosticó identificando **qué espera cada hilo**.
- ☐ El rol de ThreadSanitizer en la detección de carreras está explicado.

Ejercicio 20.19 — Acoplarte a un proceso vivo

Conceptos · `attach`, `gdb -p`, `detach`, `ptrace_scope`, `set startup-with-shell off`, `set disable-randomization on`, `info proc mappings`

Contexto · El programa ya está corriendo y no lo podés modificar: no tenés los fuentes a mano o el binario es de otro servidor. `attach` te deja entrar y mirar.

Consigna · Escribí un programa `lento.c` que:

1. entre en un bucle que duerme un poco y va incrementando un contador;
2. imprima el contador cada medio segundo, indefinidamente.

Y hacé esto:

1. Lanzalo en segundo plano (`./lento &`) y anotá su PID.
2. Conectate con `gdb -p <PID>`.
3. Verificá que el proceso **se frenó** (mirá con `ps` que su estado cambió).
4. Inspeccioná: `bt`, `info threads`, `info locals`, `info proc mappings`.
5. **Desconectate** con `detach` y verificá que el proceso **sigue vivo**.
6. Conectate de nuevo y probá `set {int}<direccion> = <valor>` para cambiar el contador en vivo.
7. Salí **sin matar** el proceso.

Entrada · Ninguna. **Salida** · La evidencia de cada paso, y la confirmación de que el proceso sobrevivió.

Escenario de prueba

```
$ ./lento &
[1] 12345
$ gdb -p 12345
...
(gdb) info locals
contador = 37

(gdb) detach
Detaching from program: 12345, Remote target
[Inferior 1 (process 12345) detached]
$ ps -p 12345
  PID TTY      STAT   TIME COMMAND
12345 pts/0    S      0:00.04 ./lento
```

Identificadores sugeridos · `lento.c` , `contador` , `pid` Restricciones/pistas

- `attach` **frena** el proceso. Si te parece que el servidor "se colgó" justo después de tu `attach", es que lo frenaste vos.
- `detach` lo suelta y sigue. `quit` con "sí" a la pregunta de matar **lo mata**.
- Si `attach` falla con *"operation not permitted"*, es el `ptrace_scope` del sistema: usá `sudo` , o pedile a quien administra la máquina que lo ajuste.
- La aleatorización de direcciones (ASLR) puede hacer que GDB muestre valores **raros**: `set disable-randomization on` (antes de conectar) lo evita.

Criterios de aceptación

- ☐ El proceso se conectó y se verificó que **se frenó** al conectar.
- ☐ `bt` , `info threads` , `info locals` e `info proc mappings` se usaron y su salida está capturada.
- ☐ Tras el `detach` , el proceso **sigue corriendo** (verificado con `ps`).
- ☐ La modificación de memoria en vivo se verificó **viendo el cambio en la salida del programa**.
- ☐ El proceso **no** fue matado al final.

Ejercicio 20.20 — Automatización: un script que trabaja solo

Conceptos · `-batch`, `-x`, `-ex`, `set confirm off`, `set logging`, `define`, `python`, automatización

Contexto · Si tenés que repetir la misma secuencia de diagnóstico 50 veces (por ejemplo, en 50 binarios de una tanda de pruebas), no la vas a repetir a mano. GDB se automatiza.

Consigna · Escribí un script `reporte.gdb` que, sobre un programa cualquiera:

1. ponga un breakpoint en `main`;
2. ejecute el programa;
3. imprima un separador y el **backtrace completo**;
4. imprima las **variables locales**;
5. imprima el **valor de los argumentos** de `main`;
6. genere un informe con `set logging` en un archivo `reporte.txt`;
7. termine solo (sin pedir confirmación).

Y ejecutalo **de las dos formas**:

```
$ gdb -batch -x reporte.gdb ./demo
$ gdb -batch -ex "break main" -ex run -ex "bt full" ./demo
```

Entrada · Ninguna. **Salida** · El archivo `reporte.txt` generado.

Escenario de prueba

```
$ gdb -batch -x reporte.gdb ./demo
Breakpoint 1 at 0x1129: file demo.c, line 8.
Breakpoint 1, main () at demo.c:8
--- INFORME ---
#0  main () at demo.c:8
argc = 1
argv = 0x7fffffffe0c8
contador = 0
$ cat reporte.txt
```

Identificadores sugeridos · `reporte.gdb`, `reporte.txt`, `informe`
Restricciones/pistas

- Sin `set confirm off`, GDB **pregunta** antes de salir en modo batch y la ejecución se queda esperando una respuesta.
- `set logging on` **debe** ir **después** de `set logging file`, y `set logging off` al final, para que el archivo no quede abierto.
- `bt full` muestra los argumentos **de todos** los frames, no solo del actual.
- Con `python` podés ir más lejos: bucles, listas, formateo. Si tu versión de GDB no tiene Python (algunas compilaciones minimalistas), seguí con los comandos nativos.

Criterios de aceptación

- [ ] El script corre **sin interacción** y produce el informe completo.
- [ ] `reporte.txt` contiene el backtrace, las locales y los argumentos.
- [ ] Las dos formas de invocación funcionan y se comparan.
- [ ] `set confirm off` y `set logging on/off` están en el script, con su razón.
- [ ] El script es **reutilizable**: funciona con un programa distinto sin cambiarse.

PARTE 8 — Taller práctico integrador

Acá juntamos todo. No es un ejercicio más: es un **caso**, con su método.

8.1 El caso

Tenés un programa que **anda en las pruebas** y **falla en producción**. El síntoma es triple:

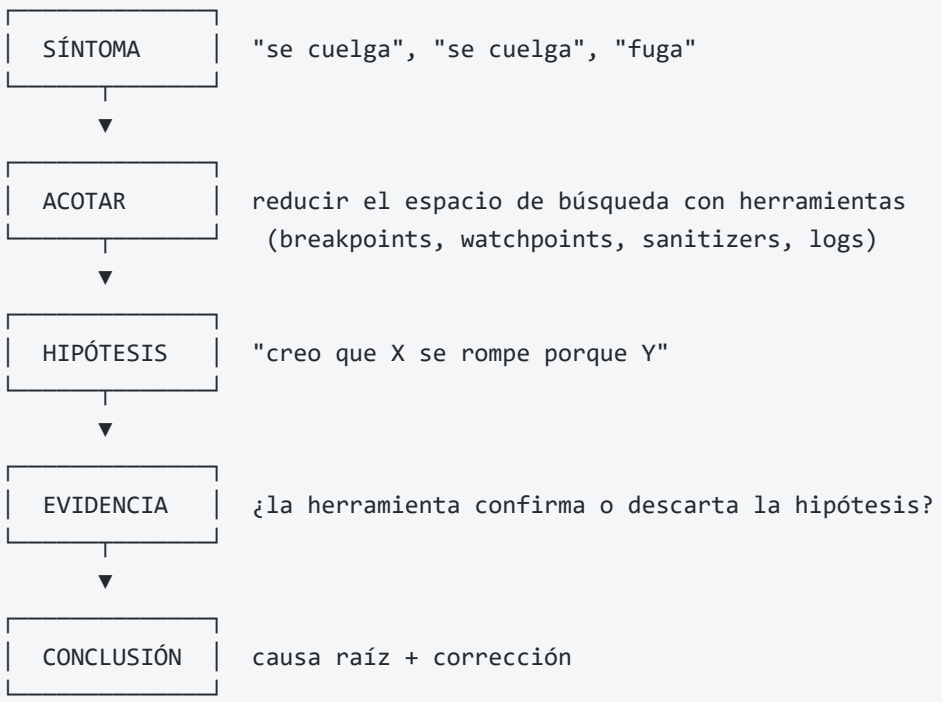
1. **Consume memoria sin parar** (el monitor de memoria del servidor se quejó).
2. **A veces se cuelga con** `Segmentation fault` (pero solo en la versión de producción).
3. **A veces no termina**: el proceso queda vivo, sin hacer nada.

El programa es un gestor de inventario: guarda una lista de artículos, cada artículo con su nombre, cantidad y precio. Tiene un `main` que arma la lista, la filtra y muestra un resumen.

Y acá viene la parte importante: **no te vamos a dar el diagnóstico**. Te vamos a dar el código, el síntoma y el método. El trabajo es tuyo.

8.2 El método: hipótesis, evidencia, conclusión

Depurar no es probar comandos al azar. Es un método:



Y hay una regla que hace todo el trabajo:

> ***Cada paso tiene que reducir la incertidumbre.*** *Si un experimento no te deja más seguro de dónde está el problema (aunque te descarte una hipótesis), está bien hecho. Si te deja igual, el experimento estaba mal planteado.*

8.3 Las cinco herramientas del taller

Herramienta	Qué te da	Cuándo la usás
ASan / LSan	Archivo y línea exactos de un error de memoria	Sospechás de memoria
Watchpoint	El instante exacto en que un valor cambia	Sospechás corrupción de datos
Breakpoint condicional	Filtrar las paradas para no ahogarte en el ruido	El bucle tiene muchas iteraciones
`bt` + **frames**	La historia de la llamada	Sospechás de un `Segfault`
`thread apply all bt`	El estado de todos los hilos	Sospechás de un cuelgue

8.4 Qué hay que entregar

Un **informe de incidente**, con esta estructura:

```
1. SÍNTOMA
   Qué se observó, cómo se reproduce, con qué frecuencia.

2. DIAGNÓSTICO
   La causa raíz, con la evidencia que la sostiene.

3. EVIDENCIA
   El comando exacto de GDB (o sanitizer) y su salida relevante.
   Un fragmento por cada afirmación del diagnóstico.

4. CORRECCIÓN
   Qué se cambió y por qué esa corrección ataca la causa y no el síntoma.

5. VERIFICACIÓN
   Cómo se comprobó que el problema se resolvió (el programa ya no se cuelga,
   ASan no reporta fugas, el bucle termina, ...).
```

> ***Por qué insistimos en el informe.*** *Porque en el trabajo real* ***nadie te va a preguntar qué comandos usaste****: te van a preguntar qué encontraste y por qué estás seguro. El informe es la forma de comunicar la evidencia. Y escribirlo es lo que convierte "toqué unos comandos" en "diagnostiqué un bug".*

Práctica de la Parte 8

Ejercicio 20.21 — (JEFE) El taller: fuga, `Segfault` y bucle infinito

Conceptos · **todo el capítulo**: ASan/LSan, watchpoints, breakpoints condicionales, `bt`, frames, `x`, core dumps, `thread apply all bt`, método de diagnóstico

Contexto · Un gestor de inventarios que, según el reporte de producción, "consume memoria sin parar, a veces se cuelga y a veces no termina". El código está abajo. Los errores **están** ahí: tu trabajo es encontrarlos, probarlos y arreglarlos.

Consigna · Diagnostica **los tres** problemas del programa y entregá el informe de incidente con la estructura de 8.4.

El programa (el material del ejercicio):

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

typedef struct {
    char  nombre[32];
    int   cantidad;
    float precio;
} Articulo;

static Articulo *inventario = NULL;
static int       total_articulos = 0;

/* Registra un articulo en el inventario. */
static void agregar(const char *nombre, int cantidad, float precio) {
    /* Copia del nombre para el registro de auditoria. */
    char *registro = malloc(strlen(nombre) + 1);
    if (registro == NULL) {
        exit(1);
    }
    strcpy(registro, nombre);

    Articulo **tmp = realloc(inventario, (size_t)(total_articulos + 1) *
sizeof(Articulo));
    if (tmp == NULL) {
        exit(1);
    }
    inventario = tmp;

    strncpy(inventario[total_articulos].nombre, nombre,
            sizeof(inventario[total_articulos].nombre));
    inventario[total_articulos].cantidad = cantidad;
    inventario[total_articulos].precio = precio;
    total_articulos++;

    printf("[agregado] %s\n", registro);
}

/* Devuelve una copia de los articulos; con solo_con_stock, solo los que tienen stock. */
static Articulo *copiar_filtrados(int solo_con_stock) {
    int n = 0;
    for (int i = 0; i < total_articulos; i++) {
        if (solo_con_stock && inventario[i].cantidad == 0) {
            continue;
        }
        n++;
    }
```

```
    Articulo *copia = malloc((size_t)n * sizeof(Articulo));
    if (copia == NULL) {
        exit(1);
    }

    int k = 0;
    for (int i = 0; i < total_articulos; i++) {
        if (solo_con_stock && inventario[i].cantidad == 0) {
            continue;
        }
        copia[k] = inventario[i];
        k++;
    }
    return copia;
}

/* Avisa por pantalla los articulos agotados. */
static void avisar_agotados(const Articulo *arts, int n) {
    int i = 0;
    while (i < n) {
        printf("   revisando: %s\n", arts[i].nombre);
        if (arts[i].cantidad == 0) {
            printf("   agotado: %s\n", arts[i].nombre);
        }
        n++;
    }
}

/* Imprime la lista de articulos. */
static void imprimir(const Articulo *arts, int n) {
    for (int i = 0; i < n; i++) {
        printf("%-16s %5d %8.2f\n", arts[i].nombre, arts[i].cantidad, arts[i].precio);
    }
}

int main(void) {
    agregar("Tornillo M8", 120, 1.5f);
    agregar("Tuerca M8", 300, 0.8f);
    agregar("Tornillo M12", 40, 3.1f);
    agregar("Arandela", 0, 0.2f);

    Articulo *con_stock = copiar_filtrados(1);
    Articulo *todos     = copiar_filtrados(0);

    char resumen[64];
    snprintf(resumen, sizeof(resumen), "Inventario: %d articulos", total_articulos);
    printf("%s\n", resumen);

    avisar_agotados(todos, total_articulos);
```

```
    imprimir(con_stock, total_articulos);
    imprimir(todos, total_articulos);

    free(con_stock);
    free(todos);
    free(inventario);
    return 0;
}
```

Entrada · Ninguna: el programa se ejecuta solo. **Salida** · El informe de incidente con los **tres** diagnósticos, sus evidencias y las correcciones aplicadas.

Escenario de prueba

```
$ gcc -g3 -O0 -Wall -Wextra -o inventario inventario.c
$ ./inventario
[agregado] Tornillo M8
[agregado] Tuerca M8
[agregado] Tornillo M12
[agregado] Arandela
Inventario: 4 articulos
   revisando: Tornillo M8
   revisando: Tornillo M8
   revisando: Tornillo M8
   ... (no termina nunca)

$ gcc -g3 -O0 -fsanitize=address -o inventario_asan inventario.c
$ ./inventario_asan
=================================================================
==4242==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x602000000010
READ of size 40 at 0x602000000010 thread T0
    #0 0x4012f7 in imprimir /home/alumno/inventario.c:78
    #1 0x401305 in main    /home/alumno/inventario.c:96
...
```

> ***Ojo con el orden.*** *Los tres problemas* ***no se ven todos a la vez****: el cuelgue tapa a los otros dos. El camino natural es primero destrabar el cuelgue, después recién aparecen los síntomas de memoria.*

> ***No te damos la lista de errores.*** *Eso es lo que el capítulo te enseñó a hacer:* ***acotar*** *con herramientas hasta que la evidencia te imponga la respuesta. Usá el método de 8.2 y el informe de 8.4.*

Identificadores sugeridos · `inventario.c`, `agregar`, `copiar_filtrados`, `inventario`, `total_articulos` **Restricciones/pistas**

- El `-Wall -Wextra` del comando de compilación **avisa** de cosas: leé los avisos con atención, son pistas gratis.
- El síntoma de "consume memoria sin parar" y el de "se cuelga" pueden venir del **mismo** lugar o de lugares distintos. Investigá cada uno por separado y **anotá** de dónde sale cada evidencia.
- Para el bucle infinito: `Ctrl+C` interrumpe el programa, y GDB te muestra **dónde** estabas. Ese es tu punto de partida.
- Para el `Segfault`: activá los core dumps y analizá el post-mortem, o usá ASan.
- El reporte de fugas de LSan aparece **cuando el programa termina**. Si el programa se cuelga, no hay salida y no hay reporte: arreglá el cuelgue primero.
- La fuga se repite **por cada artículo registrado**. Por eso el reporte de producción dice "consume memoria sin parar": acá solo se registran cuatro, así que el número es chico, pero la cuenta es la misma. Fijate **en qué** se repite, no solo cuánto suma.
- Cuando encontrás el bug, **no te conformes con el síntoma**: preguntate si el programa puede fallar **por otra razón** con los mismos datos. Buscá el error **de clase**, no el de instancia.

Criterios de aceptación

- ☐ Los **tres** problemas están diagnosticados, cada uno con **evidencia** citada (el comando exacto y su salida).
- ☐ El informe sigue la estructura de 8.4 (síntoma → diagnóstico → evidencia → corrección → verificación).
- ☐ **Cada** corrección está **verificada** con una ejecución posterior (el programa ya no se cuelga, ASan no reporta fugas, el bucle termina).
- ☐ El informe explica **por qué** la memoria crecía con el uso: qué hace que la fuga se repita y cuántas veces ocurre.
- ☐ El código corregido compila **sin avisos** con `-Wall -Wextra`, o cada aviso restante está **justificado** en el informe.
- ☐ Se identificó al menos **un** riesgo adicional que el enunciado no pedía (por ejemplo, un error de la misma clase en otro lugar del código), o se justificó por qué no lo hay.
- ☐ El diagnóstico del bucle infinito incluye **dónde** estaba el programa cuando lo interrumpiste.

20.22 Errores comunes

- **Depurar con `-O2`.** Las variables desaparecen (`<optimized out>`), el código se reordena y `next` salta. Depurá con `-O0 -g`.
- **Olvidar `-g` por completo.** GDB no tiene ni la tabla de líneas ni las variables locales: solo podés parar en nombres de función.
- **Confundir `p` con `*p`.** `p p` muestra la **dirección**; `p *p` muestra el **valor**. La mitad de los `Segfault` de principiante son esto.
- **`break` en una línea que no es ejecutable.** Comments, `}` y declaraciones de función no admiten breakpoints: GDB te ofrece moverlo.
- **Ignorar la ventana equivocada.** Si escribís un comando y no pasa nada, es que el foco no está en `cmd`. `tui focus cmd`.
- **Apilar 20 breakpoints y no saber cuál disparó.** `info breakpoints` con el campo `What` es tu mapa. Usá `commands` con `silent` para que solo se anuncie el que importa.
- **No usar `set print elements 0`.** Los arreglos grandes se muestran truncados a 200 elementos, y el `...` hace creer que el arreglo es más corto.
- **`p` de un `char[]` sin `'\0'`.** GDB imprime basura y parece un bug del depurador: es un bug del programa.
- **Confundir el frame actual.** Después de un `up`, `print total` muestra **otra** variable `total`. Volvé a `frame 0` cuando dudes.
- **Olvidar `detach` después de un `attach`.** Salís de GDB y mataste el proceso que estabas investigando.
- **`ulimit -c` en 0 y esperar un core.** El mensaje "core dumped" aparece igual, pero no se genera archivo.
- **Confiar en `core` cuando el sistema usa `systemd-coredump`.** No hay archivo en tu carpeta: usá `coredumpctl`.
- **`abort_on_error` sin `ASAN_OPTIONS`.** El sanitizer imprime el reporte y **termina**: para pararlo en GDB hay que pedirlo explícitamente.
- **Parchear memoria y creer que arreglaste el bug.** El bug sigue en el código; el próximo `run` lo vuelve a traer.
- **`step` en vez de `next` y perderse en la biblioteca estándar.** `next` no entra a las funciones; `step` sí. Si te perdiste dentro de `printf`, usá `finish`.
- **Correr `run` mil veces.** Si el bug es intermitente, el problema no es el debugger: es que no tenés una **hipótesis**. Vuelvé a 8.2.

- **No guardar la evidencia.** `set logging on` te salva la sesión entera en un archivo, y ese archivo es lo que entregás.

20.23 Cierre

Si tuvieras que quedarte con **una sola idea**, que sea esta: **un depurador no te da la respuesta, te da la evidencia**. Tu trabajo es mirar la evidencia y razonar.

- Entendiste que un binario **sin información de depuración** es un libro en clave: los símbolos (`DWARF`) son la llave, y `-g3 -O0` es la cerradura. Sabés por qué `-O2` rompe la depuración y por qué hay que **depurar y medir por separado**.
- Conocés la **TUI**: las ventanas (`src` , `asm` , `regs` , `stack` , `threads` , `cmd`), los layouts (incluido los que vos creás con `tui new-layout`), el foco y el redibujado.
- Manejás el **control del flujo** (`next` , `step` , `finish` , `continue` , `until`) y la **inspección** (`print` , `ptype` , `info locals` , `display`), y sabés cuándo usar cada uno.
- Ves **direcciones y desreferencias** (`&` , `*`), **arreglos** (incluido el arreglo artificial `p *arr@10`), **structs** (`p s` , `p s->campo`) y sabés que `p x` y `p *p` son cosas distintas.
- Usás **breakpoints condicionales**, `ignore` y `commands` para filtrar el ruido, y **watchpoints** para encontrar el instante exacto en que un valor se corrompe.
- Recorrés la **pila de llamadas** (`bt` , `frame` , `up` , `down`) y entendés por qué la misma variable local tiene distinto valor en cada frame.
- Diagnosticás **errores de memoria**: activás core dumps, analizás el post-mortem con `bt` + `info registers` + `x/i $pc` , auditás memoria cruda con `x/NFU` , y complementás con **sanitizers** (ASan, UBSan, TSan) y Valgrind.
- Bajaste al **ensamblador**: `disassemble` , `stepi` , `nexti` , la convención de llamada, y el **parcheo en caliente** para probar hipótesis.
- **Automatizaste**: `.gdbinit` , scripts con `-batch` / `-x` / `-ex` , `logging` , hilos con `thread apply all bt` , `fork` con `follow-fork-mode` , `attach` a procesos vivos y depuración remota con `gdbserver` .

Resumen de la caja de herramientas.

Quiero…	Comando	Idea en una frase
Cargar el programa	`gdb ./demo`	GDB no lo ejecuta hasta que vos lo decís
Correr hasta una parada	`run`, `start`, `starti`	`start` frena adentro de `main`
Parar en una línea	`break demo.c:42`	El breakpoint frena **antes** de esa instrucción
Parar una sola vez	`tbreak demo.c:42`	Se borra solo al dispararse
Parar solo a veces	`break ... if x > 5`, `ignore N 500`	Filtrar las paradas
Ver los breakpoints	`info breakpoints`	El campo `What` es tu mapa
Automatizar un breakpoint	`commands` … `end`	Instrumentar sin tocar el código
Avanzar una línea	`next` / `step`	`next` no entra a las funciones; `step` sí
Salir de la función	`finish`	Vuelve a quien llamó
Ver un valor	`print x`, `p/x x`, `p/c x`	Formatos para cada tipo
Ver el tipo	`ptype x`	La definición completa
Ver todas las locales	`info locals`, `info args`	El volcado del frame
Ver siempre un valor	`display x`	Se reevalúa en cada parada
Frenar si cambia	`watch x`	Hardware watchpoint sobre el valor
Ver una dirección	`p &x`, `p *p`	Dirección y valor son cosas distintas
Ver un arreglo	`p *arr@10`	El arreglo artificial
Ver una struct	`p s`, `p s→campo`, `ptype /o struct S`	Hasta los offsets
Ver la pila	`bt`, `frame N`, `up`, `down`	La historia de la llamada
Mirar la memoria cruda	`x/4xw $sp`, `x/s buf`, `x/i $pc`	Formato `N` elementos, `F` formato, `U` tamaño
Analizar un crash	`gdb ./demo core`	Post-mortem: `bt`, `info registers`, `x/i $pc`

Detectar errores de memoria	`-fsanitize=address`	Te da archivo y línea exactos
Ver el ensamblador	`disassemble /s main`	La instrucción al lado del C
Avanzar una instrucción	`stepi`, `nexti`	Un paso de máquina
Cambiar una variable	`set var x = 42`	Probar hipótesis
Escribir en memoria	`set {int}0x... = 42`	Parcheo en caliente
Ver los hilos	`info threads`, `thread apply all bt`	El estado de todos a la vez
Congelar los otros hilos	`set scheduler-locking on`	Depuración determinista
Conectarse a un proceso vivo	`gdb -p PID`, `detach`	Entrar y salir sin matar
Depurar remoto	`gdbserver :2345`, `target remote`	Cliente y servidor separados
Automatizar	`-batch -x script.gdb`, `set logging on`	El script trabaja solo

El puente a lo que sigue. *Ya podés depurar cualquier programa de C: mirar variables, recorrer la pila, encontrar la línea del bug, analizar un crash, automatizar el diagnóstico y pegarte a un proceso vivo. Eso es* ***una competencia profesional****, no un truco.*

Y el próximo paso natural es combinarla con lo que ya sabés: ***sanitizers*** *para los errores de memoria (este capítulo),* ***el capítulo*** `09` *para entender la memoria dinámica que estás depurando,* `15` *para los archivos y* `12` *para las arenas. Un gestor de inventarios con fugas, con arenas, con archivos y con depurador es un programa* ***de verdad*** *— y a esta altura ya tenés las herramientas para construirlo sin adivinar.*