Capítulo 18 — Cadenas a la carta: construí tu propia librería `String_View`

> *Capítulo avanzado, pero **escrito para que se entienda de verdad**. Se apoya en arreglos y cadenas (`06`), punteros (`07`), `struct` (`08`), tipos como `size_t` (`16`), memoria y alineación (`12`), archivos (`15`) y entrada/salida (`14`).*
>
> *Acá vas a dejar de pelear con el terminador `'\0'` y vas a **construir tu propia librería de cadenas**: un conjunto de funciones que miran, cortan, recortan y comparan texto **sin copiarlo** y **sin un solo `malloc`** . Al final vas a poder escribir un parser de archivos de configuración entero con esta caja de herramientas.*

Cómo leer este capítulo

Este es el capítulo donde "sé usar `strlen` " se convierte en "entiendo **cómo** se representa un texto en memoria y puedo diseñar mi propia forma de tratarlo". No es teoría: cada idea viene con **código para mirar**, y cada bloque cierra con **práctica** para que la ejercites con las manos.

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

Parte	De qué habla	La idea en una frase
1	Los tres pilares y la estructura	Un texto es una calle de bytes; en vez de caminarla entera, guardás **dónde empieza** y **cuánto mide**.
2	Sub-vistas	Mirar un tramo del texto **sin copiarlo**: solo movés el inicio y acortás el largo.
3	Recorte (trim)	Sacar espacios de los bordes con solo ajustar el puntero y el contador.
4	Cortar por delimitador (chop / split)	Partir un texto en pedazos, uno por uno, sin reservar memoria.
5	Reemplazo en búfer	Como la vista no es dueña de la memoria, el reemplazo **escribe** en un búfer que vos manejás.
6	El catálogo completo + jefe	Comparar, buscar subcadenas, construir, cortar… y un parser de verdad para cerrar.

Los **ejercicios** se numeran en orden: `18.1`, `18.2`, `18.3`, y así. Cada uno trae su **escenario de prueba** (qué entra y qué tiene que salir, **nunca** el algoritmo) para que compiles y compruebes.

> ***Regla de oro del libro:*** *el capítulo explica y muestra* ***cómo funciona*** *cada cosa (con código ilustrativo), pero* ***los ejercicios no traen la solución****. El enunciado te da el problema; la respuesta la escribís vos.*

Tu librería objetivo

Antes de arrancar, mirá el plano de la casa que vas a construir. Estas son las funciones que vas a tener escritas **vos** al terminar el capítulo. Acá van **solo las declaraciones** (el *contrato*: qué recibe

y qué devuelve cada una), no el cuerpo. Es tu mapa de ruta.

```c
/* ==== sv.h — el contrato de tu librería (solo prototipos) ==== */
#include <stddef.h>   /* size_t  */

typedef struct {
    const char *data;   /* dónde empieza el texto  */
    size_t      count;  /* cuántos bytes tiene     */
} String_View;

/* Impresión */
#define SV_Fmt "%.*s"
#define SV_Arg(sv) (int)(sv).count, (sv).data

/* Construcción */
String_View sv_from_cstr(const char *cstr);
String_View sv_from_parts(const char *data, size_t count);

/* Consultas */
int         sv_index_of(String_View sv, char c);     /* -1 si no está */
size_t      sv_count_char(String_View sv, char c);

/* Sub-vistas (no copian: solo ajustan data y count) */
String_View sv_left(String_View sv, size_t n);
String_View sv_right(String_View sv, size_t n);
String_View sv_chop_left(String_View *sv, size_t n); /* "come" n del frente */
String_View sv_chop_right(String_View *sv, size_t n);

/* Recorte */
void        sv_trim_left(String_View *sv);
void        sv_trim_right(String_View *sv);
void        sv_trim(String_View *sv);

/* Partición */
String_View sv_chop_by_delim(String_View *sv, char delim);

/* Comparación */
int         sv_eq(String_View a, String_View b);
int         sv_starts_with(String_View sv, String_View prefix);
int         sv_ends_with(String_View sv, String_View suffix);

/* Búsqueda de subcadena */
int         sv_find(String_View haystack, String_View needle);   /* -1 si no está */
int         sv_contains(String_View haystack, String_View needle);

/* Reemplazo (escribe en un búfer TUYO, nunca en la vista) */
size_t      sv_replace_char(String_View sv, char from, char to,
                            char *out, size_t outCap);
```

```c
/* Hacia una cadena C (necesita búfer, para poder agregar '\0') */
size_t      sv_to_cstr(String_View sv, char *out, size_t outCap);
```

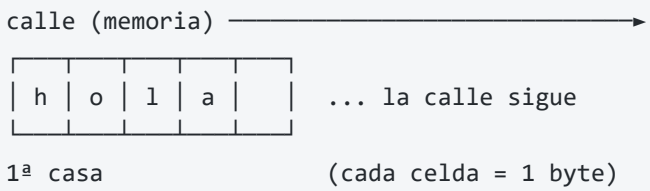
> ***Fijate bien en esto.*** *La estructura* `String_View` *aparece* ***en todas*** *las firmas. Algunas funciones la reciben* ***por valor*** *(* `String_View sv` *) porque no la modifican; otras la reciben* ***por puntero*** *(* `String_View *sv` *) porque la* ***avanzan*** *(recortar y cortar mueven el inicio). Esa distinción es el 80% del diseño.*

PARTE 1 — Los tres pilares y la estructura `String_View`

Antes de escribir una línea de tu librería, hay que ver **tres cosas** sobre cómo C maneja el texto. Si entendés estos tres pilares, todo lo demás cae solo.

1.1 Un texto es una calle, no un objeto

Imaginate una **calle con casas pegadas, una al lado de la otra**. Cada casa es un byte. Las casas no tienen huecos entre sí: están **contiguas**. Un texto en C es exactamente eso: un **bloque de bytes en memoria, uno pegado al otro**.



La palabra clave es **contigua**: los bytes están uno detrás del otro. Eso es lo que hace que "moverse por el texto" sea tan barato: avanzar una casa es **sumar 1** a la dirección.

1.2 El puntero es "la dirección de la primera casa"

Un **puntero** (`char *`) no guarda el texto: guarda **la dirección donde empieza** el texto, o sea "el número de la primera casa". Cuando escribís `"hola"` , el compilador deja esas cuatro letras en alguna parte de la memoria y te da la **dirección** de la `h` .

```c
#include <stdio.h>

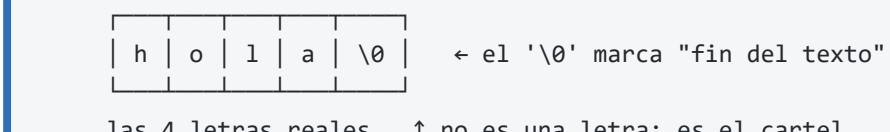
int main(void) {
    const char *s = "hola";

    /* %p imprime la DIRECCIÓN (el número de la primera casa), no el texto.
       s[0] es "la casa 0": la 'h'. s[1] es la siguiente: la 'o'. */
    printf("dirección de inicio : %p\n", (const void *)s);
    printf("la casa 0           : %c\n", s[0]);   /* h */
    printf("la casa 1           : %c\n", s[1]);   /* o */
    return 0;
}
```

> ***Momento ajá.*** `s + 1` ***no*** *es "el texto más uno": es "la dirección de la casa siguiente", o sea apunta a la* `o` *. La* ***aritmética de punteros*** *(capítulo* `07` *) es, literalmente,* ***caminar por la calle****. Todo lo que vas a hacer con* `String_View` *es caminar por la calle sin salirse.*

1.3 El cartel de FIN: el terminador `'\0'`

Acá está la primera gran decisión histórica de C. Como una calle no dice sola dónde termina, C puso un **cartel de FIN** al final de cada texto: el byte `'\0'` (el carácter nulo, valor **0**).



Ese cartel es cómodo para algunas cosas y **un dolor de cabeza** para otras. Mirá las dos caras.

Lo bueno: no hace falta saber el largo de antemano. Funciones como `printf("%s", s)` caminan la calle hasta encontrar el cartel, y listo. Puede pasar un texto de cualquier largo.

Lo malo (tres consecuencias enormes):

1. **Para saber el tamaño hay que caminar la calle entera.** `strlen` **no sabe** cuánto mide: recorre byte por byte hasta topar el `'\0'`. Un texto de un millón de letras cuesta un millón de pasos.
2. **No podés mirar un tramo sin copiarlo.** Si querés "solo la palabra del medio", no hay forma de hacerlo apuntando a la mitad: si cortás la calle, quedás sin cartel de FIN. O copiás a otra calle, o **rompés el texto original** metiendo un `'\0'` en el medio.
3. **Un `\0` en el medio te corta el texto.** Si en algún lado hay un byte 0 (y en datos binarios pasa), el texto "termina" ahí para `strlen`, aunque para vos siga.

```c
#include <stdio.h>
#include <string.h>   /* strlen */

int main(void) {
    const char *s = "hola mundo";

    /* strlen cuenta PASOS hasta el '\0'. No hay un "largo" guardado en ningún lado:
       lo descubre caminando. Por eso es O(n). */
    printf("strlen = %zu\n", strlen(s));   /* 10 */

    /* Y este ejemplo muestra la trampa del '\0' en el medio. */
    const char bytes[5] = { 'h', 'o', '\0', 'l', 'a' };
    printf("strlen del arreglo = %zu\n", strlen(bytes));   /* 2, ¡no 5! */
    return 0;
}
```

El problema de fondo. *C guarda* ***solo la dirección****, nunca el largo. El largo hay que* ***descubrirlo*** *(caminando) o* ***adivinarlo****. Vos vas a hacer lo obvio:* ***guardar el largo****, una vez, y no volver a contarlo.*

Nota fina. *En este capítulo* `strlen` *y compañía viven en* `<string.h>`*. Las vas a usar* ***para arrancar*** *(convertir lo viejo a lo nuevo), pero la gracia de tu librería es, justamente,* ***no depender*** *de tener que recorrer el texto de nuevo en cada operación.*

1.4 Los literales viven en una vitrina blindada (`.rodata`)

Cuando escribís `"hola"` **directamente en el código**, esas letras no viven en cualquier lado: el compilador las mete en una sección de **solo lectura** del binario, llamada `.rodata` (del inglés *read-only data*). Es una **vitrina blindada**: podés **mirar** lo que hay adentro, pero si intentás **tocarlo**, el sistema operativo te baja el telón con un **Segmentation Fault**.

```c
#include <stdio.h>

int main(void) {
    /* El compilador ADVIERTE con esta línea (const char *). Prestá atención. */
    const char *s = "hola";

    /* ESTO ES UN ERROR EN TIEMPO DE EJECUCIÓN: s[0] es de solo lectura.
       Descomentar la línea de abajo hace que el programa "crash" (Segmentation Fault).
       La dejamos comentada a propósito para que puedas compilar y correr. */
    /* s[0] = 'H'; */

    printf("%s\n", s);   /* imprimir (leer) sí se puede, siempre */
    return 0;
}
```

Y si querés un texto **que sí podés modificar**, no lo definas con un literal: definilo como un **arreglo** (`char buf[]`), que copia las letras a memoria escribible.

```c
#include <stdio.h>

int main(void) {
    /* Con corchetes, el compilador crea un ARREGLO con su propia copia del texto:
       esa copia SÍ es modificable. */
    char buf[] = "hola";
    buf[0] = 'H';                /* ahora sí se puede */
    printf("%s\n", buf);         /* Hola */
    return 0;
}
```

> ***La regla que te salva.*** `const char *s = "literal";` → ***solo leer***. *Si necesitás escribir, usá* `char buf[] = "literal";` *(arreglo) o una* `String_View` *, que es* ***siempre de solo lectura*** *(mirá el* `const` *en* `data` *). Por eso tu librería* ***nunca*** *va a modificar la memoria original: solo la mira.*

1.5 El nuevo paradigma: la ventana

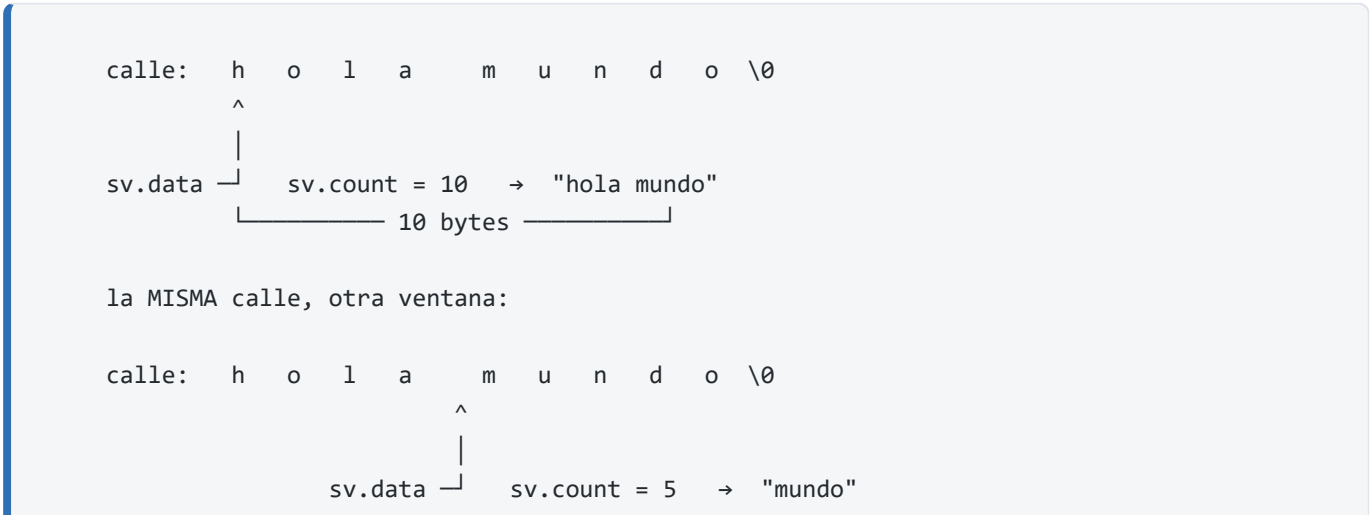
Junta las tres ideas y tenés la solución. En vez de depender del cartel de FIN y de caminar la calle, guardá **dos datos**:

- `data` : la dirección donde empieza lo que mirás.
- `count` : cuántos bytes querés mirar.

Eso es todo. Es como una **ventana** (o una foto recortada) apoyada sobre la calle: la calle sigue estando completa; vos solo decidís **desde dónde** y **hasta dónde** mirás.

```c
typedef struct {
    const char *data;   /* Puntero al inicio del texto   */
    size_t      count;  /* Cantidad exacta de bytes       */
} String_View;
```

Mirá la calle completa y una ventana sobre ella:



Fijate algo hermoso: **ninguna de las dos ventanas cambió un byte**. Las dos apuntan a la misma calle. "hola mundo" y "mundo" son **dos vistas** de la misma memoria. Eso es imposible de hacer gratis con cadenas C terminadas en `'\0'` .

> ***¿Y el `'\0'` ?*** *Con `String_View` **no lo necesitás**: vos ya sabés el `count` . La vista puede apuntar al medio de un texto sin cartel de FIN alguno. El `\0` pasa de ser una **obligación** a ser un **detalle** que solo te importa cuando tenés que entregarle el texto a una función vieja que sí lo espera (como `printf`).*

> ***Cómo se lee*** `size_t`***.*** *Es un entero* ***sin signo*** *pensado para tamaños (capítulo* `16` *). Es el tipo correcto para* `count` *: un largo* ***nunca*** *es negativo. Guardátelo para cuando quieras devolver un "no encontrado": ahí* ***no*** *vas a poder usar* `size_t` *(no tiene negativos) y vas a necesitar un centinela con signo. Lo ves en la Parte 2.*

1.6 Imprimir sin `'\0'` : el truco del `%.*s`

Acá viene la primera piedra en el zapato. Si tenés una `String_View` , **no podés** hacer `printf("%s", sv.data)` : `printf` va a caminar hasta encontrar un `\0` ... que **quizás no exista** donde vos querés (puede seguir el resto de la calle, o directamente no haber nunca). Necesitás decirle a `printf` : **"mirá exactamente** `count` **bytes, ni uno más"**.

Para eso existe una combinación del formato de `printf` : `%.*s` . Se lee así:

- El `.` dice "precisión" (para `%s` , cuántos caracteres como máximo).
- El `*` dice "y ese máximo **no lo escribo acá**: te lo paso como argumento, justo antes del puntero".

```c
#include <stdio.h>

int main(void) {
    const char *data  = "hola mundo entero";
    size_t      count = 10;             /* queremos solo "hola mundo" */

    /* %.*s = "imprimí una cadena, pero mirá a lo sumo N bytes".
       El N llega como argumento (int) ANTES del puntero. Por eso van DOS argumentos. */
    printf("[%.*s]\n", (int)count, data);   /* [hola mundo]  (sin el " entero") */
    return 0;
}
```

¿Por qué `(int)count` y no `count` a secas? Porque el `*` de `printf` espera un `int` , y `count` es `size_t` (que puede ser más ancho). El `cast` lo deja del tamaño exacto que `printf` va a leer. Si te olvidás el cast, en muchas máquinas "funciona igual", pero es una bomba de tiempo: el tipo que pasás tiene que coincidir con el que `printf` espera.

Como esto se usa **todo el tiempo**, lo natural es envolverlo en un **par de macros** que no te hagan escribir el `.%*s` ni el `(int)` a mano en cada `printf`:

```c
#include <stdio.h>

/* SV_Fmt: el "molde" de formato, para pegar DENTRO de un printf junto al resto del
texto. */
#define SV_Fmt "%.*s"

/* SV_Arg: expande a los DOS argumentos que %.*s necesita: el largo y el puntero. */
#define SV_Arg(sv) (int)(sv).count, (sv).data
```

Y se usan así, encajando en cualquier `printf`:

```c
printf("Nombre: [" SV_Fmt "]\n", SV_Arg(sv));
/*            ↑ el molde           ↑ los dos argumentos
   equivale a:
   printf("Nombre: [%.*s]\n", (int)sv.count, sv.data); */
```

Por qué las macros y no una función. *El* `%.*s` *tiene que quedar* ***pegado dentro*** *del* `printf` *(es parte del formato), y los dos argumentos tienen que ir* ***en el medio*** *de los demás. Una función no puede hacer eso: devuelve* ***un*** *valor, no "dos argumentos intercalados en un* `printf` *". Las macros sí, porque son* ***texto que se reemplaza****. Por eso esta pareja es el idioma estándar.*

Pensalo así. `%s` *le dice a* `printf` *"caminá hasta el cartel de FIN".* `%.*s` *le dice "mirá exactamente tanto como yo te diga". Cambiaste "descubrir el largo" por "decir el largo". Es la misma idea de todo el capítulo, aplicada a la impresión.*

Práctica de la Parte 1

Ejercicio 18.1 — El conversor (`sv_from_cstr`)

Conceptos · `struct` , `String_View` , `strlen` , `size_t` , punteros a `const`

Contexto · Todo tu código nuevo va a trabajar con `String_View` , pero el mundo entero está lleno de cadenas C terminadas en `'\0'` (literales, `argv` , lo que devuelve `fgets`). Necesitás una **aduana** que convierta de un mundo al otro. Simplificación: una función que mide una cadena C y la envuelve en una vista.

Consigna · Implementá `sv_from_cstr` . Debe recibir un `const char *` , calcular su largo **una sola vez** (con `strlen`) y devolver una `String_View` con ese `data` y ese `count` . Después, en el `main` , creá una vista a partir de un literal y **imprimila** con `SV_Fmt` / `SV_Arg` , mostrando también el `count` .

Entrada · Una cadena C (un literal o algo leído con `fgets`). **Salida** · El texto de la vista y su largo en bytes.

Escenario de prueba

```
entrada : "hola mundo"
salida  : texto = [hola mundo]
          count = 10

entrada : ""
salida  : texto = []
          count = 0
```

Identificadores sugeridos · `sv_from_cstr` , `sv` , `text` **Criterios de aceptación**

- [ ] `strlen` se llama **una sola vez** (el largo no se recalcula después).
- [ ] `data` apunta al **mismo** texto original (no hay copia).
- [ ] `count` coincide con `strlen` .
- [ ] La cadena vacía da `count == 0` y no se rompe.

Ejercicio 18.2 — La impresora de vistas (macro `SV_Fmt` + `sv_print`)

Conceptos · `%.*s`, `printf`, macros, `const char *`

Contexto · Cada vez que depures tu librería vas a querer ver una vista. Como `printf("%s", ...)` no sirve (no hay `'\0'` garantizado), necesitás las macros de formato y una función cómoda. Simplificación: imprimir una vista **sin** depender del terminador.

Consigna ·

1. Definí las macros `SV_Fmt` y `SV_Arg`.
2. Escribí `sv_print` que reciba una `String_View` y la imprima con `SV_Fmt` / `SV_Arg` (sin `\0`).
3. Comprobá que con una vista que **no** termina en `'\0'` (por ejemplo, un tramo en el medio de una cadena C) la impresión **no** se pasa de `count`.

Entrada · Una o varias vistas (algunas a mitad de un texto). **Salida** · El contenido exacto de cada vista, encerrado entre corchetes.

Escenario de prueba

```
cadena completa : "hola mundo"
vista           : data → "mundo", count = 5     (un tramo del medio, sin '\0' propio)

salida          : [mundo]     ← NO debe imprimir nada más allá del 'o'
```

```
vista vacía (count = 0)  →  []
```

Identificadores sugeridos · `SV_Fmt`, `SV_Arg`, `sv_print` **Criterios de aceptación**

- [ ] `sv_print` usa `%.*s` (no `%s`).
- [ ] Se imprime **exactamente** `count` bytes, sin mirar lo que sigue después.
- [ ] Una vista en el medio de un texto imprime **solo** su tramo.
- [ ] Una vista vacía imprime los corchetes vacíos, sin basura.

PARTE 2 — Sub-vistas: mirar tramos sin copiar

Ya tenés la ventana. Ahora aprendé a **moverla**. Esta parte es corta y es el corazón del paradigma: todo se reduce a **ajustar dos números**.

2.1 Recortar = mover el inicio, achicar el largo

Imaginate que querés la **primera palabra** de `"hola mundo"`. Con cadenas C: copiás a otro lado o rompés el original. Con `String_View`: **no copiás nada**, solo hacés una **vista nueva** que empieza donde mismo y mide menos.

```
original:   h o l a   m u n d o \0
            ^
            sv.data, sv.count = 10

sv_left(sv, 4):  la MISMA dirección, pero count = 4
            h o l a
            ^
            misma data, count = 4   →  "hola"

sv_right(sv, 5): esta vez SÍ movemos el inicio
                    m u n d o
                    ^
                    data = data + 5, count = 5   →  "mundo"
```

Fijate la diferencia entre las dos: `sv_left` mantiene el inicio y **acorta**; `sv_right` **mueve el inicio** hacia adelante y ajusta el largo. Las dos devuelven una **vista nueva**, sin tocar la original ni la memoria.

> ***La cuenta que va a reaparecer siempre.*** *Para "los últimos* `n` *": el nuevo inicio es* `data + (count - n)` *y el nuevo largo es* `n`*. Para "los primeros* `n` *": el inicio no cambia, y el largo pasa a ser* `min(n, count)`*. Ese* `min` *es* ***la protección*** *contra pasarte: pedir 100 de un texto de 10 no es un error, es pedir "todo".*

2.2 ¿Y si no encuentro lo que busco? El centinela

`sv_index_of` busca un carácter y devuelve **dónde** está. Pero, ¿qué devolvés si **no está**? Esta es una decisión de diseño que se equivoca todo el mundo la primera vez.

- `size_t` **no sirve** para el "no está": es **sin signo**, no tiene `-1`. Si devolvieras el tamaño del texto como "no encontrado", te confundirías con "está al final".
- Necesitás un valor que **nunca** sea un índice válido. El clásico es `-1`, y para eso necesitás un tipo **con signo** (`int` o, más correcto para índices, `ptrdiff_t`, capítulo `16`).

```
sv_index_of("banana", 'n')  →  2      (índice de la primera 'n')
sv_index_of("banana", 'z')  →  -1     (centinela: "no está")
```

> ***Regla general de diseño.*** *Cuando una función puede* ***fallar****, decidí* ***explícitamente*** *cómo avisa: o devuelve un* ***centinela*** *(como `-1`), o separa el "hubo suerte" en un `int` de retorno (como hace `sscanf`, capítulo `14`) y saca el valor por puntero. Nunca dejes el error "implícito": es el bug más caro de esta librería.*

Práctica de la Parte 2

Ejercicio 18.3 — El buscador de un carácter (`sv_index_of`)

Conceptos · recorrido por índice, `const char *`, centinela `-1`, tipo con signo

Contexto · Antes de partir texto necesitás saber **dónde** están las marcas (la coma, el `=`, el espacio). Simplificación: ubicar la primera aparición de un carácter.

Consigna · Implementá `sv_index_of`: recorre la vista y devuelve el **índice** (base 0) de la primera aparición del carácter, o `-1` si no aparece. No uses `strchr` (esa necesita `'\0'`): el recorrido es tuyo y está acotado por `count`. Comprobá con un carácter presente, uno ausente y el caso de la vista vacía.

Entrada · Una `String_View` y un carácter. **Salida** · El índice de la primera aparición, o `-1`.

Escenario de prueba

```
sv = "banana"
  sv_index_of(sv, 'b') → 0
  sv_index_of(sv, 'n') → 2      (la PRIMERA 'n')
  sv_index_of(sv, 'a') → 1
  sv_index_of(sv, 'z') → -1     (centinela)
  sv_index_of("", 'a') → -1
```

Identificadores sugeridos · `sv_index_of` , `i` , `found` **Criterios de aceptación**

- [ ] El recorrido respeta `count` (no se pasa ni necesita `'\0'`).
- [ ] Se devuelve el índice **base 0** de la **primera** aparición.
- [ ] Ausente y vacío devuelven `-1` .
- [ ] El tipo de retorno es **con signo** (para poder representar `-1`).

Ejercicio 18.4 — Los primeros n (`sv_left`)

Conceptos · sub-vista, `min` , protección de límites, `size_t`

Contexto · "Dame el prefijo de este texto" es lo que hacés para quedarte con una etiqueta, un código de país, un encabezado. Simplificación: una vista con los primeros `n` bytes.

Consigna · Implementá `sv_left` : devuelve una **vista nueva** con los primeros `n` bytes. **No** copies memoria: la `data` es la misma, solo cambia el `count` . Si `n` es **mayor** que el largo, devolvé la vista completa (protección por `min`). `n = 0` da una vista vacía.

Entrada · Una `String_View` y un `n` (`size_t`). **Salida** · Una nueva vista con la parte pedida.

Escenario de prueba

```
sv = "hola mundo"
  sv_left(sv, 4)   → "hola"
  sv_left(sv, 0)   → ""          (count = 0)
  sv_left(sv, 100) → "hola mundo" (se recorta al total, no es error)
```

Identificadores sugeridos · `sv_left` , `n` , `result` **Criterios de aceptación**

- [ ] No se copia ningún byte (misma `data`).
- [ ] `n` mayor que el largo devuelve la vista completa (usa `min`).
- [ ] `n = 0` da `count == 0`.
- [ ] La vista original **no** se modifica (se pasa por valor).

Ejercicio 18.5 — Los últimos n (`sv_right`)

Conceptos · sub-vista, aritmética de punteros (`data + offset`), protección de límites

Contexto · "Dame la extensión de este archivo" (los últimos 4 caracteres de `.txt`) es un `sv_right` de manual. Simplificación: una vista con los últimos `n` bytes.

Consigna · Implementá `sv_right`: devuelve una vista nueva con los **últimos** `n` bytes. Acá sí hay que **mover el inicio**: `data + (count - n)`. Protegé el caso `n` mayor que el largo (devolvé la vista completa) y `n = 0` (vista vacía), **sin** que la resta se dé vuelta (ojo con `size_t`, que no tiene negativos).

Entrada · Una `String_View` y un `n` (`size_t`). **Salida** · Una nueva vista con la parte pedida.

Escenario de prueba

```
sv = "hola mundo"
  sv_right(sv, 5)   → "mundo"
  sv_right(sv, 0)   → ""
  sv_right(sv, 100) → "hola mundo"
```

> ***La trampa del `size_t`.*** *`count - n` con `size_t` y `n > count` **no** da un negativo: da un número gigante (le da la vuelta). Por eso **primero** hay que asegurar `n <= count`; recién después restar.*

Identificadores sugeridos · `sv_right`, `n`, `start`, `result` **Criterios de aceptación**

- [ ] El nuevo inicio es `data + (count - n)` cuando `n <= count`.
- [ ] `n` mayor que el largo devuelve la vista completa.
- [ ] `n = 0` da una vista vacía.
- [ ] No hay restas que "den la vuelta" por el `size_t`.

PARTE 3 — Recorte (trim): sacar los bordes

Los archivos de configuración, las líneas de un CSV, lo que teclea un usuario... **todo** viene con espacios de más. El **trim** los saca. Con `String_View` es casi gratis: recortar es **mover dos números**.

3.1 `isspace`: qué cuenta como "espacio"

No es solo el espacio de la barra espaciadora. `<ctype.h>` te da `isspace`, que reconoce **seis** caracteres en blanco:

Carácter	Nombre	Cómo se escribe
espacio	*space*	`' '`
tabulación	*tab*	`'\t'`
salto de línea	*line feed*	`'\n'`
retorno de carro	*carriage return*	`'\r'`
tabulación vertical	*vertical tab*	`'\v'`
avance de formulario	*form feed*	`'\f'`

`isspace(c)` devuelve **distinto de cero** ("verdadero") si `c` es uno de esos, y `0` ("falso") si no. Es la forma **portable** de preguntar "¿es un blanco?", sin enumerar los seis a mano.

> ***El detalle que casi nadie sabe.*** *A* `isspace` *hay que pasarle un valor que sea* ***representable como*** `unsigned char` *o* `EOF`*. Si le pasás un* `char` *con signo y el byte vale, por ejemplo,* `0xC3` *(el primer byte de una* `á` *en UTF-8), un* `char` *con signo lo ve como* ***negativo****, y eso es* ***comportamiento indefinido****. La forma segura es* `isspace((unsigned char)c)`*. Guardátelo.*

3.2 Recortar a la izquierda: avanzar y achicar

`sv_trim_left` **mira desde el frente** y va descartando blancos. Cada blanco que saca **avanza** `data` y **resta 1 a** `count`. Cuando topa con un byte que no es blanco, para.

```
antes :  "   hola  "
         ^ data, count = 9

mira el 1º: ' ' → blanco → data++, count--
mira el 1º (ahora): ' ' → blanco → data++, count--
...
después:"hola  "
             ^ data ya movido hasta la 'h', count = 6
```

Fijate que **el texto original no cambia**: sigue teniendo los tres espacios al principio en memoria. Lo que cambió es **qué tramo mira** la vista.

3.3 Recortar a la derecha: achicar el contador

`sv_trim_right` **mira desde el fondo** y da el mismo tratamiento, pero acortando `count` (el inicio no se mueve).

```
antes :  "  hola  "
                     ^ mirá los últimos
mira el último: ' ' → blanco → count--
...
después:"  hola"
                   ^ count ya achicado
```

> ***El mismo dato, dos preguntas.*** `trim_left` *pregunta "¿el* ***primero*** *es blanco?";* `trim_right`*, "¿el* ***último*** *es blanco?". El primero es* `sv.data[0]`*; el último es* `sv.data[sv.count - 1]`*. ¡Cuidado con* `count == 0`*! (una vista vacía no tiene "último": hay que chequear el largo* ***antes*** *de mirar el último byte).*

3.4 `trim` = `trim_left` + `trim_right`

`sv_trim` es literalmente llamar a las dos, en cualquier orden (no se pisan: una toca el frente y la otra el fondo). El resultado: el texto "pelado", sin blancos en ninguno de los dos extremos.

```
"   hola mundo   "   ──trim──►   "hola mundo"
"     "             ──trim──►   ""     (una vista vacía: count = 0)
```

3.5 ¿Por qué estas reciben la vista por puntero?

Acá está la regla de oro del diseño de tu librería:

- Si una función **solo mira** y devuelve algo nuevo → recibe `String_View` **por valor** (una copia del par `data` / `count`; cómodo y seguro).
- Si una función **modifica la vista misma** (la avanza) → recibe `String_View *` **por puntero**.

`trim_left` cambia la vista que le pasás ("consumila"), así que va **por puntero**: `sv_trim_left(&sv)`. Así, quien llama ve el cambio. (Alternativa válida: devolver la vista nueva. Elegí una convención y mantenela en **toda** la librería.)

Práctica de la Parte 3

Ejercicio 18.6 — Recorte a la izquierda (`sv_trim_left`)

Conceptos · `isspace`, `(unsigned char)`, punteros a `struct`, avance de inicio

Contexto · Al leer una línea de un archivo de configuración, el lado izquierdo casi siempre trae indentación o tabs. Simplificación: descartar los blancos del frente, avanzando la vista.

Consigna · Implementá `sv_trim_left(String_View *sv)`: mientras el **primer** byte sea blanco (`isspace`, con el cuidado del `unsigned char`), **avanzá** `data` y **restá** `count`. Detenete al primer byte que no sea blanco o cuando la vista quede vacía. La función **no devuelve** nada: modifica la vista recibida.

Entrada · Una vista, por puntero. **Salida** · La misma vista, ya recortada por la izquierda (se ve en el `main`).

Escenario de prueba

```
antes : "    hola  "      (4 espacios al frente)
después: "hola  "

antes : "hola"            (sin blancos)
después: "hola"           (no cambia)

antes : "     "           (solo blancos)
después: ""               (count = 0)
```

Identificadores sugeridos · `sv_trim_left` , `sv` , `c` **Criterios de aceptación**

- ☐ Usa `isspace` con `(unsigned char)` (no con un `char` con signo).
- ☐ Cada blanco del frente **avanza** `data` y **decrementa** `count` .
- ☐ Para en el primer byte no blanco (o al vaciarse).
- ☐ Modifica la vista **por puntero**; el texto original no se toca.

Ejercicio 18.7 — Recorte a la derecha (`sv_trim_right`)

Conceptos · `isspace` , `count - 1` , chequear `count` antes de mirar, achicar el largo

Contexto · Las líneas de un archivo suelen venir con `'\r'` (retorno de carro) o espacios al final. Simplificación: descartar los blancos del fondo, achicando el largo.

Consigna · Implementá `sv_trim_right(String_View *sv)` : mientras la vista **no esté vacía** y el **último** byte (`data[count - 1]`) sea blanco, restá `1` a `count` . Chequeá **siempre** `count > 0` antes de mirar el último byte (mirar `data[count - 1]` con `count == 0` es leer fuera de límites).

Entrada · Una vista, por puntero. **Salida** · La misma vista, recortada por la derecha.

Escenario de prueba

```
antes : "  hola    "      (espacios al fondo)
después: "  hola"

antes : "hola\r"          (un retorno de carro al final)
después: "hola"

antes : ""                (vista vacía)
después: ""               (no se accede a data[-1])
```

Identificadores sugeridos · `sv_trim_right` , `sv` **Criterios de aceptación**

- ☐ Chequea `count > 0` **antes** de leer `data[count - 1]` .
- ☐ Cada blanco del fondo **decrementa** `count` (no mueve `data`).
- ☐ Reconocen como blancos el `'\r'` y demás (vía `isspace`).
- ☐ Una vista vacía no produce accesos fuera de rango.

Ejercicio 18.8 — Recorte total (`sv_trim`) y la trampa del doble recorte

Conceptos · composición de funciones, orden de operaciones, vistas que quedan vacías

Contexto · Un parser quiere el campo "pelado": sin blancos de ningún lado. Simplificación: recortar frente y fondo de una vez.

Consigna · Implementá `sv_trim` llamando a `sv_trim_left` **y** a `sv_trim_right` (en el orden que prefieras, y explicá en un comentario **por qué** el orden no importa). Comprobá el caso extremo: una vista hecha **solo** de blancos tiene que terminar con `count == 0` , sin romperse.

Entrada · Varias vistas, con distinta cantidad de blancos. **Salida** · Cada vista ya recortada, entre corchetes.

Escenario de prueba

```
"   hola mundo   "  →  [hola mundo]
"hola"              →  [hola]
"     "             →  []          (count = 0)
"   "               →  []          (count = 0)
"a b"               →  [a b]       (¡los espacios INTERNOS no se tocan!)
```

> ***Ojo con esto.*** `trim` *saca los blancos de los* ***bordes****, no los de adentro.* `"a b"` *sigue siendo* `"a b"` *. Confundir "recortar los bordes" con "sacar todos los blancos" es un bug clásico de parseo.*

Identificadores sugeridos · `sv_trim` **Criterios de aceptación**

- [ ] `sv_trim` reusa `sv_trim_left` y `sv_trim_right` (no reimplementa).
- [ ] La vista "solo blancos" termina con `count == 0` .
- [ ] Los espacios **internos** se conservan.
- [ ] El comentario explica por qué el orden entre los dos recortes es indiferente.

PARTE 4 — Cortar por delimitador (chop y split)

Esta es **la** parte donde el paradigma se saca la foto. Con `chop` vas a partir un texto en pedazos **uno por uno**, avanzando la vista, **sin reservar un solo byte** de memoria nueva.

4.1 La tijera: `sv_chop_by_delim`

Imaginate una **tira larga de chicle** y una **tijera** en la coma. `chop` hace tres cosas, en orden:

1. **Busca** el delimitador (`sv_index_of`).
2. **Devuelve** el trozo de **antes** del delimitador.
3. **Avanza** la vista original para que arranque **justo después** del delimitador.

Pensalo como que **le cortás el primer pedazo a la tira**, y la tira que te queda (en `*sv`) empieza después del corte.

```
antes:     sv ──► "ana,luis,pepe"

           chop(sv, ',')

devuelve:  "ana"                (el trozo de antes)
y deja:    sv ──► "luis,pepe"   (arranca DESPUÉS de la coma)
```

Y el caso especial que **siempre** hay que resolver: **¿y si ya no queda delimitador?** (el último pedazo). Entonces `chop` devuelve **todo** lo que queda y deja la vista **vacía**:

```
antes:     sv ──► "pepe"

devuelve:  "pepe"
y deja:    sv ──► ""            (count = 0 → el bucle terminó)
```

Ese vaciado es lo que hace que el **bucle de split** corte solo, sin necesitar un contador aparte.

> ***Fijate que `chop` combina todo lo anterior.*** *Usa `sv_index_of` (Parte 2) para ubicar la coma y **mueve el inicio** (como `sv_chop_left`, Parte 2) para consumirla. Lo que ya construiste se vuelve a usar. Eso es una librería: piezas pequeñas que se combinan.*

4.2 El bucle de split: partir sin `malloc`

Con `chop`, partir una línea es un bucle de tres líneas. La condición de corte es elegante: **cuando la vista queda vacía, se terminó**.

```
línea: "ana,luis,pepe"

vuelta 1:  chop → "ana"    sv = "luis,pepe"
vuelta 2:  chop → "luis"   sv = "pepe"
vuelta 3:  chop → "pepe"   sv = ""        ← sin coma, devuelve el resto
fin:       sv vacía → el bucle corta
```

Cada pedazo es una **vista nueva** que apunta al **mismo texto original**. No copiaste nada, no pediste memoria, no liberaste nada. Eso es un parser de velocidad absurda.

> ***El detalle de la coma final.*** *Si el texto es `"a,b,"`, el último `chop` devuelve la vista vacía (entre la última coma y el final no hay nada). Un split correcto suele incluir ese último campo vacío. Decidí **explícitamente** qué querés y dejalo documentado: es una fuente inagotable de discusiones (mirá cómo `"a,b,".split(',')` se comporta distinto en cada lenguaje).*

Práctica de la Parte 4

Ejercicio 18.9 — La tijera (`sv_chop_by_delim`)

Conceptos · `sv_index_of` , sub-vistas, pasar por puntero para consumir, avance de inicio

Contexto · Un tokenizador de CSV, de paths o de líneas `clave=valor` necesita **sacar el siguiente pedazo** y dejar la vista lista para el siguiente. Simplificación: cortar por un delimitador.

Consigna · Implementá `sv_chop_by_delim(String_View *sv, char delim)` :

1. busca `delim` con `sv_index_of` ;
2. si **lo encuentra** en el índice `i` , devolvé `sv_left(*sv, i)` y dejá `*sv` empezando **después** del delimitador (`data + i + 1` , `count - i - 1`);
3. si **no** lo encuentra (`-1`), devolvé **todo** `*sv` y dejá `*sv` **vacía** (`count = 0`).

Entrada · Una vista por puntero y un delimitador. **Salida** · El pedazo de antes del delimitador; la vista original queda avanzada.

Escenario de prueba

```
sv = "ana,luis,pepe", delim = ','
  chop → "ana"    ; sv = "luis,pepe"
  chop → "luis"   ; sv = "pepe"
  chop → "pepe"   ; sv = ""        (sin delimitador: devuelve el resto y vacía)
  chop → ""       ; sv = ""        (ya estaba vacía)

sv = "sin,comas,aqui", delim = ';'
  chop → "sin,comas,aqui"  ; sv = ""   (el delimitador no aparece)
```

Identificadores sugeridos · `sv_chop_by_delim` , `sv` , `delim` , `i` , `piece`
Criterios de aceptación

- ☐ Reusa `sv_index_of` (no reimplementa la búsqueda).
- ☐ Avanza la vista a `data + i + 1` con `count - i - 1` cuando encuentra el delimitador.
- ☐ Sin delimitador, devuelve el resto **y** deja la vista vacía.
- ☐ El delimitador **no** aparece en ningún pedazo devuelto.

Ejercicio 18.10 — El split completo (bucle, sin `malloc`)

Conceptos · bucle `while` con condición sobre `count`, `chop`, impresión de tokens

Contexto · El `split` es el pan de cada día: partir una línea de campos. Simplificación: imprimir cada campo de una línea separada por comas, **sin reservar memoria**.

Consigna · Escribí un programa que tome una línea separada por comas (fija en el `main`, o leída con `fgets`) y, con un **bucle** que llame a `sv_chop_by_delim`, imprima **cada campo** entre corchetes, uno por línea. Al terminar, mostrá **cuántos** campos salieron. Agregá al menos un campo con espacios alrededor para forzar un `trim` por campo (reusá `sv_trim`).

Entrada · Una línea con campos separados por comas. **Salida** · Un campo por línea y el total.

Escenario de prueba

```
entrada : "ana, luis ,pepe"

salida  :
[ana]
[luis]
[pepe]
campos: 3
```

```
entrada : "solo"

salida  :
[solo]
campos: 1
```

> ***Regla de oro del bucle.*** *La condición es* `sv.count > 0`*. Cada* `chop` ***achica o vacía*** *la vista, así que el bucle* ***siempre*** *termina. Si te quedás en un bucle infinito, es porque olvidaste que el* `chop` *sin delimitador* ***vacía*** *la vista (revisá 18.9).*

Identificadores sugeridos · `sv`, `field`, `count`, `line` **Criterios de aceptación**

- [ ] El bucle usa `sv_chop_by_delim` y corta cuando la vista queda vacía.
- [ ] Cada campo se recorta con `sv_trim` (los espacios de alrededor se van).
- [ ] Se imprime un campo por línea.
- [ ] **No** hay ninguna llamada a `malloc` (ni a `strtok`, que además rompe el texto original).

PARTE 5 — Reemplazar: mutación segura en un búfer

Llegó el momento incómodo: **reemplazar**. Acá la `String_View` muestra su límite, y entender **por qué** es la clave de esta parte.

5.1 El problema: la vista no es dueña de la memoria

Una `String_View` es una **ventana**, no una casa: apunta a memoria que **no le pertenece** y que normalmente es de **solo lectura** (recordá `.rodata`, Parte 1). Por eso no puede "crecer" el texto, ni escribir en él.

¿Y cuando el reemplazo cambia el **tamaño**? Pasa siempre:

```
reemplazar "o" por "oo":
"hola"  →  "hoola"     el texto CRECIÓ: ¿dónde metés el byte de más?
reemplazar "aa" por "b":
"baaanana" → "banana"  el texto ENCOGIÓ
```

Si el resultado ocupa distinto que la fuente, **no hay forma** de escribirlo "en el mismo lugar". La vista no tiene lugar propio. La solución honesta es: **el que llama provee un búfer destino**, y la función escribe ahí. La `String_View` sigue siendo **solo fuente de lectura**.

> ***Nadie miente en esta librería.*** *Una operación que* ***no cambia el tamaño*** *(como reemplazar un carácter de 1 byte por otro de 1 byte)* ***sí podría*** *escribirse en el lugar… pero la vista es* `const` *: no se toca. La regla clara y segura es:* ***todas las transformaciones escriben en un búfer destino que vos manejás****. Menos sorpresas, menos Segmentation Faults.*

5.2 El contrato: "escribí en MI búfer, sin desbordarlo"

Toda función de reemplazo de tu librería sigue **el mismo contrato**, y conviene que lo memorices:

```
entradas:
  sv      → de dónde LEO (mi vista, const)
  from    → qué busco reemplazar
  to      → por qué lo cambio
  out     → dónde ESCRIBO  (búfer que el que llama reservó)
  outCap  → cuántos bytes entran en out (SU tamaño)

salida:
  cuántos bytes escribí realmente (para que sepas el largo del resultado)
```

Las **tres reglas de oro** de este contrato:

1. **Nunca escribas más de `outCap` bytes.** Si el resultado no entra, tenés que **avisar** (devolver un valor especial) o **cortar**. Jamás desbordar.
2. **No confíes en que `out` es "suficientemente grande".** El que llama puede equivocarse; vos tenés que ser el guardián.
3. **La vista de entrada es de solo lectura.** Escribís **solo** en `out`.

```
sv: "banana"   from='a'  to='o'
     │
     ▼  (leo, nunca escribo acá)
   "banana"
     │
     ▼  (escribo el resultado, con cuidado de no pasarme)
out:| b | o | n | o | n | o | \0? |
     └─────────────────────────┘
     outCap = tamaño REAL de out
```

> ***¿Y el `'\0'` al final del búfer?*** *Depende del contrato que elijas. Si querés que el resultado sea también una* ***cadena C*** *usable con `%s`, tenés que* ***reservar un byte*** *para el terminador y escribirlo. Por eso el contrato de `sv_to_cstr` pide "como máximo `outCap - 1` caracteres + el `\0`". Decidí y documentá: ¿el resultado lleva `'\0'` o no? (Con `String_View`,* ***no*** *lo necesita; para `printf("%s")`,* ***sí****.)*

5.3 `sv_replace_char` : reemplazar un solo carácter

El caso más simple y el más común (cambiar comas por puntos, normalizar separadores). Recorrés la vista **byte a byte** y, por cada uno, decidís qué escribís en `out` :

- si el byte es `from` → escribís `to` ;
- si no → escribís el byte tal cual.

Contás cuántos reemplazaste (para informar) y **cuidás** no pasarte de `outCap` .

Práctica de la Parte 5

Ejercicio 18.11 — Reemplazo de un carácter (`sv_replace_char`)

Conceptos · búfer destino, contrato `out` / `outCap` , recorrido acotado, no desbordar

Contexto · Normalizar un CSV (cambiar `;` por `,`), tapar separadores, sanear texto: reemplazar un carácter es una operación de todos los días. Simplificación: reemplazar **un** carácter por **otro**, escribiendo en un búfer provisto por quien llama.

Consigna · Implementá `sv_replace_char(String_View sv, char from, char to, char *out, size_t outCap)` que copie `sv` a `out` , cambiando cada `from` por `to` , y **devuelva cuántos bytes escribió**. Reglas: nunca escribir más de `outCap` bytes (si el resultado no entra, avisá devolviendo un valor imposible, por ejemplo `(size_t)-1` , o **cortá** y documentalo). La vista `sv` **no** se modifica. Comprobá con un texto que entre justo, y con uno que **no** entre.

Entrada · Una vista, el carácter origen, el destino, y un búfer con su capacidad. **Salida** · El texto reemplazado en el búfer y la cantidad de bytes escritos.

Escenario de prueba

```
sv = "banana", from='a', to='o', outCap = 10 (búfer grande)
  resultado → "bonono"
  bytes escritos → 6

sv = "a;b;c", from=';', to=',', outCap = 10
  resultado → "a,b,c"
  bytes escritos → 5

sv = "aaaaaaaaaa" (10 'a'), from='a', to='o', outCap = 4   (¡no entra!)
  resultado → (error/corte) : NO se desborda el búfer
```

Identificadores sugeridos · `sv_replace_char` , `from` , `to` , `out` , `outCap` , `written` **Criterios de aceptación**

- [ ] Escribe **solo** en `out` ; `sv` (o su memoria) no se toca.
- [ ] Nunca escribe más de `outCap` bytes (probado con el caso que no entra).
- [ ] Devuelve la cantidad de bytes realmente escritos.
- [ ] Si no entra, avisa de forma predecible (nunca desborda ni crashea).

Ejercicio 18.12 — Reemplazo de una subcadena (`sv_replace_all`)

Conceptos · búsqueda de subcadena, búfer destino acotado, cambio de longitud, cálculo de tamaño

Contexto · Reemplazar **todas** las apariciones de una subcadena es el "buscar y reemplazar" de un editor, o la limpieza de plantillas (`{{NAME}}`). Es más difícil porque **cambia el largo**. Simplificación: reemplazar una subcadena `needle` por otra `with` , escribiendo en un búfer destino.

Consigna · Implementá `sv_replace_all(String_View sv, String_View needle, String_View with, char *out, size_t outCap)` que recorra `sv` de izquierda a derecha; cada vez que encuentre `needle` , escriba `with` en `out` ; si no, copie el byte y avance uno. **Nunca** desbordes `outCap` . Devolvé cuántos bytes escribiste.

Entrada · La vista fuente, la subcadena a buscar, la subcadena de reemplazo, y el búfer destino con su capacidad. **Salida** · El texto con **todas** las apariciones reemplazadas y la cantidad de bytes escritos.

Escenario de prueba

```
sv = "ababab", needle = "ab", with = "X", outCap = 16
  resultado → "XXX"
  bytes escritos → 3

sv = "aaabaaa", needle = "aaa", with = "Z", outCap = 16
  resultado → "ZbZ"

sv = "hola", needle = "xy", with = "!!", outCap = 16
  resultado → "hola"        (no hay coincidencias: se copia tal cual)
```

> ***El caso que te muerde: coincidencias solapadas.*** *En* `"aaa"` *buscando* `"aa"`*, ¿es una coincidencia o dos (con solape)? Decidilo y documentalo. La convención más común es* ***no solapar****: consumís la* `needle` *completa y seguís* ***después*** *de ella. Definí tu comportamiento y respetalo.*

Identificadores sugeridos · `sv_replace_all` , `needle` , `with` , `out` , `outCap` , `written` , `i` **Criterios de aceptación**

- [ ] Reemplaza **todas** las apariciones (no solo la primera).
- [ ] Nunca escribe más de `outCap` bytes.
- [ ] Reusa `sv_find` (Parte 6) o `sv_index_of` (Parte 2) para ubicar las coincidencias.
- [ ] Documenta qué hace con las coincidencias solapadas.

PARTE 6 — El catálogo completo: comparar, buscar, construir

Ya tenés el esqueleto. Esta parte **redondea** la librería con las operaciones que faltan para que sea de verdad usable. Ninguna es difícil, pero todas son **indispensables**.

6.1 Comparar sin `strcmp`: `sv_eq`

Comparar dos vistas es distinto que comparar dos cadenas C. Con cadenas C, `strcmp` camina hasta el `'\0'` de cada una. Con vistas, vos **ya sabés** los largos: podés hacer dos chequeos rápidos.

La idea del contrato de `sv_eq`:

1. Si los `count` son **distintos** → ya son distintas (no compares nada más: un cortocircuito gratis).
2. Si los `count` son **iguales** → recién ahí comparás byte a byte (con `memcmp` o con un bucle).

Esto es más rápido y más seguro que `strcmp`, y funciona con textos que no terminan en `'\0'`.

> ***Ojo con el `'\0'` embebido.*** *`strcmp("a\0b", "a\0c")` diría "iguales" (para en el primer `\0`). `sv_eq` los ve **distintos**, porque compara los `count` (3 vs 3) y byte a byte (el `'\0'` del medio también cuenta). Ese es un beneficio: tu librería sirve para **datos**, no solo para texto.*

Y los "empieza con" / "termina con" (`starts_with` / `ends_with`) son comparaciones **parciales**: dicen si el **prefijo** o el **sufijo** de una vista coincide con otra. Se usan a montones: reconocer una extensión (`.txt`), una etiqueta (`http://`), un comentario (`#`).

6.2 Buscar una subcadena: `sv_find` y `sv_contains`

`sv_index_of` busca **un** carácter. `sv_find` busca una **subcadena** completa: "¿en qué posición aparece `needle` dentro de `haystack`?". Es el "buscar" de un editor.

La forma directa: en cada posición `i` desde donde todavía quepa `needle`, compará `needle` contra la sub-vista que arranca en `i` (¡reusás `sv_left` / `sv_eq`!). Si coincide, devolvés `i`. Si no aparece en ninguna, `-1`.

```
haystack: b a n a n a
needle  : a n a
posiciones a probar: i=0..3   (no más allá: no queda lugar)
  i=0: "ban" vs "ana" → no
  i=1: "ana" vs "ana" → ¡SÍ! → devuelve 1
```

`sv_contains` es solo `sv_find(...) != -1` : la pregunta "¿está?" como `int` (booleano).

6.3 Construir y cortar: `sv_from_parts` , `sv_take` / `sv_drop`

Para cerrar la librería, dos familias que vas a usar todo el tiempo:

- `sv_from_parts(data, count)` : envuelve un puntero y un largo **sin llamar a** `strlen` . Es el equivalente "crudo" de `sv_from_cstr` , y la forma de crear una vista desde un **tramo** de memoria que ya conocés (por ejemplo, de un `malloc` , o de un archivo leído con `fread`).
- `sv_take` / `sv_drop` : la pareja canónica de los *slice*. `sv_take(sv, n)` = "quedate con los primeros `n` " (es `sv_left`). `sv_drop(sv, n)` = "avanzá `n` , tirando lo del frente" y devolvé el resto. Con estas dos, "moverse por el texto" se vuelve una receta: **tomá lo que quieras, tirá el resto**.

> ***Por qué*** `from_parts` ***es clave.*** *Cuando leas un archivo entero a memoria (capítulo* `15` *) y quieras partirlo en líneas, tenés el puntero y el largo* ***en la mano****: no hay* `'\0'` *ni razón para llamar a* `strlen` *.* `sv_from_parts(buf, n)` *te deja* ***toda*** *la librería disponible sobre datos crudos, sin copiar y sin perder tiempo midiendo.*

Práctica de la Parte 6

Ejercicio 18.13 — Comparaciones (`sv_eq` , `sv_starts_with` , `sv_ends_with`)

Conceptos · cortocircuito por `count` , comparación byte a byte, prefijo/sufijo

Contexto · Un clasificador de archivos decide por la extensión; un enrutador, por el prefijo de una URL; un intérprete, por la primera palabra. Simplificación: tres comparaciones sobre vistas.

Consigna · Implementá:

1. `sv_eq(a, b)` : devuelve distinto de cero si son **iguales**. Cortocircuito: si los `count` difieren, ya son distintas; si no, compará byte a byte.
2. `sv_starts_with(sv, prefix)` : ¿ `sv` **empieza** con `prefix` ?
3. `sv_ends_with(sv, suffix)` : ¿ `sv` **termina** con `suffix` ?

Probá casos verdaderos y falsos, incluido el `prefix` / `suffix` **más largo** que la vista (debe dar falso, sin romperse) y el caso vacío.

Entrada · Pares de vistas. **Salida** · El veredicto de cada comparación.

Escenario de prueba

```
sv_eq("hola", "hola")            → sí
sv_eq("hola", "hola ")           → no     (count distinto: 4 vs 5)
sv_eq("hola", "HOla")            → no     (distingue mayúsculas)

sv_starts_with("informe.txt", "inf")  → sí
sv_starts_with("informe.txt", "txt")  → no
sv_starts_with("txt", "informe")      → no  (prefijo más largo que la vista)

sv_ends_with("informe.txt", ".txt")   → sí
sv_ends_with("informe.txt", ".doc")   → no
```

Identificadores sugeridos · `sv_eq` , `sv_starts_with` , `sv_ends_with` **Criterios de aceptación**

- [ ] `sv_eq` cortocircuita por `count` antes de comparar bytes.
- [ ] Las comparaciones son **sensibles a mayúsculas**.
- [ ] `prefix` / `suffix` más largos que la vista devuelven **falso** sin leer fuera de límites.
- [ ] El caso vacío (`count == 0`) se maneja bien.

Ejercicio 18.14 — Búsqueda de subcadena (`sv_find` , `sv_contains`)

Conceptos · búsqueda subcadena, reusar `sv_eq` /sub-vistas, centinela, complejidad

Contexto · "¿Este log contiene `ERROR` ?", "¿en qué columna queda la clave?" — buscar subcadenas está por todos lados. Simplificación: ubicar la primera aparición de una aguja en un pajar.

Consigna · Implementá:

1. `sv_find(haystack, needle)` : devuelve el índice de la **primera** aparición de `needle` en `haystack` , o `-1` si no aparece. Reusá `sv_eq` comparando cada sub-vista candidata (o `memcmp`).
2. `sv_contains(haystack, needle)` : devuelve "verdadero" si `needle` está.

Casos a cubrir: coincidencia al principio, en el medio, al final, ausente, `needle` vacío (¿da 0?), `needle` más largo que `haystack` (debe dar `-1`).

Entrada · Pajar y aguja (vistas). **Salida** · El índice encontrado, o `-1` ; y el veredicto de `contains` .

Escenario de prueba

```
sv_find("banana", "ana")   → 1     (la primera)
sv_find("banana", "ban")   → 0     (al principio)
sv_find("banana", "na")    → 2
sv_find("banana", "xyz")   → -1
sv_find("abc", "abcd")     → -1    (aguja más larga que el pajar)
sv_find("abc", "")         → 0     (aguja vacía: decidí y documentá)

sv_contains("hola mundo", "mundo") → sí
sv_contains("hola mundo", "Mundo") → no
```

Identificadores sugeridos · `sv_find` , `sv_contains` , `haystack` , `needle` , `i`
Criterios de aceptación

- [ ] El recorrido **no** prueba posiciones donde la aguja ya no entra.
- [ ] Reusa `sv_eq` (o `memcmp`) para comparar; no reimplementa la igualdad.
- [ ] Devuelve la **primera** aparición; `-1` si no está.
- [ ] El caso "aguja vacía" está decidido y documentado.

Ejercicio 18.15 — Construir y cortar (`sv_from_parts` , `sv_take` , `sv_drop`)

Conceptos · construir sin `strlen` , *take*/*drop*, avanzar la vista

Contexto · Al partir un archivo entero en memoria tenés el puntero y el largo: no querés medir de nuevo. Y para "caminar" por el texto, querés las dos operaciones de *slice*. Simplificación: construir una vista "cruda" y moverse por ella.

Consigna · Implementá:

1. `sv_from_parts(const char *data, size_t count)` : envuelve puntero y largo **sin** `strlen` .
2. `sv_take(sv, n)` : devuelve los primeros `n` (podés reusar `sv_left`).
3. `sv_drop(sv, n)` : **avanza** la vista `n` bytes (tirando el frente) y devuelve el resto (protegé `n > count`).

Demostralo con un texto: creá la vista con `from_parts` , hacé un `take` de la primera palabra y un `drop` para saltarla, e imprimí los pedazos.

Entrada · Un puntero y un largo (o un literal), y un `n` . **Salida** · Los pedazos pedidos.

Escenario de prueba

```
sv = sv_from_parts("hola mundo", 10)     (sin llamar a strlen)
  sv_take(sv, 4)  → "hola"
  sv_drop(sv, 5)  → "mundo"    (saltea "hola " — 5 bytes)
  sv_drop(sv, 100)→ ""         (protegido: no da la vuelta)

  además: sv_take(sv, 4) seguido de sv_left(sv,4) dan lo mismo  (coherencia)
```

Identificadores sugeridos · `sv_from_parts` , `sv_take` , `sv_drop` , `n` **Criterios de aceptación**

- ☐ `sv_from_parts` **no** llama a `strlen` .
- ☐ `sv_take` coincide con `sv_left` .
- ☐ `sv_drop` con `n > count` deja la vista vacía (sin resta que dé la vuelta).
- ☐ Los pedazos apuntan al texto original (no hay copias).

Ejercicio 18.16 — El ejercicio jefe: un parser de configuración

Conceptos · todo: vistas, `trim` , `chop` , `index_of` , `eq` , `starts_with` , sin `malloc`

Contexto · Un archivo de configuración estilo `INI` tiene líneas `clave = valor` , comentarios que empiezan con `#` o `;` , y líneas en blanco que se ignoran. Simplificación: un parser que lea el contenido y muestre cada par `clave = valor` ya limpio, **sin reservar**

memoria.

Consigna · Escribí un programa que:

1. Lea la entrada **completa** de `stdin` a un búfer acotado (por ejemplo 4 KB) leyendo con `fread` o `fgets` en bucle (capítulos `14` / `15`).
2. Cree una vista del contenido con `sv_from_parts` .
3. Recorra la vista **línea por línea** (usá `\n` como delimitador de `chop`).
4. Por cada línea: `sv_trim` ; si queda **vacía**, ignorala; si **empieza con** `#` o `;` , es un comentario (ignorala o mostrala aparte).
5. Partila en `clave` y `valor` con `sv_chop_by_delim(line, '=')` ; recortá **ambos** con `sv_trim` .
6. Imprimí cada par como `clave` → `valor` , separando la salida normal de los avisos (líneas raras) por `stdout` / `stderr` .
7. Al final, mostrá **cuántas** claves válidas se leyeron.

Entrada · Un archivo INI por `stdin` (con comentarios, líneas en blanco y espacios de más).
Salida · Los pares `clave → valor` , más un resumen.

Escenario de prueba

```
entrada (stdin):
  # configuración del servidor
  host = 127.0.0.1

  ; puerto por defecto
  port=   8080
  nombre = Servidor de Pruebas

salida (stdout):
  host → 127.0.0.1
  port → 8080
  nombre → Servidor de Pruebas
  claves válidas: 3
```

```
entrada (stdin):
  sin igual aca

salida:
```

```
(stderr) línea sin '=' (ignorada): "sin igual aca"
claves válidas: 0
```

Identificadores sugeridos · `sv` , `line` , `key` , `value` , `validCount` , `buffer` , `BUFFER_SIZE` **Restricciones/pistas**

- **Prohibido** `malloc` y prohibido `strtok` (rompe el texto original metiendo `'\0'`). Todo el parseo sale de tu librería.
- Reusá lo que ya escribiste: `sv_from_parts` , `sv_chop_by_delim` , `sv_trim` , `sv_starts_with` , `sv_eq` .
- Acotá **siempre** el búfer de lectura y detectá si el archivo no entra.
- La vista del contenido es de **solo lectura**: no modifiques el búfer mientras parseás.
- Ojo con una línea con **más de un** `=` : ¿te quedás con la primera parte como clave y el resto como valor? Decidilo y documentalo.
- Avisos por `stderr` , resultado por `stdout` .

Criterios de aceptación

- [ ] **No** hay `malloc` ni `strtok` en ninguna parte.
- [ ] El contenido se envuelve con `sv_from_parts` (sin `strlen`).
- [ ] Se recorre línea por línea con `sv_chop_by_delim(sv, '\n')` .
- [ ] Cada línea se recorta con `sv_trim` ; las vacías y los comentarios se ignoran.
- [ ] Clave y valor se recortan por separado.
- [ ] Una línea sin `=` se reporta por `stderr` y **no** rompe el parser.
- [ ] El conteo de claves válidas coincide con el escenario.
- [ ] El búfer de lectura está acotado y no hay desbordamientos.

18.17 Errores comunes

- **Recorrer sin acotar por `count`.** Tentador usar `while (*p) ...`, pero tu vista **no** promete un `'\0'`. **Siempre** iterá de `0` a `count - 1`.
- **Devolver `-1` en un `size_t`.** `size_t` no tiene negativos: el "no encontrado" va en un tipo **con signo** (`int`, `ptrdiff_t`).
- **Restar restar con `size_t` cuando `n > count`.** `count - n` da la vuelta y se vuelve gigante. Chequeá `n <= count` **antes** de restar.
- **Usar `%s` con una vista.** `%s` camina hasta un `'\0'` que puede no existir. Usá `%.*s` (`SV_Fmt` / `SV_Arg`).
- **Pasar `char` con signo a `isspace`.** Un byte `>= 0x80` se vuelve negativo → comportamiento indefinido. Pasá `(unsigned char)c`.
- **Modificar memoria de solo lectura.** Un literal (`"..."`) vive en `.rodata`: escribir ahí es *Segmentation Fault*. Tu vista es `const` por eso.
- **Escribir a `data[count - 1]` con `count == 0`.** Leer/escribir "un byte antes del inicio" es acceso fuera de límites. Chequeá el largo **antes** de tocar el último byte.
- **Confundir trim con "sacar todos los blancos".** `trim` toca **solo los bordes**; los internos quedan.
- **Reemplazar sin controlar `outCap`.** Un reemplazo que **crece** el texto desborda el búfer si no chequeás el tamaño. Sé el guardián.
- **Usar `strtok`.** Rompe el texto original (mete `'\0'`) y no es *reentrante*. Con `chop` no lo necesitás.
- **Casos solapados en `replace_all`** (`"aaa"` con `"aa"`): definí la convención y documentala.
- **`sv_find` con aguja vacía**: decidí (y documentá) si devuelve `0`, `count`, o es un error.

18.18 Cierre

Si tuvieras que quedarte con **una sola idea**, que sea esta: **un texto es una calle de bytes, y una `String_View` es una ventana apoyada sobre esa calle**. Todo lo demás se deduce de ahí.

- Dejaste de depender del `'\0'`: en vez de caminar hasta el cartel, **guardás el largo** (`count`) y listo.
- Aprendiste a **mirar y cortar sin copiar**: mover `data` y ajustar `count` es toda tu librería. Por eso funciona **sin** `malloc` y a velocidad de vértigo.
- Diseñaste **contratos** claros: quién pasa por valor, quién por puntero, dónde va cada centinela, dónde vive el resultado de un reemplazo (en **tu** búfer, nunca en la vista).
- Escribiste una **librería completa**: conversión, impresión, búsqueda, sub-vistas, recorte, corte por delimitador, reemplazo, comparación y un parser de configuración **sin reservar un solo byte**.

Resumen de la caja de herramientas.

Quiero…	Función	Idea en una frase
Convertir una cadena C	`sv_from_cstr`	Mide **una vez** con `strlen` y envuelve.
Construir sin medir	`sv_from_parts`	Envolvé puntero + largo que ya tenés.
Imprimir sin `'\0'`	`SV_Fmt` / `SV_Arg`	`%.*s`: decile a `printf` cuánto mirar.
Buscar un carácter	`sv_index_of`	Índice, o `-1`.
Buscar una subcadena	`sv_find` / `sv_contains`	Comparar la aguja en cada posición.
Quedarme con el principio	`sv_left` / `sv_take`	Misma `data`, `count` más chico.
Quedarme con el final	`sv_right`	Mover `data`, ajustar `count`.
Saltear el frente	`sv_drop` / `sv_chop_left`	Avanzar `data`.
Sacar blancos	`sv_trim_left` / `_right` / `sv_trim`	Ajustar inicio y/o largo.
Partir por delimitador	`sv_chop_by_delim`	Devolver el trozo y **avanzar** la vista.
Comparar	`sv_eq`	Si `count` difiere, ya son distintas.
Prefijo / sufijo	`sv_starts_with` / `sv_ends_with`	Comparación parcial.
Reemplazar	`sv_replace_char` / `sv_replace_all`	Escribir en **tu** búfer, sin desbordar.
Hacia cadena C	`sv_to_cstr`	Copiar y **agregar** el `'\0'`.

> ***El puente a lo que sigue.*** *Ahora que tenés vistas, el texto deja de ser un enemigo. Con la misma idea (puntero + largo) podés construir un* ***arena*** *de strings (capítulo* `12` *), un* ***tokenizador*** *de un lenguaje, o el parser de una línea de comandos sin reservar memoria. Y cuando una función vieja te pida una cadena C, sabés cerrar el círculo:* `sv_to_cstr` *y el* `\0` *vuelven a escena, pero* ***solo cuando vos querés****. La calle y la ventana ahora las manejás vos.*