

UNIVERSIDAD NACIONAL DE INGENIERÍA

Introducción a la Computación

Dentro de la computadora

CPU, memoria, sistema operativo, direcciones y arreglos

Gerson Garrido

Semana 9

Hoy

1. Las piezas: CPU, RAM y almacenamiento
2. El ciclo de instrucción: una suma, paso a paso
3. La jerarquía de memoria y la cache
4. Qué hace el sistema operativo
5. Direcciones: `&` y `%p`
6. Arreglos: declarar, recorrer y cómo se guardan
7. Salirse del arreglo y `-fsanitize=address`

> **Lectura**
>
> Apuntes de la semana 9. Código en `code/week-09`. Hoy los arreglos se usan solo dentro de `main`.

SECCIÓN 01

Las piezas

01

Tres piezas

Pieza	Qué hace	¿Conserva?
CPU	Ejecuta instrucciones, una tras otra	No
RAM	Guarda los programas que corren y sus datos	No
Almacenamiento	Guarda archivos a largo plazo	Sí

Una biblioteca

La persona que estudia es la CPU. Su mesa es la RAM: a la mano, pero no cabe todo. Los estantes son el almacenamiento: mucho espacio, pero ir hasta ellos toma tiempo.

El almacenamiento es un SSD o un disco duro. Para correr un programa, el ejecutable se copia de ahí a la RAM. “¿Conserva?” quiere decir: al apagar.

Dentro de la CPU

- **Registros** *(registers)*: unas pocas casillas de memoria, las más rápidas que hay.
- **Unidad aritmética y lógica**: suma, resta y compara. Hecha de compuertas, como el sumador de la semana 4.
- **Contador de programa** *(program counter)*: guarda la dirección de la siguiente instrucción.

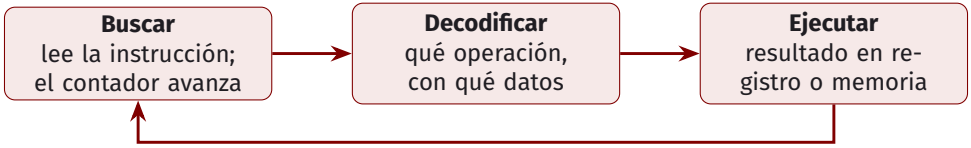
> **Recuerden, de la semana 1**
>
> La CPU solo ejecuta **código máquina**: instrucciones muy simples, guardadas como números en la memoria.

SECCIÓN 02

El ciclo de instrucción

02

Buscar, decodificar, ejecutar



Idea clave

La CPU no entiende el programa completo. Hace algo muy simple, y lo repite: en una computadora actual, miles de millones de veces por segundo.

Es un **modelo**: las CPU modernas adelantan varias instrucciones a la vez, pero en estos programas el resultado es el mismo que si fueran de una en una.

Una línea de C son varias instrucciones

add_two.c

```c
int a = 2;
int b = 3;
int c = a + b;
```

`-O0`: no optimizar, para que la suma no desaparezca.

build/add_two.s, en ARM

```
$ gcc -S -O0 add_two.c -o build/add_two.s

ldr w1, [sp, 28]
ldr w0, [sp, 24]
add w0, w1, w0
str w0, [sp, 20]
```

Solo `int c = a + b;`. En x86-64 se ve distinto.

Cómo leerlo

`w0` y `w1` son registros. `[sp, 28]` es un lugar de la memoria: una base más una distancia.

Cada fila es una vuelta del ciclo

Instrucción	Qué pide	w1	w0	c en memoria
ldr w1, [sp, 28]	Cargar a a w1	2	?	?

Cada fila es una vuelta del ciclo

Instrucción	**Qué pide**	`w1`	`w0`	`c` en memoria
`ldr w1, [sp, 28]`	Cargar `a` a `w1`	2	?	?
`ldr w0, [sp, 24]`	Cargar `b` a `w0`	2	3	?

Cada fila es una vuelta del ciclo

Instrucción	**Qué pide**	`w1`	`w0`	`c` en memoria
`ldr w1, [sp, 28]`	Cargar `a` a `w1`	2	?	?
`ldr w0, [sp, 24]`	Cargar `b` a `w0`	2	3	?
`add w0, w1, w0`	Sumar y dejar en `w0`	2	5	?

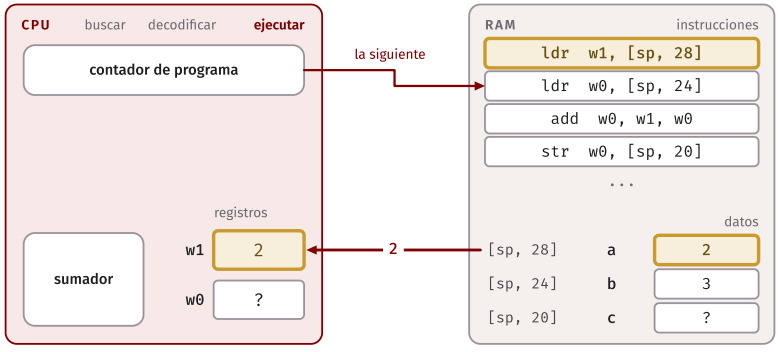
Cada fila es una vuelta del ciclo

Instrucción	Qué pide	w1	w0	c en memoria
`ldr w1, [sp, 28]`	Cargar a a w1	2	?	?
`ldr w0, [sp, 24]`	Cargar b a w0	2	3	?
`add w0, w1, w0`	Sumar y dejar en w0	2	5	?
`str w0, [sp, 20]`	Guardar w0 en c	2	5	5

Fíjense

La suma por sí sola no cambia c: la RAM se actualiza con la cuarta instrucción. Para esta línea se fue a la memoria 7 veces: 4 por instrucciones y 3 por datos.

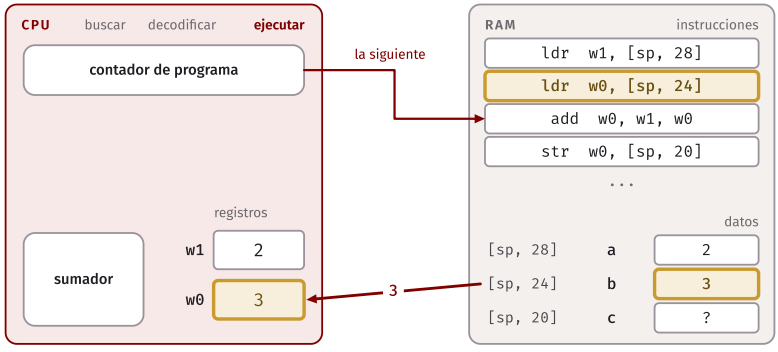
Las cuatro vueltas, en la CPU y en la RAM



ldr w1: carga a, de la memoria a w1. Dos idas: la instrucción y el dato.

```
idas a la memoria: 2
```

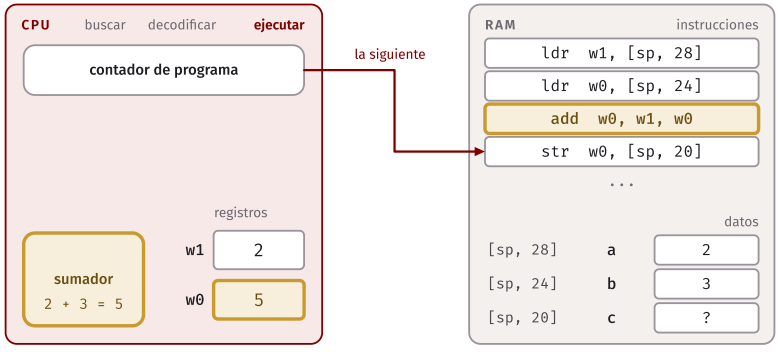
Las cuatro vueltas, en la CPU y en la RAM



ldr w0: carga b, de la memoria a w0. Otras dos idas.

```
idas a la memoria: 4
```

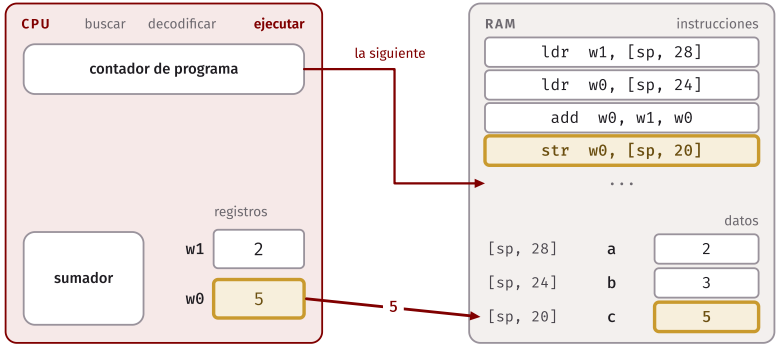
Las cuatro vueltas, en la CPU y en la RAM



add: el sumador calcula 2 + 3 y deja 5 en w0. Solo buscó la instrucción: c aún no cambia.

idas a la memoria: 5

Las cuatro vueltas, en la CPU y en la RAM



str: guarda w0 en la memoria. Recién ahora c vale 5.

idas a la memoria: 7

4 para buscar las instrucciones y 3 para los datos

La tercera fila, por dentro

Paso	Qué ocurre	Qué cambia
Buscar	La CPU lee de la RAM la instrucción `add`	El contador pasa a `str`

La tercera fila, por dentro

Paso	Qué ocurre	Qué cambia
Buscar	La CPU lee de la RAM la instrucción `add`	El contador pasa a `str`
Decodificar	Los bits dicen: suma, con `w1` y `w0`	Nada visible

La tercera fila, por dentro

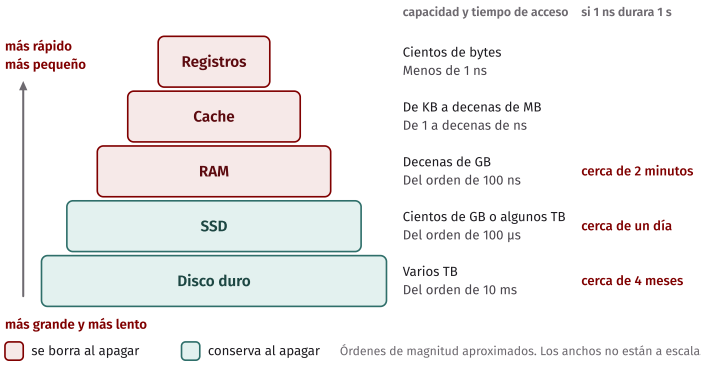
Paso	Qué ocurre	Qué cambia
Buscar	La CPU lee de la RAM la instrucción `add`	El contador pasa a `str`
Decodificar	Los bits dicen: suma, con `w1` y `w0`	Nada visible
Ejecutar	El sumador calcula 2 + 3	`w0` pasa de 3 a 5

SECCIÓN 03

La jerarquía de memoria

03

Lo rápido es pequeño; lo grande es lento



Son órdenes de magnitud, no medidas: varían mucho de una máquina a otra. Importa la distancia entre niveles.

Una escala para imaginarlo

Supongamos que **1 ns dura 1 segundo**.

Acceso	Aproximado	En esta escala
Cache	1 ns	1 segundo

Una escala para imaginarlo

Supongamos que **1 ns dura 1 segundo**.

Acceso	Aproximado	En esta escala
Cache	1 ns	1 segundo
RAM	100 ns	cerca de 2 minutos

Una escala para imaginarlo

Supongamos que **1 ns dura 1 segundo**.

Acceso	Aproximado	En esta escala
Cache	1 ns	1 segundo
RAM	100 ns	cerca de 2 minutos
SSD	100 µs	cerca de un día

Una escala para imaginarlo

Supongamos que **1 ns dura 1 segundo**.

Acceso	Aproximado	En esta escala
Cache	1 ns	1 segundo
RAM	100 ns	cerca de 2 minutos
SSD	100 µs	cerca de un día
Disco duro	10 ms	cerca de 4 meses

La cache

- Memoria pequeña y muy rápida, entre la CPU y la RAM, con copias de lo usado recientemente.
- **Acierto** *(hit)*: el dato está, y se ahorra el viaje a la RAM.
- **Fallo** *(miss)*: hay que traerlo, y queda una copia para la próxima vez.

Funciona por la localidad

Temporal: lo usado hace poco se volverá a usar (`sum` en un ciclo).
Espacial: lo que está cerca se usará pronto. Por eso se trae un bloque de bytes vecinos.

Con datos que caben en la cache, casi todo son aciertos. Y un arreglo recorrido en orden es un caso muy favorable.

SECCIÓN 04

El sistema operativo

04

Tres trabajos del sistema operativo

- **Procesos.** Un proceso es un programa en ejecución. Les da turnos muy cortos en la CPU.

Tres trabajos del sistema operativo

- **Procesos.** Un proceso es un programa en ejecución. Les da turnos muy cortos en la CPU.
- **Memoria.** Da memoria a cada proceso y los mantiene separados. Si uno toca lo que no se le dio, lo detiene: segmentation fault.

Tres trabajos del sistema operativo

- **Procesos.** Un proceso es un programa en ejecución. Les da turnos muy cortos en la CPU.
- **Memoria.** Da memoria a cada proceso y los mantiene separados. Si uno toca lo que no se le dio, lo detiene: segmentation fault.
- **Archivos.** Para el hardware solo hay bloques. Los nombres, carpetas y permisos los pone el sistema operativo.

Tres trabajos del sistema operativo

- **Procesos.** Un proceso es un programa en ejecución. Les da turnos muy cortos en la CPU.
- **Memoria.** Da memoria a cada proceso y los mantiene separados. Si uno toca lo que no se le dio, lo detiene: segmentation fault.
- **Archivos.** Para el hardware solo hay bloques. Los nombres, carpetas y permisos los pone el sistema operativo.

> **¿Y sin sistema operativo?**
>
> Cada programa tendría que manejar por su cuenta la pantalla, el teclado y el disco de cada modelo, y compartir la CPU y la RAM confiando en los demás.

Los programas le piden servicios

terminal, en esta corrida

```
$ ps -o pid,comm
  PID COMMAND
36383 bash
36389 ps
```

`ps` lista procesos. `-o` elige columnas: número (PID) y nombre. Sus números serán otros.

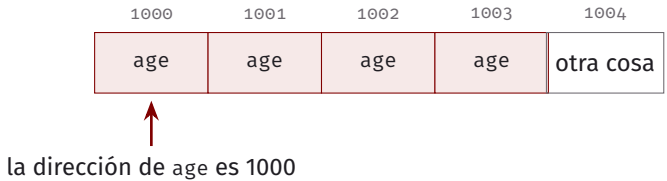
- `bash` es la shell. `ps` también es un proceso.
- `printf` termina pidiéndole que escriba en la terminal.
- El `return 1` de `main` llega al sistema operativo, y de ahí a `echo $?`.

SECCIÓN 05

Direcciones

05

Una fila de bytes numerados



Dirección de memoria

El número que identifica un byte. La dirección de una variable es la de su **primer byte**. (1000 es un número inventado, para el ejemplo.)

Ver direcciones

variable_addresses.c

```c
printf("age     vale %d, ocupa %zu bytes, está en %p\n",
       age, sizeof(age), (void *) &age);
```

- `&age`: la dirección de `age`, como con `scanf`.
- `%p`: escribe una dirección; aquí, en hexadecimal con `0x`.
- `(void *)`: un cast, como `(double)`. `void *` es “una dirección, sin decir de qué tipo es el dato”. `%p` siempre va con él.
- `%zu`: el formato de `sizeof`.

Las direcciones cambian

terminal

```
$ build/variable_addresses
age     vale 19, ocupa 4 bytes, está en 0xffff92c00030
average vale 14.5, ocupa 8 bytes, está en 0xffff92c00040
section vale A, ocupa 1 byte, está en 0xffff92c00020
$ build/variable_addresses
age     vale 19, ocupa 4 bytes, está en 0xffff81c00030
average vale 14.5, ocupa 8 bytes, está en 0xffff81c00040
section vale A, ocupa 1 byte, está en 0xffff81c00020
```

Dependen de la máquina y de la corrida

Sus direcciones serán otras. Y C no promete nada sobre el orden o la distancia entre **variables sueltas**: aquí incluso hay huecos que deja `-fsanitize=address`.

SECCIÓN 06

Arreglos

06

Para qué sirve un arreglo

Un problema que aún no podíamos resolver

¿Cuántos estudiantes superan el promedio del salón?

- Primero hay que leerlas todas, para saber el promedio. Después, comparar cada una.
- Con un acumulador, cada nota se pierde al leer la siguiente.
- Con `grade1`, `grade2`, … no se puede usar un ciclo.

Arreglo ***(array)***

Una secuencia de elementos del mismo tipo, uno al lado del otro en la memoria, a los que se llega con un número: el **índice** *(index)*.

Primer arreglo

first_array.c

```c
int grades[5] = {14, 9, 17, 11, 20};

grades[1] = 12;
grades[4] = grades[4] - 1;

for (int i = 0; i < 5; i++) {
    printf("grades[%d] = %d\n", i, grades[i]);
}
```

terminal

```
$ build/first_array
grades[0] = 14
grades[1] = 12
grades[2] = 17
grades[3] = 11
grades[4] = 19
```

- En la declaración, 5 es el **tamaño**. En `grades[4]`, 4 es un **índice**.
- Cinco elementos: índices del 0 al 4. `grades[5]` no existe.
- El índice puede ser una variable: **un ciclo lo recorre**.

En pseudocódigo

pseudocódigo

```
algoritmo escribir_arreglo(a, n)
    para i desde 0 hasta n - 1 hacer
        escribir a[i]
    fin para
fin algoritmo
```

De pseudocódigo a C

`hasta n - 1` se escribe como `i < n`. `i++` suma 1, e `int i` dentro del `for` solo existe en ese ciclo.

Inicializar y dar nombre al tamaño

array_init.c

```c
#define SIZE 5

int full[SIZE] = {3, 1, 4, 1, 5};
int partial[SIZE] = {7, 8};
int zeros[SIZE] = {0};
int counted[] = {10, 20, 30};
```

- Si faltan valores, los demás quedan en **0**.
- `counted`: el compilador cuenta 3.
- Sin inicializar, leer un elemento es un error, como con una variable.

`#define`

El preprocesador reemplaza `SIZE` por `5` antes de compilar. Sin `=` ni `;`, y en mayúsculas. Para cambiar el tamaño se cambia una sola línea.

`sizeof` con arreglos

array_init.c

```c
printf("sizeof(full)    = %zu\n", sizeof(full));
printf("sizeof(full[0]) = %zu\n", sizeof(full[0]));
printf("elementos de full    = %zu\n", sizeof(full) / sizeof(full[0]));
```

terminal

```
$ build/array_init
sizeof(full)    = 20
sizeof(full[0]) = 4
elementos de full    = 5
```

- El tamaño de **todo el arreglo**, en bytes. 20 y 4 valen aquí, con `int` de 4 bytes. C no fija ese tamaño. La división da 5 en cualquier máquina.

No se asigna ni se compara entero

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/assign_array.c -o build/assign_array
errors/assign_array.c:8:10: error: assignment to expression with array type
$ gcc -Wall -Wextra -std=c17 errors/compare_arrays.c -o build/compare_arrays
errors/compare_arrays.c:8:15: warning: comparison between two arrays [-Warray-compare]
errors/compare_arrays.c:8:15: note: use ‘&first[0] == &second[0]’ to compare the addresses
$ build/compare_arrays
distintos
```

copy_array.c

```
1  for (int i = 0; i < SIZE; i++) {
2      copy[i] = original[i];
3  }
```

== entre arreglos pregunta si están **en el mismo lugar**, no si guardan lo mismo. Para copiar o comparar, un ciclo.

SECCIÓN 07

Un arreglo en memoria

07

Seguidos, sin huecos

array_addresses.c

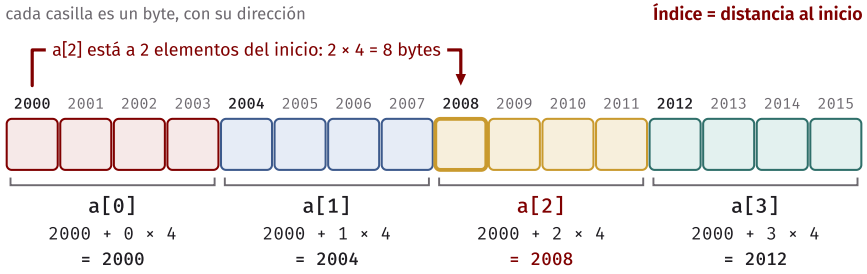
```c
printf("prices[%d] = %4d está en %p\n", i, prices[i], (void *) &prices[i]);
```

terminal

```
$ build/array_addresses
sizeof(int) = 4
prices[0] =  350 está en 0xffffba300020
prices[1] = 1200 está en 0xffffba300024
prices[2] =   80 está en 0xffffba300028
prices[3] =  999 está en 0xffffba30002c
```

- `%4d`: al menos 4 de ancho, con espacios a la izquierda.
- Direcciones de esta corrida. El salto no cambia: 4 bytes, un `int` aquí. C los garantiza seguidos.

La cuenta de la dirección



dirección de a[i] = dirección de a[0] + i × tamaño de un elemento

Con `int` de 4 bytes, como en este entorno; el estándar de C no fija ese tamaño. Calcular dónde está `a[3000]` cuesta lo mismo que para `a[3]`: una multiplicación y una suma. Lo que tarde el acceso depende de la cache.

Por qué los índices empiezan en 0

Idea clave

El índice no es “el número del elemento”. Es su **distancia al inicio**. El primero está a distancia 0 de sí mismo.

- Con índices desde 1, la cuenta necesitaría un `- 1` adicional.
- Es la misma forma de llegar a un dato que en el ensamblador: `[sp, 28]`, una base más una distancia.
- El arreglo no es lo mismo que la dirección de su inicio: `sizeof` da el tamaño de todo el arreglo. Lo completamos la próxima semana.

SECCIÓN 08

Patrones

08



Suma y mayor: el pseudocódigo

pseudocódigo

```
algoritmo resumen_temperaturas(t, n)
    suma ← 0
    para i desde 0 hasta n - 1 hacer
        suma ← suma + t[i]
    fin para
    dia_mayor ← 0
    para i desde 1 hasta n - 1 hacer
        si t[i] > t[dia_mayor] entonces
            dia_mayor ← i
        fin si
    fin para
    escribir suma
    escribir suma / n
    escribir t[dia_mayor], dia_mayor
fin algoritmo
```

- Guardamos el **índice** del mayor: así también sabemos el día.
- `dia_mayor` empieza con el primer elemento.
- Por eso `i` empieza en 1.

Suma y mayor: en C

temperatures.c

```c
int temperatures[DAYS] = {-2, 1, 4, -5, 0, 3, 2};
int sum = 0;

for (int i = 0; i < DAYS; i++) {
    sum = sum + temperatures[i];
}

int max_day = 0;
for (int i = 1; i < DAYS; i++) {
    if (temperatures[i] > temperatures[max_day]) {
        max_day = i;
    }
}
```

i	t[i]	max_day
1	1	1
2	4	2
3	−5	2
4	0	2
5	3	2
6	2	2

terminal

```
$ build/temperatures
Suma: 3
Promedio: 0.43
Mayor: 4, en el día 2
```

Error común: empezar el mayor en 0 (`int max = 0;`) y no con el primer elemento. Con temperaturas todas negativas, diría que la mayor es 0.

Capacidad y cantidad usada

reverse.c

```c
int values[MAX_VALUES];
...
if (n < 1 || n > MAX_VALUES) {
    printf("n debe estar entre 1 y %d\n",
      MAX_VALUES);
    return 1;
}
for (int i = 0; i < n; i++) {
    if (scanf("%d", &values[i]) != 1) {
        printf("Entrada inválida\n");
        return 1;
    }
}
for (int i = n - 1; i >= 0; i--) {
    printf("%d\n", values[i]);
}
```

- **Capacidad:** constante.
- **Cantidad usada:** variable.
- `n` se revisa antes de leer. Nada de `int values[n];`.

terminal

```
$ echo 3 10 20 30 | build/reverse
30
20
10
$ echo 200 | build/reverse
n debe estar entre 1 y 100
```

Un arreglo de contadores

votes.c

```c
if (vote >= 1 && vote <= CANDIDATES) {
    counts[vote - 1]++;
} else {
    invalid++;
}
```

terminal

```
$ echo 1 2 2 3 7 2 0 | build/votes
Candidato 1: 1
Candidato 2: 3
Candidato 3: 1
Inválidos: 1
```

- El **dato** es el índice: el voto `v` va en `v - 1`.
- Los candidatos empiezan en 1; los índices, en 0.

Error común

Usar un dato de la entrada como índice sin revisarlo. Sin ese `if`, el voto 7 escribiría fuera del arreglo.

SECCIÓN 09

Salirse del arreglo

09

`a[n]` está más allá del final

Comportamiento indefinido

En un arreglo de `n` elementos, los índices válidos son de 0 a `n - 1`. **Leer o escribir** `a[n]` es comportamiento indefinido: C no dice qué pasa.

- `a[i]` es una cuenta y un acceso a esa dirección.
- El programa no guarda el tamaño para compararlo.
- Revisar cada acceso costaría tiempo, y C deja esa responsabilidad al programador.

Leer de más, sin ninguna ayuda

errors/off_by_one.c

```c
for (int i = 0; i <= SIZE; i++) {
    printf("grades[%d] = %d\n", i,
     grades[i]);
    sum = sum + grades[i];
}
```

terminal, en esta corrida

```
$ build/off_by_one
grades[0] = 14
grades[1] = 9
grades[2] = 17
grades[3] = 11
grades[4] = 20
grades[5] = 65535
Suma: 65606
```

Un `<=` donde iba un `<`. Compilado con `-Wall -Wextra` y sin sanitizer, no da avisos y termina sin quejarse. El 65535 no significa nada, y en su máquina puede ser otro.

Escribir de más

terminal, en esta corrida

```
$ echo 3 | build/write_out
values[3] = 99
before = 111, after = 222
$ echo 4 | build/write_out
values[4] = 99
before = 111, after = 99
$ echo 5 | build/write_out
values[5] = 99
before = 99, after = 222
$ echo -1 | build/write_out
values[99] = 0
before = 111, after = 222
$ echo 100000 | build/write_out
Segmentation fault
```

`values` tiene 4 elementos, y el programa no toca `before` ni `after`.

Escribir de más

terminal, en esta corrida

```
$ echo 3 | build/write_out
values[3] = 99
before = 111, after = 222
$ echo 4 | build/write_out
values[4] = 99
before = 111, after = 99
$ echo 5 | build/write_out
values[5] = 99
before = 99, after = 222
$ echo -1 | build/write_out
values[99] = 0
before = 111, after = 222
$ echo 100000 | build/write_out
Segmentation fault
```

`values` tiene 4 elementos, y el programa no toca `before` ni `after`.

- **4 y 5:** cambió otra variable, sin error. El caso más peligroso.

Escribir de más

terminal, en esta corrida

```
$ echo 3 | build/write_out
values[3] = 99
before = 111, after = 222
$ echo 4 | build/write_out
values[4] = 99
before = 111, after = 99
$ echo 5 | build/write_out
values[5] = 99
before = 99, after = 222
$ echo -1 | build/write_out
values[99] = 0
before = 111, after = 222
$ echo 100000 | build/write_out
Segmentation fault
```

`values` tiene 4 elementos, y el programa no toca `before` ni `after`.

- **4 y 5:** cambió otra variable, sin error. El caso más peligroso.
- **−1:** el 99 cayó sobre `index`.

Escribir de más

terminal, en esta corrida

```
$ echo 3 | build/write_out
values[3] = 99
before = 111, after = 222
$ echo 4 | build/write_out
values[4] = 99
before = 111, after = 99
$ echo 5 | build/write_out
values[5] = 99
before = 99, after = 222
$ echo -1 | build/write_out
values[99] = 0
before = 111, after = 222
$ echo 100000 | build/write_out
Segmentation fault
```

`values` tiene 4 elementos, y el programa no toca `before` ni `after`.

- **4 y 5:** cambió otra variable, sin error. El caso más peligroso.
- **−1:** el 99 cayó sobre `index`.
- **100000:** tocó memoria ajena al proceso y el sistema operativo lo detuvo.

Escribir de más

terminal, en esta corrida

```
$ echo 3 | build/write_out
values[3] = 99
before = 111, after = 222
$ echo 4 | build/write_out
values[4] = 99
before = 111, after = 99
$ echo 5 | build/write_out
values[5] = 99
before = 99, after = 222
$ echo -1 | build/write_out
values[99] = 0
before = 111, after = 222
$ echo 100000 | build/write_out
Segmentation fault
```

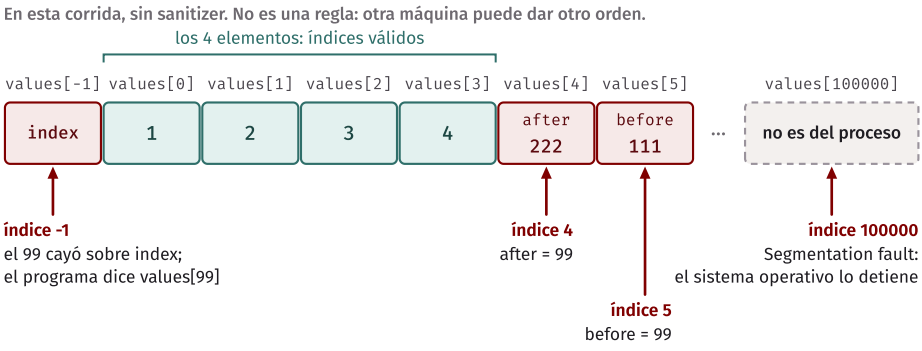
`values` tiene 4 elementos, y el programa no toca `before` ni `after`.

- **4 y 5:** cambió otra variable, sin error. El caso más peligroso.
- **−1:** el 99 cayó sobre `index`.
- **100000:** tocó memoria ajena al proceso y el sistema operativo lo detuvo.

No son reglas

Es una corrida. En otra máquina, todo puede ser distinto.

Qué tocó cada índice, en esta corrida



Con `-fsanitize=address`

terminal, recortado

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address errors/off_by_one.c -o build/off_by_one_asan
$ build/off_by_one_asan
grades[4] = 20
==26852==ERROR: AddressSanitizer: stack-buffer-overflow on address 0xffffa5000034 ...
READ of size 4 at 0xffffa5000034 thread T0
    #0 0xaaaae2270db8 in main errors/off_by_one.c:11
    [32, 52) 'grades' (line 7) <== Memory access at offset 52 overflows this variable
SUMMARY: AddressSanitizer: stack-buffer-overflow errors/off_by_one.c:11 in main
```

- `-fsanitize=address` agrega revisiones de memoria, con zonas prohibidas alrededor de cada arreglo. `-g` guarda archivos y líneas.
- **No es un error de compilación**: es un informe del programa al correr. Solo aparece si se llega al acceso malo.
- `READ` o `WRITE`, cuántos bytes, en qué línea y qué arreglo.

Dos sanitizers

con las opciones del Makefile, recortado

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/write_out.c -o build/write_out_both
$ echo 4 | build/write_out_both
errors/write_out.c:17:11: runtime error: index 4 out of bounds for type 'int [4]'
ERROR: AddressSanitizer: stack-buffer-overflow ...
WRITE of size 4 ...
```

- Con las opciones del `Makefile`, que usan los dos, primero habla UBSan: revisa el índice contra el tamaño.
- Después sigue ASan, con su informe y se detiene.

El sanitizer no ve todo

terminal, en esta corrida

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address errors/write_out.c -o build/write_out_asan
$ echo 100 | build/write_out_asan
values[100] = 99
before = 111, after = 222
```

Idea clave

Un índice lejano puede saltar la zona prohibida, y solo se revisa lo que se ejecuta. Compilen con sanitizers, prueben los bordes y revisen todo índice que venga de afuera.

SECCIÓN 10

Laboratorio

10

Cuatro programas

1. **Direcciones** de un arreglo de `int` y uno de `char`. ¿Cuánto aumentan de uno a otro?
2. **N notas:** promedio, mayor y cuántas superan el promedio. Se recorre el arreglo otra vez, ya con el promedio.
3. **Un dado**, 10 000 lanzamientos, con un arreglo de contadores. ¿Qué índice le toca a cada cara?
4. **Salirse a propósito**, con y sin `-fsanitize=address`. Comparen con varios índices.

> **Comprueben**
>
> `make test-lab` falla hasta que terminen. Prueben `N` igual a 1, a 100 y a 101.

Para la próxima clase

- Hoy, en el laboratorio: los cuatro programas de arreglos.
- Leer los apuntes de la semana 10: arreglos y funciones, y cadenas.
- Compilen siempre con `-fsanitize=address,undefined`.

Preguntas

¿Por qué un programa corre más rápido cuando sus datos caben en la cache?
Un arreglo de `int` empieza en 1000 y cada `int` ocupa 4 bytes: ¿dónde está `a[5]`?
¿Qué puede pasar al escribir en `a[10]` si `a` tiene 10 elementos?