

Introducción a la Computación

Arreglos en funciones y cadenas

Direcciones, `'\0'`, `fgets` y buffer overflows

Gerson Garrido

Semana 10

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

1. Pasar un arreglo a una función: se pasa su dirección, y el tamaño va aparte
2. Mantener un vector: insertar y eliminar
3. Qué es una cadena: un arreglo de `char` terminado en `'\0'`
4. Recorrer una cadena. `my_strlen` y los bytes de la `ñ`
5. Leer texto con límite: `%19s` y `fgets`
6. Buffer overflow: leer el informe y por qué es un problema de seguridad

> **Lectura**
>
> Apuntes de la semana 10. Código en `code/week-10`. Hoy las cadenas se recorren a mano: `string.h` llega la próxima semana.

SECCIÓN 01

Arreglos y funciones

01

Pasar un arreglo a una función

sum_array.c

```c
int sum(int a[], int n) {
    int total = 0;
    for (int i = 0; i < n; i++) {
        total = total + a[i];
    }
    return total;
}
...
int grades[N] = {14, 9, 17, 11, 20};
int total = sum(grades, N);
```

- `int a[]`: `a` es un arreglo de `int`. Corchetes vacíos.
- Al llamar: solo el nombre, sin corchetes.
- La longitud va en **otro parámetro**.

terminal

```
$ build/sum_array
Suma de las 5 notas: 71
Promedio: 14.20
Suma de las 3 primeras: 40
```

¿Y si la función usa `sizeof`?

errors/sizeof_param.c

```c
void print_size(int a[]) {
    printf("en la función: %zu bytes
     \n", sizeof(a));
}
```

terminal

```
$ build/sizeof_param
en main: 20 bytes
en la función: 8 bytes
con [5] en el parámetro: 8 bytes
```

gcc avisa

```
warning: ‘sizeof’ on array function parameter ‘a’ will return size of ‘int *’
```

- `int *` se lee “dirección de un `int`”: un **puntero** (*(pointer)*). Ocupa 8 bytes en este entorno; C no fija ese número.
- En `main`: 5 `int` de 4 bytes en este entorno, 20 bytes.

Lo que viaja es la dirección

Idea clave

La función recibe **la dirección donde empieza**, no el arreglo. Una dirección no dice cuántos elementos vienen después. Por eso la longitud va aparte, y un número entre los corchetes del parámetro no cambia nada.

Una función sí puede cambiar el arreglo

bonus.c

```c
void add_one(int x) {
    x = x + 1;
}

void add_bonus(int a[], int n, int bonus) {
    for (int i = 0; i < n; i++) {
        a[i] = a[i] + bonus;
    }
}
```

Con `grade` = 14 y `grades` = 14, 9, 17:

terminal

```
$ build/bonus
grade: 14
grades: 16 11 19
```

Una función sí puede cambiar el arreglo

bonus.c

```c
void add_one(int x) {
    x = x + 1;
}

void add_bonus(int a[], int n, int bonus) {
    for (int i = 0; i < n; i++) {
        a[i] = a[i] + bonus;
    }
}
```

Con `grade` = 14 y `grades` = 14, 9, 17:

terminal

```
$ build/bonus
grade: 14
grades: 16 11 19
```

Se llama con	Qué se copia	Qué toca la función
`add_one(grade)`	el valor 14	su propia copia
`add_bonus(grades, N, 2)`	la dirección de inicio	la memoria de `grades`

Una función sí puede cambiar el arreglo

bonus.c

```c
void add_one(int x) {
    x = x + 1;
}

void add_bonus(int a[], int n, int bonus) {
    for (int i = 0; i < n; i++) {
        a[i] = a[i] + bonus;
    }
}
```

Con `grade` = 14 y `grades` = 14, 9, 17:

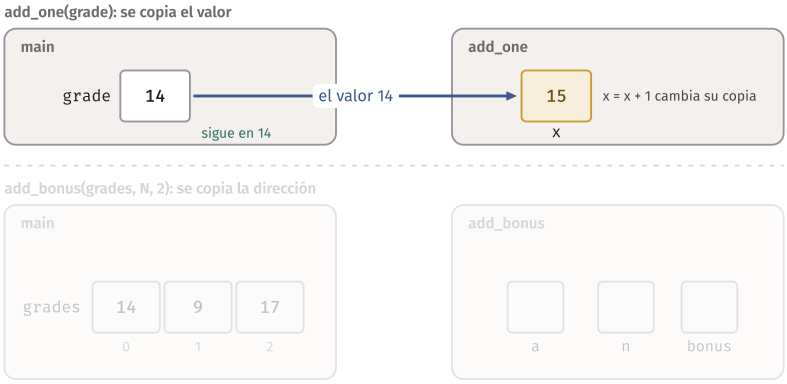
terminal

```
$ build/bonus
grade: 14
grades: 16 11 19
```

Se llama con	Qué se copia	Qué toca la función
`add_one(grade)`	el valor 14	su propia copia
`add_bonus(grades, N, 2)`	la dirección de inicio	la memoria de `grades`

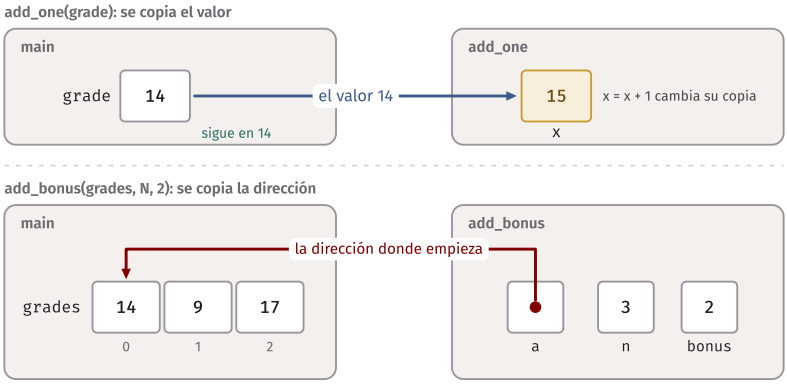
C sigue pasando por valor, siempre. Lo que se copia es una dirección.

Se copia el valor, o se copia la dirección



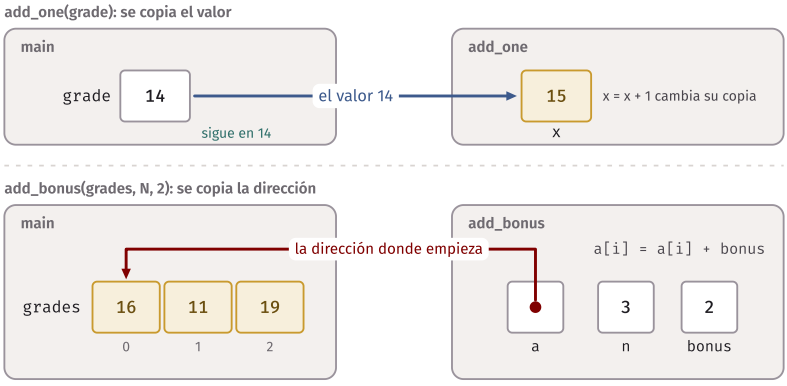
grade no cambió: la función trabajó sobre su propia copia.

Se copia el valor, o se copia la dirección



a es una copia de la dirección, y lleva a los mismos elementos de main. No hay un segundo arreglo.

Se copia el valor, o se copia la dirección



grades cambió: 16 11 19. C siguió pasando por valor, y el valor copiado era una dirección.

```
grade: 14    grades: 16 11 19
```

Comprobémoslo con %p

terminal

```
$ build/same_address
en main, grades:  0xffff8c500020
en la función, a: 0xffff8c500020
```

- Las direcciones cambian con cada máquina y cada corrida. Importa que las dos líneas muestran la misma.
- `a[i]` se ubica desde el inicio, avanzando `i` veces el tamaño de un elemento. Esa cuenta no sabe dónde termina el arreglo: **nadie revisa el límite**.

Un arreglo no es una dirección: en `main`, `sizeof(grades)` da 20. Solo a la función viaja su dirección.

`const`: prometer que no se cambia

const_param.c

```c
int max_value(const int a[], int n) {
```

La función solo lee los elementos, y se puede llamar con un arreglo normal. Si intenta escribir, **no compila**:

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/const_write.c -o build/const_write
errors/const_write.c: In function ‘reset’:
errors/const_write.c:6:14: error: assignment of read-only location ‘*(a + (sizetype)((long unsigned int)i * 4))’
    6 |         a[i] = 0;
      |              ^
```

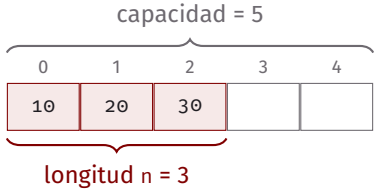
Regla del curso desde hoy: si la función no cambia el arreglo, su parámetro lleva `const`.

SECCIÓN 02

Vectores

02

Capacidad y longitud



Vector

Un arreglo con una **capacidad** fija y una **longitud** `n` que cambia. Los elementos en uso son de `v[0]` a `v[n - 1]`.

Se matricula un estudiante, otro se retira: la capacidad no cambia, `n` sí.

Insertar: abrir un espacio desde el último

pseudocódigo

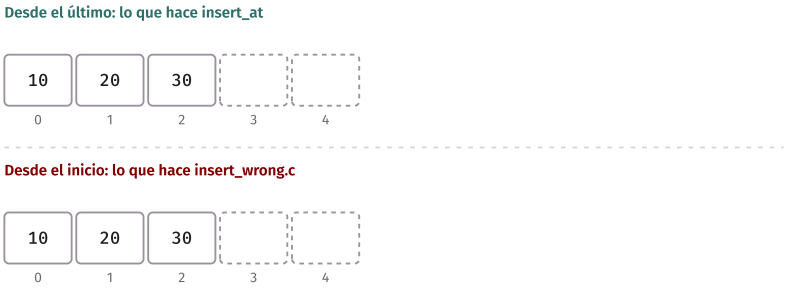
```
algoritmo insertar(v, n, capacidad, pos, valor)
    si n ≥ capacidad o pos < 0 o pos > n
      entonces
        devolver n
    fin si
    i ← n
    mientras i > pos hacer
        v[i] ← v[i - 1]
        i ← i - 1
    fin mientras
    v[pos] ← valor
    devolver n + 1
fin algoritmo
```

vector.c

```c
int insert_at(int v[], int n, int capacity
      , int pos, int value) {
    if (n >= capacity || pos < 0 || pos >
      n) {
        return n;
    }
    for (int i = n; i > pos; i--) {
        v[i] = v[i - 1];
    }
    v[pos] = value;
    return n + 1;
}
```

Devuelve la **longitud nueva**, porque `n` llega por valor. Si no se puede, devuelve la misma. `i--` resta 1 a `i`.

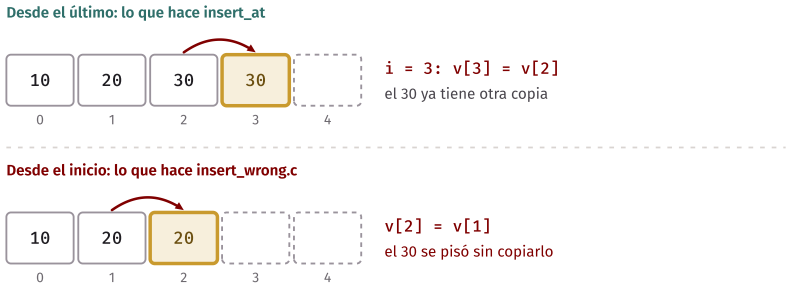
Insertar 99 en la posición 1



Vector 10, 20, 30, con capacidad 5. Las casillas punteadas no se usan.

Por un momento hay valores repetidos. Es normal: cada valor se copió antes de que su casilla se pisara.

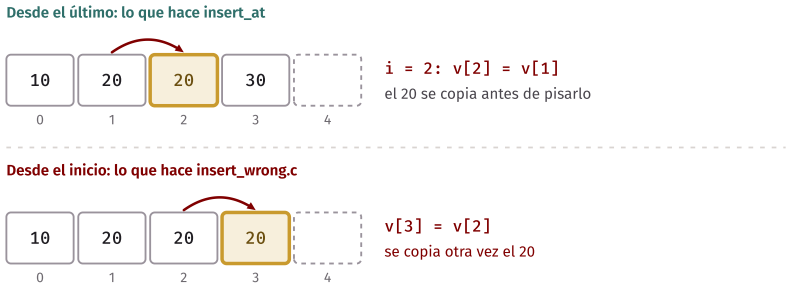
Insertar 99 en la posición 1



El primer movimiento decide: arriba se abre el espacio al final, abajo se pisa el 30.

Por un momento hay valores repetidos. Es normal: cada valor se copió antes de que su casilla se pisara.

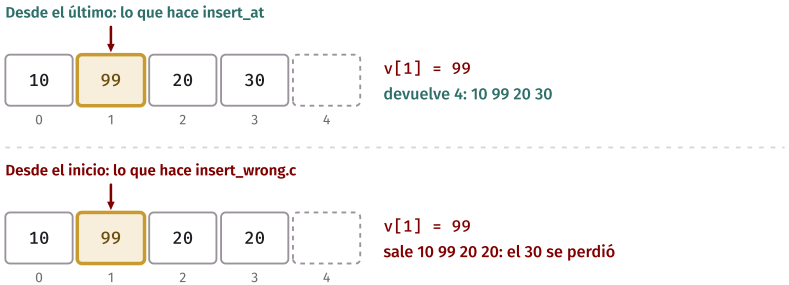
Insertar 99 en la posición 1



Cada valor debe copiarse antes de que su casilla se pise.

Por un momento hay valores repetidos. Es normal: cada valor se copió antes de que su casilla se pisara.

Insertar 99 en la posición 1



El mismo número de pasos, y un solo resultado correcto.

Por un momento hay valores repetidos. Es normal: cada valor se copió antes de que su casilla se pisara.

Si se mueve desde el inicio, se pierde un valor

errors/insert_wrong.c

```c
for (int i = pos; i < n; i++) {
    v[i + 1] = v[i];
}
v[pos] = 99;
```

terminal

```
$ gcc -Wall -Wextra -std=c17
    errors/insert_wrong.c -o
    build/insert_wrong
$ build/insert_wrong
 10 99 20 20
```

Error común

Primero `v[2] = v[1]`, y después `v[3] = v[2]`. Pero `v[2]` ya no tiene el 30: se perdió.

Eliminar: mover desde el primero

vector.c

```c
int remove_at(int v[], int n, int pos) {
    if (pos < 0 || pos >= n) {
        return n;
    }
    for (int i = pos; i < n - 1; i++) {
        v[i] = v[i + 1];
    }
    return n - 1;
}
```

- Ahora se mueve a la izquierda.
- `i < n - 1` evita leer `v[n]`.
- El último sigue en su casilla, pero ya no cuenta.

terminal

```
$ build/vector
n = 3: 10 20 30
n = 4: 10 99 20 30
n = 5: 10 99 20 30 40
n = 5: 10 99 20 30 40
n = 4: 99 20 30 40
n = 3: 99 20 30
n = 3: 99 20 30
```

La cuarta línea: vector lleno. La última: una posición que no existe. No cambia nada.

SECCIÓN 03

Cadenas

03



Una cadena es un arreglo de `char` terminado en `'\0'`

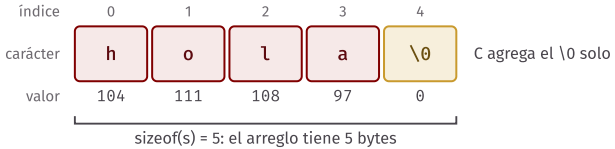
Cadena (*string*)

Un arreglo de `char` que termina en el carácter nulo, `'\0'`, un byte con valor 0. No es un tipo aparte.

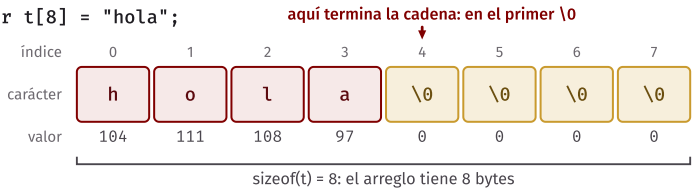
- `"hola"` es un **literal de cadena**. C agrega `'\0'` solo: 4 letras, 5 bytes.
- `'A'` es un carácter, un número. `"A"` es una cadena de 2 bytes.

s y t, byte por byte

```
char s[] = "hola";
```




```
char t[8] = "hola";
```



Las casillas que el texto no alcanza quedan en 0.

La longitud y el tamaño son dos cosas distintas

string_bytes.c

```c
char s[] = "hola";
char t[8] = "hola";
```

terminal

```
$ build/string_bytes
sizeof(s) = 5
sizeof(t) = 8
 104 111 108 97 0 0 0 0
'\0' vale 0 y '0' vale 48
```

- Las casillas que el texto no alcanza quedan en 0.
- La cadena termina en el **primer** `'\0'`.
- `'\0'` vale 0. El dígito `'0'` vale 48.

%s se detiene en el primer '\0'

cut_string.c

```c
s[0] = 'H';
printf("%s\n", s);
s[2] = '\0';
printf("%s\n", s);
```

terminal

```
$ build/cut_string
hola
Hola
Ho
sizeof(s) sigue siendo 5
```

Idea clave

`printf` recibe la dirección de inicio y escribe hasta encontrar un `'\0'`. Los bytes de después siguen ahí; nadie los mira.

Olvidar el espacio para `'\0'`

errors/no_terminator.c

```c
char word[4] = "hola";
printf("%s\n", word);
```

- Es válido en C, y este `gcc` **no avisa**. Pero no es una cadena.
- `printf` sigue leyendo fuera del arreglo: comportamiento indefinido.
- `-g` agrega los números de línea. `-fsanitize=address` agrega revisiones que corren **mientras el programa se ejecuta**.

una corrida, recortada

```
$ build/no_terminator_san
==27466==ERROR: AddressSanitizer: stack-buffer-overflow on address 0xffff89b00024 [...]
READ of size 5 at 0xffff89b00024 thread T0
    #0 0xffff8c8aa4d8 in puts ../../../../src/libsanitizer/sanitizer_common/[...]
    #1 0xaaaaafd60c94 in main errors/no_terminator.c:7
```

Para un texto de `k` bytes hacen falta `k + 1` bytes.

No cambiar un literal

errors/literal_write.c

```c
void set_first(char s[], char c) {
    s[0] = c;
}
...
char copy[] = "hola";
set_first(copy, 'H');
set_first("hola", 'H');
```

- `char copy[] = "hola"` crea **un arreglo propio**, con una copia: se puede cambiar.
- El literal por sí solo no se debe cambiar: es comportamiento indefinido.
- Compila sin advertencias. En esta corrida, el programa terminó con **segmentation fault**; el lenguaje no promete eso.
- Si la función no cambia la cadena, `const`: recibir un literal es seguro.

SECCIÓN 04

Recorrer una cadena

04

La longitud, a mano

pseudocódigo

```
algoritmo longitud(s)
    i ← 0
    mientras s[i] ≠ NULO hacer
        i ← i + 1
    fin mientras
    devolver i
fin algoritmo
```

my_strlen.c

```c
int my_strlen(const char s[]) {
    int i = 0;
    while (s[i] != '\0') {
        i++;
    }
    return i;
}
```

`NULO`: el carácter que termina la cadena.

Para **leer** una cadena basta su dirección. `strlen` hace lo mismo: la próxima semana.

La seguimos con `"sol"`

i	s[i]	¿s[i] != '\0'?
0	s	sí, `i` pasa a 1

La seguimos con `"sol"`

`i`	`s[i]`	`¿s[i] != '\0'?`
0	`s`	sí, `i` pasa a 1
1	`o`	sí, `i` pasa a 2

La seguimos con `"sol"`

i	s[i]	¿s[i] != '\0'?
0	s	sí, i pasa a 1
1	o	sí, i pasa a 2
2	l	sí, i pasa a 3

La seguimos con `"sol"`

i	s[i]	¿s[i] != '\0'?
0	s	sí, i pasa a 1
1	o	sí, i pasa a 2
2	l	sí, i pasa a 3
3	'\0'	no, se devuelve 3

La seguimos con `"sol"`

i	s[i]	¿s[i] != '\0'?
0	s	sí, i pasa a 1
1	o	sí, i pasa a 2
2	l	sí, i pasa a 3
3	'\0'	no, se devuelve 3

- `'\0'` no se cuenta.
- Con `""`, el ciclo no corre ninguna vez: da 0.
- Tarda más cuanto más larga es la cadena: tiene que recorrerla.

UTF-8: se cuentan bytes, no letras

terminal

```
$ build/my_strlen
my_strlen("ano") = 3
my_strlen("año") = 4
```

Índice	0	1	2	3	4
Contenido	a	dos bytes de ñ		o	'\0'
Hexadecimal	61	c3	b1	6f	00

Idea clave

La longitud cuenta **bytes**: `strlen("año")` vale 4, y hacen falta 5 bytes.

Probar letras: solo ASCII

count_lowercase.c

```c
int count_lowercase(const char s[]) {
    int count = 0;
    for (int i = 0; s[i] != '\0'; i++) {
        if (s[i] >= 'a' && s[i] <= 'z') {
            count++;
        }
    }
    return count;
}
```

terminal

```
$ build/count_lowercase
Hola Mundo: 7
LIMA 2026: 0
ano: 3
año: 2
```

- Los bytes de la `ñ` no están entre `'a'` y `'z'`, tenga signo `char` o no.
- Solo entiende ASCII: no reconoce `ñ` ni `á`. En este curso basta.
- Si usan `ctype.h`: `islower((unsigned char) s[i])`. Un `char` negativo sin cast es comportamiento indefinido.

SECCIÓN 05

Leer texto con límite

05

`scanf` con ancho: una palabra

read_word.c

```c
char word[20];
if (scanf("%19s", word) != 1) {
```

- `%19s` para 20 bytes: siempre uno menos.
- Sin `&`: `word` ya entrega la dirección.
- Se detiene en el primer espacio. El resto espera la siguiente lectura.

terminal

```
$ echo "Ana María" | build/read_word
Primera lectura: Ana
Segunda lectura: María
$ echo abcdefghijklmnopqrstuvwxyz | build/read_word
Primera lectura: abcdefghijklmnopqrs
Segunda lectura: tuvwxyz
```

`fgets`: una línea

read_name.c

```
1  if (fgets(s, size, stdin) == NULL) {
```

- Recibe dónde guardar, **el tamaño del buffer** y de dónde leer. `stdin` es la entrada estándar.
- Lee hasta el salto de línea, o hasta `size - 1` bytes. Siempre agrega `'\0'`.
- Se pasa el tamaño tal cual: `fgets` ya reserva el lugar.
- Devuelve `NULL`, "ninguna dirección", si se acabó la entrada.

Buffer

Una zona de memoria reservada para guardar datos que van llegando. Nuestros arreglos de `char` son buffers.

Los cuatro casos de `fgets`, con un buffer de 8 bytes

terminal

```
$ printf 'Ana\n' | build/read_line
leído: "Ana\n"
fgets devolvió NULL
$ printf 'Ana' | build/read_line
leído: "Ana"
fgets devolvió NULL
```

terminal

```
$ printf 'Alejandra\n' | build/
    read_line
leído: "Alejand"
leído: "ra\n"
fgets devolvió NULL
$ printf '' | build/read_line
fgets devolvió NULL
```

- Cabe: **conserva el `\n`**.
- Última línea sin salto: no hay `\n` que guardar.
- No cabe: toma 7 bytes, y la siguiente llamada sigue donde quedó.
- Sin datos: `NULL` de inmediato. Con teclado, el fin de la entrada es `Ctrl+D`.

Quitar el salto y descartar el resto

read_name.c

```c
int remove_newline(char s[]) {
    for (int i = 0; s[i] != '\0'; i++) {
        if (s[i] == '\n') {
            s[i] = '\0';
            return 1;
        }
    }
    return 0;
}

void discard_rest(void) {
    int c = getchar();
    while (c != '\n' && c != EOF) {
        c = getchar();
    }
}
```

terminal

```
$ printf 'Ana María\nLima\n' | build
    /read_name
Nombre: Ana María
Ciudad: Lima
$ printf 'Maria de los Angeles
    Quispe\nLima\n' | build/
    read_name
Nombre: Maria de los Angele
Ciudad: Lima
```

Sin `discard_rest`, la ciudad recibiría el resto del nombre.

Mezclar `scanf` y `fgets`

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/scanf_then_fgets.c -o build/scanf_then_fgets
$ printf '18\nAna\n' | build/scanf_then_fgets
Edad: 18
Nombre:
$ printf '18\nAna\n' | build/age_name
Edad: 18
Nombre: Ana
```

Error común

`scanf("%d", &age)` deja el salto de línea en la entrada, y `fgets` termina al encontrarlo. La corrección, en `age_name.c`: llamar a `discard_rest` después del `scanf`.

Y nunca `gets`: lee una línea sin límite. Se eliminó del lenguaje en C11.

SECCIÓN 06

Buffer overflow

06



El error clásico

errors/overflow_scanf.c

```c
char name[8];
if (scanf("%s", name) != 1) {
```

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/overflow_scanf.c -o build/overflow_san
$ echo Ana | build/overflow_san
Hola, Ana
```

- Compila **sin ninguna advertencia**.
- Con un nombre corto funciona: las pruebas normales pasan.
- Con `Alejandra`: 9 letras y `'\0'` son 10 bytes. `name` tiene 8.
- No se equivocó un índice: **el dato llegó de afuera**.

Leer el informe

una corrida, recortada

```
$ echo Alejandra | build/overflow_san
==27460==ERROR: AddressSanitizer: stack-buffer-overflow on address 0xffff91b00028 [...]
WRITE of size 10 at 0xffff91b00028 thread T0
    #0 0xffff94955f38 in scanf_common ../../../../src/libsanitizer/sanitizer_common/[...]
    [...]
    #3 0xaaaaab740e44 in main errors/overflow_scanf.c:7
  [...]
  This frame has 1 object(s):
    [32, 40) 'name' (line 5) <== Memory access at offset 40 overflows this variable
```

Es un error al correr, no al compilar. Las direcciones y el número de proceso cambian en cada corrida.

Leer el informe

una corrida, recortada

```
$ echo Alejandra | build/overflow_san
==27460==ERROR: AddressSanitizer: stack-buffer-overflow on address 0xffff91b00028 [...]
WRITE of size 10 at 0xffff91b00028 thread T0
    #0 0xffff94955f38 in scanf_common ../../../../src/libsanitizer/sanitizer_common/[...]
    [...]
    #3 0xaaaaab740e44 in main errors/overflow_scanf.c:7
  [...]
  This frame has 1 object(s):
    [32, 40) 'name' (line 5) <== Memory access at offset 40 overflows this variable
```

1. **Qué pasó:** `stack-buffer-overflow`.

Es un error al correr, no al compilar. Las direcciones y el número de proceso cambian en cada corrida.

Leer el informe

una corrida, recortada

```
$ echo Alejandra | build/overflow_san
==27460==ERROR: AddressSanitizer: stack-buffer-overflow on address 0xffff91b00028 [...]
WRITE of size 10 at 0xffff91b00028 thread T0
    #0 0xffff94955f38 in scanf_common ../../../../src/libsanitizer/sanitizer_common/[...]
    [...]
    #3 0xaaaaab740e44 in main errors/overflow_scanf.c:7
  [...]
  This frame has 1 object(s):
    [32, 40) 'name' (line 5) <== Memory access at offset 40 overflows this variable
```

1. **Qué pasó:** `stack-buffer-overflow`.
2. **Qué operación:** `WRITE of size 10`. Es nuestra cuenta.

Es un error al correr, no al compilar. Las direcciones y el número de proceso cambian en cada corrida.

Leer el informe

una corrida, recortada

```
$ echo Alejandra | build/overflow_san
==27460==ERROR: AddressSanitizer: stack-buffer-overflow on address 0xffff91b00028 [...]
WRITE of size 10 at 0xffff91b00028 thread T0
    #0 0xffff94955f38 in scanf_common ../../../../src/libsanitizer/sanitizer_common/[...]
    [...]
    #3 0xaaaaab740e44 in main errors/overflow_scanf.c:7
  [...]
  This frame has 1 object(s):
    [32, 40) 'name' (line 5) <== Memory access at offset 40 overflows this variable
```

1. **Qué pasó:** `stack-buffer-overflow`.
2. **Qué operación:** `WRITE of size 10`. Es nuestra cuenta.
3. **Dónde:** la primera línea con un archivo nuestro, `overflow_scanf.c:7`. La da `-g`.

Es un error al correr, no al compilar. Las direcciones y el número de proceso cambian en cada corrida.

Leer el informe

una corrida, recortada

```
$ echo Alejandra | build/overflow_san
==27460==ERROR: AddressSanitizer: stack-buffer-overflow on address 0xffff91b00028 [...]
WRITE of size 10 at 0xffff91b00028 thread T0
    #0 0xffff94955f38 in scanf_common ../../../../src/libsanitizer/sanitizer_common/[...]
    [...]
    #3 0xaaaaab740e44 in main errors/overflow_scanf.c:7
  [...]
  This frame has 1 object(s):
    [32, 40) 'name' (line 5) <== Memory access at offset 40 overflows this variable
```

1. **Qué pasó:** `stack-buffer-overflow`.
2. **Qué operación:** `WRITE of size 10`. Es nuestra cuenta.
3. **Dónde:** la primera línea con un archivo nuestro, `overflow_scanf.c:7`. La da `-g`.
4. **Qué variable:** `'name'`. Sus 8 bytes son `[32, 40)`: del 32 al 39; el 40 ya no es suyo.

Es un error al correr, no al compilar. Las direcciones y el número de proceso cambian en cada corrida.

Sin el sanitizer, nadie avisa

en esta corrida, en este entorno

```
$ gcc -Wall -Wextra -std=c17 errors/overflow_scanf.c -o build/overflow_plain
$ echo Alejandra | build/overflow_plain
Hola, Alejandra
```

Comportamiento indefinido

Puede pasar cualquier cosa, y puede no notarse. No se puede esperar lo mismo en otra máquina ni con otro compilador.

- Parece funcionar, y **eso es lo peor que puede pasar**.
- Los bytes de más cayeron en memoria que no era de `name`.
- ¿Qué había ahí?

Una variable que nadie cambió

errors/overflow_neighbor.c

```c
int is_admin = 0;
char name[8];
...
if (is_admin != 0) {
```

en esta corrida, en este entorno

```
$ echo Ana | build/overflow_neighbor
Hola, Ana
Acceso normal
$ echo Alejandra_Quispe | build/overflow_neighbor
Hola, Alejandra_Quispe
Acceso de administrador (is_admin = 1701868393)
```

Idea clave

La entrada puede cambiar datos que nadie le dio permiso de cambiar. C no dice dónde se guarda cada variable, así que en otra máquina puede salir otra cosa. Lo seguro es que el programa está mal.

Por qué es un problema de seguridad

- Quien escribe la entrada **elige los bytes** que caen fuera del buffer.

Por qué es un problema de seguridad

- Quien escribe la entrada **elige los bytes** que caen fuera del buffer.
- En muchos sistemas, la pila también guarda la dirección a la que debe volver la función. Pisarla con un valor elegido puede desviar el programa.

Por qué es un problema de seguridad

- Quien escribe la entrada **elige los bytes** que caen fuera del buffer.
- En muchos sistemas, la pila también guarda la dirección a la que debe volver la función. Pisarla con un valor elegido puede desviar el programa.
- Un error de programación se convierte en una forma de tomar control de un programa, o de la computadora donde corre.

Por qué es un problema de seguridad

- Quien escribe la entrada **elige los bytes** que caen fuera del buffer.
- En muchos sistemas, la pila también guarda la dirección a la que debe volver la función. Pisarla con un valor elegido puede desviar el programa.
- Un error de programación se convierte en una forma de tomar control de un programa, o de la computadora donde corre.
- El gusano Morris de 1988 usaba un buffer overflow en un programa de red que leía con `gets`.

Por qué es un problema de seguridad

- Quien escribe la entrada **elige los bytes** que caen fuera del buffer.
- En muchos sistemas, la pila también guarda la dirección a la que debe volver la función. Pisarla con un valor elegido puede desviar el programa.
- Un error de programación se convierte en una forma de tomar control de un programa, o de la computadora donde corre.
- El gusano Morris de 1988 usaba un buffer overflow en un programa de red que leía con `gets`.
- Hoy los sistemas agregan defensas que dificultan estos ataques, pero no reparan el programa.

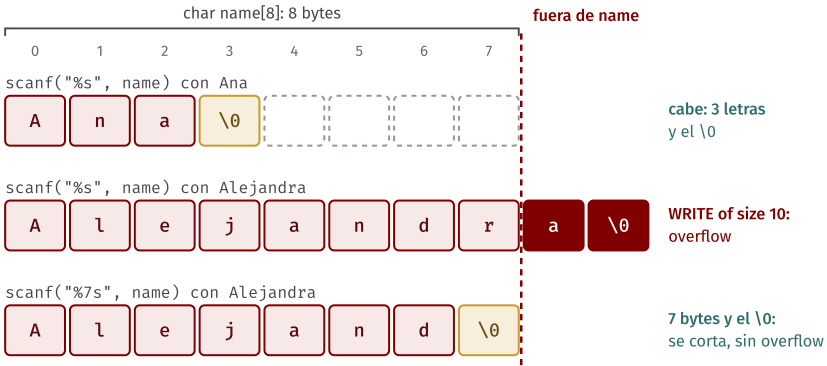
Por qué es un problema de seguridad

- Quien escribe la entrada **elige los bytes** que caen fuera del buffer.
- En muchos sistemas, la pila también guarda la dirección a la que debe volver la función. Pisarla con un valor elegido puede desviar el programa.
- Un error de programación se convierte en una forma de tomar control de un programa, o de la computadora donde corre.
- El gusano Morris de 1988 usaba un buffer overflow en un programa de red que leía con `gets`.
- Hoy los sistemas agregan defensas que dificultan estos ataques, pero no reparan el programa.

La regla

Toda escritura en un buffer lleva un límite, y ese límite viene del tamaño real del buffer.

Con límite, lo que no cabe se queda afuera



Las casillas punteadas son de name, pero nadie les escribió.

Nuestras funciones también reciben el tamaño

copy_text.c

```c
void copy_text(char dest[], const char src
     [], int size) {
    int i = 0;
    while (src[i] != '\0' && i < size - 1) {
        dest[i] = src[i];
        i++;
    }
    dest[i] = '\0';
}
```

En un arreglo de 5 bytes, con `"sol"`, `"luna"` y `"estrella"`:

terminal

```
$ build/copy_text
sol
luna
estr
```

`"luna"` cabe justo: 4 letras y `'\0'`. Es el borde que hay que probar.

Al corregir, `%7s` lee una palabra; `fgets` cambia el significado: lee una línea completa.

SECCIÓN 07

El laboratorio

07

`my_strcmp`: hasta la primera diferencia

pseudocódigo

```
algoritmo comparar(a, b)
    i ← 0
    mientras a[i] ≠ NULO y a[i] = b[i] hacer
        i ← i + 1
    fin mientras
    devolver a[i] - b[i]
fin algoritmo
```

a	b	**Signo**
`"ana"`	`"ana"`	0
`"ana"`	`"ano"`	negativo
`"an"`	`"ana"`	negativo

- Solo importa el **signo**. Sin incluir `string.h`.
- En C, resten como `unsigned char`: un byte de la `ñ` puede verse negativo en una máquina y positivo en otra.
- Orden por códigos: `"Z"` va antes que `"a"`.
- `to_upper`: sin `const`. De `'a'` a `'z'`, resten `'a' - 'A'`. La `ñ` no cambia.

`is_palindrome`: decidir qué significa “igual”

Función	Regla	`Ana`	`Anita lava la tina`
`is_palindrome`	byte por byte, sin excepciones	no	no
`is_palindrome_text`	ignora espacios y mayúsculas de `a` a `z`	sí	sí

- Dos índices, uno en cada extremo, que avanzan hacia el centro.
- En un espacio, se mueve solo ese índice y se vuelve a preguntar.
- Bordes: la cadena vacía y una de solo espacios **son** palíndromos.

Consecuencia de los bytes

`"aña"` no es palíndromo para `is_palindrome`: en bytes es `61 c3 b1 61`, y `c3` no es `b1`.

Provocar un buffer overflow y corregirlo

1. `lab/greet.c` viene con 8 bytes y `%s` sin límite, a propósito.
2. Compilen con `make build/lab/greet` y escriban un nombre de más de 7 letras.
3. Lean el informe: qué pasó, qué operación, qué línea, qué variable.
4. Compílenlo sin `-fsanitize`, prueben el mismo nombre y comparen.
5. Corríjanlo con `fgets`, y quiten el `\n` recorriendo a mano.

Cómo comprobar

`make test-lab`. `Ana Li` debe salir completo. Un nombre largo debe salir cortado y sin informe.

Para la próxima clase

- Hoy, en el laboratorio: `my_strings.c` y `greet.c`.
- Leer los apuntes de la semana 11: la biblioteca de cadenas y matrices.
- Nunca `%s` sin límite. Revisen siempre el resultado de `scanf` y `fgets`.

Preguntas

Dibujen `char s[8] = "hola";`. ¿Cuánto valen su longitud y `sizeof(s)`?
¿Por qué una función puede cambiar un arreglo recibido, pero no un `int`?
¿Por qué `strlen("año")` vale 4?