



LUCAS EGA
ACADEMY



Linux y Shell

{ }

> { _

&&



```
user@linux:~$ ls -la /bin
dr~yr-xr-x 2 root root 4096 Jan 12 bin
user@linux:~$ ./setup.sh --install
Installing packages... [OK]
Automation ready.
```

/ \$

LUCAS EGA ACADEMY

TERMINAL • BASH • AUTOMATIZACIÓN

Curso Completo de Linux y Shell Scripting

Autor: Esteban

lucasegaacademy.com

*Guía completa con explicaciones extensas por unidad,
prácticas y examen final.*

CRÉDITOS

Curso Completo de Linux y Shell Scripting

© 2026 Esteban Gutiérrez

Colibrí Editorial EGA

Hecho en México

Primera edición digital · septiembre de 2026

Sin ISBN

PRÓLOGO

Bienvenido a "Curso Completo de Linux y Shell Scripting", el libro oficial del curso gratuito de Lucas EGA Academy. Si estás leyendo estas líneas es porque decidiste dominar la herramienta que está detrás de casi todo lo que existe en tecnología: la terminal de Linux. Y déjame decirte algo desde el inicio: tomaste una gran decisión.

Linux está en todas partes aunque no lo veas. Está en los servidores que alojan las páginas web que visitas, en los teléfonos Android, en los sistemas de los bancos, en la nube donde viven tus fotos, en las supercomputadoras que predicen el clima y hasta en la Estación Espacial Internacional. Y la forma de hablarle a Linux, de darle órdenes y de automatizarlo, es la terminal con el shell. Quien domina la terminal domina la computadora de verdad: hace en segundos lo que a otros les toma horas de clics, y automatiza tareas repetitivas con scripts que trabajan solos mientras duermes.

Este libro nació del curso en video de 30 clases de Lucas EGA Academy, que puedes tomar gratis en lucasegaacademy.com. Pero no es una simple transcripción. Aquí cada tema se explica con mucha más profundidad: con analogías de la vida real para que los conceptos se te queden grabados, con los errores más comunes que cometen los principiantes (y cómo evitarlos), con buenas prácticas profesionales, con ejemplos de comandos y scripts comentados línea por línea que puedes ejecutar tal cual, y con prácticas con soluciones explicadas para que compruebes que de verdad dominas cada tema.

Para quién es este libro

Este libro es para ti si estás en alguno de estos casos:

- Nunca has abierto una terminal en tu vida y quieres empezar desde cero, sin suponer nada.
- Usas Linux de vez en cuando pero solo copias comandos de internet sin entender qué hacen.
- Eres estudiante de programación y te das cuenta de que los trabajos piden "manejo de Linux" y "scripting".
- Trabajas en soporte, redes o desarrollo y quieres automatizar tus tareas repetitivas con scripts bash.
- Quieres entender cómo funcionan por dentro los servidores donde viven las páginas web.
- Te interesa la ciberseguridad y sabes que todo hacker ético vive en la terminal.

No importa tu edad ni a qué te dediques. La única condición es que practiques: leer sobre la terminal sin abrir una terminal es como aprender a manejar leyendo un manual sin subirte al carro. En cada unidad hay ejercicios con soluciones, úsalos. Todo lo que aquí aparece lo puedes probar en tu propia computadora.

Cómo está organizado este libro

El libro está dividido en 10 unidades, y cada unidad agrupa 3 clases del curso en video. En total cubrimos las 30 clases completas:

- UNIDAD 1: La terminal y tus primeros comandos. Qué es Linux, qué es el shell, cómo abrir la terminal y moverte por el sistema de archivos con `pwd`, `ls` y `cd`.
- UNIDAD 2: Archivos y directorios. Crear, copiar, mover, renombrar y borrar archivos; ver su contenido y buscar archivos perdidos.
- UNIDAD 3: Permisos y propiedad. El sistema de permisos `rwx`, `chmod`, `chown`, usuarios, grupos y `sudo`.
- UNIDAD 4: Redirección, tuberías y filtros. Entrada y salida estándar, pipes, y los comandos de texto `sort`, `uniq`, `cut`, `tr` y `wc`.
- UNIDAD 5: `grep`, `sed` y `awk`. Búsqueda con expresiones regulares, edición de texto en flujo y procesamiento de columnas.
- UNIDAD 6: Variables y tu primer script. Variables, comillas, argumentos, códigos de salida y tu primer script `bash` ejecutable.
- UNIDAD 7: Condicionales y bucles. Decisiones con `if` y `case`, repetición con `for` y `while`.
- UNIDAD 8: Funciones, depuración y buenas prácticas. Funciones reutilizables, modo estricto, `getopts` y estilo profesional.
- UNIDAD 9: Procesos, tareas programadas y redes. Control de procesos, `cron`, `SSH`, `SCP` y herramientas de red.
- UNIDAD 10: Proyecto final y graduación. Un sistema de respaldo automatizado completo, repaso integral del curso y tu graduación.

Cada unidad termina con prácticas con soluciones y un resumen de puntos clave. Al final del libro hay un examen final de 50 preguntas para que midas tu dominio real.

Cómo estudiar este libro

No leas este libro como una novela: léelo con una terminal abierta al lado. Cada vez que veas un bloque de comandos, ejecútalo. Cambia los nombres de archivo, prueba otras opciones, rompe cosas a propósito en una carpeta de prueba. El error es un maestro excelente cuando es barato: aquí equivocarte no cuesta nada y te enseña el doble.

Un consejo de oro: crea una carpeta de práctica, por ejemplo `~/practica-linux`, y haz todos los ejercicios ahí dentro. Así puedes experimentar sin miedo a borrar algo importante de tu sistema. La regla número uno del principiante en la terminal es no ejecutar comandos destructivos fuera de tu zona de práctica, y en este libro te voy a enseñar exactamente cuáles son esos comandos y por qué.

Qué necesitas para empezar

Necesitas una computadora con Linux. Si ya tienes Ubuntu, Debian, Fedora o cualquier distribución, estás listo. Si estás en Windows, puedes instalar WSL (Windows Subsystem for Linux) gratis desde la tienda de Microsoft, que te da una terminal Ubuntu real dentro de Windows. Si estás en Mac, la terminal usa un shell parecido (zsh) y casi todo lo de este libro te sirve igual. No necesitas una computadora potente: Linux corre bien hasta en máquinas viejas.

También necesitas un editor de texto para escribir tus scripts. Puedes usar nano, que viene instalado y se maneja con el teclado, o Visual Studio Code si prefieres algo visual. Con eso basta.

Lo que vas a lograr

Al terminar este libro vas a moverte por la terminal con la soltura de un profesional: navegar, manipular archivos, controlar permisos, procesar texto como un experto con grep, sed y awk, y lo más valioso de todo: vas a escribir scripts bash que automaticen tareas reales. Tu proyecto final será un sistema de respaldo automatizado con rotación de copias, registro de actividades y ejecución programada con cron. Eso no es un ejercicio de juguete: es una herramienta que podrías usar en un trabajo de verdad.

Además, este curso te abre la puerta a los demás cursos de la academia: el de Hacking Ético, el de Networking y el de Python asumen que sabes moverte en la terminal. Cuando los tomes, ya no te vas a perder.

Unas palabras del autor

Aprender la terminal intimida al principio. La pantalla negra, el cursor parpadeando, ningún botón a la vista: parece que la computadora te está retando. Pero te prometo algo: después de las primeras tres unidades vas a sentir un clic en la cabeza, y de ahí en adelante cada comando nuevo se va a sentir como un superpoder. La terminal no muerde. Solo habla otro idioma, y este libro es tu traductor.

Nos vemos en la Unidad 1. Abre tu terminal. Empecemos.

UNIDAD 1: La terminal y tus primeros comandos

Bienvenido al punto donde todo empieza. Antes de que puedas automatizar respaldos, procesar texto como un profesional o administrar servidores, necesitas entender la herramienta base de todo lo demás: la terminal. Esta primera unidad no es un trámite que debas saltarte; es la cimentación. Si la cimentación queda chueca, todo lo que construyas encima se va a tambalear. Si queda firme, cada unidad que sigue se va a sentir natural.

En esta unidad vas a aprender qué es Linux y por qué gobierna el mundo de los servidores, qué es exactamente el shell y en qué se diferencia de la terminal, cómo abrir tu terminal en cualquier sistema, y tus primeros comandos para orientarte: `pwd` para saber dónde estás, `ls` para ver qué hay y `cd` para moverte. Vas a entender las rutas absolutas y relativas, el significado de `.` y `..`, y cómo pedir ayuda con `man`, `--help` y `help`. También vas a conocer dos superpoderes que te van a ahorrar miles de tecleos: el autocompletado con `Tab` y el historial de comandos.

Por qué importa tanto esta unidad: la terminal no es un tema más del curso, es el tema del curso. Todo lo demás que vas a estudiar vive dentro de la terminal. Los permisos de la Unidad 3 se cambian con comandos. El texto de las Unidades 4 y 5 se procesa con comandos. Los scripts de las Unidades 6 a 8 se escriben y se ejecutan en la terminal. Aprender a moverte primero es como aprender a caminar antes de correr. Sin este paso no hay nada posible, así de simple.

Hay otra razón de peso: la terminal premia a quien domina lo básico. Mucha gente copia comandos de internet sin entenderlos, y eso es peligroso: un comando mal entendido puede borrar archivos o exponer un servidor. El que entiende qué hace cada parte de un comando trabaja rápido y seguro. Lo básico no es lo aburrido: lo básico es lo poderoso. Cuando entiendes el porqué detrás de cada comando, la terminal deja de ser una caja negra y se convierte en tu herramienta favorita.

Cómo encaja esta unidad en el curso completo: el "Curso Completo de Linux y Shell Scripting" de Lucas EGA Academy está diseñado como una escalera. Esta unidad es el primer peldaño: abrir la terminal, orientarte y pedir ayuda. En la siguiente subirás al siguiente nivel: crear y manipular archivos y directorios. Después vendrán los permisos, la redirección, el procesamiento de texto, y finalmente la programación de scripts. Cada unidad nueva usa sin piedad lo que aprendiste en las anteriores.

Una nota sobre cómo estudiar esta unidad. No la leas como si fuera una novela: léela con la terminal abierta. Cada vez que veas un bloque de comandos, ejecútalo en tu zona de práctica. Cambia los nombres, prueba otras opciones, equivócate a propósito y observa qué pasa. El

error es un maestro excelente cuando es barato. Los ejercicios de la sección de Práctica están diseñados exactamente para eso.

También quiero que sepas algo desde ahora: no necesitas nada caro para seguir este curso. Si tienes Linux instalado, ya tienes todo. Si estás en Windows, instala WSL gratis. Si estás en Mac, abre la Terminal que ya viene incluida. La barrera de entrada es bajísima, y eso es una noticia buenísima.

Así que manos a la obra. Esta unidad es larga a propósito: cada concepto viene explicado a fondo, con analogías de la vida real, con ejemplos de comandos comentados, con errores típicos de principiante y con buenas prácticas profesionales. Tómate tu tiempo. Cuando termines, vas a moverte por el sistema de archivos desde la terminal sin miedo, y eso ya es más de lo que mucha gente que "usa Linux" puede hacer bien. Empecemos.

Explicación detallada

Qué es Linux y por qué manda en el mundo

Linux es un sistema operativo, es decir, el programa maestro que administra tu computadora: decide qué programa usa el procesador, organiza los archivos en el disco y controla quién puede hacer qué. Nació en 1991 cuando un estudiante finlandés llamado Linus Torvalds publicó el núcleo (kernel) de un sistema gratuito, y miles de programadores de todo el mundo se sumaron a mejorarlo. Hoy Linux es software libre: cualquiera puede usarlo, estudiarlo, modificarlo y compartirlo sin pagar licencias.

¿Por qué te debe importar? Porque Linux gobierna la infraestructura del mundo digital. Más del 90 por ciento de los servidores que alojan las páginas web corren Linux. Android, el sistema de los celulares, está basado en Linux. La nube de Amazon, Google y Microsoft corre sobre Linux. Las supercomputadoras más potentes del planeta usan Linux. Cuando aprendes Linux no estás aprendiendo un sistema "alternativo": estás aprendiendo el sistema donde vive el internet.

Piensa en los sistemas operativos como idiomas. Windows y macOS son como el idioma que hablas con turistas: amable, con señalamientos por todos lados. Linux es el idioma que hablan los ingenieros entre ellos: preciso, potente y sin adornos innecesarios. Este curso te enseña a hablarlo con fluidez.

Existen muchas "distribuciones" de Linux, que son versiones empaquetadas por distintas comunidades o empresas: Ubuntu, Debian, Fedora, Arch, Mint, entre otras. Todas comparten el mismo corazón (el kernel Linux) y los mismos comandos básicos que vas a aprender aquí.

Si dominas la terminal en Ubuntu, la dominas en casi cualquier distribución. En este libro usamos ejemplos de Ubuntu/Debian porque son las más populares para empezar, pero el 99 por ciento de lo que aprendas funciona en todas.

La terminal, el shell y el prompt: tres cosas distintas

Aquí hay una confusión clásica que vamos a eliminar desde el día uno. La terminal, el shell y el prompt son tres cosas diferentes que trabajan juntas.

La terminal es la ventana: el programa que abre una pantalla donde puedes escribir texto y ver respuestas. En Ubuntu se llama "Terminal" o "GNOME Terminal". En Windows con WSL es la ventana de Ubuntu. En Mac es la aplicación "Terminal". La terminal por sí sola no hace nada inteligente: es solo el mensajero, el micrófono y la bocina.

El shell es el programa que interpreta lo que escribes. Es el cerebro. Cuando escribes ``ls`` y presionas Enter, la terminal le pasa ese texto al shell, el shell lo entiende, ejecuta el programa correspondiente y le devuelve el resultado a la terminal para que lo muestre. El shell más usado en Linux se llama bash (Bourne Again SHell), y es el que usaremos en todo este curso. Hay otros (zsh, fish), pero bash es el estándar: si aprendes bash, te sirve en todas partes.

El prompt es el texto que ves antes del cursor, el que te invita a escribir. Normalmente se ve algo así:

```
usuario@computadora:~$
```

Te dice tres cosas: quién eres (usuario), en qué máquina estás (computadora) y en qué carpeta estás (~ significa tu carpeta personal). El símbolo \$ al final significa que eres un usuario normal. Si vieras # en lugar de \$, significaría que eres el superusuario (root), el que tiene poder total, y eso exige cuidado extra.

Analogía: la terminal es el teléfono, el shell es la persona con la que hablas por teléfono, y el prompt es el "bueno, ¿dígame?" con el que te contesta. Tú hablas con el shell a través de la terminal, y el prompt te avisa que está listo para escucharte.

Tu primer comando: quién eres y dónde estás

Vamos a romper el hielo. Abre tu terminal y escribe esto, luego presiona Enter:

```
whoami
```

La computadora te responde con tu nombre de usuario. Acabas de darle tu primera orden al shell y él te obedeció. Felicidades: ya usaste Linux.

Ahora escribe:

```
pwd
```

pwd significa "print working directory", imprime el directorio de trabajo, o sea, la carpeta donde estás parado ahorita. Te va a responder algo como `/home/tuusuario`. Esa es tu carpeta personal, tu casa dentro del sistema. El símbolo `~` del prompt es un atajo para esa misma carpeta.

Y ahora el comando más usado de todos:

```
ls
```

ls significa "list", lista el contenido de la carpeta donde estás: archivos y subcarpetas. Si tu carpeta personal está recién creada, quizá veas poco o nada. No importa: lo importante es que ya sabes pedirle a la computadora que te muestre qué hay.

Observa el ritual: escribes un comando, presionas Enter, el shell lo ejecuta y te muestra el resultado. Ese ciclo de escribir, Enter y leer lo vas a repetir miles de veces. En cinco minutos lo tienes dominado.

Rutas absolutas y relativas: las dos formas de decir "dónde"

Para moverte necesitas entender cómo Linux nombra los lugares. Todo en Linux cuelga de una sola raíz, representada por la diagonal `/`. De ahí salen las carpetas principales: `/home` (carpetas de usuarios), `/etc` (configuración), `/var` (datos variables), `/usr` (programas), `/tmp` (temporales). Es como un árbol genealógico invertido: la raíz arriba y todo cuelga de ella.

Hay dos formas de indicar una ubicación. La ruta absoluta empieza desde la raíz y dice la dirección completa, por ejemplo `/home/tuusuario/documentos`. Funciona igual estés donde estés, como dar la dirección completa de una casa con calle, número, colonia y ciudad.

La ruta relativa empieza desde donde estás parado ahorita. Si estás en `/home/tuusuario` y quieres ir a documentos, basta con decir `documentos`. Es como decir "la casa de al lado": solo tiene sentido si sé desde dónde la dices.

Dos nombres especiales te van a salvar la vida. El punto `.` significa "aquí mismo, la carpeta actual". Los dos puntos `..` significan "la carpeta de arriba, la que contiene a esta". Si estás en

``/home/tuusuario/documentos``, entonces ``..`` es ``/home/tuusuario`` y ``../..`` es ``/home``. Son atajos universales que funcionan en todos los comandos.

cd: moverte por el sistema

cd significa "change directory", cambiar de carpeta. Es tu medio de transporte:

```
cd documentos
```

Te lleva a la subcarpeta documentos (ruta relativa). Para volver arriba:

```
cd ..
```

Para ir a tu carpeta personal desde donde sea:

```
cd ~
```

O simplemente ``cd`` sin nada: también te lleva a tu casa. Y para ir a la raíz del sistema:

```
cd /
```

Combínalo con `pwd` para verificar dónde quedaste después de cada salto. Al principio vas a usar `pwd` a cada rato; con la práctica dejarás de necesitarlo porque el prompt ya te dice dónde estás.

Un error típico de principiante: escribir ``cd documentos`` cuando no hay ninguna carpeta llamada documentos ahí. El shell te responde "no existe el fichero o el directorio". No es un error grave: solo significa que la ruta está mal. Usa ``ls`` primero para ver qué hay, y luego ``cd`` al lugar correcto. La terminal no te regaña, solo te informa.

ls con esteroides: ver de verdad qué hay

El comando ``ls`` solo ya es útil, pero con opciones se vuelve una herramienta de inspección seria. Las opciones son letras que modifican el comportamiento y casi siempre empiezan con guion:

```
ls -l
```

La opción ``-l`` (de "long", largo) muestra el formato detallado: permisos, dueño, tamaño, fecha y nombre de cada archivo. Es la vista que vas a usar el 90 por ciento del tiempo.

```
ls -a
```

La opción `-a` (de "all", todo) muestra también los archivos ocultos, que en Linux son los que empiezan con punto, como `.bashrc`. Son archivos de configuración que normalmente no quieres ver, pero a veces necesitas.

```
ls -la
```

Puedes combinar opciones: `-la` es lo mismo que `-l -a`. Formato largo incluyendo ocultos.

```
ls -lh
```

La opción `-h` (de "human", humano) muestra los tamaños en formato legible: 4K, 2M, 1G en lugar de números crudos de bytes. `ls -lh` es probablemente el comando que más vas a escribir en tu vida con Linux. Memorízalo.

Prueba también `ls /etc` para ver el contenido de una carpeta sin moverte a ella: le pasas la ruta como argumento. Los comandos aceptan argumentos que les dicen sobre qué trabajar.

Pedir ayuda: `man`, `--help` y `help`

Nadie se memoriza todos los comandos con todas sus opciones. Los profesionales no tienen mejor memoria: tienen mejor técnica para pedir ayuda. Linux trae la ayuda incluida.

```
man ls
```

`man` abre el manual del comando: una página con descripción, opciones y ejemplos. Para salir del manual presiona la tecla `q`. Pruébalo con `man pwd` y `man cd`. Al principio el manual se ve intimidante porque está en inglés y es denso, pero con la práctica se vuelve tu mejor amigo.

La mayoría de los comandos también aceptan la opción `--help` para un resumen rápido:

```
ls --help
```

Y `bash` tiene ayuda integrada para sus comandos internos:

```
help cd
```

La diferencia: `man` es para programas externos, `help` es para órdenes internas del shell como `cd`. Si `man cd` te confunde, usa `help cd`.

Consejo profesional: antes de copiar un comando de internet, revisa su manual. Entender qué hace cada parte te protege de comandos destructivos disfrazados de soluciones. El que lee el manual nunca borra su disco por accidente.

Tus dos superpoderes: Tab y el historial

El autocompletado con la tecla Tab es el regalo más grande del shell. Escribe las primeras letras de un comando o de una ruta y presiona Tab: el shell lo completa solo. Si hay varias opciones, presiona Tab dos veces y te las muestra. Esto no solo ahorra tecleo: evita errores de escritura en nombres largos. Úsalo siempre. Siempre.

El historial guarda los comandos que ya ejecutaste. Presiona la flecha hacia arriba y verás el comando anterior; sigue presionando y viajas al pasado. ¿Necesitas repetir algo con un cambio mínimo? Flecha arriba, edítalo, Enter. Además, el comando ``history`` te muestra la lista numerada de todo lo que has escrito, y `!número`` repite el comando con ese número. Por ejemplo, `!42`` ejecuta el comando 42 del historial.

Estos dos trucos juntos multiplican tu velocidad. Un principiante escribe cada comando letra por letra; un profesional escribe tres letras, Tab, Enter. La diferencia se nota al final del día.

Ejemplos paso a paso

Ejemplo 1: exploración guiada de tu casa

Este ejemplo es tu primer paseo. Vas a abrir la terminal, descubrir quién eres, dónde estás y qué hay a tu alrededor, usando solo los comandos de esta unidad.

```
whoami
```

El shell responde con tu nombre de usuario, por ejemplo ``esteban``. Es tu identidad dentro del sistema.

```
pwd
```

Responde algo como ``/home/esteban``. Esa es tu carpeta personal: tu casa. Todo lo que crees para practicar vivirá aquí dentro.

```
ls -la
```

Ves el formato largo con ocultos. Cada línea es un archivo o carpeta. Fíjate en los que empiezan con punto: son tus archivos de configuración personal, como ``.bashrc`` (configuración de tu shell) y ``.profile``. No los borres: son los que hacen que tu terminal se comporte como se comporta.

```
ls -lh /etc | head -20
```

Miras dentro de `/etc`` sin moverte de tu casa. El ``| head -20`` lo veremos a fondo en la Unidad 4; por ahora solo observa que limita la salida a 20 líneas para no saturar tu pantalla.

```
cd /tmp
pwd
```

Te mueves a `/tmp``, la carpeta de archivos temporales, y verificas con `pwd` que llegaste: debe decir `/tmp``.

```
cd ~
pwd
```

Regresas a tu casa con ``cd ~`` y lo confirmas. Ya sabes ir y volver. Eso es navegar.

Explicación línea por línea: cada comando te dio información o te movió de lugar, y `pwd` fue tu brújula para confirmar. Este ciclo de explorar, moverte y verificar es el patrón base de todo lo que harás en la terminal. Cuando algo no salga como esperas, el primer diagnóstico siempre es: ¿dónde estoy (`pwd`) y qué hay aquí (`ls`)?

Ejemplo 2: el manual como compañero

Aquí practicas pedir ayuda como un profesional, sin depender de nadie.

```
man ls
```

Se abre el manual. Lee la primera sección (NAME y DESCRIPTION) y luego presiona ``/`` seguido de ``-h`` y Enter para buscar la opción `-h` dentro del manual. Presiona ``q`` para salir. Acabas de aprender a buscar dentro del manual: una habilidad que usarás toda la vida.

```
ls --help | head -30
```

Ves el resumen rápido de opciones, limitado a 30 líneas. Es más digerible que el manual completo cuando solo quieres recordar una opción.

```
help cd
```

Ves la ayuda interna del comando `cd`. Observa que menciona que ``cd`` sin argumentos te lleva a tu carpeta personal: confirmación oficial del truco que aprendiste.

```
man man
```

El manual del manual. Suena a broma pero es real: te explica cómo leer los manuales, incluyendo qué significan los números de sección. Es el nivel final de pedir ayuda: ayuda sobre la ayuda.

Explicación: la técnica es siempre la misma, de lo rápido a lo profundo: primero `--help`` para un recordatorio, luego ``man`` para el detalle completo, y ``help`` para los comandos internos del shell. Con estas tres herramientas nunca estás solo frente a un comando desconocido.

Ejemplo 3: velocidad con Tab e historial

Este ejemplo te entrena los dos superpoderes hasta que salgan solos.

```
cd /etc
```

Escribe ``cd /e`` y presiona Tab: el shell completa a ``/etc/``. Un tecleo en lugar de cuatro, y sin riesgo de error.

```
ls -l /etc/hostname
```

Escribe ``ls -l /etc/hos`` y presiona Tab: completa a ``hostname``. El archivo `hostname` contiene el nombre de tu computadora.

```
cat /etc/hostname
```

Muestra el contenido del archivo: el nombre de tu máquina. (cat lo veremos a fondo en la Unidad 2; por ahora solo es "muéstrame qué dice este archivo".)

Ahora presiona la flecha hacia arriba tres veces: viajas por tu historial hasta el ``cd /etc``. Presiona Enter para repetirlo. Luego escribe ``history | tail -10`` para ver tus últimos 10 comandos numerados.

Explicación: Tab completa y evita errores de dedo; el historial te evita reescribir. Juntos son la diferencia entre una terminal que se siente lenta y una que se siente como una extensión de tus manos. Práctícalos hasta que sean reflejo.

Ejemplo 4: orientarte en un sistema desconocido

Imagina que entras a un servidor que nunca habías visto. Este es el protocolo profesional de reconocimiento en 60 segundos.

```
whoami  
pwd  
ls -la
```

Quién soy, dónde estoy, qué hay aquí. Tres comandos, diez segundos, y ya tienes el mapa inicial.

```
ls /
```

Miras la raíz del sistema para reconocer la distribución: si ves carpetas típicas de Ubuntu/Debian ya sabes dónde estás parado.

```
cat /etc/os-release
```

Este archivo te dice exactamente qué distribución y versión de Linux es. En un servidor real, esto es lo primero que verifica un administrador.

```
df -h
```

Muestra el espacio libre en los discos en formato humano. En un servidor, el disco lleno es una de las causas más comunes de fallas: este comando te lo dice en un segundo.

```
free -h
```

Muestra la memoria RAM disponible. Entre `df -h` y `free -h` tienes el chequeo de salud básico de cualquier máquina Linux.

Explicación: este protocolo (identidad, ubicación, sistema, disco, memoria) es lo que hace un profesional al conectarse a una máquina nueva. No es memorización: es un hábito. En la Unidad 9 volverás a estos comandos cuando administres procesos y conexiones remotas.

Errores comunes y casos reales

Esta sección existe por una razón muy simple: los errores que vienen aquí te van a pasar sí o sí en tus primeras semanas. No es amenaza, es garantía. La diferencia entre el que se atora una hora y el que se atora cinco minutos es que el segundo ya vio el error antes y sabe exactamente qué significa. Léelos, provócalos a propósito en tu zona de práctica y aprende a salir de cada uno sin pánico.

Caso 1: "No existe el fichero o el directorio" al usar cd

Síntoma: escribes ``cd documentos``, presionas Enter y la terminal te responde con algo como ``bash: cd: documentos: No existe el fichero o el directorio``. Tú juras que la carpeta existe porque la acabas de ver.

Por qué ocurre: en el 90% de los casos es una de estas tres: 1) la escribiste con mayúsculas o minúsculas distintas al nombre real (``Documentos`` no es lo mismo que ``documentos``); 2) el nombre tiene espacios y escribiste ``cd Mis Documentos`` sin comillas, entonces el shell cree que le pasaste dos carpetas distintas; 3) no estás parado donde crees que estás, así que la ruta relativa ``documentos`` se busca en la carpeta equivocada.

Solución: primero confirma dónde estás y qué hay realmente ahí. Luego navega usando Tab para que la terminal escriba el nombre exacto por ti.

```
pwd                # confirma en qué carpeta estás parado
ls                 # mira los nombres EXACTOS, con sus mayúsculas
cd Documentos      # respeta mayúsculas y minúsculas tal cual
cd "Mis Documentos" # si tiene espacios, pon el nombre entre comillas
cd Mis<TAB>        # o deja que Tab complete el nombre exacto por ti
```

Lección profesional: cuando un ``cd`` falla, el problema casi nunca es el comando, es tu suposición sobre dónde estás o cómo se llama la carpeta. Verifica con ``pwd`` y ``ls`` antes de insistir.

Caso 2: "command not found" por escribir el comando en mayúsculas

Síntoma: escribes ``LS -la`` o ``PWD`` y la terminal responde ``LS: orden no encontrada`` o ``command not found``. En Windows eso funcionaba igual, aquí no.

Por qué ocurre: Linux distingue entre mayúsculas y minúsculas en todo: comandos, nombres de archivo, carpetas. ``LS`` y ``ls`` son dos palabras completamente distintas para el shell, y solo existe la versión en minúsculas.

Solución: escribe los comandos siempre en minúsculas. Si te vuelve a pasar, no reinstales nada ni busques en internet: simplemente corrige las mayúsculas.

```
LS -la      # ERROR: orden no encontrada
ls -la      # CORRECTO: lista el contenido en formato largo
PwD         # ERROR: orden no encontrada
pwd         # CORRECTO: muestra tu carpeta actual
```

Lección profesional: este error es el filtro más rápido para saber si alguien viene de Windows. En Linux, la tecla de mayúsculas (Bloq Mayús) activada por accidente es una fuente clásica de "nada me funciona". Si de pronto todo te marca error, revisa que no tengas el Bloq Mayús prendido.

Caso 3: `ls` no muestra nada y crees que borraste todo

Síntoma: ejecutas `ls` y la terminal no muestra nada, solo te regresa el prompt. Entrás en pánico pensando que tus archivos desaparecieron.

Por qué ocurre: casi siempre estás parado en la carpeta equivocada (una carpeta vacía, como `/tmp` recién limpiado o una carpeta nueva), o los archivos que buscas son ocultos (empiezan con punto) y `ls` solo no los muestra.

Solución: verifica tu ubicación con `pwd` y lista incluyendo ocultos con `ls -la`. El pánico se cura con información.

```
pwd         # ¿dónde estoy parado realmente?
ls          # ¿qué hay aquí sin contar ocultos?
ls -la      # ¿y contando los archivos ocultos?
cd ~        # si te perdiste, regresa a tu casa y empieza de nuevo
ls          # ahora sí deberías ver tus cosas
```

Lección profesional: "no veo nada" nunca significa "se borró todo" hasta que lo confirmes. El profesional primero verifica dónde está parado y luego busca con las opciones adecuadas. El 99% de las veces los archivos siguen ahí, solo estabas mirando en otro lado.

Caso 4: la ruta relativa te traiciona

Síntoma: escribes `cd etc` para entrar a la carpeta de configuraciones y te dice que no existe. O ejecutas un comando sobre `archivo.txt` y te dice que no lo encuentra, aunque lo acabas de crear.

Por qué ocurre: las rutas relativas parten de tu ubicación actual. Si estás en tu casa (`/home/tuusuario`), `cd etc` busca `/home/tuusuario/etc`, que no existe. La carpeta real está en la raíz: `/etc`. Es el error más común de quien aún no interioriza la diferencia entre relativo y absoluto.

Solución: usa `pwd` para saber desde dónde parte tu ruta relativa, y cuando dudes, usa la ruta absoluta completa que empieza con `/` y funciona desde donde sea.

```
pwd          # ejemplo: /home/esteban
cd etc       # ERROR: busca /home/esteban/etc, que no existe
cd /etc      # CORRECTO: la ruta absoluta siempre funciona
pwd          # confirma: /etc
ls           # mira los archivos de configuración del sistema
```

Lección profesional: las rutas absolutas son tu red de seguridad. En scripts y en documentación siempre prefiere rutas absolutas, porque las relativas dependen de dónde se pare quien las ejecute, y eso cambia.

Caso 5: quedas atrapado dentro de `man` o `less` sin saber salir

Síntoma: abres `man ls`, la pantalla se llena de texto, presionas Enter, flechas, Escape, Ctrl+C, y nada te regresa al prompt. Sientes que la terminal se trabó.

Por qué ocurre: `man` y `less` son paginadores: toman el control de la pantalla para mostrarte texto largo por partes. No están trabados, están esperando sus propias teclas. El prompt no va a volver hasta que salgas del programa con la tecla correcta.

Solución: presiona `q` (de quit) para salir. Y ya que estás dentro, aprovecha: `/texto` busca, `n` va a la siguiente coincidencia, flechas o espacio avanzan.

```
man ls       # abre el manual; la pantalla cambia de modo
/t          # busca la primera aparición de "t" en el texto
n           # salta a la siguiente coincidencia
q           # SALE del manual y regresa al prompt
less archivo-grande.txt # lo mismo aplica con less
q           # siempre q para salir
```

Lección profesional: ningún principiante se salva de este momento de pánico, y todos los profesionales lo vivieron. Si alguna vez de verdad no puedes salir, abre otra terminal y listo, pero el 99% de las veces la respuesta es simplemente `q`.

Caso 6: Tab no completa nada y crees que está descompuesto

Síntoma: escribes ``cd Doc`` y presionas Tab, pero no pasa nada. Lo presionas más fuerte, varias veces, y sigue sin completar. Piensas que el autocompletado no sirve.

Por qué ocurre: Tab solo completa cuando hay UNA única coincidencia. Si en tu carpeta hay ``Documentos``, ``Docker`` y ``Descargas``, al escribir ``Doc`` el shell no sabe cuál de las tres quieres y se queda callado. No está roto: está esperando más información.

Solución: presiona Tab DOS veces seguidas y la terminal te lista todas las opciones. Agrega más letras para desambiguar y vuelve a presionar Tab.

```
cd D<TAB>          # no pasa nada: hay varias carpetas que empiezan con D
cd D<TAB><TAB>     # muestra: Descargas/ Documentos/ Docker/
cd De<TAB>         # ahora sí completa a Descargas/
```

Lección profesional: el doble Tab es una herramienta de exploración, no solo de completado. Los profesionales lo usan para descubrir qué archivos, comandos y opciones existen sin tener que memorizarlos. Ante la duda, doble Tab.

Caso real 1: tu primer día frente a un servidor que no conoces

Te dan acceso SSH a un servidor de la empresa. No sabes qué distribución es, quién eres ahí, qué hay instalado ni cuánto espacio queda. Un profesional no empieza a "moverle", primero se orienta con el ritual de reconocimiento, todo con comandos de esta unidad:

```
whoami            # ¿quién soy en esta máquina?
pwd               # ¿en qué carpeta me dejaron parado?
ls -la           # ¿qué hay aquí? ¿hay archivos de configuración?
cat /etc/os-release # ¿qué distribución y versión de Linux es?
df -h            # ¿cuánto espacio libre tiene el disco?
free -h          # ¿cuánta memoria RAM hay disponible?
history | tail -20 # ¿qué comandos ejecutó quien estuvo aquí antes?
```

Con esos siete comandos ya sabes más del servidor que la mayoría de la gente que lo usa todos los días. Este ritual se hace en cada máquina nueva, sin excepción: primero reconoces el terreno, después actúas.

Caso real 2: reconstruir qué pasó usando el historial

Un compañero dice "ayer en la tarde el servidor se puso lento después de que moví unas cosas". Nadie anotó qué comandos ejecutó. En vez de adivinar, usas el historial como caja negra:

```
history | grep -E "rm|mv|chmod"    # ¿borró, movió o cambió permisos de algo?
history | tail -50                 # los últimos 50 comandos, en orden
```

El historial te muestra la secuencia exacta de lo que se hizo, con lo que puedes revertir cambios o al menos entender la causa. Por eso los profesionales cuidan su historial y nunca lo borran a la ligera: es la bitácora de vuelo de la terminal.

Miniproyecto guiado de la unidad

Vas a construir tu **Mapa de Exploración del Sistema**: un reporte en texto donde documentas tu máquina como lo haría un profesional al recibirla por primera vez. Usarás todo lo de la unidad: terminal, prompt, `pwd`, `ls` con opciones, `cd` con rutas absolutas y relativas, `man`, `help`, Tab y el historial. Al terminar tendrás un archivo que demuestra que ya sabes orientarte en cualquier Linux.

Paso 1: prepara tu zona de trabajo

Crea una carpeta para el proyecto dentro de tu casa y entra en ella. Verifica cada movimiento con `pwd` hasta que se vuelva reflejo.

```
cd ~                # asegúrate de partir de tu casa
pwd                 # confirma: debe mostrar /home/tuusuario
mkdir mapa-sistema  # crea la carpeta del proyecto
ls                  # verifica que se creó
cd mapa-sistema     # entra en ella
pwd                 # confirma: /home/tuusuario/mapa-sistema
```

Verificación: `pwd` debe mostrar la ruta completa terminando en `mapa-sistema`, y `ls` dentro de ella no debe mostrar nada todavía (está vacía, es normal).

Paso 2: registra tu identidad y la del sistema

Abre un archivo de notas donde irás pegando tus hallazgos. Usa el editor más simple (`nano`) y guarda con Ctrl+O, sal con Ctrl+X. Si prefieres, usa Visual Studio Code.

```
nano reporte.txt
```

Dentro del archivo escribe tu primera sección con los datos que obtengas de estos comandos (ejecútalos en otra terminal o sal de nano momentáneamente):

```
whoami          # tu nombre de usuario
pwd             # tu carpeta personal (ruta absoluta)
cat /etc/os-release # distribución de Linux (busca PRETTY_NAME)
df -h          # espacio libre del disco principal
free -h        # memoria RAM total y disponible
```

Verificación: tu `reporte.txt` debe tener al menos estos cinco datos anotados con tus palabras. Ejemplo de cómo se ve una línea bien documentada: `Usuario: esteban | Casa: /home/esteban | Sistema: Ubuntu 24.04`.

Paso 3: expedición por las carpetas clave del sistema

Vas a visitar cinco carpetas importantes del sistema usando rutas absolutas, y en cada una harás una inspección con `ls`. Anota en tu reporte qué encontraste en cada parada.

```
cd /etc && ls | head -20          # configuraciones del sistema: ¿qué archivos
reconoces?
cd /var/log && ls -ltr | tail -5   # registros: los 5 archivos modificados más
recientemente
cd /home && ls                    # carpetas personales: ¿cuántos usuarios hay?
cd /tmp && ls -la                 # temporal: ¿hay archivos ocultos?
cd /usr/bin | head -5            # espera... ¿esto funciona? (pista: no, y es a
propósito)
```

Verificación: el último comando debe fallarte. ¿Por qué? Porque `cd` no acepta un pipe como destino: el pipe le pasa texto a la entrada del comando, pero `cd` no lee de la entrada estándar. Anota en tu reporte qué error te dio y por qué crees que pasó. Acabas de aprender algo sobre cómo `cd` realmente funciona, y eso vale más que cinco comandos que sí funcionaron.

Ahora hazlo bien y completa la visita:

```
cd /usr/bin && ls | head -5      # programas del sistema: cuenta cuántos hay con ls |
wc -l
cd ~/mapa-sistema               # regresa a tu proyecto con ruta absoluta
pwd                             # confirma dónde estás
```

Agrega a tu reporte una línea por cada carpeta visitada: qué es y un dato interesante que viste.

Paso 4: interroga al manual como detective

Elige tres comandos de esta unidad que aún no domines del todo y sácales información usando solo el sistema de ayuda. Nada de internet: el reto es sacarlo del propio sistema.

```
man ls      # busca /-t y anota para qué sirve; luego /-r; sal con q
ls --help   # compara: ¿qué tan distinto es del manual completo?
help cd     # cd es un comando interno del shell: su ayuda vive en help, no en
man
man man     # sí, el manual tiene manual. Léelo: te explica las secciones
numeradas
```

Verificación: en tu reporte debes tener anotado, con tus palabras: para qué sirve `ls -t`, para qué sirve `ls -r`, y una diferencia entre `man` y `--help`. Si lograste explicar con tus palabras por qué `help cd` funciona y `man cd` no tanto, ya entendiste la diferencia entre comandos internos y externos.

Paso 5: domina Tab y el historial bajo presión

Este paso es un juego de velocidad. Vas a navegar solo con autocompletado y luego rescatar comandos del historial.

```
cd /u<TAB>s<TAB>r<TAB>/s<TAB>h<TAB> # llega a /usr/share usando solo Tab
pwd                               # verifica que llegaste
cd -                             # regresa a la carpeta anterior de un salto
history | tail -15              # mira tus últimos 15 comandos
```

Verificación: `pwd` después del tercer comando debe mostrarte de nuevo tu carpeta `mapa-sistema` (porque `cd -` te regresó). En el historial debes ver reflejada toda tu expedición: es tu bitácora automática.

Ahora rescata un comando largo del historial sin reescribirlo: busca su número con `history | grep "var/log"` y repítelo con `!número`. Si funcionó, acabas de ahorrarte teclear una ruta larga.

Paso 6: arma y presenta tu mapa final

Agrega una última sección a tu `reporte.txt`: "Lo que aprendí", con cinco líneas honestas sobre lo que más te costó y lo que ya dominas. Luego presenta tu trabajo:

```
cd ~/mapa-sistema
ls -lh          # verifica que reporte.txt existe y pesa algo
wc -l reporte.txt # ¿cuántas líneas tiene tu reporte?
cat reporte.txt  # léelo completo: este es tu primer entregable del curso
```

Verificación final: tu proyecto está completo si tienes la carpeta `~/mapa-sistema` con `reporte.txt` dentro, el archivo contiene identidad del sistema, las cinco carpetas visitadas con un dato de cada una, las tres notas del manual, y tus cinco líneas de "lo que aprendí". Si algo falta, vuelve al paso correspondiente: un profesional no entrega incompleto.

Paso 7: convierte tu expedición en un ritual repetible

Un profesional no improvisa cada vez: convierte lo que funciona en un ritual que repite igual siempre. Vas a extraer de tu historial los comandos de reconocimiento que usaste y convertirlos en tu checklist personal, guardado en un archivo.

Primero, rescata del historial solo los comandos útiles de tu expedición:

```
history | grep -E "whoami|pwd|ls|df|free|cat /etc" | tail -20
```

Mira la lista: ahí está tu ritual, en el orden en que lo ejecutaste. Ahora créalo como documento. Abre `nano ritual-reconocimiento.txt` y escribe, con tus palabras, los pasos numerados de tu ritual de reconocimiento: qué comando va primero, qué va después y qué información te da cada uno. No copies y pegues a ciegas: escríbelo como si se lo fueras a explicar a un compañero nuevo en tu trabajo.

```
nano ritual-reconocimiento.txt
ls -lh          # verifica que existen tus dos archivos: reporte.txt y
ritual-reconocimiento.txt
```

Verificación: tu archivo `ritual-reconocimiento.txt` debe tener al menos 6 pasos numerados y cada paso debe decir qué comando usar y qué dato te entrega. Si un extraño pudiera seguir tu ritual y orientarse en una máquina nueva, está bien escrito.

Ahora la prueba de fuego: abre una terminal NUEVA (limpia, sin tu historial reciente a la vista) y ejecuta tu ritual paso a paso sobre tu propia máquina, como si fuera un servidor desconocido. Cronométrate: un profesional hace el reconocimiento completo en menos de dos minutos. Anota tu tiempo al final del archivo.

Y un ejercicio de honestidad brutal: provoca a propósito tres de los errores de la sección "Errores comunes" (por ejemplo, un `cd` a una carpeta con mayúsculas mal escritas, un `Tab` ambiguo y quedarte "atrapado" en `man`), y sal de cada uno usando la solución indicada. Marca en tu reporte cuáles provocaste y cuánto tardaste en salir. El que practica el error cuando no hay presión, no se paraliza cuando sí la hay.

Verificación final del paso: debes tener tres archivos en `~/mapa-sistema` (`reporte.txt`, `ritual-reconocimiento.txt` y la evidencia de tus tres errores provocados anotada en el reporte), y debes haber ejecutado tu ritual completo en una terminal fresca en menos de cinco minutos. Si tardaste más, no es fracaso: es tu primera marca personal. La próxima vez será más rápido.

Checklist de entrega del miniproyecto

Antes de dar por terminado el proyecto, pásale esta lista a tu trabajo como lo haría un revisor exigente. Cada punto debe cumplirse; si uno falla, regresa al paso indicado y corrígelo.

```
cd ~/mapa-sistema && pwd          # 1. estás en la carpeta del proyecto
ls -lh                          # 2. ves reporte.txt y ritual-reconocimiento.txt
wc -l reporte.txt ritual-reconocimiento.txt # 3. ningún archivo está vacío
grep -c "Paso" ritual-reconocimiento.txt    # 4. tu ritual tiene pasos numerados
history | wc -l                      # 5. tu historial refleja una sesión real de
trabajo
```

La lista en palabras: 1) la carpeta `~/mapa-sistema` existe y estás parado dentro de ella; 2) contiene `reporte.txt` con tu identidad del sistema, las cinco carpetas visitadas con un dato de cada una, tus notas del manual y tus cinco líneas de "lo que aprendí"; 3) contiene `ritual-reconocimiento.txt` con al menos 6 pasos numerados escritos con tus palabras; 4) ejecutaste el ritual completo en una terminal fresca y anotaste tu tiempo; 5) provocaste y resolviste tres errores comunes y lo documentaste. Si los cinco puntos están en verde, tu primer miniproyecto está entregado. Guárdalo: en la Unidad 6 vas a automatizar este ritual con un script, y te vas a alegrar de tenerlo bien documentado.

Reto final (sin solución, te toca a ti)

Sin mirar tus notas: abre una terminal nueva y, solo con comandos de esta unidad, averigua y anota en un papel: 1) la ruta absoluta de la carpeta donde guarda Linux los manuales de los comandos (pista: está bajo `/usr` y su nombre empieza con "sh"), 2) cuántos comandos distintos ejecutaste hoy según tu historial, 3) el nombre del archivo modificado más recientemente dentro de `/etc`. Si logras las tres sin buscar en internet, ya piensas como usuario de terminal. Si te atorras, el manual y el doble Tab son tus aliados, no tu acordeón.

Práctica

Llegó la hora de ensuciarte las manos. Estos cinco ejercicios están ordenados de menor a mayor dificultad y cubren todo lo visto en la unidad. Hazlos en orden, sin mirar la solución hasta haberlo intentado de verdad: la solución está para verificar y para aprender, no para copiar. Haz todos los ejercicios dentro de tu carpeta personal.

Ejercicio 1: tu ficha de identidad del sistema

Enunciado: usando solo comandos de esta unidad, averigua y anota: tu nombre de usuario, tu carpeta personal (ruta absoluta), qué distribución de Linux usas y cuánto espacio libre tiene tu disco principal.

Solución explicada paso a paso. Paso 1: ejecuta ``whoami`` y anota el resultado: ese es tu nombre de usuario. Paso 2: ejecuta ``pwd`` nada más abrir la terminal (o ``cd ~`` primero para asegurarte) y anota la ruta absoluta de tu casa, por ejemplo ``/home/esteban``. Paso 3: ejecuta ``cat /etc/os-release`` y busca la línea que dice NAME o PRETTY_NAME: ahí está tu distribución, por ejemplo "Ubuntu 24.04". Paso 4: ejecuta ``df -h`` y busca la fila del disco raíz (la que tiene ``/`` en la última columna): la columna "Disponible" te dice el espacio libre. Paso 5: escribe estos cuatro datos en un archivo de texto o en tu cuaderno: son tu ficha de identidad del sistema y la vas a necesitar en varias unidades. Si algún comando te dice "no se encontró", revisa que lo escribiste bien con Tab para autocompletar.

Ejercicio 2: paseo con brújula

Enunciado: sin usar el mouse, realiza este recorrido solo con `cd` y verifica cada parada con `pwd`: ve a la raíz ``/``, luego a ``/etc``, luego sube un nivel con ``..``, luego entra a ``/tmp``, y finalmente regresa a tu carpeta personal. Anota la ruta absoluta de cada parada.

Solución explicada paso a paso. Paso 1: ``cd /`` y luego ``pwd``: debe decir ``/``. Paso 2: ``cd /etc`` (puedes escribir ``cd /e`` + Tab) y ``pwd``: debe decir ``/etc``. Paso 3: ``cd ..`` y ``pwd``: debe decir ``/`` otra vez, porque subiste un nivel desde `/etc`. Paso 4: ``cd /tmp`` y ``pwd``: debe decir ``/tmp``. Paso 5: ``cd ~`` (o ``cd`` solo) y ``pwd``: debe decir tu carpeta personal. Paso 6: compara tus anotaciones: ``/``, ``/etc``, ``/``, ``/tmp``, ``/home/tuusuario``. Si alguna no coincide, repite ese salto observando con ``ls`` antes de moverte. El truco que te llevas: `pwd` es tu brújula, úsalo sin pena cada vez que dudes dónde estás.

Ejercicio 3: detective con ls

Enunciado: investiga tu carpeta personal con ls y responde: ¿cuántos archivos y carpetas hay en total (sin contar ocultos)? ¿Hay archivos ocultos? ¿Cuál es el archivo más grande de tu carpeta personal? Usa las opciones adecuadas.

Solución explicada paso a paso. Paso 1: ejecuta `ls | wc -l` para contar las líneas, o sea, los elementos no ocultos. (`wc -l` cuenta líneas; lo veremos a fondo en la Unidad 4, pero ya lo puedes usar.) Paso 2: ejecuta `ls -la` y cuenta las líneas que empiezan con punto: esos son tus archivos ocultos. Si el número total con `ls -la | wc -l` es mayor que con `ls | wc -l`, hay ocultos. Paso 3: ejecuta `ls -lhS` (la S ordena por tamaño, de mayor a menor) y mira la primera línea: ese es tu archivo más grande. Paso 4: anota tus tres respuestas. Si tu carpeta está vacía, crea primero unos archivos de prueba en la Unidad 2 y vuelve a este ejercicio: un detective necesita pistas.

Ejercicio 4: el manual te salva

Enunciado: sin buscar en internet, usa solo el manual para descubrir para qué sirve la opción `-t` de ls y la opción `-r`. Luego úsalas juntas para listar tu carpeta /etc ordenada por fecha de modificación, de la más reciente a la más antigua.

Solución explicada paso a paso. Paso 1: ejecuta `man ls`. Paso 2: dentro del manual, escribe `/t` y presiona Enter para buscar; lee la descripción: ordena por fecha de modificación. Paso 3: busca `/r` de la misma forma: invierte el orden. Paso 4: presiona `q` para salir. Paso 5: ejecuta `ls -ltr /etc`: la `-t` ordena por fecha, la `-l` da el formato largo y la `-r` invierte para que lo más reciente quede abajo (o quita la `-r` con `ls -lt /etc` si lo quieres arriba). Paso 6: observa las fechas: los archivos modificados recientemente suelen ser configuraciones que algo tocó. Acabas de hacer una investigación forense básica solo con el manual y ls.

Ejercicio 5: tu cheat sheet personal

Enunciado: crea tu primera hoja de referencia: un archivo de texto llamado `mis-comandos.txt` en tu carpeta personal con los 10 comandos de esta unidad, cada uno con una descripción de una línea escrita con tus palabras. Luego practica el historial: encuentra con `history` el número del comando con el que creaste el archivo y repítelo con `!número`.

Solución explicada paso a paso. Paso 1: crea el archivo con `nano mis-comandos.txt` (nano es un editor simple: escribes, guardas con `Ctrl+O`, sales con `Ctrl+X`) o con Visual Studio Code. Paso 2: escribe tus 10 comandos con descripciones propias, por ejemplo: "pwd - me dice en qué carpeta estoy parado". Escribir con tus palabras es lo que fija el conocimiento. Paso 3:

guarda y sal. Paso 4: ejecuta ``history | grep nano`` para encontrar el número del comando con el que abriste el editor. Paso 5: ejecuta ``!número`` con ese número para repetirlo y agregar dos comandos más a tu lista. Paso 6: a partir de hoy, cada unidad agrega sus comandos a este archivo: al final del curso tendrás tu propio manual personalizado, que vale más que cualquier acordeón descargado porque lo escribiste tú.

Tarjeta de referencia rápida

Tu acordeón de la unidad. No es para memorizar de golpe: es para tenerlo a la mano mientras practicas, hasta que los dedos lo aprendan solos. Imprímelo, pégalo junto a tu monitor o guárdalo en tu archivo ``mis-comandos.txt``.

Comandos clave de la unidad

| Comando | Descripción | |---|---| | ``pwd`` | Muestra la ruta absoluta de la carpeta donde estás parado. Tu brújula. | | ``ls`` | Lista el contenido de la carpeta actual (sin archivos ocultos). | | ``ls -l`` | Lista en formato largo: permisos, dueño, tamaño, fecha y nombre. | | ``ls -lh`` | Formato largo con tamaños legibles (K, M, G). Tu inspección diaria. | | ``ls -a`` | Incluye archivos ocultos (los que empiezan con punto). | | ``ls -la`` | Formato largo incluyendo ocultos. Radiografía completa de una carpeta. | | ``ls -ltr`` | Ordena por fecha de modificación, lo más reciente al final. | | ``ls -lhS`` | Ordena por tamaño, de mayor a menor. Para cazar lo que ocupa espacio. | | ``cd carpeta`` | Entra a una carpeta (ruta relativa o absoluta). | | ``cd ..`` | Sube un nivel en el árbol de directorios. | | ``cd ~`` o ``cd`` | Regresa a tu carpeta personal, estés donde estés. | | ``cd -`` | Regresa a la carpeta anterior. El botón de "atrás". | | ``cd /`` | Va a la raíz del sistema de archivos. | | ``whoami`` | Muestra tu nombre de usuario actual. | | ``man comando`` | Abre el manual completo del comando (``q`` para salir, ``/`` para buscar). | | ``comando --help`` | Muestra el resumen rápido de opciones del comando. | | ``help comando`` | Ayuda para comandos internos del shell como ``cd``. | | ``history`` | Muestra los comandos que ejecutaste recientemente, numerados. | | ``!número`` | Repite el comando del historial con ese número. | | ``!!`` | Repite el último comando tal cual lo escribiste. | | ``clear`` o Ctrl+L | Limpia la pantalla sin borrar nada del historial. | | ``Tab`` | Autocompleta nombres de comandos, archivos y carpetas. | | ``Tab Tab`` | Lista todas las coincidencias cuando hay varias opciones. | | ``df -h`` | Muestra el espacio libre en los discos, en formato legible. | | ``free -h`` | Muestra la memoria RAM total, usada y disponible. |

One-liners que te salvan el día

```
cd ~ && pwd && ls
```

Vuelve a tu casa, confirma dónde quedaste y lista el contenido, todo de un golpe. El "reinicio" universal cuando te pierdes navegando.

```
history | grep "ssh"
```

Encuentra todos los comandos antiguos que contenían la palabra ssh, sin desplazarte por cientos de líneas. Cambia "ssh" por lo que busques: es tu memoria externa.

```
ls -ltr /var/log | tail -5
```

Muestra los 5 archivos modificados más recientemente en la carpeta de registros del sistema. La forma más rápida de ver qué estuvo pasando en la máquina.

```
sudo !!
```

Repite el último comando agregándole `sudo` al inicio. Para ese momento clásico en que el comando falla por falta de permisos y no quieres reescribirlo todo.

```
man -k "copy"
```

Busca en las descripciones de todos los manuales qué comandos hablan de "copy". Cuando sabes lo que quieres hacer pero no recuerdas el nombre del comando, este es tu detective.

Atajos de teclado que te hacen ver profesional

La terminal se usa con el teclado, y estos atajos son la diferencia entre teclear como principiante y moverte como alguien que vive en la terminal. No necesitas memorizarlos todos hoy: empieza con Ctrl+C, Ctrl+L y las flechas.

| Atajo | Descripción | |---|---| | `Ctrl+C` | Cancela el comando en curso. El botón de pánico que siempre funciona. | | `Ctrl+L` | Limpia la pantalla (igual que `clear`) sin borrar el historial. | | `Ctrl+D` | Cierra la terminal (equivale a escribir `exit`). | | `Flecha arriba/abajo` | Recorre los comandos anteriores del historial, uno por uno. | | `Ctrl+R` | Búsqueda interactiva hacia atrás en el historial: escribe y te sugiere. | | `Ctrl+A` | Lleva el cursor al inicio de la línea actual. | | `Ctrl+E` | Lleva el cursor al final de la línea actual. | | `Ctrl+U` | Borra todo lo que escribiste

desde el cursor hasta el inicio de la línea. | | `Ctrl+K` | Borra todo lo que escribiste desde el cursor hasta el final de la línea. | | `Ctrl+W` | Borra la palabra anterior al cursor. Para corregir sin borrar letra por letra. | | `Tab` | Autocompleta el comando, archivo o carpeta que estás escribiendo. | | `Tab Tab` | Muestra todas las opciones cuando hay varias coincidencias. |

Combinaciones de comandos de uso diario

Estos no son comandos nuevos: son los de esta unidad combinados como lo hace alguien que ya lleva tiempo en la terminal. Prueba cada uno y fíjate en el patrón: moverse, verificar, inspeccionar.

```
cd /etc && ls -la | head -20
```

Entra a `/etc` y muestra los primeros 20 elementos en formato largo. El `&&` significa "si el cd funcionó, entonces lista": así no listas la carpeta equivocada por accidente.

```
cd ~/proyectos && pwd && ls -lh
```

El trío de aterrizaje: muévete a tu carpeta de proyectos, confirma dónde quedaste con `pwd` e inspecciónala con `ls -lh`. Hazlo siempre en este orden.

```
ls -ltr | tail -10
```

Los 10 archivos modificados más recientemente en tu carpeta actual, con el más nuevo al final. Ideal para ver en qué estuviste trabajando.

```
history | grep "cd /" | tail -5
```

Tus últimas 5 visitas a rutas absolutas: un mapa de a dónde sueles moverte en el sistema.

```
man ls | head -30
```

Lee solo las primeras 30 líneas del manual de `ls`: la descripción y la sinopsis, sin ahogarte en las 200 opciones. Cuando un manual te abrume, empieza por el encabezado.

```
ls -d ~/.* | head -20
```

Lista los archivos y carpetas ocultas de tu casa (los que empiezan con punto). Así descubres dónde guarda cada programa su configuración, sin necesidad de filtros complicados.

Resumen de la unidad

- Linux es el sistema operativo que gobierna los servidores, la nube, Android y las supercomputadoras: aprenderlo es aprender donde vive el internet.
- La terminal es la ventana, el shell (bash) es el intérprete que ejecuta tus órdenes, y el prompt te dice quién eres, dónde estás y que está listo.
- El ritual base es: escribes un comando, presionas Enter, lees el resultado. Todo lo demás se construye sobre eso.
- whoami te dice tu usuario, pwd te dice tu carpeta actual, ls lista el contenido: tu trío de orientación.
- Las rutas absolutas empiezan en `/` y funcionan desde donde sea; las relativas parten de tu ubicación actual.
- `.` es "aquí" y `..` es "la carpeta de arriba": atajos universales para navegar y construir rutas.
- `cd` te mueve: `cd carpeta`, `cd ..`, `cd ~` o `cd` solo para tu casa, `cd /` para la raíz.
- `ls -lh` es tu comando de inspección diario: formato largo con tamaños legibles por humanos.
- `man`, `--help` y `help` son tu sistema de ayuda: del resumen rápido al manual completo, nunca estás solo.
- Tab autocompleta y evita errores de dedo; el historial (flecha arriba, `history`, !número) te evita reescribir.
- Practica siempre en tu zona de prueba y verifica con `pwd` y `ls` antes de actuar: el profesional confirma, no supone.
- La terminal es la ventana y el shell es quien ejecuta tus órdenes: cambia de programa de terminal cuando quieras, tus comandos no cambian.
- Lee tu prompt antes de escribir: `usuario@equipo:carpeta$` te dice quién eres y dónde estás; si ves `#` en lugar de `$`, eres root y cada comando es de alto riesgo.
- `cd -` es el botón de "atrás" de la terminal: te regresa a la carpeta anterior de un solo salto.
- `ls -a` revela los archivos ocultos (empiezan con punto): ahí viven tus configuraciones personales, no los ignores.
- Si Tab no completa, pulsa Tab dos veces: la terminal te lista todas las coincidencias para que elijas en lugar de adivinar.

- `!!` repite el último comando tal cual; `sudo !!` lo reintenta como administrador cuando te faltó permiso.
- `history | grep palabra` rescata comandos viejos sin recorrer cientos de líneas a mano: tu memoria externa.
- Dentro de `man` o `less`, escribe `/texto` para buscar y `q` para salir: así se doman los manuales sin pánico.
- Linux distingue mayúsculas de minúsculas en todo: `Archivo.txt` y `archivo.txt` son dos archivos distintos, y `LS` no es un comando.

UNIDAD 2: Archivos y directorios

Ya sabes orientarte en la terminal: sabes quién eres, dónde estás y cómo pedir ayuda. Ahora viene el siguiente superpoder: manipular archivos y carpetas sin tocar el mouse. Crear, copiar, mover, renombrar, borrar, ver contenido y encontrar archivos perdidos, todo con comandos. Esta es la unidad que más vas a usar en tu vida diaria con Linux, porque todo trabajo real empieza manejando archivos.

En esta unidad vas a aprender a crear carpetas con `mkdir` y archivos vacíos con `touch`, a copiar con `cp`, mover y renombrar con `mv`, y borrar con `rm` (incluyendo su lado peligroso). Vas a ver el contenido de archivos con `cat`, `less`, `head` y `tail`, vas a usar los comodines `*` y `?` para trabajar con grupos de archivos de un golpe, y vas a encontrar cualquier archivo del sistema con `find` y `locate`.

Por qué importa tanto esta unidad: los archivos son la materia prima de todo. Los scripts que escribirás desde la Unidad 6 son archivos. Las configuraciones que tocarás como administrador son archivos. Los respaldos de tu proyecto final son archivos. Quien no domina la manipulación de archivos en la terminal trabaja a la mitad de velocidad y con miedo. Quien la domina, mueve cientos de archivos con un solo comando mientras otros siguen haciendo clic.

Hay otra razón: esta unidad te enseña respeto por el comando `rm`. Borrar en la terminal no tiene papelera de reciclaje: lo borrado, borrado está. Vas a aprender la disciplina de verificar antes de borrar, y esa disciplina te va a salvar de un desastre algún día. Créeme: todo administrador Linux tiene una historia de terror con `rm`. La tuya no tiene por qué existir si aprendes la lección aquí, en tu zona de práctica, y no en un servidor de producción.

Cómo encaja esta unidad en el curso: la Unidad 1 te dio la brújula; esta te da las manos. En la Unidad 3 les pondrás cerradura a estos archivos con los permisos. En la Unidad 4 los vas a encadenar con pipes. Y desde la Unidad 6, cada script que escribas vivirá en un archivo que crearás, editarás y ejecutarás con lo aprendido aquí.

Una nota de estudio: haz TODOS los ejemplos dentro de una carpeta de práctica, por ejemplo `~/practica-linux`. Créala al empezar y no salgas de ella en esta unidad. El comando `rm` es como una motosierra: utilísima y peligrosa. En tu zona de práctica puedes experimentar sin consecuencias.

Empecemos. Al terminar esta unidad vas a crear estructuras de carpetas completas, organizar archivos por lotes y encontrar cualquier cosa en el sistema, todo desde la terminal y más rápido que con cualquier explorador gráfico.

Explicación detallada

mkdir y touch: crear carpetas y archivos

Para crear una carpeta se usa mkdir, de "make directory":

```
mkdir mis-documentos
```

Listo: nació la carpeta. Para crear carpetas anidadas de un golpe, la opción -p (de "parents", padres) crea toda la ruta aunque los niveles intermedios no existan:

```
mkdir -p proyectos/web/imagenes
```

Sin -p, ese comando fallaría porque proyectos no existe. Con -p, el shell crea proyectos, luego web dentro, luego imagenes dentro. Es la forma profesional de armar estructuras completas en un solo comando.

Para crear un archivo vacío se usa touch:

```
touch notas.txt
```

Originalmente touch servía para actualizar la fecha de un archivo, pero su efecto secundario (crear el archivo si no existe) se volvió su uso más popular. Es la forma rápida de crear archivos vacíos para practicar o como marcadores.

Analogía: mkdir es como construir cuartos nuevos en tu casa, y touch es como poner hojas en blanco sobre el escritorio, listas para escribir. La opción -p es como contratar albañiles que construyen toda la planta de un jalón.

cp: copiar archivos y carpetas

cp significa "copy". Su forma básica es origen y destino:

```
cp notas.txt copia-notas.txt
```

Esto crea copia-notas.txt con el mismo contenido. El original no se toca. Para copiar a otra carpeta:

```
cp notas.txt documentos/
```

Para copiar carpetas completas necesitas la opción -r (de "recursive", recursiva), que entra a la carpeta y copia todo lo de adentro:

```
cp -r proyectos respaldo-proyectos
```

Sin -r, copiar una carpeta falla con un mensaje de que es un directorio. Grábate esto: carpetas siempre con -r.

Otras opciones útiles: `-i` (de "interactive") te pregunta antes de sobrescribir un archivo existente, y `-v` (de "verbose") te muestra lo que va copiando. `cp -iv` es la combinación del precavido: te avisa y te pregunta.

mv: mover y renombrar (son lo mismo)

Aquí hay un dato curioso: en Linux, mover y renombrar son la misma operación. mv significa "move":

```
mv notas.txt documentos/
```

Mueve el archivo a la carpeta documentos. Y para renombrar:

```
mv notas.txt apuntes.txt
```

Lo "mueve" al mismo lugar pero con otro nombre. Por eso no hay un comando separado para renombrar: renombrar es mover al mismo directorio con nombre distinto.

Igual que cp, mv acepta `-i` para preguntar antes de sobrescribir. Úsalo cuando no estés seguro. Mover cientos de archivos con un comodín y darte cuenta de que sobrescribiste algo importante es una de esas lecciones que solo quieres aprender una vez, y mejor en teoría que en la práctica.

rm: borrar (sin papelera, sin perdón)

rm significa "remove". Borra archivos:

```
rm archivo-viejo.txt
```

Para borrar carpetas, otra vez necesitas -r:

```
rm -r carpeta-vieja
```

Y aquí viene la parte seria. Existe la opción `-f` (de "force", forzar), que borra sin preguntar ni quejarse. La combinación ``rm -rf`` es la motosierra de Linux: borra carpetas enteras con todo su contenido sin pedir confirmación. Es utilísima y es la protagonista de las historias de terror.

La regla de oro: nunca ejecutes ``rm -rf`` con una ruta que no hayas verificado con ``ls`` justo antes. Y jamás, bajo ninguna circunstancia, ejecutes ``rm -rf /`` ni ``rm -rf ~``: el primero intenta borrar todo el sistema, el segundo toda tu carpeta personal. Los sistemas modernos tienen protecciones, pero no apuestes tu trabajo a eso.

Buenas prácticas: usa ``rm -i`` (pregunta antes de cada borrado) cuando estés inseguro, verifica la ruta con ``ls`` o con ``pwd`` antes de borrar, y mantén respaldos de lo importante (en la Unidad 10 vas a automatizar eso). Borrar en la terminal es para siempre: no hay papelera, no hay "deshacer".

Ver contenido: `cat`, `less`, `head` y `tail`

`cat` significa "concatenate" y su uso más común es mostrar el contenido completo de un archivo:

```
cat notas.txt
```

Perfecto para archivos cortos. Para archivos largos, `cat` te inunda la pantalla, así que usas `less`, que te deja navegar con las flechas y salir con ``q``:

```
less archivo-largo.txt
```

`head` muestra las primeras líneas (10 por defecto) y `tail` las últimas:

```
head -20 registro.log
tail -30 registro.log
```

`tail` tiene un truco legendario: la opción `-f` (de "follow", seguir) mantiene el archivo abierto y te muestra las líneas nuevas conforme se escriben:

```
tail -f /var/log/syslog
```

Esto es oro para administradores: ves en vivo lo que está pasando en el sistema, como ver el electrocardiograma de la máquina. Para salir, `Ctrl+C`.

Piensa en estos comandos como formas de leer: `cat` es leer la carta completa de un jalón, `less` es leer el libro con calma pasando páginas, `head` es leer solo la introducción, `tail` es leer solo la conclusión, y `tail -f` es quedarte pegado a la ventana viendo pasar la historia en tiempo real.

Comodines: * y ? para trabajar en lote

Los comodines (wildcards) son patrones que el shell expande a los archivos que coincidan. El asterisco `*` significa "cualquier cosa, de cualquier largo":

```
ls *.txt
```

Lista todos los archivos que terminan en `.txt`. El signo `?` significa "exactamente un carácter cualquiera":

```
ls foto?.jpg
```

Coincide con `foto1.jpg`, `fotoA.jpg`, pero no con `foto12.jpg` (ese tiene dos caracteres).

Los comodines se combinan con cualquier comando:

```
cp *.txt respaldo/  
mv foto*.jpg imagenes/  
rm *.tmp
```

Esa última línea borra todos los `.tmp` de un golpe: potentísimo y peligrosísimo a la vez. Antes de un `rm` con comodín, ejecuta primero el mismo patrón con `ls` para ver exactamente qué vas a borrar. `ls *.tmp` te muestra las víctimas; si la lista es correcta, cambias `ls` por `rm`. Este hábito te va a salvar más de una vez.

find: el detective privado de archivos

`find` busca archivos recorriendo carpetas con criterios. Su forma básica:

```
find ~/practica-linux -name "*.txt"
```

Busca en tu carpeta de práctica todos los archivos cuyo nombre termine en `.txt`. Los criterios más útiles:

```
find . -name "foto*.jpg"  
find . -type f -size +10M  
find . -type d -empty  
find . -mtime -7
```

De arriba abajo: en la carpeta actual, archivos jpg que empiecen con foto; archivos (-type f) de más de 10 megabytes; carpetas (-type d) vacías; archivos modificados en los últimos 7 días (-mtime -7, "modification time"). find es el comando más expresivo para búsquedas: si puedes describir lo que buscas, find lo encuentra.

locate: búsqueda instantánea

locate busca por nombre en una base de datos indexada del sistema, así que es instantáneo:

```
locate notas.txt
```

La desventaja: la base de datos se actualiza periódicamente, así que un archivo creado hace dos minutos quizá no aparezca todavía. En Ubuntu la actualizas con `sudo updatedb`. Regla práctica: locate para búsquedas rápidas por nombre en todo el sistema, find para búsquedas precisas con criterios y en tiempo real.

Ejemplos paso a paso

Ejemplo 1: construye tu zona de práctica

Vas a crear la carpeta donde harás todos los ejercicios del curso, con subcarpetas organizadas.

```
mkdir -p ~/practica-linux/{documentos,imagenes,scripts,respaldo}
```

Las llaves `{}` son expansión del shell: crean las cuatro subcarpetas de un golpe. Verifica:

```
ls -R ~/practica-linux
```

La opción -R (recursiva) lista la carpeta y todo lo que hay dentro, nivel por nivel. Debes ver documentos, imagenes, scripts y respaldo.

```
cd ~/practica-linux
pwd
touch documentos/nota1.txt documentos/nota2.txt
ls -lh documentos/
```

Te paras en tu zona de práctica, creas dos archivos de prueba y confirmas que existen. A partir de ahora, todos los ejemplos de este libro se hacen aquí dentro.

Explicación: la expansión con llaves te ahorra cuatro comandos `mkdir`; ``ls -R`` te da el mapa completo de tu estructura. Esta zona es tu laboratorio: aquí puedes romper cosas sin consecuencias.

Ejemplo 2: organiza archivos por lote

Simula el caos típico: archivos regados, y los ordenas con comodines.

```
cd ~/practica-linux
touch foto1.jpg foto2.jpg fotoA.jpg reporte-ene.txt reporte-feb.txt temp1.tmp
temp2.tmp
ls
```

Creas 7 archivos de prueba. Ahora organízalos:

```
mv foto*.jpg imagenes/
mv reporte-*.txt documentos/
ls
```

Los jpg se fueron a imagenes, los reportes a documentos. Quedan solo los .tmp.

```
ls *.tmp
rm -i *.tmp
ls
```

Primero inspeccionas con `ls` qué vas a borrar (las víctimas), luego borras con `-i` para confirmar cada uno. La carpeta queda limpia.

Explicación: el flujo profesional con lotes es inspeccionar con `ls`, actuar con `mv/cp/rm` usando el mismo patrón. Los comodines convierten 20 operaciones en una sola. Y el paso de inspección previa es tu seguro de vida.

Ejemplo 3: espía un archivo en vivo

Vas a ver cómo se comporta `tail -f`, la herramienta de monitoreo en tiempo real.

```
cd ~/practica-linux
touch mi-diario.log
tail -f mi-diario.log
```

La terminal se queda "colgada" esperando: está vigilando el archivo. Ahora abre OTRA terminal (o pestaña) y escribe:

```
echo "primera línea del diario" >> ~/practica-linux/mi-diario.log
echo "segunda línea del diario" >> ~/practica-linux/mi-diario.log
```

(El `>>` agrega texto al final del archivo; lo veremos a fondo en la Unidad 4.) Vuelve a la primera terminal: las líneas aparecieron solas, en vivo. Presiona `Ctrl+C` para salir del modo vigilancia.

```
cat mi-diario.log
head -1 mi-diario.log
tail -1 mi-diario.log
```

Confirma el contenido completo, la primera y la última línea.

Explicación: `tail -f` es como ponerle una cámara de seguridad al archivo. Los administradores lo usan para ver registros del sistema en tiempo real cuando algo falla. El `>>` por ahora solo úsalo como "agregar línea"; su teoría completa viene en la Unidad 4.

Ejemplo 4: encuentra la aguja en el pajar

Vas a esconder un archivo y encontrarlo con `find` y `locate`.

```
cd ~/practica-linux
mkdir -p escondite/nivel1/nivel2/nivel3
echo "tesoro" > escondite/nivel1/nivel2/nivel3/tesoro-secreto.txt
find ~/practica-linux -name "*tesoro*"
```

`find` recorre todo y te devuelve la ruta completa del archivo escondido, sin importar qué tan profundo esté.

```
find ~/practica-linux -type f -name "*.txt" | head -5
touch -d "10 days ago" ~/practica-linux/viejo.txt
find ~/practica-linux -mtime +7
```

Buscas todos los txt (limitando a 5 con head), creas un archivo con fecha de hace 10 días (touch -d ajusta la fecha), y find con -mtime +7 lo encuentra por antigüedad: archivos modificados hace más de 7 días.

Explicación: find combina criterios (nombre, tipo, tamaño, fecha) para búsquedas quirúrgicas. Cuando administres un servidor con miles de archivos, esta precisión es la diferencia entre encontrar el problema en un minuto o en una hora.

Errores comunes y casos reales

Manipular archivos desde la terminal es rapidísimo, y por eso mismo los errores aquí duelen más: un `rm` mal escrito no tiene botón de deshacer. Todos los casos de esta sección le han pasado a alguien de verdad, varios a profesionales con años de experiencia. Léelos con la terminal abierta y provoca cada uno en tu zona de práctica: es más barato aprender con archivos de mentira que con los de verdad.

Caso 1: `cp` se niega a copiar una carpeta ("omitting directory")

Síntoma: ejecutas `cp mis-fotos respaldo-fotos` y la terminal responde `cp: -r not specified; omitting directory 'mis-fotos'`. No se copió nada.

Por qué ocurre: `cp` por defecto solo copia archivos. Las carpetas (que pueden contener miles de archivos y subcarpetas) requieren que le digas explícitamente que copie de forma recursiva con `-r`. Es una protección: copiar una carpeta entera por accidente puede duplicar gigabytes sin que te des cuenta.

Solución: agrega `-r` cuando el origen sea una carpeta. Y verifica después con `ls`.

```
cp mis-fotos respaldo-fotos      # ERROR: omite el directorio
cp -r mis-fotos respaldo-fotos   # CORRECTO: copia la carpeta con todo adentro
ls respaldo-fotos                # verifica que el contenido llegó
```

Lección profesional: cada vez que copies algo y el resultado "no aparece", pregunta primero si era una carpeta y te faltó `-r`. Es el error número uno al hacer respaldos manuales.

Caso 2: `mv` "desaparece" tu archivo

Síntoma: ejecutas `mv reporte.txt reportes/` para moverlo a una carpeta, luego `ls` y el archivo ya no está donde estaba... pero tampoco está en `reportes/`. Pánico.

Por qué ocurre: casi siempre la carpeta destino no existía o la escribiste mal. Si `reportes/` no existe, `mv` no falla: interpreta que quieres RENOMBRAR `reporte.txt` a un archivo llamado `reportes`. Tu archivo sigue existiendo, solo que ahora se llama `reportes` y está en la carpeta actual.

Solución: verifica con `ls` qué pasó realmente, y si fue un renombrado accidental, renómbralo de vuelta. Luego crea la carpeta y mueve de nuevo.

```
mv reporte.txt reportes/      # si reportes/ no existe, RENOMBRA el archivo a
"reportes"
ls                             # sorpresa: hay un archivo llamado "reportes"
mv reportes reporte.txt       # deshaz el renombrado accidental
mkdir -p reportes             # crea la carpeta destino primero
mv reporte.txt reportes/      # ahora sí lo mueve adentro
ls reportes/                  # verifica que llegó
```

Lección profesional: `mv` hace dos cosas distintas (mover y renombrar) según el contexto, y eso lo vuelve traicionero. Antes de un `mv`, verifica que el destino existe con `ls`. Los profesionales usan `mv -i` para que pregunte antes de sobrescribir.

Caso 3: el comodín que borró de más

Síntoma: querías borrar solo los respaldos viejos con `rm *.bak` y de pronto notas que también desaparecieron archivos que no eran `.bak`, o peor: ejecutaste `rm *.bak` (con un espacio de más) y borraste TODO lo de la carpeta.

Por qué ocurre: el shell expande los comodines ANTES de que `rm` los vea. `rm *.bak` se convierte en "borra todo (*) y además `.bak`". Un solo espacio convierte una limpieza selectiva en una masacre. Y `rm` no tiene papelera: lo borrado no vuelve.

Solución: nunca ejecutes un `rm` con comodín sin antes inspeccionar con `ls` usando EXACTAMENTE el mismo patrón. Si el patrón es peligroso, usa `-i` para que pregunte archivo por archivo.

```
ls *.bak                       # PRIMERO mira qué va a borrar el patrón
rm -i *.bak                    # con -i confirma cada archivo antes de borrarlo
rm *.bak                       # solo cuando ya viste que la lista es correcta
```

Lección profesional: la regla de oro es "ls primero, rm después". Muchos administradores tienen alias que obligan a ``rm -i`` siempre. El espacio de más en ``rm * .txt`` ha destruido más trabajo que cualquier virus.

Caso 4: ``rm -rf`` ejecutado en la carpeta equivocada

Síntoma: querías limpiar ``~/proyectos/prueba`` y ejecutaste ``rm -rf`` estando parado en ``~/proyectos`` sin darte cuenta, o con una ruta construida con una variable vacía. El comando no pregunta, no avisa, no perdona.

Por qué ocurre: ``rm -rf`` es el comando más peligroso del sistema: ``-r`` borra recursivamente y ``-f`` elimina las confirmaciones. Combinado con estar en la carpeta equivocada o una ruta mal escrita, borra en segundos lo que tardaste meses en crear.

Solución: no hay solución después del hecho (por eso es el caso más serio). La prevención es el único antídoto: verifica con ``pwd`` dónde estás, usa rutas absolutas, y para limpiezas grandes prueba primero con ``ls -R`` o mueve a una carpeta "papelera" temporal en vez de borrar directo.

```
pwd                # ¿dónde estoy parado ANTES de borrar?
ls -R carpeta-a-borrar  # mira TODO lo que contiene, recursivamente
mkdir -p ~/papelera-tmp  # en vez de rm directo, mueve ahí primero
mv carpeta-a-borrar ~/papelera-tmp/
ls ~/papelera-tmp      # confirma que se movió lo correcto
# ... una semana después, ya seguro:
rm -rf ~/papelera-tmp/carpeta-a-borrar
```

Lección profesional: los profesionales paranoides no usan ``rm -rf`` directo sobre cosas importantes: mueven a una papelera temporal y borran días después. La prisa y ``rm -rf`` son enemigos mortales.

Caso 5: ``cat`` en un archivo binario deja la terminal "loca"

Síntoma: por curiosidad haces ``cat programa`` sobre un ejecutable o ``cat foto.jpg``, y la terminal se llena de símbolos raros, suena la campana, y después tu prompt se ve extraño o no muestra lo que escribes.

Por qué ocurre: ``cat`` está hecho para texto. Al mostrar bytes binarios, algunos se interpretan como secuencias de control de la terminal y cambian su configuración (colores, fuentes, eco de teclado).

Solución: escribe ``reset`` y presiona Enter (aunque no veas lo que escribes) para restaurar la terminal. Y usa ``file`` antes de ``cat`` cuando dudes de qué tipo de archivo es.

```
cat foto.jpg      # ERROR típico: basura binaria en pantalla
reset             # restaura la terminal (escribelo a ciegas si hace falta)
file foto.jpg     # te dice: "JPEG image data" -> no es texto, no uses cat
file notas.txt    # te dice: "ASCII text" -> este sí se puede ver con cat
```

Lección profesional: ``file`` es el comando que casi nadie conoce y todos deberían usar. Antes de abrir cualquier archivo desconocido, ``file`` te dice qué es en una línea.

Caso 6: ``find`` con el patrón sin comillas no encuentra nada (o encuentra de más)

Síntoma: ejecutas ``find . -name *.log`` y te dice que no hay nada, o te devuelve resultados rarísimos que no tienen sentido.

Por qué ocurre: sin comillas, el SHELL expande ``*.log`` antes de que ``find`` lo vea, usando los archivos de tu carpeta ACTUAL. Si ahí hay un ``sistema.log``, el comando se convierte en ``find . -name sistema.log``, que busca solo ese nombre en vez del patrón. Si no hay ningún ``*.log`` en tu carpeta actual, el patrón llega intacto y a veces "funciona", lo que lo hace un error intermitente y confuso.

Solución: el patrón de ``-name`` siempre va entre comillas simples o dobles, para que lo interprete ``find`` y no el shell.

```
find . -name *.log      # MAL: el shell expande el * antes que find
find . -name "*.log"    # BIEN: find recibe el patrón intacto
find /var/log -name "*.log" -mtime -7  # los .log modificados en los últimos 7 días
```

Lección profesional: "entrecomilla los patrones" es una regla que aplica a ``find``, a ``grep`` y a casi todo lo que recibe comodines como argumento. Si un comando con ``*`` se comporta raro, lo primero que se revisa son las comillas.

Caso real 1: respaldo manual antes de actualizar un proyecto

Vas a actualizar el sistema o a hacer un cambio riesgoso en tu proyecto. Un profesional no confía en su memoria: hace un respaldo con fecha antes de tocar nada, usando ``cp -r`` y ``mkdir -p``:

```
cd ~
mkdir -p respaldos/2026-09-20
cp -r mi-proyecto respaldos/2026-09-20/
ls -lh respaldos/2026-09-20/
du -sh respaldos/2026-09-20/mi-proyecto
```

Con esas cinco líneas tienes una copia completa y fechada de tu trabajo. Si la actualización rompe algo, restauras copiando de vuelta. Este hábito de "respaldo con fecha antes del cambio" es lo que separa al que duerme tranquilo del que reza.

Caso real 2: limpieza de registros viejos en un servidor

El disco de un servidor se está llenando por registros viejos. En vez de borrar a mano (lento y peligroso), usas `find` con criterio de fecha, pero PRIMERO inspeccionas y luego borras:

```
find /var/log -name "*.log" -mtime +30 | head -20    # ¿cuáles son? mira primero
find /var/log -name "*.log" -mtime +30 | wc -l        # ¿cuántos son?
du -sh /var/log                                       # ¿cuánto pesan en total?
# solo cuando la lista es correcta:
find /var/log -name "*.log" -mtime +30 -delete       # borra los de más de 30 días
df -h                                                # verifica cuánto espacio
liberaste
```

Fíjate en el patrón profesional: inspeccionar, contar, medir y solo entonces actuar. El `-delete` de `find` es tan definitivo como `rm`, así que se usa con la misma disciplina.

Miniproyecto guiado de la unidad

Vas a construir un **Sistema de Organización de Descargas**: un conjunto de carpetas y reglas para clasificar archivos automáticamente por tipo, usando todo lo de la unidad (`mkdir`, `touch`, `cp`, `mv`, `rm`, `cat`, `less`, `head`, `tail`, comodines y `find`). Al terminar tendrás una estructura real que puedes seguir usando, y habrás practicado cada comando en un escenario con sentido.

Paso 1: crea la estructura del proyecto

Dentro de tu casa, crea la carpeta del proyecto con subcarpetas para cada tipo de archivo. Usa `mkdir -p` para crear toda la rama de un golpe.

```
cd ~
mkdir -p organizador/{entrada,documentos,imagenes,scripts,respaldos}
ls -R organizador
```

Verificación: `ls -R organizador` debe mostrar las 5 subcarpetas dentro de `organizador`. Si alguna falta, créala por separado y fíjate qué escribiste mal: los nombres deben ser exactos.

Paso 2: genera archivos de prueba realistas

Simula una carpeta de descargas caótica: crea 15 archivos de distintos tipos con `touch`, con nombres variados para practicar comodines después.

```
cd ~/organizador/entrada
touch reporte-enero.txt reporte-febrero.txt reporte-marzo.txt notas.txt lista.txt
touch foto-playa.jpg foto-montana.jpg logo.png captura.png
touch respaldo-enero.bak respaldo-febrero.bak viejo.tmp
touch instalar.sh actualizar.sh
ls | wc -l    # debes contar 15 archivos
```

Verificación: `ls | wc -l` debe decir 15. Luego inspecciona por grupos con comodines para confirmar que los nombres quedaron bien:

```
ls *.txt          # los 5 de texto
ls *.jpg *.png    # las 4 imágenes
ls *.sh           # los 2 scripts
ls *.bak          # los 2 respaldos viejos
```

Paso 3: clasifica con `mv` y comodines

Mueve cada grupo a su carpeta usando comodines. Un comando por grupo: así se siente el poder de aplicar una operación a muchos archivos de un golpe.

```
cd ~/organizador/entrada
mv *.txt ../documentos/
mv *.jpg *.png ../imagenes/
mv *.sh ../scripts/
ls                # ¿qué quedó en entrada? solo respaldo-*.bak y viejo.tmp
ls ../documentos  # verifica que llegaron los 5
ls ../imagenes    # verifica las 4 imágenes
ls ../scripts     # verifica los 2 scripts
```

Verificación: `ls` en `entrada` debe mostrar solo 3 archivos (los `.bak` y el `.tmp`). Si algún `mv` movió de más o de menos, revisa con `ls` qué patrón usaste: recuerda que `\*.txt` también habría movido un archivo llamado `notas.txt.bak` si existiera... bueno, no, porque termina en `.bak`. Pero sí movería `respaldo.txt.bak`? No: termina en `.bak`. El punto es: verifica siempre después de un `mv` con comodín.

Paso 4: inspecciona el contenido con cat, head y tail

Dale contenido real a algunos archivos para practicar inspección. Usa `cat` con redirección simple de eco... espera, la redirección es de la Unidad 4. Por ahora, crea contenido con un editor o con `printf`. Lo más simple: usa `nano` para escribir unas líneas en dos archivos.

```
cd ~/organizador/documentos
nano reporte-enero.txt      # escribe 10 líneas de texto de prueba, guarda con Ctrl+O
cat reporte-enero.txt      # míralo completo: es corto, cat es perfecto
head -n 3 reporte-enero.txt # solo las primeras 3 líneas
tail -n 3 reporte-enero.txt # solo las últimas 3 líneas
```

Ahora genera un archivo largo para practicar `less`: copia el contenido varias veces.

```
cat reporte-enero.txt reporte-febrero.txt reporte-marzo.txt > grande.txt
wc -l grande.txt          # ¿cuántas líneas tiene?
less grande.txt           # navega con flechas y espacio, sal con q
```

Ups: acabamos de usar `>` (redirección), que es tema de la Unidad 4. No te preocupes: funciona igual que un `cp` que concatena. Lo importante aquí es practicar `less` con un archivo largo de verdad. Verificación: `less` debe permitirte navegar y salir con `q` sin dramas.

Paso 5: busca como detective con `find`

Practica búsquedas con criterios combinados sobre tu proyecto, como lo harías en un servidor real.

```
cd ~/organizador
find . -name "*.txt"          # todos los txt, en cualquier subcarpeta
find . -type f | wc -l       # ¿cuántos archivos hay en total?
find . -type d               # solo las carpetas
find . -name "*.bak"         # los respaldos viejos que quedaron en entrada
find . -size +1k             # archivos de más de 1 kilobyte
find . -name "*.sh" -exec ls -lh {} +
```

Ese último comando es avanzado: ``-exec`` ejecuta ``ls -lh`` sobre cada archivo encontrado. No necesitas dominarlo hoy, pero mira lo que hace: ``find`` no solo busca, también actúa sobre lo que encuentra. Verificación: cada ``find`` debe devolver resultados coherentes con lo que creaste en los pasos 1 a 4. Si ``find . -type f | wc -l`` no te da el número que esperas, cuenta a mano con ``ls -R`` y detecta la diferencia.

Paso 6: respalda y limpia con disciplina

Haz un respaldo fechado de todo el proyecto con ``cp -r``, y luego limpia los archivos basura (``.bak`` y ``.tmp``) siguiendo el protocolo seguro: inspeccionar primero, borrar después.

```
cd ~
cp -r organizador respaldos-organizador-2026-09-20
ls respaldos-organizador-2026-09-20 # verifica el respaldo
cd ~/organizador/entrada
ls *.bak *.tmp # PASO 1 del protocolo: inspecciona qué se va a borrar
rm -i *.bak viejo.tmp # PASO 2: borra con confirmación
ls # PASO 3: verifica que quedó vacío y limpio
```

Verificación: ``entrada`` debe quedar vacía, ``documentos`` con 6 archivos (5 txt + grande.txt), ``imagenes`` con 4, ``scripts`` con 2, y tu respaldo en ``~`` debe ser una copia completa. Compruébalo:

```
find ~/organizador -type f | wc -l
find ~/respaldos-organizador-2026-09-20 -type f | wc -l
```

Ambos deben dar el mismo número... espera, no: el respaldo se hizo ANTES de borrar los ``.bak`` y ``.tmp``, así que el respaldo debe tener 3 archivos más. Si entendiste por qué los números difieren, entendiste el orden de las operaciones. Anótalo.

Paso 7: documenta tu sistema de organización

Crea un archivo ``LEEME.txt`` en la raíz del proyecto explicando tu sistema: qué va en cada carpeta y cuál es el procedimiento para clasificar descargas nuevas. Usa ``cat`` para verificarlo al final.

```
cd ~/organizador
nano LEEME.txt
cat LEEME.txt
ls -R
```

Verificación final: tu proyecto está completo si ``ls -R ~/organizador`` muestra las 5 carpetas con su contenido clasificado, el ``LEEME.txt`` explica el sistema con tus palabras, y tienes el respaldo fechado en tu casa. Este es un sistema real: a partir de hoy puedes usar estas carpetas para tus descargas de verdad.

Paso 8: simulacro de desastre y recuperación

Aquí viene la parte que convierte este proyecto en entrenamiento profesional de verdad: vas a romper tu organizador a propósito y luego restaurarlo desde el respaldo. En la vida real los desastres no avisan, así que se ensayan.

Primero, verifica que tu respaldo esté completo y sea usable. Un respaldo que nunca se probó no es un respaldo, es un deseo.

```
find ~/respaldos-organizador-2026-09-20 -type f | sort > /tmp/lista-respaldo.txt
find ~/organizador -type f | grep -v LEEME | sort > /tmp/lista-actual.txt
wc -l /tmp/lista-respaldo.txt /tmp/lista-actual.txt
```

Compara las listas: el respaldo debe tener todo lo que el organizador tenía antes de la limpieza (incluidos los ``.bak`` y el ``.tmp``), más la diferencia que ya analizaste. Si algo del organizador actual NO está en el respaldo (excepto el ``LEEME.txt`` que creaste después), tu respaldo está incompleto y debes rehacerlo antes de seguir. Esta verificación es exactamente lo que hace un administrador antes de confiar en una copia de seguridad.

Ahora el desastre simulado: borra por accidente (a propósito, para el ejercicio) la carpeta de documentos.

```
rm -rf ~/organizador/documentos
ls ~/organizador      # documentos ya no está: el desastre ocurrió
```

Siente el micro-pánico: es real aunque sea simulado. Ahora ejecuta el plan de recuperación, paso a paso, sin improvisar:

```
ls ~/respaldos-organizador-2026-09-20/documentos      # 1. confirma que el respaldo
tiene lo que necesitas
cp -r ~/respaldos-organizador-2026-09-20/documentos ~/organizador/  # 2. restaura
ls ~/organizador/documentos      # 3. verifica que volvió todo
find ~/organizador/documentos -type f | wc -l          # 4. cuenta: ¿coincide con lo
esperado?
```

Verificación: la carpeta `documentos` debe estar de vuelta con sus 6 archivos. Anota en tu `LEEME.txt` una sección nueva: "Plan de recuperación ante desastres", con los 4 pasos numerados. Un sistema de organización sin plan de recuperación es un castillo de naipes: se ve bien hasta que sopla el viento.

Y la lección más valiosa del simulacro: mide cuánto tardaste en recuperarte. En empresas reales eso se llama RTO (tiempo objetivo de recuperación) y es una métrica que se reporta. La tuya hoy fue de unos minutos para 6 archivos; imagina hacerlo sin respaldo y sin práctica.

Checklist de entrega

```
test -d ~/organizador/documentos && echo "carpetas OK"
find ~/organizador -name "*.bak" | wc -l      # debe ser 0: limpieza completa
ls ~/respaldos-organizador-2026-09-20        # el respaldo existe
cat ~/organizador/LEEME.txt | wc -l           # el LEEME tiene contenido
```

Reto final (sin solución, te toca a ti)

Sin mirar tus notas: 1) crea en tu casa una estructura `reto/{2024,2025}/{ene,feb,mar}` con un solo comando; 2) dentro de `reto` crea 6 archivos vacíos llamados `foto-1.jpg` hasta `foto-6.jpg` usando comodines... no, espera: `touch` no expande rangos con comodines. ¿Cómo crearías esos 6 archivos sin escribir los 6 nombres? (pista: las llaves `{1..6}` son expansión del shell, no comodines); 3) mueve las fotos de los meses impares a una carpeta `seleccionadas` usando UN solo comando con un patrón. Si lo logras, dominas la diferencia entre comodines y expansión del shell, que es tema de entrevista técnica.

Práctica

Llegó la hora de ensuciarte las manos. Haz estos ejercicios dentro de `~/practica-linux`, sin mirar la solución hasta haberlo intentado de verdad.

Ejercicio 1: estructura de proyecto

Enunciado: crea de un solo comando la estructura `proyecto/{src,docs,tests}` dentro de tu zona de práctica, y dentro de `src` crea tres archivos vacíos: `main.sh`, `util.sh` y `config.txt`. Verifica la estructura completa con un solo comando.

Solución explicada paso a paso. Paso 1: ejecuta `mkdir -p ~/practica-linux/proyecto/{src,docs,tests}`: la `-p` crea los padres y las llaves expanden las tres

subcarpetas. Paso 2: ejecuta ``touch ~/practica-linux/proyecto/src/main.sh ~/practica-linux/proyecto/src/util.sh ~/practica-linux/proyecto/src/config.txt``: touch acepta varios archivos en un solo comando. Paso 3: verifica con ``ls -R ~/practica-linux/proyecto``: debes ver proyecto con src, docs y tests, y dentro de src los tres archivos. Paso 4: si algo faltó, repite el paso correspondiente: -p no se queja si la carpeta ya existe, así que puedes reintentar sin miedo.

Ejercicio 2: mudanza con comodines

Enunciado: en tu zona de práctica crea 6 archivos: ``a1.log`, `a2.log`, `b1.txt`, `b2.txt`, `c.tmp`, `d.tmp``. Mueve todos los .log a una carpeta ``logs/``, copia (no muevas) los .txt a ``documentos/``, y borra los .tmp verificando antes con ls. Muestra el resultado final.

Solución explicada paso a paso. Paso 1: crea los archivos con ``touch a1.log a2.log b1.txt b2.txt c.tmp d.tmp`` dentro de `~/practica-linux`. Paso 2: ``mkdir logs`` y luego ``mv *.log logs/``: los .log desaparecen de aquí y aparecen en logs. Paso 3: ``cp *.txt documentos/``: los .txt ahora están en ambos lugares, porque cp copia y mv mueve. Verifica con ``ls *.txt`` (siguen aquí) y ``ls documentos/*.txt`` (también allá). Paso 4: ``ls *.tmp`` para inspeccionar las víctimas, y luego ``rm -i *.tmp`` confirmando cada borrado. Paso 5: ``ls -R`` para ver el estado final: logs con dos archivos, documentos con los txt más los de antes, y ningún .tmp. La lección clave: mv traslada, cp duplica; y antes de rm con comodín, siempre ls primero.

Ejercicio 3: lectura selectiva

Enunciado: crea un archivo ``numeros.txt`` con los números del 1 al 50, uno por línea, usando un solo comando. Luego muestra solo las primeras 5 líneas, solo las últimas 5, y las líneas 20 a 25.

Solución explicada paso a paso. Paso 1: genera el archivo con ``seq 1 50 > ~/practica-linux/numeros.txt``: seq genera secuencias de números y ``>`` los guarda en el archivo (redirección básica, teoría completa en la Unidad 4). Paso 2: ``head -5 numeros.txt`` muestra del 1 al 5. Paso 3: ``tail -5 numeros.txt`` muestra del 46 al 50. Paso 4: para las líneas 20 a 25 combina: ``head -25 numeros.txt | tail -6``: head toma las primeras 25 y tail se queda con las últimas 6 de esas, o sea de la 20 a la 25. El pipe ``|`` lo dominarás en la Unidad 4, pero ya viste su poder: encadenar comandos simples para lograr algo preciso.

Ejercicio 4: búsqueda forense

Enunciado: dentro de tu zona de práctica, encuentra todos los archivos ``*.sh`` que estén vacíos, y todos los archivos modificados hoy. Usa find con los criterios adecuados.

Solución explicada paso a paso. Paso 1: ``find ~/practica-linux -type f -name "*.sh" -empty``: combina tipo archivo, patrón de nombre y la prueba `-empty` (vacío). Debería encontrar `main.sh` y `util.sh` del ejercicio 1. Paso 2: ``find ~/practica-linux -type f -mtime 0``: `-mtime 0` significa modificados en las últimas 24 horas, o sea "hoy". Paso 3: si quieres ver detalles, agrega ``-ls`` al final del find: ``find ~/practica-linux -type f -name "*.sh" -ls`` muestra el formato largo de cada hallazgo. Paso 4: reflexiona: cada criterio que agregas es un filtro más fino. find lee como una frase: "busca aquí, que sea archivo, que se llame así, y que cumpla esto".

Ejercicio 5: limpieza segura

Enunciado: tu zona de práctica se llenó de archivos de prueba. Diseña y ejecuta una limpieza segura: primero lista todo lo que vas a borrar, bórralo con confirmación, y verifica que tus carpetas importantes (proyecto, documentos, scripts) sigan intactas.

Solución explicada paso a paso. Paso 1: decide qué es basura: los `.tmp`, los archivos de prueba sueltos, pero NUNCA las carpetas del proyecto. Paso 2: ``ls ~/practica-linux/*.tmp ~/practica-linux/numeros.txt`` para ver exactamente las víctimas. Paso 3: ``rm -i ~/practica-linux/*.tmp ~/practica-linux/numeros.txt`` y confirma uno por uno. Paso 4: ``ls -R ~/practica-linux/proyecto`` y ``ls ~/practica-linux/documentos`` para verificar que lo importante sigue ahí. Paso 5: escribe en tu cheat sheet la regla de oro: "ls antes de rm, -i cuando dudes, y nunca rm -rf fuera de mi zona de práctica". Este ejercicio no evalúa velocidad, evalúa disciplina: el profesional no es el que borra más rápido, es el que nunca borra lo que no debe.

Tarjeta de referencia rápida

Tu acordeón de la unidad: creación, copia, movimiento, borrado, visualización y búsqueda de archivos. Tenlo a la mano mientras practicas.

Comandos clave de la unidad

Comando	Descripción
<code>`mkdir carpeta`</code>	Crea una carpeta nueva.
<code>`mkdir -p a/b/c`</code>	Crea toda la ruta de carpetas de un golpe, aunque los papás no existan.
<code>`touch archivo`</code>	Crea un archivo vacío; si ya existe, actualiza su fecha sin tocar el contenido.
<code>`cp origen destino`</code>	Copia un archivo. Con <code>`-i`</code> pregunta antes de sobrescribir.
<code>`cp -r origen destino`</code>	Copia carpetas con todo su contenido, recursivamente.
<code>`mv origen destino`</code>	Mueve o renombra: la misma operación según el contexto. Con <code>`-i`</code> pregunta antes de sobrescribir.
<code>`rm archivo`</code>	Borra un archivo. Sin papelera, sin deshacer.
<code>`rm -r carpeta`</code>	Borra una carpeta con todo su contenido.
<code>`rm -rf carpeta`</code>	Borra sin preguntar ni perdonar: úsalo con disciplina extrema.
<code>`rmdir carpeta`</code>	Borra solo carpetas vacías. La versión prudente.
<code>`cat`</code>	

archivo` | Muestra el contenido completo de archivos cortos. Con `-n` numera líneas. | | `less
archivo` | Navega archivos largos por partes (`q` sale, `/` busca). | | `head archivo` | Muestra
las primeras 10 líneas. Con `-n 20` ajustas cuántas. | | `tail archivo` | Muestra las últimas 10
líneas. Con `-n 20` ajustas cuántas. | | `tail -f archivo` | Vigila un archivo en vivo: muestra líneas
nuevas conforme llegan. | | `find ruta -name "patrón"` | Busca por nombre (el patrón siempre
entre comillas). | | `find ruta -type f` | Busca solo archivos; con `-type d` solo carpetas. | | `find
ruta -size +100M` | Busca archivos de más de 100 MB (también `k`, `G`). | | `find ruta -mtime
-7` | Busca modificados en los últimos 7 días (`+30` = hace más de 30 días). | | `find ruta
-name "*.log" -delete` | Busca y borra lo encontrado: potentísimo, inspecciona primero. | | `file
archivo` | Te dice qué tipo de archivo es realmente. Úsalo antes de `cat` con desconocidos. | |
`du -sh carpeta` | Muestra cuánto pesa una carpeta en total, en formato legible. | | `wc -l
archivo` | Cuenta las líneas de un archivo (adelanto de la Unidad 4, útil desde ya). |

Patrones de comodines

| Patrón | Descripción | |---|---| | `\*` | Cero o más caracteres cualesquiera. `\*.txt` = todo lo que
termine en .txt. | | `?` | Exactamente un carácter cualquiera. `foto-?.jpg` = foto-1.jpg pero no
foto-10.jpg. | | `[abc]` | Un carácter del conjunto. `[aeiou]\*` = lo que empiece con vocal. | | `[0-9]`
| Un dígito. `reporte-202[0-9].txt` = reporte-2020.txt a reporte-2029.txt. | | `[a-z]` | Una letra
minúscula. Útil para filtrar nombres. | | `{1..6}` | Expansión de rango (no es comodín): genera 1
2 3 4 5 6. | | `{a,b,c}` | Expansión de llaves: genera a b c. `mkdir -p p/{a,b,c}` crea tres carpetas.
|

One-liners que te salvan el día

```
mkdir -p ~/proyectos/nuevo/{src,docs,tests} && cd ~/proyectos/nuevo
```

Crea la estructura estándar de un proyecto (código, documentación y pruebas) y entra en ella
de un golpe. El esqueleto que todo proyecto debería tener al nacer.

```
ls *.log 2>/dev/null || echo "no hay logs"
```

Lista los archivos `.log` si existen; si no hay ninguno, en vez de un error feo muestra un
mensaje amable. (El `2>/dev/null` esconde errores: lo verás a fondo en la Unidad 4.)

```
find . -name "*.tmp" -delete && echo "limpieza completa"
```

Borra todos los archivos temporales bajo la carpeta actual y confirma. Úsalo solo después de inspeccionar con ``find . -name "*.tmp"`` sin el ``-delete``.

```
cp -r proyecto proyecto.bak-$(date +%F)
```

Copia de seguridad de tu proyecto con la fecha de hoy en el nombre (ej: ``proyecto.bak-2026-09-20``). El respaldo rápido antes de un cambio riesgoso.

```
find /var/log -type f -size +100M | head -10
```

Encuentra los 10 archivos de registro más grandes de más de 100 MB. El primer comando que ejecuta un profesional cuando el disco se llena sin explicación.

Recetas de ``find`` por criterio (guárdalas, las vas a usar toda la vida)

``find`` es el comando más expresivo de esta unidad porque combina criterios. Estas son las recetas que un administrador usa cada semana:

```
find ~/proyectos -name "*.py" -mtime -3
```

Archivos Python modificados en los últimos 3 días bajo tus proyectos: para ver en qué estuviste trabajando esta semana.

```
find /tmp -type f -atime +10 -delete
```

Borra de ``/tmp`` los archivos que nadie ha abierto en más de 10 días. Limpieza segura de temporales abandonados (inspecciona sin ``-delete`` primero).

```
find . -type f -empty
```

Encuentra archivos vacíos (0 bytes) en la carpeta actual. Perfectos candidatos para borrar o señal de que algo falló al escribirlos.

```
find ~/descargas -type f -size +500M
```

Los archivos de más de 500 MB en tus descargas: los sospechosos habituales cuando el disco se llena.

```
find /etc -name "*.conf" | wc -l
```

Cuenta cuántos archivos de configuración `.conf` hay en `/etc`. Te da una idea del tamaño real de la configuración de un sistema.

Combinaciones de comandos de uso diario

```
mkdir -p respaldo-$(date +%F) && cp -r documentos/ respaldo-$(date +%F)/
```

Crea una carpeta con la fecha de hoy y respalda tus documentos en ella, todo en una línea. El hábito de "respaldo fechado antes del cambio" en su forma más compacta.

```
ls -lhS | head -10
```

Los 10 archivos más pesados de tu carpeta actual, de mayor a menor. Tu radar para cazar qué se está comiendo el disco.

```
cat LEEME.txt README.md 2>/dev/null | head -40
```

Muestra el contenido de los archivos de documentación más comunes (si existen) y enseña las primeras 40 líneas. Así se revisa un proyecto desconocido en segundos.

Resumen de la unidad

- `mkdir` crea carpetas; `mkdir -p` crea rutas completas de un golpe; `touch` crea archivos vacíos.
- `cp` copia (origen y destino); para carpetas siempre agrega `-r`; `-i` pregunta antes de sobrescribir.
- `mv` mueve y renombra: es la misma operación; renombrar es mover al mismo lugar con otro nombre.
- `rm` borra sin papelera: `rm -rf` es potentísimo y peligroso; verifica con `ls` antes de borrar.
- `cat` muestra archivos cortos, `less` navega archivos largos (sal con `q`), `head` y `tail` muestran los extremos.
- `tail -f` vigila un archivo en vivo: el electrocardiograma del sistema para administradores.
- Los comodines `*` (todo) y `?` (un carácter) aplican un comando a grupos de archivos de un golpe.

- Antes de `rm` con comodín, inspecciona con `ls` usando el mismo patrón: tu seguro de vida.
- `find` busca con criterios combinables: nombre, tipo, tamaño, fecha. `locate` busca instantáneo por nombre.
- Trabaja siempre en tu zona de práctica y confirma con `ls` y `pwd`: el profesional verifica, no supone.
- `mkdir -p a/b/c` crea toda la rama de carpetas de un golpe aunque las carpetas papá no existan.
- `touch` también sirve para actualizar la fecha de un archivo existente sin tocar su contenido.
- `cp` sin `-r` se niega a copiar carpetas: el `-r` es obligatorio cuando el origen es un directorio.
- `mv -i` pregunta antes de sobrescribir: el cinturón de seguridad al mover y renombrar.
- `rmdir` solo borra carpetas vacías: la versión prudente cuando no quieres arriesgar.
- `cat -n` numera las líneas del archivo: útil para referenciar partes exactas al pedir ayuda.
- `head -n 20` y `tail -n 20` controlan cuántas líneas ves: ajusta la ventana a lo que necesites.
- `[abc]` elige un carácter del conjunto y `[0-9]` un dígito: comodines de precisión quirúrgica.
- En `find`, el patrón de `-name` siempre va entre comillas (`-name "*.log"`): que lo interprete `find`, no el shell.

UNIDAD 3: Permisos y propiedad

Ya sabes crear y manipular archivos. Ahora viene la unidad que separa a los usuarios de los administradores: los permisos. En Linux, cada archivo tiene un dueño, un grupo y un juego de reglas que dice quién puede leerlo, modificarlo y ejecutarlo. Entender este sistema es entender la seguridad de Linux desde sus cimientos.

En esta unidad vas a aprender a leer la cadena de permisos (ese `-rwxr-xr-x` que ves en `ls -l`), a cambiar permisos con `chmod` en modo simbólico y en modo octal, a cambiar dueños y grupos con `chown` y `chgrp`, a entender usuarios y grupos, y a usar `sudo` para actuar como superusuario sin vivir como root. También verás `umask`, que define los permisos con los que nacen los archivos nuevos.

Por qué importa tanto esta unidad: los permisos son el sistema inmunológico de Linux. Un servidor web funciona porque los archivos tienen los permisos correctos: el servidor puede leer las páginas pero los visitantes no pueden modificarlas. Un script funciona porque tiene permiso de ejecución. La mayoría de los errores de "permiso denegado" que vas a ver en tu vida se resuelven entendiendo esta unidad, no copiando comandos mágicos de internet.

Hay otra razón: el error más común y peligroso del principiante es el `chmod 777`. Cuando algo no funciona por permisos, muchos le ponen 777 (todos pueden todo) y "se arregla". Eso es como quitarle la puerta a tu casa porque perdiste la llave: funciona, pero cualquiera puede entrar. Aquí vas a aprender a poner los permisos correctos, no a tumbar las paredes.

Cómo encaja en el curso: las Unidades 1 y 2 te dieron navegación y manipulación; esta te da el control de acceso. En la Unidad 6 harás tus scripts ejecutables con `chmod +x`. En la Unidad 9 administrarás procesos y tareas que corren con distintos usuarios. Y en el proyecto final, tu script de respaldo necesitará los permisos correctos para leer lo que respalda y escribir donde guarda.

Empecemos. Al terminar vas a leer `ls -l` como un libro abierto y a configurar permisos con precisión quirúrgica.

Explicación detallada

Leyendo la cadena de permisos

Ejecuta `ls -l` y mira la primera columna de cualquier archivo:

```
-rw-r--r-- 1 esteban esteban 220 ene 15 10:30 notas.txt
-rwxr-xr-x 1 esteban esteban 856 ene 15 10:31 respaldo.sh
```

Esos 10 caracteres son la historia de seguridad del archivo. El primero dice el tipo: `-` es archivo normal, `d` es directorio, `l` es enlace simbólico. Los 9 siguientes son tres grupos de tres: permisos del dueño (usuario), del grupo y de los demás (otros).

Cada grupo tiene tres letras: `r` (read, leer), `w` (write, escribir/modificar) y `x` (execute, ejecutar). Un guion `-` significa que ese permiso está negado. Así, `rw-r--r--` se lee: el dueño puede leer y escribir, el grupo solo leer, los demás solo leer. Y `rwxr-xr-x`: el dueño puede todo, el grupo y los demás pueden leer y ejecutar pero no modificar.

Para directorios, los permisos significan algo ligeramente distinto: `r` es ver qué hay dentro (listar), `w` es crear/borrar archivos dentro, y `x` es poder entrar al directorio (hacer cd) y acceder a su contenido. Un directorio con `r` pero sin `x` es como una vitrina: ves qué hay pero no puedes tocar nada.

Las columnas siguientes también importan: el dueño (esteban), el grupo (esteban), el tamaño, la fecha y el nombre. Dueño y grupo son los protagonistas de la siguiente sección.

chmod simbólico: permisos con letras

chmod significa "change mode", cambiar los permisos. El modo simbólico usa letras para decir a quién (u=usuario/dueño, g=grupo, o=otros, a=todos) qué operación (+ agregar, - quitar, = asignar exacto) y qué permiso (r, w, x):

```
chmod u+x script.sh
```

Le da permiso de ejecución al dueño. El script que antes era un texto inerte ahora se puede ejecutar.

```
chmod go-w notas.txt
```

Le quita el permiso de escritura al grupo y a los demás: ahora solo el dueño puede modificarlo.

```
chmod a+r documento.txt
```

Le da permiso de lectura a todos (a = all). Útil para archivos que quieres compartir.

```
chmod u=rwx,g=rx,o=r script.sh
```

El ``='` asigna los permisos exactos: dueño todo, grupo leer y ejecutar, otros solo leer. Es la forma más clara cuando quieres dejar los permisos en un estado conocido, sin importar cómo estaban antes.

chmod octal: permisos con números

El modo octal representa los mismos permisos con tres dígitos, uno por grupo (dueño, grupo, otros). Cada permiso vale un número: r=4, w=2, x=1. Sumas los que quieres activos: 7 es rwx (4+2+1), 6 es rw- (4+2), 5 es r-x (4+1), 4 es r-- (4 solo), 0 es --- (nada).

```
chmod 755 script.sh
```

755 significa: dueño 7 (rwx, todo), grupo 5 (r-x, leer y ejecutar), otros 5 (r-x). Es el permiso clásico de scripts y programas: el dueño controla, los demás usan.

```
chmod 644 documento.txt
```

644 significa: dueño 6 (rw-, leer y escribir), grupo 4 (r--), otros 4 (r--). Es el permiso clásico de archivos de datos: el dueño edita, los demás solo leen.

Memoriza estos dos: 755 para ejecutables, 644 para archivos normales. Con esos dos resuelves el 90 por ciento de los casos. Y recuerda la regla de oro: nunca 777 (rwxrwxrwx, todos pueden todo) salvo que sepas exactamente por qué y sea temporal. El 777 es la puerta quitada de la casa.

Tabla de referencia rápida: 7=rwx, 6=rw-, 5=r-x, 4=r--, 3=-wx, 2=-w-, 1=-x, 0=---. En la práctica solo usarás 7, 6, 5, 4 y 0.

chown y chgrp: cambiar dueño y grupo

chown significa "change owner": cambia el dueño de un archivo. chgrp ("change group") cambia el grupo:

```
sudo chown ana notas.txt
sudo chgrp equipo notas.txt
```

Ahora el archivo es de ana y del grupo equipo. Estos comandos normalmente requieren sudo porque cambiar dueños es una operación administrativa: si cualquiera pudiera adueñarse de archivos ajenos, el sistema de seguridad no serviría de nada.

La opción -R los hace recursivos para carpetas completas:

```
sudo chown -R ana:equipo /srv/proyecto
```

La sintaxis ``usuario:grupo`` cambia ambos de un golpe. Esto lo usarás cuando despliegues aplicaciones: los archivos del sitio web deben pertenecer al usuario del servidor web, por ejemplo.

Usuarios, grupos y sudo

Linux es multiusuario desde su nacimiento: varias personas pueden usar la misma máquina, cada una con su cuenta, sus archivos y sus permisos. Los grupos agrupan usuarios para darles permisos en conjunto: en lugar de dar permiso a 20 personas una por una, creas el grupo "contadores" y le das permiso al grupo.

El superusuario se llama root: tiene poder total sobre el sistema, sin restricciones. Vivir como root es peligroso porque no hay red de seguridad: un error tipográfico puede destruir el sistema. Por eso el modelo moderno es sudo: trabajas como usuario normal y solo elevas privilegios para comandos específicos:

```
sudo apt update
```

sudo te pide tu contraseña (no la de root) y ejecuta ese comando con poderes de administrador. Es como pedir la llave maestra solo para abrir una puerta específica, en lugar de cargarla todo el día en el bolsillo.

Comandos útiles: ``id`` muestra tu usuario y tus grupos; ``whoami`` tu nombre; ``groups`` tus grupos. Para crear usuarios y grupos (con sudo): ``sudo adduser ana``, ``sudo addgroup equipo``, ``sudo usermod -aG equipo ana`` (mete a ana al grupo equipo).

umask: los permisos de nacimiento

Cuando creas un archivo nuevo, ¿qué permisos trae? Los decide umask, una máscara que le quita permisos a los valores base (666 para archivos, 777 para directorios). Con el umask típico de 022: los archivos nacen con 644 (666 menos 022) y los directorios con 755 (777 menos 022). Por eso tus archivos nuevos ya vienen con permisos sensatos sin que hagas nada.

```
umask
```

Te muestra tu máscara actual (normalmente 0022). Si algún día tus archivos nuevos nacen con permisos raros, el culpable es umask. No necesitas cambiarlo en este curso, pero sí entender que existe: es la razón por la que no tienes que hacer chmod a cada archivo que creas.

Ejemplos paso a paso

Ejemplo 1: del texto inerte al script ejecutable

Vas a vivir el ciclo completo: crear un script, ver por qué no corre, y darle vida con chmod.

```
cd ~/practica-linux
echo 'echo "Hola desde mi primer script"' > hola.sh
ls -l hola.sh
```

El archivo nace con `-rw-r--r--`: se puede leer y escribir, pero no ejecutar. Intenta correrlo:

```
./hola.sh
```

El shell responde "permiso denegado". No es un error del script: es el sistema de permisos haciendo su trabajo. El archivo no tiene la x de ejecución.

```
chmod u+x hola.sh
ls -l hola.sh
./hola.sh
```

Ahora la cadena dice `-rwxr--r--` y el script corre: imprime "Hola desde mi primer script". El `./` al inicio le dice al shell "ejecuta el archivo que está aquí mismo": es necesario porque la carpeta actual no está en la ruta de búsqueda de programas (medida de seguridad que veremos en la Unidad 6).

Explicación: este es el ritual que repetirás con cada script del curso: escribir, dar permiso de ejecución, correr. El "permiso denegado" dejará de asustarte: será una señal clara de "falta la x", no un misterio.

Ejemplo 2: laboratorio de permisos

Vas a experimentar con combinaciones para entender qué hace cada permiso en la práctica.

```
cd ~/practica-linux
echo "contenido secreto" > secreto.txt
chmod 600 secreto.txt
ls -l secreto.txt
```

600 significa: solo el dueño puede leer y escribir; grupo y otros, nada. Es el permiso de los archivos privados, como tus llaves SSH.

```
chmod 644 secreto.txt
chmod u-w secreto.txt
ls -l secreto.txt
echo "intento escribir" >> secreto.txt
```

Primero lo dejas en 644, luego le quitas la escritura al dueño con `u-w`: queda `r--r--r--`. El intento de agregar texto falla con "permiso denegado": ni el dueño puede modificar un archivo sin permiso w. Los permisos mandan incluso sobre el dueño.

```
chmod u+w secreto.txt
echo "ahora sí" >> secreto.txt
cat secreto.txt
```

Devuelves la escritura y el texto entra. Moraleja: los permisos no son sugerencias, son reglas que el sistema hace cumplir siempre.

Explicación: este laboratorio te enseña a pensar en permisos como estados. Cuando algo falle por permisos en el futuro, tu diagnóstico será: `ls -l`, leer la cadena, identificar qué falta y agregarlo con precisión, en lugar del martillazo del 777.

Ejemplo 3: el caso del 777 (y su arreglo correcto)

Vas a reproducir el escenario clásico: un programa que no puede escribir, la "solución" perezosa y la solución correcta.

```
cd ~/practica-linux
mkdir -p sitio && echo "<h1>Hola</h1>" > sitio/index.html
chmod 755 sitio
chmod 644 sitio/index.html
ls -l sitio/
```

Simulas una carpeta de sitio web con permisos correctos: la carpeta 755, el archivo 644.

```
chmod 555 sitio
touch sitio/nuevo.txt
```

Quitamos el permiso w de la carpeta (555 = r-xr-xr-x): ahora nadie puede crear archivos dentro, y el touch falla. Este es el "no funciona" que frustra a los principiantes.

La solución perezosa sería `chmod 777 sitio`: cualquiera puede escribir, incluidos atacantes. NO lo hagas. La solución correcta es devolver la escritura solo al dueño:

```
chmod 755 sitio
touch sitio/nuevo.txt
ls -l sitio/
```

El archivo se crea sin regalar permisos. Problema resuelto sin tumbar la puerta.

Explicación: cada vez que sientas la tentación del 777, pregúntate: ¿quién necesita escribir aquí realmente? Casi siempre la respuesta es "solo el dueño" o "solo un grupo específico". El permiso correcto es el mínimo que hace funcionar las cosas: ese es el principio del menor privilegio, la base de la seguridad.

Ejemplo 4: detective de propiedad

Vas a investigar quién es dueño de qué en tu sistema, como harías en un servidor real.

```
ls -l /etc/hostname /etc/passwd
```

Ves que los archivos de configuración del sistema pertenecen a root. Tú puedes leerlos pero no modificarlos: por eso existen.

```
id
groups
```

Ves tu identidad completa: usuario, grupo principal y grupos secundarios. Si instalaste WSL o Ubuntu, probablemente tu usuario está en el grupo sudo, que es el que permite usar sudo.

```
sudo ls /root
```

Con sudo puedes asomarte a la carpeta personal de root, normalmente prohibida. El sistema te pide tu contraseña la primera vez y luego la recuerda un rato.

```
ls -l $(which sudo)
```

`which sudo` te dice dónde está el programa sudo, y `ls -l` sobre esa ruta muestra algo especial: en lugar de x hay una `s` en el permiso del dueño (`-rwsr-xr-x`). Esa `s` es el bit setuid: hace que sudo se ejecute con los poderes de su dueño (root) aunque lo invoque un usuario normal. Es el mecanismo que hace posible todo el sistema sudo. No necesitas memorizarlo, pero ahora sabes que existe y por qué sudo funciona.

Explicación: la propiedad y los grupos son el mapa del poder en Linux. Cuando administres servidores, este mapa te dirá quién puede tocar qué, y sudo será tu llave maestra controlada.

Errores comunes y casos reales

Los permisos son el tema donde los principiantes cometen los errores más caros: un `chmod` mal aplicado puede dejar un servidor expuesto o inservible. La buena noticia es que casi todos los errores de permisos se diagnostican igual: lees la cadena de `ls -l`, identificas cuál trío te aplica (dueño, grupo u otros) y actúas con precisión quirúrgica en vez de a martillazos.

Caso 1: "Permiso denegado" al ejecutar tu propio script

Síntoma: escribes tu primer script `hola.sh`, lo intentas correr con `./hola.sh` y la terminal responde `bash: ./hola.sh: Permiso denegado`. El archivo es tuyo, lo acabas de crear, y aun así no te deja.

Por qué ocurre: los archivos nuevos nacen SIN permiso de ejecución, por seguridad. Linux no asume que todo archivo de texto quiere ser programa. La `x` hay que otorgarla explícitamente, y eso es una característica, no un bug.

Solución: otorga el permiso de ejecución al dueño y vuelve a intentarlo.

```
ls -l hola.sh          # mira: -rw-r--r--, no hay ninguna x
chmod u+x hola.sh      # otorga ejecución al dueño (u = user)
ls -l hola.sh          # ahora: -rwxr--r--, ya tiene la x
./hola.sh              # ahora sí se ejecuta
```

Lección profesional: "Permiso denegado" al ejecutar algo tuyo significa falta la `x` el 95% de las veces. El diagnóstico completo es siempre `ls -l` primero: si no hay `x` donde la necesitas, ya sabes qué hacer.

Caso 2: arreglar todo con `chmod 777` (el pecado capital)

Síntoma: algo no funciona por permisos, buscas en internet, y la "solución" más votada dice `chmod -R 777`. Lo aplicas, funciona... y acabas de quitarle la puerta a tu casa: ahora CUALQUIER usuario del sistema puede leer, modificar y ejecutar esos archivos.

Por qué ocurre: el 777 es la salida fácil porque elimina todas las restricciones en vez de entender cuál era el problema real. Funciona como "arreglo", pero a cambio de la seguridad completa. En un servidor compartido o público, es una invitación abierta a que modifiquen o infecten tus archivos.

Solución: usa el permiso mínimo necesario. Para un script que solo tú ejecutas: `755` si otros deben leerlo, `700` si es solo tuyo. Para archivos de configuración: `644` o `640`.

```
chmod -R 777 mi-sitio/      # NUNCA hagas esto en algo real
chmod -R 755 mi-sitio/      # BIEN: tú controlas, otros leen y ejecutan
chmod 700 secreto.txt       # BIEN: solo tú puedes verlo
chmod 644 documento.txt     # BIEN: archivo normal, tú escribes, otros leen
```

Lección profesional: cada vez que sientas la tentación del 777, detente y pregunta: ¿quién necesita QUÉ permiso exactamente? El profesional otorga lo mínimo necesario, no lo máximo posible. El 777 es confesión de que no se entendió el problema.

Caso 3: `chmod -R` en la ruta equivocada (el desastre silencioso)

Síntoma: querías cambiar permisos de `~/proyecto` y escribiste `sudo chmod -R 777 /` por un error de dedo, o aplicaste el recursivo estando en la carpeta equivocada. El sistema empieza a comportarse raro: nada funciona como antes, servicios que no arrancan, `sudo` que se queja.

Por qué ocurre: `-R` aplica el cambio a TODO el árbol bajo la ruta indicada, y la raíz `/` es todo el sistema. Cambiar permisos de archivos del sistema rompe la seguridad y el funcionamiento de programas que esperan permisos específicos.

Solución: no hay botón de deshacer. La prevención es triple: 1) verifica con `pwd` dónde estás antes del comando; 2) prueba primero SIN `-R` sobre un archivo; 3) nunca combines `sudo` + `-R` + rutas absolutas sin haber verificado la ruta con `ls` antes.

```
pwd                      # ¿dónde estoy parado?
ls -ld carpeta-objetivo  # ¿esta es la carpeta correcta? (-d la muestra a
                          # ella, no su contenido)
chmod 755 carpeta-objetivo/archivo.txt  # prueba en UN archivo primero
chmod -R 755 carpeta-objetivo           # solo entonces, el recursivo
```

Lección profesional: los administradores veteranos le tienen respeto reverencial a ``chmod -R`` con ``sudo``. Algunos incluso escriben la ruta con Tab para que la terminal la complete y no haya errores de dedo. La prisa no es compatible con el recursivo.

Caso 4: ``chown`` responde "Operación no permitida"

Síntoma: intentas ``chown ana reporte.txt`` para darle un archivo a tu compañera y la terminal responde ``chown: changing ownership of 'reporte.txt': Operación no permitida``. El archivo es tuyo y aun así no puedes regalarlo.

Por qué ocurre: cambiar el dueño de un archivo es una operación privilegiada: solo root puede hacerlo. La razón es de seguridad: si cualquiera pudiera regalar archivos, podría llenar de basura la cuota de disco de otro usuario o plantar archivos maliciosos a su nombre.

Solución: usa ``sudo`` para ejecutar el cambio como administrador.

```
chown ana reporte.txt      # ERROR: operación no permitida
sudo chown ana reporte.txt # CORRECTO: como administrador sí se puede
ls -l reporte.txt         # verifica que el dueño cambió a ana
sudo chown $USER reporte.txt # para devolverlo a tu usuario si fue una prueba
```

Lección profesional: que necesites ``sudo`` para ``chown`` no es un estorbo, es el sistema protegiéndote. Y ojo: al cambiar el dueño, TÚ pierdes el control del archivo. Hazlo con intención, no por accidente.

Caso 5: archivos nuevos con permisos que no esperabas (culpa de umask)

Síntoma: creas un archivo y nace como ``-rw-----`` (solo tú puedes leerlo) cuando esperabas ``-rw-r--r-``, o al revés: querías privacidad y nació legible para todos. No cambiaste nada conscientemente.

Por qué ocurre: cada archivo nuevo recibe sus permisos de la ``umask``, una máscara que resta permisos a los valores base (666 para archivos, 777 para carpetas). Si alguien (o algún programa) cambió tu umask a ``077``, todo nace ultra privado; si está en ``000``, todo nace abierto.

Solución: revisa tu umask actual y ajústala si no es la que quieres. ``022`` es el estándar equilibrado.

```
umask          # muestra tu máscara actual, ej: 0022
touch prueba.txt && ls -l prueba.txt  # ¿con qué permisos nació?
umask 022      # establece la máscara estándar (archivos 644, carpetas 755)
touch prueba2.txt && ls -l prueba2.txt # compara: ahora nace 644
rm prueba.txt prueba2.txt
```

Lección profesional: cuando "de pronto" los archivos nacen con permisos raros, no es magia: es la umask. Revísala antes de andar corrigiendo archivo por archivo con `chmod`.

Caso 6: `sudo` pide contraseña y "no escribe nada"

Síntoma: ejecutas `sudo` por primera vez, te pide tu contraseña, la escribes... y el cursor no se mueve, no aparecen asteriscos ni puntos. Parece que el teclado no funciona y te da miedo presionar Enter.

Por qué ocurre: es completamente normal y es por seguridad: `sudo` desactiva el eco de la contraseña para que nadie pueda ver ni cuántos caracteres tiene mirando por encima de tu hombro. Tu teclado Sí está escribiendo, solo no lo muestra.

Solución: escribe tu contraseña con confianza y presiona Enter. Si es correcta, el comando se ejecuta. Si te equivocas tres veces, `sudo` se rinde y tendrás que intentarlo de nuevo.

```
sudo ls /root      # te pide contraseña: escríbela aunque no veas nada
# [escribes tu contraseña a ciegas y presionas Enter]
# funciona: ves el contenido de /root
sudo -l            # lista qué comandos PUEDES ejecutar con sudo
```

Lección profesional: todos pasamos por ese momento de "¿está escribiendo o no?". Ahora ya lo sabes, y cuando veas a un compañero nuevo paralizado frente al prompt de `sudo`, le ahorras cinco minutos de angustia con una frase.

Caso real 1: publicar un script en un servidor compartido

Tu equipo usa un servidor donde varios desarrollan. Escribes una herramienta `despliega.sh` que todos deben poder ejecutar, pero solo tú debes poder modificarla. El procedimiento profesional:

```
cp despliega.sh /opt/herramientas/      # con sudo si hace falta
sudo chown root:devops /opt/herramientas/despliega.sh
sudo chmod 755 /opt/herramientas/despliega.sh
ls -l /opt/herramientas/despliega.sh    # verifica: -rwxr-xr-x root devops
```

Resultado: cualquiera lo ejecuta (`x` para todos), solo root lo modifica, y el grupo `devops` queda registrado como responsable. Así se publican herramientas en entornos reales: dueño claro, permisos mínimos, verificación al final.

Caso real 2: carpeta de proyecto colaborativa con grupo

Tres personas trabajan en el mismo proyecto y necesitan leer y escribir en `/srv/proyecto`, pero nadie más debe entrar. La solución profesional usa grupos, no 777:

```
sudo groupadd proyecto          # crea el grupo del equipo
sudo usermod -aG proyecto ana   # agrega a cada compañero al grupo
sudo usermod -aG proyecto luis
sudo chgrp -R proyecto /srv/proyecto # el grupo dueño de todo el proyecto
sudo chmod -R 770 /srv/proyecto   # dueño y grupo: todo; otros: nada
ls -ld /srv/proyecto            # verifica: drwxrwx--- ... proyecto
```

Con `770`, el equipo colabora sin fricción y el resto del sistema ni siquiera puede entrar a curiosear. Esto es administración de permisos de verdad, y es exactamente lo opuesto al `chmod 777`.

Miniproyecto guiado de la unidad

Vas a construir una **Fortaleza de Permisos**: una estructura de carpetas que simula un pequeño servidor con zonas de distinta sensibilidad (pública, de equipo, privada y de scripts), cada una blindada con los permisos correctos. Auditarás tu fortaleza como lo haría un administrador de seguridad, provocarás intrusiones simuladas y documentarás el esquema. Al terminar sabrás diseñar permisos en vez de improvisarlos.

Paso 1: levanta las murallas (crea la estructura)

Crea la fortaleza con cuatro zonas, cada una con archivos de ejemplo dentro.

```
cd ~
mkdir -p fortaleza/{publico,equipo,privado,scripts}
cd fortaleza
touch publico/bienvenida.txt publico/precios.txt
touch equipo/plan.txt equipo/minuta.txt
touch privado/claves.txt privado/diario.txt
touch scripts/respaldo.sh scripts/limpieza.sh
find . -type f | sort
```

Verificación: ``find . -type f | sort`` debe listar los 8 archivos. Esta es tu maqueta de servidor: ``publico`` es lo que cualquiera puede ver, ``equipo`` lo del grupo de trabajo, ``privado`` solo tuyo, y ``scripts`` tus herramientas.

Paso 2: asigna permisos a cada zona según su sensibilidad

Aplica el esquema de permisos zona por zona. Piensa en cada número antes de ejecutarlo: ¿quién necesita qué?

```
chmod 755 publico && chmod 644 publico/*.txt
# público: todos entran y leen, solo tú modificas

chmod 770 equipo && chmod 660 equipo/*.txt
# equipo: solo tú y tu grupo entran y escriben, otros ni miran

chmod 700 privado && chmod 600 privado/*.txt
# privado: solo tú. Ni tu grupo, ni otros, nadie más.

chmod 755 scripts && chmod 755 scripts/*.sh
# scripts: todos pueden ejecutar tus herramientas, solo tú las modificas
```

Verificación: inspecciona el resultado completo y compáralo con el diseño.

```
ls -lR
```

Cada línea debe coincidir con el esquema: ``drwxr-xr-x`` en publico, ``drwxrwx---`` en equipo, ``drwx-----`` en privado, y ``-rwxr-xr-x`` en los scripts. Si algo no cuadra, corrígelo ahora: en seguridad, "casi bien" es "mal".

Paso 3: haz una auditoría de intruso

Ponte en los zapatos de un atacante (o de un compañero curioso): ¿a qué puedes llegar si NO eres el dueño? Como no puedes crear otro usuario fácilmente sin root, haz la auditoría por inspección: para cada zona, responde qué podría hacer alguien del grupo y alguien de fuera.

```
ls -ld publico equipo privado scripts
```

Anota en un archivo ``AUDITORIA.txt`` tus hallazgos con este formato por zona: "Zona X: un extraño puede [leer/entrar/nada] porque los permisos son [NNN]". Ejemplo: "Zona privado: un extraño no puede ni entrar (700) ni leer los archivos (600): blindaje correcto".

Verificación: tu auditoría debe cubrir las 4 zonas y en cada una identificar correctamente qué puede hacer el "otro" (el tercer trío). Si en alguna zona el "otro" puede más de lo que el diseño dice, tienes una brecha: corrígela con `chmod`.

Paso 4: el ritual del script ejecutable

Convierte tus scripts en herramientas reales: dales contenido, hazlos ejecutables y ejecútalos.

```
nano scripts/respaldo.sh
```

Escribe dentro (es un script mínimo, lo entenderás a fondo en la Unidad 6):

```
#!/bin/bash
echo "Respalando documentos..."
cp -r ~/fortaleza/documentos ~/fortaleza-respaldo 2>/dev/null || echo "nada que respaldar aún"
echo "Listo."
```

Ahora el ritual completo:

```
chmod u+x scripts/respaldo.sh # otorga ejecución
ls -l scripts/respaldo.sh     # verifica la x
./scripts/respaldo.sh         # ejecuta... ¿funciona? (pista: el ./ importa)
cd scripts && ./respaldo.sh     # prueba también desde dentro de la carpeta
```

Verificación: el script debe ejecutarse y mostrar sus mensajes. Fíjate en algo clave: `./respaldo.sh` funciona, pero `respaldo.sh` solo probablemente dice "orden no encontrada". ¿Por qué? Porque la carpeta actual no está en el PATH por seguridad: Linux no ejecuta programas del directorio actual a menos que lo pidas explícito con `./`. Anota esto en tu auditoría: es una defensa contra troyanos con nombres de comandos comunes.

Paso 5: experimenta con umask en un laboratorio aislado

Abre una subshell (un shell hijo temporal) para jugar con umask sin afectar tu configuración real: al salir, todo vuelve a la normalidad.

```

bash                # entras a una subshell: tu laboratorio
umask 022
touch normal.txt && ls -l normal.txt    # nace 644: el estándar
umask 077
touch secreto.txt && ls -l secreto.txt  # nace 600: solo tú
umask 027
mkdir carpeta-grupo && ls -ld carpeta-grupo # nace 750: grupo entra, otros no
rm normal.txt secreto.txt && rmdir carpeta-grupo
exit                # sales del laboratorio: tu umask real intacta
umask                # confirma que volvió a tu valor original

```

Verificación: debes haber visto nacer archivos con 644, 600 y una carpeta con 750, y al salir tu umask original intacta. Anota en tu `AUDITORIA.txt`: "umask 022 = estándar, 077 = paranoico, 027 = trabajo en equipo".

Paso 6: caza permisos peligrosos como un auditor

Los auditores de seguridad buscan patrones peligrosos en todo el sistema. Practica las búsquedas sobre tu fortaleza y tu casa:

```

find ~/fortaleza -perm -002 -type f    # archivos escribibles por CUALQUIERA: no
debe haber ninguno
find ~/fortaleza -perm -004 -type f    # archivos legibles por cualquiera: solo los
de publico
find ~ -maxdepth 2 -perm -002 -type f 2>/dev/null | head    # revisa tu casa: ¿hay
sorpresas?

```

Verificación: la primera búsqueda debe devolver vacío (tu fortaleza no tiene archivos world-writable). La segunda debe listar solo los de `publico`. Si la tercera muestra archivos raros en tu casa, investiga con `ls -l`: acabas de hacer tu primera auditoría de seguridad real.

Paso 7: documenta el esquema de seguridad

Crea `ESQUEMA.txt` en la raíz de la fortaleza explicando el diseño completo: cada zona, sus permisos en octal y simbólico, quién puede qué, y por qué elegiste esos números y no otros. Un esquema de permisos sin documentar se degrada en meses; documentado, se puede auditar.

```
cd ~/fortaleza
nano ESQUEMA.txt
cat ESQUEMA.txt
ls -lR
```

Verificación final: tu fortaleza está completa si tiene las 4 zonas con los permisos del diseño, los 2 scripts ejecutables, `AUDITORIA.txt` con el análisis de intruso y las notas de umask, y `ESQUEMA.txt` documentando todo. Este es el nivel de rigor que se espera en un servidor de verdad.

Paso 8: lleva el esquema a tu vida real

La fortaleza fue el laboratorio; ahora toca la realidad. Vas a auditar tu propia carpeta personal con ojos de auditor y a crear tu primera zona privada de verdad.

Primero, la auditoría de tu casa: busca archivos y carpetas con permisos demasiado abiertos.

```
find ~ -maxdepth 3 -perm -002 -type f 2>/dev/null | head -20
find ~ -maxdepth 2 -type d -perm -007 2>/dev/null | head -20
```

La primera búsqueda encuentra archivos que CUALQUIER usuario puede modificar; la segunda, carpetas donde cualquiera puede entrar, ver y borrar. Revisa cada resultado con `ls -l` y pregúntate: ¿esto DEBE estar abierto? Lo más común son archivos heredados de instalaciones o de memorias USB. Para cada uno que no tenga razón para estar abierto, ciérralo:

```
chmod 644 archivo-que-debe-ser-normal.txt
chmod 755 carpeta-que-debe-ser-normal
```

Segundo, crea tu caja fuerte personal: una carpeta `~/caja-fuerte` con permiso `700` para documentos sensibles (contraseñas, documentos escaneados, respaldos privados).

```
mkdir -p ~/caja-fuerte
chmod 700 ~/caja-fuerte
touch ~/caja-fuerte/.no-borrar
ls -ld ~/caja-fuerte
```

Verificación: ``ls -ld ~/caja-fuerte`` debe mostrar ``drwx-----``. A partir de hoy, todo lo sensible va ahí, no en el escritorio ni en descargas. Y una regla de oro para cerrar el paso: cada mes, repite la auditoría con esos dos comandos ``find``. La seguridad no es algo que se configura una vez: es un hábito que se revisa.

Anota en tu ``ESQUEMA.txt`` una última sección: "Aplicado a mi sistema real", con la fecha, cuántos archivos demasiado abiertos encontraste y qué hiciste con cada uno. Esa línea con fecha es el inicio de tu bitácora de seguridad personal.

Trampas de examen: planta errores y cázalos

El mejor entrenamiento de auditoría es plantar trampas y detectarlas. Pídele a alguien (o hazlo tú con los ojos cerrados, en serio) que aplique TRES cambios de permisos incorrectos en tu fortaleza: por ejemplo, un ``777`` en privado, quitar la ``x`` a un script, o abrir lectura a otros en un archivo del equipo. Luego, sin saber cuáles son, haz una auditoría completa y encuentra las tres.

Tu protocolo de caza, en orden:

```
cd ~/fortaleza
find . -perm -002 | head           # 1. lo escribible por cualquiera:
sospechoso siempre
ls -lR | grep -E "^-...w..."    # 2. archivos con w para "otros": casi
nunca legítimo
ls -l scripts/                    # 3. ¿todos los scripts tienen la x?
ls -ld publico equipo privado scripts # 4. ¿las carpetas conservan su diseño?
```

Espera: el paso 2 usa ``grep -E`` con un patrón entre comillas, tema de la Unidad 5, pero la idea es simple: busca líneas de ``ls -l`` donde el "otro" tenga permiso de escritura. Si prefieres no usarlo aún, el paso 1 (``find -perm -002``) ya caza lo más grave.

Verificación: debes encontrar las tres trampas y corregir cada una al valor del diseño original, anotando en ``AUDITORIA.txt`` qué encontraste, dónde y cómo lo corregiste. Repite el juego cada que quieras: con el tiempo, detectar un permiso mal puesto te tomará segundos. Esa velocidad es exactamente lo que distingue a un administrador que "sabe permisos" de uno que solo los ha leído.

Checklist de entrega

```
test -d ~/fortaleza/privado && echo "fortaleza existe"
stat -c "%a %n" ~/fortaleza/privado           # debe decir 700
stat -c "%a %n" ~/fortaleza/equipo             # debe decir 770
find ~/fortaleza -perm -002 -type f | wc -l     # debe ser 0
test -x ~/fortaleza/scripts/respaldo.sh && echo "script ejecutable OK"
ls ~/fortaleza/*.txt                           # AUDITORIA.txt y ESQUEMA.txt
presentes
```

Reto final (sin solución, te toca a ti)

Sin mirar tus notas: 1) crea un archivo `trampa.txt` en tu fortaleza con permisos `777` a propósito, luego encuentra el comando `find` exacto para cazarlo y corrígelo a `644`; 2) explica con tus palabras por qué `chmod 666 script.sh` NO lo hace ejecutable aunque "tiene todos los permisos de lectura y escritura"; 3) un compañero te dice "me da permiso denegado al entrar a tu carpeta equipo aunque soy del grupo": lista las 3 cosas que revisarías, en orden, para diagnosticarlo. Si respondes las tres sin dudar, los permisos ya son tu territorio.

Práctica

Haz estos ejercicios dentro de `~/practica-linux`. La regla de esta unidad: antes de cada `chmod`, predice qué va a pasar; después verifica con `ls -l`.

Ejercicio 1: lectura de cadenas

Enunciado: ejecuta `ls -l` en tu carpeta personal y elige 5 archivos o carpetas. Para cada uno escribe: tipo (archivo/directorio), qué puede hacer el dueño, qué puede hacer el grupo y qué pueden hacer los demás, todo en palabras.

Solución explicada paso a paso. Paso 1: ejecuta `ls -l ~` y elige 5 líneas variadas (algunas con `d` al inicio, otras con `-`). Paso 2: para cada línea, separa los 10 caracteres: el primero es el tipo, los siguientes 9 son tres tríos. Paso 3: traduce cada trío: por ejemplo `rwx` = "leer, escribir y ejecutar", `r--` = "solo leer", `---` = "nada". Paso 4: escribe tus 5 traducciones en tu cheat sheet. Paso 5: verifica una predicción práctica: elige un archivo con `r--` para otros e intenta leerlo (podrás) e intenta escribir en él como dueño quitándote antes la `w` (no podrás). Traducir cadenas a palabras es la habilidad base: si no puedes leer los permisos, no puedes administrarlos.

Ejercicio 2: el script que sí corre

Enunciado: crea un archivo ``saludo.sh`` que contenga un mensaje con `echo`, hazlo ejecutable solo con modo simbólico, ejecútalo, luego quítale el permiso de ejecución al grupo y a los demás con modo octal dejándolo en 744, y verifica cada paso con `ls -l`.

Solución explicada paso a paso. Paso 1: ``echo 'echo "Saludos desde Linux"' > ~/practica-linux/saludo.sh``. Paso 2: ``chmod u+x ~/practica-linux/saludo.sh`` y verifica con ``ls -l``: debe decir ``-rwxr--r--``. Paso 3: ejecútalo con ``~/practica-linux/saludo.sh`` o entrando a la carpeta y ``./saludo.sh``. Paso 4: ``chmod 744 ~/practica-linux/saludo.sh``: 7 para el dueño (rwx), 4 y 4 para grupo y otros (r--). Verifica: ``-rwxr--r--``. Espera, eso ya estaba así: el ejercicio te enseña que 744 es exactamente "dueño todo, demás solo leer". Paso 5: anota en tu cheat sheet: "script personal ejecutable = 744 o 755; la diferencia es si los demás pueden ejecutarlo (755) o solo leerlo (744)".

Ejercicio 3: caja fuerte personal

Enunciado: crea una carpeta ``privado/`` con un archivo ``clave.txt`` dentro, y configúralos de modo que solo tú puedas ver y modificar su contenido (nadie más puede ni listarla). Verifica intentando entrar como "otro" no es posible simularlo fácilmente, así que verifica con `ls -l` que los permisos sean los correctos.

Solución explicada paso a paso. Paso 1: ``mkdir ~/practica-linux/privado`` y ``echo "mi clave secreta" > ~/practica-linux/privado/clave.txt``. Paso 2: ``chmod 700 ~/practica-linux/privado``: la carpeta queda ``rwx-----``: solo el dueño entra y lista. Paso 3: ``chmod 600 ~/practica-linux/privado/clave.txt``: el archivo queda ``rw-----``: solo el dueño lee y escribe. Paso 4: verifica con ``ls -ld ~/practica-linux/privado`` (la `-d` muestra la carpeta misma, no su contenido) y ``ls -l ~/practica-linux/privado/``. Paso 5: anota la receta: "carpeta privada = 700, archivo privado = 600". Esta es la configuración de tu carpeta `~/ssh` en la vida real: si esos permisos están mal, SSH se niega a funcionar por seguridad.

Ejercicio 4: el grupo de trabajo

Enunciado: simula un proyecto compartido: crea la carpeta ``equipo/``, ponle permiso 770, crea dentro un archivo ``plan.txt`` con permiso 660, y explica con tus palabras qué puede hacer el dueño, alguien del grupo y alguien ajeno, en cada caso.

Solución explicada paso a paso. Paso 1: ``mkdir ~/practica-linux/equipo`` y ``chmod 770 ~/practica-linux/equipo``: ``rwxrwx---``: dueño y grupo hacen todo, los ajenos ni siquiera pueden entrar a ver. Paso 2: ``echo "plan del proyecto" > ~/practica-linux/equipo/plan.txt`` y ``chmod 660`

~/practica-linux/equipo/plan.txt: `rw-rw----`: dueño y grupo leen y escriben, ajenos nada. Paso 3: escribe tu explicación: "El dueño: todo. Alguien del grupo: entrar a la carpeta, ver y editar el archivo. Alguien ajeno: nada, ni siquiera sabe qué hay dentro". Paso 4: reflexiona: 770/660 es el patrón de "trabajo en equipo privado". En un servidor real, además pondrías la carpeta en el grupo del equipo con chgrp, pero eso requiere sudo y lo practicarás cuando administres tu propio servidor.

Ejercicio 5: auditoría de seguridad

Enunciado: haz una auditoría básica de tu zona de práctica: encuentra todos los archivos con permiso 777 (si hay), todos los scripts (.sh) sin permiso de ejecución, y escribe un mini-reporte de 5 líneas con lo que encontraste y qué corregiste.

Solución explicada paso a paso. Paso 1: `find ~/practica-linux -perm 777`: busca archivos con exactamente esos permisos. Si no hay, bien: tu zona está sana. (Crea uno a propósito con `touch malo.txt && chmod 777 malo.txt` para practicar la detección, y luego corrígelo.) Paso 2: `find ~/practica-linux -name "\*.sh" ! -perm -u+x`: encuentra scripts sin permiso de ejecución para el dueño. Paso 3: corrige lo que encuentres: al 777 ponle 644 si es un dato o 755 si debe ejecutarse; a los scripts sin +x dales `chmod u+x` si son scripts de verdad. Paso 4: escribe tu reporte: qué encontraste, por qué era un problema y qué permiso le pusiste. Paso 5: agrega a tu cheat sheet el comando de auditoría: `find RUTA -perm 777`. Los profesionales auditan permisos periódicamente; ahora tú también sabes hacerlo.

Tarjeta de referencia rápida

Tu acordeón de permisos. La tabla octal y los comandos de esta unidad son de esas cosas que al principio consultas cada cinco minutos y al mes ya salen solas.

Comandos clave de la unidad

| Comando | Descripción | |---|---| | `ls -l` | Muestra permisos, dueño, grupo, tamaño, fecha y nombre. Donde empieza todo diagnóstico. | | `chmod u+x archivo` | Otorga ejecución al dueño. El ritual de todo script nuevo. | | `chmod 755 archivo` | `rwxr-xr-x`: tú control total, otros leen y ejecutan. Para scripts públicos. | | `chmod 644 archivo` | `rw-r--r--`: tú escribes, otros solo leen. Para archivos normales. | | `chmod 700 archivo` | `rwx-----`: solo tú, nada para nadie más. Para lo privado. | | `chmod 600 archivo` | `rw-----`: solo tú lees y escribes. Para claves y secretos. | | `chmod 770 carpeta` | `rwxrwx---`: tú y tu grupo todo, otros nada. Para trabajo en equipo. | | `chmod -R 755 carpeta` | Aplica 755 a la carpeta y todo lo que contiene, recursivamente. Con respeto. | | `chmod a-w archivo` | Quita escritura a todos (a = all). Para

congelar un archivo importante. | | `chmod go-rwx archivo` | Quita todo permiso a grupo y otros de un golpe. Blindaje rápido. | | `chown usuario archivo` | Cambia el dueño (requiere sudo). Con `-R` para árboles completos. | | `chown usuario:grupo archivo` | Cambia dueño y grupo de un solo golpe. | | `chgrp grupo archivo` | Cambia solo el grupo dueño (requiere sudo normalmente). Con `-R` recursivo. | | `umask` | Muestra tu máscara de permisos para archivos nuevos. | | `umask 022` | Máscara estándar: archivos nuevos 644, carpetas nuevas 755. | | `umask 077` | Máscara paranoica: archivos nuevos 600, carpetas 700. Solo tú. | | `id` | Muestra tu usuario, grupo principal y todos tus grupos. Tu credencial completa. | | `groups` | Lista los grupos a los que perteneces. | | `whoami` | Tu nombre de usuario actual (repaso de la Unidad 1, clave aquí). | | `sudo comando` | Ejecuta un comando como administrador. Te pide tu contraseña (a ciegas). | | `sudo -l` | Lista qué comandos puedes ejecutar con sudo. Conócelo antes de necesitarlo. | | `stat -c "%a %n" archivo` | Muestra los permisos en octal de forma compacta. Para verificaciones rápidas. | | `find ruta -perm -002` | Caza archivos escribibles por cualquiera: el radar del auditor. |

La tabla octal (tatúatela en la mente)

Número	Permisos	Significado
7	`rwx`	Leer, escribir y ejecutar: control total.
6	`rw-`	Leer y escribir, sin ejecutar: para archivos de datos.
5	`r-x`	Leer y ejecutar, sin escribir: para scripts que otros usan.
4	`r--`	Solo leer: para archivos de solo lectura.
3	`-wx`	Escribir y ejecutar, sin leer: rarísimo en la práctica.
2	`-w-`	Solo escribir: casi nunca se usa solo.
1	`--x`	Solo ejecutar: para programas que no deben leerse.
0	`---`	Nada: acceso totalmente negado.

Combinaciones que debes reconocer al instante: `755` = programa público, `644` = archivo normal, `700`/`600` = privado, `770`/`660` = equipo, `777` = peligro.

One-liners que te salvan el día

```
chmod u+x script.sh && ./script.sh
```

Haz ejecutable tu script y córrelo en una línea. El ritual completo del "hola mundo" de permisos.

```
find . -perm -002 -type f
```

Encuentra archivos que cualquiera puede modificar bajo la carpeta actual. Si aparece algo que no esperabas, tienes una brecha que cerrar.

```
sudo chown -R $USER:$USER ~/mi-proyecto
```

Recupera la propiedad de todo tu proyecto (útil cuando algo quedó como root por un `sudo` descuidado). El `\$USER` se reemplaza solo con tu nombre de usuario.

```
chmod -R a-w documentos-importantes/
```

Congela una carpeta: quita el permiso de escritura a todos, incluido tú. Nada se modifica por accidente hasta que lo descongeles con `u+w`.

```
ls -l /etc/shadow /etc/passwd
```

Compara: `shadow` (contraseñas) solo lo lee root, `passwd` (usuarios) lo lee cualquiera. En una línea entiendes por qué existen los tríos de permisos.

Diagnóstico de "Permiso denegado" en 5 pasos

Cuando algo te dice "Permiso denegado", no adivines: sigue este protocolo en orden. Funciona para archivos, carpetas y scripts.

```
ls -l archivo-problema      # PASO 1: lee la cadena de permisos completa
id                           # PASO 2: ¿quién eres? ¿a qué grupos perteneces?
```

Paso 3: identifica cuál trío te aplica. Si tu usuario es el dueño, manda el primer trío; si no eres dueño pero compartes grupo, el segundo; si no eres ni uno ni otro, el tercero. La mayoría de los diagnósticos mueren aquí porque la gente salta este paso.

Paso 4: verifica la letra exacta que te falta. ¿Quieres leer y no hay `r`? ¿Ejecutar y no hay `x`? ¿Entrar a una carpeta y no hay `x` en la carpeta? Cada síntoma apunta a una letra.

```
namei -l /ruta/al/archivo/problema # PASO 5 (bonus): muestra los permisos de
                                     # CADA nivel de la ruta. A veces el problema
                                     # no está en el archivo, sino en una carpeta
papá
```

El caso clásico que `namei` resuelve: tienes `rwx` en el archivo pero "permiso denegado" al abrirlo... porque una carpeta superior no tiene `x` para ti y ni siquiera puedes atravesarla. Sin `namei`, ese diagnóstico toma una eternidad.

Combinaciones de comandos de uso diario

```
touch secreto.txt && chmod 600 secreto.txt && ls -l secreto.txt
```

Crea un archivo privado y bléndalo en la misma línea: nace y de inmediato queda solo para ti. El patrón "crear y proteger" que deberías usar para todo lo sensible.

```
mkdir -p compartido && chgrp devops compartido && chmod 770 compartido
```

Crea una carpeta de equipo, asígnale el grupo y cierra el acceso a extraños, todo de un golpe. (El `chgrp` puede pedir sudo según tu sistema.)

```
find ~/proyectos -type f -perm -002
```

Auditoría rápida: busca en tus proyectos cualquier archivo que cualquiera pueda modificar. Debería devolver vacío; si no, investiga.

Resumen de la unidad

- La cadena de `ls -l` se lee así: tipo + 3 tríos (dueño, grupo, otros) de rwx; el guion es permiso negado.
- r = leer, w = escribir, x = ejecutar; en directorios, x significa poder entrar y r ver el contenido.
- chmod simbólico: a quién (u/g/o/a), operación (+/-/=), permiso (rwx). Ej: `chmod u+x script.sh`.
- chmod octal: r=4, w=2, x=1; suma por grupo. 755 = ejecutable público, 644 = archivo normal.
- Nunca uses 777 salvo excepciones temporales y conscientes: es quitarle la puerta a la casa.
- chown cambia el dueño y chgrp el grupo; normalmente requieren sudo; -R para carpetas completas.
- Linux es multiusuario: usuarios y grupos organizan quién puede qué; root tiene poder total.
- sudo ejecuta un comando como administrador sin vivir como root: la llave maestra bajo control.

- umask define los permisos de nacimiento: 022 da 644 a archivos y 755 a directorios.
- El ritual del script: escribir, ``chmod u+x``, ejecutar con ``.``. "Permiso denegado" = falta la x.
- ``chmod -R`` aplica a todo el árbol bajo la ruta: verifícalo dos veces, porque no hay botón de deshacer.
- ``chown usuario:grupo archivo`` cambia dueño y grupo de un solo golpe, sin dos comandos.
- En ``ls -l``, las columnas 3 y 4 son dueño y grupo: ahí empieza toda auditoría de permisos.
- Sin ``x`` en una carpeta no puedes entrar con ``cd``, aunque veas su nombre listado: ``x`` es la llave de la puerta.
- ``sudo -l`` te dice qué comandos puedes ejecutar como administrador: conócelo antes de necesitarlo.
- Los archivos nuevos nunca nacen ejecutables: la ``x`` siempre se otorga a mano, y eso es seguridad por diseño.
- ``umask 027`` es más estricta que ``022``: lo nuevo queda legible para tu grupo pero cerrado para otros.
- ``id`` muestra tu usuario, tu grupo principal y todos tus grupos: tu credencial completa en una línea.
- Ante "Permiso denegado", el orden es: ``ls -l`` para ver, identificar qué trío te aplica, y actuar con precisión (nunca con 777).

UNIDAD 4: Redirección, tuberías y filtros

Ya sabes moverte, manipular archivos y controlar permisos. Ahora viene la unidad donde la terminal muestra su verdadera magia: conectar comandos entre sí como piezas de Lego para hacer cosas que ningún comando solo podría hacer. Redirección y tuberías son el corazón de la filosofía Unix, y dominarlas es lo que convierte a un usuario en un operador.

En esta unidad vas a aprender los tres flujos de todo programa (entrada, salida y errores), a redirigir la salida a archivos con `>` y `>>`, a alimentar programas desde archivos con `<`, a separar los errores con `2>`, y a encadenar comandos con el pipe `|`. También vas a conocer la caja de herramientas de texto: `sort`, `uniq`, `cut`, `tr`, `wc` y `tee`.

Por qué importa esta unidad: casi todo el procesamiento de datos en Linux es tuberías. Analizar registros de un servidor, limpiar una lista de correos, contar visitas a una página, extraer columnas de un reporte: todo se hace encadenando comandos pequeños. Cada comando hace una sola cosa bien, y el pipe los une en máquinas de procesamiento. Esta es la idea más poderosa de Unix, y tiene más de 50 años funcionando.

Hay otra razón: la redirección es como tus scripts van a guardar resultados, crear archivos y manejar errores. Sin `>` y `>>`, tus scripts solo podrían escupir texto a la pantalla y nada quedaría registrado. Con ellos, tus scripts escriben reportes, alimentan bitácoras y se comunican con otros programas. Es el pegamento de la automatización.

Cómo encaja en el curso: las Unidades 1 a 3 te dieron archivos y permisos; esta te da el flujo de datos entre programas. En la Unidad 5 procesarás ese texto con `grep`, `sed` y `awk` (que se alimentan de pipes). Desde la Unidad 6, tus scripts usarán redirección para guardar resultados y pipes para procesar datos internamente.

Empecemos. Al terminar vas a construir cadenas de 4 o 5 comandos que hagan en una línea lo que a otros les toma un programa entero.

Explicación detallada

Los tres flujos: `stdin`, `stdout` y `stderr`

Todo programa en Linux habla por tres canales numerados. La entrada estándar (stdin, canal 0) es por donde el programa recibe datos: normalmente tu teclado. La salida estándar (stdout, canal 1) es por donde entrega su resultado normal: normalmente tu pantalla. Y la salida de errores (stderr, canal 2) es por donde reporta problemas: también tu pantalla, pero por un canal separado.

¿Por qué separar la salida normal de los errores? Para poder tratarlos distinto. Puedes guardar los resultados en un archivo y seguir viendo los errores en pantalla, o mandar los errores a un registro mientras los resultados siguen su camino por una tubería. Esta separación es una de las decisiones de diseño más inteligentes de Unix.

Analogía: un programa es como un restaurante con tres puertas. Por la puerta 0 entra la materia prima (tus datos). Por la puerta 1 sale el platillo terminado (el resultado). Por la puerta 2 sale el mesero a avisarte si algo salió mal (los errores). La redirección te permite cambiar a dónde lleva cada puerta.

> y >>: guardar la salida en archivos

El operador `>` redirige la salida estándar a un archivo, creándolo o SOBRESCRIBIÉNDOLO si ya existe:

```
ls -lh > lista.txt
```

La lista ya no aparece en pantalla: quedó guardada en lista.txt. Cuidado: si lista.txt ya existía con contenido valioso, `>` lo borra sin preguntar y escribe lo nuevo encima. Es el equivalente en redirección del rm: potente y sin perdón.

El operador `>>` agrega al final del archivo sin borrar lo que había:

```
echo "línea nueva" >> diario.txt
```

Si el archivo no existe, lo crea; si existe, le pega la línea al final. Regla práctica: `>` es "escribir de nuevo" (para reportes frescos), `>>` es "agregar al diario" (para bitácoras que crecen).

<: alimentar desde archivos y 2>: domar los errores

El operador `<` invierte el flujo: en lugar de que el programa lea del teclado, lee de un archivo:

```
wc -l < lista.txt
```

wc cuenta las líneas del archivo sin que tengas que escribir el nombre como argumento. No todos los comandos lo necesitan (muchos aceptan el archivo como argumento directo), pero `<` es esencial cuando un programa solo sabe leer de su entrada estándar.

Para los errores, el `>` dirige solo el canal 2:

```
find / -name "tesoro.txt" 2> errores.log
```

find va a soltar cientos de "permiso denegado" al recorrer el sistema; con `>` esos errores se van al archivo y tu pantalla queda limpia mostrando solo los resultados buenos. Y si quieres mandar AMBOS canales al mismo archivo:

```
comando > todo.log 2>&1
```

El `>&1` dice "los errores, al mismo lugar que la salida normal". Es el patrón estándar para bitácoras completas de scripts.

Para ignorar la salida por completo existe el agujero negro `/dev/null`:

```
comando 2> /dev/null
```

Los errores desaparecen. Úsalo con moderación: ignorar errores es útil cuando sabes que son ruido inofensivo, pero esconder errores reales es como tapar el foco del tablero del carro con cinta.

El pipe |: la estrella del show

El pipe (tubería), escrito como barra vertical `|`, conecta la salida de un comando con la entrada del siguiente:

```
ls -lh | head -10
```

ls produce la lista, el pipe se la pasa a head, y head muestra solo 10 líneas. Ningún comando cambió: solo se conectaron. Puedes encadenar tantos como quieras:

```
cat registro.log | grep ERROR | cut -d: -f2 | sort | uniq -c | sort -rn | head -5
```

Esta bestia hace algo real: toma un registro, filtra las líneas de error, extrae el segundo campo, ordena, cuenta repetidos, ordena por frecuencia y muestra el top 5. Siete comandos, una línea, cero programas escritos. Cuando veas a un administrador hacer esto en un servidor en llamas, entenderás por qué la terminal impone respeto.

La filosofía: cada comando hace UNA cosa bien. grep filtra, sort ordena, uniq cuenta. El pipe los convierte en una fábrica. No necesitas un programa "analizador de registros": lo ensamblas.

sort y uniq: ordenar y resumir

sort ordena líneas alfabéticamente:

```
sort nombres.txt
```

Opciones clave: `-n`` ordena numéricamente (para que 10 no quede antes que 2), `-r`` invierte el orden, `-u`` elimina duplicados al ordenar:

```
sort -n numeros.txt
sort -ru lista.txt
```

uniq elimina líneas repetidas CONSECUTIVAS (por eso casi siempre va después de sort):

```
sort lista.txt | uniq
```

Y `uniq -c`` cuenta cuántas veces se repite cada línea, el clásico para estadísticas:

```
cut -d: -f1 /etc/passwd | sort | uniq -c
```

Esto cuenta cuántos usuarios usan cada shell en el sistema. Datos reales, en una línea.

cut, tr y wc: el cuchillo, el traductor y la báscula

cut corta columnas de texto. Con `-d`` defines el separador y con `-f`` qué campos quieres:

```
cut -d: -f1,3 /etc/passwd
```

Extrae el campo 1 (usuario) y 3 (ID) del archivo de usuarios, separado por dos puntos. Para CSV con comas: `cut -d, -f2 datos.csv``.

tr traduce o borra caracteres:

```
echo "HOLA MUNDO" | tr 'A-Z' 'a-z'
```

Convierte mayúsculas a minúsculas. También borra con -d:

```
tr -d '0-9' < texto.txt
```

Elimina todos los dígitos. Es el corrector ortográfico de caracteres del shell.

wc (word count) cuenta líneas (-l), palabras (-w) y bytes (-c):

```
wc -l documento.txt
```

¿Cuántas líneas tiene? La pregunta más común al procesar archivos. `ls *.txt | wc -l` cuenta cuántos txt hay. Simple y omnipresente.

tee: ver y guardar a la vez

A veces quieres ver la salida EN PANTALLA y además guardarla en un archivo. Para eso existe tee, que bifurca el flujo como una T de plomería:

```
ls -lh | tee lista.txt
```

Ves la lista en tu pantalla Y queda guardada en lista.txt. Con `-a` agrega en lugar de sobrescribir: `tee -a`. Es el favorito para bitácoras: ``comando | tee -a bitacora.log`` te muestra el progreso y lo registra.

Ejemplos paso a paso

Ejemplo 1: tu primera bitácora

Vas a construir el hábito de registrar lo que hacen tus comandos, como un profesional.

```
cd ~/practica-linux  
ls -lh > inventario.txt  
cat inventario.txt
```

El inventario quedó guardado. Ahora agrega la fecha al inicio del archivo... bueno, al final, que es lo que `>>` hace:

```
date >> inventario.txt
cat inventario.txt
```

Ahora separa los errores en una búsqueda ambiciosa:

```
find /etc -name "*.conf" 2> errores-busqueda.txt | head -5
cat errores-busqueda.txt | head -5
```

Los resultados buenos salieron por el pipe a head; los "permiso denegado" quedaron en el archivo de errores, sin ensuciar tu pantalla.

Explicación: este patrón (resultado limpio por un lado, errores registrados por otro) es exactamente como se ejecutan los trabajos automatizados en servidores: lo importante se procesa, los errores se guardan para revisarlos después.

Ejemplo 2: la fábrica de estadísticas

Vas a crear datos de mentira y extraer estadísticas reales con una tubería.

```
cd ~/practica-linux
printf "ana\nluis\nana\nmaria\nluis\nana\njose\nmaria\nana\n" > visitas.txt
cat visitas.txt
```

Tienes un registro de visitas: cada línea es una persona que entró a tu sitio.

```
sort visitas.txt | uniq -c | sort -rn
```

Magia en una línea: ordena los nombres, cuenta repetidos, y ordena por conteo descendente. Resultado: ana 4, luis 2, maria 2, jose 1. Acabas de hacer analítica web con tres comandos.

```
sort visitas.txt | uniq -c | sort -rn | head -2 > top-visitantes.txt
cat top-visitantes.txt
```

Y guardas el top 2 en un archivo. Cada pieza de la tubería hace una sola cosa; juntas hacen el reporte completo.

Explicación: este es EL caso de uso de las tuberías. Cuando en la Unidad 9 veas registros reales de un servidor, aplicarás exactamente esta técnica para encontrar las IPs más activas, los errores más frecuentes o las páginas más visitadas.

Ejemplo 3: cirugía de texto con cut y tr

Vas a extraer y limpiar datos como lo harías con un archivo CSV real.

```
cd ~/practica-linux
printf "nombre,edad,ciudad\nana,25,cdmx\nluis,30,monterrey\nmaria,22,cdmx\n" >
personas.csv
cat personas.csv
```

Un CSV clásico. Extrae solo los nombres:

```
cut -d, -f1 personas.csv
```

Extrae nombre y ciudad (campos 1 y 3):

```
cut -d, -f1,3 personas.csv
```

Ahora normaliza: todo a minúsculas y sin la línea de encabezado:

```
tail -n +2 personas.csv | tr 'A-Z' 'a-z'
```

`tail -n +2` salta la primera línea (el encabezado) y tr baja todo a minúsculas. Datos limpios, listos para procesar.

Explicación: cut + tr + tail son el kit de limpieza de datos del shell. Antes de analizar cualquier archivo de texto real, casi siempre hay que extraer columnas, quitar encabezados y normalizar formato. Ya sabes hacerlo.

Ejemplo 4: el reporte completo con tee

Vas a armar un mini-reporte del sistema que se muestra y se guarda a la vez.

```
cd ~/practica-linux
{
echo "=== REPORTE DEL SISTEMA ==="
date
echo "--- Espacio en disco ---"
df -h | head -5
echo "--- Memoria ---"
free -h
echo "--- Usuarios conectados ---"
who
} | tee reporte-sistema.txt
```

Las llaves `{}` agrupan varios comandos para que el pipe los trate como uno solo. Todo el reporte aparece en tu pantalla Y queda en reporte-sistema.txt.

```
cat reporte-sistema.txt
```

Confirmas que el archivo tiene exactamente lo mismo que viste. Este patrón (grupo + tee) es como se generan los reportes automáticos en servidores: se ejecutan, se ven en vivo y quedan archivados.

Explicación: las llaves son el pegamento que convierte una secuencia en una unidad. Combinadas con tee, tienes el generador de reportes más simple y efectivo de Linux.

Errores comunes y casos reales

La redirección y los pipes son el tema donde un solo carácter de diferencia cambia todo el resultado: `>` contra `>>` es la diferencia entre guardar y destruir. Los errores de esta unidad son silenciosos y traicioneros: el comando no se queja, simplemente te da un resultado vacío o trunco y tú ni cuenta te das. Por eso cada caso incluye cómo VERIFICAR, no solo cómo arreglar.

Caso 1: `>` te borró el contenido del archivo (querías `>>`)

Síntoma: tenías un archivo `notas.txt` con un mes de apuntes, ejecutaste `echo "nueva idea" > notas.txt` para agregar una línea, y al abrirlo solo está la nueva línea. Todo lo demás desapareció.

Por qué ocurre: `>` SOBREScribe: trunca el archivo a cero bytes antes de escribir. `>>` AGREGA al final. Un solo carácter de diferencia, consecuencias opuestas. Es el error más doloroso de esta unidad porque es irreversible.

Solución: usa `>>` cuando quieras agregar. Y si quieres un seguro permanente, activa `noclobber`: con él, `>` se niega a sobrescribir archivos existentes.

```
echo "línea 1" > notas.txt      # crea el archivo con una línea
echo "línea 2" >> notas.txt     # AGREGA la segunda línea al final
cat notas.txt                  # verifica: deben estar las dos líneas
set -o noclobber               # activa el seguro anti-sobrescritura
echo "ups" > notas.txt          # ahora se niega: "cannot overwrite existing file"
echo "sí puedo" >> notas.txt    # agregar sigue permitido
set +o noclobber               # desactiva el seguro si te estorba
```

Lección profesional: antes de CUALQUIER `>`, pregúntate: "¿quiero reemplazar o agregar?". Los profesionales que viven agregando a bitácoras usan `>>` por reflejo y activan `noclobber` en sus sesiones importantes.

Caso 2: `2>&1 > archivo` no guarda los errores (el orden sí importa)

Síntoma: ejecutas `comando 2>&1 > salida.txt`, el comando falla, y al abrir `salida.txt` no están los mensajes de error. "¿A dónde se fueron?", te preguntas, mientras los errores se mostraron en pantalla y se perdieron.

Por qué ocurre: las redirecciones se procesan de IZQUIERDA a DERECHA. En `2>&1 > salida.txt`, primero `2>&1` dice "stderr, ve a donde vaya stdout AHORITA" (que aún es la pantalla), y DESPUÉS `> salida.txt` manda stdout al archivo. Resultado: errores a la pantalla, salida al archivo. El orden correcto es al revés.

Solución: primero redirige stdout al archivo, y luego manda stderr a seguirlo.

```
ls /no-existe 2>&1 > salida.txt  # MAL: el error se va a la pantalla, no al
                                # archivo
cat salida.txt                  # vacío: el error no se guardó
ls /no-existe > salida.txt 2>&1  # BIEN: todo (salida y errores) va al archivo
cat salida.txt                  # ahí está el mensaje de error, guardado
```

Lección profesional: "primero el destino, luego el seguidor". Esta regla del orden aparece en exámenes, en entrevistas y en scripts reales. Si un log "no captura los errores", lo primero que se revisa es el orden de las redirecciones.

Caso 3: el pipe que no hace nada (``echo hola | cd /tmp``)

Síntoma: ejecutas ``echo /tmp | cd`` esperando entrar a ``/tmp``, o ``ls *.bak | rm`` esperando borrarlos, y no pasa nada (o ``cd`` se queja). El pipe está ahí, pero el comando de la derecha lo ignora.

Por qué ocurre: NO todos los comandos leen de stdin. ``cd``, ``rm``, ``cp``, ``mv`` trabajan con ARGUMENTOS, no con entrada estándar. El pipe les entrega texto por stdin y ellos ni lo miran. Es como gritarle instrucciones a alguien con audífonos.

Solución: usa comandos que sí leen stdin (``grep``, ``sort``, ``wc``, ``cut``), o convierte el texto en argumentos con ``xargs`` (adelanto útil desde ya).

```
echo /tmp | cd          # MAL: cd no lee stdin, no pasa nada
cd /tmp                # BIEN: cd quiere el destino como argumento
ls *.bak | rm          # MAL: rm no lee stdin
rm *.bak               # BIEN: el comodín ya genera los argumentos
find . -name "*.tmp" | xargs rm  # BIEN: xargs convierte stdin en argumentos
```

Lección profesional: ante un pipe que "no hace nada", la pregunta no es "¿está roto el pipe?", sino "¿el comando de la derecha lee stdin?". Esa pregunta te ahorra horas de confusión.

Caso 4: ``sort`` pone el 10 antes que el 2

Síntoma: ordenas una lista de números o tamaños con ``sort`` y el resultado es ``1, 10, 100, 2, 20, 3``. Numéricamente es un desastre, aunque "se ve ordenado".

Por qué ocurre: ``sort`` por defecto ordena ALFABÉTICAMENTE, carácter por carácter: compara "1" contra "1", luego "0" contra nada... y "10" queda antes que "2" porque "1" < "2" como texto. Para números necesitas decirle explícitamente que ordene por valor numérico.

Solución: agrega ``-n`` para orden numérico. Y ``-r`` si lo quieres de mayor a menor.

```
echo "10
2
100
20
3" > numeros.txt
sort numeros.txt      # MAL: 10, 100, 2, 20, 3 (alfabético)
sort -n numeros.txt   # BIEN: 2, 3, 10, 20, 100 (numérico)
sort -nr numeros.txt  # de mayor a menor: 100, 20, 10, 3, 2
rm numeros.txt
```

Lección profesional: este error aparece en reportes de "top 10" que salen mal y nadie sabe por qué. Regla: si lo que ordenas son números (tamaños, conteos, puertos), `sort` siempre va con `-n`.

Caso 5: `uniq` no elimina duplicados (porque no están juntos)

Síntoma: pasas una lista por `uniq` esperando líneas únicas y el resultado sigue teniendo repetidos. "¿No se supone que uniq elimina duplicados?", reclamas.

Por qué ocurre: `uniq` solo elimina duplicados ADYACENTES (líneas iguales pegadas). Si tu lista es `a, b, a, c, b`, `uniq` no toca nada porque ningún repetido está junto a su gemelo. Por eso el patrón canónico es `sort | uniq`: primero juntas a los gemelos, luego los eliminas.

Solución: siempre `sort` antes de `uniq`. Y `uniq -c` para contar cuántas veces apareció cada línea.

```
echo "a
b
a
c
b" > letras.txt
uniq letras.txt          # MAL: no hay adyacentes, no elimina nada
sort letras.txt | uniq   # BIEN: a, b, c
sort letras.txt | uniq -c # BIEN y útil: cuenta repeticiones
rm letras.txt
```

Lección profesional: `sort | uniq -c | sort -nr` es LA tubería de análisis de frecuencia en Unix. Aparece en logs, en encuestas, en reportes: memoriza ese trío porque lo vas a escribir cientos de veces.

Caso 6: `cut` devuelve líneas completas en vez de columnas

Síntoma: tienes un CSV (`nombre,edad,ciudad`) y ejecutas `cut -f2 datos.csv` esperando la columna de edad, pero te devuelve las líneas completas, sin cortar nada.

Por qué ocurre: `cut` por defecto usa el TABULADOR como separador. Si tu archivo usa comas (o punto y coma, o dos puntos), `cut` no encuentra ningún tabulador, asume que toda la línea es "el campo 1"... y con `-f2` no hay campo 2, así que el comportamiento te confunde.

Solución: declara el separador con `-d` según tu archivo: `-d','` para CSV, `-d':'` para `/etc/passwd`, `-d';` para algunos CSV europeos.

```
cut -f2 datos.csv          # MAL: sin -d asume tabulador, no corta
cut -d',' -f2 datos.csv    # BIEN: declara la coma como separador
cut -d':' -f1 /etc/passwd  # los nombres de usuario del sistema
cut -d',' -f1,3 datos.csv  # primera y tercera columna
```

Lección profesional: ante un `cut` que "no corta", el diagnóstico es siempre el separador: mira el archivo con `head -1` y `cat -A` (muestra los caracteres invisibles) para ver qué separa realmente las columnas.

Caso real 1: el servidor web está lento, ¿quién lo está golpeando?

Son las 3 AM, el monitoreo dice que el servidor web responde lento. Te conectas y, sin herramientas elegantes, solo con tuberías sobre el registro de accesos:

```
tail -1000 /var/log/nginx/access.log | cut -d' ' -f1 | sort | uniq -c | sort -nr |
head -10
```

Esa tubería te dice, de las últimas 1000 peticiones, qué IPs pidieron más, ordenadas de mayor a menor. Si una IP tiene 800 de 1000, contrasta a tu atacante o a un scraper descontrolado. Todo el diagnóstico en UNA línea, con comandos de esta unidad. Esto no es teoría: es exactamente lo que se hace en incidentes reales.

Caso real 2: reporte de ventas desde un CSV

Te pasan `ventas.csv` con formato `fecha,producto,monto` y te piden: ¿cuánto se vendió por producto? Sin abrir hojas de cálculo:

```
cut -d',' -f2 ventas.csv | sort | uniq -c | sort -nr > top-productos.txt
cut -d',' -f3 ventas.csv | awk '{s+=$1} END {print s}'
cat top-productos.txt
```

La primera línea genera el ranking de productos y lo guarda; la segunda suma todos los montos. En segundos tienes el reporte que alguien haría en media hora con una hoja de cálculo. (El `awk` lo verás a fondo en la Unidad 5; aquí úsalo como calculadora prestada.)

Miniproyecto guiado de la unidad

Vas a montar una **Fábrica de Análisis de Registros**: generarás un registro de accesos simulado (como el de un servidor web real) y construirás tuberías para extraer inteligencia de él: quién ataca, qué páginas fallan, a qué horas hay más tráfico. Al terminar tendrás un reporte profesional generado 100% con redirección y pipes.

Paso 1: fabrica tu materia prima (el registro simulado)

Crea un archivo `accesos.log` con 40 líneas que simulen peticiones web reales: IP, fecha, página pedida y código de respuesta. Usa `printf` con redirección `>>` para agregarlo por bloques (así practicas agregar sin sobrescribir).

```
cd ~
mkdir -p fabrica-logs && cd fabrica-logs
echo "192.168.1.10 [10:01] /index.html 200" > accesos.log
echo "192.168.1.20 [10:02] /productos 200" >> accesos.log
echo "10.0.0.5 [10:03] /admin 403" >> accesos.log
echo "192.168.1.10 [10:04] /index.html 200" >> accesos.log
echo "10.0.0.5 [10:05] /admin 403" >> accesos.log
echo "10.0.0.5 [10:06] /wp-login 404" >> accesos.log
```

Agrega unas 35 líneas más variando IPs (unas 6 distintas), páginas (`/index.html`, `/productos`, `/contacto`, `/admin`, `/wp-login`) y códigos (`200`, `404`, `403`, `500`). Que algunas IPs se repitan mucho (simulan usuarios normales y al atacante). Para hacerlo rápido, repite bloques con pequeñas variaciones: lo importante es el volumen y la variedad, no la perfección literaria.

Verificación: `wc -l accesos.log` debe decir al menos 40, y `cat accesos.log | head -5` debe mostrar el formato correcto en cada línea.

Paso 2: inspección básica del material

Antes de procesar, conoce tu materia prima como un chef conoce sus ingredientes.

```
wc -l accesos.log          # ¿cuántas peticiones hay?
head -3 accesos.log        # ¿cómo se ve el inicio?
tail -3 accesos.log        # ¿cómo se ve el final?
cut -d' ' -f1 accesos.log | sort -u  # ¿cuántas IPs distintas hay? (sort -u =
únicas)
```

Verificación: debes poder responder de memoria: total de líneas, formato de cada línea y número de IPs distintas. Si no puedes, repite la inspección: un analista que no conoce sus datos, inventa conclusiones.

Paso 3: caza al atacante (top de IPs)

Construye la tubería estrella: extrae la columna de IP, cuenta repeticiones y ordena de mayor a menor.

```
cut -d' ' -f1 accesos.log | sort | uniq -c | sort -nr | head -5
```

Desarma la tubería por etapas para entender cada pieza (esta técnica de depuración por etapas te va a servir toda la vida):

```
cut -d' ' -f1 accesos.log | head -5          # etapa 1: ¿extrae bien las IPs?
cut -d' ' -f1 accesos.log | sort | head -5  # etapa 2: ¿las agrupa?
cut -d' ' -f1 accesos.log | sort | uniq -c  # etapa 3: ¿las cuenta?
```

Verificación: la tubería completa debe mostrarte las 5 IPs más activas con sus conteos. La más activa es tu "atacante" simulado (o tu usuario más fiel, depende de cómo generaste los datos). Anota el resultado: lo necesitarás para el reporte.

Paso 4: separa el trigo de la paja (códigos de error)

No todo el tráfico es igual: filtra solo las peticiones problemáticas (códigos 4xx y 5xx) y analiza qué páginas fallan.

```
grep " 404" accesos.log | wc -l      # ¿cuántos "no encontrado" hubo?
grep " 403" accesos.log | wc -l      # ¿cuántos "prohibido" hubo?
grep " 404" accesos.log | cut -d' ' -f3 | sort | uniq -c | sort -nr
```

La última tubería te dice qué páginas dan más errores 404, ordenadas. Verificación: debes poder responder "la página con más 404 es X con N casos". (El `grep` lo dominarás en la Unidad 5; aquí es un filtro prestado. Fíjate que el patrón lleva un espacio antes del código para no confundir "404" con parte de una IP.)

Paso 5: normaliza con `tr` y cuenta palabras

Los datos reales vienen sucios: mayúsculas mezcladas, espacios de más. Practica la limpieza con `tr` sobre las páginas visitadas.

```
cut -d' ' -f3 accesos.log | tr 'A-Z' 'a-z' | sort | uniq -c | sort -nr
```

Si en tu log hay ``/Admin`` y ``/admin``, después del ``tr`` cuentan como la misma página. Verificación: compara el resultado con y sin el ``tr``: si cambian los conteos, tenías datos sucios y la limpieza sí importaba. Este paso parece menor, pero en datos reales es el que más errores evita.

Otro truco de limpieza: ``tr -s ' '`` colapsa espacios múltiples en uno solo, por si tu log tiene separadores irregulares.

Paso 6: arma el reporte final con ``tee``

Genera ``reporte.txt`` con los hallazgos, mostrando cada resultado en pantalla Y guardándolo a la vez con ``tee``. Esta es la forma profesional de documentar un análisis: lo ves y queda guardado.

```
{
echo "=== REPORTE DE ACCESOS ==="
echo ""
echo "Total de peticiones:"
wc -l < accesos.log
echo ""
echo "Top 5 IPs más activas:"
cut -d' ' -f1 accesos.log | sort | uniq -c | sort -nr | head -5
echo ""
echo "Errores 404 por página:"
grep " 404" accesos.log | cut -d' ' -f3 | sort | uniq -c | sort -nr
echo ""
echo "Errores 403 totales:"
grep -c " 403" accesos.log
} | tee reporte.txt
```

Las llaves ``{}`` agrupan todos los comandos para que el ``tee`` del final capture TODO de un golpe. Verificación: ``cat reporte.txt`` debe mostrar el reporte completo y bien formado, y debes haber visto cada sección aparecer en pantalla mientras se generaba. Si una sección salió vacía, depura esa tubería por separado (técnica del paso 3).

Paso 7: maneja los errores como profesional

Repite el análisis pero capturando también lo que salga mal: redirige errores a un archivo separado para que tu reporte nunca se contamine.

```
cut -d' ' -f1 accesos.log | sort | uniq -c | sort -nr > top-ips.txt 2> errores.log
ls -lh top-ips.txt errores.log
cat errores.log          # debe estar vacío: no hubo errores, y eso también es
información
```

Ahora provoca un error a propósito para ver el mecanismo funcionar:

```
cut -d' ' -f1 no-existe.log > nada.txt 2> errores.log
cat errores.log          # ahora sí hay un mensaje de error, capturado limpiamente
cat nada.txt             # vacío: el error no contaminó la salida normal
```

Verificación: debes entender la diferencia entre lo que va a `top-ips.txt` (stdout) y lo que va a `errores.log` (stderr). Limpia los archivos de prueba (`rm nada.txt`) y deja tu fábrica ordenada.

Paso 8: compara dos días y detecta anomalías

Un reporte de un solo día es una foto; dos días son una película. Genera un segundo registro simulando el día siguiente, con un par de cambios: una IP nueva muy activa (un posible atacante nuevo) y una IP del día 1 que desaparece.

```
cd ~/fabrica-logs
cp accesos.log accesos-dia1.log
echo "203.0.113.9 [11:01] /wp-login 404" > accesos-dia2.log
echo "203.0.113.9 [11:02] /wp-login 404" >> accesos-dia2.log
echo "203.0.113.9 [11:03] /admin 403" >> accesos-dia2.log
echo "192.168.1.10 [11:04] /index.html 200" >> accesos-dia2.log
echo "192.168.1.20 [11:05] /productos 200" >> accesos-dia2.log
```

Agrega unas 30 líneas más al día 2 (repite el patrón: la IP nueva `203.0.113.9` muy insistente en `/wp-login` y `/admin`, las normales tranquilas, y OMITE por completo una IP que sí estaba en el día 1).

Ahora la comparación. Primero extrae las listas de IPs únicas de cada día, ordenadas (requisito para comparar):

```
cut -d' ' -f1 accesos-dia1.log | sort -u > ips-dia1.txt
cut -d' ' -f1 accesos-dia2.log | sort -u > ips-dia2.txt
cat ips-dia1.txt
cat ips-dia2.txt
```

Y ahora las preguntas de detective, respondidas con `comm` (compara dos archivos ordenados línea por línea):

```
comm -13 ips-dia1.txt ips-dia2.txt # IPs que están SOLO en el día 2: las nuevas
comm -23 ips-dia1.txt ips-dia2.txt # IPs que estaban en el día 1 y DESAPARECIERON
comm -12 ips-dia1.txt ips-dia2.txt # IPs presentes ambos días: tu base fiel
```

Verificación: el primer comando debe mostrarte `203.0.113.9` (tu atacante nuevo), el segundo la IP que omitiste a propósito, y el tercero las habituales. Si no cuadra, revisa que ambos archivos estén ordenados (`comm` lo exige) y que generaste bien el día 2.

Cierra con el veredicto: agrega al final de `reporte.txt` una sección "Comparativa día 1 vs día 2" con tres datos: total de peticiones por día (`wc -l` de cada log), la IP nueva detectada y su comportamiento (mira sus líneas: ¿qué páginas pide? ¿qué códigos recibe?), y la IP desaparecida. Un analista que entrega "hay una IP nueva haciendo fuerza bruta contra /wp-login" vale oro: acabas de hacer exactamente eso, con tuberías.

Checklist de entrega

```
cd ~/fabrica-logs
test -f accesos.log && echo "materia prima OK"
test -f reporte.txt && echo "reporte OK"
grep -c "Top 5" reporte.txt # el reporte tiene la sección del top
wc -l accesos.log # al menos 40 líneas de materia prima
```

Tu fábrica está completa si tienes `accesos.log` con 40+ líneas, `reporte.txt` generado con `tee` y todas las secciones con datos coherentes. Este miniproyecto es, en pequeño, lo que hacen los analistas de sistemas todos los días.

Tu bitácora de tuberías reutilizables

Las tuberías que construiste hoy no son de un solo uso: son herramientas. Guárdalas en un archivo `tuberias.txt` con una línea de explicación cada una, como tu caja de herramientas personal de análisis.

```
cd ~/fabrica-logs
nano tuberias.txt
```

Escribe al menos estas cinco, con tus palabras explicando qué hace cada etapa:

1. La contadora de frecuencia: `cut -d' ' -f1 accesos.log | sort | uniq -c | sort -nr | head`

2. La cazadora de errores: ``grep " 404" accesos.log | cut -d' ' -f3 | sort | uniq -c | sort -nr``
3. La comparadora de días: ``comm -13 ips-dia1.txt ips-dia2.txt``
4. La normalizadora: ``cut -d' ' -f3 accesos.log | tr 'A-Z' 'a-z' | sort | uniq -c``
5. La generadora de reportes: ``{ comandos; } | tee reporte.txt``

Verificación: dentro de un mes, cuando necesites analizar cualquier archivo de texto, abrirás este archivo antes de pensar. Si cada tubería tiene su explicación en tus palabras, la bitácora funciona; si solo copiaste los comandos, es un adorno. Las herramientas que no entiendes no te pertenecen.

Reto final (sin solución, te toca a ti)

Sin mirar tus notas: 1) con UNA tubería, encuentra la hora (el campo ``[10:01]``) con más peticiones en tu log; 2) genera un segundo reporte ``reporte-errores.txt`` que contenga SOLO las líneas con códigos 4xx o 5xx, ordenadas por IP, sin usar ``grep`` (pista: hay otras formas de filtrar... o puedes usar ``grep``, pero el reto es hacerlo sin él para que pienses en alternativas); 3) explica por qué ``sort | uniq`` funciona y ``uniq | sort`` no sirve para contar frecuencias. Si resuelves las tres, las tuberías ya son tu idioma.

Práctica

Haz estos ejercicios en `~/practica-linux`. El objetivo: pensar en flujos, no en comandos aislados.

Ejercicio 1: bitácora de errores

Enunciado: ejecuta una búsqueda de archivos ``.log`` en todo ``/etc`` guardando los resultados en ``hallazgos.txt`` y los errores en ``errores.txt``, sin que nada aparezca en tu pantalla. Luego cuenta cuántas líneas tiene cada archivo.

Solución explicada paso a paso. Paso 1: ``find /etc -name "*.log" > hallazgos.txt 2> errores.txt``: la salida normal va a un archivo, los errores a otro, y la pantalla queda limpia. Paso 2: ``wc -l hallazgos.txt errores.txt``: cuentas las líneas de ambos de un golpe; `wc` acepta varios archivos. Paso 3: ``head -3 hallazgos.txt`` y ``head -3 errores.txt`` para inspeccionar qué tipo de contenido cayó en cada uno. Paso 4: reflexiona: los errores son casi todos "permiso denegado", que es ruido esperado al recorrer `/etc` como usuario normal. Separar flujos te deja ver la señal sin el ruido.

Ejercicio 2: top de palabras

Enunciado: crea un archivo `frutas.txt` con 15 líneas de nombres de frutas (algunas repetidas). Encuentra las 3 frutas más mencionadas con una sola tubería y guarda el resultado en `top-frutas.txt`.

Solución explicada paso a paso. Paso 1: crea el archivo, por ejemplo con `printf "manzana\npera\nmanzana\nuva\npera\nmanzana\nkiwi\nuva\nmanzana\npera\nmango\nuva\nmanzana\nkiwi\npera\n" > frutas.txt`. Paso 2: la tubería mágica: `sort frutas.txt | uniq -c | sort -rn | head -3 | tee top-frutas.txt`. Paso 3: desarma la tubería mentalmente: `sort` agrupa iguales, `uniq -c` cuenta, `sort -rn` ordena por conteo, `head -3` corta el top, `tee` muestra y guarda. Paso 4: `cat top-frutas.txt` para verificar. Si cambias los datos y repites la tubería, funciona igual: esa es la gracia de las herramientas genéricas.

Ejercicio 3: limpieza de CSV

Enunciado: con el archivo `personas.csv` del Ejemplo 3 (o créalo de nuevo), genera un archivo `ciudades.txt` que contenga solo los nombres de las ciudades en minúsculas, sin encabezado y sin duplicados, ordenadas alfabéticamente.

Solución explicada paso a paso. Paso 1: `tail -n +2 personas.csv` quita el encabezado. Paso 2: `cut -d, -f3` extrae la tercera columna (ciudad). Paso 3: `tr 'A-Z' 'a-z'` normaliza a minúsculas. Paso 4: `sort -u` ordena y quita duplicados de un golpe. Paso 5: todo junto: `tail -n +2 personas.csv | cut -d, -f3 | tr 'A-Z' 'a-z' | sort -u > ciudades.txt`. Paso 6: `cat ciudades.txt` para verificar. Cada pipe es un paso de la receta de limpieza; si el resultado sale mal, pruebas cada etapa por separado agregando pipes de a poco.

Ejercicio 4: el agujero negro

Enunciado: ejecuta `ls /root` (que dará error de permiso como usuario normal) de tres formas: mostrando el error en pantalla, ocultándolo por completo, y guardándolo en un archivo. Explica la diferencia de cada una.

Solución explicada paso a paso. Paso 1: `ls /root` muestra el error en pantalla: el canal 2 va a tu terminal por defecto. Paso 2: `ls /root 2> /dev/null` no muestra nada: el error se fue al agujero negro. El comando "falló en silencio". Paso 3: `ls /root 2> error-ls.txt` guarda el error; verifícalo con `cat error-ls.txt`. Paso 4: escribe tu explicación: "Sin redirección veo el error; con `/dev/null` lo descarto porque sé que es ruido esperado; con archivo lo conservo para analizarlo después". Paso 5: la lección profesional: en scripts automatizados, los errores esperados van a `/dev/null` y los inesperados a un registro. Decidir a dónde va cada error es parte del diseño.

Ejercicio 5: reporte matutino

Enunciado: crea un script mental (todavía no lo escribas como archivo, solo la línea) que genere un reporte con fecha, espacio en disco y los 5 archivos más grandes de tu carpeta personal, mostrándolo en pantalla y guardándolo en `reporte.txt` con una sola ejecución.

Solución explicada paso a paso. Paso 1: la fecha: `date`. Paso 2: el disco: `df -h /`. Paso 3: los 5 más grandes: `ls -lhS ~/practica-linux | head -6` (6 porque la primera línea es el total). Paso 4: agrúpalos: `{ date; echo "--- Disco ---"; df -h /; echo "--- Top 5 ---"; ls -lhS ~/practica-linux | head -6; } | tee reporte.txt`. Paso 5: ejecútalo y verifica con `cat reporte.txt`. Paso 6: guarda esta línea en tu cheat sheet: es el embrión de tu primer script de monitoreo, que escribirás como archivo en la Unidad 6 y programarás con cron en la Unidad 9.

Tarjeta de referencia rápida

Tu acordeón de flujos, redirección y tuberías. Esta es la unidad que más vas a consultar hasta que las tuberías te salgan solas.

Comandos clave de la unidad

| Comando | Descripción | |---|---| | `comando > archivo` | Manda stdout al archivo, SOBRESCRIBIENDO lo que había. | | `comando >> archivo` | Manda stdout al archivo, AGREGANDO al final. | | `comando 2> archivo` | Manda solo los errores (stderr) al archivo. | | `comando > archivo 2>&1` | Manda salida Y errores al mismo archivo (el orden importa). | | `comando 2> /dev/null` | Descarta los errores: el agujero negro del sistema. | | `comando < archivo` | Alimenta stdin desde un archivo en vez del teclado. | | `a | b` | Pipe: la salida de `a` entra como entrada de `b`. La estrella de Unix. | | `sort` | Ordena líneas alfabéticamente. Con `-n` numérico, `-r` inverso, `-u` únicas. | | `uniq` | Elimina duplicados ADYACENTES. Siempre después de `sort`. Con `-c` cuenta. | | `sort | uniq -c` | sort -nr` | LA tubería de frecuencia: cuenta y ordena de mayor a menor. | | `cut -d',' -f2` | Extrae el campo 2 usando coma como separador. | | `cut -d':' -f1` | Extrae el campo 1 con `:` como separador (ej: usuarios de /etc/passwd). | | `tr 'A-Z' 'a-z'` | Convierte mayúsculas a minúsculas. Normaliza antes de contar. | | `tr -d ','` | Borra todas las comas del texto. | | `tr -s ' '` | Colapsa espacios múltiples en uno solo. Limpia separadores irregulares. | | `wc -l` | Cuenta líneas. Con `-w` palabras, `-c` bytes, `-m` caracteres. | | `wc -l < archivo` | Cuenta líneas sin mostrar el nombre del archivo (salida limpia). | | `tee archivo` | Muestra en pantalla Y guarda en archivo a la vez. Con `-a` agrega. | | `{ cmds; } | tee f` | Agrupa comandos para que el pipe/tee capture todo el bloque. | | `comm -13 a b` | Líneas solo en `b` (archivos ordenados). Para comparar listas. | | `set -o noclobber` | Seguro: `>` se niega a sobrescribir archivos existentes. |

Los descriptores de archivo (memorízalos)

| Número | Nombre | Descripción | |---|---|---| | 0 | stdin | Entrada estándar: por defecto, tu teclado. | | 1 | stdout | Salida estándar: por defecto, tu pantalla. Los resultados normales. | | 2 | stderr | Error estándar: por defecto, tu pantalla. Solo los mensajes de error. |

One-liners que te salvan el día

```
tail -1000 /var/log/syslog | grep -i error | head -20
```

Las últimas 1000 líneas del registro del sistema, filtradas por "error", mostrando 20. El primer vistazo profesional cuando algo falla. (El `grep -i` ignora mayúsculas: lo dominarás en la Unidad 5.)

```
cut -d' ' -f1 /var/log/nginx/access.log | sort | uniq -c | sort -nr | head -10
```

Las 10 IPs que más peticiones hicieron al servidor web. Diagnóstico de ataques y scrapers en una línea.

```
du -sh */ 2>/dev/null | sort -rh | head -10
```

Cuánto pesa cada carpeta (solo un nivel), ordenadas de mayor a menor, escondiendo errores de permiso. Tu radar cuando el disco se llena.

```
history | cut -d' ' -f5- | sort | uniq -c | sort -nr | head -10
```

Tus 10 comandos más usados, con conteo. Radiografía de tus hábitos en la terminal: lo que más usas es lo que más debes dominar.

```
cat /etc/passwd | cut -d':' -f1 | sort | tr 'a-z' 'A-Z' | head -10
```

Usuarios del sistema en mayúsculas y ordenados. Parece un juguete, pero practica la tubería completa: extraer, ordenar, transformar y limitar.

Recetas de tuberías por tarea (tu caja de herramientas)

Cada receta resuelve un problema real completo. Léelas como frases: "toma esto, ordénalo, cuenta, quédate con lo importante".

```
ps aux | sort -nr -k4 | head -10
```

Los 10 procesos que más memoria consumen (la columna 4 de `ps aux` es %MEM). Cuando la máquina se arrastra, esta tubería encuentra al culpable en segundos.

```
ls -la /etc | tr -s ' ' | cut -d' ' -f9 | sort
```

Nombres de archivos en `/etc`, limpios y ordenados. El `tr -s ' '` colapsa los espacios del formato largo para que `cut` pueda extraer la columna 9 (el nombre) sin errores.

```
dmesg | grep -iE "error|fail" | tail -20
```

Los últimos 20 errores o fallos del arranque y el hardware del sistema. Lo primero que revisa un técnico cuando una máquina se comporta raro. (El `grep` con alternativas lo dominarás en la Unidad 5.)

```
cat /var/log/auth.log | grep "Failed" | cut -d' ' -f11 | sort | uniq -c | sort -nr  
| head -5
```

Las 5 IPs con más intentos fallidos de acceso SSH. Si ves cientos de intentos de una IP desconocida, estás viendo un ataque de fuerza bruta en vivo.

```
find /usr/share/doc -name "*.txt" | head -5 | xargs wc -l | sort -n
```

Cuenta las líneas de los primeros 5 archivos de documentación y ordénalos por tamaño. Practica cómo `xargs` conecta `find` (que no produce stdín útil para `wc`) con el contador.

Combinaciones de comandos de uso diario

```
mkdir -p ~/reportes && { echo "Reporte del $(date +%F)"; df -h | grep -E "^/dev"; }  
| tee ~/reportes/salud.txt
```

Crea tu carpeta de reportes y guarda la salud del disco con fecha, viéndola en pantalla y archivándola a la vez. El chequeo matutino de un servidor, en una línea.

```
sort datos.txt | uniq > limpios.txt && wc -l datos.txt limpios.txt
```

Limpia duplicados de un archivo y compara los conteos antes y después: la prueba de que la limpieza funcionó y cuánto ruido eliminó.

```
cut -d',' -f1 ventas.csv | sort | uniq -c | sort -nr | tee top.txt | head -3
```

Ranking de la primera columna de un CSV: lo guarda completo en `top.txt` y te muestra solo el top 3 en pantalla. Análisis y archivo en una sola pasada gracias a `tee`.

Resumen de la unidad

- Todo programa tiene 3 flujos: stdin (0, entrada), stdout (1, salida normal) y stderr (2, errores).
- `>` guarda la salida sobrescribiendo; `>>` agrega al final; `<` alimenta desde un archivo.
- `2>` redirige solo errores; `> archivo 2>&1` manda todo al mismo archivo; `/dev/null` descarta.
- El pipe `|` conecta la salida de un comando con la entrada del siguiente: la estrella de Unix.
- La filosofía: comandos pequeños que hacen una cosa bien, encadenados en fábricas de procesamiento.
- `sort` ordena (-n numérico, -r inverso, -u único); `sort | uniq -c` cuenta repeticiones.
- `cut` extrae columnas (-d separador, -f campos); `tr` traduce/borra caracteres; `wc` cuenta líneas/palabras/bytes.
- `tee` bifurca: muestra en pantalla y guarda en archivo a la vez; -a para agregar.
- Las llaves `{ }` agrupan comandos para tratarlos como una unidad ante un pipe o redirección.
- Piensa en flujos: inspecciona cada etapa de la tubería por separado cuando algo salga mal.
- El orden en `> archivo 2>&1` importa: primero se define a dónde va stdout y luego stderr lo sigue.
- No todos los comandos leen stdin: si un pipe "no hace nada", verifica que el de la derecha acepte entrada.
- `sort -n` ordena por valor numérico: sin `-n`, el 10 queda antes que el 2.
- `uniq` solo elimina duplicados ADYACENTES: siempre ordénalos antes con `sort`.
- `cut -d',' -f2` extrae la segunda columna de un CSV: `-d` declara el separador, `-f` elige campos.

- ``tr 'A-Z' 'a-z'`` normaliza mayúsculas antes de contar: "Error" y "error" dejan de ser dos cosas distintas.
- ``wc -l < archivo`` cuenta líneas sin mostrar el nombre del archivo: salida limpia para tus reportes.
- Construye tuberías de izquierda a derecha probando cada etapa: una tubería se depura por partes, no de un golpe.
- ``comm`` compara dos archivos ordenados: ``-13`` muestra lo nuevo, ``-23`` lo desaparecido, ``-12`` lo común.

UNIDAD 5: grep, sed y awk

Ya sabes mover datos con tuberías y filtros básicos. Ahora vienen las tres herramientas pesadas del procesamiento de texto en Unix: grep para buscar, sed para editar en flujo y awk para procesar columnas como una base de datos. Son el trío que todo administrador y programador usa a diario, y dominarlas te pone en otro nivel.

En esta unidad vas a aprender grep con sus opciones esenciales y expresiones regulares básicas, sed para sustituir, borrar e insertar texto sin abrir un editor, y awk para trabajar con campos, filtrar filas y hasta hacer aritmética. Todo con ejemplos reales que puedes ejecutar.

Por qué importa esta unidad: los registros de servidores, los archivos de configuración, los CSV, los reportes: todo es texto. grep encuentra la aguja, sed la transforma y awk la analiza. Cuando un servidor falla a las 3 de la mañana, nadie abre un editor gráfico: se hace ``grep ERROR app.log | tail -20`` y en diez segundos sabes qué pasó. Esa velocidad es la diferencia entre un susto y un desastre.

Hay otra razón: estas herramientas son el puente hacia la programación. Las expresiones regulares que aprendas con grep te servirán en Python, JavaScript y cualquier lenguaje. La lógica de patrones de awk es la misma que la de las hojas de cálculo. No estás aprendiendo tres comandos: estás aprendiendo a pensar en patrones, y eso te sirve en toda tu carrera.

Cómo encaja en el curso: la Unidad 4 te dio las tuberías; esta te da los procesadores que van dentro de ellas. En las Unidades 6 a 8, tus scripts usarán grep, sed y awk internamente para analizar archivos y tomar decisiones. En el proyecto final, tu script de respaldo los usará para filtrar qué respaldar y generar reportes.

Empecemos. Al terminar vas a buscar, transformar y analizar texto con la precisión de un cirujano.

Explicación detallada

grep: el buscador incansable

grep significa "global regular expression print": busca líneas que coincidan con un patrón y las imprime. Su forma básica:

```
grep "error" app.log
```

Muestra todas las líneas de app.log que contienen "error". Opciones esenciales:

```
grep -i "error" app.log
grep -n "error" app.log
grep -v "error" app.log
grep -r "TODO" ~/practica-linux
grep -c "error" app.log
```

La `-i` ignora mayúsculas/minúsculas (encuentra ERROR, Error y error). La `-n` muestra el número de línea (para ir directo al problema). La `-v` invierte: muestra las que NO coinciden (útil para quitar ruido). La `-r` busca recursivamente en carpetas enteras. La `-c` solo cuenta cuántas coinciden.

grep brilla en tuberías:

```
dmesg | grep -i "usb"
ps aux | grep nginx
```

El primero busca mensajes del sistema sobre USB; el segundo encuentra el proceso nginx entre todos los procesos. Estos dos patrones los usarás toda la vida.

Expresiones regulares básicas

Una expresión regular (regex) es un patrón de búsqueda con superpoderes. Las piezas básicas:

```
grep "^ERROR" app.log
grep "fin$" app.log
grep "col.r" datos.txt
grep "a*" texto.txt
```

El `^` ancla al inicio de la línea (líneas que EMPIEZAN con ERROR). El `$` ancla al final (líneas que TERMINAN con "fin"). El `.` es cualquier carácter (color, calor). El `*` significa "zero o más del anterior".

Con `grep -E` (extendido) ganas `+` (uno o más), `?` (opcional), `|` (o) y paréntesis:

```
grep -E "[0-9]+" archivo.txt
grep -E "error|fallo" app.log
grep -E "^[0-9]{3}-[0-9]{4}$" telefonos.txt
```

Los corchetes definen conjuntos: `[0-9]` es cualquier dígito, `[a-z]` cualquier minúscula. Las llaves `{3}` dicen cuántas veces. El último ejemplo valida teléfonos con formato 555-1234: tres dígitos, guion, cuatro dígitos. Las regex son un lenguaje universal: lo que aprendas aquí te sirve en Python, JavaScript y editores.

sed: el editor sin pantalla

sed significa "stream editor": edita texto en flujo, sin abrir ventana. Su uso estrella es sustituir:

```
sed 's/error/ERROR/g' app.log
```

La sintaxis `s/qué/con qué/banderas`: sustituye "error" por "ERROR". La `g` al final significa "global": todas las ocurrencias de cada línea, no solo la primera. Sin `g`, solo cambia la primera por línea.

sed no modifica el archivo original: imprime el resultado. Para guardar, rediriges:

```
sed 's/2024/2025/g' reporte.txt > reporte-nuevo.txt
```

(O la opción `-i` edita el archivo en su lugar, pero es peligrosa: no hay deshacer. Úsala solo con respaldo previo.)

Más trucos de sed:

```
sed '/^#/d' config.txt
sed -n '10,20p' archivo.txt
sed 's/^/>>> /' lista.txt
```

El primero borra (`d`) las líneas que empiezan con # (comentarios): limpia configuraciones. El segundo imprime (`p`) solo las líneas 10 a 20: extrae rangos. El tercero agrega ">>> " al inicio (`^`) de cada línea: etiqueta líneas. sed es el bisturí del texto: cortes precisos sin abrir el archivo.

awk: columnas con cerebro

awk procesa texto dividido en campos (columnas). Por defecto separa por espacios:

```
awk '{print $1}' usuarios.txt
```

Imprime el primer campo de cada línea. ``$1`` es campo 1, ``$2`` el 2, ``$0`` es la línea completa, ``NF`` es cuántos campos tiene la línea:

```
awk '{print $1, $3, $NF}' datos.txt
ps aux | awk '{print $2, $11}'
```

El segundo ejemplo extrae el PID (campo 2) y el comando (campo 11) de la lista de procesos: administración real en una línea.

Con ``-F`` cambias el separador:

```
awk -F: '{print $1, $6}' /etc/passwd
```

El archivo de usuarios usa ``:`` como separador: extraes usuario y carpeta personal. Para CSV: ``awk -F, '{print $2}' datos.csv``.

awk también filtra con condiciones:

```
awk '$3 > 100' numeros.txt
awk -F: '$3 >= 1000' /etc/passwd
```

El primero imprime líneas donde el tercer campo sea mayor que 100; el segundo lista usuarios normales (ID mayor o igual a 1000) del sistema. Y puede calcular:

```
awk '{suma += $2} END {print suma}' ventas.txt
```

Suma la columna 2 de todo el archivo e imprime el total al final (END se ejecuta cuando termina). Acabas de hacer una hoja de cálculo con una línea.

El trío en acción combinada

La verdadera potencia sale de combinarlos:

```
grep -v "^#" config.conf | grep -v "^$" | awk -F= '{print $1}'
```

Limpia un archivo de configuración: quita comentarios (``^#``), quita líneas vacías (``^$``), y extrae los nombres de las opciones antes del ``=``. Tres herramientas, una línea, configuración limpia.

```
sed 's/,/;/g' datos.csv | awk -F';' '{print $2}' | sort | uniq -c
```

Cambia el separador del CSV, extrae la columna 2, cuenta valores únicos. Esto es procesamiento de datos real, del que se cobra por hora.

Ejemplos paso a paso

Ejemplo 1: cacería de errores en un registro

Vas a crear un registro simulado y a diagnosticarlo como un profesional.

```
cd ~/practica-linux
printf "[INFO] servidor iniciado\n[ERROR] no se pudo abrir config\n[INFO]
reintentando\n[WARNING] memoria al 85%%\n[ERROR] timeout en base de datos\n[INFO]
todo estable\n" > app.log
cat app.log
```

Ahora el diagnóstico:

```
grep "ERROR" app.log
grep -n "ERROR" app.log
grep -c "ERROR" app.log
```

Primero ves los errores, luego con número de línea para ubicarlos, luego el conteo: 2 errores. El flujo profesional es exactamente ese: ver, ubicar, cuantificar.

```
grep -v "INFO" app.log
grep -i "warning|error" app.log
```

Quitás el ruido informativo (-v) y buscas warnings o errores sin importar mayúsculas. En un registro real de miles de líneas, estos filtros son la diferencia entre encontrar el problema y ahogarte en datos.

Explicación: el diagnóstico con grep sigue un embudo: del total al subconjunto relevante, del subconjunto al detalle. Nunca leas un registro completo: fíltralo hasta que solo quede lo importante.

Ejemplo 2: limpieza quirúrgica con sed

Vas a normalizar un archivo desordenado sin abrirlo en un editor.

```
cd ~/practica-linux
printf "  Nombre: Ana  \n# este es un comentario\nNombre: Luis\n\nNombre: María\n"
> sucio.txt
cat -A sucio.txt | head -8
```

`cat -A` muestra los caracteres invisibles (\$ al final de línea, espacios visibles): ves el desorden real, incluyendo espacios de más y la línea vacía.

```
sed 's/^ *//; s/ *$//' sucio.txt
```

Quita espacios al inicio (^ *) y al final (*\$) de cada línea: el `;` separa dos sustituciones en un solo comando.

```
sed '/^#/d; /^$/d' sucio.txt
```

Borra comentarios y líneas vacías. Y todo junto, guardando el resultado:

```
sed 's/^ *//; s/ *$//; /^#/d; /^$/d' sucio.txt > limpio.txt
cat limpio.txt
```

Cuatro operaciones en una pasada, archivo limpio guardado, original intacto.

Explicación: sed es ideal para limpiezas repetibles. Si mañana te llega otro archivo sucio con el mismo formato, ejecutas la misma línea y listo. Lo que haces una vez a mano, sed lo hace un millón de veces igual.

Ejemplo 3: análisis de datos con awk

Vas a analizar un reporte de ventas como lo haría un analista.

```
cd ~/practica-linux
printf
"producto,cantidad,precio\nlaptop,2,15000\nmouse,10,250\nteclado,5,800\nmonitor,3,4
500\nmouse,7,250\n" > ventas.csv
cat ventas.csv
```

Extrae columnas:

```
awk -F, 'NR>1 {print $1, $2}' ventas.csv
```

`NR>1` salta el encabezado (NR es el número de registro/línea). Imprime producto y cantidad.

Calcula el total vendido por producto (cantidad × precio):

```
awk -F, 'NR>1 {print $1, $2*$3}' ventas.csv
```

Y el gran total de la venta:

```
awk -F, 'NR>1 {total += $2*$3} END {print "Total: $" total}' ventas.csv
```

Resultado: Total: \$51750. Verifícalo a mano si quieres: $2 \times 15000 + 10 \times 250 + 5 \times 800 + 3 \times 4500 + 7 \times 250 = 30000 + 2500 + 4000 + 13500 + 1750 = 51750$. awk no se equivoca.

Explicación: awk convierte un CSV en una mini base de datos consultable. Filtrar filas, extraer columnas y agregar totales cubre el 80 por ciento del análisis de datos tabulares. Y todo sin abrir una hoja de cálculo.

Ejemplo 4: el pipeline del administrador

Vas a armar la cadena completa que usarías para auditar un sistema.

```
cd ~/practica-linux
printf "192.168.1.10 OK\n192.168.1.11 FALLO\n192.168.1.10 OK\n192.168.1.12 FALLO\n192.168.1.11 FALLO\n192.168.1.10 FALLO\n" > accesos.log
```

Un registro de accesos con IPs y resultados. Encuentra las IPs con fallos, ordenadas por cantidad:

```
grep "FALLO" accesos.log | awk '{print $1}' | sort | uniq -c | sort -rn
```

grep filtra los fallos, awk extrae la IP, sort+uniq cuentan, sort -rn rankea. Resultado: la IP más problemática arriba.

Ahora genera un reporte limpio con sed dándole formato:

```
grep "FALLO" accesos.log | awk '{print $1}' | sort | uniq -c | sort -rn | sed 's/^*\([0-9]*\) /La IP /; s/$/ tiene fallos/'
```

Bueno, ese sed con grupos es avanzado; la versión simple y legible:

```
grep "FALLO" accesos.log | awk '{print $1}' | sort | uniq -c | sort -rn | awk '{print "IP " $2 " con " $1 " fallos"}'
```

awk también formatea la salida final. El reporte queda legible para humanos.

Explicación: este pipeline (filtrar -> extraer -> contar -> rankear -> formatear) es EL patrón del análisis de registros. Cambia "FALLO" por "404" y accesos.log por el registro de tu servidor web, y tienes el reporte de errores de un sitio real.

Errores comunes y casos reales

Buscar y transformar texto con patrones es un superpoder, pero los patrones son literales hasta lo cruel: un punto que creías literal era un comodín, un `\*` que creías "todo" era "cero o más del anterior". Los errores de esta unidad no avisan: `grep` no te dice "tu patrón está mal", simplemente te devuelve cero líneas o diez mil, y tú decides si eso tiene sentido. Por eso cada caso enseña a verificar el patrón antes de confiar en él.

Caso 1: el patrón sin comillas que el shell se come

Síntoma: ejecutas `grep \*.log errores.txt` o `grep [0-9] datos.txt` y obtienes resultados absurdos, o un error raro como "grep: a.log: No existe el fichero".

Por qué ocurre: sin comillas, el SHELL expande los comodines antes de que `grep` vea el patrón. `grep \*.log errores.txt` se convierte en `grep a.log b.log errores.txt`: grep cree que `a.log` es el patrón y el resto archivos. Con `[0-9]`, si hay archivos de un dígito en tu carpeta, el patrón se reemplaza por sus nombres.

Solución: el patrón de `grep` (y de `sed`) SIEMPRE va entre comillas. Es la regla número uno de esta unidad.

```
grep *.log errores.txt      # MAL: el shell expande *.log primero
grep "ERROR" errores.txt    # BIEN: comillas protegen el patrón
grep -E "[0-9]+" datos.txt  # BIEN: los corchetes viajan intactos hasta grep
grep 'usuario=ana' app.log   # comillas simples también protegen (mejor aún)
```

Lección profesional: si un `grep` se comporta de forma "imposible", lo primero que se revisa son las comillas. Esta regla la vas a aplicar también en `find`, en `sed` y en todo lo que reciba patrones.

Caso 2: el punto que no era un punto

Síntoma: buscas `grep "192.168.1.10" accesos.log` y entre los resultados aparece una línea con `192x168y1z10`. "¿Cómo llegó eso ahí?", te preguntas.

Por qué ocurre: en regex, el ``.` significa "cualquier carácter", no "punto". Tu patrón dice "192, cualquier cosa, 168, cualquier cosa, 1, cualquier cosa, 10", y ``192x168y1z10`` cumple perfectamente. En logs grandes, esto mete ruido silencioso en tus resultados.

Solución: usa ``[.]`` para un punto literal (truco limpio, sin barras invertidas), o el punto escapado ``\.`` si prefieres la forma clásica.

```
grep "192.168.1.10" accesos.log      # RUIDOSO: el . es comodín
grep "192[.]168[.]1[.]10" accesos.log # BIEN: [.] es un punto de verdad
grep -E "192\.168\.1\.10" accesos.log # BIEN también: punto escapado
```

Lección profesional: cada vez que tu patrón incluya puntos (IPs, dominios, versiones como ``2.0``, decimales), pregúntate si los quieres literales. El 90% de las veces sí, y el 90% de la gente lo olvida.

Caso 3: ``*`` no significa "cualquier cosa"

Síntoma: ejecutas ``grep "a*" archivo.txt`` esperando "líneas que contienen a" y te devuelve TODAS las líneas del archivo, incluidas las que no tienen ninguna a. O ``grep "error*"`` también coincide con "erro" sin la r final.

Por qué ocurre: en regex, ``*`` significa "zero o más veces EL CARÁCTER ANTERIOR". ``a*`` se lee "zero o más aes", y zero aes lo cumple cualquier línea (incluso la vacía). No es el comodín del shell: es un cuantificador del carácter previo.

Solución: si quieres "una a seguida de cualquier cosa", escribe ``a.*`` (el ``.`` es "cualquier carácter" y el ``*`` lo repite). Y para "una o más aes", usa ``a+`` con ``grep -E``.

```
grep "a*" archivo.txt      # MAL: coincide con todo (zero aes vale)
grep "a.*" archivo.txt     # BIEN: una a seguida de lo que sea
grep -E "a+" archivo.txt   # BIEN: una o más aes (al menos una)
grep -E "colou?r" texto.txt # el ? = zero o una vez: color y colour
```

Lección profesional: la confusión entre el ``*`` del shell y el ``*`` de regex es legendaria y le pasa a todos. Grábatelo: en regex, el ``*`` nunca va solo al inicio de lo que quieres decir; siempre califica al carácter anterior.

Caso 4: ``sed`` que solo reemplaza la primera vez

Síntoma: ejecutas ``sed 's/error/fallo/' reporte.txt`` sobre una línea que dice "error: error de error" y el resultado es "fallo: error de error". Solo cambió la primera ocurrencia.

Por qué ocurre: sin la bandera `g` (global), `sed` sustituye únicamente la PRIMERA coincidencia de cada línea y pasa a la siguiente. Es el comportamiento por defecto y es intencional, pero sorprende a quien espera un reemplazo total.

Solución: agrega `g` al final cuando quieras todas las ocurrencias. Y verifica siempre con `grep -c` cuántas quedaban antes y después.

```
echo "error: error de error" | sed 's/error/fallo/'      # da "fallo: error de error"
echo "error: error de error" | sed 's/error/fallo/g'     # da "fallo: fallo de fallo"
grep -c "staging" config.txt                            # ¿cuántas quedan por
cambiar?
sed 's/staging/produccion/g' config.txt | grep -c "staging" # debe dar 0
```

Lección profesional: antes de un reemplazo masivo, cuenta las ocurrencias esperadas. Si `grep -c` dice 15 y después del `sed` siguen apareciendo, te faltó la `g`. Contar antes y después es el control de calidad del reemplazo.

Caso 5: `sed -i` sin respaldo (el punto sin retorno)

Síntoma: ejecutas `sed -i 's/http/https/g' sitio.conf`, el comando no muestra nada, y al revisar el archivo descubres que tu patrón también cambió líneas que no debía (por ejemplo, comentarios o URLs de documentación). El original ya no existe.

Por qué ocurre: `-i` edita el archivo EN SU LUGAR, sin pedir confirmación y sin guardar copia. Si el patrón fue más amplio de lo que creías, el daño está hecho y no hay deshacer.

Solución: usa `-i.bak` para que `sed` guarde automáticamente una copia con extensión `.bak` antes de modificar. Y antes del `-i`, prueba el reemplazo SIN `-i` para ver el resultado en pantalla.

```
sed 's/http/https/g' sitio.conf | head -20      # PRIMERO: mira el resultado sin tocar
nada
sed -i.bak 's/http/https/g' sitio.conf          # DESPUÉS: edita guardando respaldo
ls sitio.conf*                                # verifica: sitio.conf y
sitio.conf.bak
diff sitio.conf.bak sitio.conf                 # revisa exactamente qué cambió
```

Lección profesional: el protocolo es "probar en pantalla, respaldar, aplicar, comparar". Los cuatro pasos toman un minuto y te salvan de horas de reconstrucción. Un `sed -i` sin respaldo previo es como un `rm` sin `ls` previo: imprudencia.

Caso 6: `awk` que parte mal las columnas del CSV

Síntoma: tienes `ventas.csv` con `producto,cantidad,precio` y ejecutas `awk '{print \$2}' ventas.csv` esperando las cantidades, pero obtienes líneas enteras o valores sin sentido.

Por qué ocurre: `awk` por defecto separa campos por ESPACIOS (y tabuladores), no por comas. En `laptop,2,15000` no hay espacios, así que toda la línea es el campo `\$1` y `\$2` no existe. El CSV necesita que declares la coma con `-F','`.

Solución: siempre declara el separador con `-F` según tu archivo: `-F','` para CSV, `-F:` para `/etc/passwd`, `-F';` para CSV europeos.

```
awk '{print $2}' ventas.csv          # MAL: sin -F parte por espacios
awk -F',' '{print $2}' ventas.csv    # BIEN: declara la coma, $2 = cantidad
awk -F',' '{print $1, $3}' ventas.csv # producto y precio
awk -F':' '{print $1}' /etc/passwd   # usuarios del sistema
```

Lección profesional: ante un `awk` que "da campos raros", el diagnóstico es idéntico al de `cut`: mira el separador real con `head -1` y decláralo con `-F`. Separador mal declarado, columnas mal extraídas: siempre.

Caso real 1: cazar la causa de un error en producción

Son las 2 AM, la aplicación falla y el log tiene 200,000 líneas. Nadie lo lee completo: se diagnostica con embudo, de lo general a lo específico, todo con `grep`.

```
grep -c "ERROR" app.log              # ¿cuántos errores hay? dimensiona el
problema
grep -n "ERROR" app.log | tail -5     # los 5 últimos, con número de línea
grep -B3 -A8 "timeout en base" app.log | head -40 # el error con 3 líneas antes y
8 después
grep -E "^[ERROR\\.]*timeout" app.log | cut -d' ' -f2 | sort | uniq -c
```

La última tubería te dice a qué horas ocurrieron los timeouts. Con esos cuatro comandos pasaste de 200,000 líneas a "los timeouts empezaron a las 01:47": eso es diagnóstico profesional con regex, y es exactamente el flujo que se usa en incidentes reales.

Caso real 2: migración de URLs en un despliegue

Vas a pasar la aplicación de staging a producción y hay 40 archivos de configuración con `api.staging.empresa.com`. Hacerlo a mano es lento y propenso a olvidos; con `sed` es un comando, pero con protocolo de seguridad:

```

grep -rl "api\.staging\.empresa\.com" config/    # ¿qué archivos lo contienen? (-r
recursivo, -l solo nombres)
grep -rc "api\.staging\.empresa\.com" config/ | grep -v ":0"    # ¿cuántas
ocurrencias por archivo?
sed -i.bak 's/api\.staging\.empresa\.com/api.empresa.com/g' config/*.conf
grep -rn "staging" config/ | head                # verificación: ¿quedó algún
staging?

```

Fíjate en el patrón profesional completo: localizar, contar, respaldar (`-i.bak`), aplicar y verificar. Si algo sale mal, los `.bak` restauran todo en segundos. Así se hacen migraciones sin miedo.

Miniproyecto guiado de la unidad

Vas a montar una **Central de Inteligencia de Registros**: generarás el registro de una aplicación simulada (con niveles, usuarios, acciones, IPs y tiempos de respuesta) y construirás el arsenal completo de análisis: `grep` con opciones, regex de precisión, `sed` para limpiar y anonimizar, y `awk` para reportes con cálculos. Al terminar tendrás un informe de operación digno de un equipo de sistemas.

Paso 1: genera el registro de la aplicación

Crea `app.log` con 50 líneas que simulen la operación real: usuarios normales, algunas advertencias y varios errores. Usa `printf` agregando por bloques.

```

cd ~
mkdir -p central-logs && cd central-logs
printf "[INFO] 09:01 usuario=ana accion=login ip=192.168.1.10 tiempo=120ms\n" >
app.log
printf "[INFO] 09:02 usuario=luis accion=reporte ip=192.168.1.11 tiempo=850ms\n" >>
app.log
printf "[WARN] 09:03 usuario=ana accion=exportar ip=192.168.1.10 tiempo=2500ms
detalle=lento\n" >> app.log
printf "[ERROR] 09:04 usuario=maria accion=pago ip=10.0.0.5 tiempo=0ms
detalle=timeout\n" >> app.log
printf "[INFO] 09:05 usuario=ana accion=logout ip=192.168.1.10 tiempo=90ms\n" >>
app.log
printf "[ERROR] 09:06 usuario=luis accion=pago ip=10.0.0.5 tiempo=0ms
detalle=timeout\n" >> app.log

```

Agrega unas 44 líneas más con el mismo formato: mezcla `[INFO]`, `[WARN]` y `[ERROR]` (que los errores sean como 10), varios usuarios (`ana`, `luis`, `maria`, `pedro`), acciones

(`login`, `reporte`, `pago`, `exportar`, `logout`), un par de IPs sospechosas (`10.0.0.5` insiste mucho en `pago`), y tiempos variados (los `[WARN]` con tiempos altos, los `[ERROR]` con `tiempo=0ms`).

Verificación: `wc -l app.log` debe decir 50 o más, `grep -c "ERROR" app.log` debe dar tu número de errores, y `head -3 app.log` debe mostrar el formato exacto. Si el formato varía entre líneas, tus patrones van a fallar después: la consistencia del formato es la base de todo análisis.

Paso 2: domina `grep` con todas sus opciones

Haz el reconocimiento del registro usando cada opción importante de `grep`. Anota los resultados: son la materia prima de tu informe.

```
grep -c "\[ERROR\]" app.log      # ¿cuántos errores? (los corchetes se escapan)
grep -c "\[WARN\]" app.log      # ¿cuántas advertencias?
grep -c "\[INFO\]" app.log      # ¿cuántas líneas informativas?
grep -n "timeout" app.log       # ¿en qué líneas aparece "timeout"?
grep -i "error" app.log | head -3 # -i ignora mayúsculas: ¿qué más captura?
grep -v "\[INFO\]" app.log | wc -l # todo MENOS lo informativo: el ruido real
```

Verificación: la suma de los tres conteos (`-c` de cada nivel) debe dar el total de líneas (`wc -l app.log`). Si no cuadra, alguna línea tiene un formato distinto: encuéntrala con `grep -v` y corrígela. Este chequeo de "las partes suman el todo" es control de calidad básico de todo análisis.

Paso 3: regex de precisión sobre el registro

Ahora extrae información con patrones. Cada comando responde una pregunta de detective.

```
grep -o "ip=[0-9][0-9]*[.][0-9][0-9]*[.][0-9][0-9]*[.][0-9][0-9]*" app.log | sort |
uniq -c | sort -nr
```

Esa es LA regex de IP sin barras invertidas: cuatro grupos de dígitos separados por `[.]` (punto literal). El `-o` muestra solo lo que coincidió. La tubería completa te da el ranking de IPs.

```

grep -o "usuario=[a-z]*" app.log | sort | uniq -c | sort -nr      # ranking de
usuarios
grep "^\[ERROR\]" app.log                                     # líneas que EMPIEZAN
con [ERROR]
grep "tiempo=0ms$" app.log | wc -l                            # líneas que
TERMINAN en tiempo=0ms
grep -E "detalle=(timeout|lento)" app.log                     # errores o lentos
(alternativa con |)

```

Verificación: debes poder responder "¿cuál es la IP más activa?", "¿qué usuario tiene más errores?" y "¿cuántas operaciones terminaron en 0ms?". Desarma cada patrón en voz alta: qué significa cada pieza. Si no puedes explicar tu regex, no la entiendes todavía.

Paso 4: limpia y anonimiza con `sed`

Los registros reales no se comparten tal cual: las IPs son datos sensibles. Practica la anonimización y la limpieza con `sed`, SIEMPRE sobre copias, nunca sobre el original.

```

sed -E 's/[0-9][0-9]*[.][0-9][0-9]*[.][0-9][0-9]*[.][0-9][0-9]*/IP-OCULTA/g'
app.log > app-anonimo.log
head -3 app-anonimo.log      # verifica: las IPs deben decir IP-OCULTA
grep -c "192[.]" app-anonimo.log # debe dar 0: no quedó ninguna IP real

```

Ahora una limpieza: quita el campo `detalle=` para un reporte limpio, y normaliza los niveles a minúsculas para unificar formato.

```

sed 's/ detalle=[^ ]*//g' app.log > app-limpio.log
sed -E 's/\[(INFO|WARN|ERROR)\]/[\L\1]/g' app-limpio.log | head -3

```

Espera: esa segunda usa secuencias especiales de `sed` que son avanzadas. Si te confunde, hazlo simple y directo, que también es válido y más legible:

```

sed -e 's/\[INFO\]/[info]/g' -e 's/\[WARN\]/[warn]/g' -e 's/\[ERROR\]/[error]/g'
app-limpio.log > app-final.log
head -3 app-final.log

```

Verificación: `app-anonimo.log` no debe contener IPs reales, y `app-final.log` debe tener los niveles en minúsculas y sin campo detalle. Compara con `diff app.log app-final.log | head -20` para ver exactamente qué transformaciones aplicaste.

Paso 5: reportes con cálculos usando `awk`

Aquí `awk` brilla: no solo extrae, también calcula. Construye las métricas de tu informe.

```
awk '{print $1}' app.log | sort | uniq -c      # conteo por nivel (el campo 1 es
[INFO], [WARN]...)
grep -o "tiempo=[0-9]*ms" app.log | grep -o "[0-9]*" | awk '{suma+=$1; n++} END
{print "promedio:", suma/n "ms", "muestras:", n}'
```

La segunda tubería extrae los tiempos, los suma y calcula el promedio. Desármala por etapas para verificar cada pieza.

```
awk '$1=="[ERROR]" {print $3}' app.log | sort | uniq -c | sort -nr      # usuarios con
más errores
awk '$1=="[WARN]"' app.log | grep -o "tiempo=[0-9]*ms" | grep -o "[0-9]*" | sort
-nr | head -3
```

La última te da los 3 tiempos más altos entre las advertencias: tus operaciones más lentas. Verificación: debes poder responder "¿cuál es el tiempo promedio de respuesta?", "¿qué usuario acumula más errores?" y "¿cuál fue la operación más lenta?". Si un número te parece imposible (promedio de 0, por ejemplo), revisa que tu extracción de tiempos no esté capturando los `0ms` de los errores: decide conscientemente si los incluyes o los filtras.

Paso 6: el pipeline estrella de la central

Combina todo en la tubería definitiva: filtra errores, extrae usuarios, cuenta, ordena y formatea el resultado con `sed`.

```
grep "\[ERROR\]" app.log | awk '{print $3}' | sort | uniq -c | sort -nr | sed -E
's/^ *([0-9]+) usuario=([a-z]+)/El usuario \2 tiene \1 errores/'
```

Lee la tubería como frase: "de las líneas de error, toma el campo usuario, cuenta cuántas veces aparece cada uno, ordena de mayor a menor, y reescribe cada línea como una frase legible". El `sed` final usa `\\1` y `\\2` (lo capturado por cada paréntesis) para armar el mensaje.

Verificación: la salida debe ser una lista de frases como "El usuario maria tiene 4 errores", ordenada. Si el `sed` no transforma nada, revisa que los grupos `(\\...)` de tu `grep -E`... digo, de tu `sed -E` coincidan con el formato real de la entrada (espacios iniciales incluidos: por eso el patrón empieza con `^ \*`).

Paso 7: genera el informe final

Arma `informe.txt` con todas tus métricas usando agrupación y `tee`, como en la Unidad 4 pero con el arsenal de esta unidad.

```
{
echo "=== INFORME DE OPERACION ==="
echo ""
echo "Total de eventos: $(wc -l < app.log)"
echo "Errores: $(grep -c "\[ERROR\]" app.log) | Advertencias: $(grep -c "\[WARN\]" app.log)"
echo ""
echo "Top de usuarios con errores:"
grep "\[ERROR\]" app.log | awk '{print $3}' | sort | uniq -c | sort -nr
echo ""
echo "Top de IPs activas:"
grep -o "ip=[0-9][0-9]*[.][0-9][0-9]*[.][0-9][0-9]*[.][0-9][0-9]*" app.log | sort |
uniq -c | sort -nr | head -5
echo ""
echo "Tiempo promedio de respuesta:"
grep -o "tiempo=[0-9]*ms" app.log | grep -o "[0-9]*" | awk '{s+=$1; n++} END {print
s/n " ms sobre " n " muestras"}'
} | tee informe.txt
```

Los `$(...)` ejecutan el comando y pegan su resultado dentro del `echo`: así tu informe se genera solo, sin copiar números a mano. Verificación: `cat informe.txt` debe mostrar todas las secciones con números coherentes entre sí (los errores del encabezado deben coincidir con las líneas del top). Si algo no cuadra, depura esa métrica por separado.

Paso 8: tu bitácora de patrones reutilizables

Guarda en `patrones.txt` los patrones que construiste, cada uno con una explicación en tus palabras y un ejemplo de qué captura y qué NO captura. Mínimo estos cinco:

1. IP: `[0-9][0-9]*[.][0-9][0-9]*[.][0-9][0-9]*[.][0-9][0-9]*` (captura 10.0.0.5, no captura "version 2.0")
2. Nivel al inicio: `^\[ERROR\]` (captura líneas que empiezan con [ERROR])
3. Final con tiempo cero: `tiempo=0ms$`
4. Alternativa: `detalle=(timeout|lento)` con `grep -E`
5. Campo usuario: `usuario=[a-z]*` con `grep -o`

Verificación: para cada patrón, prueba un caso que SÍ debe coincidir y uno que NO, usando ``echo "prueba" | grep "patrón"``. Si tu patrón captura lo que no debe (como el punto sin escapar del Caso 2), anótalo como advertencia en tu bitácora. Una bitácora de patrones con sus trampas documentadas vale más que una lista de patrones "perfectos".

Checklist de entrega

```
cd ~/central-logs
test -f app.log && echo "registro OK"
test -f informe.txt && echo "informe OK"
test -f patrones.txt && echo "bitácora OK"
test -f app-anonimo.log && echo "anonimizado OK"
grep -c "192[.]" app-anonimo.log      # debe ser 0
wc -l app.log                         # 50 o más líneas
```

Reto final (sin solución, te toca a ti)

Sin mirar tus notas: 1) escribe el patrón ``grep -E`` para validar horas en formato ``HH:MM`` (de ``00:00`` a ``23:59``) y pruébalo contra ``23:59``, ``24:00`` y ``9:30`` (pista: los rangos ``[0-2][0-9`` necesitan refinarse para las horas 20-23 contra 24-29); 2) con ``sed``, transforma ``usuario=ana`` en ``user(ana)`` en TODO el log sin modificar el original y guarda el resultado en otro archivo; 3) con ``awk``, encuentra la ACCIÓN más lenta del registro (no el tiempo suelto: la línea completa de la acción con mayor ``tiempo=)``). Si resuelves las tres, los patrones ya piensan por ti.

Práctica

En `~/practica-linux`. Piensa en patrones: ¿qué forma tiene lo que busco?

Ejercicio 1: grep de precisión

Enunciado: crea un archivo ``mezcla.txt`` con 12 líneas que contengan las palabras "Error", "ERROR", "error", "aviso" y "todo bien" repartidas. Luego: encuentra todas las líneas con alguna variante de "error" sin importar mayúsculas; muéstralas con número de línea; cuenta cuántas son; y muestra las que NO contienen "error".

Solución explicada paso a paso. Paso 1: crea el archivo con printf y las 12 líneas variadas. Paso 2: ``grep -i "error" mezcla.txt`` encuentra todas las variantes. Paso 3: ``grep -in "error" mezcla.txt`` agrega números de línea. Paso 4: ``grep -ci "error" mezcla.txt`` las cuenta. Paso 5: ``grep -vi "error" mezcla.txt`` muestra las que no coinciden. Paso 6: anota el combo ``grep -in``: es el que más usarás para depurar, porque te dice qué y dónde.

Ejercicio 2: regex de validación

Enunciado: crea ``correos.txt`` con 8 líneas: 5 correos válidos con formato `usuario@dominio.com` y 3 líneas inválidas (sin @, sin dominio, con espacios). Usa `grep -E` para mostrar solo los válidos.

Solución explicada paso a paso. Paso 1: crea el archivo con líneas como ``ana@mail.com``, ``sin-arroba.com``, ``pedro@``, ``con espacios@mail.com``, ``luis@empresa.org``. Paso 2: el patrón: ``grep -E "[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$" correos.txt``. Paso 3: desarma el patrón: ``^`` inicio, ``[a-zA-Z0-9._-]+`` usuario de uno o más caracteres válidos, ``@`` literal, dominio similar, ``\.`` punto literal, ``[a-zA-Z]{2,}`` terminación de 2 o más letras, ``$`` fin. Paso 4: verifica que solo salgan los 5 válidos. Paso 5: prueba romper el patrón a propósito (quita el ``^``) y observa cómo se cuelan inválidos: los anclajes son los que dan precisión.

Ejercicio 3: sed de rescate

Enunciado: crea ``precios.txt`` con 6 líneas formato "producto: \$precio" pero con inconsistencias: algunas líneas con espacios de más, una con el precio en formato "\$1,500" con coma, y una línea de comentario con #. Produce ``precios-limpios.txt`` sin comentario, sin espacios extra y sin comas en los números.

Solución explicada paso a paso. Paso 1: crea el archivo desordenado con printf. Paso 2: ``/^#/d`` borra el comentario. Paso 3: ``s/^ */; s/ *$//`` quita espacios en bordes; ``s/ *: */: /g`` normaliza los dos puntos con un solo espacio. Paso 4: ``s/,//g`` quita las comas de los miles. Paso 5: todo junto: ``sed '/^#/d; s/,//g; s/^ */; s/ *$//; s/ *: */: /g' precios.txt > precios-limpios.txt``. Paso 6: compara con ``diff precios.txt precios-limpios.txt`` (diff muestra diferencias entre archivos: otro comando que vale oro). El orden de las operaciones importa: primero borra líneas, luego limpia caracteres.

Ejercicio 4: awk analista

Enunciado: con ``ventas.csv`` del Ejemplo 3, muestra: los productos con cantidad mayor a 5; el producto más caro (mayor precio unitario); y la cantidad total de artículos vendidos.

Solución explicada paso a paso. Paso 1: ``awk -F, 'NR>1 && $2>5 {print $1}' ventas.csv``: filtra filas con cantidad mayor a 5. Paso 2: el más caro: ``awk -F, 'NR>1 {if($3>max){max=$3; prod=$1}} END{print prod, max}' ventas.csv``: guarda el máximo mientras recorre. Paso 3: total de artículos: ``awk -F, 'NR>1 {total+=$2} END{print total}' ventas.csv``. Paso 4: verifica los tres resultados contra el archivo a ojo. Paso 5: anota el patrón ``NR>1`` (saltar encabezado) y la estructura ``{acumular} END {reportar}``: son los dos moldes de awk que más vas a usar.

Ejercicio 5: el limpiador de configuración

Enunciado: crea un archivo ``app.conf`` de mentira con 15 líneas: comentarios (`#`), líneas vacías, y opciones formato ``clave = valor`` con espacios inconsistentes. Produce ``app-limpia.conf`` con solo las opciones, formato ``clave=valor`` (sin espacios alrededor del `=`).

Solución explicada paso a paso. Paso 1: crea `app.conf` con printf incluyendo de todo. Paso 2: ``grep -v "^#" app.conf | grep -v "^$"``: quita comentarios y vacías. Paso 3: ``sed 's/ *= */=/g``: normaliza el igual sin espacios. Paso 4: todo junto: ``grep -v "^#" app.conf | grep -v "^$" | sed 's/ *= */=/g' > app-limpia.conf``. Paso 5: ``cat app-limpia.conf`` para verificar. Paso 6: este pipeline es reutilizable: guárdalo en tu cheat sheet como "limpiador de configs". En la Unidad 8 lo convertirás en una función de tu biblioteca de scripts.

Tarjeta de referencia rápida

Tu acordeón de búsqueda y procesamiento de texto. La tabla de regex es la que más vas a consultar: guárdala donde la veas.

Comandos clave de la unidad

| Comando | Descripción | |---|---| | ``grep "patrón" archivo`` | Muestra las líneas que coinciden (el patrón siempre entre comillas). | | ``grep -i`` | Ignora mayúsculas y minúsculas al buscar. | | ``grep -n`` | Numera las líneas coincidentes. Para ubicar rápido en archivos grandes. | | ``grep -v`` | Invierte: muestra las líneas que NO coinciden. | | ``grep -c`` | Cuenta coincidencias en vez de mostrarlas. | | ``grep -r`` | Busca recursivamente en todos los archivos bajo una carpeta. | | ``grep -l`` | Muestra solo los NOMBRES de archivos con coincidencias. | | ``grep -o`` | Muestra solo la parte que coincidió, no la línea completa. | | ``grep -A3 -B3`` | Muestra 3 líneas después (-A) y 3 antes (-B) de cada coincidencia: contexto. | | ``grep -E`` | Modo extendido: ``+``, ``?``, ``\|`` alternativo y grupos sin barras invertidas. | | ``grep -F`` | Patrones literales, sin regex: para texto con ``[``, ``.`` o ``*`` que no quieres interpretar. | | ``sed 's/a/b/g`` | Sustituye todas las ``a`` por ``b`` en cada línea (sin ``g``, solo la primera). | | ``sed -i.bak 's/a/b/g`` | Edita el archivo en su lugar, guardando respaldo ``.bak`` automático. | | ``sed '/patrón/d`` | Borra las líneas que coinciden con el patrón. |

Tabla de expresiones regulares

One-liners que te salvan el día

```
sed -i.bak 's/192\.\168\.\1\.\10/SERVIDOR-1/g' reporte.txt && diff reporte.txt.bak
reporte.txt | head
```

Anonimiza una IP en un reporte (guardando respaldo) y muestra qué cambió. El protocolo completo "respaldar, aplicar, verificar" en una línea.

```
awk -F',' '{ventas[$1]+=$3} END {for (p in ventas) print p, ventas[p]}' ventas.csv  
| sort -nr -k2
```

Suma las ventas (campo 3) agrupadas por producto (campo 1) y ordena por total. Un reporte agregado digno de hoja de cálculo, en una tubería.

```
grep -oE "[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+[.][a-zA-Z]{2,}" contactos.txt | sort -u
```

Extrae todos los correos válidos de un archivo desordenado, sin duplicados. El patrón de correo que aparece en entrevistas, listo para usar.

Patrones listos para copiar y adaptar

Estos patrones resuelven búsquedas típicas de todos los días. Cópielos, pruébalos con tus datos y adáptalos: un patrón probado vale más que uno inventado.

```
grep -E "^[0-9]{2}:[0-9]{2}" agenda.txt
```

Líneas que empiezan con una hora (`09:30`, `14:05`). El `^` ancla al inicio para no capturar horas en medio del texto.

```
grep -E "[0-9]{3}-[0-9]{3}-[0-9]{4}" clientes.txt
```

Teléfonos en formato `555-123-4567`. Cambia los `{3}` según el formato de tu país.

```
grep -E "https?://[^\ ]+" notas.md
```

URLs en un archivo markdown: `http` con `s` opcional (`?`), `://` literal y "todo hasta el espacio" (`[^\ ]+`).

```
sed -E 's/[ \t]+$//g' sucio.txt > limpio.txt
```

Quita los espacios y tabuladores al final de cada línea. La limpieza invisible que evita errores en comparaciones y scripts.

```
awk 'length($0) > 120' texto.txt
```

Muestra las líneas de más de 120 caracteres. Para encontrar líneas anormalmente largas en registros o código.

Combinaciones de comandos de uso diario

```
grep -r "password" ~/proyectos 2>/dev/null | grep -v ".git" | head -10
```

Busca la palabra "password" en tus proyectos, escondiendo errores de permiso y excluyendo la carpeta `.git`. Auditoría rápida de secretos expuestos en código.

```
sed 's/^#.*//' config.conf | grep -v "^$" | sort -u > config-limpia.conf
```

Quita comentarios, elimina líneas vacías y ordena sin duplicados: un archivo de configuración limpio y auditable en una tubería.

Resumen de la unidad

- `grep` busca líneas por patrón: `-i` ignora mayúsculas, `-n` numera, `-v` invierte, `-r` busca en carpetas, `-c` cuenta.
- Las regex básicas: `^`` inicio, `$`` fin, `.`` cualquier carácter, `*`` cero o más, `[]`` conjuntos, `{n}`` repeticiones.
- `grep -E`` activa el modo extendido: `+``, `?``, `|`` y paréntesis para patrones complejos.
- `sed` edita en flujo: `s/qué/con qué/g`` sustituye; `/patrón/d`` borra líneas; `-n '10,20p`` extrae rangos.
- `sed` no toca el original: redirige a un archivo nuevo; `-i`` edita en su lugar (peligroso, respalda antes).
- `awk` trabaja por campos: `$1``, `$2``, `$NF``, `NF``; `-F`` define el separador; `NR`` es el número de línea.
- `awk` filtra (`$3 > 100``), calcula (`{suma += $2}``) y reporta (`END {print ...}``): una mini base de datos.
- El pipeline clásico: `grep` filtra -> `awk` extrae -> `sort/uniq` cuentan -> `sed/awk` formatean.
- Diagnostica con embudo: del total al subconjunto, del subconjunto al detalle; nunca leas registros completos.
- Las regex y los patrones son universales: te sirven en Python, JavaScript y todos los lenguajes.

- ``grep -A3 -B3`` muestra contexto alrededor de cada coincidencia: el error con su antes y su después.
- ``grep -v`` invierte la búsqueda: a veces es más fácil decir lo que NO quieres ver.
- En regex, ``.`` es "cualquier carácter": para un punto literal usa ``\[.]``, el truco sin barras invertidas.
- ``*`` significa "cero o más del carácter anterior", no "cualquier cosa": ``a*`` también coincide con líneas sin ninguna a.
- Sin la ``g``, ``sed s/a/b/`` solo cambia la primera ocurrencia de cada línea; la ``g`` es la que hace el reemplazo total.
- En el reemplazo de ``sed``, ``&`` significa "todo lo que coincidió": ``s/error/[&]/g`` envuelve cada error en corchetes.
- ``awk -F`,`` declara la coma como separador: sin ``-F``, awk parte por espacios y los campos salen mal.
- ``$0`` en awk es la línea completa y ``NF`` cuántos campos tiene: ``NF != 5`` caza líneas malformadas al instante.
- ``grep -E`` te da ``+``, ``?``, ``|`` y grupos ``( )`` sin barras invertidas: para patrones complejos, siempre modo extendido.

UNIDAD 6: Variables y tu primer script

Llegó el momento que estabas esperando: vas a dejar de ser usuario de comandos para convertirte en creador de programas. Un script bash es un archivo de texto con una secuencia de comandos que el shell ejecuta en orden, y con las herramientas de esta unidad (variables, argumentos, códigos de salida) tus scripts van a empezar a comportarse como programas de verdad.

En esta unidad vas a aprender qué son las variables y cómo usarlas sin los errores clásicos de espacios y comillas, las tres formas de entrecomillar y cuándo usar cada una, la sustitución de comandos, los argumentos que recibe tu script (\$1, \$2, \$@, \$#), los códigos de salida y cómo leer la entrada del usuario con read. Y vas a escribir, dar permiso y ejecutar tus primeros scripts completos.

Por qué importa esta unidad: todo lo anterior fueron comandos sueltos; esto es programación. Un script guarda tu conocimiento en un archivo reutilizable: lo que hoy haces en 20 comandos, mañana lo haces ejecutando un archivo. Los administradores viven de scripts: respaldos, monitoreo, despliegues, limpieza. Cada script que escribas es tiempo futuro que te ahorras.

Hay otra razón: esta unidad te enseña la disciplina que separa los scripts de juguete de los profesionales. El shebang correcto, los permisos de ejecución, los códigos de salida con significado, los mensajes de error útiles. Un script bien hecho se puede programar con cron, lo puede ejecutar otra persona y falla de forma informativa. Un script mal hecho solo funciona en tu máquina cuando tú lo miras.

Cómo encaja en el curso: las Unidades 1 a 5 te dieron los comandos; esta te enseña a empaquetarlos en programas. Las Unidades 7 y 8 agregarán decisiones, repeticiones y funciones: la lógica completa. La Unidad 9 programará tus scripts con cron. Y el proyecto final será el script más grande que hayas escrito.

Empecemos. Al terminar vas a escribir scripts que reciben datos, toman decisiones básicas y reportan si les fue bien o mal.

Explicación detallada

Variables: cajitas con nombre

Una variable guarda un valor para usarlo después. En bash se crea así:

```
nombre="Ana"  
edad=25
```

Regla de oro número uno: sin espacios alrededor del `=`. `nombre = "Ana"` falla porque el shell interpreta cada parte como un comando separado. Esta es la causa del 50 por ciento de los errores de principiantes en scripts.

Para usar el valor, le pones `\$` al nombre:

```
echo "Hola, $nombre. Tienes $edad años."
```

Imprime: Hola, Ana. Tienes 25 años. Las llaves sirven para delimitar el nombre cuando va pegado a más texto:

```
echo "${nombre}2024"
```

Sin llaves, `\$nombre2024` buscaría una variable llamada "nombre2024" que no existe. Con llaves, queda claro dónde termina el nombre.

Analogía: una variable es una caja etiquetada. `nombre="Ana"` mete "Ana" en la caja etiquetada "nombre". `\$nombre` es abrir la caja y usar lo que hay dentro. Y las cajas de bash guardan texto: todo es texto hasta que demuestres lo contrario.

Las tres comillas: cuándo usar cada una

Bash tiene tres formas de entrecomillar y confundirlas causa errores sutiles:

```
nombre="Ana"  
echo "Hola $nombre"  
echo 'Hola $nombre'
```

Las comillas dobles `""` permiten la expansión: `\$nombre` se convierte en "Ana". Las comillas simples `` ` ` ` son literales: imprimen `Hola \$nombre` tal cual, sin expandir nada. Regla práctica: usa dobles cuando quieras que las variables se expandan (casi siempre), simples cuando quieras texto literal exacto.

La tercera forma es `\$()`, la sustitución de comandos: ejecuta lo de adentro y usa su resultado como valor:

```
hoy=$(date +%Y-%m-%d)
echo "Hoy es $hoy"
```

Guarda la fecha en la variable `hoy`. La forma vieja con backticks `` `` `` hace lo mismo pero es menos legible: prefiere `\$( )` siempre.

Error clásico: `echo 'Hoy es \$(date) '` imprime el texto literal porque las comillas simples bloquean la sustitución. Si algo "no se expande", revisa tus comillas primero.

Argumentos: tu script recibe datos

Los scripts profesionales no tienen los datos escritos dentro: los reciben como argumentos al ejecutarse:

```
./saluda.sh Ana 25
```

Dentro del script, `\$1` es el primer argumento ("Ana"), `\$2` el segundo ("25"), `\$0` es el nombre del propio script, `\$#` es cuántos argumentos llegaron, y `\$@` son todos los argumentos juntos:

```
#!/bin/bash
echo "Hola, $1. Tienes $2 años."
echo "Me pasaste $# argumentos."
```

Si lo ejecutas sin argumentos, `\$1` queda vacío y el saludo sale raro. En la Unidad 7 aprenderás a validar que los argumentos existan antes de usarlos.

`\$@` brilla en bucles (Unidad 7): procesa "todos los que me pasen" sin saber cuántos son. Es como decirle al script "haz esto con cada cosa de la lista", sea una o cien.

Códigos de salida: el semáforo de los programas

Todo comando, al terminar, devuelve un número al shell: 0 significa éxito, cualquier otro número significa algún error. Es el semáforo universal de Linux:

```
ls /existe
echo $?
ls /no-existe
echo $?
```

El primero imprime 0 (éxito), el segundo imprime 2 (error). La variable especial ` \$? ` guarda el código de salida del último comando ejecutado. ¡Ojo! ` \$? ` cambia con CADA comando: si ejecutas ` echo \$? ` dos veces seguidas, la segunda muestra el código del primer echo (que fue 0). Guárdalo de inmediato si lo necesitas: ` codigo=\$? `.

Tus scripts también devuelven códigos con ` exit `:

```
#!/bin/bash
if [ ! -f "$1" ]; then
    echo "Error: el archivo no existe" >&2
    exit 1
fi
echo "El archivo existe"
exit 0
```

El ` >&2 ` manda el mensaje de error al canal de errores (stderr), como debe ser. Los códigos con significado (0=éxito, 1=error genérico, 2=uso incorrecto) permiten que otros scripts y cron sepan si tu script funcionó. Un script que siempre devuelve 0 aunque falle es un mentiroso: nadie puede confiar en él para automatizar.

read: conversar con el usuario

read pausa el script y espera que el usuario escriba algo:

```
#!/bin/bash
echo -n "¿Cómo te llamas? "
read nombre
echo "Hola, $nombre. Bienvenido a bash."
```

La opción ` -p ` pone el mensaje en el mismo comando: ` read -p "¿Cómo te llamas? " nombre `. Y ` -s ` oculta lo que se escribe (para contraseñas): ` read -sp "Contraseña: " pass `.

read es para scripts interactivos. Pero ojo: un script que se va a ejecutar solo con cron no puede preguntar nada porque no hay nadie para responder. En la Unidad 8 verás cómo diseñar scripts que funcionen en ambos modos.

El shebang y el ritual completo

La primera línea de todo script bash debe ser:

```
#!/bin/bash
```

Se llama shebang y le dice al sistema "ejecuta este archivo usando /bin/bash". Sin ella, el sistema adivina con qué ejecutar tu archivo, y adivinar en producción es mala idea.

El ritual completo de un script nuevo:

```
nano miscript.sh
```

Escribes el script con su shebang. Luego:

```
chmod u+x miscript.sh
./miscript.sh
```

Permiso de ejecución y a correr. Si estás en la carpeta del script, el `./` es obligatorio (la carpeta actual no está en el PATH por seguridad). Si quieres ejecutarlo desde donde sea sin `./`, muévelo a una carpeta del PATH como `~/bin` o `/usr/local/bin`, o agrega su carpeta al PATH. Por ahora, `./` es tu amigo.

Ejemplos paso a paso

Ejemplo 1: tu primer script útil

Vas a crear un script que genera un reporte del sistema, usando variables y sustitución de comandos.

```
#!/bin/bash
# reporte.sh - Genera un reporte básico del sistema

fecha=$(date +%Y-%m-%d)
usuario=$(whoami)
disco=$(df -h / | tail -1 | awk '{print $4}')

echo "=== Reporte del $fecha ==="
echo "Usuario: $usuario"
echo "Espacio libre en /: $disco"
echo "Reporte generado correctamente."
```

Guárdalo como `~/practica-linux/reporte.sh`, dale permiso y ejecútalo:

```
chmod u+x ~/practica-linux/reporte.sh
~/practica-linux/reporte.sh
```

Cada variable captura el resultado de un comando: la fecha, tu usuario y el espacio libre (df, última línea, cuarto campo). El script imprime el reporte armado. Ya no son comandos sueltos: es un programa que produce un informe.

Explicación: observa el patrón "capturar con \$(), usar con \$variable". Es el esqueleto de todos los scripts de monitoreo. En la Unidad 9, este script se ejecutará solo todas las mañanas con cron.

Ejemplo 2: script con argumentos

Un script que saluda de forma personalizada según lo que le pases.

```
#!/bin/bash
# saluda.sh - Saluda usando argumentos

echo "Script: $0"
echo "Hola, $1 $2."
echo "Recibí $# argumentos en total."
echo "Todos fueron: $@"
```

Ejecútalo de varias formas:

```
chmod u+x ~/practica-linux/saluda.sh
~/practica-linux/saluda.sh Ana Torres
~/practica-linux/saluda.sh Pedro
~/practica-linux/saluda.sh
```

Con dos argumentos saluda completo; con uno, el `\$2` queda vacío; sin argumentos, todo queda vacío pero no falla. En la Unidad 7 aprenderás a exigir los argumentos y a mostrar ayuda si faltan.

Explicación: `\$0`, `\$1`, `\$2`, `\$#` y `\$@` son el vocabulario de los scripts flexibles. Un script sin argumentos es una receta fija; con argumentos es una herramienta que se adapta.

Ejemplo 3: códigos de salida con significado

Un script que verifica si un archivo existe y reporta honestamente el resultado.

```
#!/bin/bash
# existe.sh - Verifica si un archivo existe

if [ -f "$1" ]; then
    echo "El archivo '$1' existe."
    exit 0
else
    echo "Error: el archivo '$1' no existe." >&2
    exit 1
fi
```

Pruébalo y observa los códigos:

```
chmod u+x ~/practica-linux/existe.sh
~/practica-linux/existe.sh ~/practica-linux/reporte.sh; echo "Código: $?"
~/practica-linux/existe.sh /no/existe.txt; echo "Código: $?"
```

El `;` separa dos comandos en la misma línea. El primero devuelve 0, el segundo 1. El mensaje de error fue al canal de errores (no lo ves distinto aquí, pero si rediriges la salida normal, el error sigue apareciendo: prueba `./existe.sh /no/existe.txt > /dev/null` y verás que el error sobrevive porque va por el canal 2).

Explicación: el `if [ -f ... ]` lo desmenuzaremos en la Unidad 7; por ahora quédate con el patrón: verificar, informar por el canal correcto y salir con el código honesto. Este script ya es un componente confiable para automatizar.

Ejemplo 4: script interactivo con read

Un asistente que crea tu ficha de programador preguntándote datos.

```
#!/bin/bash
# ficha.sh - Crea tu ficha de programador

read -p "Tu nombre: " nombre
read -p "Tu lenguaje favorito: " lenguaje
read -p "¿Años programando? " anios

archivo="$HOME/practica-linux/ficha.txt"
{
echo "=== Ficha de programador ==="
echo "Nombre: $nombre"
echo "Lenguaje: $lenguaje"
echo "Experiencia: $anios años"
echo "Fecha: $(date +%Y-%m-%d)"
} > "$archivo"

echo "Ficha guardada en $archivo"
cat "$archivo"
```

Ejecútalo, responde las preguntas y verifica el archivo creado. Observa las llaves `{}` agrupando para redirigir todo de un golpe (Unidad 4) y las variables entrecomilladas con dobles comillas: `"\$archivo"` protege contra rutas con espacios.

Explicación: este script combina entrada (read), variables, sustitución de comandos y redirección agrupada. Es un programa completo e interactivo. La lección silenciosa: entrecomilla SIEMPRE tus variables con dobles comillas; el día que una ruta tenga un espacio, te acordarás de este consejo.

Errores comunes y casos reales

Estos son los errores que vas a cometer sí o sí. Mejor que los veas aquí, con su diagnóstico, y no a las 2 de la mañana con un script roto en producción.

Caso 1: espacios alrededor del `=` en la asignación

Síntoma: escribes `nombre = "Ana"` y el shell responde `nombre: orden no encontrada`.

Por qué ocurre: el shell separa todo por espacios antes de interpretar. `nombre` lo toma como un comando a ejecutar, `=` como su primer argumento y `"Ana"` como el segundo. No existe el comando `nombre`, así que falla. En bash la asignación es pegada o no es asignación.

Solución: quita los espacios. Y si el valor tiene espacios, las comillas van pegadas al `=`:

```
nombre="Ana"
ciudad="Ciudad de México"
echo "$nombre vive en $ciudad"
```

Truco de diagnóstico: cuando veas "orden no encontrada" y la "orden" es el nombre de tu variable, el 99 por ciento de las veces hay un espacio de más alrededor del `=`.

Caso 2: comillas simples que "se tragan" tus variables

Síntoma: haces `echo 'Hola, \$nombre` esperando `Hola, Ana` y obtienes literalmente `Hola, \$nombre`.

Por qué ocurre: las comillas simples son literales a muerte: no expanden variables, no ejecutan `\$( )`, no interpretan nada. Es su trabajo, no un bug. El bug es usarlas cuando querías expansión.

Solución: usa dobles cuando necesites que el shell sustituya valores:

```
nombre="Ana"
echo 'Hola, $nombre'    # imprime: Hola, $nombre (literal)
echo "Hola, $nombre"    # imprime: Hola, Ana (expansión)
echo "Hoy es $(date +%A)" # las dobles sí permiten $( )
```

Regla de memoria: simples = "déjalo como está", dobles = "procesa lo que hay dentro".

Caso 3: variables sin entrecomillar que se rompen con espacios

Síntoma: tu script funciona con `archivo=notas.txt` pero explota con `archivo="mis notas.txt"`: comandos que reciben dos argumentos donde esperaban uno, o peor, `rm` borrando lo que no debía.

Por qué ocurre: sin comillas, el shell divide el valor en palabras (word splitting) y además expande comodines (globbing). `rm \$archivo` con `archivo="mis notas.txt"` se convierte en `rm mis notas.txt`: intenta borrar dos archivos que no existen... o que sí existen y no querías tocar.

Solución: entrecomilla SIEMPRE las expansiones de variables con dobles comillas:

```
archivo="mis notas.txt"
cp "$archivo" "$archivo.bak" # bien: un solo argumento
rm "$archivo"                # bien
# rm $archivo <-- MAL: se parte en dos palabras
```

Este es el error que más daño hace en scripts reales. Acostúmbrate: cada vez que escribas ``$variable``, pregúntate si lleva comillas. La respuesta casi siempre es sí.

Caso 4: leer ``$?`` demasiado tarde

Síntoma: tu script verifica si un comando falló, pero la verificación siempre dice que todo salió bien (o siempre dice lo contrario de la realidad).

Por qué ocurre: ``$?`` contiene el código de salida del ÚLTIMO comando ejecutado, y cada comando lo sobrescribe, incluyendo un inocente ``echo`` de depuración que metiste en medio.

Solución: captura ``$?`` inmediatamente en una variable, en la línea siguiente al comando:

```
grep "ERROR" /var/log/app.log > /dev/null
codigo=$? # captúralo AQUÍ, en la línea inmediata
if [ "$codigo" -ne 0 ]; then
    echo "No se encontraron errores (o falló grep)" >&2
fi
```

Versión rota típica: ``grep ... ; echo "buscando..." ; if [ $? -eq 0 ]`` - ese ``$?`` ya es el código del ``echo``, no del ``grep``. Siempre.

Caso 5: ``$*`` y ``$@`` sin comillas pierden argumentos con espacios

Síntoma: tu script recibe `./saluda.sh "María José" "Juan Pérez"` pero adentro los nombres llegan partidos: ``$#`` dice 4 en vez de 2.

Por qué ocurre: ``$*`` junta todos los argumentos en una sola cadena separada por espacios, y ``$@`` sin comillas también se parte por espacios. Solo ``"$@"`` (entrecomillado) preserva cada argumento original como una unidad.

Solución:

```
#!/bin/bash
echo "Recibí $# argumentos:"
for persona in "$@"; do # "$@" preserva "María José" como uno solo
    echo " - Hola, $persona"
done
```

Memoria rápida: ``$*`` = "todo revuelto", ``$@`` sin comillas = "todo partido", ``"$@"`` = "cada argumento intacto". En scripts serios, ``"$@"`` es la única forma correcta.

Caso 6: "permiso denegado" al ejecutar tu script

Síntoma: `./mi_script.sh` responde ``permiso denegado``, o lo ejecutas y parece que bash ignora tu shebang.

Por qué ocurre: dos causas: (a) el archivo no tiene permiso de ejecución (te faltó ``chmod u+x`` o ``chmod +x``); (b) el shebang está mal escrito (``#!/bin/bash`` con espacio después de ``#`` suele funcionar, pero ``#!/bin/sh`` te cambia el intérprete y cosas como ``[[]]` dejan de funcionar).

Solución:

```
chmod u+x mi_script.sh    # o chmod +x mi_script.sh
./mi_script.sh             # ahora sí
head -1 mi_script.sh      # verifica: debe decir #!/bin/bash
```

Y ojo con ejecutar con ``sh mi_script.sh``: sh no es bash en muchos sistemas (es dash), y tu script puede fallar en cosas que en bash funcionaban. Si tu shebang dice bash, ejecútalo como bash.

Caso real 1: el despliegue que borró la carpeta equivocada

En una empresa pequeña, el script de despliegue hacía ``rm -rf $DESTINO/tmp`` donde ``DESTINO`` venía de una variable de entorno. Un día el pipeline corrió sin esa variable definida: ``DESTINO`` quedó vacía y el comando se convirtió en ``rm -rf /tmp``. Se llevó archivos temporales de otros servicios y tiró dos aplicaciones.

La lección profesional: nunca operes con variables que pueden venir vacías sin protegerte. La sintaxis ``${var:?mensaje}`` aborta el script con tu mensaje si la variable está vacía o sin definir:

```
#!/bin/bash
rm -rf "${DESTINO:?ERROR: la variable DESTINO no está definida}"/tmp
echo "Limpieza de ${DESTINO}/tmp terminada"
```

Con eso, si ``DESTINO`` viene vacía el script muere en esa línea con tu mensaje, en vez de borrar lo que no debe. En scripts que tocan ``rm -rf``, esta es armadura obligatoria.

Caso real 2: el respaldo que "funcionaba" y no respaldaba nada

Un administrador tenía un cron que corría ``respaldo.sh`` cada noche y el log decía "respaldo completado". El día que se necesitó restaurar, el archivo ``tar.gz`` pesaba 200 bytes: el script hacía ``tar -czf $DESTINO/respaldo.tar.gz $ORIGEN`` sin comillas, y como ``$ORIGEN`` contenía

un espacio (`/datos/ventas 2024`), tar recibía dos rutas inexistentes, fallaba... pero el script nunca revisaba `\$?` e imprimía "completado" de todos modos.

Dos fallas clásicas juntas: variable sin entrecomillar y código de salida ignorado. La versión profesional:

```
#!/bin/bash
tar -czf "$DESTINO/respaldo.tar.gz" "$ORIGEN"
if [ $? -ne 0 ]; then
    echo "ERROR: falló la creación del respaldo" >&2
    exit 1
fi
echo "Respaldo completado: $DESTINO/respaldo.tar.gz"
```

Moraleja: un script que no verifica sus códigos de salida miente. Y en respaldos, una mentira cuesta datos.

Miniproyecto guiado de la unidad

Vas a construir `ficha.sh`: un script que crea la ficha de un proyecto nuevo. Pide datos al usuario (o los recibe como argumentos), crea la estructura de carpetas, genera un `LEEME.md` con fecha y autor, y reporta todo con códigos de salida honestos. Usa variables, comillas, `\$()`, argumentos, `\$?`, `read` y shebang: toda la unidad en un solo programa útil.

Paso 1: el esqueleto con shebang y validación de argumentos

Crea el archivo `~/practica-linux/ficha.sh`:

```
#!/bin/bash
# ficha.sh - crea la ficha inicial de un proyecto
# Uso: ./ficha.sh [nombre-del-proyecto]

nombre="${1:-}"
if [ -z "$nombre" ]; then
    read -p "Nombre del proyecto: " nombre
fi

if [ -z "$nombre" ]; then
    echo "ERROR: necesitas dar un nombre de proyecto" >&2
    echo "Uso: $0 [nombre-del-proyecto]" >&2
    exit 1
fi
```

Qué está pasando: `\${1:-}` toma el primer argumento o cadena vacía si no hay. Si sigue vacío, `read -p` lo pide de forma interactiva. Si ni así hay nombre, error a stderr y `exit 1`. Tres niveles de robustez en diez líneas.

Paso 2: variables, fecha y rutas seguras

Agrega debajo:

```
base="$HOME/practica-linux/proyectos"
carpeta="$base/$nombre"
fecha=$(date '+%Y-%m-%d %H:%M')
autor="${USER:-desconocido}"

if [ -e "$carpeta" ]; then
    echo "ERROR: ya existe $carpeta" >&2
    exit 2
fi

mkdir -p "$carpeta/docs" "$carpeta/src" || {
    echo "ERROR: no se pudo crear $carpeta" >&2
    exit 3
}
```

Observa: todas las variables entrecomilladas (`"\$carpeta"`) sobrevive a nombres con espacios), `\${date ...}` captura la fecha, `\${USER:-desconocido}` da valor por defecto, y cada `exit` usa un código distinto (1 = sin nombre, 2 = ya existe, 3 = no se pudo crear). Códigos distintos = diagnóstico sin abrir el script.

Paso 3: generar el LEEME.md con sustitución y redirección

```
cat > "$carpeta/LEEME.md" <<FIN
# $nombre

- Creado: $fecha
- Autor: $autor
- Carpeta: $carpeta

## Descripción

_Pendiente: escribe aquí de qué trata este proyecto._

## Estructura

- `src/`: código fuente
- `docs/`: documentación
FIN

echo "Proyecto '$nombre' creado en $carpeta"
exit 0
```

El heredoc `<<FIN` escribe varias líneas de una vez; como el delimitador no está entrecomillado, las variables SÍ se expanden adentro. (Si lo entrecomillaras como `<<'FIN'`, saldría el texto literal `\$nombre`. Útil cuando generes plantillas.)

Paso 4: permiso de ejecución y primera prueba

```
chmod u+x ~/practica-linux/ficha.sh
cd ~/practica-linux
./ficha.sh mi-app
echo "Código de salida: $?"
ls -R proyectos/mi-app
cat proyectos/mi-app/LEEME.md
```

Verificación esperada: `Código de salida: 0`, la estructura `proyectos/mi-app/{docs,src}` creada y el LEEME.md con el nombre, la fecha real y tu usuario. Si algo falla, revisa `\$?` inmediatamente después de cada comando para ubicar el paso roto.

Paso 5: prueba los caminos de error

Un script profesional se prueba también en lo que debe fallar:

```
./ficha.sh mi-app; echo "segunda vez: $?"          # debe dar 2 (ya existe)
./ficha.sh "mi super app"; echo "con espacios: $?" # debe dar 0
ls -d proyectos/"mi super app"                    # existe con espacios intactos
printf '
' | ./ficha.sh; echo "vacío: $?"                  # debe dar 1
```

Si el caso "mi super app" falla o crea dos carpetas, busca la variable sin entrecomillar: es el Caso 3 de errores comunes en acción.

Paso 6: modo interactivo puro

Ejecuta `./ficha.sh` sin argumentos y escribe el nombre cuando lo pida. Verifica que el flujo `read` funciona y que el `LEEME.md` queda igual de bien formado. Este doble modo (argumento o pregunta) es el patrón de las herramientas amigables: automatizables por argumento, usables a mano por pregunta.

Reto final (sin solución)

Extiende `ficha.sh` con estas tres mejoras, sin mirar atrás:

1. Agrega un segundo argumento opcional `$2` para la descripción del proyecto; si se da, debe aparecer en el `LEEME.md` en vez de `"_Pendiente_"`. Si no se da, pregunta con `read -p` pero permite dejarlo vacío (Enter = pendiente).
2. Crea además un archivo `src/main.sh` con shebang, permiso de ejecución y un `echo "Hola desde $nombre"` que funcione al ejecutarlo.
3. Haz que el script, al terminar con éxito, imprima un resumen de una línea con el número de archivos creados (pista: `find "$carpeta" -type f | wc -l` y captura el resultado con `$( )`).

Si logras las tres sin errores de comillas ni de `$?`, dominas la unidad.

Práctica

En `~/practica-linux`. Escribe cada script en un archivo, con shebang, permiso y prueba real.

Ejercicio 1: variables sin dolor

Enunciado: en la terminal (no en script todavía), crea las variables ``ciudad="Ciudad de México"`` y ``anio=2024``, e imprime la frase "Vivo en Ciudad de México desde 2024" usando las variables. Luego provoca a propósito los dos errores clásicos (espacios alrededor del = y comillas simples) y observa qué pasa.

Solución explicada paso a paso. Paso 1: ``ciudad="Ciudad de México"`` (sin espacios alrededor del =) y ``anio=2024``. Paso 2: ``echo "Vivo en $ciudad desde $anio"``: las dobles comillas expanden y respetan los espacios internos. Paso 3: provoca el error uno: ``ciudad = "Puebla"`` y observa el mensaje "orden no encontrada": el shell intentó ejecutar un comando llamado "ciudad". Paso 4: provoca el error dos: ``echo 'Vivo en $ciudad'``: imprime el texto literal porque las simples no expanden. Paso 5: anota en tu cheat sheet las dos reglas: "sin espacios en la asignación" y "dobles para expandir, simples para literal". Estos dos errores te van a morder al menos una vez más; cuando pase, ya sabrás dónde mirar.

Ejercicio 2: el presentador

Enunciado: escribe ``presenta.sh`` que reciba nombre y profesión como argumentos e imprima una presentación formal. Debe funcionar con 0, 1 o 2 argumentos sin romperse (usa valores por defecto si faltan).

Solución explicada paso a paso. Paso 1: crea el archivo con shebang. Paso 2: usa la sintaxis de valor por defecto: ``nombre=${1:-"visitante"}`` significa "usa \$1, y si está vacío, usa 'visitante'". Igual para profesión con "persona curiosa". Paso 3: imprime: ``echo "Les presento a $nombre, $profesion."``. Paso 4: ``chmod u+x presenta.sh`` y pruébalo con 2, 1 y 0 argumentos. Paso 5: la sintaxis ``${var:-default}`` es oro para scripts robustos: nunca dejes que una variable vacía rompa tu programa si puedes darle un valor sensato.

Ejercicio 3: el verificador honesto

Enunciado: escribe ``revisa.sh`` que reciba una ruta como argumento e informe si es un archivo regular, un directorio o no existe, saliendo con código 0 si existe (sea lo que sea) y 1 si no existe. Los errores deben ir a stderr.

Solución explicada paso a paso. Paso 1: shebang y captura del argumento en una variable: ``ruta="$1"``. Paso 2: usa las pruebas ``-f`` (archivo regular) y ``-d`` (directorio): ``[ -f "$ruta" ]``, ``[ -d "$ruta" ]``. Paso 3: estructura if/elif/else (adelanto de la Unidad 7, pero la sintaxis es legible): si es archivo imprime "es archivo", si es directorio "es directorio", si no "no existe" a stderr con ``>&2`` y ``exit 1``. Paso 4: prueba con un archivo real, una carpeta real y una ruta inventada, verificando ``$?`` después de cada una. Paso 5: este script ya es una herramienta reutilizable: en la Unidad 7 lo usarás dentro de otros scripts para validar antes de actuar.

Ejercicio 4: calculadora de propinas

Enunciado: escribe ``propina.sh`` que pida con `read` el total de la cuenta y el porcentaje de propina, calcule el monto de la propina y el total a pagar, y lo muestre con dos decimales.

Solución explicada paso a paso. Paso 1: ``read -p "Total de la cuenta: " cuenta`` y ``read -p "Porcentaje de propina: " pct``. Paso 2: `bash` no maneja decimales en aritmética nativa, así que usa `awk` o `bc`: ``propina=$(echo "$cuenta $pct" | awk '{printf "%.2f", $1*$2/100}')`. Paso 3: ``total=$(echo "$cuenta $propina" | awk '{printf "%.2f", $1+$2}')`. Paso 4: imprime: ``echo "Propina: $$propina | Total: $$total"`. Cuidado: para imprimir un signo de pesos literal antes de una variable necesitas ``$``? No: ``$`` es el PID del script. Para el signo de moneda usa: ``echo "Propina: \$$propina"` o simplemente escribe "MXN": ``echo "Propina: $propina MXN"`. Paso 5: prueba con 500 y 15: propina 75.00, total 575.00. La lección: `bash` delega los decimales a `awk/bc`; no pelees con la aritmética entera del shell.

Ejercicio 5: bitácora personal

Enunciado: escribe ``diario.sh`` que reciba un mensaje como argumento (todas las palabras juntas con "\$@") y lo agregue a `~/practica-linux/diario.txt`` con fecha y hora al inicio. Si se ejecuta sin mensaje, debe mostrar un error útil y salir con código 1.

Solución explicada paso a paso. Paso 1: shebang; verifica argumentos: ``if [ $# -eq 0 ]; then echo "Uso: $0 <mensaje>" && exit 1; fi``. Paso 2: ``mensaje="$@"`` junta todos los argumentos en una frase. Paso 3: ``echo "[$(date '+%Y-%m-%d %H:%M')] $mensaje" >> ~/practica-linux/diario.txt``. Paso 4: confirma: ``echo "Anotado."``. Paso 5: ``chmod u+x diario.sh``, pruébalo con ``./diario.sh "Hoy aprendí variables en bash"`` y sin argumentos para ver el error útil. Paso 6: ``cat ~/practica-linux/diario.txt`` para ver tu bitácora crecer. Este script ya es una herramienta diaria real: úsalo para anotar lo que aprendas en cada unidad.

Tarjeta de referencia rápida

Tu machete de la unidad. Si solo te quedas con una sección para consultar, que sea esta.

| Comando / sintaxis | Descripción | |---|---| | ``nombre="Ana"`` | Asigna valor a una variable (SIN espacios alrededor del ``=``). | | ``$nombre`` / ``${nombre}`` | Expande el valor de la variable; las llaves delimitan el nombre (``${nombre}s``). | | ``${var:-default}`` | Usa ``default`` si ``var`` está vacía o sin definir. | | ``${var:?mensaje}`` | Aborta con ``mensaje`` si ``var`` está vacía o sin definir (protección contra ``rm -rf``). | | ``$(comando)`` | Sustitución de comandos: captura la salida como texto. | | `` `` `comando` ``` | Backticks: lo mismo que ``$( )``, pero no se pueden anidar bien; prefiere ``$( )``. | | ``$0`` | Nombre del script tal como se invocó. | | ``$1``, ``$2``, ... | Argumentos

posicionales primero, segundo, etc. | | ` \$# ` | Número de argumentos recibidos. | | ` \$@ ` / ` "\$@" ` | Todos los argumentos; entrecomillado preserva los que tienen espacios. | | ` \$\* ` | Todos los argumentos revueltos en una cadena (casi nunca lo quieres). | | ` \$? ` | Código de salida del último comando (léelo de inmediato, se sobrescribe). | | ` \$\$ ` | PID del script en ejecución. | | ` exit N ` | Termina el script con código N (0 = éxito). | | ` echo "msg" >&2 ` | Imprime el mensaje en stderr (salida de error). | | ` read -p "Texto: " var ` | Pide un dato mostrando el mensaje en la misma línea. | | ` read -s -p "Clave: " pass ` | Pide sin mostrar lo tecleado (contraseñas). | | `#!/bin/bash` | Shebang: primera línea, le dice al sistema que use bash. | | ` chmod u+x script.sh ` | Da permiso de ejecución al script. | | ` ./script.sh ` | Ejecuta el script del directorio actual. | | ` export VAR="valor" ` | Hace la variable visible para los programas hijos. |

5 one-liners que vas a usar toda la vida

```
fecha=$(date +%F); echo "Hoy es $fecha"
```

Captura la salida de un comando en una variable; `%F` da la fecha como `2026-09-20`.

```
nombre="${1:-mundo}"; echo "Hola, $nombre"
```

Toma el primer argumento o usa "mundo" si no vino ninguno: scripts que no se rompen.

```
[ $# -eq 0 ] && { echo "Uso: $0 <archivo>" >&2; exit 1; }
```

Valida argumentos en una línea: si no hay, muestra el uso en stderr y sale con error.

```
read -p "¿Cómo te llamas? " nombre && echo "Hola, $nombre"
```

Pide un dato y úsalo de inmediato; `&&` solo continúa si el read tuvo éxito.

```
cp "$f" "$f.bak" && echo "respaldo ok" || echo "falló" >&2
```

Haz algo y reporta el resultado: `&&` = salió bien, `||` = salió mal. El `if` de una línea.

Más comandos que completan la unidad

| Comando / sintaxis | Descripción | |---|---| | ` unset var ` | Borra una variable (útil para limpiar temporales sensibles como passwords). | | ` \${#var} ` | Longitud de la variable en caracteres: ` \${#nombre} `. | | ` \${var:0:3} ` | Subcadena: los primeros 3 caracteres (también ` \${var: -4} ` = últimos 4). | | ` \${var/pat/rep} ` | Sustituye la primera ocurrencia de ` pat ` por ` rep ` (con `//`

sustituye todas). | | `\${var%.txt}` / `\${var##\*/}` | Quita sufijo `.txt` / quita todo hasta la última `/` (nombre base de una ruta). | | `printf "%s " "\$v"` | Imprime con formato controlado; más predecible que `echo` con caracteres raros. | | `shift` / `shift 2` | Descarta `\$1` (o los 2 primeros): así se procesan listas largas de argumentos. | | `\$REPLY` | Donde cae lo que el usuario teclea si `read` no lleva nombre de variable. | | `tmp=\$(mktemp)` | Crea un archivo temporal con nombre único y seguro (bórralo al terminar). | | `cd "\$(dirname "\$0")"` | El script se para en SU propia carpeta: las rutas relativas siempre funcionan. | | `nombre=\${1:?falta nombre}` | Como `\${var:?...}` pero para argumentos: aborta si no vino el parámetro. | | `set -- a b c` | Reinventa los argumentos del script: útil para probar funciones con datos fijos. |

3 combinaciones que valen oro

```
dir_base="$(cd "$(dirname "$0")" && pwd)"
```

Tu script siempre sabe dónde vive, sin importar desde dónde lo llamen. Base para rutas relativas confiables.

```
tmp=$(mktemp) && trap 'rm -f "$tmp"' EXIT
```

Temporal seguro + limpieza garantizada aunque el script muera: el patrón profesional para archivos intermedios.

```
usuario="${1:?Usó: $0 <usuario>}"; echo "Hola, ${usuario^^}"
```

Valida el argumento al instante y `\${usuario^^}` lo pone en MAYÚSCULAS (`,` lo pone en minúsculas). Validación + transformación en dos líneas.

Resumen de la unidad

- Las variables se asignan SIN espacios: `nombre="Ana"`; se usan con `\$`: `\${nombre}`; `\${nombre}` delimita el nombre.
- Comillas dobles expanden variables, simples son literales; `\$( )` ejecuta un comando y captura su resultado.
- Los argumentos llegan en `\$1`, `\$2`...; `\$0` es el script, \$# cuántos son, `\$@` todos juntos.
- `\${var:-default}` da un valor por defecto si la variable está vacía: scripts que no se rompen.

- Todo programa devuelve un código: 0 = éxito, otro = error; ``$?` lo contiene (léelo de inmediato).
- ``exit N`` termina tu script con código N; los errores van a stderr con ``>&2``.
- `read` pide datos al usuario: ``-p`` para el mensaje en línea, ``-s`` para ocultar (contraseñas).
- Todo script empieza con ``#!/bin/bash`` (shebang); luego ``chmod u+x`` y se ejecuta con ``.``.
- Entrecomilla siempre tus variables con dobles comillas: ```$archivo``` sobrevive a rutas con espacios.
- Un script bien hecho informa, valida y sale con códigos honestos: así se puede automatizar.
- ``${}`` es el PID de tu script y ``${}!` el del último proceso lanzado al fondo: útiles para registros y limpieza.
- Los backticks ``` ` ` ``` hacen lo mismo que ``$( )``, pero no se anidan: ``$( )`` siempre es la opción moderna.
- ```"$@"``` mantiene cada argumento intacto (incluso con espacios); ``${}`` los revuelve: en scripts serios solo usa ```"$@"```.
- ``export`` comparte variables con los programas hijos; sin export, la variable muere en tu script.
- ``${var:?mensaje de error}`` aborta si la variable está vacía: armadura obligatoria antes de un ``rm -rf``.
- ``read`` sin nombre de variable guarda en ``${}REPLY``; ``read -r`` evita que la diagonal invertida escape caracteres.
- ``shift`` recorre los argumentos (``${}2`` pasa a ser ``${}1``): así se procesan listas largas de archivos.
- El shebang decide el intérprete: ``#!/bin/bash`` no es lo mismo que ``#!/bin/sh``; ejecútalo con el que declares.
- ``exit`` sin número conserva el código del último comando: a veces es justo lo que quieres propagar.

UNIDAD 7: Condicionales y bucles

Tus scripts ya reciben datos y reportan resultados. Ahora les vas a dar cerebro: la capacidad de tomar decisiones y de repetir tareas. Con condicionales y bucles, tus scripts dejan de ser recetas lineales y se convierten en programas que se adaptan a cada situación. Esta es la unidad donde bash se vuelve un lenguaje de programación completo.

En esta unidad vas a aprender el comando test y sus formas `[ ]` y `[[ ]]`, las comparaciones de archivos, cadenas y números, la estructura if/elif/else, la selección múltiple con case, y los bucles for, while y until con break y continue para controlarlos.

Por qué importa esta unidad: la automatización real exige decisiones. "Si el respaldo de ayer existe, no lo repitas." "Para cada archivo de la carpeta, comprímelo." "Mientras el servicio no responda, espera y reintenta." Sin condicionales y bucles tendrías que hacer todo a mano, caso por caso. Con ellos, escribes la lógica una vez y el script la aplica mil veces.

Hay otra razón: aquí es donde la mayoría abandona bash por "raro". La sintaxis de `[ ]` con sus espacios obligatorios y sus operadores `-eq`, `-f`, `-z` parece caprichosa hasta que entiendes que es un comando como cualquier otro, con sus reglas. Esta unidad te quita ese miedo: al final, los corchetes te van a parecer naturales.

Cómo encaja en el curso: la Unidad 6 te dio variables y scripts; esta les da lógica. La Unidad 8 organizará esa lógica en funciones reutilizables. El proyecto final usará condicionales para decidir qué respaldar y bucles para procesar cada archivo.

Empecemos. Al terminar, tus scripts van a decidir, repetir y adaptarse como programas de verdad.

Explicación detallada

test, [] y [[]]: el juez de bash

El comando test evalúa una condición y devuelve 0 si es verdadera o 1 si es falsa. Tiene dos disfraces: `[ ]` (la forma clásica) y `[[ ]]` (la moderna, con superpoderes):

```
[ -f archivo.txt ]
echo $?
[[ -f archivo.txt ]]
echo $?
```

Ambos preguntan "¿existe archivo.txt y es un archivo regular?" y guardan la respuesta en `\$?`. La regla de oro de los corchetes: ESPACIOS por dentro. `[ -f archivo ]` funciona; `[ -f archivo ]` falla porque el shell no reconoce el comando. Piensa en `[` como un programa cuyo nombre es corchete: necesita sus espacios como cualquier comando necesita separar sus argumentos.

`[[ ]]` es la versión mejorada: maneja mejor los patrones y no se confunde con variables vacías. En scripts nuevos, prefiere `[[ ]]`. En esta unidad verás ambas porque te vas a topar con `[ ]` en scripts ajenos toda la vida.

Pruebas de archivos, cadenas y números

Las pruebas de archivos son las más usadas en administración:

```
[ -f ruta ]
[ -d ruta ]
[ -r ruta ]
[ -w ruta ]
[ -x ruta ]
[ -s ruta ]
[ -e ruta ]
```

De arriba abajo: es archivo regular, es directorio, se puede leer, se puede escribir, se puede ejecutar, tiene tamaño mayor a cero, existe (sea lo que sea). Antes de borrar, copiar o procesar algo, un script profesional verifica con estas pruebas. Es la diferencia entre "espero que exista" y "sé que existe".

Para cadenas de texto:

```
[ -z "$var" ]
[ -n "$var" ]
[ "$a" = "$b" ]
[ "$a" != "$b" ]
```

`-z` pregunta si está vacía (zero length), `-n` si NO está vacía, `=` compara igualdad, `!=` desigualdad. Las variables van entrecomilladas: si `\$var` está vacía y sin comillas, el test recibe un argumento menos y se confunde.

Para números, bash usa operadores con letras (porque `<` y `>` ya significan redirección):

```
[ "$n" -eq 5 ]
[ "$n" -ne 5 ]
[ "$n" -gt 10 ]
[ "$n" -lt 10 ]
[ "$n" -ge 10 ]
[ "$n" -le 10 ]
```

Igual (equal), distinto (not equal), mayor que (greater than), menor que (less than), mayor o igual, menor o igual. `[ \$n -gt 10 ]` funciona igual. Memoriza las letras: eq, ne, gt, lt, ge, le.

if/elif/else: tomar decisiones

La estructura de decisión:

```
if [ -f "$1" ]; then
    echo "Es un archivo."
elif [ -d "$1" ]; then
    echo "Es un directorio."
else
    echo "No existe o es otra cosa."
fi
```

Se lee casi como español: si la condición se cumple, haz esto; si no, pero se cumple esta otra, haz esto otro; si no, haz lo último. El `fi` (if al revés) cierra el bloque. El `then` va en la misma línea después de punto y coma, o en la línea siguiente.

Condiciones compuestas con `&&` (y) y `||` (o):

```
if [ -f "$archivo" ] && [ -r "$archivo" ]; then
    echo "Existe y se puede leer."
fi
```

Con `[ ]` puedes usar `&&` y `||` adentro: `[ -f "\$a" && -r "\$a" ]`. Más legible en scripts nuevos.

case: el menú de opciones

Cuando hay muchas opciones posibles, case es más limpio que una escalera de elif:

```

case "$1" in
    iniciar)
        echo "Iniciando servicio..."
        ;;
    detener)
        echo "Deteniendo servicio..."
        ;;
    estado)
        echo "Consultando estado..."
        ;;
    *)
        echo "Uso: $0 {iniciar|detener|estado}"
        exit 1
        ;;
esac

```

Cada opción termina con `;;` y `esac` (case al revés) cierra. El `\*` es el comodín: atrapa todo lo demás, perfecto para el mensaje de uso. Los patrones aceptan comodines: `\*.txt` coincide con cualquier txt, `[Ss]i` con "Si" o "si".

case es el patrón clásico de los scripts que controlan servicios: `servicio.sh iniciar`, `servicio.sh detener`. Lo verás en sistemas reales por todas partes.

for: repetir sobre listas

El bucle for recorre una lista, ejecutando el bloque una vez por elemento:

```

for archivo in *.txt; do
    echo "Procesando: $archivo"
done

```

Si hay 10 txt, el bloque corre 10 veces con `\$archivo` valiendo cada nombre. También con listas explícitas y rangos:

```

for nombre in Ana Luis María; do
    echo "Hola, $nombre"
done

for i in {1..5}; do
    echo "Intento $i"
done

```

`{1..5}` genera 1 2 3 4 5. Y la forma clásica estilo C para contar:

```
for ((i=1; i<=5; i++)); do
    echo "Vuelta $i"
done
```

for es el caballo de batalla del procesamiento por lotes: "para cada archivo, haz esto". En el proyecto final, un for recorrerá todo lo que hay que respaldar.

while y until: repetir con condición

while repite MIENTRAS la condición sea verdadera:

```
contador=1
while [ $contador -le 5 ]; do
    echo "Vuelta $contador"
    contador=$((contador + 1))
done
```

`=$(( ))` es la aritmética del shell: suma, resta, multiplica. Ojo con el peligro clásico: si olvidas incrementar el contador, el bucle es infinito. Siempre asegúrate de que la condición pueda volverse falsa.

until es el gemelo inverso: repite HASTA que la condición sea verdadera (mientras sea falsa):

```
until ping -c1 servidor > /dev/null 2>&1; do
    echo "Esperando al servidor..."
    sleep 5
done
echo "Servidor en línea."
```

"Hasta que el ping funcione, espera 5 segundos y reintentar." Este patrón de reintento es oro en automatización: los scripts profesionales no se rinden al primer fallo, esperan y reintentan.

break y continue: el control fino

break sale del bucle por completo; continue salta a la siguiente vuelta:

```
for archivo in *; do
    [ -f "$archivo" ] || continue
    echo "Archivo: $archivo"
done
```

Si no es archivo regular, continue salta al siguiente sin procesarlo. Es el filtro elegante: en lugar de anidar ifs, descartas lo que no te interesa y sigues.

```
while true; do
    read -p "Escribe 'salir' para terminar: " texto
    [ "$texto" = "salir" ] && break
    echo "Dijiste: $texto"
done
```

`while true` es un bucle infinito intencional que solo break puede detener: el patrón del menú interactivo. Lo verás en el Ejemplo 4.

Ejemplos paso a paso

Ejemplo 1: el inspector de rutas

Un script que diagnostica cualquier ruta que le pases, usando todas las pruebas de archivos.

```
#!/bin/bash
# inspecciona.sh - Diagnostica una ruta

ruta="$1"

if [ -z "$ruta" ]; then
    echo "Uso: $0 <ruta>" >&2
    exit 1
fi

if [ ! -e "$ruta" ]; then
    echo "'$ruta' no existe." >&2
    exit 1
fi

[ -f "$ruta" ] && echo "Tipo: archivo regular"
[ -d "$ruta" ] && echo "Tipo: directorio"
[ -r "$ruta" ] && echo "Lectura: sí" || echo "Lectura: no"
[ -w "$ruta" ] && echo "Escritura: sí" || echo "Escritura: no"
[ -x "$ruta" ] && echo "Ejecución: sí" || echo "Ejecución: no"

if [ -f "$ruta" ]; then
    echo "Tamaño: $(wc -c < "$ruta") bytes"
fi
exit 0
```

Pruébalo:

```
chmod u+x ~/practica-linux/inspecciona.sh
~/practica-linux/inspecciona.sh /etc/hostname
~/practica-linux/inspecciona.sh /tmp
~/practica-linux/inspecciona.sh /no/existe
~/practica-linux/inspecciona.sh
```

Observa el truco `[ cond ] && sí || no` : si la condición es verdadera ejecuta lo de la izquierda, si no, lo de la derecha. Es el if de una línea para casos simples.

Explicación: este script valida ANTES de actuar (¿me dieron argumento? ¿existe la ruta?) y luego reporta. Ese orden (validar, luego trabajar) es la marca del script profesional.

Ejemplo 2: el clasificador de archivos

Un script que organiza una carpeta por extensiones usando for y case.

```
#!/bin/bash
# clasifica.sh - Organiza archivos por tipo en una carpeta

origen="${1:-.}"
destino="$origen/organizado"

mkdir -p "$destino"/{imagenes,documentos,scripts,otros}

for archivo in "$origen"/*; do
    [ -f "$archivo" ] || continue
    nombre=$(basename "$archivo")
    case "$nombre" in
        *.jpg|*.png|*.gif)
            mv "$archivo" "$destino/imagenes/"
            ;;
        *.txt|*.md|*.pdf)
            mv "$archivo" "$destino/documentos/"
            ;;
        *.sh)
            mv "$archivo" "$destino/scripts/"
            ;;
        *)
            mv "$archivo" "$destino/otros/"
            ;;
    esac
    echo "Movido: $nombre"
done
echo "Clasificación terminada en $destino."
```

Pruébalo en una carpeta de prueba (¡no en tu zona principal!):

```
mkdir -p ~/practica-linux/caos && cd ~/practica-linux/caos
touch foto.jpg doc.txt prog.sh raro.xyz
~/practica-linux/clasifica.sh ~/practica-linux/caos
ls -R ~/practica-linux/caos/organizado
```

Cada archivo cae en su categoría por su extensión; lo desconocido va a "otros". `basename` extrae solo el nombre del archivo sin la ruta.

Explicación: for recorre, case decide, mv ejecuta. Es el patrón "clasificar y actuar" que aparece en respaldos, despliegues y limpieza. Y nota el valor por defecto `\${1:-.}`: si no le pasas carpeta, trabaja en la actual.

Ejemplo 3: el vigilante con reintentos

Un script que espera a que un archivo aparezca, con límite de intentos (para no esperar eternamente).

```
#!/bin/bash
# espera.sh - Espera un archivo con límite de intentos

archivo="$1"
max_intentos="${2:-10}"
intento=1

if [ -z "$archivo" ]; then
    echo "Uso: $0 <archivo> [intentos]" >&2
    exit 1
fi

until [ -f "$archivo" ]; do
    if [ "$intento" -gt "$max_intentos" ]; then
        echo "Tiempo agotado esperando '$archivo'." >&2
        exit 1
    fi
    echo "Intento $intento/$max_intentos: esperando '$archivo'..."
    sleep 2
    intento=$((intento + 1))
done

echo "| '$archivo' apareció!"
exit 0
```

Pruébalo: en una terminal ejecuta `~/practica-linux/espera.sh /tmp/senal.txt 5``; en otra terminal, espera unos segundos y crea el archivo con `touch /tmp/senal.txt``. El vigilante lo detecta y termina con éxito. Si no creas el archivo, agota los intentos y falla honestamente.

Explicación: `until + contador + sleep` es el patrón de espera con timeout. Los scripts que coordinan procesos (uno genera un archivo, otro lo consume) viven de este patrón. Y el límite de intentos evita el "colgado eterno", el peor fallo de un automatismo.

Ejemplo 4: menú interactivo

El clásico menú que no termina hasta que eliges salir.

```
#!/bin/bash
# menu.sh - Menú de utilidades

while true; do
    echo ""
    echo "=== MENÚ ==="
    echo "1) Ver fecha"
    echo "2) Ver espacio en disco"
    echo "3) Ver usuarios conectados"
    echo "4) Salir"
    read -p "Elige una opción: " op

    case "$op" in
        1) date ;;
        2) df -h / ;;
        3) who ;;
        4) echo "¡Hasta luego!"; break ;;
        *) echo "Opción inválida, intenta de nuevo." ;;
    esac
done
exit 0
```

Ejecútalo y juega con las opciones. `while true` + `case` + `break` es la receta del menú interactivo: infinita paciencia, decisiones limpias y una salida clara.

Explicación: este ejemplo une while, case, read y break en un programa interactivo completo. Es la base de los asistentes e instaladores por terminal: presentan opciones, ejecutan la elegida y vuelven al menú.

Errores comunes y casos reales

La lógica de control es donde los scripts dejan de ser recetas y empiezan a pensar. También es donde aparecen los bugs más tercos. Estos seis te van a pasar; aprende a olerlos.

Caso 1: `[ \$x -eq 5 ]` con variable vacía o sin entrecomillar

Síntoma: `[ : se esperaba un argumento ]` o `[ : demasiados argumentos ]`, y tu `if` se comporta al azar.

Por qué ocurre: si `x` está vacía, `[ \$x -eq 5 ]` se convierte en `[ -eq 5 ]`: a `[` le falta un operando. Si `x` tiene espacios, se convierte en varios argumentos y `[` se confunde. Los corchetes son quisquillosos con la cantidad exacta de piezas.

Solución: entrecomilla SIEMPRE las variables dentro de `[ ]`, o mejor aún, usa `[ [ ] ]` que es más tolerante:

```
x=""
if [ "$x" -eq 5 ]; then # bien entrecomillado: no truena, da falso
    echo "es cinco"
fi
if [[ $x -eq 5 ]]; then # [[ ]] también lo maneja sin sustos
    echo "es cinco"
fi
```

Caso 2: comparar números con `>` dentro de `[ ]`

Síntoma: `[ "\$a" > "\$b" ]` siempre dice que es verdadero... y además aparece un archivo vacío llamado `b` (o el nombre del valor) en tu carpeta.

Por qué ocurre: dentro de `[ ]`, `>` no es "mayor que": es una REDIRECCIÓN. El shell crea un archivo con el nombre del segundo operando antes de que `[ ]` siquiera vea la comparación. Para números se usan `<-gt`, `<-lt`, etc.

Solución:

```
a=10; b=9
if [ "$a" -gt "$b" ]; then # -gt = mayor que (números)
    echo "$a es mayor"
fi
# Para texto sí se usa > pero escapado o en [[ ]]:
if [[ "$nombre1" > "$nombre2" ]]; then # orden alfabético, sin redirección
    echo "$nombre1 va después"
fi
```

Si ves archivos misteriosos con nombres de tus variables después de correr un script, ya sabes quién fue.

Caso 3: `for` sobre `\$(ls)` o `\$(cat)` con nombres que tienen espacios

Síntoma: `for f in \$(ls)` procesa "mis" y "notas.txt" como dos archivos distintos cuando existe "mis notas.txt". Todo lo que sigue (mover, borrar, comprimir) opera sobre rutas que no existen.

Por qué ocurre: la sustitución `\$( )` sin comillas se parte por espacios, tabs y saltos de línea. El `for` no itera archivos: itera PALABRAS.

Solución: itera con globs directos (el shell los expande bien) o con ``while read``:

```
for f in *.txt; do          # el glob respeta nombres con espacios
    echo "Procesando: $f"
done

# O para listas venidas de un comando, línea por línea:
find . -name "*.log" -print0 | while IFS= read -r -d '' f; do
    echo "Log: $f"
done
```

Regla: ``for`` es para listas que TÚ escribes o globs; ``while read`` es para salida de comandos.

Caso 4: ``while read`` con tubería pierde las variables al terminar

Síntoma: cuentas líneas dentro de ``cat archivo | while read linea; do total=$((total+1)); done`` y al salir del bucle ``total`` vale 0, como si nada hubiera pasado.

Por qué ocurre: cada parte de una tubería corre en un SUBSHELL: una copia del shell. Las variables que modificas adentro mueren con la copia cuando la tubería termina. Tu ``total`` sí se incrementó... en un universo paralelo.

Solución: alimenta el ``while`` con redirección en vez de tubería, o usa sustitución de proceso:

```
total=0
while IFS= read -r linea; do
    total=$((total + 1))
done < archivo.txt          # redirección: mismo shell, variable sobrevive
echo "Total: $total"

# Alternativa cuando SÍ necesitas un comando generador:
while IFS= read -r linea; do
    total=$((total + 1))
done < <(grep "ERROR" app.log)
```

Este bug es legendario: el script "funciona" (procesa todo adentro) pero los resultados se evaporan afuera.

Caso 5: bucle infinito por condición que nunca cambia

Síntoma: el script se queda colgado, la terminal no responde, tienes que matarlo con Ctrl+C. Clásico con ``while`` y ``until``.

Por qué ocurre: la variable de la condición nunca se actualiza dentro del cuerpo, o se actualiza en la dirección contraria. `until` (repite HASTA que sea verdad) confunde más porque la lógica está invertida respecto a `while`.

Solución: verifica que algo dentro del cuerpo acerque la condición a su fin, y ponle un fusible a los bucles peligrosos:

```
intentos=0
until [ "$intentos" -ge 5 ]; do
    echo "Intento $intentos"
    intentos=$((intentos + 1)) # sin esta línea: infinito
done

# Fusible para bucles sobre cosas externas (red, archivos):
vueltas=0
while ! ping -c1 servidor > /dev/null 2>&1; do
    vueltas=$((vueltas + 1))
    [ "$vueltas" -ge 10 ] && { echo "servidor inalcanzable, abandono" >&2; exit 1; }
    sleep 2
done
```

Caso 6: `=` vs `-eq`, o mezclar comparaciones de texto y número

Síntoma: `[ 10 > 9 ]`... no. Más común: `[ "\$edad" = 25 ]` funciona "a veces", y `[ 010 -eq 10 ]` dice que son iguales cuando como texto no lo son.

Por qué ocurre: `=` compara TEXTO (carácter por carácter), `-eq` compara NÚMEROS (valor matemático). `010` como texto != `10`; como número son iguales. Y `[ "\$a" == "\$b" ]` con doble `=` solo es válido en `[[]]`; en `[ ]` el estándar es un solo `=`.

Solución:

```
edad="025"
if [ "$edad" -eq 25 ]; then echo "tiene 25 (numérico)"; fi
if [ "$edad" = "025" ]; then echo "el texto es 025"; fi
# En [[]] puedes usar == y patrones:
if [ [ "$archivo" == *.txt ] ]; then echo "es un txt"; fi
```

Memoria: números -> `-eq -ne -gt -lt`; texto -> `=` !=`; patrones comodín -> solo en `[[]]`.

Caso real 1: el script de limpieza que borró el directorio de trabajo

Un cron nocturno recorría ``for d in $(ls /datos/tmp)`` y borraba carpetas viejas con ``rm -rf``. Un usuario creó una carpeta llamada ``respaldo enero`` (con espacio). El ``for`` la partió en ``respaldo`` y ``enero``; ``rm -rf respaldo`` no existía (falló sin pena), pero la lógica de "si no existe, crea y sigue" terminó operando sobre rutas parciales y el script, en su rama de error, hizo ``cd`` a un lugar inesperado y el siguiente ``rm -rf *`` limpió el directorio de trabajo del servicio.

La versión profesional itera con glob y verifica cada paso:

```
#!/bin/bash
for d in /datos/tmp/*/*; do
    [ -d "$d" ] || continue
    echo "Limpiando: $d"
    # rm -rf "$d"    # solo cuando todo lo demás esté probado
done
```

Moraleja: ``for`` + ``$(ls)`` + ``rm -rf`` es la trinidad del desastre. Globs entrecomillados y pruebas antes de borrar.

Caso real 2: el monitoreo que spameó 400 alertas en una noche

Un script vigilaba un servicio con ``while true``: si ``pgrep`` no encontraba el proceso, mandaba un correo y hacía ``sleep 60``. El servicio se cayó a las 3 am y el SMTP tardó 90 segundos por intento... el ``sleep 60`` corría DESPUÉS del envío, así que el bucle giraba cada ~90 segundos mandando correos, pero peor: cuando el admin reinició el servicio a mano, el script siguió mandando "servicio caído" porque verificaba el nombre del proceso con ``pgrep servicio`` y el nuevo proceso se llamaba ``servicio-v2``.

Dos lecciones: pon el ``sleep`` donde controle el ritmo pase lo que pase, y verifica de forma robusta (puerto, archivo pid o respuesta real, no solo el nombre):

```
#!/bin/bash
while true; do
    if ! curl -sf http://localhost:8080/salud > /dev/null; then
        echo "$(date): servicio caído" >> /var/log/vigilante.log
        # enviar una sola alerta y esperar más antes de reintentar
    fi
    sleep 300
done
```

Miniproyecto guiado de la unidad

Vas a construir `ordena.sh`: un organizador que clasifica los archivos de una carpeta por extensión usando `for`, `case`, `if` y `[ ]`. Incluye modo de prueba (dry-run) que solo muestra lo que haría sin mover nada. Es el programa más parecido a una herramienta real que has hecho hasta ahora.

Paso 1: estructura base con validación de argumentos

```
#!/bin/bash
# ordena.sh - organiza archivos por extensión
# Uso: ./ordena.sh <carpeta> [--prueba]

carpeta="$1"
modo="$2"

if [ $# -lt 1 ]; then
    echo "Uso: $0 <carpeta> [--prueba]" >&2
    exit 1
fi

if [ ! -d "$carpeta" ]; then
    echo "ERROR: '$carpeta' no es un directorio" >&2
    exit 2
fi

if [ "$modo" = "--prueba" ]; then
    echo ">>> MODO PRUEBA: no se moverá nada <<<"
fi
```

Valida antes de actuar: argumentos presentes, carpeta existente. El modo `--prueba` es oro: te deja ensayar la lógica sin consecuencias.

Paso 2: el corazón con for + case

```

movidos=0
for ruta in "$carpeta"/*; do
    [ -f "$ruta" ] || continue          # salta lo que no sea archivo
    archivo=$(basename "$ruta")

    case "$archivo" in
        *.jpg|*.jpeg|*.png|*.gif)
            destino="$carpeta/imagenes" ;;
        *.mp4|*.mkv|*.avi)
            destino="$carpeta/videos" ;;
        *.pdf|*.doc|*.docx|*.txt)
            destino="$carpeta/documentos" ;;
        *.zip|*.tar.gz|*.rar)
            destino="$carpeta/comprimidos" ;;
        *)
            destino="$carpeta/otros" ;;
    esac

    if [ "$modo" = "--prueba" ]; then
        echo "[prueba] $archivo -> $(basename "$destino")/"
    else
        mkdir -p "$destino"
        mv "$ruta" "$destino/"
        movidos=$((movidos + 1))
    fi
done

echo "Archivos movidos: $movidos"

```

Fíjate en el `case` con patrones múltiples separados por `|`, el `\*)` que atrapa lo demás, y `continue` saltando directorios. El contador `=\$((movidos+1))` es aritmética de la unidad.

Paso 3: banco de pruebas y modo prueba

```

mkdir -p ~/practica-linux/desorden
cd ~/practica-linux/desorden
touch foto1.jpg video1.mp4 tesis.pdf datos.zip notas.txt script.sh "mi foto.png"
chmod u+x ~/practica-linux/ordena.sh
~/practica-linux/ordena.sh ~/practica-linux/desorden --prueba

```

Verificación: debe listar a dónde iría cada archivo SIN mover nada (`ls` debe mostrar todo en su lugar). Observa que `"mi foto.png"` se clasifica como una sola pieza: el `for` con glob entrecomillado no la parte.

Paso 4: ejecución real y verificación

```
~/practica-linux/ordena.sh ~/practica-linux/desorden
echo "Código: $?"
ls -R ~/practica-linux/desorden
```

Verificación esperada: carpetas `imagenes/` (con foto1.jpg y "mi foto.png"), `videos/`, `documentos/`, `comprimidos/` y `otros/` (con script.sh). Código 0 y "Archivos movidos: 7".

Paso 5: prueba los errores

```
~/practica-linux/ordena.sh; echo "sin args: $?" # 1
~/practica-linux/ordena.sh /no/existe; echo "mala ruta: $?" # 2
~/practica-linux/ordena.sh ~/practica-linux/desorden/imagenes --prueba # carpeta
vacía de archivos sueltos
```

El último caso prueba el glob sin coincidencias: ``\$carpeta/*` con nada adentro se expande al patrón literal. ¿Qué imprime tu script? (Pista: es el próximo bug que vas a cazar; la Unidad 8 te dará `shopt -s nullglob`.)

Paso 6: agrega un while interactivo de confirmación

Mejora el script: antes de mover, si NO está en modo prueba, pregunta una vez con `read`:

```
if [ "$modo" != "--prueba" ]; then
    read -p "¿Mover archivos en '$carpeta'? [s/N] " resp
    case "$resp" in
        s|S|si|SI) echo "Adelante..." ;;
        *) echo "Cancelado."; exit 0 ;;
    esac
fi
```

Otro `case` en acción, ahora para entrada del usuario. Patrón clásico de instaladores y herramientas destructivas.

Reto final (sin solución)

1. Haz que el script acepte `--prueba` en cualquier posición (`./ordena.sh --prueba carpeta` también debe funcionar). Pista: recorre ``\$@" con un `for` antes de la lógica principal.

2. Agrega una categoría ``codigo/`` para ``.sh``, ``.py``, ``.js`` y haz que los ``.sh`` conserven su permiso de ejecución al moverlos (verifica con ``ls -l``).
3. Al final, si ``movidos`` es 0, el script debe salir con código 3 e imprimir "Nada que organizar" en `stderr`.

Si lo logras, ya piensas como programador de shell: validar, clasificar, actuar y reportar.

Práctica

En `~/practica-linux`. Cada script con shebang, permiso y pruebas con casos buenos y malos.

Ejercicio 1: semáforo de números

Enunciado: escribe ``semáforo.sh`` (mejor ``semaforo.sh`` sin acento en el nombre de archivo) que reciba un número y diga si es positivo, negativo o cero, usando `if/elif/else` con comparaciones numéricas. Valida que el argumento sea un número.

Solución explicada paso a paso. Paso 1: valida que haya argumento: ``[ -z "$1" ]`` muestra uso y sale con 1. Paso 2: valida que sea número con regex: ``[[ "$1" =~ ^-?[0-9]+$ ]] || { echo "No es un número" && exit 1; }``. El ``=~`` de ``[[ ]]`` compara contra una expresión regular: opcionalmente un menos, luego dígitos. Paso 3: ``if [ "$1" -gt 0 ]; then echo "positivo"; elif [ "$1" -lt 0 ]; then echo "negativo"; else echo "cero"; fi``. Paso 4: prueba con 5, -3, 0, "hola" y sin argumentos: los 5 casos. Paso 5: anota la diferencia: para números ``-gt/-lt/-eq``, para texto ``!=``; mezclarlos es un error clásico que ahora sabes evitar.

Ejercicio 2: contador de tipos

Enunciado: escribe ``cuenta-tipos.sh`` que reciba una carpeta y cuente cuántos archivos regulares, directorios y otros elementos hay dentro, usando un `for` con pruebas de tipo.

Solución explicada paso a paso. Paso 1: valida que el argumento exista y sea directorio: ``[ -d "$1" ] || { echo "No es un directorio" && exit 1; }``. Paso 2: inicializa contadores en 0: ``arch=0; dirs=0; otros=0``. Paso 3: el bucle: ``for e in "$1"/*; do [ -f "$e" ] && arch=$((arch+1)); [ -d "$e" ] && dirs=$((dirs+1)); ... done``. Cuidado: si la carpeta está vacía, ``${1}/*`` no expande y el patrón literal se procesa; agrégale ``[ -e "$e" ] || continue`` al inicio para saltar ese caso. Paso 4: imprime el reporte con los tres contadores. Paso 5: pruébalo en `~/practica-linux` y en una carpeta vacía. La lección: los contadores con ``$(( ))`` y el ``continue`` defensivo son el pan de cada día de los bucles reales.

Ejercicio 3: tabla de multiplicar

Enunciado: escribe `tabla.sh` que reciba un número y muestre su tabla de multiplicar del 1 al 10 con formato bonito, usando un `for` estilo C.

Solución explicada paso a paso. Paso 1: valida el argumento numérico como en el ejercicio 1. Paso 2: `n="$1"` y el bucle: `for ((i=1; i<=10; i++)); do echo "$n x $i = $((n * i))"; done`. Paso 3: formato bonito con `printf`: `printf "%2d x %2d = %3d\n" "$n" "$i" "$((n*i))"` alinea las columnas. Paso 4: prueba con 7 y verifica un par de resultados a mano. Paso 5: `printf` es el hermano elegante de `echo` para salidas tabulares: `%d` número, `%s` texto, `\n` salto de línea; guárdalo en tu cheat sheet.

Ejercicio 4: adivinanza con intentos

Enunciado: escribe `adivina.sh`: el script piensa un número del 1 al 20 (usa `$RANDOM`), el usuario tiene 5 intentos para adivinarlo, y después de cada intento el script dice "más alto", "más bajo" o "¡correcto!". Si se acaban los intentos, revela el número.

Solución explicada paso a paso. Paso 1: `secreto=$((RANDOM % 20 + 1))`: `RANDOM` da un número aleatorio grande; el módulo 20 lo limita a 0-19 y el +1 lo lleva a 1-20. Paso 2: `intentos=5` y un `while [ $intentos -gt 0 ]`: mientras queden intentos. Paso 3: dentro, `read -p "Adivina (quedan $intentos): " num`; valida que sea número o usa `continue` para no gastar el intento en basura. Paso 4: compara: igual -> "¡correcto!" y `break`; mayor -> "más bajo"; menor -> "más alto"; resta un intento. Paso 5: si sale del bucle sin adivinar, revela: `echo "Era el $secreto. ¡Suerte la próxima!"`. Paso 6: juega dos rondas. Este ejercicio une `RANDOM`, `while`, `if`, `read` y `break`: un juego completo en 30 líneas.

Ejercicio 5: limpiador inteligente

Enunciado: escribe `limpia-tmp.sh` que reciba una carpeta, liste todos los `.tmp` de más de 7 días de antigüedad, pida confirmación ("¿borrar? s/n") y solo los borre si el usuario confirma. Debe contar cuántos borró.

Solución explicada paso a paso. Paso 1: valida la carpeta con `[ -d "$1" ]`. Paso 2: encuentra candidatos: `mapfile -t viejos < <(find "$1" -name "*.tmp" -mtime +7)`: `mapfile` mete cada línea en un arreglo. (Si `mapfile` te parece avanzado, un `for` sobre el `find` también funciona.) Paso 3: si el arreglo está vacío (`[ ${#viejos[@]} -eq 0 ]`), informa "nada que limpiar" y sale con 0. Paso 4: muestra la lista con un `for`, luego `read -p "¿Borrar estos ${#viejos[@]} archivos? (s/n): " resp`. Paso 5: `case "$resp" in s|S|si) ... ;; *) echo "Cancelado."; exit 0;; esac`. Paso 6: en el sí, un `for` borrando con `rm -v` (muestra lo que borra) y un contador; al final informa "Se borraron

N archivos". Paso 7: prueba creando .tmp viejos con ``touch -d "10 days ago"`. Este es un script de mantenimiento real: confirma antes de destruir y reporta lo que hizo.

Tarjeta de referencia rápida

| Comando / sintaxis | Descripción | |---|---| | `[ cond ]` / `test cond` | Evalúa la condición; 0 si verdadera. Exige espacios: `[ -f x ]`. | | [[cond]] | Versión mejorada: patrones, `&&` dentro, sin partir variables. | | `-e -f -d -r -w -x -s` | Existe, es archivo regular, es directorio, legible, escribible, ejecutable, no vacío. | | `-z "$v"` / `-n "$v" | Cadena vacía / cadena no vacía. | | `-eq -ne -gt -lt -ge -le` | Comparadores NUMÉRICOS: igual, distinto, mayor, menor, mayor/menor o igual. | | `= !=` | Comparadores de TEXTO dentro de `[]` / [[ ]]. | | `if [c]; then ... fi` | Ejecuta el bloque si la condición es verdadera. | | `elif [c2]; then` | "Si no, pero si...": encadena condiciones. | | `else` | Lo que pasa si nada anterior fue verdad. | | `cmd1 && cmd2` | Ejecuta cmd2 solo si cmd1 tuvo éxito. | | `cmd1 || cmd2` | Ejecuta cmd2 solo si cmd1 falló. | | `case $v in pat) ... ;; esac` | Ramifica por patrones; `*` atrapa lo demás; `|` separa alternativas. | | `for x in lista; do ... done` | Repite con x tomando cada valor de la lista. | | `{1..5}` / `{a..z}` | Genera secuencias para el for. | | `for ((i=1; i<=n; i++))` | For estilo C con contador numérico. | | `while [c]; do ... done` | Repite MIENTRAS la condición sea verdad. | | `until [c]; do ... done` | Repite HASTA QUE la condición sea verdad (while invertido). | | `break` / `continue` | Sale del bucle / salta a la siguiente vuelta (`break 2` sale dos niveles). | | `$((expr))` | Aritmética entera: `$((a + b * 2))`. | | `((i++))` | Incrementa; también sirve como condición numérica en if/while. | | `while IFS= read -r l; do ... done < f` | Lee un archivo línea por línea sin subshell. | | `sleep N` | Pausa N segundos (acepta decimales: `sleep 0.5`). |`

5 one-liners que vas a usar toda la vida

```
[ -f "$f" ] && echo "existe" || echo "no existe"
```

El if de una línea: verifica y reporta sin escribir cinco líneas.

```
for f in *.jpg; do echo "JPG: $f"; done
```

Procesa todos los JPG de la carpeta; el glob respeta nombres con espacios.

```
n=0; while [ $n -lt 3 ]; do echo "vuelta $n"; n=$((n+1)); done
```

Bucle contado clásico con while y aritmética.

```
case "$1" in -h|--help) echo "ayuda";; *) echo "otro: $1";; esac
```

Despacha el primer argumento: base de cualquier script con opciones.

```
while IFS= read -r linea; do echo ">> $linea"; done < datos.txt
```

Lee el archivo línea por línea en el shell actual (las variables sobreviven).

Más comandos que completan la unidad

| Comando / sintaxis | Descripción | |---|---| | `archivo -nt otro` | Verdadero si `archivo` es MÁS NUEVO que `otro` (comparar fechas sin `stat`). | | `a -ef b` | Verdadero si son el MISMO archivo (detecta enlaces duros/suaves al mismo inodo). | | `shopt -s nullglob` | Un glob sin coincidencias se expande a NADA en vez de al patrón literal. | | `select o in a b c; do ... done` | Menú numerado interactivo automático; la elección cae en `\$o`, el número en `\$REPLY`. | | `let "x += 1"` | Aritmética alternativa (hoy se prefiere `\$(())` por legibilidad). | | `(( x > 5 ))` | `(( ))` como CONDICIÓN: 0 (verdadero) si la expresión es distinta de cero. | | `:` 'comentario' | `:` no hace nada: truco para comentarios multilínea o marcar ramas vacías del case. | | `until cmd; do ... done` | Repite HASTA que el comando tenga éxito: `until ping -c1 srv; do sleep 5; done`. | | `while ;; do ... done` | `:` siempre es verdadero: bucle infinito explícito (con `break` adentro). | | `for ((;;)); do ... done` | Bucle infinito estilo C; úsalo con fusible (contador máximo) por seguridad. | | `continue 2` / `break 2` | Afectan DOS niveles de bucles anidados (el número indica cuántos). | | `IFS=, read -r a b c` | Lee una línea CSV repartiéndola en variables con `IFS` temporal. |

3 combinaciones que valen oro

```
shopt -s nullglob; for f in *.bak; do rm "$f"; done
```

Con `nullglob`, si no hay `\*.bak` el for simplemente no itera: sin él, intentaría borrar un archivo llamado literalmente `\*.bak`.

```
intentos=0; until [ -f /tmp/listo ] || [ "$intentos" -ge 20 ]; do sleep 3; intentos=$((intentos+1)); done
```

Espera a que aparezca un archivo (señal de otro proceso) con límite de 20 vueltas: `until` + fusible, el patrón de sincronización entre scripts.

```
while IFS=, read -r nombre edad ciudad; do echo "$nombre ($edad) - $ciudad"; done <
datos.csv
```

Procesa un CSV línea por línea repartiendo columnas en variables: tubería de datos con herramientas de la Unidad 5, sin subshell que pierda variables.

Resumen de la unidad

- `test/[ ]/[ ]` evalúan condiciones: devuelve 0 si es verdadera; los corchetes exigen espacios internos.
- Pruebas de archivo: `-e` existe, `-f` regular, `-d` directorio, `-r/-w/-x` permisos, `-s` no vacío, `-z` cadena vacía.
- Números: `-eq -ne -gt -lt -ge -le`; texto: `= !=`; no los mezcles.
- `if/elif/else/fi` para decisiones; `&&` y `&&` combinan condiciones; `[ c ] && sí || no` es el `if` de una línea.
- `case/in/esac` para menús y múltiples opciones; `*`` atrapa lo demás; acepta comodines en patrones.
- `for` recorre listas: `for x in lista`, {1..5}`, for ((i=1;i<=n;i++))` : el rey del procesamiento por lotes.`
- `while` repite mientras sea verdad; `until` hasta que sea verdad; cuidado con los bucles infinitos.
- ``$(( ))`` es la aritmética del shell; ``RANDOM`` da aleatorios; ``sleep N`` pausa N segundos.
- `break` sale del bucle, `continue` salta a la siguiente vuelta: el control fino.
- Valida antes de actuar: argumentos presentes, rutas existentes, formatos correctos; el script profesional no supone.
- ``[[ ]]`` supera a ``[ ]``: acepta ``==`` con patrones (``*.txt``), ``&&`/`||`` adentro y no parte variables sin comillas.
- ``-a`` y ``-o`` dentro de ``[ ]`` están obsoletos y son ambiguos: combina con ``[ c1 ] && [ c2 ]`` en su lugar.
- ``(( ))`` evalúa aritmética y comparaciones numéricas: ``if ((edad >= 18))`` es más legible que ``-ge``.
- ``while read`` con tubería corre en subshell y pierde variables: usa ``done < archivo`` o ``done < <(comando)``.

- ``IFS=`` evita que `read` recorte espacios; ``-r`` evita que la diagonal invertida escape: ``while IFS= read -r l``.
- ``select nombre in op1 op2; do ... done`` crea menús numerados interactivos automáticamente.
- ``break 2`` y ``continue 2`` controlan bucles anidados indicando cuántos niveles afectar.
- ``for ((;;))`` es un bucle infinito intencional: úsalo con ``break`` y un fusible de seguridad.
- ``;;&`` en `case` continúa probando el siguiente patrón en vez de salir: para acciones acumulativas.

UNIDAD 8: Funciones, depuración y buenas prácticas

Tus scripts ya deciden y repiten. Ahora van a madurar: vas a organizar el código en funciones reutilizables, a blindarlos con el modo estricto para que fallen rápido y ruidoso en lugar de fallar en silencio, a aceptar opciones elegantes con getopts y a depurar como un profesional. Esta es la unidad que convierte scripts caseros en herramientas confiables.

En esta unidad vas a aprender a definir y usar funciones con parámetros y valores de retorno, variables locales, el modo estricto (set -euo pipefail), el procesamiento de opciones con getopts, técnicas de depuración (bash -x, set -x, trampas), y las buenas prácticas de estilo que usan los profesionales: comentarios, nombres claros y estructura.

Por qué importa esta unidad: un script de 20 líneas se entiende de un vistazo; uno de 200 sin funciones es una pesadilla. Las funciones dividen el problema en piezas con nombre: cada pieza hace una cosa y se puede probar por separado. Y el modo estricto es el cinturón de seguridad: convierte errores silenciosos (una variable mal escrita que vale vacío) en fallos inmediatos y visibles. Los bugs silenciosos son los caros; los ruidosos se arreglan en minutos.

Hay otra razón: getopts es lo que hace que tus scripts se sientan como comandos reales de Linux, con `-h` para ayuda y `-v` para detalles. Un script con buena interfaz lo usa cualquiera; uno que solo entiende argumentos posicionales misteriosos solo lo entiende su autor. Si quieres que tus herramientas vivan más allá de tu memoria, dales buena interfaz.

Cómo encaja en el curso: las Unidades 6 y 7 te dieron scripts con lógica; esta los profesionaliza. En la Unidad 9 los programarás con cron, donde no hay nadie para ver errores: por eso el modo estricto y los registros son críticos. El proyecto final aplicará todo: funciones, modo estricto, opciones y depuración.

Empecemos. Al terminar, tus scripts van a estar organizados, blindados y depurables.

Explicación detallada

Funciones: piezas con nombre

Una función agrupa comandos bajo un nombre para reutilizarlos:

```
saludar() {  
    echo "Hola, $1"  
}
```

```
saludar "Ana"  
saludar "Luis"
```

La defines una vez y la llamas cuantas veces quieras. Los parámetros llegan como `\$1`, `\$2`... igual que en el script, pero locales a la función. Para devolver un resultado hay dos vías: el código de salida con `return` (solo números 0-255) o imprimir el valor y capturarlo con `\$( )`:

```
doble() {  
    echo $(( $1 * 2 ))  
}  
  
resultado=$(doble 21)  
echo "El doble es $resultado"
```

Este segundo patrón (imprimir y capturar) es el más usado para devolver texto o números grandes. `return` se reserva para éxito/fallo: `return 0` todo bien, `return 1` algo falló.

Variables locales: no ensucies la casa ajena

Por defecto, las variables en bash son globales: una variable creada dentro de una función existe también fuera de ella. Eso causa choques misteriosos cuando dos funciones usan el mismo nombre. La solución es `local`:

```
contador_global=0  
  
incrementar() {  
    local contador=0  
    contador=$((contador + 1))  
    echo "Dentro: $contador"  
}  
  
incrementar  
echo "Fuera: $contador_global"
```

``local`` crea la variable solo dentro de la función: al salir, desaparece. Regla profesional: toda variable que solo se use dentro de una función debe ser ``local``. Las globales se reservan para configuración que todo el script necesita. Esta disciplina evita la categoría entera de bugs "¿por qué cambió mi variable?".

El modo estricto: falla rápido, falla ruidoso

Tres líneas al inicio de tus scripts cambian las reglas del juego:

```
#!/bin/bash
set -euo pipefail
```

``set -e``: si cualquier comando falla (código distinto de 0), el script se detiene de inmediato. Sin esto, bash sigue ejecutando como si nada después de un error, y el desastre se encadena. ``set -u``: usar una variable no definida es un error fatal. Atrapa los typos: ``$nomrbe`` en lugar de ``$nombre`` ya no vale vacío en silencio, detiene el script y te avisa. ``set -o pipefail``: en una tubería, si cualquier comando de la cadena falla (no solo el último), la tubería se considera fallida. Sin esto, ``comando-que-falla | grep algo`` reporta éxito porque grep sí funcionó.

Con el modo estricto, los scripts fallan en el punto exacto del problema con un mensaje claro, en lugar de seguir cojeando y fallar 50 líneas después de forma misteriosa. Es incómodo al principio (tu script se detiene más), pero cada detención es un bug que encuentras gratis.

getopts: opciones como los grandes

Los comandos reales usan opciones con guion: ``ls -lh``, ``grep -ri``. `getopts` te da eso para tus scripts:

```
#!/bin/bash
verbose=0
archivo=""

while getopts "vha:" opt; do
    case "$opt" in
        v) verbose=1 ;;
        h) echo "Uso: $0 [-v] [-a archivo]"; exit 0 ;;
        a) archivo="$OPTARG" ;;
        *) echo "Opción inválida" && exit 1 ;;
    esac
done
```

La cadena `"vha:"` declara las opciones: ``v`` y ``h`` son banderas (sin argumento), ``a:`` con dos puntos significa que REQUIERE un argumento, que llega en ``$OPTARG``. Uso: ``./script.sh -v -a datos.txt``. Después del bucle, ``shift $((OPTIND-1))`` descarta las opciones procesadas y deja los argumentos restantes en ``$1``, ``$2``...

Con `getopts` tu script acepta ``-v -a archivo`` o ``-va archivo`` (opciones combinadas), muestra ayuda con ``-h`` y rechaza opciones inválidas. Interfaz profesional en 15 líneas.

Depuración: ver lo que el script ve

Cuando un script se porta raro, necesitas ver lo que realmente está pasando. La herramienta principal es el modo de rastreo:

```
bash -x miscript.sh
```

Cada comando se imprime con un ``+`` antes de ejecutarse, con las variables ya expandidas. Ves exactamente qué está haciendo el script, no lo que crees que hace. Para rastrear solo una sección, actívalo y desactívalo dentro del script:

```
set -x
# ... sección sospechosa ...
set +x
```

Otras técnicas: ``echo "DEBUG: variable=$variable"`` en puntos clave (quitalos al final o protégelos con una bandera verbose), y ``trap`` para atrapar errores:

```
trap 'echo "Falló en la línea $LINENO"' ERR
```

Con ``set -e`` activo, este trap te dice exactamente en qué línea murió el script. ``$LINENO`` es el número de línea actual: el GPS del depurador.

Buenas prácticas de estilo

Los scripts se leen más de lo que se escriben: escríbelos para el lector, que muchas veces serás tú dentro de seis meses.

Primero, encabezado con documentación: qué hace el script, cómo se usa, quién lo hizo y cuándo. Segundo, nombres descriptivos: ``respaldar_fotos`` dice más que ``rf``. En bash, la convención es minúsculas_con_guiones_bajos para funciones y variables. Tercero, comenta el PORQUÉ, no el QUÉ: ``i=$((i+1)) # avanza al siguiente archivo`` no aporta; ``# reintenta porque el servidor tarda en despertar`` sí.

Cuarto, valida entradas al inicio y falla temprano: si faltan argumentos o una ruta no existe, dilo y sal con código de error antes de hacer nada. Quinto, usa rutas absolutas o calcula la ruta del script (``DIR="$(cd "$(dirname "$0")" && pwd)"``) para que funcione desde donde sea. Sexto, entrecomilla variables, usa ``[[ ]]``, prefiere ``$( )`` sobre backticks: todo lo que ya viste en este curso, aplicado con disciplina.

Ejemplos paso a paso

Ejemplo 1: biblioteca de utilidades

Vas a crear un archivo de funciones reutilizables y usarlo desde otro script.

```
#!/bin/bash
# lib.sh - Biblioteca de funciones comunes

log() {
    echo "[$(date '+%Y-%m-%d %H:%M:%S')] $1"
}

error() {
    echo "[ERROR] $1" >&2
}

requiere_archivo() {
    local ruta="$1"
    if [ ! -f "$ruta" ]; then
        error "No existe el archivo: $ruta"
        return 1
    fi
    return 0
}

mayor_de() {
    local a="$1" b="$2"
    if [ "$a" -gt "$b" ]; then
        echo "$a"
    else
        echo "$b"
    fi
}

```

Guárdalo como `~/practica-linux/lib.sh` (sin permiso de ejecución: es biblioteca, no programa).
Ahora úsalo:

```
#!/bin/bash
# usa-lib.sh - Ejemplo de uso de la biblioteca
set -euo pipefail

source "$HOME/practica-linux/lib.sh"

log "Iniciando proceso..."
if requiere_archivo "$HOME/practica-linux/datos.txt"; then
    log "Archivo encontrado, continuando."
fi
m=$(mayor_de 42 17)
log "El mayor es $m"
log "Terminado."
```

`source` (o su alias `.`) carga las funciones en tu script como si estuvieran escritas ahí. Crea `datos.txt` para la prueba, dale permiso a `usa-lib.sh` y ejecútalo.

Explicación: `source` es el "import" de bash. Las bibliotecas evitan copiar y pegar funciones entre scripts: una sola versión de `log` usada por todos tus scripts. Cuando mejores `log`, todos mejoran.

Ejemplo 2: script blindado con modo estricto

Vas a ver la diferencia entre un script descuidado y uno con modo estricto, con el mismo bug.

```
#!/bin/bash
# fragil.sh - SIN modo estricto (mal ejemplo)

nombre_archivo="importante.txt"
# ... 30 líneas después, un typo:
rm "$nombre_archvo"
echo "Limpieza terminada."
```

Con el typo (`archvo` en lugar de `archivo`), `\$nombre\_archvo` vale vacío y el comando se convierte en `rm ""`, que falla... pero el script sigue y dice "Limpieza terminada." ¡Mintiendo! El usuario cree que todo salió bien.

```
#!/bin/bash
# blindado.sh - CON modo estricto (bien)
set -euo pipefail

nombre_archivo="importante.txt"
rm "$nombre_archivo"
echo "Limpieza terminada."
```

Con `set -u`, la variable indefinida detiene el script de inmediato con un mensaje claro: `nombre\_archivo: unbound variable`. No miente, no sigue: falla rápido y te dice exactamente dónde.

Explicación: el modo estricto convierte bugs silenciosos en errores ruidosos. En un script que corre solo a las 3 AM con cron, la diferencia es entre "falló y me avisó" y "falló y nadie se enteró hasta que fue tarde".

Ejemplo 3: herramienta con getopt

Una herramienta de búsqueda de texto con interfaz profesional.

```
#!/bin/bash
# buscar.sh - Busca texto en archivos con opciones
set -euo pipefail

ignorar=0
solo_nombres=0

uso() {
    echo "Uso: $0 [-i] [-l] <patrón> [ruta]"
    echo "  -i Ignorar mayúsculas/minúsculas"
    echo "  -l Mostrar solo nombres de archivo"
    echo "  -h Esta ayuda"
}

while getopts "ilh" opt; do
    case "$opt" in
        i) ignorar=1 ;;
        l) solo_nombres=1 ;;
        h) uso; exit 0 ;;
        *) uso >&2; exit 1 ;;
    esac
done
shift $((OPTIND - 1))

if [ $# -lt 1 ]; then
    uso >&2
    exit 1
fi

patron="$1"
ruta="${2:-.}"

opciones=""
[ "$ignorar" -eq 1 ] && opciones="{opciones}i"
[ "$solo_nombres" -eq 1 ] && opciones="{opciones}l"

grep -r${opciones} -- "$patron" "$ruta"
```

Pruébalo: `./buscar.sh -i "error" ~/practica-linux`, ./buscar.sh -l "TODO" .`, ./buscar.sh -h`.
 Observa el --` antes del patrón: le dice a grep "lo que sigue no son opciones", protegiendo contra patrones que empiecen con guion.`

Explicación: este script se siente como un comando real: tiene ayuda, opciones combinables y valida sus argumentos. `getopts` + función ``uso`` + validación es la receta de la interfaz profesional.

Ejemplo 4: cacería de un bug con `-x`

Vas a depurar un script con un bug escondido usando el rastreo.

```
#!/bin/bash
# promedio.sh - Calcula el promedio (con un bug escondido)
suma=0
n=0
for num in "$@"; do
    suma=$((suma + num))
    n=$((n + 1))
done
promedio=$((suma / n))
echo "Promedio: $promedio"
```

Pruébalo: `./promedio.sh 10 20 30` -> dice 20. Correcto. Pero ./promedio.sh` sin argumentos -> "división por cero" y un error feo. Activa el rastreo para verlo:`

```
bash -x ~/practica-linux/promedio.sh
```

En el rastreo ves ``+ n=0`` y luego el intento de ``suma / n`` con `n` valiendo 0: el bug a la vista. La corrección:

```
if [ $# -eq 0 ]; then
    echo "Uso: $0 <números...>" >&2
    exit 1
fi
```

al inicio. Vuelve a probar con y sin argumentos: ahora falla con un mensaje útil.

Explicación: ``bash -x`` te muestra la película de lo que el script realmente hace, con variables expandidas. El 90 por ciento de los bugs se encuentran así: ves el valor real donde esperabas otro. Depurar no es adivinar: es observar.

Errores comunes y casos reales

Aquí el nivel sube: los bugs ya no son de sintaxis, son de diseño. Un script con funciones y modo estricto mal usado falla de formas elegantes y difíciles de ver. Estos son los clásicos.

Caso 1: ``return "texto"`` para devolver datos de una función

Síntoma: tu función hace ``return "$resultado"`` y al capturarlo obtienes un número raro (o vacío), nunca el texto.

Por qué ocurre: ``return`` solo devuelve CÓDIGOS de salida (0-255), igual que ``exit`` pero para funciones. No es un "regresa este valor" de otros lenguajes: es un "terminé así". Si devuelves texto, bash lo convierte a número o falla.

Solución: los datos salen por stdout y se capturan con ``$( )``; ``return`` queda para éxito/fracaso:

```
cuadrado() {
    local n="$1"
    if ! [[ "$n" =~ ^[0-9]+$ ]]; then
        return 1          # código: falló (no era número)
    fi
    echo $((n * n))        # dato: sale por stdout
    return 0
}

r=$(cuadrado 7)           # r = "49"
if cuadrado "hola"; then
    echo "ok"
else
    echo "falló con código $?"
fi
```

Regla de oro: ``return`` habla de CÓMO terminó; ``echo`` entrega QUÉ produjo.

Caso 2: variables sin ``local`` que se pisan entre funciones

Síntoma: dos funciones usan una variable llamada ``i`` o ``tmp``; de pronto una función altera el valor que la otra necesitaba. El bug aparece y desaparece según el orden de las llamadas: una pesadilla de depurar.

Por qué ocurre: en bash TODAS las variables son globales por defecto, incluso las creadas dentro de funciones. Sin `local`, tu `contador` de la función A es el mismo `contador` de la función B y del script principal.

Solución: declara `local` en la primera línea de cada función para todo lo que sea temporal:

```
suma_hasta() {  
    local limite="$1"  
    local total=0  
    local i  
    for ((i=1; i<=limite; i++)); do  
        total=$((total + i))  
    done  
    echo "$total"  
}
```

Acostúmbrate: si la variable no necesita salir de la función, lleva `local`. Punto.

Caso 3: `set -e` que mata tu script en el `if` o en un `grep` sin coincidencias

Síntoma: activas `set -e` (modo estricto) y tu script muere a la mitad "sin razón": justo en un `grep` que no encontró nada o en una verificación que dio falso.

Por qué ocurre: `set -e` aborta ante CUALQUIER comando con código distinto de 0, y `grep` devuelve 1 cuando no hay coincidencias (eso es informativo, no un error). Además hay excepciones sutiles: los comandos dentro de `if`, `while` o `||` no disparan el aborto, lo que vuelve el comportamiento impredecible si no lo entiendes.

Solución: protege las verificaciones que PUEDEN dar falso legítimamente con `|| true`, o escríbelas dentro de `if`:

```
set -euo pipefail  
# grep puede no encontrar nada y eso está bien:  
coincidencias=$(grep -c "ERROR" app.log || true)  
  
# Mejor aún: la intención queda explícita con if  
if grep -q "ERROR" app.log; then  
    echo "hay errores en el log"  
fi
```

`set -e` es excelente, pero exige que cada comando que "puede fallar sin ser error" esté marcado. Si no, el modo estricto se vuelve modo paranoico.

Caso 4: getopt sin `:` y sin `shift \$((OPTIND-1))`

Síntoma: `./tool.sh -n Ana archivo.txt` ignora el valor "Ana", o el script trata `archivo.txt` como si fuera una opción más y se queja de "opción inválida".

Por qué ocurre: en la cadena de `getopts ("vha:")`, los dos puntos DESPUÉS de una letra significan "esta opción recibe un valor". Sin los dos puntos, `-n Ana` deja "Ana" colgando. Y sin `shift $((OPTIND-1))` al terminar el bucle, los argumentos posicionales (como `archivo.txt`) siguen mezclados con las opciones ya procesadas.

Solución: el patrón completo, siempre igual:

```
#!/bin/bash
verbose=0; nombre=""
while getopts "vhn:" opt; do
    case "$opt" in
        v) verbose=1 ;;
        h) echo "Uso: $0 [-v] [-n nombre] archivo"; exit 0 ;;
        n) nombre="$OPTARG" ;;
        *) echo "Opción inválida" >&2; exit 1 ;;
    esac
done
shift $((OPTIND - 1)) # quita las opciones procesadas; lo que queda son archivos
echo "verbose=$verbose nombre=$nombre archivos: $@"
```

Memoria: la letra con `:` come valor (`$OPTARG`); al final, `shift` limpia la mesa.

Caso 5: `\$1` dentro de una función no es el `\$1` del script

Síntoma: tu función recibe parámetros pero "no llegan": adentro `$1` está vacío o tiene otro valor, aunque al script sí le pasaste argumentos.

Por qué ocurre: cada función tiene SUS PROPIOS parámetros posicionales. Cuando llamas `mi_func "hola"`, adentro `$1` es "hola", y los argumentos del script quedan ocultos mientras la función ejecuta. Si llamas la función sin argumentos, su `$1` está vacío aunque el script tenga diez.

Solución: pasa explícitamente lo que la función necesite, o guarda los args del script en variables con nombre antes:

```
#!/bin/bash
archivo="$1" # captúralo con nombre ANTES

procesar() {
    local f="$1" # este $1 es el de la FUNCIÓN
    echo "Procesando $f"
}

procesar "$archivo" # pásalo explícito
```

Nunca supongas que la función "ve" los argumentos del script: llévalos de la mano.

Caso 6: depurar a ciegas en vez de usar `bash -x`

Síntoma: agregas `echo "llegué aquí 1"`, `echo "llegué aquí 2"` por todo el script, corres, borras, agregas más... media hora para un bug de una línea.

Por qué ocurre: estás adivinando en vez de observar. Bash ya trae la radiografía integrada: `bash -x` muestra cada comando DESPUÉS de expandir variables, justo antes de ejecutarlo.

Solución:

```
bash -x ./mi_script.sh arg1 arg2
```

Verás líneas como `+ archivo='mis notas.txt` y `+ rm -rf mis notas.txt` (¡sin comillas!) y el bug salta a la vista. Para rastrear solo una sección sin ensuciar todo el log:

```
set -x # empieza el rastreo aquí
# ... sección sospechosa ...
set +x # termina el rastreo aquí
```

El 90 por ciento de los bugs se rinden ante `bash -x`. Los `echo` de depuración son del siglo pasado.

Caso real 1: el respaldo que se detenía a la mitad cada martes

Una empresa tenía `respaldo.sh` con `set -e`. Funcionaba meses... hasta que un martes falló a la mitad dejando un `.tar.gz` corrupto. La causa: el script verificaba espacio libre con `df | grep "/datos" | awk ...` y ese martes el punto de montaje `/datos` no estaba montado (mantenimiento). `grep` devolvió 1 (sin coincidencias), `set -e` abortó el script EN MEDIO, después de haber creado el tar parcial pero antes de verificarlo y rotarlo.

La lección: ``set -e`` no sustituye el manejo de errores pensado. Las verificaciones críticas van en ``if`` explícitos con mensajes y salidas controladas:

```
#!/bin/bash
set -euo pipefail

if ! df | grep -q "/datos"; then
    echo "ERROR: /datos no está montado, aborto el respaldo" >&2
    exit 1
fi
# ... resto del respaldo, ahora sí protegido por set -e ...
```

Con ``set -e``, cada comando "que puede fallar por razones normales" debe estar en un ``if`` o llevar ``||``. El modo estricto es un cinturón de seguridad, no un piloto automático.

Caso real 2: la función ``limpiar`` que limpió de más

Un script de despliegue tenía una función ``limpiar()`` que usaba una variable ``dir``... y el script principal también usaba ``dir`` para el directorio de la aplicación. Un refactor movió la llamada a ``limpiar`` a otro punto, donde ``dir`` ya apuntaba a la app en producción en vez de al temporal. ``rm -rf "$dir"/cache`` borró el caché de producción en hora pico.

Sin ``local``, la función leyó la variable global que "casualmente" tenía otro valor en ese momento. La versión profesional aísla todo:

```
limpiar() {
    local dir="$1"    # la función SOLO usa lo que recibe
    [ -n "$dir" ] || { echo "limpiar: falta directorio" >&2; return 1; }
    echo "Limpiando $dir/cache"
    # rm -rf "$dir"/cache
}
```

Moraleja: las funciones deben ser cajas cerradas: entran parámetros, salen resultados, nada de variables globales "prestadas".

Miniproyecto guiado de la unidad

Vas a construir ``notas.sh``: un gestor de notas por terminal con opciones de verdad (``-a`` agregar, ``-l`` listar, ``-b`` buscar, ``-v`` verboso, ``-h`` ayuda), funciones con ``local``, modo estricto ``set -euo pipefail`` y mensajes de depuración. Es tu primer programa con interfaz de línea de comandos profesional.

Paso 1: esqueleto con modo estricto y funciones base

```
#!/bin/bash
# notas.sh - gestor simple de notas en texto plano
# Uso: notas.sh [-v] [-a texto] [-l] [-b patron] [-h]
set -euo pipefail

ARCHIVO="$HOME/practica-linux/notas.txt"
VERBOSE=0

log() { # mensajes de depuración, solo con -v
    if [ "$VERBOSE" -eq 1 ]; then
        echo "[debug] $" >&2
    fi
}

die() { # error fatal: mensaje a stderr y salida con código
    echo "ERROR: $" >&2
    exit 1
}

uso() {
    cat <<'AYUDA'
    Uso: notas.sh [-v] [-a texto] [-l] [-b patron] [-h]
    -a texto    Agrega una nota con fecha
    -l          Lista todas las notas numeradas
    -b patron   Busca notas que contengan el patrón
    -v          Modo verboso (muestra depuración)
    -h          Esta ayuda
    AYUDA
}
```

Observa: `die` centraliza los errores fatales, `log` escribe a stderr para no contaminar la salida útil, y el heredoc de `uso` usa `<<'AYUDA'` entrecomillado para que nada se expanda adentro.

Paso 2: funciones que hacen el trabajo

```

agregar() {
    local texto="$1"
    [ -n "$texto" ] || die "el texto de la nota no puede estar vacío"
    local fecha
    fecha=$(date '+%Y-%m-%d %H:%M')
    echo "[$fecha] $texto" >> "$ARCHIVO"
    log "nota agregada en $ARCHIVO"
    echo "Nota guardada."
}

listar() {
    if [ ! -f "$ARCHIVO" ]; then
        echo "No hay notas todavía."
        return 0
    fi
    local n=0
    local linea
    while IFS= read -r linea; do
        n=$((n + 1))
        echo "$n. $linea"
    done < "$ARCHIVO"
}

buscar() {
    local patron="$1"
    [ -n "$patron" ] || die "indica un patrón de búsqueda"
    [ -f "$ARCHIVO" ] || die "no hay notas todavía"
    if grep -i -- "$patron" "$ARCHIVO"; then
        log "búsqueda '$patron' con coincidencias"
    else
        echo "Sin coincidencias para '$patron'."
    fi
}

```

Todo con `local`, validación temprana con `die`, y `grep --` para que un patrón que empiece con `-` no se confunda con opción.

Paso 3: el despachador con getopt

```

accion=""
parametro=""

while getopts "vha:lb:" opt; do
    case "$opt" in
        v) VERBOSE=1 ;;
        h) uso; exit 0 ;;
        a) accion="agregar"; parametro="$OPTARG" ;;
        l) accion="listar" ;;
        b) accion="buscar"; parametro="$OPTARG" ;;
        *) uso >&2; exit 1 ;;
    esac
done
shift $((OPTIND - 1))

case "$accion" in
    agregar) agregar "$parametro" ;;
    listar) listar ;;
    buscar) buscar "$parametro" ;;
    *) uso >&2; exit 1 ;;
esac

```

Dos `case` encadenados: el primero parsea opciones, el segundo despacha la acción. Si no se dio acción válida, ayuda y salida 1.

Paso 4: prueba funcional completa

```

chmod u+x ~/practica-linux/notas.sh
cd ~/practica-linux
./notas.sh -h
./notas.sh -a "Aprendí funciones con local"
./notas.sh -a "set -euo pipefail me salvó de un bug"
./notas.sh -l
./notas.sh -b "funciones"
./notas.sh -v -b "nada-que-exista"

```

Verificación: `-l` numera las notas con fecha, `-b` filtra sin importar mayúsculas, y con `-v` ves las líneas `[debug]` en stderr sin ensuciar la salida.

Paso 5: prueba los errores y depura con `bash -x`

```
./notas.sh -a ""; echo "vacía: $?"          # debe morir con die -> código 1
./notas.sh -z; echo "opción mala: $?"       # getopt se queja -> código 1
./notas.sh; echo "sin acción: $?"          # ayuda + código 1
bash -x ./notas.sh -a "prueba x" 2>&1 | head -20
```

Con ``bash -x`` verás cada expansión: ``+ texto='prueba x'``, ``+ fecha='2026-09-20 ...'``. Si algo se comporta raro, la radiografía lo delata.

Paso 6: rompe algo a propósito (laboratorio de `set -e`)

Comenta temporalmente una validación y observa: cambia ``[ -n "$texto" ] || die ...`` por nada y corre `./notas.sh -a ""`. Verás una línea vacía con fecha en tus notas: el script "funcionó" pero guardó basura. Descomenta y confirma que ahora muere con mensaje útil. Esta es la diferencia entre un script que corre y un script confiable.

Reto final (sin solución)

1. Agrega la opción ``-e NUM`` que borre la nota número NUM (pista: ``sed -i "${NUM}d``) sobre el archivo, validando que NUM sea número positivo y que exista esa línea).
2. Agrega ``-c`` que muestre cuántas notas hay, usando una función ``contar`` que devuelva el número por stdout (no con ``return``).
3. Haz que ``-a``, ``-b`` y ``-e`` puedan combinarse en una sola invocación ejecutándose en orden (pista: en vez de una sola variable ``accion``, acumula acciones y recórrelas con un ``for`` al final).

Si lo terminas, tienes un programa CLI de verdad: parseo de opciones, funciones aisladas, errores honestos y modo estricto.

Práctica

En `~/practica-linux`. A partir de ahora, TODOS tus scripts llevan ``set -euo pipefail`` y shebang.

Ejercicio 1: calculadora con funciones

Enunciado: escribe ``calc.sh`` con funciones ``sumar``, ``restar``, ``multiplicar`` y ``dividir`` que reciban dos números y devuelvan el resultado. El script principal recibe operación y dos números como argumentos. La división entre cero debe fallar con mensaje útil.

Solución explicada paso a paso. Paso 1: shebang + ``set -euo pipefail``. Paso 2: define las cuatro funciones; cada una hace ``echo $(( $1 OP $2 ))``. En dividir, valida: ``if [ "$2" -eq 0 ]; then echo "Error: división entre cero" && return 1; fi``. Paso 3: el principal usa case con ``$1`` (suma|sumar, resta|restar, etc.) y llama a la función con ``$2`` y ``$3``, capturando con ``$( )``. Paso 4: valida que haya 3 argumentos al inicio. Paso 5: prueba las 4 operaciones y la división entre cero. Las funciones convierten tu script en una caja de herramientas: mañana puedes hacer ``source calc.sh`` y usar ``sumar`` en otro script.

Ejercicio 2: el modo estricto te salva

Enunciado: escribe a propósito un script ``typo.sh`` SIN modo estricto que use una variable mal escrita y observe cómo falla en silencio. Luego agrega ``set -euo pipefail``, ejecútalo de nuevo y observa la diferencia. Documenta lo aprendido en tu cheat sheet.

Solución explicada paso a paso. Paso 1: escribe el script con ``archivo="datos.txt"``, luego ``echo "Procesando $archivo"`` (typo) y ``wc -l "$archivo" > reporte.txt``. Sin modo estricto: imprime "Procesando " vacío y crea reporte.txt vacío, sin quejarse. Paso 2: agrega ``set -euo pipefail`` después del shebang y ejecútalo: se detiene en la línea del typo con "unbound variable". Paso 3: corrige el typo y verifica que funciona. Paso 4: en tu cheat sheet escribe: "set -euo pipefail = el script me avisa de mis typos en lugar de mentirme". Este ejercicio se siente tonto hasta que te salva de un bug real a las 2 AM.

Ejercicio 3: renombrador con opciones

Enunciado: escribe ``renombra.sh`` que renombre archivos agregando un prefijo, con opciones ``-p`` (prefijo, requerido), ``-v`` (verbose, muestra cada cambio) y ``-h`` (ayuda). Debe recibir la lista de archivos como argumentos restantes.

Solución explicada paso a paso. Paso 1: shebang + modo estricto + función ``uso``. Paso 2: ``while getopts "vp:h" opt`` con ``p`` prefijo="``$OPTARG``", ``v`` verbose=1, ``h`` uso; exit 0. Paso 3: ``shift $((OPTIND-1))``; valida que el prefijo no esté vacío y que haya al menos un archivo. Paso 4: ``for f in "$@"``: verifica ``[ -f "$f" ]``, construye ``nuevo="$(dirname "$f")/${prefijo}${basename "$f}"`` y ``mv "$f" "$nuevo"``; si verbose, informa cada movimiento. Paso 5: pruébalo en una carpeta de prueba: ``./renombra.sh -p "2024-" -v *.txt``. getopts convierte tu script en una herramienta que otros pueden usar sin leer el código.

Ejercicio 4: depura el contador

Enunciado: este script tiene un bug. Encuéntralo con ``bash -x``, corrígelo y explica qué pasaba:

```
#!/bin/bash
total=0
for f in $1/*.log; do
    lineas=$(wc -l < "$f")
    total=$((total + lineas))
done
echo "Total de líneas: $total"
```

Solución explicada paso a paso. Paso 1: guárdalo como ``cuenta-log.sh`` y ejecútalo con ``bash -x`` sobre una carpeta con `.log`: en el rastreo verás que ``$1`` sin comillas se parte si la ruta tiene espacios, y que si no hay `.log`, el patrón literal se procesa como archivo. Paso 2: los bugs: (a) ``$1`` sin comillas, (b) sin validación de que el argumento exista, (c) sin ``continue`` si el glob no expande. Paso 3: la versión corregida: ``carpeta="${1:?Falta la carpeta}"`` (el ``:?`` falla con mensaje si está vacío), ``[ -d "$carpeta" ]`` para validar, ``for f in "$carpeta"/*.log`` entrecomillado, y ``[ -e "$f" ] || continue`` dentro. Paso 4: prueba con ruta con espacios (crea "mi carpeta" con un `.log` dentro) y con carpeta vacía. El rastreo ``-x`` te mostró la película; ahora sabes leerla.

Ejercicio 5: tu biblioteca personal

Enunciado: crea ``~/practica-linux/mi-lib.sh`` con al menos 4 funciones útiles que hayas necesitado en el curso (por ejemplo: ``log``, ``error``, ``requiere_arg``, ``confirmar``). Luego escribe un script que la use con ``source`` y demuestre las 4 funciones.

Solución explicada paso a paso. Paso 1: define ``log`` (mensaje con fecha), ``error`` (a `stderr`), ``requiere_arg`` (verifica que `$1` no esté vacío o sale con error y uso), ``confirmar`` (pregunta s/n y devuelve 0/1 según respuesta). Paso 2: usa ``local`` en todas las variables internas. Paso 3: escribe ``demo-lib.sh`` con modo estricto que haga ``source`` de tu biblioteca y ejerza cada función: un log, un error provocado, una validación de argumento y una confirmación. Paso 4: ejecútalo y verifica cada comportamiento. Paso 5: a partir de hoy, todo script nuevo empieza con ``source ~/practica-linux/mi-lib.sh``: tu biblioteca crece contigo y tus scripts nacen profesionales.

Tarjeta de referencia rápida

| Comando / sintaxis | Descripción | `|---|---` | ``nombre() { ...; }`` | Define una función (también ``function nombre { ...; }``). | ``$1``, ``$2`` en función | Parámetros DE LA FUNCIÓN, no del script. |
 | ``local var="x"`` | Variable visible solo dentro de la función (úsala siempre). | ``return N`` | Termina la función con código N (0-255); los datos salen por ``echo``. | ``resultado=$(func args)`` | Captura lo que la función imprime en `stdout`. | ``set -e`` | Aborta si un comando falla (lo que

puede fallar legítimamente va en `if`). || `set -u` | Aborta si usas una variable sin definir (atrapa typos). || `set -o pipefail` | La tubería falla si CUALQUIER parte falla, no solo la última. || `set -euo pipefail` | El trío de modo estricto: ponlo tras el shebang. || `set -x` / `set +x` | Activa/desactiva el rastreo de comandos expandidos. || `bash -x script.sh` | Ejecuta mostrando cada comando expandido: la radiografía. || `bash -n script.sh` | Revisa la sintaxis SIN ejecutar nada. || `while getopts "vha:" opt` | Bucle que parsea opciones; `:` = la opción recibe valor. || `OPTARG` | Valor de la opción actual (solo si llevaba `:`). || `OPTIND` | Índice del próximo argumento sin procesar. || `shift \$((OPTIND-1))` | Descarta las opciones ya parseadas; deja los archivos. || `trap 'cmd' ERR` | Ejecuta cmd cuando un comando falla (con `\$LINENO` sabes dónde). || `trap 'cmd' EXIT` | Limpieza garantizada al terminar (borrar temporales). || `source lib.sh` / `.` lib.sh | Carga funciones de otro archivo (biblioteca). || `readonly VAR="x"` | Constante: nadie la puede reasignar después. || `die() { echo "\$\*" >&2; exit 1; }` | Patrón de error fatal reutilizable. |

5 one-liners que vas a usar toda la vida

```
set -euo pipefail
```

Tres palabras al inicio de cada script serio: fallos, variables fantasma y tuberías rotas, detectados.

```
bash -n mi_script.sh && echo "sintaxis ok"
```

Verifica que el script al menos parsea bien, sin ejecutarlo ni tocar nada.

```
bash -x mi_script.sh 2>&1 | grep -A2 "sospechoso"
```

Rastrea la ejecución y filtra la zona del bug: depuración quirúrgica.

```
die() { echo "ERROR: $*" >&2; exit 1; }; [ -f "$f" ] || die "no existe $f"
```

Define tu salida de emergencia en una línea y úsala para validar todo.

```
trap 'rm -f /tmp/mi_$$tmp' EXIT
```

Garantiza que tu archivo temporal se borre aunque el script muera a la mitad.

Más comandos que completan la unidad

| Comando / sintaxis | Descripción | |---|---| | ``$-`` | Letras de las opciones activas del shell (``himBH`...`): detecta modo interactivo. | | ``$BASH_VERSION`` | Versión de bash en ejecución (para scripts que necesitan funciones modernas). | | ``${FUNCNAME[0]}`` | Nombre de la función actual: mensajes de error que dicen DÓNDE fallaron. | | ``${BASH_SOURCE[0]}`` | Archivo donde está definida la función (útil en bibliotecas con ``source``). | | ``caller`` | Muestra quién llamó a la función actual: rastreo de llamadas manual. | | ``set -o`` | Lista todas las opciones del shell y su estado (`errexit`, `nounset`, `pipefail`...). | | ``set -o nounset`` | Lo mismo que ``set -u``; ``set -o pipefail`` = ``set -o pipefail`` explícito y legible. | | ``command ls`` | Ejecuta ``ls`` IGNORANDO funciones y alias con ese nombre (evita recursión). | | ``builtin cd`` | Fuerza el comando interno de bash aunque exista una función ``cd``. | | ``type -a nombre`` | Dice si es alias, función, interno o ejecutable (y dónde está cada uno). | | ``declare -i n`` | Variable entera: ``n="a"`` falla en vez de guardar basura silenciosa. | | ``printf -v var "%s" "$x"`` | Guarda el formateo en ``var`` sin subshell (más rápido que ``var=$(printf ...)``). | | ``PS4='+ ${LINENO}: `` | Personaliza el prefijo de ``bash -x`` para ver el NÚMERO DE LÍNEA de cada traza. |

3 combinaciones que valen oro

```
export PS4='+ ${BASH_SOURCE}:${LINENO}: '; bash -x ./script.sh 2>&1 | head -30
```

La radiografía con esteroides: cada línea de traza dice archivo y número de línea. Depurar así es hacer trampa (legal).

```
main() { parsear_args "$@"; validar; ejecutar; }; main "$@"
```

El patrón ``main``: todo el script vive en funciones y la última línea lo arranca. Orden, testeabilidad y ``local`` en todas partes.

```
if ! mkdir -p "$dir"; then die "no pude crear $dir"; fi
```

``!`` invierte el código: maneja el error del comando directamente en el ``if``, compatible con ``set -e`` sin trucos. La forma elegante de "intenta o muere".

Resumen de la unidad

- Las funciones agrupan comandos reutilizables: parámetros en ``$1`...`, retorno con ``return`` (códigos) o imprimiendo y capturando con ``$()``.

- ``local`` limita las variables a la función: evita choques de nombres y bugs misteriosos.
- ``set -euo pipefail`` es el modo estricto: detiene ante errores, variables indefinidas y fallos en tuberías.
- `getopts` procesa opciones ``-v`, `-h`, `-a valor``: ``while getopts "vha:" opt`, con `OPTARG` y `shift $((OPTIND-1))``.
- Todo script con opciones necesita ``-h`` con una función ``uso``: la interfaz es parte del programa.
- ``bash -x`` muestra cada comando expandido antes de ejecutarlo: la radiografía del script.
- ``set -x`` / ``set +x`` activan el rastreo por secciones; ``trap '...' ERR`` con ``$LINENO`` ubica el fallo exacto.
- `source` carga bibliotecas de funciones: no copies y pegues, reutiliza.
- Estilo profesional: encabezado documentado, nombres claros, comenta el porqué, valida temprano, entrecomilla siempre.
- Un script bien hecho se puede programar, lo puede usar otro y falla de forma informativa: eso es confiabilidad.
- ``return`` devuelve códigos, no datos: para datos usa ``echo` + `$( )``; un ``return 1`` sin mensaje es un misterio.
- ``local`` va en la primera línea de la función: lo que no sale de la función no debe ensuciar lo global.
- ``set -u`` convierte un typo (``$usuario``) en error inmediato en vez de una cadena vacía silenciosa.
- ``pipefail`` es vital en respaldos y despliegues: sin él, un ``tar`` roto al inicio de la tubería pasa desapercibido.
- `getopts` no entiende opciones largas (``--verbose``): para esas usa ``getopt`` (el comando externo) o parseo manual.
- ``bash -n`` valida sintaxis sin ejecutar: pásalo siempre antes de programar un script en cron.
- ``trap ... EXIT`` garantiza limpieza de temporales aunque el usuario mate el script con Ctrl+C.
- ``readonly`` protege constantes como rutas base y versiones: nadie las reasigna por accidente.

- Encabezado profesional mínimo: qué hace, uso con ejemplo, autor y fecha; tu yo del futuro te lo agradecerá.

UNIDAD 9: Procesos, tareas programadas y redes

Tus scripts ya son programas profesionales. Ahora van a salir al mundo: vas a controlar los procesos del sistema, a programar tus scripts para que se ejecuten solos con cron, y a usar la red desde la terminal con SSH, SCP y herramientas de diagnóstico. Esta es la unidad del administrador: lo que separa al que usa Linux del que opera sistemas.

En esta unidad vas a aprender qué son los procesos y cómo verlos con ps, a controlarlos con kill, jobs, bg y fg, a programar tareas con cron y crontab, a conectarte a máquinas remotas con SSH, a copiar archivos con SCP, y a diagnosticar redes con ping, curl y ss.

Por qué importa esta unidad: los servidores no se administran a mano todos los días: se administran con scripts programados que corren solos. El respaldo de tu proyecto final no sirve si tienes que acordarte de ejecutarlo: sirve cuando cron lo ejecuta cada noche sin que pienses en ello. Y SSH es la puerta de entrada a todos los servidores del mundo: el 99 por ciento de la administración remota se hace por SSH.

Hay otra razón: entender procesos te quita el miedo a "colgar" la terminal. Un programa trabado ya no es un problema: lo identificas, lo pausas, lo mandas al fondo o lo terminas con precisión quirúrgica. Y entender cron te da superpoderes de automatización: reportes matutinos, limpieza semanal, respaldos nocturnos, todo sin tu intervención.

Cómo encaja en el curso: las Unidades 6 a 8 te dieron scripts profesionales; esta los pone a trabajar solos y en red. El proyecto final unirá todo: un script de respaldo con funciones y modo estricto, programado con cron, capaz de respaldar incluso máquinas remotas por SSH.

Empecemos. Al terminar vas a ver el sistema como un conjunto de procesos que puedes dirigir, y la red como una extensión de tu terminal.

Explicación detallada

Procesos: los programas en ejecución

Un proceso es un programa corriendo. Tu terminal es un proceso, cada comando que ejecutas crea un proceso, el sistema tiene cientos corriendo en este momento. Cada proceso tiene un número único: el PID.

```
ps aux
```

Muestra todos los procesos del sistema: usuario, PID, uso de CPU y memoria, y el comando. Es la radiografía completa. Para buscar uno específico, el clásico:

```
ps aux | grep nginx
```

Las columnas importantes: PID (el número para controlarlo), %CPU y %MEM (cuánto consume), STAT (su estado: R corriendo, S dormido, Z zombie), y COMMAND (qué es). Cuando un servidor va lento, `ps aux --sort=-%cpu | head` muestra los 10 procesos más tragones de CPU: el diagnóstico de 10 segundos.

`top` (o su primo moderno `htop`) muestra los procesos en vivo, actualizándose: como ps pero en tiempo real, ordenados por consumo. Para salir, `q`. Es el monitor de signos vitales del sistema.

jobs, bg, fg y kill: dirigir la orquesta

Cuando ejecutas un comando largo, la terminal se queda ocupada hasta que termina. Pero puedes tomar el control. Presiona Ctrl+Z para pausar el proceso actual (no lo mata, lo congela):

```
sleep 100
```

(Ctrl+Z) y luego:

```
jobs
```

Muestra tus procesos pausados o en segundo plano, numerados. Para reanudarlo en segundo plano (sigue corriendo sin ocupar tu terminal):

```
bg
```

Para traerlo de vuelta al frente:

```
fg
```

Y para terminar un proceso, kill con su PID (primero averígualo con ps o jobs):

```
kill 1234
```

A pesar de su nombre, kill por defecto no "mata": envía la señal TERM, que le pide al proceso que termine de forma ordenada, guardando lo que tenga que guardar. Solo si se niega usas la fuerza bruta:

```
kill -9 1234
```

La señal 9 (KILL) no se puede ignorar ni atrapar: el sistema lo fulmina al instante, sin darle chance de guardar nada. Regla: primero kill normal, espera unos segundos, y solo si sigue vivo, kill -9. El -9 es el último recurso, no el primero.

También puedes lanzar directamente en segundo plano agregando `&` al final:

```
./respaldo-largo.sh &
```

El script corre liberando tu terminal de inmediato. Y `nohup` lo hace inmune a que cierres la terminal: `nohup ./tarea.sh &` sigue corriendo aunque te desconectes. Así se lanzan los trabajos largos en servidores.

cron: el mayordomo automático

cron es el programador de tareas de Linux: ejecuta comandos automáticamente en horarios definidos. Cada usuario tiene su lista de tareas, que se edita con:

```
crontab -e
```

Cada línea tiene 5 campos de tiempo y el comando:

```
30 2 * * * /home/esteban/respaldo.sh >> /home/esteban/respaldo.log 2>&1
```

Los campos son: minuto (0-59), hora (0-23), día del mes (1-31), mes (1-12), día de la semana (0-7, donde 0 y 7 son domingo). El asterisco significa "todos". Esta línea dice: "a las 2:30 de la mañana, todos los días, ejecuta el respaldo y guarda todo (salida y errores) en el registro".

Ejemplos para tu cheat sheet:

```
0 8 * * 1-5 /home/esteban/reporte.sh
```

A las 8:00 de lunes a viernes: el reporte matutino de días laborales.

```
*/15 * * * * /home/esteban/vigila.sh
```

Cada 15 minutos (`\*/15` = "cada 15"): monitoreo frecuente.

```
0 0 1 * * /home/esteban/limpieza-mensual.sh
```

El día 1 de cada mes a medianoche: limpieza mensual.

Reglas de oro de cron: usa RUTAS ABSOLUTAS en todo (cron corre con un ambiente mínimo, sin tu PATH personal), redirige siempre la salida a un registro (si no, cron intenta mandarte correos que nadie lee), y prueba el script a mano antes de programarlo. Para ver tus tareas: `crontab -l`; para borrarlas todas (con cuidado): `crontab -r`.

SSH: tu terminal en cualquier servidor del mundo

SSH (Secure SHell) te conecta de forma segura a otra máquina y te da su terminal como si estuvieras ahí:

```
ssh usuario@servidor.com
```

Te pide la contraseña y listo: tu prompt ahora es el del servidor remoto. Todo lo que aprendiste en este curso funciona igual allá: ls, grep, tus scripts. Para salir, `exit` o Ctrl+D.

Escribir contraseñas cada vez es tedioso e inseguro para automatizar, así que existen las llaves SSH: un par de archivos (una llave privada que nunca compartes y una pública que instalas en el servidor) que te autentican sin contraseña:

```
ssh-keygen -t ed25519  
ssh-copy-id usuario@servidor.com
```

El primero genera tu par de llaves (acepta todo con Enter para lo básico); el segundo instala tu llave pública en el servidor. Desde entonces, `ssh usuario@servidor.com` entra directo. Y tus scripts pueden conectarse solos: `ssh usuario@servidor "df -h"` ejecuta el comando allá y te trae el resultado. Así se administran flotas de servidores desde un solo script.

Seguridad: tu llave privada (`~/.ssh/id\_ed25519`) debe tener permiso 600 y nadie más debe leerla jamás. Si alguien la copia, tiene tus accesos. Es la llave de tu casa digital: cuídala como tal.

SCP: copiar archivos por la red

SCP copia archivos entre máquinas usando la conexión SSH:

```
scp reporte.txt usuario@servidor:/home/usuario/
```

Sube el archivo al servidor. Al revés para descargar:

```
scp usuario@servidor:/var/log/app.log ./app-remoto.log
```

Con ``r`` copia carpetas completas. La sintaxis se lee como el `cp` de la Unidad 2, pero con direcciones remotas ``usuario@servidor:ruta``. Para automatizar respaldos hacia otro servidor, `scp` es la pieza que mueve los archivos.

ping, curl y ss: el estetoscopio de red

ping verifica si una máquina responde:

```
ping -c 4 google.com
```

Manda 4 paquetes y reporta los tiempos. Si no responde, o no hay red o la máquina está caída o bloquea pings. Es el "¿me escuchas?" de las redes.

curl descarga o consulta URLs desde la terminal:

```
curl -s https://api.ejemplo.com/estado
```

Trae el contenido de una web o API al terminal. La ``-s`` lo pone silencioso. Es la herramienta para probar APIs, descargar archivos en scripts y verificar que un sitio responde. ``curl -I https://lucasegaacademy.com`` muestra solo las cabeceras HTTP: el chequeo rápido de "¿mi sitio está vivo?".

ss muestra las conexiones y puertos de tu máquina:

```
ss -tlnp
```

Lista los puertos TCP en escucha (`-l`) con números (`-n`) y qué proceso los usa (`-p`). Si tu servidor web no responde, este comando te dice si el programa siquiera está escuchando. Es el diagnóstico de "¿está prendido el servicio?".

Ejemplos paso a paso

Ejemplo 1: doma un proceso rebelde

Vas a crear un proceso largo, perder el control de la terminal y recuperarlo como profesional.

```
sleep 300
```

La terminal queda ocupada. Presiona Ctrl+Z: el proceso se pausa y recuperas tu prompt. Ahora:

```
jobs
```

Ves algo como `[1]+ Detenido sleep 300`: el trabajo 1, pausado. Mándalo al fondo:

```
bg  
jobs
```

Ahora dice "En ejecución": sigue corriendo sin ocupar tu terminal. Puedes trabajar normalmente. Para terminarlo con elegancia:

```
kill %1  
jobs
```

`kill %1` mata el trabajo 1 (el `%` se refiere a trabajos, no a PIDs). La lista de jobs queda vacía: proceso terminado.

Ahora la versión directa: lanza en segundo plano desde el inicio y mátalos por PID:

```
sleep 300 &  
ps aux | grep "sleep 300" | grep -v grep  
kill <PID>
```

El `&` lo lanzó libre, `ps | grep` encontró su PID (el `grep -v grep` excluye al propio grep de los resultados, truco clásico), y kill lo terminó.

Explicación: Ctrl+Z, bg, fg, jobs y kill son el panel de control de tus procesos. Un programa colgado ya no te asusta: lo pausas, lo mueves o lo terminas a voluntad.

Ejemplo 2: tu primera tarea programada

Vas a programar un saludo automático con cron y verificar que se ejecutó.

```
#!/bin/bash
# saludo-cron.sh
echo "[$(date '+%Y-%m-%d %H:%M:%S')] Hola desde cron" >>
"$HOME/practica-linux/saludos.log"
```

Guárdalo, dale permiso y Pruébalo a mano primero (regla de oro):

```
chmod u+x ~/practica-linux/saludo-cron.sh
~/practica-linux/saludo-cron.sh
cat ~/practica-linux/saludos.log
```

Funciona. Ahora prográmalo cada minuto (solo para la prueba):

```
crontab -e
```

Agrega la línea (usa tu ruta real):

```
* * * * * /home/TU_USUARIO/practica-linux/saludo-cron.sh
```

Guarda y sal. Espera dos minutos y revisa:

```
cat ~/practica-linux/saludos.log
```

Hay líneas nuevas con la hora: cron está ejecutando tu script solo. Ahora limpia: ``crontab -e`` y borra o comenta la línea (con ``#`` al inicio), porque un saludo cada minuto para siempre es spam contra ti mismo.

Explicación: el ciclo es probar a mano, programar, verificar en el registro, y ajustar. Nunca programes lo que no probaste: depurar un cron a ciegas es frustrante porque no ves los errores en vivo (por eso el registro es obligatorio).

Ejemplo 3: sesión SSH completa (simulada en local)

Si no tienes un servidor remoto, puedes practicar SSH contra tu propia máquina instalando el servidor local (en Ubuntu: ``sudo apt install openssh-server``). Si no puedes instalarlo, lee el ejemplo como mapa para cuando tengas tu servidor.

```
ssh-keygen -t ed25519 -f ~/.ssh/id_ed25519 -N ""
```

Genera tu par de llaves sin frase de contraseña (-N "" vacía; en producción usa una frase).
Luego:

```
ssh-copy-id $USER@localhost
```

Instala tu llave pública para entrar a tu propia máquina. Ahora:

```
ssh $USER@localhost "hostname; whoami"
```

Entras sin contraseña, ejecutas dos comandos allá (que es aquí, en este ejercicio) y sales automáticamente. La salida muestra el resultado de los comandos remotos.

```
echo "hola remoto" > /tmp/remoto-test.txt  
scp /tmp/remoto-test.txt $USER@localhost:/tmp/copia-remota.txt  
ssh $USER@localhost "cat /tmp/copia-remota.txt"
```

Subes un archivo por SCP y lo lees ejecutando cat remotamente. El ciclo completo: conectar, copiar, ejecutar, todo sin contraseña gracias a las llaves.

Explicación: con llaves SSH, tus scripts pueden administrar servidores sin intervención humana. Es la base de la automatización remota: el proyecto final podrá respaldar un servidor a kilómetros de distancia igual que respalda tu carpeta local.

Ejemplo 4: chequeo de salud de un sitio

Vas a armar el chequeo que harías para vigilar que tu página siga en línea.

```
#!/bin/bash
# vigila-sitio.sh - Verifica que un sitio responda
set -euo pipefail

sitio="${1:-https://lucasegaacademy.com}"
codigo=$(curl -s -o /dev/null -w "%{http_code}" --max-time 10 "$sitio")

if [ "$codigo" -eq 200 ]; then
    echo "[$(date '+%H:%M:%S')] $sitio OK ($codigo)"
    exit 0
else
    echo "[$(date '+%H:%M:%S')] ALERTA: $sitio responde $codigo" >&2
    exit 1
fi
```

`curl -w "%{http\_code}"` extrae solo el código HTTP de respuesta; `-o /dev/null` descarta el contenido (solo nos importa el código); `--max-time 10` no espera más de 10 segundos. Pruébalo:

```
chmod u+x ~/practica-linux/vigila-sitio.sh
~/practica-linux/vigila-sitio.sh https://lucasegaacademy.com; echo "Código: $"
~/practica-linux/vigila-sitio.sh https://sitio-que-no-existe-xyz.com; echo "Código: $"
```

El primero reporta OK con código 0; el segundo alerta con código 1. Programado con cron cada 15 minutos y con el registro correspondiente, tienes un monitor de sitios web casero y funcional.

Explicación: curl + código de salida honesto + cron = monitoreo. Este patrón (verificar, reportar, programar) es exactamente como funcionan los sistemas de vigilancia profesionales, solo que en versión de una página.

Errores comunes y casos reales

Esta unidad toca sistemas vivos: procesos corriendo, tareas programadas, servidores remotos. Los errores aquí no solo rompen scripts: tiran servicios y dejan puertas abiertas. Apréndelos con respeto.

Caso 1: `kill -9` como primera opción en vez de la última

Síntoma: matas un proceso con `kill -9`, y luego la aplicación no arranca bien, deja archivos de bloqueo, corrompe su base de datos o pierde la última transacción.

Por qué ocurre: `9` (SIGKILL) fulmina el proceso sin darle oportunidad de nada: no cierra archivos, no guarda estado, no libera recursos, no avisa a nadie. Es el equivalente a jalar el cable de corriente. Muchas aplicaciones necesitan su apagado ordenado para quedar consistentes.

Solución: escala de menor a mayor. Primero TERM (el `kill` sin número, que pide amablemente), espera unos segundos, y solo si sigue vivo, KILL:

```
kill 1234                # SIGTERM: "por favor termina"
sleep 5
kill -0 1234 2>/dev/null || echo "ya terminó"
kill -9 1234            # solo si el amable no funcionó
```

`kill -0 PID` no mata nada: solo pregunta si el proceso existe. Úsalo para verificar antes de escalar.

Caso 2: `pkill` o `killall` que matan de más (incluyendo tu propia sesión)

Síntoma: `pkill python` para detener un script colgado... y también mata tu servidor web en Python, tu herramienta de monitoreo y el proceso del compañero. O `kill %1` mata el job equivocado porque los números cambiaron.

Por qué ocurre: `pkill`/`killall` matan por NOMBRE, y los nombres se repiten. `killall ssh` en el servidor remoto donde estás conectado por SSH puede cortarte tu propia sesión. Los números de job (`%1`) se reasignan cuando los jobs terminan.

Solución: verifica ANTES con `pgrep -a` (muestra qué mataría) y sé específico:

```
pgrep -a python          # mira la lista ANTES de matar
pkill -f "mi_script.py"  # -f busca en la línea completa: más específico
# o mejor, mata por PID exacto:
kill 1234
```

Regla: `pgrep` primero, `pkill` después. Nunca al revés.

Caso 3: el cron que "funciona a mano" pero no hace nada programado

Síntoma: tu script corre perfecto en la terminal, lo programas en cron, y no pasa nada. O peor: sí corre pero falla a la mitad. El log está vacío o tiene errores rarísimos.

Por qué ocurre: cron ejecuta con un entorno mínimo: PATH reducido (`/usr/bin:/bin`), sin tus variables, sin tu `~/.bashrc`, con otro directorio de trabajo. Lo que en tu terminal se resuelve por tu entorno configurado, en cron no existe.

Solución: el checklist del cron profesional:

```
# En el crontab, define lo básico arriba:
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
# En la línea del cron: rutas ABSOLUTAS y log SIEMPRE:
*/5 * * * * /home/esteban/scripts/vigilante.sh >> /home/esteban/logs/vigilante.log
2>&1
```

Y la regla de oro: prueba el script con el entorno pelado antes de programarlo: `env -i PATH=/usr/bin:/bin /ruta/script.sh`. Si funciona así, funcionará en cron.

Caso 4: SSH que sigue pidiendo contraseña aunque ya copiaste la llave

Síntoma: hiciste `ssh-copy-id`, todo pareció funcionar, pero `ssh servidor` sigue pidiendo password. La automatización sin contraseña no jala.

Por qué ocurre: SSH es paranoico con los permisos (y con razón). Si `~/.ssh` no es 700, si `authorized\_keys` no es 600, o si tu HOME es escribible por otros, el servidor IGNORA tus llaves en silencio y cae al password. También falla si pegaste la llave con un salto de línea roto o espacios de más.

Solución: verifica y corrige permisos en AMBOS lados:

```
# En tu máquina y en el servidor:
chmod 700 ~/.ssh
chmod 600 ~/.ssh/authorized_keys
# Diagnóstico verboso para ver qué está pasando:
ssh -v usuario@servidor 2>&1 | grep -i "auth"
```

En la salida de `-v` busca `Offering public key` y luego `Server accepts key` (bien) o `Authentications that can continue: password` (tu llave fue rechazada: revisa permisos y formato).

Caso 5: `scp` con el `:` olvidado o el orden invertido

Síntoma: `scp archivo usuario@servidor/ruta` crea un archivo local llamado `usuario@servidor/ruta` (sí, con arroba y todo en el nombre), o copias el archivo AL REVÉS de lo que querías y sobrescribes el original bueno con el viejo.

Por qué ocurre: sin los dos puntos después del host, scp cree que todo es una ruta local. Y el orden es siempre `origen destino`: si los inviertes, el "respaldo" pisa al original.

Solución:

```
# BIEN: dos puntos separan host de ruta
scp reporte.pdf usuario@servidor:/home/usuario/docs/
# Traer del servidor a tu máquina (origen remoto, destino local):
scp usuario@servidor:/home/usuario/docs/reporte.pdf ./
# Carpetas: -r, y verifica con ls qué hay de cada lado ANTES
ls -la ./local/ && ssh usuario@servidor "ls -la /home/usuario/docs/"
```

Hábito profesional: lista origen y destino antes de copiar, sobre todo si el destino ya existe.

Caso 6: `curl` que "descarga bien" un error 404

Síntoma: tu script descarga con `curl -o datos.json https://api...`, el archivo existe, el script sigue feliz... pero `datos.json` contiene una página HTML de "404 Not Found" y todo lo que sigue procesa basura.

Por qué ocurre: por defecto curl considera exitosa cualquier respuesta HTTP que llegue completa, INCLUYENDO 404s y 500s. El código de salida es 0 aunque el servidor te dijo "no existe".

Solución: `-f` (`fail`) hace que curl devuelva error ante respuestas 400/500, y `-sS` silencia el progreso pero muestra errores reales:

```
if curl -fsSL -o datos.json "https://api.ejemplo.com/datos"; then
    echo "descarga ok"
else
    echo "ERROR: falló la descarga (código $?)" >&2
    exit 1
fi
```

Para APIs, agrega `-w "%{http_code}"` cuando necesites decidir según el código exacto (200, 404, 429...).

Caso real 1: el `kill -9` al postgres un viernes en la noche

Un administrador vio el postgres al 100% de CPU, supuso "se colgó" e hizo `kill -9` al proceso principal. Postgres murió sin hacer checkpoint: al rearrancar tuvo que hacer recovery de varios GB de WAL, tardó 40 minutos, y dos transacciones quedaron en estado ambiguo que hubo que conciliar a mano el lunes.

El procedimiento profesional ante un servicio atascado:

```
# 1. Identifica con precisión
ps aux | grep postgres | grep -v grep
# 2. Intenta el apagado ordenado del SERVICIO, no del proceso
sudo systemctl stop postgresql
# 3. Si no responde, TERM al PID y espera
kill 1234; sleep 10; kill -0 1234 2>/dev/null && echo "sigue vivo"
# 4. KILL solo como último recurso, y avisa al equipo
```

Moraleja: los servicios con estado (bases de datos, colas) se apagan por su mecanismo oficial. `kill -9` es para procesos sin estado o cuando ya todo lo demás falló.

Caso real 2: el cron que llenó el disco con correos

Un script de monitoreo corría cada minuto en cron e imprimía su reporte en stdout "para depurar después". Cron, por diseño, manda por correo local TODO lo que el job imprime. Nadie leía ese correo. En tres meses, `/var/spool/mail` creció a 18 GB y el disco se llenó, tirando otros servicios.

Dos hábitos que lo evitan:

```
# 1. En cron: todo a un log con rotación, nada al correo
* * * * * /home/esteban/scripts/mon.sh >> /home/esteban/logs/mon.log 2>&1
# 2. Si el script solo debe hablar cuando algo anda mal: silencia lo normal
* * * * * /home/esteban/scripts/mon.sh > /dev/null 2>&1
# (y que el script mismo escriba a su log solo eventos importantes)
```

Y si quieres desactivar el correo de cron del todo: `MAILTO=""` al inicio del crontab. Pero mejor: log con fecha y tamaño controlado.

Miniproyecto guiado de la unidad

Vas a construir `vigilante.sh`: un monitor que revisa si tu servicio web responde, lo registra todo en un log con fecha, y está diseñado para correr en cron cada 5 minutos. Usa `pgrep`/`curl`, códigos de salida honestos, rutas absolutas y el patrón verificar -> reportar -> programar de los sistemas profesionales.

Paso 1: el chequeo básico con curl y código honesto

```
#!/bin/bash
# vigilante.sh - vigila que el servicio web responda
# Diseñado para correr en cron cada 5 minutos
set -euo pipefail

URL="{1:-http://localhost:8080/salud}"
LOG="$HOME/practica-linux/vigilante.log"

marca() { date '+%Y-%m-%d %H:%M:%S'; }

if curl -fsS --max-time 10 -o /dev/null "$URL"; then
    echo "$(marca) OK: $URL responde" >> "$LOG"
    exit 0
else
    codigo=$?
    echo "$(marca) FALLO: $URL no responde (curl=$codigo)" >> "$LOG"
    exit 1
fi
```

`--max-time 10` evita que un servidor colgado congele tu cron eternamente. El `codigo=$?` inmediato captura el código real de curl.

Paso 2: verificación del proceso como segunda opinión

Un servicio puede tener el proceso vivo pero no responder (o al revés). Agrega chequeo de proceso con `pgrep`:

```

NOMBRE="${2:-python3}"

if pgrep -f "$NOMBRE" > /dev/null; then
    echo "$(marca) PROCESO: '$NOMBRE' en ejecución (PIDs: $(pgrep -f "$NOMBRE" | tr
,
' ' '))" >> "$LOG"
else
    echo "$(marca) PROCESO: '$NOMBRE' NO encontrado" >> "$LOG"
fi

```

Pon este bloque ANTES del chequeo curl. Así tu log cuenta la historia completa: proceso vivo + responde = todo bien; proceso vivo + no responde = servicio atascado; proceso muerto = caído.

Paso 3: prueba manual contra algo real

Levanta un servidor de prueba con Python y vigílalo:

```

chmod u+x ~/practica-linux/vigilante.sh
# Terminal 1: servidor de prueba
cd ~/practica-linux && python3 -m http.server 8080
# Terminal 2: vigila (ajusta URL porque http.server no tiene /salud)
~/practica-linux/vigilante.sh http://localhost:8080/ python3
echo "código: $?"
cat ~/practica-linux/vigilante.log

```

Verificación: código 0, y el log con líneas `OK` y `PROCESO ... en ejecución`.

Paso 4: simula la caída y observa el FALLO

```

# Mata el servidor de la terminal 1 con Ctrl+C (o pkill -f "http.server")
~/practica-linux/vigilante.sh http://localhost:8080/ python3
echo "código: $?"
tail -3 ~/practica-linux/vigilante.log

```

Verificación: código 1 y líneas `FALLO` en el log. Tu monitor ya distingue "todo bien" de "algo mal" con códigos honestos: eso es lo que cron y otros sistemas necesitan.

Paso 5: programa en cron (con el checklist)

```
crontab -e
```

Agrega (ajusta tu usuario real):

```
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
*/5 * * * * /home/esteban/practica-linux/vigilante.sh http://localhost:8080/
python3 >> /home/esteban/practica-linux/vigilante.log 2>&1
```

Antes de guardar, prueba con entorno pelado como lo haría cron:

```
env -i PATH=/usr/bin:/bin /home/esteban/practica-linux/vigilante.sh
http://localhost:8080/ python3
echo "con entorno mínimo: $?"
```

Si funciona con `env -i`, funcionará en cron. Verifica que quedó programado con `crontab -l`.

Paso 6: lee tu log como un profesional

Después de unos minutos (o simulando varias corridas), analiza:

```
grep -c "OK" ~/practica-linux/vigilante.log
grep "FALLO" ~/practica-linux/vigilante.log | tail -5
awk '{print $1}' ~/practica-linux/vigilante.log | sort | uniq -c
```

La última línea te dice cuántas revisiones hubo por día. Esto es monitoreo básico real: recolección automática + análisis con las herramientas de la Unidad 5.

Reto final (sin solución)

1. Agrega un contador de fallos consecutivos (guárdalo en un archivo `/tmp/vigilante.fallos`): solo escribe "ALERTA" al log cuando haya 3 fallos SEGUIDOS, y reinicia el contador cuando haya un OK. Así evitas spam por un fallo aislado.
2. Si hay 3 fallos seguidos, intenta reiniciar el servicio tú mismo (pista: el comando de arranque como variable `ARRANQUE`, ejecútalo con `nohup ... &`) y registra "REINICIO INTENTADO".
3. Agrega una opción `--resumen` que imprima cuántos OK, FALLOS y ALERTAS hay en el log, usando `grep -c`. Debe funcionar aunque el log no exista todavía.

Si lo logras, tienes un mini-sistema de monitoreo con alerta inteligente y auto-recuperación: exactamente lo que hacen las herramientas caras, en 60 líneas.

Práctica

En `~/practica-linux`. Todo script con shebang, modo estricto y rutas absolutas (piensa en cron).

Ejercicio 1: top de tragones

Enunciado: con una sola línea, muestra los 5 procesos que más CPU consumen en este momento, con usuario, PID, %CPU y comando. Luego identifica si hay alguno tuyo en la lista.

Solución explicada paso a paso. Paso 1: ``ps aux --sort=-%cpu | head -6``: ordena por CPU descendente y toma 6 líneas (1 de encabezado + 5 procesos). Paso 2: lee las columnas: USER, PID, %CPU, COMMAND. Paso 3: busca tu usuario en la columna USER: esos son tus procesos. Paso 4: si ves uno que no reconoces consumiendo mucho, investiga con ``ps -p PID -o pid,ppid,cmd`` (muestra su proceso padre: quién lo lanzó). Paso 5: anota el comando en tu cheat sheet como "diagnóstico de lentitud en 10 segundos". Nunca mates un proceso que no entiendas: primero investiga qué es con su PPID y su comando.

Ejercicio 2: el dormilón controlado

Enunciado: lanza ``sleep 120`` en segundo plano, verifica que está corriendo con jobs y ps, tráelo al frente con fg, paúsalo con Ctrl+Z, mándalo al fondo con bg y finalmente térmalo con kill. Documenta cada estado.

Solución explicada paso a paso. Paso 1: ``sleep 120 &``: corre libre; anota el número de trabajo. Paso 2: ``jobs`` debe decir "En ejecución"; ``ps aux | grep "sleep 120" | grep -v grep`` muestra su PID. Paso 3: ``fg``: lo traes al frente (la terminal se ocupa); presiona Ctrl+Z: se pausa y recuperas el prompt. Paso 4: ``bg``: vuelve a correr en el fondo. Paso 5: ``kill %1`` (o con su PID): termina; ``jobs`` queda vacío y el ps ya no lo muestra. Paso 6: escribe en tu cheat sheet el mapa: "& = nace libre, Ctrl+Z = pausa, bg = al fondo, fg = al frente, kill = termina". Este es el control total de procesos interactivos.

Ejercicio 3: reporte programado

Enunciado: crea ``reporte-cron.sh`` que genere en `~/practica-linux/reporte-diario.log`` una línea diaria con fecha, espacio libre en disco y memoria libre. Pruébalo a mano, prográmalo con cron para las 9:00 AM y verifica al día siguiente (o simula la verificación ejecutándolo tú).

Solución explicada paso a paso. Paso 1: el script con modo estricto: ``fecha=$(date +%F)``, ``disco=$(df -h / | awk 'NR==2{print $4}``, ``mem=$(free -h | awk '/Mem:/{print $7}``, y ``echo "$fecha | Disco libre: $disco | Mem libre: $mem" >> "$HOME/practica-linux/reporte-diario.log"`. Usa ``$HOME`` en lugar de ``~`` porque en scripts el ``~`` no siempre expande. Paso 2: pruébalo a mano dos veces y verifica que el log crece con ``>>``. Paso 3: ``crontab -e`` y agrega ``0 9 * * * /home/TU_USUARIO/practica-linux/reporte-cron.sh``. Paso 4: verifica con ``crontab -l`` que quedó registrada. Paso 5: al día siguiente (o simulando), ``cat`` el log: debe tener la línea automática. La lección: cron + rutas absolutas + registro = automatización confiable.

Ejercicio 4: copia de seguridad remota (simulada)

Enunciado: usando scp contra localhost (o entre dos carpetas simulando origen y destino si no tienes SSH), copia tu carpeta ``~/practica-linux/documentos`` a un "servidor" y verifica que llegó completa comparando el número de archivos.

Solución explicada paso a paso. Paso 1: si tienes SSH local: ``scp -r ~/practica-linux/documentos "$USER@localhost:/tmp/respaldo-remoto/"``. Si no: simula con ``cp -r`` a ``/tmp/respaldo-remoto/`` entendiendo que en producción sería scp. Paso 2: cuenta origen: ``find ~/practica-linux/documentos -type f | wc -l``. Paso 3: cuenta destino: ``ssh $USER@localhost "find /tmp/respaldo-remoto -type f | wc -l"`` (o find local en la simulación). Paso 4: los números deben coincidir: verificación de integridad básica. Paso 5: anota el patrón: "copiar + contar ambos lados + comparar" es la verificación mínima de todo respaldo. En el proyecto final harás exactamente esto antes de dar el respaldo por bueno.

Ejercicio 5: el vigilante nocturno

Enunciado: escribe ``vigilante.sh`` que cada vez que se ejecute verifique: espacio en disco (alerta si queda menos del 20%), un sitio web (usa tu script vigila-sitio.sh o intégralo) y escriba todo en un registro con fecha. Debe salir con código 1 si algo falló.

Solución explicada paso a paso. Paso 1: shebang + modo estricto + ``fallos=0`` como contador. Paso 2: disco: ``uso=$(df / | awk 'NR==2{print $5}' | tr -d '%')``; ``if [ "$uso" -gt 80 ]``: alerta a stderr y ``fallos=$((fallos+1))``. Paso 3: sitio: llama a tu verificador o integra el curl; si el código no es 200, alerta y suma un fallo. Paso 4: todo lo informativo va al registro con ``>>`` y fecha; las alertas además a stderr. Paso 5: ``exit $fallos``: si hubo fallos, el código es distinto de 0 y cron lo notará. Paso 6: pruébalo a mano en condición normal (código 0) y provocando un fallo (verifica un sitio inventado: código 1). Este script es el corazón de un sistema de monitoreo: en el proyecto final lo adaptarás para vigilar tus respaldos.

Tarjeta de referencia rápida

| Comando / sintaxis | Descripción | |---|---| | ``ps aux`` | Lista TODOS los procesos con usuario, CPU, memoria y comando. | | ``ps -ef`` | Vista alternativa con PPID (quién lanzó a quién). | | ``pgrep -a nombre`` | PIDs que coinciden con el nombre (muestra el comando con ``-a``). | | ``pgrep -f "texto"`` | Busca en la línea de comando completa, más específico. | | ``top`` / ``htop`` | Monitor en vivo de procesos (htop más amigable). | | ``kill PID`` | Envía SIGTERM: pide al proceso que termine con educación. | | ``kill -9 PID`` | SIGKILL: fulmina sin avisar. Último recurso. | | ``kill -l`` | Lista todas las señales disponibles. | | ``kill -0 PID`` | No mata: solo pregunta si el proceso existe (código 0 = sí). | | ``pkill nombre`` | Mata por nombre (verifica antes con ``pgrep -a``). | | ``jobs`` | Lista los trabajos de tu sesión actual. | | ``comando &`` | Lanza el comando en segundo plano, libre desde el inicio. | | ``Ctrl+Z`` / ``bg`` / ``fg`` | Pausa el proceso actual / lo manda al fondo / lo trae al frente. | | ``nohup cmd &`` | El proceso sobrevive aunque cierres la terminal. | | ``crontab -e`` / ``-l`` / ``-r`` | Edita / lista / borra TUS tareas programadas. | | ``* * * * *` | Campos cron: minuto hora día-mes mes día-semana. | | ``*/5 * * * *` | Cada 5 minutos. ``0 2 * * *` = diario a las 2 am. | | ``@daily``, ``@reboot`` | Atajos: una vez al día / al arrancar el sistema. | | ``ssh usuario@servidor`` | Abre la terminal de otro servidor. | | ``ssh -i llave.pem usuario@host`` | Usa una llave específica en vez de la default. | | ``ssh-keygen -t ed25519`` | Crea tu par de llaves (moderno y rápido). | | ``ssh-copy-id usuario@servidor`` | Instala tu llave pública en el servidor. | | ``scp archivo usuario@host:ruta`` | Copia un archivo al servidor (el ``:`` es obligatorio). | | ``scp -r carpeta usuario@host:ruta`` | Copia una carpeta completa. | | ``ping -c4 host`` | 4 paquetes para probar conectividad básica. | | ``curl -fsSL URL -o archivo`` | Descarga: falla ante errores HTTP, silencioso, sigue redirects. | | ``curl -I URL`` | Solo los encabezados HTTP (ver si responde y qué dice). | | ``ss -tlnp`` | Puertos TCP en escucha con el proceso dueño (con sudo ves todo). |

5 one-liners que vas a usar toda la vida

```
pgrep -af "mi_app" || echo "no está corriendo"
```

¿Mi servicio está vivo? Una línea, código honesto, listo para un ``if``.

```
kill 1234; sleep 5; kill -0 1234 2>/dev/null && kill -9 1234
```

Apagado escalonado: primero amable, espera, y solo si sigue vivo, fulmina.

```
*/10 * * * * /home/esteban/scripts/tarea.sh >> /home/esteban/logs/tarea.log 2>&1
```

El patrón cron completo: cada 10 minutos, rutas absolutas, todo al log.

```
ssh -i ~/.ssh/mi_llave.pem -o StrictHostKeyChecking=no usuario@servidor "uptime"
```

Ejecuta un comando remoto sin interacción: base de toda automatización SSH.

```
ss -tlnp | grep ":8080"
```

¿Hay algo escuchando en el puerto 8080? Diagnóstico de "no me responde el servicio" en 2 segundos.

Más comandos que completan la unidad

| Comando / sintaxis | Descripción | |---|---| | `timeout 10 comando` | Mata el comando si tarda más de 10 s: fusible contra colgados en scripts. | | `watch -n5 comando` | Repite el comando cada 5 s en pantalla: monitoreo manual exprés. | | `lsof -i :8080` | Qué proceso usa el puerto 8080 (alternativa detallada a `ss -tlnp`). | | `fuser -k 8080/tcp` | Mata lo que esté usando el puerto TCP 8080 (cuando "el puerto está ocupado"). | | `nice -n 10 comando` | Lanza con prioridad baja; `renice -n 10 -p PID` la cambia en caliente. | | `uptime` / `w` | Carga del sistema y quién está conectado: el parte diario del servidor. | | `free -h` | Memoria RAM y swap en formato legible: "¿me estoy quedando sin memoria?". | | `df -h` / `du -sh dir` | Espacio libre por disco / cuánto pesa una carpeta: el dúo del disco lleno. | | `dmesg` | Mensajes del kernel: hardware, discos y errores a bajo nivel (combínalo con `tail`). | | `systemctl status srv` | Estado de un servicio systemd; `restart`/`stop`/`start` lo controlan. | | `journalctl -u srv -f` | Bitácora en vivo de un servicio systemd (`-f` = seguir como `tail -f`). | | `rsync -avz o/ usuario@h:d/` | Copia inteligente por red: solo transfiere lo cambiado, con compresión. | | `sftp usuario@host` | FTP sobre SSH interactivo: `put`/`get` para mover archivos a mano. | | `last` | Quién entró al sistema y cuándo: auditoría básica de accesos. |

3 combinaciones que valen oro

```
timeout 10 curl -fsS http://srv/salud || echo "ALERTA: sin respuesta en 10s" | tee -a alerta.log
```

Chequeo con fusible: si el servicio se cuelga, `timeout` lo mata a los 10 s y la alerta queda registrada. Sin `timeout`, tu cron se congela.

```
ps aux --sort=-%cpu | head -6
```

Los 5 procesos que más CPU comen, ordenados: el primer diagnóstico cuando el servidor "está lento". Cambia a ``-%mem`` para memoria.

```
rsync -avz --delete /datos/ usuario@respaldo:/respaldos/datos/
```

Réplica exacta en el servidor de respaldo: transfiere solo cambios y ``--delete`` borra allá lo que ya no existe aquí. El espejo profesional (úsalo con cuidado: ``--delete`` no perdona).

Resumen de la unidad

- Un proceso es un programa corriendo; `ps aux` los muestra todos, `top/htop` en vivo; el PID es su número único.
- `Ctrl+Z` pausa, `jobs` lista, `bg` manda al fondo, `fg` trae al frente, ``&`` lanza libre desde el inicio.
- `kill` pide terminar con educación (TERM); ``kill -9`` fulmina sin avisar: último recurso, no primero.
- `cron` programa tareas: ``crontab -e`` edita; 5 campos de tiempo (min hora día mes día-semana); ``*`` es "todos".
- En `cron` usa rutas absolutas, redirige todo a un registro y prueba el script a mano antes de programarlo.
- SSH te da la terminal de otro servidor: ``ssh usuario@servidor``; las llaves evitan contraseñas y permiten automatizar.
- ``ssh-keygen`` crea tu par de llaves; ``ssh-copy-id`` instala la pública; cuida tu privada con permiso 600.
- SCP copia por la red: ``scp archivo usuario@servidor:ruta``; ``-r`` para carpetas.
- `ping` verifica conectividad, `curl` consulta URLs y APIs, `ss -tlnp` muestra puertos en escucha.
- Automatizar = script probado + `cron` + registro: el triángulo de la administración confiable.
- Escala de apagado: TERM, espera, verifica con ``kill -0``, y solo entonces KILL; ``-9`` primero es de amateurs.
- ``pgrep -a`` antes de ``pkill`` siempre: matar por nombre sin mirar es jugar a la ruleta.
- ``SIGHUP`` (1) muchos demonios lo usan para recargar configuración sin reiniciar: ``kill -HUP PID``.

- ``disown`` desvincula un job de tu sesión para que no muera al cerrar la terminal (alternativa a `nohup`).
 - En cron, ``MAILTO=""`` desactiva el correo automático; mejor aún: todo a log y correo solo en emergencias.
 - ``%`` en una línea de cron significa salto de línea: escápalo anteponiendo una barra invertida o evítalo en comandos con ``date +%F``.
 - ``ssh -t`` reserva una terminal remota: necesario para ``sudo`` interactivo en el servidor.
 - En scripts automáticos con SSH, ``StrictHostKeyChecking=no`` evita que se atoren... úsalo sabiendo que confías en la red.
 - ``curl -w "%{http_code} %{time_total}``
- `""`` mide código y tiempo de respuesta: monitoreo con esteroides.

UNIDAD 10: Proyecto final y graduación

Llegó la hora de la verdad. Todo lo que aprendiste en las 9 unidades anteriores se une aquí en un solo proyecto real: un sistema de respaldo automatizado completo, con rotación de copias, registro de actividades, verificación de integridad y ejecución programada. No es un ejercicio de juguete: es una herramienta que podrías usar en un trabajo de verdad desde mañana.

En esta unidad vas a diseñar y construir el script de respaldo pieza por pieza: configuración, funciones, lógica principal, rotación, verificación y reporte. Luego lo vas a programar con cron, a probarlo rompiendo cosas a propósito y a documentarlo. Y cerraremos con el repaso integral de todo el curso y tu graduación.

Por qué este proyecto: el respaldo es la tarea más noble de la administración de sistemas. Los discos fallan, los archivos se borran por accidente, los ataques existen. El profesional no reza para que no pase: se prepara para cuando pase. Un buen sistema de respaldo es la diferencia entre "perdimos todo" y "restauramos en una hora". Y construirlo tú mismo te enseña más que diez tutoriales.

Cómo encaja: cada unidad aportó una pieza. La 1 y 2: navegar y manipular archivos. La 3: permisos correctos. La 4: redirección y registros. La 5: filtrar y verificar con grep/awk. La 6: scripts con argumentos. La 7: decisiones y bucles. La 8: funciones, modo estricto y estilo. La 9: cron y red. El proyecto las usa todas, sin excepción.

Empecemos. Al terminar tendrás un sistema funcionando, programado y documentado, y el conocimiento completo del curso integrado.

Explicación detallada

Diseñando el sistema de respaldo

Antes de escribir código, diseñamos. Un buen sistema de respaldo responde estas preguntas:

¿Qué se respalda? Las carpetas importantes: documentos, proyectos, configuraciones. En el proyecto usaremos `~/practica-linux/documentos` como ejemplo, pero el script aceptará cualquier lista.

¿Dónde se guarda? En una carpeta de respaldos, idealmente en otro disco o servidor. Usaremos `~/respaldos`.

¿Cuántas copias se guardan? La rotación: guardamos las últimas N copias (por ejemplo 7, una por día de la semana) y borramos las más viejas. Sin rotación, el disco se llena; con rotación, siempre tienes la última semana.

¿Cómo se comprime? Con `tar + gzip` en un solo archivo por respaldo: ``respaldo-2024-01-15.tar.gz``. Un archivo por día es fácil de gestionar, mover y verificar.

¿Cómo sabemos que funcionó? Con verificación (contar archivos antes y después, como en la Unidad 9) y un registro con fecha de cada ejecución: éxitos y fallos.

El diseño en funciones: ``cargar_config``, ``verificar_origen``, ``crear_respaldo``, ``rotar``, ``verificar_integridad``, ``registrar``. Cada función hace una cosa, con nombres claros. Así se construye software serio en bash.

El script completo, pieza por pieza

Aquí está el sistema completo. Léelo por partes: cada bloque usa algo que ya dominas.

```
#!/bin/bash
# respaldo.sh - Sistema de respaldo automatizado con rotación
# Uso: respaldo.sh [-n copias] [-v] origen1 [origen2 ...]
set -euo pipefail

# --- Configuración ---
DESTINO="$HOME/respalos"
COPIAS=7
VERBOSE=0
LOG="$HOME/respalos/respaldo.log"

# --- Biblioteca mínima ---
log() {
    local msg="[$(date '+%Y-%m-%d %H:%M:%S')] $1"
    echo "$msg" | tee -a "$LOG"
}

error() {
    echo "[$(date '+%Y-%m-%d %H:%M:%S')] ERROR: $1" | tee -a "$LOG" >&2
}

uso() {
    echo "Uso: $0 [-n copias] [-v] origen1 [origen2 ...]"
    echo "  -n N   Guarda las últimas N copias (default: 7)"
    echo "  -v     Modo detallado"
    echo "  -h     Esta ayuda"
}

# --- Opciones ---
while getopts "n:vh" opt; do
    case "$opt" in
        n) COPIAS="$OPTARG" ;;
        v) VERBOSE=1 ;;
        h) uso; exit 0 ;;
        *) uso >&2; exit 1 ;;
    esac
done
shift $((OPTIND - 1))
```

Después de procesar las opciones, el script valida que haya al menos un origen y que el número de copias sea válido:

```

if [ $# -eq 0 ]; then
    uso >&2
    exit 1
fi

if ! [[ "$COPIAS" =~ ^[0-9]+$ ]] || [ "$COPIAS" -lt 1 ]; then
    error "El número de copias debe ser un entero mayor a 0."
    exit 1
fi

mkdir -p "$DESTINO"

```

Observa: modo estricto, funciones `log`/`error` que escriben a pantalla Y al registro con tee, getopt con `-n` requerido, validación de argumentos y del número de copias con regex, y `mkdir -p` del destino. Todo de las Unidades 6, 8 y 4.

```

# --- Respaldo ---
fecha=$(date +%Y-%m-%d_%H%M)
archivo="$DESTINO/respaldo-$fecha.tar.gz"

verificar_origen() {
    local origen="$1"
    if [ ! -e "$origen" ]; then
        error "No existe: $origen (se omite)"
        return 1
    fi
    return 0
}

crear_respaldo() {
    local lista=()
    for origen in "$@"; do
        if verificar_origen "$origen"; then
            lista+=("$origen")
        fi
    done
    if [ ${#lista[@]} -eq 0 ]; then
        error "No hay orígenes válidos para respaldar."
        exit 1
    fi
    log "Respaldando ${#lista[@]} origen(es)..."
    if [ "$VERBOSE" -eq 1 ]; then
        tar -czvf "$archivo" "${lista[@]}" >> "$LOG" 2>&1
    else
        tar -czf "$archivo" "${lista[@]}" >> "$LOG" 2>&1
    fi
    log "Respaldo creado: $archivo"
}

```

`tar -czf` crea el archivo comprimido: c=crear, z=gzip, f=archivo. Con `-v` muestra cada archivo (modo detallado). Los orígenes inválidos se omiten con advertencia en lugar de abortar todo: robustez.

```
# --- Rotación ---
rotar() {
    local total
    total=$(ls -1 "$DESTINO"/respaldo-*.tar.gz 2>/dev/null | wc -l)
    if [ "$total" -gt "$COPIAS" ]; then
        local borrar=$((total - COPIAS))
        log "Rotando: eliminando $borrar copia(s) antigua(s)..."
        ls -ltr "$DESTINO"/respaldo-*.tar.gz | head -n "$borrar" | while read -r
viej; do
            rm -v "$viej" >> "$LOG" 2>&1
            log "Eliminada copia antigua: $(basename "$viej")"
        done
    else
        log "Rotación: $total copia(s), dentro del límite ($COPIAS)."
    fi
}

```

`ls -ltr` lista por fecha, la más vieja primero; `head -n \$borrar` toma las que sobran; el while las borra una por una. La rotación mantiene el historial en el tamaño justo.

```
# --- Verificación ---
verificar() {
    if [ ! -f "$archivo" ]; then
        error "No se creó el archivo de respaldo."
        exit 1
    fi
    if tar -tzf "$archivo" > /dev/null 2>&1; then
        local tam
        tam=$(du -h "$archivo" | cut -f1)
        log "Verificación OK: el archivo es válido ($tam)."
    else
        error "El archivo de respaldo está corrupto."
        exit 1
    fi
}

# --- Principal ---
log "=== Inicio de respaldo ==="
crear_respaldo "$@"
verificar
rotar
log "=== Respaldo completado con éxito ==="
exit 0

```

`tar -tzf` lista el contenido sin extraer: si el archivo está corrupto, falla y el modo estricto detiene todo con el error registrado. El programa principal es legible como una receta: inicio, crear, verificar, rotar, fin.

Programación y prueba de fuego

Con el script probado a mano, lo programas:

```
chmod u+x ~/practica-linux/respaldo.sh
~/practica-linux/respaldo.sh -v ~/practica-linux/documentos
ls -lh ~/respaldos/
```

Verificas el archivo creado y el registro. Luego a cron, cada noche a las 2 AM:

```
crontab -e
0      2      *      *      *      /home/TU_USUARIO/practica-linux/respaldo.sh
/home/TU_USUARIO/practica-linux/documentos  >>  /home/TU_USUARIO/respaldos/cron.log
2>&1
```

Y la prueba de fuego: la restauración. Un respaldo que no se puede restaurar no sirve:

```
mkdir -p /tmp/prueba-restore
tar -xzf ~/respaldos/respaldo-*.tar.gz -C /tmp/prueba-restore
ls -R /tmp/prueba-restore | head -20
```

Extraes la copia más reciente en una carpeta temporal y verificas que tus archivos están ahí, intactos. Haz esta prueba al menos una vez: el día que necesites restaurar de verdad, ya sabrás que funciona.

Documentación del sistema

Un sistema sin documentación es un misterio para el próximo administrador (que podrías ser tú dentro de un año). Crea un `LEEME-respaldo.txt` junto al script con: qué respalda, dónde guarda, cada cuánto corre (y la línea de cron), cuántas copias guarda, dónde está el registro, y cómo restaurar paso a paso. Cinco minutos escribiendo te ahorran horas de adivinanzas después.

Repaso integral: todo lo aprendido, unidades 1 a 9

Antes de graduarte, el recorrido completo. No es nostalgia: es inventario. Un profesional sabe exactamente qué herramientas tiene.

Unidad 1: la terminal y tus primeros comandos. Clases 1, 2 y 3. Aprendiste qué es Linux y por qué gobierna los servidores, la diferencia entre terminal, shell y prompt, y tus comandos de orientación: `whoami`, `pwd`, `ls` con sus opciones, y `cd` para moverte. Aprendiste rutas absolutas y relativas, los atajos `.` y `..`, y a pedir ayuda con `man`, `--help` y `help`. Y tus superpoderes: `Tab` para autocompletar y el historial para no reescribir. En tu proyecto final, esta unidad vive en cada navegación: sabes dónde estás y qué hay antes de actuar.

Unidad 2: archivos y directorios. Clases 4, 5 y 6. Aprendiste `mkdir -p` y `touch` para crear, `cp -r` para copiar, `mv` para mover y renombrar, y `rm` con su disciplina de verificación previa. Aprendiste a ver contenido con `cat`, `less`, `head` y `tail -f`, a trabajar por lotes con los comodines `*` y `?`, y a encontrar todo con `find` y `locate`. En tu proyecto, esta unidad es la manipulación de cada archivo respaldado y cada copia rotada.

Unidad 3: permisos y propiedad. Clases 7, 8 y 9. Aprendiste a leer la cadena `rwx` de `ls -l`, `chmod` simbólico y octal (755 para ejecutables, 644 para archivos, nunca 777), `chown` y `chgrp`, usuarios, grupos y `sudo`. En tu proyecto, tu script es ejecutable gracias a `chmod +x` y tus respaldos tienen los permisos correctos.

Unidad 4: redirección, tuberías y filtros. Clases 10, 11 y 12. Aprendiste los tres flujos, `>` y `>>`, `2>`, el pipe `|`, y `sort`, `uniq`, `cut`, `tr`, `wc` y `tee`. En tu proyecto, los registros crecen con `>>`, los errores se separan con `2>&1` y `tee` escribe a pantalla y archivo a la vez.

Unidad 5: `grep`, `sed` y `awk`. Clases 13, 14 y 15. Aprendiste a buscar con `grep` y regex, a editar en flujo con `sed` y a procesar columnas con `awk`. En tu proyecto, `grep` filtra los registros y `awk` extrae los datos para los reportes.

Unidad 6: variables y tu primer script. Clases 16, 17 y 18. Aprendiste variables sin espacios, las tres comillas, `$()`, los argumentos `$1/$@/$#`, los códigos de salida y `read`. En tu proyecto, todo: configuración en variables, orígenes en argumentos, códigos honestos.

Unidad 7: condicionales y bucles. Clases 19, 20 y 21. Aprendiste `test`, `[ ]` y `[[ ]]`, pruebas de archivos, `if/elif/else`, `case`, `for`, `while/until`, `break` y `continue`. En tu proyecto, las validaciones son `if`, la rotación es un bucle y las opciones son `case` vía `getopts`.

Unidad 8: funciones, depuración y buenas prácticas. Clases 22, 23 y 24. Aprendiste funciones con `local`, `set -euo pipefail`, `getopts`, `bash -x` y el estilo profesional. Tu proyecto final ES esta unidad aplicada: funciones, modo estricto, opciones `-n/-v/-h` y estructura limpia.

Unidad 9: procesos, cron y redes. Clases 25, 26 y 27. Aprendiste `ps`, `kill`, `jobs`, `cron` con sus 5 campos, SSH con llaves, SCP, `ping`, `curl` y `ss`. En tu proyecto, `cron` ejecuta el respaldo cada noche y SSH/SCP lo llevarían a un servidor remoto.

Treinta clases. Diez unidades. Un sistema de respaldo funcionando. Cero excusas.

Ejemplos paso a paso

Ejemplo 1: construye el proyecto pieza por pieza

Vas a armar el script de respaldo en tu zona de práctica y probarlo con datos reales.

```
mkdir -p ~/practica-linux/documentos/fotos
echo "documento importante 1" > ~/practica-linux/documentos/doc1.txt
echo "documento importante 2" > ~/practica-linux/documentos/doc2.txt
echo "foto 1" > ~/practica-linux/documentos/fotos/foto1.txt
```

Crea datos de prueba. Ahora copia el script completo de la sección de Explicación detallada a `~/practica-linux/respaldo.sh` (usa nano o tu editor), dale permiso y ejecútalo:

```
chmod u+x ~/practica-linux/respaldo.sh
~/practica-linux/respaldo.sh -v ~/practica-linux/documentos
```

Observa la salida detallada: cada archivo comprimido aparece en el registro. Verifica:

```
ls -lh ~/respaldos/
cat ~/respaldos/respaldo.log
```

El archivo .tar.gz existe y el registro cuenta la historia completa: inicio, respaldo, verificación y fin. Tu sistema funciona.

Explicación: construir por piezas y probar cada una es la metodología profesional. Nadie escribe 150 líneas de un jalón y reza: se escribe por funciones, se prueba cada función y luego se integran.

Ejemplo 2: prueba la rotación

Vas a verificar que el sistema no deja crecer el historial sin control.

```
~/practica-linux/respaldo.sh -n 3 ~/practica-linux/documentos
sleep 65
~/practica-linux/respaldo.sh -n 3 ~/practica-linux/documentos
sleep 65
~/practica-linux/respaldo.sh -n 3 ~/practica-linux/documentos
sleep 65
~/practica-linux/respaldo.sh -n 3 ~/practica-linux/documentos
ls -1 ~/respaldos/respaldo-*.tar.gz | wc -l
```

(El sleep 65 es para que cada respaldo tenga un nombre de archivo distinto, ya que el nombre incluye hora y minuto. En la práctica real, los respaldos diarios nunca chocan.)

Después de 4 ejecuciones con `-n 3`, debe haber exactamente 3 archivos: la rotación borró el más viejo. Revisa el registro para ver los mensajes de "Eliminada copia antigua".

Explicación: la rotación es lo que hace sostenible el sistema. Sin ella, un respaldo diario llena cualquier disco en meses. Con ella, el espacio usado es constante para siempre: siempre las últimas N copias.

Ejemplo 3: rompe y restaura

La prueba de fuego: simula un desastre y recupérate.

```
# El desastre: borras tus documentos
rm -rf ~/practica-linux/documentos
ls ~/practica-linux/documentos
```

No existe: el desastre es real (en tu zona de práctica, sin consecuencias). Ahora restaura desde tu último respaldo:

```
ultimo=$(ls -1t ~/respaldos/respaldo-*.tar.gz | head -1)
echo "Restaurando desde: $ultimo"
mkdir -p /tmp/rescate
tar -xzf "$ultimo" -C /tmp/rescate
find /tmp/rescate -name "doc1.txt" -exec cat {} \;
```

`ls -1t` ordena por fecha, la más reciente primero; `head -1` toma esa. Extraes en /tmp/rescate y verificas que doc1.txt está ahí con su contenido. Tus "documentos importantes" sobrevivieron al desastre.

```
mkdir -p ~/practica-linux/documentos
cp -r /tmp/rescate/home/*/practica-linux/documentos/* ~/practica-linux/documentos/
2>/dev/null || cp -r /tmp/rescate/*/practica-linux/documentos/*
~/practica-linux/documentos/
ls ~/practica-linux/documentos/
```

Los restauras a su lugar. Desastre simulado, recuperación exitosa, cero pérdidas.

Explicación: esta es LA prueba que importa. Un respaldo nunca probado es una esperanza, no un respaldo. Ahora sabes restaurar, y el día que lo necesites de verdad, será un trámite y no un pánico.

Ejemplo 4: el reporte semanal automático

Vas a extender el sistema con un reporte que resuma la semana de respaldos.

```
#!/bin/bash
# reporte-respaldos.sh - Resume la semana de respaldos
set -euo pipefail

LOG="$HOME/respaldos/respaldo.log"
DESTINO="$HOME/respaldos"

echo "=== Reporte semanal de respaldos ==="
echo "Fecha: $(date +%Y-%m-%d)"
echo ""
echo "Copias guardadas: $(ls -1 "$DESTINO"/respaldo-*.tar.gz 2>/dev/null | wc -l)"
echo "Espacio usado: $(du -sh "$DESTINO" | cut -f1)"
echo ""
echo "Últimos 5 respaldos:"
grep "Respaldo creado" "$LOG" | tail -5
echo ""
echo "Errores de la semana:"
errores=$(grep -c "ERROR" "$LOG" 2>/dev/null || true)
echo "$errores errores registrados"
```

Ejecútalo y observa el resumen: cuántas copias, cuánto espacio, los últimos respaldos y el conteo de errores. grep y tail sobre tu propio registro: las Unidades 4 y 5 trabajando para ti.

Programa este reporte los lunes a las 8 AM con cron y tendrás visibilidad semanal sin esfuerzo:

```
0      8      *      *      1      /home/TU_USUARIO/practica-linux/reporte-respaldos.sh      >>
/home/TU_USUARIO/respalos/reporte-semanal.log 2>&1
```

Explicación: un sistema sin visibilidad es un sistema ciego. El reporte semanal te dice de un vistazo que todo funciona, y te avisa si algo falló. Monitoreo básico, profesional y automático.

Errores comunes y casos reales

El proyecto final junta todo el curso en un sistema de respaldo real. Aquí los errores cuestan datos de verdad: un respaldo mal hecho es peor que no tener respaldo, porque te da confianza falsa. Estos son los pecados capitales del backup.

Caso 1: tar con rutas absolutas que al restaurar pisa el sistema

Síntoma: haces ``tar -czf respaldo.tar.gz /home/esteban/datos`` y al restaurar con ``tar -xzf respaldo.tar.gz -C /``... o peor, sin ``-C``, los archivos vuelven a ``/home/esteban/datos`` aplastando lo actual sin preguntar. En el peor caso documentado, alguien restauró un tar con rutas absolutas sobre ``/`` y mezcló dos sistemas.

Por qué ocurre: tar guarda las rutas TAL COMO se las das. Con rutas absolutas, la extracción también es absoluta: no hay "carpeta de prueba" posible, todo cae en su lugar original.

Solución: empaqueta con ``-C`` (cambia de directorio primero) y rutas relativas:

```
# BIEN: dentro del tar las rutas son relativas (datos/archivo)
tar -czf respaldo-2026-09-20.tar.gz -C /home/esteban datos

# Restauración segura en carpeta temporal:
mkdir -p /tmp/prueba-restore && tar -xzf respaldo-2026-09-20.tar.gz -C
/tmp/prueba-restore
ls -R /tmp/prueba-restore # verifica ANTES de tocar lo real
```

Bonus: tar moderno quita el ``/`` inicial con un aviso ("Removing leading /"); no confíes en ese aviso como estrategia.

Caso 2: la rotación que borra la copia MÁS NUEVA

Síntoma: tu script "conserva las últimas 7 copias" pero un día descubres que borró la de anoche y dejó la de hace un mes. O borra todas menos una porque el patrón del `find` no coincidió.

Por qué ocurre: ordenar por nombre NO es ordenar por fecha (`respaldo-2026-9-2` vs `respaldo-2026-10-1`: alfabéticamente gana septiembre... mal). Y `ls | head` en scripts es frágil: los nombres con espacios o saltos de línea lo rompen.

Solución: nombres con fecha en formato `YYYY-MM-DD` (orden alfabético = orden cronológico) y borrado por antigüedad con `find -mtime`:

```
# Nombres que sí ordenan bien:
# respaldo-2026-09-20.tar.gz (nunca respaldo-20-09-2026)

# Borra copias de más de 7 días, solo archivos que coincidan:
find /respaldos -maxdepth 1 -name "respaldo-*.tar.gz" -mtime +7 -delete
```

Verifica SIEMPRE con `find -print` antes de cambiar a `-delete`: `find ... -print` te muestra las víctimas antes del crimen.

Caso 3: el respaldo que nadie verificó (el clásico del desastre)

Síntoma: tres años de respaldos nocturnos "exitosos". El día del disco muerto, el primer tar que intentas abrir dice `gzip: stdin: unexpected end of file`. El segundo también. Resulta que el disco de respaldos tenía sectores dañados desde hacía meses y nadie lo notó.

Por qué ocurre: crear el archivo no es respaldar: RESPALDAR es crear + verificar + probar restaurar. Un tar corrupto se ve igual que uno bueno en `ls -lh`.

Solución: verificación en dos niveles, automática:

```
# Nivel 1: el tar se puede listar completo (detecta truncados)
tar -tzf /respaldos/respaldo-2026-09-20.tar.gz > /dev/null      && echo "estructura
ok" || echo "TAR CORRUPTO" >&2

# Nivel 2: checksum para detectar corrupción silenciosa después
sha256sum /respaldos/respaldo-*.tar.gz > /respaldos/CHECKSUMS.txt
sha256sum -c /respaldos/CHECKSUMS.txt
```

Y el nivel 3, el que de verdad importa: restaura en `/tmp` al menos una vez al mes y abre archivos reales.

Caso 4: cron con `%` o sin log que falla en silencio

Síntoma: programas `0 2 \* \* \* /home/esteban/respaldo.sh \$(date +%F)` y el respaldo nunca se crea, o se crea con un nombre rarísimo. El correo de cron (si lo hay) dice algo críptico.

Por qué ocurre: en crontab, `%` significa SALTO DE LÍNEA: cron corta tu comando ahí y manda el resto como stdin. `date +%F` se convierte en `date +` más basura. Además, sin redirección a log, los errores se van al correo local que nadie lee.

Solución: la fecha VA DENTRO del script, nunca en la línea de cron; y todo al log:

```
# EN EL SCRIPT (respaldo.sh):
fecha=$(date +%F)
tar -czf "/respaldos/respaldo-{$fecha}.tar.gz" -C /home/esteban datos

# EN EL CRONTAB (sin %, con log):
0 2 * * * /home/esteban/respaldo.sh >> /home/esteban/logs/respaldo.log 2>&1
```

Si algún día necesitas un `%` literal en cron: escápalo anteponiendo una barra invertida.

Caso 5: respaldar TODO incluyendo basura (cachés, temporales, descargas)

Síntoma: tu "respaldo de documentos" pesa 80 GB porque incluyó `~/cache` (12 GB de miniaturas), `~/Descargas` con ISOs viejas y `node\_modules` de tres proyectos (otros 9 GB). Tarda horas, llena el destino y la rotación borra copias útiles para guardar basura.

Por qué ocurre: `tar -czf todo.tar.gz /home/esteban` no discrimina. El 90% del peso de un home típico es regenerable o prescindible.

Solución: `exclude` con patrones, y respalda por SEPARADO lo valioso:

```
tar -czf datos-2026-09-20.tar.gz --exclude='.cache' --exclude='Descargas'
--exclude='node_modules' --exclude='*.tmp' -C /home/esteban documentos
proyectos fotos
```

Documenta qué excluyes y por qué en tu LEEME: el día de la restauración nadie debe preguntarse "¿dónde están mis descargas?".

Caso 6: respaldos legibles por todo el mundo (con tus datos privados)

Síntoma: tus respaldos en el disco externo o en el servidor tienen permiso 644 y pertenecen a tu usuario... en una máquina compartida, cualquiera los lee: documentos, fotos, el `notas.txt` con contraseñas (no lo hagas, pero la gente lo hace).

Por qué ocurre: tar conserva los permisos originales, y la umask default suele crear archivos legibles por el grupo/otros. Nadie piensa en permisos hasta que alguien curioso hace `ls` en el disco de respaldos.

Solución: ajusta permisos al crear y considera cifrado para lo sensible:

```
tar -czf /respaldos/datos-2026-09-20.tar.gz -C /home/esteban documentos
chmod 600 /respaldos/datos-2026-09-20.tar.gz # solo tú lo lees

# Para lo realmente sensible: cifra con gpg (te pedirá contraseña)
tar -czf - -C /home/esteban documentos | gpg --symmetric --cipher-algo AES256 -o
datos-2026-09-20.tar.gz.gpg
```

Un respaldo sin control de acceso es una filtración esperando turno.

Caso real 1: el "respaldo" que era un enlace simbólico

En una oficina, el script hacía `cp -r /datos /respaldos/datos` y el log decía éxito todas las noches. Cuando el RAID de `/datos` murió, descubrieron que `/respaldos/datos` era un ENLACE SIMBÓLICO a `/datos` (alguien lo creó "temporalmente" hacía un año para ahorrar espacio durante una migración y se le olvidó revertirlo). El `cp -r` copiaba los enlaces, no los datos. Perdieron 6 meses de facturación.

Las lecciones, en código:

```
# 1. Verifica que el destino NO sea un enlace antes de respaldar:
[ -L "/respaldos/datos" ] && { echo "PELIGRO: es un enlace simbólico" >&2; exit 1;
}

# 2. Verifica el tamaño: un respaldo de 0 bytes o idéntico al anterior es
sospechoso
tam=$(stat -c%s "/respaldos/datos-2026-09-20.tar.gz")
[ "$tam" -gt 1000 ] || { echo "respaldo sospechosamente pequeño: $tam bytes" >&2;
exit 1; }

# 3. tar -h sigue enlaces y guarda el CONTENIDO (cuando eso es lo que quieres):
tar -czhf respaldo.tar.gz -C /home/esteban datos
```

Moraleja: el script debe desconfiar de su propio entorno: verifica destino, tamaño y tipo antes de declarar éxito.

Caso real 2: el ransomware y las copias que sí sobrevivieron

Una pyme fue cifrada por ransomware un lunes a las 9 am: todos los archivos del servidor, terminación `.locked`. Tenían "respaldos" en un disco USB... conectado al mismo servidor: también cifrado. Lo que los salvó: un script humilde como el de este curso que cada viernes copiaba el tar semanal a un disco que SOLO se conectaba para eso (air gap manual) más una copia mensual en la nube con `scp`. Perdieron 3 días de trabajo, no 4 años.

La regla 3-2-1 que todo profesional recita de memoria:

```
# 3 copias, 2 medios distintos, 1 fuera del sitio:
# copia 1: /respaldos/ local (rotación diaria, 7 días)
# copia 2: disco externo (semanal, se desconecta después)
# copia 3: servidor remoto (mensual)
scp /respaldos/semanal-*.tar.gz usuario@remoto:/respaldos/
```

Tu script de esta unidad implementa la copia 1. El reto final te lleva a la 2 y la 3. Un respaldo que vive junto a los datos originales es un adorno.

Miniproyecto guiado de la unidad

Vas a construir el sistema completo: `respaldo.sh` (con funciones, getopt, verificación y rotación), su programación en cron y su `LEEME.md` de documentación. Es la suma de las 10 unidades: variables, lógica, funciones, modo estricto, procesos, cron y texto, todo trabajando junto.

Paso 1: configuración y funciones base

```
#!/bin/bash
# respaldo.sh - sistema de respaldo con rotación y verificación
# Uso: respaldo.sh [-v] [-n dias] [-h] [origen...]
set -euo pipefail

DESTINO="$HOME/practica-linux/respaldos"
LOG="$HOME/practica-linux/logs/respaldo.log"
DIAS=7
VERBOSE=0

log() { echo "[$(date '+%F %T')] $" >> "$LOG"; }
info() { log "INFO: $*"; [ "$VERBOSE" -eq 1 ] && echo "$*"; }
die() { log "ERROR: $*"; echo "ERROR: $" >&2; exit 1; }

uso() {
    cat <<'AYUDA'
Uso: respaldo.sh [-v] [-n dias] [-h] [origen...]
    -n dias    Conserva respaldos de los últimos N días (default: 7)
    -v         Muestra en pantalla además del log
    -h         Esta ayuda
    origen     Carpetas a respaldar (default: documentos)
AYUDA
}

```

El log con marca de tiempo es tu caja negra: cuando algo falle a las 2 am, el log cuenta la historia.

Paso 2: parseo de opciones y validaciones previas

```

while getopts "vhn:" opt; do
    case "$opt" in
        v) VERBOSE=1 ;;
        h) uso; exit 0 ;;
        n) DIAS="$OPTARG" ;;
        *) uso >&2; exit 1 ;;
    esac
done
shift $((OPTIND - 1))

[[ "$DIAS" =~ ^[0-9]+$ ]] || die "-n necesita un número de días"
[ "$DIAS" -ge 1 ] || die "-n necesita al menos 1 día"

ORIGENES=("$@")
[ ${#ORIGENES[@]} -eq 0 ] && ORIGENES=("documentos")

mkdir -p "$DESTINO" || die "no pude crear $DESTINO"
[ -L "$DESTINO" ] && die "$DESTINO es un enlace simbólico, aborto por seguridad"

for o in "${ORIGENES[@]}"; do
    [ -e "$HOME/$o" ] || die "no existe $HOME/$o"
done
info "Inicio de respaldo. Orígenes: ${ORIGENES[*]}, retención: $DIAS días"

```

Validación antes de actuar (Unidad 7), `local` implícito en funciones (Unidad 8), protección contra el enlace simbólico del Caso real 1. Todo el curso en 15 líneas.

Paso 3: crear, verificar y rotar

```

fecha=$(date +%F)
archivo="$DESTINO/respaldo-${fecha}.tar.gz"

info "Creando $archivo ..."
tar -czf "$archivo" --exclude='.cache' --exclude='node_modules'
--exclude='*.tmp' -C "$HOME" "${ORIGENES[@]}" || die "falló tar"

tam=$(stat -c%s "$archivo")
[ "$tam" -gt 1000 ] || die "respaldo sospechosamente pequeño ($tam bytes)"
info "Creado: $archivo ($(tam / 1024)) KB"

info "Verificando estructura del tar ..."
tar -tzf "$archivo" > /dev/null || die "el tar está corrupto"
info "Verificación ok"

info "Generando checksum ..."
sha256sum "$archivo" > "${archivo}.sha256"

info "Rotando: borrando copias de más de $DIAS días ..."
find "$DESTINO" -maxdepth 1 -name "respaldo-*.tar.gz" -mtime +"$DIAS" -print
-delete >> "$LOG" 2>&1
info "Respaldo completado con éxito"

```

Observa el orden sagrado: crear -> verificar TAMAÑO -> verificar ESTRUCTURA -> checksum -> rotar. La rotación va AL FINAL: nunca borres lo viejo antes de confirmar que lo nuevo sirve.

Paso 4: primera corrida manual y revisión del log

```

mkdir -p ~/practica-linux/documentos ~/practica-linux/logs
echo "contrato importante" > ~/practica-linux/documentos/contrato.txt
chmod u+x ~/practica-linux/respaldo.sh
~/practica-linux/respaldo.sh -v
echo "código: $?"
ls -lh ~/practica-linux/respaldos/
cat ~/practica-linux/logs/respaldo.log

```

Verificación: código 0, el `tar.gz` y su `sha256` en respaldos, y el log narrando cada paso con hora. Si algo falla, el log te dice exactamente dónde.

Paso 5: prueba de restauración (el paso que casi nadie hace)

```
mkdir -p /tmp/restore-test
tar -xzf ~/practica-linux/respaldos/respaldo-*.tar.gz -C /tmp/restore-test
find /tmp/restore-test -type f
cat /tmp/restore-test/documentos/contrato.txt
sha256sum -c ~/practica-linux/respaldos/respaldo-*.sha256
```

Verificación: `contrato.txt` dice "contrato importante" y el checksum dice `OK`. Acabas de probar tu sistema de punta a punta: esto es lo que separa un respaldo real de una esperanza.

Paso 6: programa en cron y documenta

```
crontab -e
# Agrega:
0 2 * * * /home/esteban/practica-linux/respaldo.sh >>
/home/esteban/practica-linux/logs/respaldo.log 2>&1
```

Y crea el LEEME.md que documenta el sistema (un sistema sin documentación es un misterio para tu yo del futuro):

```
cat > ~/practica-linux/respaldos/LEEME.md <<'FIN'
# Sistema de respaldo

- Qué: carpetas `documentos` (+ las que se pasen como argumento).
- Dónde: `~/practica-linux/respaldos/respaldo-YYYY-MM-DD.tar.gz`
- Cuándo: diario a las 2:00 am vía cron (`crontab -l` para ver).
- Retención: 7 días (`-n` para cambiar).
- Verificación: tamaño + `tar -tzf` + sha256 en cada corrida.
- Log: `~/practica-linux/logs/respaldo.log`

## Cómo restaurar

1. Elige la copia: `ls -lh ~/practica-linux/respaldos/`
2. Verifica: `sha256sum -c respaldo-FECHA.tar.gz.sha256`
3. Prueba en temporal: `tar -xzf respaldo-FECHA.tar.gz -C /tmp/restore`
4. Restaura lo necesario copiando desde /tmp/restore a su lugar.
FIN
```

Reto final (sin solución)

1. Agrega la opción `-r` (remoto): después de un respaldo exitoso, copia el tar y su sha256 a un servidor con `scp` (destino configurable en variable `REMOTO="usuario@servidor:/respaldos/"`). Si el scp falla, el script debe registrarlo pero NO marcar el respaldo local como fallido (el local ya está verificado).
2. Agrega una función `resumen` y la opción `--resumen` (parseo manual, porque getopt no maneja opciones largas): debe imprimir cuántos respaldos hay, cuánto pesan en total (`du -sh`) y la fecha del más reciente y del más viejo.
3. Haz que el script envíe una ALERTA al log (y a stderr) si el espacio libre en el destino es menor al doble del tamaño del último respaldo (pista: `df` + `awk`, Unidad 5 al rescate).

Termínalo y tendrás un sistema de respaldo que muchos profesionistas envidiarían: verificado, rotado, documentado y programado.

Práctica

Estos ejercicios integran todo el curso. Tómalos como tu examen práctico antes del examen final.

Ejercicio 1: instala tu sistema

Enunciado: deja tu sistema de respaldo instalado y funcionando: script en su lugar definitivo, probado con tus documentos reales de práctica, programado en cron para las 2 AM, y documentado en un LEEME.

Solución paso a paso. Paso 1: copia `respaldo.sh` a `~/bin/respaldo.sh` (crea `~/bin` si no existe) o déjalo en tu zona de práctica con ruta absoluta clara. Paso 2: ejecútalo a mano con tus documentos reales: `respaldo.sh -v ~/practica-linux/documentos` y verifica el `.tar.gz` y el registro. Paso 3: programa cron: `crontab -e` y la línea `0 2 * * * /home/TU_USUARIO/bin/respaldo.sh /home/TU_USUARIO/practica-linux/documentos >> /home/TU_USUARIO/respaldos/cron.log 2>&1` (ajusta rutas). Paso 4: verifica con `crontab -l`. Paso 5: escribe `LEEME-respaldo.txt` con: qué respalda, dónde guarda, horario de cron (copia la línea), cuántas copias, dónde está el log y cómo restaurar (los comandos del Ejemplo 3). Paso 6: este ejercicio no tiene solución única: la tienes tú, con tu sistema funcionando. Cuando termines, tendrás algo que pocos logran: automatización real instalada.

Ejercicio 2: respaldo con exclusiones

Enunciado: modifica ``respaldo.sh`` (o crea ``respaldo2.sh``) para que acepte un archivo de exclusiones: un archivo de texto con patrones (uno por línea) de lo que NO se debe respaldar, usando la opción ``--exclude-from`` de tar.

Solución paso a paso. Paso 1: crea ``exclusiones.txt`` con patrones como ``*.tmp``, ``*.log`` y ``cache/``. Paso 2: agrega la opción ``-e`` al `getopts`: ``e` EXCLUIR="$OPTARG"`. Paso 3: valida que el archivo de exclusiones exista si se pasó la opción. Paso 4: en el comando tar, agrega condicionalmente: ``excl=()`` y ``if [-n "${EXCLUIR:-}"]; then excl=(--exclude-from="$EXCLUIR"); fi``, luego ``tar -czf "$archivo" "${excl[@]}" "${lista[@]}"``. El arreglo ``excl`` vacío no agrega nada; con archivo, agrega la opción. Paso 5: prueba con archivos ``.tmp`` en tus documentos: deben quedar fuera del respaldo. Verifica con ``tar -tzf`` listando el contenido. Paso 6: las exclusiones son esenciales en respaldos reales: nadie quiere respaldar cachés, temporales ni archivos de 10 GB que se regeneran solos.

Ejercicio 3: alerta por correo (simulada)

Enunciado: extiende el sistema para que, si un respaldo falla, escriba una alerta en un archivo ``ALERTAS.txt`` con fecha y motivo. Simula un fallo (origen inexistente con `-n 1...` mejor: un destino sin permiso de escritura) y verifica la alerta.

Solución explicada paso a paso. Paso 1: en tu script, donde hay ``exit 1`` por fallos, antes agrega: ``echo "[$(date)] ALERTA: <motivo>" >> "$DESTINO/ALERTAS.txt"``. Centralízalo en la función ``error`` para no repetir código: la función ``error`` ya escribe al log; agrégale que también agregue a `ALERTAS.txt`. Paso 2: simula el fallo: ejecuta el respaldo contra un origen que no existe: ``./respaldo.sh /no/existe``: debe omitirlo, y si no hay orígenes válidos, fallar con alerta. Paso 3: verifica ``cat ~/respaldos/ALERTAS.txt``: la alerta está registrada. Paso 4: en producción, esa alerta se enviaría por correo con el comando ``mail``; el mecanismo es el mismo: detectar el fallo y notificar. Un sistema que falla en silencio no es un sistema: es una ilusión.

Ejercicio 4: inventario completo del curso

Enunciado: sin ver tus apuntes, escribe de memoria la lista de los 30 comandos o conceptos más importantes del curso (3 por unidad) con una línea de descripción cada uno. Luego compárala con tu cheat sheet y califícate: ¿cuántos te faltaron?

Solución paso a paso. Paso 1: en papel o archivo, escribe 3 por unidad sin mirar: por ejemplo U1: `pwd`, `ls -lh`, `man`; U2: `cp -r`, `rm -i`, `find`; U3: `chmod 755`, `sudo`, `ls -l`; U4: `>`, `|`, `tee`; U5: `grep -i`,

sed s///g, awk '{print \$1}'; U6: \$(), \$1/\$@, exit; U7: if[[]], for, while; U8: set -euo pipefail, getopts, bash -x; U9: ps aux, crontab -e, ssh. Paso 2: compara con tu cheat sheet y marca los que olvidaste. Paso 3: los olvidados son tu lista de repaso: vuelve a sus unidades y practícalos una vez más. Paso 4: si recordaste 25 o más, tu base es sólida; si menos de 20, repasa las unidades flojas antes del examen. Este ejercicio mide tu memoria activa, que es lo que cuenta cuando estás frente a una terminal real sin apuntes.

Ejercicio 5: tu próximo paso

Enunciado: elige un problema real de tu vida digital que podrías automatizar con lo aprendido (organizar descargas, limpiar archivos viejos, vigilar un sitio, respaldar algo) y escribe el diseño en papel: qué hace, cada cuánto corre, qué funciones tendría y qué comandos usaría. No lo programes todavía: solo diseñalo.

Solución paso a paso. Paso 1: elige el problema más molesto y repetitivo que tengas: ese es tu candidato. Paso 2: escribe el objetivo en una frase: "cada domingo, mover mis descargas a carpetas por tipo y borrar los .tmp de más de 30 días". Paso 3: lista las funciones: `clasificar()`, `limpiar\_viejos()`, `reportar()`. Paso 4: lista los comandos por función: clasificar usa for+case+mv (Unidad 7, como el Ejemplo 2); limpiar usa find -mtime +30 -delete; reportar usa log + tee. Paso 5: define el cron: `0 10 \* \* 0` (domingos 10 AM). Paso 6: guarda este diseño: es tu primer proyecto post-curso. La diferencia entre el que aprende y el profesional es que el profesional automatiza sus propios problemas. Ya tienes las herramientas: úsalas.

Tarjeta de referencia rápida

| Comando / sintaxis | Descripción | |---|---| | `tar -czf f.tar.gz -C dir origen` | Crea comprimido con gzip; `-C` + ruta relativa = restauración segura. | | `tar -tzf f.tar.gz` | Lista el contenido SIN extraer: verificación rápida. | | `tar -xzf f.tar.gz -C /tmp/x` | Extrae en una carpeta temporal para probar antes. | | `tar -tvf f.tar.gz` | Lista verbosa con permisos, tamaños y fechas. | | `tar --exclude='patron'` | Excluye archivos/carpetas (cachés, temporales). | | `date +%F` | Fecha `YYYY-MM-DD`: ordena alfabética y cronológicamente. | | `sha256sum archivo` | Huella digital del archivo; detecta corrupción silenciosa. | | `sha256sum -c suma.sha256` | Verifica el archivo contra su checksum guardado. | | `find dir -name "pat" -mtime +N -delete` | Borra archivos con más de N días (rotación). ¡Prueba con `-print` primero! | | `stat -c%s archivo` | Tamaño en bytes: para detectar respaldos sospechosamente pequeños. | | `du -sh dir` | Cuánto pesa una carpeta, en formato legible. | | `df -h` | Espacio libre por disco: verifica ANTES de respaldar. | | `gzip` / `gunzip` | Comprime/descomprime rápido (el `-z` de tar). | | `gpg --symmetric -o f.gpg f` | Cifra con contraseña: para respaldos sensibles. | | `scp f.tar.gz usuario@host:ruta` | Copia el

respaldo fuera del sitio (regla 3-2-1). | | `0 2 \* \* \* /ruta/script.sh` | Cron diario a las 2 am, con ruta absoluta y log. | | `[ -L ruta ]` | Detecta enlaces simbólicos: protégelos antes de respaldar. | | `chmod 600 respaldo.tar.gz` | Solo tú lees tus respaldos. |

5 one-liners que vas a usar toda la vida

```
tar -czf respaldo-$(date +%F).tar.gz -C /home/esteban documentos
```

El respaldo diario en una línea: fecha ordenable, rutas relativas, listo para cron.

```
tar -tzf respaldo-2026-09-20.tar.gz > /dev/null && echo "íntegro" || echo "CORRUPTO"
```

Verificación exprés: si el tar no se puede listar completo, no sirve.

```
find /respaldos -maxdepth 1 -name "respaldo-*.tar.gz" -mtime +7 -print
```

Muestra qué borraría la rotación; cambia `-print` por `-delete` cuando estés seguro.

```
sha256sum /respaldos/*.tar.gz > /respaldos/CHECKSUMS.txt && sha256sum -c /respaldos/CHECKSUMS.txt
```

Genera y verifica huellas de todas tus copias en dos pasos.

```
du -sh /respaldos/* | sort -h | tail -5
```

Tus 5 respaldos más pesados: el primer sospechoso cuando el disco se llena.

Más comandos que completan la unidad

| Comando / sintaxis | Descripción | |---|---| | `tar -cjf f.tar.bz2 ...` | Compresión bzip2: más lento que gzip pero archivos más pequeños. | | `tar -cJf f.tar.xz ...` | Compresión xz: la más compacta (ideal para archivar a largo plazo). | | `tar --listed-incremental=snap` | Respaldo INCREMENTAL: solo lo cambiado desde la última vez (nivel profesional). | | `tar --newer="2026-09-01"` | Solo archivos modificados después de esa fecha: incremental manual. | | `rsync -a --delete o/ d/` | Espejo exacto: copia solo cambios y borra en destino lo que ya no existe. | | `du --max-depth=1 -h dir` | Cuánto pesa cada subcarpeta: encuentra al culpable del disco lleno. | | `find dir -size +100M` | Archivos de más de 100 MB: candidatos a excluir del respaldo diario. | | `split -b 500M f.tar.gz` | Parte un respaldo gigante en trozos de 500 MB (`cat

x*` los reúne). | | `md5sum` / `sha1sum` | Checksums rápidos (sha256 es el estándar serio; md5 solo para detección casual). | | `stat archivo` | Metadatos completos: tamaño, permisos, fechas de acceso/modificación/cambio. | | `touch -d "2026-01-01" f` | Cambia la fecha de un archivo: útil para probar la rotación por `mtime`. | | `logrotate` | El servicio del sistema que rota logs automáticamente: inspira tu rotación. | | `ionice -c3 tar ...` | Respalda con prioridad de disco mínima: no satura el servidor en horario laboral. |

3 combinaciones que valen oro

```
tar -cjf - -C /home/esteban documentos | split -b 500M - /respaldos/docs-2026-09-20.tar.bz2.
```

Respaldo comprimido partido en trozos de 500 MB en una sola tubería: cabe en DVDs, FAT32 o subidas con límite de tamaño.

```
ionice -c3 nice -n 10 tar -czf /respaldos/d.tar.gz -C /home/esteban documentos
```

El respaldo "invisible": prioridad mínima de CPU y disco para no afectar a los usuarios mientras corre. Cortesía de sysadmin.

```
tar -czf /respaldos/inc-$(date +%F).tar.gz --listed-incremental=/respaldos/snap.dat -C /home/esteban documentos
```

Respaldo incremental real: el archivo `snap.dat` recuerda el estado anterior y solo empaqueta lo nuevo o modificado. Diario incremental + semanal completo = estrategia profesional completa.

Resumen de la unidad

- Un buen respaldo responde: qué, dónde, cuántas copias (rotación), cómo se comprime y cómo se verifica.
- tar -czf crea el archivo comprimido; tar -tzf lo verifica sin extraer; tar -xzf lo restaura.
- La rotación mantiene N copias y borra las viejas: espacio constante para siempre.
- Un respaldo no probado es una esperanza: restaura en una carpeta temporal al menos una vez.
- cron ejecuta tu script solo: prueba a mano primero, usa rutas absolutas y registra todo.

- Documenta tu sistema: qué, dónde, cuándo, cuántas copias y cómo restaurar, en un LEEME.
- Unidad 1: orientación (pwd, ls, cd, man, Tab). Unidad 2: archivos (cp, mv, rm, find). Unidad 3: permisos (chmod, sudo).
- Unidad 4: flujos (>, >>, |, tee). Unidad 5: texto (grep, sed, awk). Unidad 6: scripts (variables, argumentos, exit).
- Unidad 7: lógica (if, for, while, case). Unidad 8: profesionalismo (funciones, set -euo pipefail, getopts).
- Unidad 9: operación (ps, kill, cron, SSH). Todo unido en un sistema de respaldo real.
- Ya dominas Linux y Shell Scripting: automatiza tus problemas reales y sigue con Hacking Ético o Networking en lucasegaacademy.com.
- Regla 3-2-1: 3 copias, en 2 medios distintos, 1 fuera del sitio; lo que vive junto a los datos no es respaldo.
- ``-C`` + rutas relativas: el tar debe extraerse en cualquier carpeta temporal sin peligro de pisar el sistema.
- ``--exclude`` saca cachés y temporales: respalda lo irreemplazable, no la basura regenerable.
- Verifica en capas: tamaño mínimo, ``tar -tzf`` (estructura), sha256 (integridad a futuro) y restauración de prueba mensual.
- La rotación va DESPUÉS de verificar lo nuevo: nunca borres lo viejo antes de confirmar que lo nuevo sirve.
- ``find -mtime +N -delete`` rota por antigüedad real, no por orden alfabético de nombres (que engaña).
- ``date +%F`` en los nombres: formato YYYY-MM-DD que ordena bien en ``ls`` y en scripts.
- Desconfía de tu entorno: verifica que el destino no sea un enlace simbólico y que el tamaño tenga sentido antes de declarar éxito.
- ``chmod 600`` (y gpg para lo sensible): un respaldo legible por todos es una filtración en espera.

EXAMEN FINAL

Curso Completo de Linux y Shell Scripting - Lucas EGA Academy (lucasegaacademy.com)

Instrucciones

Bienvenido a tu examen final. Si llegaste hasta aquí es porque ya viste las 30 clases del curso, así que este examen es tu oportunidad de demostrar lo que aprendiste y de detectar qué temas necesitas repasar. Tómallo en serio, pero sin miedo: nadie nació sabiendo Linux y cada error que encuentres aquí es una lección que ya no se te va a olvidar.

Cómo presentar el examen:

- Busca un lugar tranquilo donde puedas concentrarte sin interrupciones.
- Ten a la mano una terminal abierta para las preguntas prácticas: algunas piden ejecutar comandos.
- No uses tus apuntes, el libro ni internet durante el examen. La idea es medir lo que realmente se te quedó en la cabeza.
- Lee cada pregunta completa antes de responder. Muchas se pierden por contestar a la carrera.
- En las preguntas de comandos, escribe el comando completo aunque creas que es obvio.
- En las preguntas abiertas de explicación, usa tus propias palabras. No se trata de memorizar definiciones, se trata de entender.

Reglas del examen:

- Tiempo sugerido: 90 minutos. Si terminas antes, usa el tiempo extra para revisar tus respuestas.
- El examen tiene 50 preguntas divididas en tres tipos: 20 de opción múltiple, 15 de verdadero o falso y 15 abiertas.
- En las preguntas de verdadero o falso debes escribir V o F y justificar tu respuesta en una sola línea. Sin justificación, la respuesta no cuenta aunque la letra sea correcta.
- En las preguntas abiertas de comandos debes escribir el comando exacto. En las de explicación, cada respuesta debe indicar los puntos que se evalúan.
- No te quedes atorado en una pregunta difícil. Márcala, sigue con las demás y regresa al final si te sobra tiempo.

Sistema de puntaje:

- Cada pregunta vale 2 puntos. No hay preguntas que valgan más que otras.
- 50 preguntas por 2 puntos dan un total de 100 puntos.
- La calificación mínima aprobatoria es 60 puntos.
- Las preguntas de opción múltiple y de verdadero o falso se califican como correctas o incorrectas, sin puntos parciales.
- Las preguntas abiertas se califican con criterio: 2 puntos si la respuesta está completa y correcta, 1 punto si está parcialmente correcta o le faltan elementos importantes, 0 puntos si está mal o vacía.

Tabla de calificaciones:

- 90 a 100 puntos: Excelente. Dominas Linux y Shell Scripting de verdad. Estás listo para Hacking Ético o Networking.
- 80 a 89 puntos: Muy bien. Tienes una base sólida, repasa los temas donde fallaste y sigue adelante.
- 70 a 79 puntos: Bien. Aprobaste con margen, pero hay temas que todavía no dominas. Repásalos antes de continuar.
- 60 a 69 puntos: Suficiente. Pasaste de panzazo. Te recomiendo repasar el curso completo una vez más antes de continuar.
- 0 a 59 puntos: No aprobado. No te desanimes: vuelve a las clases, practica con el proyecto de respaldo y presenta el examen de nuevo.

Rúbrica detallada para calificar las preguntas abiertas:

- Preguntas de comandos (36, 38, 39, 41, 43, 45, 46, 47, 49 y 50): 2 puntos si el comando es correcto y completo, con las opciones adecuadas; 1 punto si la idea es correcta pero faltan opciones o hay errores menores; 0 puntos si el comando no resuelve lo pedido o está vacío.
- Preguntas de explicación (37, 40, 42, 44 y 48): 2 puntos si la respuesta incluye todos los puntos clave indicados en la solución, explicados con tus propias palabras; 1 punto si incluye la idea central pero le faltan puntos clave; 0 puntos si la explicación es incorrecta o no responde lo preguntado.

Mapa del examen: cada pregunta y la clase del curso que evalúa:

- Preguntas 1 y 21: Clase 1 (Qué es Linux, terminal, shell y prompt)

- Preguntas 2 y 22: Clase 2 (Navegación: pwd, ls, cd y rutas)
- Preguntas 3 y 23: Clase 3 (Ayuda: man, --help, Tab e historial)
- Preguntas 4 y 24: Clase 4 (Crear y copiar: mkdir, touch, cp)
- Preguntas 5 y 25: Clase 5 (Mover, renombrar y borrar: mv, rm)
- Preguntas 6 y 26: Clase 6 (Ver contenido y comodines: cat, less, *, ?)
- Preguntas 7 y 27: Clase 7 (Leer permisos y chmod)
- Preguntas 8 y 28: Clase 8 (Usuarios, grupos y sudo)
- Preguntas 9 y 29: Clase 9 (chown, chgrp y umask)
- Preguntas 10 y 30: Clase 10 (Flujos y redirección)
- Preguntas 11 y 31: Clase 11 (Pipes y filtros básicos)
- Preguntas 12 y 32: Clase 12 (sort, uniq, cut, tr, wc, tee)
- Preguntas 13 y 33: Clase 13 (grep y opciones)
- Preguntas 14 y 34: Clase 14 (Expresiones regulares)
- Preguntas 15 y 35: Clase 15 (sed y awk)
- Pregunta 16: Clase 16 (Variables y comillas)
- Pregunta 17: Clase 17 (Argumentos y códigos de salida)
- Pregunta 18: Clase 18 (read y shebang)
- Pregunta 19: Clase 19 (test y comparaciones)
- Pregunta 20: Clase 20 (if y case)
- Preguntas 36 y 41: Clase 21 (Bucles for y while)
- Pregunta 37: Clase 22 (Funciones)
- Pregunta 38: Clase 23 (Modo estricto y getopts)
- Pregunta 39: Clase 24 (Depuración)
- Pregunta 40: Clase 25 (Procesos)
- Pregunta 42: Clase 26 (cron)
- Pregunta 43: Clase 27 (SSH y redes)
- Preguntas 44, 45 y 50: Clases 28, 29 y 30 (Proyecto: sistema de respaldo)

Usa este mapa para repasar: si fallaste varias preguntas de la misma clase, ya sabes exactamente qué parte del curso volver a estudiar.

Consejos para aprovechar tu resultado:

- No te califiques con coraje ni con lástima: califícate con honestidad. El examen es una herramienta de diagnóstico, no un juicio sobre tu capacidad.
- Marca las preguntas que fallaste y escribe a un lado el número de clase correspondiente. En una semana vuelve a presentar solo esas preguntas y mide tu avance.
- Si aprobaste con más de 90 puntos, tu siguiente paso natural es el curso de Hacking Ético en lucasegaacademy.com: ahí usarás todo lo que hoy dominas para seguridad ofensiva.
- Si no aprobaste, no es el fin del mundo: es información valiosa. Repasa las clases marcadas, reconstruye el proyecto de respaldo con calma y vuelve a intentarlo. La constancia le gana al talento, siempre.

Recomendación final: cuando termines, califica tu examen con la sección de respuestas y soluciones. No te limites a contar aciertos: lee cada explicación, sobre todo la de las preguntas que fallaste. Ahí está el verdadero aprendizaje. El que repasa sus errores avanza el doble de rápido.

¡Mucho éxito! Demuestra de qué estás hecho.

Preguntas

PREGUNTA 1:

¿Qué es el shell en Linux? A) La ventana gráfica donde se escriben comandos B) El programa que interpreta los comandos que escribes y los ejecuta C) El fondo de pantalla del sistema operativo D) Un antivirus incluido en todas las distribuciones

PREGUNTA 2:

Estás en `/home/ana/documentos` y ejecutas ``cd ..`` seguido de ``pwd``. ¿Qué muestra `pwd`? A) `/home/ana/documentos` B) `/home/ana` C) `/home` D) `/`

PREGUNTA 3:

¿Para qué sirve presionar la tecla Tab en la terminal? A) Para cerrar la terminal rápidamente B) Para autocompletar comandos y rutas, evitando errores de escritura C) Para cambiar de usuario D) Para borrar el comando actual

PREGUNTA 4:

¿Qué comando crea la estructura completa proyectos/web/imagenes de un solo golpe, aunque no exista nada? A) mkdir proyectos/web/imagenes B) mkdir -p proyectos/web/imagenes C) touch proyectos/web/imagenes D) cp -r proyectos/web/imagenes

PREGUNTA 5:

¿Qué hace `mv notas.txt documentos/`? A) Copia notas.txt a la carpeta documentos B) Mueve notas.txt a la carpeta documentos C) Renombra documentos como notas.txt D) Borra notas.txt y crea documentos

PREGUNTA 6:

¿Qué muestra `tail -f /var/log/syslog`? A) Las primeras 10 líneas del archivo una sola vez B) El archivo completo paginado C) Las últimas líneas y sigue mostrando las nuevas conforme se escriben D) Nada, porque -f no es una opción válida

PREGUNTA 7:

En la cadena `-rwxr-xr--`, ¿qué permisos tiene el grupo? A) Lectura, escritura y ejecución B) Lectura y ejecución C) Solo lectura D) Ninguno

PREGUNTA 8:

¿Qué hace `sudo`? A) Cambia tu contraseña de usuario B) Ejecuta un comando con privilegios de administrador sin vivir como root C) Apaga el sistema inmediatamente D) Muestra los usuarios conectados

PREGUNTA 9:

¿Qué significa el permiso octal 644? A) Dueño todo, grupo y otros solo lectura y ejecución B) Dueño lectura y escritura, grupo y otros solo lectura C) Todos pueden leer, escribir y ejecutar D) Solo el dueño puede leer

PREGUNTA 10:

¿Qué hace `ls /noexiste 2> errores.txt`? A) Guarda la lista del directorio en errores.txt B) Muestra el error en pantalla y crea un archivo vacío C) Manda el mensaje de error al archivo errores.txt sin mostrarlo en pantalla D) Borra el archivo errores.txt

PREGUNTA 11:

¿Qué hace el operador `|` (pipe)? A) Borra la salida del comando anterior B) Conecta la salida de un comando con la entrada del siguiente C) Ejecuta dos comandos al mismo tiempo en paralelo D) Guarda la salida en un archivo

PREGUNTA 12:

¿Qué hace ``sort lista.txt | uniq -c``? A) Ordena y muestra solo las líneas únicas sin contar B) Ordena, cuenta cuántas veces se repite cada línea y muestra el conteo C) Mezcla las líneas aleatoriamente D) Borra las líneas duplicadas del archivo original

PREGUNTA 13:

¿Qué hace ``grep -ri "TODO" ~/proyectos``? A) Busca archivos llamados TODO en la carpeta B) Busca recursivamente el texto "TODO" sin importar mayúsculas en todos los archivos de la carpeta C) Reemplaza TODO por otra palabra en todos los archivos D) Cuenta cuántos archivos hay en la carpeta

PREGUNTA 14:

¿Qué líneas encuentra ``grep "^ERROR" app.log``? A) Todas las que contienen ERROR en cualquier parte B) Solo las que empiezan con ERROR C) Solo las que terminan con ERROR D) Ninguna, porque `^` no es válido en grep

PREGUNTA 15:

¿Qué hace ``sed 's/2024/2025/g' reporte.txt``? A) Modifica el archivo reporte.txt reemplazando 2024 por 2025 B) Muestra el contenido con 2024 reemplazado por 2025, sin modificar el archivo C) Borra todas las líneas que contienen 2024 D) Crea un archivo nuevo llamado 2025

PREGUNTA 16:

¿Por qué falla ``nombre = "Ana"`` en bash? A) Porque Ana debe ir sin comillas B) Porque no debe haber espacios alrededor del ``=`` en una asignación C) Porque nombre es palabra reservada D) Porque falta el comando echo

PREGUNTA 17:

En un script ejecutado como ``./prog.sh uno dos tres``, ¿cuánto vale ``$#``? A) 0 B) 1 C) 3 D) 4

PREGUNTA 18:

¿Qué debe llevar todo script bash en su primera línea? A) ``#`` comentario inicial B) ``#!/bin/bash`` (el shebang) C) ``set -euo pipefail`` D) Nada, la primera línea es opcional

PREGUNTA 19:

¿Qué devuelve ``[ -f "/etc/hostname" ]`` en un sistema normal? A) Siempre 1 porque es un archivo del sistema B) 0 si existe y es un archivo regular, 1 si no C) El contenido del archivo D) Nada, porque `-f` no existe

PREGUNTA 20:

¿Qué hace la estructura ``case`` en bash? A) Repite un bloque mientras una condición sea verdadera B) Elige entre múltiples opciones comparando un valor contra patrones C) Define una función D) Sale del script inmediatamente

PREGUNTA 21:

La terminal, el shell y el prompt son lo mismo con distinto nombre. Justifica en una línea.

PREGUNTA 22:

``cd ~`` y ``cd`` sin argumentos te llevan al mismo lugar. Justifica en una línea.

PREGUNTA 23:

El comando ``man`` sirve para buscar archivos en el sistema. Justifica en una línea.

PREGUNTA 24:

``cp -r`` es necesario para copiar carpetas porque sin ``-r`` el comando falla con directorios. Justifica en una línea.

PREGUNTA 25:

En Linux, renombrar un archivo y moverlo son operaciones distintas que usan comandos distintos. Justifica en una línea.

PREGUNTA 26:

``cat`` es la mejor herramienta para leer archivos de 5000 líneas. Justifica en una línea.

PREGUNTA 27:

El permiso 777 es la solución recomendada cuando un programa no puede escribir por permisos. Justifica en una línea.

PREGUNTA 28:

Trabajar directamente como usuario root todo el día es una práctica segura si eres cuidadoso. Justifica en una línea.

PREGUNTA 29:

``umask 022`` hace que los archivos nuevos nazcan con permiso 644. Justifica en una línea.

PREGUNTA 30:

`>` y `>>` hacen exactamente lo mismo al escribir en archivos. Justifica en una línea.

PREGUNTA 31:

En una tubería como `a | b | c`, cada comando hace una sola cosa y el pipe los conecta en una fábrica de procesamiento. Justifica en una línea.

PREGUNTA 32:

`uniq` elimina duplicados aunque las líneas repetidas no estén juntas. Justifica en una línea.

PREGUNTA 33:

`grep -v` muestra las líneas que NO coinciden con el patrón. Justifica en una línea.

PREGUNTA 34:

En una expresión regular, `^` significa "cualquier carácter". Justifica en una línea.

PREGUNTA 35:

`awk -F: '{print $1}' /etc/passwd` muestra el primer campo de cada línea usando `:` como separador. Justifica en una línea.

PREGUNTA 36:

Escribe un bucle `for` que recorra todos los archivos `.txt` de la carpeta actual e imprima "Procesando: <nombre>" para cada uno.

PREGUNTA 37:

Explica con tus palabras qué hace `set -euo pipefail` y por qué un script profesional lo lleva. Puntos a evaluar: qué hace cada letra (e, u, pipefail) y el beneficio de "fallar rápido y ruidoso".

PREGUNTA 38:

Escribe el esqueleto de un script que use `getopts` para aceptar las opciones `-v` (verbose) y `-h` (ayuda), mostrando el uso con `-h`.

PREGUNTA 39:

Tienes un script que falla misteriosamente en la línea 40. Escribe el comando para ejecutarlo viendo el rastreo de cada comando expandido.

PREGUNTA 40:

Explica la diferencia entre `kill PID` y `kill -9 PID`, y cuándo usar cada uno. Puntos a evaluar: señal TERM vs KILL, terminación ordenada vs forzada, y la regla de intentar primero sin -9.

PREGUNTA 41:

Escribe un bucle `while` que espere a que exista el archivo `/tmp/listo.txt`, reintentando cada 3 segundos, con un máximo de 10 intentos, e informe si se agotó el tiempo.

PREGUNTA 42:

Explica qué significan los 5 campos de tiempo de una línea de crontab y escribe una línea que ejecute `/home/ana/respaldo.sh` todos los días a las 3:30 AM. Puntos a evaluar: orden de los campos, significado del `*` y la línea correcta.

PREGUNTA 43:

Escribe el comando para copiar la carpeta `~/documentos` al servidor `servidor.com` como usuario `ana`, a la ruta `/home/ana/respaldo/`, usando la conexión SSH.

PREGUNTA 44:

Explica el diseño de un sistema de respaldo con rotación: qué preguntas debe responder el diseño y cómo funciona la rotación. Puntos a evaluar: qué/dónde/cuántas copias/verificación, y la lógica de borrar las más viejas manteniendo N.

PREGUNTA 45:

Escribe una tubería completa que, a partir de un archivo `accesos.log` con líneas formato "IP resultado" (ej: "192.168.1.10 FALLO"), muestre las IPs con más fallos ordenadas de mayor a menor, con el conteo.

PREGUNTA 46:

Escribe el comando `find` que busque en `/var/log` los archivos `.log` modificados hace más de 30 días y de más de 100 MB.

PREGUNTA 47:

Escribe el comando para ver los 5 procesos que más CPU consumen, mostrando usuario, PID, %CPU y comando.

PREGUNTA 48:

Explica con tus palabras la diferencia entre `>`, `>>` y `|`, con un ejemplo de uso típico de cada uno. Puntos a evaluar: sobrescribir vs agregar vs conectar comandos, y un ejemplo correcto de cada uno.

PREGUNTA 49:

Escribe el comando ``tar`` para crear un respaldo comprimido ``respaldo-2024-01-15.tar.gz`` de las carpetas ``~/documentos`` y ``~/fotos``, excluyendo los archivos ``*.tmp`` según la lista ``exclusiones.txt``.

PREGUNTA 50:

Diseña (en pseudocódigo o pasos) un script ``vigilante.sh`` que verifique cada ejecución: espacio en disco (alerta si uso mayor al 80%), un sitio web vía curl (alerta si no responde 200) y escriba todo en un registro con fecha, saliendo con código distinto de 0 si algo falló.

Respuestas y soluciones

RESPUESTA 1:

Respuesta correcta: B) El programa que interpreta los comandos que escribes y los ejecuta. La terminal es solo la ventana (el micrófono y la bocina), el prompt es el texto que te invita a escribir ("¿dígame?"), y el shell (normalmente bash) es el cerebro que entiende tus órdenes y las ejecuta. La opción A describe la terminal, no el shell. La opción C no tiene sentido técnico. La opción D es inventada: Linux no trae antivirus integrado de esa forma.

RESPUESTA 2:

Respuesta correcta: B) `/home/ana. `cd ../`` sube un nivel desde `/home/ana/documentos`, quedando en `/home/ana`, y `pwd` confirma la ubicación. La opción A sería el resultado de no moverse. La opción C sería con dos niveles (``cd ../../``). La opción D sería con ``cd /``.

RESPUESTA 3:

Respuesta correcta: B) Para autocompletar comandos y rutas, evitando errores de escritura. Escribes las primeras letras y Tab completa el resto; con dos Tab muestra las opciones si hay varias. No solo ahorra tecleo: evita errores de dedo en nombres largos. Las demás opciones describen funciones que Tab no tiene.

RESPUESTA 4:

Respuesta correcta: B) `mkdir -p proyectos/web/imagenes`. La opción `-p` (parents) crea todos los niveles intermedios que no existan. Sin `-p` (opción A), el comando falla porque `proyectos` no existe. La opción C crea un archivo, no carpetas. La opción D copia, no crea.

RESPUESTA 5:

Respuesta correcta: B) Mueve notas.txt a la carpeta documentos. mv mueve el archivo: desaparece del origen y aparece en el destino. La opción A describe a cp. En Linux no hay comando separado para renombrar: renombrar es mover al mismo lugar con otro nombre.

RESPUESTA 6:

Respuesta correcta: C) Las últimas líneas y sigue mostrando las nuevas conforme se escriben. La opción -f (follow) mantiene el archivo abierto vigilando cambios: es el "electrocardiograma" del sistema que usan los administradores para ver registros en vivo. Para salir se usa Ctrl+C. La opción A describe a `head`, la B a `less`.

RESPUESTA 7:

Respuesta correcta: C) Solo lectura. La cadena se divide en tríos: dueño `rwx`, grupo `r-x`... espera, la cadena es `rwxr-xr--`: dueño rwx, grupo r-x, otros r--. El grupo tiene r-x (lectura y ejecución), que corresponde a la opción B. Respuesta correcta: B) Lectura y ejecución. Releyendo con cuidado: `` tipo archivo, `rwx` dueño, `r-x` grupo, `r--` otros. El grupo puede leer y ejecutar (entrar si fuera directorio), pero no escribir. La opción B es la correcta.

RESPUESTA 8:

Respuesta correcta: B) Ejecuta un comando con privilegios de administrador sin vivir como root. sudo pide tu contraseña y eleva privilegios solo para ese comando: es la llave maestra bajo control. Vivir como root todo el día es peligroso porque no hay red de seguridad ante errores. Las demás opciones no describen a sudo.

RESPUESTA 9:

Respuesta correcta: B) Dueño lectura y escritura, grupo y otros solo lectura. 644 = 6 (4+2, rw-) para el dueño, 4 (r--) para grupo y otros. Es el permiso clásico de archivos de datos. La opción A describe el 755. La opción C describe el 777, que casi nunca es correcto.

RESPUESTA 10:

Respuesta correcta: C) Manda el mensaje de error al archivo errores.txt sin mostrarlo en pantalla. El `2>` redirige solo el canal de errores (stderr, canal 2); la salida normal seguiría yendo a la pantalla. Es la técnica para separar la señal del ruido: resultados limpios por un lado, errores registrados por otro.

RESPUESTA 11:

Respuesta correcta: B) Conecta la salida de un comando con la entrada del siguiente. El pipe es la estrella de Unix: permite encadenar comandos pequeños en fábricas de procesamiento, donde cada uno hace una sola cosa bien. La opción D describe a `>`.

RESPUESTA 12:

Respuesta correcta: B) Ordena, cuenta cuántas veces se repite cada línea y muestra el conteo. `sort` agrupa las iguales (`uniq` solo elimina repetidas consecutivas, por eso va después de `sort`) y `uniq -c` antepone el conteo a cada línea. Es el clásico para estadísticas: top de visitas, errores más frecuentes, etc.

RESPUESTA 13:

Respuesta correcta: B) Busca recursivamente el texto "TODO" sin importar mayúsculas en todos los archivos de la carpeta. La `-r` recorre subcarpetas y la `-i` ignora mayúsculas/minúsculas. Es el combo de "encuentra dónde dejé ese pendiente". La opción C describe a `sed`, no a `grep`.

RESPUESTA 14:

Respuesta correcta: B) Solo las que empiezan con ERROR. El `^` ancla el patrón al inicio de la línea. Sin el `^`, `grep` encontraría ERROR en cualquier posición. El `$` ancla al final. Son los anclajes más usados de las expresiones regulares.

RESPUESTA 15:

Respuesta correcta: B) Muestra el contenido con 2024 reemplazado por 2025, sin modificar el archivo. `sed` trabaja en flujo: transforma y muestra, no toca el original. Para guardar el resultado se redirige a un archivo nuevo (`> nuevo.txt`). Solo la opción `-i` edita en su lugar, y es peligrosa porque no hay deshacer.

RESPUESTA 16:

Respuesta correcta: B) Porque no debe haber espacios alrededor del `=` en una asignación. El shell interpreta `nombre`, `=` y `"Ana"` como tres comandos separados e intenta ejecutar un comando llamado "nombre". Es el error número uno de principiantes en bash. La forma correcta es `nombre="Ana"`.

RESPUESTA 17:

Respuesta correcta: C) 3. `$#` cuenta los argumentos sin incluir `$0` (el nombre del script). Los argumentos son "uno", "dos" y "tres": tres en total. `$0` sería `./prog.sh`, `$1` sería "uno".

RESPUESTA 18:

Respuesta correcta: B) `#!/bin/bash` (el shebang). Le dice al sistema con qué intérprete ejecutar el archivo. Sin ella, el sistema adivina, y adivinar en producción es mala idea. El `set -euo pipefail` (opción C) es recomendable pero va después del shebang.

RESPUESTA 19:

Respuesta correcta: B) 0 si existe y es un archivo regular, 1 si no. test no imprime nada: devuelve el resultado como código de salida (0 verdadero, 1 falso), que se lee con ``$?'` o se usa directo en un ``if``. Es el juez silencioso de bash.

RESPUESTA 20:

Respuesta correcta: B) Elige entre múltiples opciones comparando un valor contra patrones. Cada patrón termina con ``;;``, ``esac`` cierra, y ``*`` atrapa lo demás. Es más limpio que una escalera de elif cuando hay muchas opciones, y acepta comodines en los patrones.

RESPUESTA 21:

F) Son tres cosas distintas: la terminal es la ventana, el shell es el programa que interpreta y ejecuta los comandos, y el prompt es el texto que indica que está listo para recibir órdenes.

RESPUESTA 22:

V) Ambos te llevan a tu carpeta personal: ``~`` es el atajo de la casa y ``cd`` sin argumentos usa ese mismo destino por defecto.

RESPUESTA 23:

F) ``man`` muestra el manual de un comando (su documentación); para buscar archivos se usa ``find`` o ``locate``.

RESPUESTA 24:

V) Sin ``-r`` (recursiva), cp se niega a copiar directorios porque no sabe cómo manejar su contenido; ``-r`` le dice que entre y copie todo lo de adentro.

RESPUESTA 25:

F) En Linux son la misma operación con ``mv``: renombrar es mover el archivo al mismo lugar con un nombre distinto, por eso no existe un comando separado para renombrar.

RESPUESTA 26:

F) ``cat`` vuelca todo el contenido de golpe e inunda la pantalla; para archivos largos se usa ``less`` (navegable, se sale con q) o ``head`/`tail`` para los extremos.

RESPUESTA 27:

F) El 777 le da todos los permisos a todo el mundo y es un agujero de seguridad; lo correcto es dar el permiso mínimo necesario (típicamente 755 para ejecutables y 644 para archivos).

RESPUESTA 28:

F) Como root no hay red de seguridad: cualquier error tipográfico puede destruir el sistema; lo correcto es trabajar como usuario normal y elevar privilegios solo con ``sudo`` para comandos específicos.

RESPUESTA 29:

V) Los archivos nacen con 666 menos la máscara: $666 - 022 = 644$; los directorios nacen con $777 - 022 = 755$.

RESPUESTA 30:

F) ``>`` sobrescribe el archivo (borra lo anterior sin avisar) mientras que ``>>`` agrega al final conservando el contenido existente.

RESPUESTA 31:

V) Esa es la filosofía Unix: herramientas pequeñas que hacen una cosa bien, compuestas con pipes en soluciones potentes sin escribir programas nuevos.

RESPUESTA 32:

F) ``uniq`` solo elimina repetidas consecutivas; por eso el patrón clásico es ``sort | uniq``: primero se agrupan las iguales y luego se eliminan los duplicados.

RESPUESTA 33:

V) La opción ``-v`` invierte la selección: muestra las líneas que no contienen el patrón, útil para quitar ruido (como líneas INFO de un registro).

RESPUESTA 34:

F) ``^`` ancla el patrón al inicio de la línea; "cualquier carácter" es el punto ``.``; ``$`` ancla al final.

RESPUESTA 35:

V) ``-F:`` define ``:`` como separador de campos y ``$1`` es el primer campo: en `/etc/passwd` ese campo es el nombre de usuario.

RESPUESTA 36:

```
for archivo in *.txt; do
    echo "Procesando: $archivo"
done
```

El `for` recorre la expansión del comodín `*.txt` y en cada vuelta `$archivo` vale un nombre distinto. Si no hay txt, conviene agregar `[-e "$archivo"] || continue` como protección.

RESPUESTA 37:

`set -e` detiene el script al primer comando que falle; `set -u` convierte en error fatal el uso de variables no definidas (atrapa typos); `set -o pipefail` hace que una tubería falle si cualquier eslabón falla, no solo el último. Juntos logran "fallar rápido y ruidoso": el script se detiene en el punto exacto del problema con un mensaje claro, en lugar de seguir ejecutándose con datos erróneos y fallar 50 líneas después de forma misteriosa. En automatización (cron) esto es crítico porque nadie está mirando.

RESPUESTA 38:

```
#!/bin/bash
uso() {
    echo "Uso: $0 [-v] [-h]"
    echo "  -v  Modo detallado"
    echo "  -h  Esta ayuda"
}
while getopts "vh" opt; do
    case "$opt" in
        v) echo "Modo verbose activado" ;;
        h) uso; exit 0 ;;
        *) uso >&2; exit 1 ;;
    esac
done
```

`getopts "vh"` declara las banderas; cada vuelta procesa una opción en `$opt`; `-h` muestra el uso y sale; `*)` atrapa opciones inválidas.

RESPUESTA 39:

```
bash -x miscript.sh
```

El modo `-x` imprime cada comando con un `+` antes de ejecutarlo, con las variables ya expandidas: ves la película real de lo que hace el script y detectas dónde el valor real difiere del esperado. Para una sección: `set -x` antes y `set +x` después.

RESPUESTA 40:

`kill PID` envía la señal TERM: le pide al proceso que termine de forma ordenada, dándole oportunidad de guardar su trabajo y liberar recursos. `kill -9 PID` envía KILL: el sistema lo

fulmina de inmediato sin que pueda ignorarla, atraparla ni despedirse, con riesgo de dejar datos inconsistentes. La regla es intentar primero sin `9`, esperar unos segundos y solo usar `9` si el proceso se niega a terminar: es el último recurso, no el primero.

RESPUESTA 41:

```
intento=1
until [ -f /tmp/listo.txt ]; do
    if [ "$intento" -gt 10 ]; then
        echo "Tiempo agotado." >&2
        exit 1
    fi
    echo "Intento $intento/10..."
    sleep 3
    intento=$((intento + 1))
done
echo "¡Archivo encontrado!"
```

`until` repite mientras el archivo no exista; el contador con límite evita la espera eterna; `sleep 3` pausa entre intentos. Es el patrón de espera con timeout.

RESPUESTA 42:

Los campos son: minuto (0-59), hora (0-23), día del mes (1-31), mes (1-12) y día de la semana (0-7). El `\*` significa "todos los valores posibles" de ese campo. La línea:

```
30 3 * * * /home/ana/respaldo.sh
```

Minuto 30, hora 3, todos los días del mes, todos los meses, todos los días de la semana: todos los días a las 3:30 AM. En cron se usan rutas absolutas y se redirige la salida a un registro.

RESPUESTA 43:

```
scp -r ~/documentos ana@servidor.com:/home/ana/respaldo/
```

La `r` copia la carpeta recursivamente; la sintaxis es `origen usuario@servidor:ruta-destino`, igual que `cp` pero a través de SSH. Con llaves SSH configuradas no pide contraseña y se puede automatizar.

RESPUESTA 44:

El diseño responde: qué se respalda (orígenes), dónde se guarda (destino, idealmente otro disco o servidor), cuántas copias se conservan (rotación), cómo se comprime (tar.gz por fecha)

y cómo se verifica (integridad del archivo y registro de cada ejecución). La rotación funciona así: cada respaldo nuevo se suma al historial y si el total supera N, se borran las copias más antiguas (las de fecha menor) hasta quedar en N. Así el espacio usado es constante para siempre y siempre tienes las últimas N copias.

RESPUESTA 45:

```
grep "FALLO" accesos.log | awk '{print $1}' | sort | uniq -c | sort -rn
```

grep filtra las líneas de fallo, awk extrae la IP (campo 1), sort agrupa, `uniq -c` cuenta, `sort -rn` ordena numéricamente de mayor a menor. Es el pipeline clásico de análisis de registros: filtrar, extraer, contar, rankear.

RESPUESTA 46:

```
find /var/log -name "*.log" -mtime +30 -size +100M
```

`-name` filtra por patrón, `-mtime +30` archivos modificados hace más de 30 días, `-size +100M` más de 100 megabytes. Los tres criterios se combinan con Y lógico: solo pasan los que cumplen todo.

RESPUESTA 47:

```
ps aux --sort=-%cpu | head -6
```

`ps aux` lista todos los procesos con usuario, PID, %CPU y comando; `--sort=-%cpu` ordena por CPU descendente; `head -6` toma el encabezado más los 5 primeros. Es el diagnóstico de lentitud en 10 segundos.

RESPUESTA 48:

`>` redirige la salida a un archivo SOBRESCRIBIÉNDOLO (ej: `ls > lista.txt` crea un reporte fresco; cuidado: borra lo anterior). `>>` AGREGA al final sin borrar (ej: `echo "línea" >> diario.txt` para bitácoras que crecen). `|` no toca archivos: conecta la salida de un comando con la entrada del siguiente (ej: `cat datos.txt | grep ERROR` filtra en memoria). En resumen: `>` escribe de nuevo, `>>` continúa escribiendo, `|` procesa en cadena.

RESPUESTA 49:

```
tar -czf respaldo-2024-01-15.tar.gz --exclude-from=exclusiones.txt ~/documentos ~/fotos
```

`-c` crea, `-z` comprime con gzip, `-f` nombra el archivo; `--exclude-from` lee los patrones a excluir del archivo indicado. El orden de las opciones largas puede ir antes de los orígenes sin problema.

RESPUESTA 50:

```
#!/bin/bash
set -euo pipefail
LOG="$HOME/vigilante.log"
fallos=0

fecha=$(date '+%Y-%m-%d %H:%M:%S')

uso=$(df / | awk 'NR==2{print $5}' | tr -d '%')
if [ "$uso" -gt 80 ]; then
    echo "[$fecha] ALERTA: disco al $uso%" | tee -a "$LOG" >&2
    fallos=$((fallos+1))
else
    echo "[$fecha] Disco OK: $uso% usado" >> "$LOG"
fi

codigo=$(curl -s -o /dev/null -w "%{http_code}" --max-time 10
"https://lucasegaacademy.com")
if [ "$codigo" -ne 200 ]; then
    echo "[$fecha] ALERTA: sitio responde $codigo" | tee -a "$LOG" >&2
    fallos=$((fallos+1))
else
    echo "[$fecha] Sitio OK: $codigo" >> "$LOG"
fi

echo "[$fecha] Revisión terminada con $fallos fallo(s)" >> "$LOG"
exit $fallos
```

Cada verificación suma al contador de fallos; las alertas van a stderr y al registro; `exit \$fallos` devuelve 0 si todo bien o el número de fallos si algo falló, para que cron lo detecte. Puntos clave: modo estricto, rutas absolutas (\$HOME), registro con fecha, códigos honestos.

ÍNDICE

PRÓLOGO	4
Para quién es este libro	4
Cómo está organizado este libro	5
Cómo estudiar este libro	6
Qué necesitas para empezar	6
Lo que vas a lograr	6
Unas palabras del autor	7
UNIDAD 1: La terminal y tus primeros comandos	8
Explicación detallada	9
Ejemplos paso a paso	14
Errores comunes y casos reales	18
Miniproyecto guiado de la unidad	22
Práctica	27
Tarjeta de referencia rápida	29
Resumen de la unidad	32
UNIDAD 2: Archivos y directorios	34
Explicación detallada	35
Ejemplos paso a paso	39
Errores comunes y casos reales	42
Miniproyecto guiado de la unidad	46
Práctica	51
Tarjeta de referencia rápida	53
Resumen de la unidad	56
UNIDAD 3: Permisos y propiedad	58
Explicación detallada	58
Ejemplos paso a paso	62
Errores comunes y casos reales	65
Miniproyecto guiado de la unidad	69
Práctica	75
Tarjeta de referencia rápida	77
Resumen de la unidad	80
UNIDAD 4: Redirección, tuberías y filtros	82
Explicación detallada	82
Ejemplos paso a paso	86
Errores comunes y casos reales	89

Miniproyecto guiado de la unidad	94
Práctica	99
Tarjeta de referencia rápida	101
Resumen de la unidad	104
UNIDAD 5: grep, sed y awk	106
Explicación detallada	106
Ejemplos paso a paso	110
Errores comunes y casos reales	113
Miniproyecto guiado de la unidad	117
Práctica	122
Tarjeta de referencia rápida	124
Resumen de la unidad	127
UNIDAD 6: Variables y tu primer script	129
Explicación detallada	129
Ejemplos paso a paso	133
Errores comunes y casos reales	136
Miniproyecto guiado de la unidad	140
Práctica	143
Tarjeta de referencia rápida	145
Resumen de la unidad	147
UNIDAD 7: Condicionales y bucles	149
Explicación detallada	149
Ejemplos paso a paso	154
Errores comunes y casos reales	158
Miniproyecto guiado de la unidad	163
Práctica	166
Tarjeta de referencia rápida	168
Resumen de la unidad	170
UNIDAD 8: Funciones, depuración y buenas prácticas	172
Explicación detallada	172
Ejemplos paso a paso	176
Errores comunes y casos reales	182
Miniproyecto guiado de la unidad	186
Práctica	190
Tarjeta de referencia rápida	192
Resumen de la unidad	194
UNIDAD 9: Procesos, tareas programadas y redes	197

Explicación detallada	197
Ejemplos paso a paso	202
Errores comunes y casos reales	205
Miniproyecto guiado de la unidad	210
Práctica	213
Tarjeta de referencia rápida	215
Resumen de la unidad	217
UNIDAD 10: Proyecto final y graduación	219
Explicación detallada	219
Ejemplos paso a paso	227
Errores comunes y casos reales	230
Miniproyecto guiado de la unidad	234
Práctica	239
Tarjeta de referencia rápida	241
Resumen de la unidad	243
EXAMEN FINAL	245
Instrucciones	245
Preguntas	248
Respuestas y soluciones	254

Curso Completo de Linux y Shell Scripting

De tus primeros comandos en la terminal a la automatización con shell scripting: archivos, permisos, redirección, grep, sed y awk, variables, condicionales y bucles, funciones, procesos y cron. 10 unidades con prácticas y examen final.

Colibrí Editorial EGA

lucasegaacademy.com

Hecho en México