

Introducción a la Programación

Funciones como valores

Punteros a función, callbacks, `qsort` y contexto `void *`

Gerson Garrido

Semana 12

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

1. Una función tiene dirección: punteros a función
2. Separar el qué del cómo: map, filter y fold
3. `qsort` y `bsearch` por dentro
4. C no tiene clausuras: el contexto `void *`
5. Tablas de funciones
6. Un backtracker que no sabe qué resuelve

> **Lectura**
>
> Apuntes de la semana 12. Código en `code/week-12`.

La pregunta de la semana

N reinas y el sudoku, lado a lado:

La pregunta de la semana

N reinas y el sudoku, lado a lado:

- El esquema es el mismo: elegir, explorar, deshacer.

La pregunta de la semana

N reinas y el sudoku, lado a lado:

- El esquema es el mismo: elegir, explorar, deshacer.
- Cambia poco: qué opciones hay, cuándo una es válida y qué se hace con una solución.

La pregunta de la semana

N reinas y el sudoku, lado a lado:

- El esquema es el mismo: elegir, explorar, deshacer.
- Cambia poco: qué opciones hay, cuándo una es válida y qué se hace con una solución.
- Y lo escribimos dos veces.

La pregunta de la semana

N reinas y el sudoku, lado a lado:

- El esquema es el mismo: elegir, explorar, deshacer.
- Cambia poco: qué opciones hay, cuándo una es válida y qué se hace con una solución.
- Y lo escribimos dos veces.

> **Hasta ahora**
>
> Un parámetro cambia **con qué datos** trabaja una función. Hoy cambiará **qué hace**.

SECCIÓN 01

Punteros a función

01



Una función también tiene dirección

las direcciones dependen del entorno

```
$ gdb -q -batch -ex 'print add' -ex 'print &add' -ex 'print multiply' build/function_pointer_gdb
$1 = {int (int, int)} 0x7a8 <add>
$2 = (int (*)(int, int)) 0x7a8 <add>
$3 = {int (int, int)} 0x7c8 <multiply>
```

Una función también tiene dirección

las direcciones dependen del entorno

```
$ gdb -q -batch -ex 'print add' -ex 'print &add' -ex 'print multiply' build/function_pointer_gdb
$1 = {int (int, int)} 0x7a8 <add>
$2 = (int (*)(int, int)) 0x7a8 <add>
$3 = {int (int, int)} 0x7c8 <multiply>
```

- Un programa son instrucciones en la memoria. Llamar es saltar a una dirección.

Una función también tiene dirección

las direcciones dependen del entorno

```
$ gdb -q -batch -ex 'print add' -ex 'print &add' -ex 'print multiply' build/function_pointer_gdb
$1 = {int (int, int)} 0x7a8 <add>
$2 = (int (*)(int, int)) 0x7a8 <add>
$3 = {int (int, int)} 0x7c8 <multiply>
```

- Un programa son instrucciones en la memoria. Llamar es saltar a una dirección.
- El nombre de una función, **sin paréntesis**, se convierte en su dirección. Como un arreglo.

Una función también tiene dirección

las direcciones dependen del entorno

```
$ gdb -q -batch -ex 'print add' -ex 'print &add' -ex 'print multiply' build/function_pointer_gdb
$1 = {int (int, int)} 0x7a8 <add>
$2 = (int (*)(int, int)) 0x7a8 <add>
$3 = {int (int, int)} 0x7c8 <multiply>
```

- Un programa son instrucciones en la memoria. Llamar es saltar a una dirección.
- El nombre de una función, **sin paréntesis**, se convierte en su dirección. Como un arreglo.
- Por eso se escribía `qsort(grades, N, sizeof(grades[0]), compare_ints)`, sin `&`.

Una función también tiene dirección

las direcciones dependen del entorno

```
$ gdb -q -batch -ex 'print add' -ex 'print &add' -ex 'print multiply' build/function_pointer_gdb
$1 = {int (int, int)} 0x7a8 <add>
$2 = (int (*)(int, int)) 0x7a8 <add>
$3 = {int (int, int)} 0x7c8 <multiply>
```

- Un programa son instrucciones en la memoria. Llamar es saltar a una dirección.
- El nombre de una función, **sin paréntesis**, se convierte en su dirección. Como un arreglo.
- Por eso se escribía `qsort(grades, N, sizeof(grades[0]), compare_ints)`, sin `&`.

Error común

`add` es la función. `add(2, 3)` ya la llamó, y es un `int`.

La sintaxis: desde el nombre hacia afuera

function_pointer.c

```c
int (*op)(int, int) = add;
```

Parte	Se lee
`op`	`op` es
`(*op)`	un puntero
`(*op)(int, int)`	a una función que recibe dos `int`
`int (*op)(int, int)`	y devuelve un `int`

La sintaxis: desde el nombre hacia afuera

function_pointer.c

```c
int (*op)(int, int) = add;
```

Parte	Se lee
`op`	`op` es
`(*op)`	un puntero
`(*op)(int, int)`	a una función que recibe dos `int`
`int (*op)(int, int)`	y devuelve un `int`

La sintaxis: desde el nombre hacia afuera

function_pointer.c

```c
int (*op)(int, int) = add;
```

Parte	Se lee
`op`	`op` es
`(*op)`	un puntero
`(*op)(int, int)`	a una función que recibe dos `int`
`int (*op)(int, int)`	y devuelve un `int`

La sintaxis: desde el nombre hacia afuera

function_pointer.c

```c
int (*op)(int, int) = add;
```

Parte	Se lee
`op`	`op` es
`(*op)`	un puntero
`(*op)(int, int)`	a una función que recibe dos `int`
`int (*op)(int, int)`	y devuelve un `int`

Es una variable: la misma línea hace otra cosa

function_pointer.c

```c
int (*op)(int, int) = add;
printf("op(2, 3) = %d\n", op(2, 3));

op = multiply;
printf("op(2, 3) = %d\n", op(2, 3));
printf("(*op)(2, 3) = %d\n", (*op)(2, 3));
printf("op == multiply: %d\n", op == multiply
    );
```

Es una variable: la misma línea hace otra cosa

function_pointer.c

```c
int (*op)(int, int) = add;
printf("op(2, 3) = %d\n", op(2, 3));

op = multiply;
printf("op(2, 3) = %d\n", op(2, 3));
printf("(*op)(2, 3) = %d\n", (*op)(2, 3));
printf("op == multiply: %d\n", op == multiply
    );
```

terminal

```
$ build/function_pointer
op(2, 3) = 5
op(2, 3) = 6
(*op)(2, 3) = 6
op == multiply: 1
```

Es una variable: la misma línea hace otra cosa

function_pointer.c

```c
int (*op)(int, int) = add;
printf("op(2, 3) = %d\n", op(2, 3));

op = multiply;
printf("op(2, 3) = %d\n", op(2, 3));
printf("(*op)(2, 3) = %d\n", (*op)(2, 3));
printf("op == multiply: %d\n", op == multiply
    );
```

terminal

```
$ build/function_pointer
op(2, 3) = 5
op(2, 3) = 6
(*op)(2, 3) = 6
op == multiply: 1
```

Qué función corre se decide al correr, mirando qué dirección tiene `op`.

Es una variable: la misma línea hace otra cosa

function_pointer.c

```c
int (*op)(int, int) = add;
printf("op(2, 3) = %d\n", op(2, 3));

op = multiply;
printf("op(2, 3) = %d\n", op(2, 3));
printf("(*op)(2, 3) = %d\n", (*op)(2, 3));
printf("op == multiply: %d\n", op == multiply
    );
```

terminal

```
$ build/function_pointer
op(2, 3) = 5
op(2, 3) = 6
(*op)(2, 3) = 6
op == multiply: 1
```

Qué función corre se decide al correr, mirando qué dirección tiene `op`.

`op(2, 3)` y `(*op)(2, 3)` son lo mismo. Usamos la primera.

Con `typedef` se lee mejor

function_pointer.c

```c
typedef int (*BinaryOp)(int, int);

int apply(BinaryOp op, int a, int b) {
    return op(a, b);
}
```

Con `typedef` se lee mejor

function_pointer.c

```c
typedef int (*BinaryOp)(int, int);

int apply(BinaryOp op, int a, int b) {
    return op(a, b);
}
```

- Se escribe como una declaración, con `typedef` delante. El nombre queda donde iría la variable.

Con `typedef` se lee mejor

function_pointer.c

```c
typedef int (*BinaryOp)(int, int);

int apply(BinaryOp op, int a, int b) {
    return op(a, b);
}
```

- Se escribe como una declaración, con `typedef` delante. El nombre queda donde iría la variable.
- Sirve en una variable, un campo, un arreglo o un parámetro.

Con `typedef` se lee mejor

function_pointer.c

```c
typedef int (*BinaryOp)(int, int);

int apply(BinaryOp op, int a, int b) {
    return op(a, b);
}
```

- Se escribe como una declaración, con `typedef` delante. El nombre queda donde iría la variable.
- Sirve en una variable, un campo, un arreglo o un parámetro.
- Una función que se pasa para que otra la llame es un **callback**.

Con `typedef` se lee mejor

function_pointer.c

```c
typedef int (*BinaryOp)(int, int);

int apply(BinaryOp op, int a, int b) {
    return op(a, b);
}
```

- Se escribe como una declaración, con `typedef` delante. El nombre queda donde iría la variable.
- Sirve en una variable, un campo, un arreglo o un parámetro.
- Una función que se pasa para que otra la llame es un **callback**.
- `apply` no sabe qué operación hace. Lo sabe quien la llama.

Lo que dice `gcc`

Sin los paréntesis, es una función que devuelve `int *`:

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/missing_parens.c -o build/missing_parens
errors/missing_parens.c:9:5: error: function 'op' is initialized like a variable
```

Lo que dice `gcc`

Sin los paréntesis, es una función que devuelve `int *`:

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/missing_parens.c -o build/missing_parens
errors/missing_parens.c:9:5: error: function 'op' is initialized like a variable
```

El tipo incluye parámetros y retorno. `negate` recibe un solo `int`:

extracto de errors/wrong_signature.c

```
errors/wrong_signature.c:11:19: error: initialization of 'BinaryOp' ... from incompatible pointer type 'int (*)(int)'
```

Lo que dice `gcc`

Sin los paréntesis, es una función que devuelve `int *`:

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/missing_parens.c -o build/missing_parens
errors/missing_parens.c:9:5: error: function 'op' is initialized like a variable
```

El tipo incluye parámetros y retorno. `negate` recibe un solo `int`:

extracto de errors/wrong_signature.c

```
errors/wrong_signature.c:11:19: error: initialization of 'BinaryOp' ... from incompatible pointer type 'int (*)(int)'
```

Un `void *` guarda direcciones de **datos**. El estándar no dice que pueda guardar una función: por eso no las imprimimos con `%p`.

SECCIÓN 02

El qué y el cómo

02

El ciclo repetido

repeated_loops.c

```c
void add_tax_all(double *prices, int n) {
    for (int i = 0; i < n; i++) {
        prices[i] = prices[i] * 1.18;
    }
}
```

El ciclo repetido

repeated_loops.c

```c
void add_tax_all(double *prices, int n) {
    for (int i = 0; i < n; i++) {
        prices[i] = prices[i] * 1.18;
    }
}
```

repeated_loops.c

```c
void half_price_all(double *prices, int n)
    {
    for (int i = 0; i < n; i++) {
        prices[i] = prices[i] / 2.0;
    }
}
```

El ciclo repetido

repeated_loops.c

```c
void add_tax_all(double *prices, int n) {
    for (int i = 0; i < n; i++) {
        prices[i] = prices[i] * 1.18;
    }
}
```

repeated_loops.c

```c
void half_price_all(double *prices, int n)
    {
    for (int i = 0; i < n; i++) {
        prices[i] = prices[i] / 2.0;
    }
}
```

prices.c

```c
typedef double (*PriceChange)(double);

void change_all(double *prices, int n,
     PriceChange change) {
    for (int i = 0; i < n; i++) {
        prices[i] = change(prices[i]);
    }
}
```

El ciclo repetido

repeated_loops.c

```c
void add_tax_all(double *prices, int n) {
    for (int i = 0; i < n; i++) {
        prices[i] = prices[i] * 1.18;
    }
}
```

repeated_loops.c

```c
void half_price_all(double *prices, int n)
    {
    for (int i = 0; i < n; i++) {
        prices[i] = prices[i] / 2.0;
    }
}
```

prices.c

```c
typedef double (*PriceChange)(double);

void change_all(double *prices, int n,
     PriceChange change) {
    for (int i = 0; i < n; i++) {
        prices[i] = change(prices[i]);
    }
}
```

Cómo se recorre: una vez. **Qué** se hace: una función de una línea, y lo decide quien llama.

Tres formas de recorrer un arreglo

Forma	Qué produce	Callback	Ejemplo
map	Mismo tamaño, cada elemento transformado	Recibe un elemento y devuelve el nuevo	El impuesto
filter	Los elementos que cumplen algo	Recibe un elemento y dice sí o no	Los aprobados
fold	Un solo valor	Recibe el acumulado y un elemento, y devuelve el nuevo acumulado	La suma, el mayor

Fold: el mismo ciclo, suma o mayor

prices.c

```c
double fold_prices(const double *prices, int n,
      PriceFold combine, double initial) {
    double result = initial;
    for (int i = 0; i < n; i++) {
        result = combine(result, prices[i]);
    }
    return result;
}
```

prices.c

```c
printf("total: %.2f\n", fold_prices(prices, COUNT,
       add_prices, 0.0));
printf("mayor: %.2f\n", fold_prices(prices, COUNT,
       larger_price, prices[0]));
```

Fold: el mismo ciclo, suma o mayor

prices.c

```c
double fold_prices(const double *prices, int n,
      PriceFold combine, double initial) {
    double result = initial;
    for (int i = 0; i < n; i++) {
        result = combine(result, prices[i]);
    }
    return result;
}
```

prices.c

```c
printf("total: %.2f\n", fold_prices(prices, COUNT,
       add_prices, 0.0));
printf("mayor: %.2f\n", fold_prices(prices, COUNT,
       larger_price, prices[0]));
```

Los precios son 10, 25, 4.5 y 100. El acumulado de la suma:

Fold: el mismo ciclo, suma o mayor

prices.c

```c
double fold_prices(const double *prices, int n,
      PriceFold combine, double initial) {
    double result = initial;
    for (int i = 0; i < n; i++) {
        result = combine(result, prices[i]);
    }
    return result;
}
```

prices.c

```c
printf("total: %.2f\n", fold_prices(prices, COUNT,
       add_prices, 0.0));
printf("mayor: %.2f\n", fold_prices(prices, COUNT,
       larger_price, prices[0]));
```

Los precios son 10, 25, 4.5 y 100. El acumulado de la suma:

$0 \longrightarrow 10 \longrightarrow 35 \longrightarrow 39.5 \longrightarrow \mathbf{139.5}$

Fold: el mismo ciclo, suma o mayor

prices.c

```c
double fold_prices(const double *prices, int n,
      PriceFold combine, double initial) {
    double result = initial;
    for (int i = 0; i < n; i++) {
        result = combine(result, prices[i]);
    }
    return result;
}
```

prices.c

```c
printf("total: %.2f\n", fold_prices(prices, COUNT,
       add_prices, 0.0));
printf("mayor: %.2f\n", fold_prices(prices, COUNT,
       larger_price, prices[0]));
```

Los precios son 10, 25, 4.5 y 100. El acumulado de la suma:

$0 \longrightarrow 10 \longrightarrow 35 \longrightarrow 39.5 \longrightarrow \mathbf{139.5}$

terminal

```
$ build/prices
total: 139.50
mayor: 100.00
```

Filter: un índice para cada arreglo

pseudocódigo

```
algoritmo filtrar(entrada, n, conservar, salida)
    k ← 0
    para i desde 0 hasta n - 1 hacer
        si conservar(entrada[i]) entonces
            salida[k] ← entrada[i]
            k ← k + 1
        fin si
    fin para
    devolver k
fin algoritmo
```

Filter: un índice para cada arreglo

pseudocódigo

```
algoritmo filtrar(entrada, n, conservar, salida)
    k ← 0
    para i desde 0 hasta n - 1 hacer
        si conservar(entrada[i]) entonces
            salida[k] ← entrada[i]
            k ← k + 1
        fin si
    fin para
    devolver k
fin algoritmo
```

`k` avanza más lento que `i`: por eso la salida puede ser el mismo arreglo. En C lo escriben en el laboratorio.

SECCIÓN 03

qsort y bsearch por dentro

03



`qsort`: ya pueden leer la firma

stdlib.h, con nombres más claros

```
void qsort(void *base, size_t count, size_t size, int (*compare)(const void *, const void *));
```

`qsort`: ya pueden leer la firma

stdlib.h, con nombres más claros

```
void qsort(void *base, size_t count, size_t size, int (*compare)(const void *, const void *));
```

- `void *base`: la dirección, sin decir de qué tipo.

`qsort`: ya pueden leer la firma

stdlib.h, con nombres más claros

```
void qsort(void *base, size_t count, size_t size, int (*compare)(const void *, const void *));
```

- `void *base`: la dirección, sin decir de qué tipo.
- `size`: sin el tipo, es la única forma de saber dónde empieza el siguiente elemento.

`qsort`: ya pueden leer la firma

stdlib.h, con nombres más claros

```
void qsort(void *base, size_t count, size_t size, int (*compare)(const void *, const void *));
```

- `void *base`: la dirección, sin decir de qué tipo.
- `size`: sin el tipo, es la única forma de saber dónde empieza el siguiente elemento.
- `compare`: lo único que sabe del significado de los datos.

`qsort`: ya pueden leer la firma

stdlib.h, con nombres más claros

```
void qsort(void *base, size_t count, size_t size, int (*compare)(const void *, const void *));
```

- `void *base`: la dirección, sin decir de qué tipo.
- `size`: sin el tipo, es la única forma de saber dónde empieza el siguiente elemento.
- `compare`: lo único que sabe del significado de los datos.
- `const void *`: `void` porque `qsort` solo tiene direcciones; `const` porque comparar no cambia nada.

`qsort`: ya pueden leer la firma

stdlib.h, con nombres más claros

```
void qsort(void *base, size_t count, size_t size, int (*compare)(const void *, const void *));
```

- `void *base`: la dirección, sin decir de qué tipo.
- `size`: sin el tipo, es la única forma de saber dónde empieza el siguiente elemento.
- `compare`: lo único que sabe del significado de los datos.
- `const void *`: `void` porque `qsort` solo tiene direcciones; `const` porque comparar no cambia nada.

> **Idea clave**
>
> Todo parámetro está ahí por lo mismo: `qsort` **no sabe qué está ordenando**.

Escribamos uno

any_sort.c

```c
void any_sort(void *base, size_t count, size_t size,
      Compare compare) {
    unsigned char *bytes = base;
    for (size_t i = 0; i < count; i++) {
        size_t smallest = i;
        for (size_t j = i + 1; j < count; j++) {
            if (compare(bytes + j * size, bytes +
  smallest * size) < 0) {
                smallest = j;
            }
        }
        if (smallest != i) {
            swap_bytes(bytes + i * size, bytes +
  smallest * size, size);
        }
    }
}
```

El ordenamiento por selección, con tres cambios:

Escribamos uno

any_sort.c

```c
void any_sort(void *base, size_t count, size_t size,
      Compare compare) {
    unsigned char *bytes = base;
    for (size_t i = 0; i < count; i++) {
        size_t smallest = i;
        for (size_t j = i + 1; j < count; j++) {
            if (compare(bytes + j * size, bytes +
  smallest * size) < 0) {
                smallest = j;
            }
        }
        if (smallest != i) {
            swap_bytes(bytes + i * size, bytes +
  smallest * size, size);
        }
    }
}
```

El ordenamiento por selección, con tres cambios:

1. `a[j] < a[s]` es una llamada a `compare`.

Escribamos uno

any_sort.c

```c
void any_sort(void *base, size_t count, size_t size,
      Compare compare) {
    unsigned char *bytes = base;
    for (size_t i = 0; i < count; i++) {
        size_t smallest = i;
        for (size_t j = i + 1; j < count; j++) {
            if (compare(bytes + j * size, bytes +
  smallest * size) < 0) {
                smallest = j;
            }
        }
        if (smallest != i) {
            swap_bytes(bytes + i * size, bytes +
  smallest * size, size);
        }
    }
}
```

El ordenamiento por selección, con tres cambios:

1. `a[j] < a[s]` es una llamada a `compare`.
2. `a[j]` es `bytes + j * size`: aritmética de punteros a mano.

Escribamos uno

any_sort.c

```c
void any_sort(void *base, size_t count, size_t size,
      Compare compare) {
    unsigned char *bytes = base;
    for (size_t i = 0; i < count; i++) {
        size_t smallest = i;
        for (size_t j = i + 1; j < count; j++) {
            if (compare(bytes + j * size, bytes +
  smallest * size) < 0) {
                smallest = j;
            }
        }
        if (smallest != i) {
            swap_bytes(bytes + i * size, bytes +
  smallest * size, size);
        }
    }
}
```

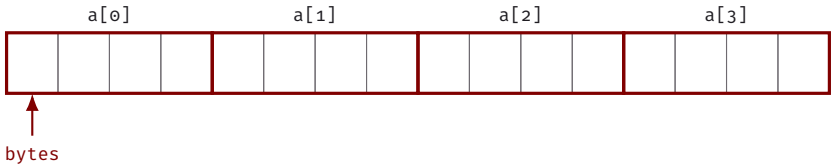
El ordenamiento por selección, con tres cambios:

1. `a[j] < a[s]` es una llamada a `compare`.
2. `a[j]` es `bytes + j * size`: aritmética de punteros a mano.
3. Intercambiar es copiar `size` bytes, uno por uno.

Dónde está cada elemento

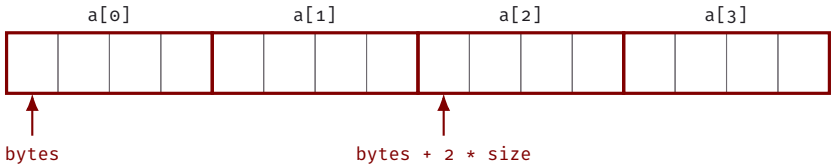


Dónde está cada elemento



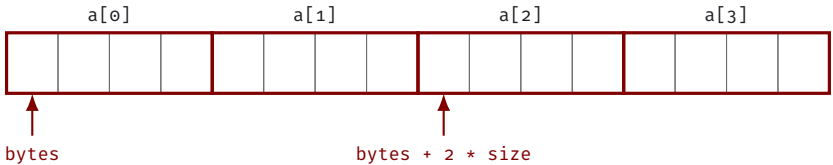
- Aquí `size` vale 4, porque en este entorno un `int` ocupa 4 bytes.

Dónde está cada elemento



- Aquí `size` vale 4, porque en este entorno un `int` ocupa 4 bytes.
- `any_sort` no sabe que son `int`: sabe que el elemento 2 está `2 * size` bytes después.

Dónde está cada elemento



- Aquí `size` vale 4, porque en este entorno un `int` ocupa 4 bytes.
- `any_sort` no sabe que son `int`: sabe que el elemento 2 está `2 * size` bytes después.
- `unsigned char` ocupa un byte, así que sumarle 1 avanza un byte.

No se suma a un `void *`

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/void_arithmetic.c -o build/void_arithmetic
$ build/void_arithmetic
9
$ gcc -Wall -Wextra -std=c17 -Wpointer-arith errors/void_arithmetic.c -o build/void_arithmetic
errors/void_arithmetic.c:7:23: warning: pointer of type 'void *' used in arithmetic [-Wpointer-arith]
```

No se suma a un `void *`

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/void_arithmetic.c -o build/void_arithmetic
$ build/void_arithmetic
9
$ gcc -Wall -Wextra -std=c17 -Wpointer-arith errors/void_arithmetic.c -o build/void_arithmetic
errors/void_arithmetic.c:7:23: warning: pointer of type 'void *' used in arithmetic [-Wpointer-arith]
```

- `void` no tiene tamaño. En C no está definido.

No se suma a un `void *`

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/void_arithmetic.c -o build/void_arithmetic
$ build/void_arithmetic
9
$ gcc -Wall -Wextra -std=c17 -Wpointer-arith errors/void_arithmetic.c -o build/void_arithmetic
errors/void_arithmetic.c:7:23: warning: pointer of type 'void *' used in arithmetic [-Wpointer-arith]
```

- `void` no tiene tamaño. En C no está definido.
- `gcc` lo acepta como extensión propia, y con `-Wall -Wextra` no dice nada.

No se suma a un `void *`

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/void_arithmetic.c -o build/void_arithmetic
$ build/void_arithmetic
9
$ gcc -Wall -Wextra -std=c17 -Wpointer-arith errors/void_arithmetic.c -o build/void_arithmetic
errors/void_arithmetic.c:7:23: warning: pointer of type 'void *' used in arithmetic [-Wpointer-arith]
```

- `void` no tiene tamaño. En C no está definido.
- `gcc` lo acepta como extensión propia, y con `-Wall -Wextra` no dice nada.
- Conviertan primero a `unsigned char *`.

La forma del comparador no se negocia

extracto de errors/typed_compare.c

```
errors/typed_compare.c:11:41: error: passing argument 4 of 'qsort' from incompatible pointer type
```

La forma del comparador no se negocia

extracto de errors/typed_compare.c

```
errors/typed_compare.c:11:41: error: passing argument 4 of 'qsort' from incompatible pointer type
```

Con un cast se calla, y sigue estando mal:

errors/typed_compare_cast.c

```
1   qsort(grades, 3, sizeof(grades[0]), (Compare)
        compare_ints);
```

compila sin avisos

```
$ build/typed_compare_cast
9 14 20
```

La forma del comparador no se negocia

extracto de errors/typed_compare.c

```
errors/typed_compare.c:11:41: error: passing argument 4 of 'qsort' from incompatible pointer type
```

Con un cast se calla, y sigue estando mal:

errors/typed_compare_cast.c

```
1   qsort(grades, 3, sizeof(grades[0]), (Compare)
        compare_ints);
```

compila sin avisos

```
$ build/typed_compare_cast
9 14 20
```

Llamar a una función con un puntero de otro tipo es **comportamiento indefinido**. Aquí ordenó bien. Ya saben lo que vale “me funcionó”.

Comparadores de `struct`

sort_students.c

```c
int compare_by_name(const void *first, const void
      *second) {
    const Student *a = first;
    const Student *b = second;
    return strcmp(a->name, b->name);
}

int compare_by_grade_then_name(const void *first,
      const void *second) {
    int by_grade = compare_by_grade_desc(first,
     second);
    if (by_grade != 0) {
        return by_grade;
    }
    return compare_by_name(first, second);
}
```

Comparadores de `struct`

sort_students.c

```c
int compare_by_name(const void *first, const void
      *second) {
    const Student *a = first;
    const Student *b = second;
    return strcmp(a->name, b->name);
}

int compare_by_grade_then_name(const void *first,
      const void *second) {
    int by_grade = compare_by_grade_desc(first,
      second);
    if (by_grade != 0) {
        return by_grade;
    }
    return compare_by_name(first, second);
}
```

terminal

```
$ build/sort_students
por nota y luego nombre:
  Abel     17
  Carmen   17
  Nadia    14
  Rosa     14
  Luis      9
```

Comparadores de `struct`

sort_students.c

```c
int compare_by_name(const void *first, const void
      *second) {
    const Student *a = first;
    const Student *b = second;
    return strcmp(a->name, b->name);
}

int compare_by_grade_then_name(const void *first,
      const void *second) {
    int by_grade = compare_by_grade_desc(first,
     second);
    if (by_grade != 0) {
        return by_grade;
    }
    return compare_by_name(first, second);
}
```

terminal

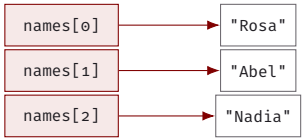
```
$ build/sort_students
por nota y luego nombre:
  Abel     17
  Carmen   17
  Nadia    14
  Rosa     14
  Luis      9
```

`qsort` **no promete ser estable**: si dos dan 0, pueden quedar en cualquier orden. Si les importa, el comparador decide.

Un nivel más de punteros

sort_names.c

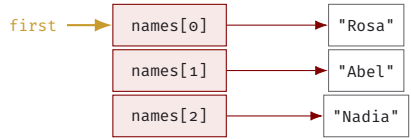
```c
int compare_names(const void *first, const
      void *second) {
    const char *const *a = first;
    const char *const *b = second;
    return strcmp(*a, *b);
}
```



Un nivel más de punteros

sort_names.c

```c
int compare_names(const void *first, const
      void *second) {
    const char *const *a = first;
    const char *const *b = second;
    return strcmp(*a, *b);
}
```



`qsort` pasa la dirección **del elemento**, y el elemento es un puntero.

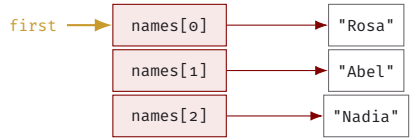
Un nivel más de punteros

sort_names.c

```c
int compare_names(const void *first, const
      void *second) {
    const char *const *a = first;
    const char *const *b = second;
    return strcmp(*a, *b);
}
```

errors/wrong_level.c: strcmp(first, second), sin avisos

```
$ build/wrong_level
Nadia
Luis
Rosa
Abel
```



`qsort` pasa la dirección **del elemento**, y el elemento es un puntero.

Compila, ningún sanitizer se queja, y no está ordenado.

Por qué no se resta

errors/subtract_compare.c

```c
int a = *(const int *) first;
int b = *(const int *) second;
return a - b;
```

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/subtract_compare.c -o build/subtract_compare
$ build/subtract_compare
errors/subtract_compare.c:8:14: runtime error: signed integer overflow: -2147483648 - 1 cannot be represented in type 'int'
con resta: 2147483647
sin resta: -1
```

Por qué no se resta

errors/subtract_compare.c

```c
int a = *(const int *) first;
int b = *(const int *) second;
return a - b;
```

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/subtract_compare.c -o build/subtract_compare
$ build/subtract_compare
errors/subtract_compare.c:8:14: runtime error: signed integer overflow: -2147483648 - 1 cannot be represented in type 'int'
con resta: 2147483647
sin resta: -1
```

- `INT_MIN` es menor que 1. El comparador dijo lo contrario: el signo salió al revés.

Por qué no se resta

errors/subtract_compare.c

```c
int a = *(const int *) first;
int b = *(const int *) second;
return a - b;
```

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/subtract_compare.c -o build/subtract_compare
$ build/subtract_compare
errors/subtract_compare.c:8:14: runtime error: signed integer overflow: -2147483648 - 1 cannot be represented in type 'int'
con resta: 2147483647
sin resta: -1
```

- `INT_MIN` es menor que 1. El comparador dijo lo contrario: el signo salió al revés.
- Con notas de 0 a 20 sí funciona. Por eso pasa las pruebas, hasta que llega un dato grande.

Y `qsort` ordena mal

errors/subtract_sort.c

```c
int values[COUNT] = {2000000000, -2000000000, 5,
      1500000000, -1500000000};
```

el resultado depende del entorno

```
$ gcc -Wall -Wextra -std=c17 errors/subtract_sort.c -o build/subtract_sort
$ build/subtract_sort
5
1500000000
2000000000
-2000000000
-1500000000
```

Y `qsort` ordena mal

errors/subtract_sort.c

```c
int values[COUNT] = {2000000000, -2000000000, 5,
      1500000000, -1500000000};
```

el resultado depende del entorno

```
$ gcc -Wall -Wextra -std=c17 errors/subtract_sort.c -o build/subtract_sort
$ build/subtract_sort
5
1500000000
2000000000
-2000000000
-1500000000
```

Sin aviso, sin error, y con los negativos al final.

Y `qsort` ordena mal

errors/subtract_sort.c

```c
int values[COUNT] = {2000000000, -2000000000, 5,
      1500000000, -1500000000};
```

el resultado depende del entorno

```
$ gcc -Wall -Wextra -std=c17 errors/subtract_sort.c -o build/subtract_sort
$ build/subtract_sort
5
1500000000
2000000000
-2000000000
-1500000000
```

Sin aviso, sin error, y con los negativos al final.

La forma correcta compara:

any_sort.c

```c
return (a > b) - (a < b)
      ;
```

Y `qsort` ordena mal

errors/subtract_sort.c

```c
int values[COUNT] = {2000000000, -2000000000, 5,
      1500000000, -1500000000};
```

el resultado depende del entorno

```
$ gcc -Wall -Wextra -std=c17 errors/subtract_sort.c -o build/subtract_sort
$ build/subtract_sort
5
1500000000
2000000000
-2000000000
-1500000000
```

Sin aviso, sin error, y con los negativos al final.

La forma correcta compara:

any_sort.c

```c
return (a > b) - (a < b)
      ;
```

Da 1, -1 o 0, y nunca hace overflow.

Con `double`, otro problema

errors/double_compare.c

```c
double a = *(const double *) first;
double b = *(const double *) second;
return a - b;
```

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/double_compare.c -o build/double_compare
$ build/double_compare
2.9 2.1 2.5 1.8
```

Con `double`, otro problema

errors/double_compare.c

```c
double a = *(const double *) first;
double b = *(const double *) second;
return a - b;
```

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/double_compare.c -o build/double_compare
$ build/double_compare
2.9 2.1 2.5 1.8
```

- Se devuelve un `int`: 0.8 se convierte en 0, “da igual”.

Con `double`, otro problema

errors/double_compare.c

```c
double a = *(const double *) first;
double b = *(const double *) second;
return a - b;
```

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/double_compare.c -o build/double_compare
$ build/double_compare
2.9 2.1 2.5 1.8
```

- Se devuelve un `int`: 0.8 se convierte en 0, “da igual”.
- Esa conversión está bien definida. Ninguna herramienta avisa.

Con `double`, otro problema

errors/double_compare.c

```c
double a = *(const double *) first;
double b = *(const double *) second;
return a - b;
```

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/double_compare.c -o build/double_compare
$ build/double_compare
2.9 2.1 2.5 1.8
```

- Se devuelve un `int`: 0.8 se convierte en 0, “da igual”.
- Esa conversión está bien definida. Ninguna herramienta avisa.

Un comparador compara. No resta, ni con enteros ni con decimales.

bsearch

search_students.c

```c
int compare_code_with_student(const void *key,
      const void *element) {
    int code = *(const int *) key;
    const Student *student = element;
    return (code > student->code) - (code <
      student->code);
}
```

terminal

```
$ build/search_students
1311: nota 12
1311: nota 12
1312: no está
1312: no está
```

bsearch

search_students.c

```c
int compare_code_with_student(const void *key,
      const void *element) {
    int code = *(const int *) key;
    const Student *student = element;
    return (code > student->code) - (code <
      student->code);
}
```

terminal

```
$ build/search_students
1311: nota 12
1311: nota 12
1312: no está
1312: no está
```

- Devuelve la dirección del elemento, o `NULL`.

bsearch

search_students.c

```c
int compare_code_with_student(const void *key,
      const void *element) {
    int code = *(const int *) key;
    const Student *student = element;
    return (code > student->code) - (code <
      student->code);
}
```

terminal

```
$ build/search_students
1311: nota 12
1311: nota 12
1312: no está
1312: no está
```

- Devuelve la dirección del elemento, o `NULL`.
- El arreglo debe estar ordenado **por el mismo criterio**.

bsearch

search_students.c

```c
int compare_code_with_student(const void *key,
      const void *element) {
    int code = *(const int *) key;
    const Student *student = element;
    return (code > student->code) - (code <
      student->code);
}
```

terminal

```
$ build/search_students
1311: nota 12
1311: nota 12
1312: no está
1312: no está
```

- Devuelve la dirección del elemento, o `NULL`.
- El arreglo debe estar ordenado **por el mismo criterio**.
- El primer argumento del comparador es **la clave**. No tiene que ser del tipo del elemento.

bsearch

search_students.c

```c
int compare_code_with_student(const void *key,
      const void *element) {
    int code = *(const int *) key;
    const Student *student = element;
    return (code > student->code) - (code <
      student->code);
}
```

terminal

```
$ build/search_students
1311: nota 12
1311: nota 12
1312: no está
1312: no está
```

- Devuelve la dirección del elemento, o `NULL`.
- El arreglo debe estar ordenado **por el mismo criterio**.
- El primer argumento del comparador es **la clave**. No tiene que ser del tipo del elemento.
- `any_search`, en el mismo archivo, hace lo mismo: cada resultado sale dos veces.

SECCIÓN 04

Contexto

04

El problema

Queremos sumar las notas mayores que un límite, con un callback.

El problema

Queremos sumar las notas mayores que un límite, con un callback.

- `is_above_15`. ¿Y mayores que 10? `is_above_10`.

El problema

Queremos sumar las notas mayores que un límite, con un callback.

- `is_above_15`. ¿Y mayores que 10? `is_above_10`.
- ¿Y si el usuario escribe el límite al correr? Las funciones se escriben antes de compilar.

El problema

Queremos sumar las notas mayores que un límite, con un callback.

- `is_above_15`. ¿Y mayores que 10? `is_above_10`.
- ¿Y si el usuario escribe el límite al correr? Las funciones se escriben antes de compilar.
- ¿Una variable global? Dos búsquedas se pisan, y está prohibida.

El problema

Queremos sumar las notas mayores que un límite, con un callback.

- `is_above_15`. ¿Y mayores que 10? `is_above_10`.
- ¿Y si el usuario escribe el límite al correr? Las funciones se escriben antes de compilar.
- ¿Una variable global? Dos búsquedas se pisan, y está prohibida.

C no tiene clausuras

Una **clausura** (closure) es una función que se lleva consigo los datos del lugar donde se creó. En C no existen: un puntero a función es una dirección de código, y nada más.

El patrón: el dato viaja aparte

for_each.c

```c
typedef void (*Visitor)(int value, void *
      context);

void for_each(const int *values, int n,
      Visitor visit, void *context) {
    for (int i = 0; i < n; i++) {
        visit(values[i], context);
    }
}
```

El patrón: el dato viaja aparte

for_each.c

```c
typedef void (*Visitor)(int value, void *
      context);

void for_each(const int *values, int n,
      Visitor visit, void *context) {
    for (int i = 0; i < n; i++) {
        visit(values[i], context);
    }
}
```

`for_each` solo transporta `context`.

El patrón: el dato viaja aparte

for_each.c

```c
typedef void (*Visitor)(int value, void *
      context);

void for_each(const int *values, int n,
      Visitor visit, void *context) {
    for (int i = 0; i < n; i++) {
        visit(values[i], context);
    }
}
```

for_each.c

```c
void add_if_above(int value, void *
      context) {
    SumAbove *state = context;
    if (value > state->limit) {
        state->sum += value;
        state->taken++;
    }
}
```

`for_each` solo transporta `context`. El callback lo convierte. Quien llama pasa `&strict`.

El patrón: el dato viaja aparte

for_each.c

```c
typedef void (*Visitor)(int value, void *
    context);

void for_each(const int *values, int n,
    Visitor visit, void *context) {
    for (int i = 0; i < n; i++) {
        visit(values[i], context);
    }
}
```

for_each.c

```c
void add_if_above(int value, void *
    context) {
    SumAbove *state = context;
    if (value > state->limit) {
        state->sum += value;
        state->taken++;
    }
}
```

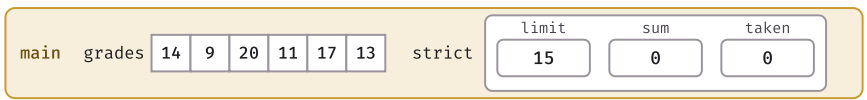
terminal

```
$ build/for_each
mayores que 15: 2, suman 37
mayores que 10: 5, suman 75
```

`for_each` solo transporta `context`. El callback lo convierte. Quien llama pasa `&strict`.

El contexto, en los marcos

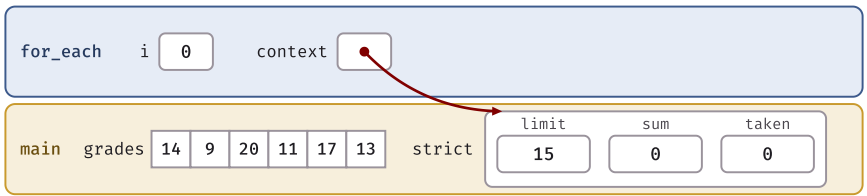
```
SumAbove strict = {15, 0, 0};
```



main prepara sus datos en una variable local.

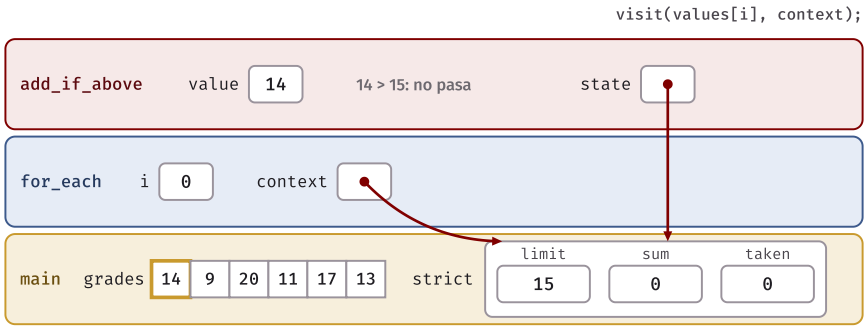
El contexto, en los marcos

```
for_each(grades, COUNT, add_if_above, &strict);
```



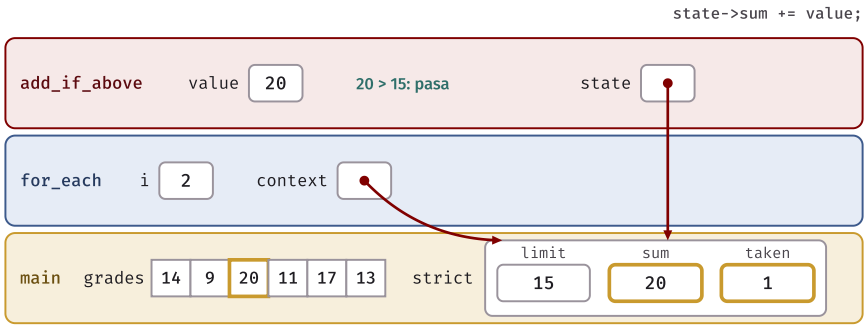
context guarda la dirección de strict. for_each no la mira: solo la transporta.

El contexto, en los marcos



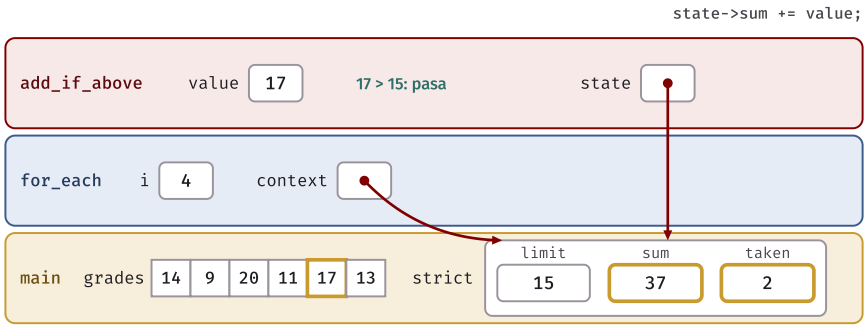
El callback convierte el contexto en state. 14 no supera el límite: nada cambia.

El contexto, en los marcos



El 9 tampoco pasó, y el 20 sí. El callback escribe en el marco de main.

El contexto, en los marcos



Con el 17, strict queda en {15, 37, 2}. El 13 no cambia nada, y main lo lee al volver.

Lo que da el contexto

- Entra datos, como `limit`, y saca datos, como `sum`.

Lo que da el contexto

- Entra datos, como `limit`, y saca datos, como `sum`.
- Dos búsquedas no se estorban: cada una tiene su variable.

Lo que da el contexto

- Entra datos, como `limit`, y saca datos, como `sum`.
- Dos búsquedas no se estorban: cada una tiene su variable.
- Es seguro: `strict` vive en `main`, que termina después de que `for_each` volvió.

Lo que da el contexto

- Entra datos, como `limit`, y saca datos, como `sum`.
- Dos búsquedas no se estorban: cada una tiene su variable.
- Es seguro: `strict` vive en `main`, que termina después de que `for_each` volvió.
- `qsort` y `bsearch` son más antiguos: su comparador **no recibe contexto**.

Qué se pierde: la revisión de tipos

extracto; errors/wrong_context.c pasa la dirección de un int

```
$ build/wrong_context
==162==ERROR: AddressSanitizer: stack-buffer-overflow on address 0xffff9d200034 ...
READ of size 4 at 0xffff9d200034 thread T0
    #0 0xaaaacc19164c in add_if_above errors/wrong_context.c:17
```

- Compila sin avisos. ASan solo lo ve si el callback se sale de la variable.

Qué se pierde: la revisión de tipos

extracto; errors/wrong_context.c pasa la dirección de un int

```
$ build/wrong_context
==162==ERROR: AddressSanitizer: stack-buffer-overflow on address 0xffff9d200034 ...
READ of size 4 at 0xffff9d200034 thread T0
    #0 0xaaaacc19164c in add_if_above errors/wrong_context.c:17
```

errors/wrong_context_silent.c: el callback espera un int, y recibe un double de 15.0

```
$ build/wrong_context_silent
14 pasa
9 pasa
20 pasa
fin
```

- Compila sin avisos. ASan solo lo ve si el callback se sale de la variable.
- El 9 “pasa”: lee 4 de los 8 bytes del `double`, y en esta máquina son ceros.

Qué se pierde: la revisión de tipos

extracto; errors/wrong_context.c pasa la dirección de un int

```
$ build/wrong_context
==162==ERROR: AddressSanitizer: stack-buffer-overflow on address 0xffff9d200034 ...
READ of size 4 at 0xffff9d200034 thread T0
    #0 0xaaaacc19164c in add_if_above errors/wrong_context.c:17
```

errors/wrong_context_silent.c: el callback espera un int, y recibe un double de 15.0

```
$ build/wrong_context_silent
14 pasa
9 pasa
20 pasa
fin
```

- Compila sin avisos. ASan solo lo ve si el callback se sale de la variable.
- El 9 “pasa”: lee 4 de los 8 bytes del `double`, y en esta máquina son ceros.
- Como la etiqueta de una unión: la concordancia es una promesa que **ustedes** deben cumplir.

SECCIÓN 05

Tablas de funciones

05

Elegir con un arreglo

calculator.c

```c
typedef struct {
    const char *name;
    BinaryOp run;
} Operation;
```

calculator.c

```c
const Operation table[] = {
    {"suma", add},
    {"resta", subtract},
    {"producto", multiply},
    {"mayor", larger},
};
```

Elegir con un arreglo

calculator.c

```c
typedef struct {
    const char *name;
    BinaryOp run;
} Operation;
```

calculator.c

```c
const Operation table[] = {
    {"suma", add},
    {"resta", subtract},
    {"producto", multiply},
    {"mayor", larger},
};
```

terminal

```
$ build/calculator suma
suma de 12 y 5: 17
$ build/calculator mayor
mayor de 12 y 5: 12
$ build/calculator potencia
no conozco la operación potencia
```

Elegir con un arreglo

calculator.c

```c
typedef struct {
    const char *name;
    BinaryOp run;
} Operation;
```

calculator.c

```c
const Operation table[] = {
    {"suma", add},
    {"resta", subtract},
    {"producto", multiply},
    {"mayor", larger},
};
```

terminal

```
$ build/calculator suma
suma de 12 y 5: 17
$ build/calculator mayor
mayor de 12 y 5: 12
$ build/calculator potencia
no conozco la operación potencia
```

Agregar una operación es una función y una fila.

Elegir con un arreglo

calculator.c

```c
typedef struct {
    const char *name;
    BinaryOp run;
} Operation;
```

calculator.c

```c
const Operation table[] = {
    {"suma", add},
    {"resta", subtract},
    {"producto", multiply},
    {"mayor", larger},
};
```

terminal

```
$ build/calculator suma
suma de 12 y 5: 17
$ build/calculator mayor
mayor de 12 y 5: 12
$ build/calculator potencia
no conozco la operación potencia
```

Agregar una operación es una función y una fila.

La tabla se puede recorrer: por ejemplo, para escribir un mensaje de ayuda.

Lo que se pierde: el aviso

errors/table_gap.c

```c
enum op_kind { OP_ADD, OP_SUBTRACT, OP_LARGER, OP_COUNT };
const BinaryOp table[OP_COUNT] = {add, subtract};
```

extracto; compila sin avisos

```
$ build/table_gap
==188==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000 (pc 0x000000000000 ...)
==188==Hint: pc points to the zero page.
```

Lo que se pierde: el aviso

errors/table_gap.c

```c
enum op_kind { OP_ADD, OP_SUBTRACT, OP_LARGER, OP_COUNT };
const BinaryOp table[OP_COUNT] = {add, subtract};
```

extracto; compila sin avisos

```
$ build/table_gap
==188==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000 (pc 0x000000000000 ...)
==188==Hint: pc points to the zero page.
```

	`switch` sobre un `enum`	Tabla de funciones
Agregar un caso	Un `case` en cada `switch`	Una fila
Si olvidan un caso	`-Wswitch` avisa	Nadie avisa; `NULL` al correr
Listar las operaciones	Hay que escribir cada caso	Se recorre la tabla

SECCIÓN 06

Un backtracker genérico

06

Primer paso: qué hacer con cada resultado

permutations.c

```c
void explore(int *items, int n, int position,
     OnPermutation visit, void *context) {
    if (position == n) {
        visit(items, n, context);
        return;
    }
    for (int i = position; i < n; i++) {
        swap(&items[position], &items[i]);
        explore(items, n, position + 1, visit,
      context);
        swap(&items[position], &items[i]);
    }
}
```

Primer paso: qué hacer con cada resultado

permutations.c

```c
void explore(int *items, int n, int position,
     OnPermutation visit, void *context) {
    if (position == n) {
        visit(items, n, context);
        return;
    }
    for (int i = position; i < n; i++) {
        swap(&items[position], &items[i]);
        explore(items, n, position + 1, visit,
      context);
        swap(&items[position], &items[i]);
    }
}
```

Un callback las imprime. Otro las cuenta, con un `struct` de contexto: 120 con 5 elementos.

Primer paso: qué hacer con cada resultado

permutations.c

```c
void explore(int *items, int n, int position,
     OnPermutation visit, void *context) {
    if (position == n) {
        visit(items, n, context);
        return;
    }
    for (int i = position; i < n; i++) {
        swap(&items[position], &items[i]);
        explore(items, n, position + 1, visit,
      context);
        swap(&items[position], &items[i]);
    }
}
```

Un callback las imprime. Otro las cuenta, con un `struct` de contexto: 120 con 5 elementos. ¿Qué queda fijo? Los pasos, las opciones y la forma de poner y deshacer.

Segundo paso: preguntarlo todo

pseudocódigo

```
algoritmo explorar(problema, contexto, paso)
    si paso = problema.pasos entonces
        devolver problema.en_solucion(contexto)
    fin si
    para opcion desde 0 hasta problema.opciones(contexto, paso) - 1 hacer
        si problema.es_valida(contexto, paso, opcion) entonces
            problema.poner(contexto, paso, opcion)
            detener ← explorar(problema, contexto, paso + 1)
            problema.deshacer(contexto, paso, opcion)
            si detener entonces
                devolver verdadero
            fin si
        fin si
    fin para
    devolver falso
fin algoritmo
```

Segundo paso: preguntarlo todo

pseudocódigo

```
algoritmo explorar(problema, contexto, paso)
    si paso = problema.pasos entonces
        devolver problema.en_solucion(contexto)
    fin si
    para opcion desde 0 hasta problema.opciones(contexto, paso) - 1 hacer
        si problema.es_valida(contexto, paso, opcion) entonces
            problema.poner(contexto, paso, opcion)
            detener ← explorar(problema, contexto, paso + 1)
            problema.deshacer(contexto, paso, opcion)
            si detener entonces
                devolver verdadero
            fin si
        fin si
    fin para
    devolver falso
fin algoritmo
```

Lean lo que **no** aparece: tableros, reinas, dígitos. El estado está detrás de `contexto`.

La interfaz es un `struct`

lab/backtrack.h

```
1  typedef struct {
2      int steps;
3      int (*options)(void *context, int step);
4      int (*is_valid)(void *context, int step, int
         option);
5      void (*place)(void *context, int step, int
         option);
6      void (*undo)(void *context, int step, int
         option);
7      int (*on_solution)(void *context);
8  } Problem;
```

La interfaz es un `struct`

lab/backtrack.h

```
1  typedef struct {
2      int steps;
3      int (*options)(void *context, int step);
4      int (*is_valid)(void *context, int step, int
         option);
5      void (*place)(void *context, int step, int
         option);
6      void (*undo)(void *context, int step, int
         option);
7      int (*on_solution)(void *context);
8  } Problem;
```

- Un conjunto de operaciones que se pasa como argumento.

La interfaz es un `struct`

lab/backtrack.h

```
1  typedef struct {
2      int steps;
3      int (*options)(void *context, int step);
4      int (*is_valid)(void *context, int step, int
         option);
5      void (*place)(void *context, int step, int
         option);
6      void (*undo)(void *context, int step, int
         option);
7      int (*on_solution)(void *context);
8  } Problem;
```

- Un conjunto de operaciones que se pasa como argumento.
- `undo` se llama **siempre**, aunque haya que detenerse. Por eso `on_solution` copia.

La interfaz es un `struct`

lab/backtrack.h

```
1  typedef struct {
2      int steps;
3      int (*options)(void *context, int step);
4      int (*is_valid)(void *context, int step, int
         option);
5      void (*place)(void *context, int step, int
         option);
6      void (*undo)(void *context, int step, int
         option);
7      int (*on_solution)(void *context);
8  } Problem;
```

- Un conjunto de operaciones que se pasa como argumento.
- `undo` se llama **siempre**, aunque haya que detenerse. Por eso `on_solution` copia.
- Las soluciones se cuentan en el contexto: el motor no sabe qué le interesa a cada problema.

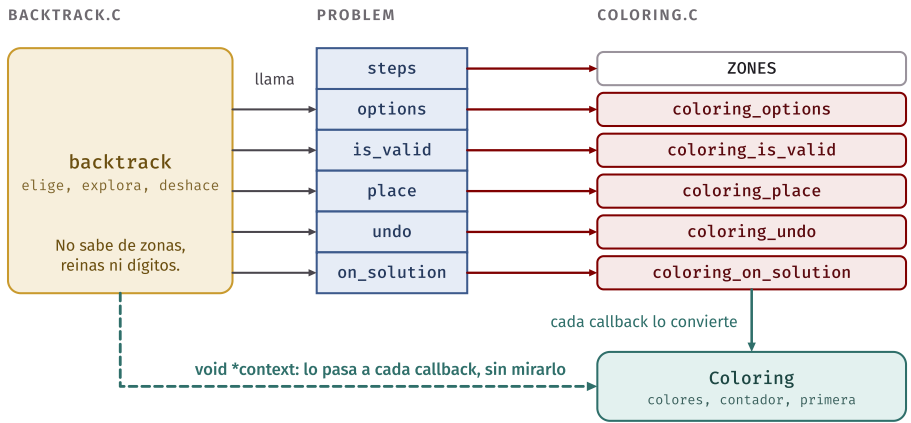
La interfaz es un `struct`

lab/backtrack.h

```
1  typedef struct {
2      int steps;
3      int (*options)(void *context, int step);
4      int (*is_valid)(void *context, int step, int
         option);
5      void (*place)(void *context, int step, int
         option);
6      void (*undo)(void *context, int step, int
         option);
7      int (*on_solution)(void *context);
8  } Problem;
```

- Un conjunto de operaciones que se pasa como argumento.
- `undo` se llama **siempre**, aunque haya que detenerse. Por eso `on_solution` copia.
- Las soluciones se cuentan en el contexto: el motor no sabe qué le interesa a cada problema.
- El motor solo devuelve cuántos nodos visitó.

El motor solo conoce `Problem`



queens.c y sudoku.c llenan el mismo struct con sus funciones. backtrack.c no cambia.

El mismo motor, tres problemas

Pregunta	Pintar un mapa	N reinas	Sudoku
¿Un paso?	Una zona	Una fila	Una celda vacía
¿Una opción?	Un color	Una columna	Un dígito menos 1
¿Cuántos pasos?	Las zonas	`n`	Las celdas vacías
¿Válida?	Ninguna vecina anterior tiene ese color	Ninguna reina anterior la ataca	`board_can_place`
¿Contexto?	Colores y contador	Columnas y contador	Tablero, celdas vacías y contador

El modelo: pintar un mapa

lab/coloring.c

```
1  static int coloring_is_valid(void *context,
       int step, int option) {
2      const Coloring *state = context;
3      for (int zone = 0; zone < step; zone++)
         {
4          if (NEIGHBORS[step][zone] && state
       ->color[zone] == option) {
5              return 0;
6          }
7      }
8      return 1;
9  }
```

El modelo: pintar un mapa

lab/coloring.c

```c
static int coloring_is_valid(void *context,
      int step, int option) {
    const Coloring *state = context;
    for (int zone = 0; zone < step; zone++)
      {
        if (NEIGHBORS[step][zone] && state
      ->color[zone] == option) {
            return 0;
        }
    }
    return 1;
}
```

lab/coloring.c

```c
Problem problem = {
    .steps = ZONES,
    .options = coloring_options,
    .is_valid = coloring_is_valid,
    .place = coloring_place,
    .undo = coloring_undo,
    .on_solution = coloring_on_solution
     ,
};
```

El modelo: pintar un mapa

lab/coloring.c

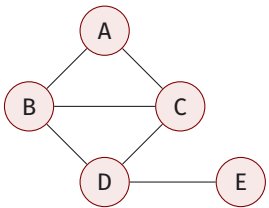
```c
static int coloring_is_valid(void *context,
      int step, int option) {
    const Coloring *state = context;
    for (int zone = 0; zone < step; zone++)
      {
        if (NEIGHBORS[step][zone] && state
      ->color[zone] == option) {
            return 0;
        }
    }
    return 1;
}
```

lab/coloring.c

```c
Problem problem = {
    .steps = ZONES,
    .options = coloring_options,
    .is_valid = coloring_is_valid,
    .place = coloring_place,
    .undo = coloring_undo,
    .on_solution = coloring_on_solution
     ,
};
```

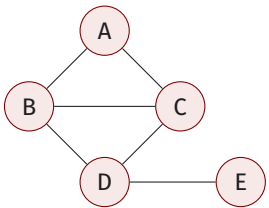
Cada callback es corto, y empieza convirtiendo el contexto. Armar el problema es llenar el `struct`.

Comprobar a mano



Cinco zonas, tres colores, y dos vecinas no pueden tener el mismo.

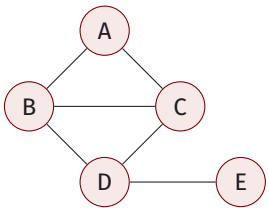
Comprobar a mano



Cinco zonas, tres colores, y dos vecinas no pueden tener el mismo.

- A, B y C son vecinas entre sí: $3 \times 2 \times 1 = 6$ formas.

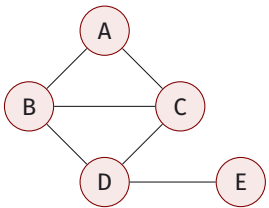
Comprobar a mano



Cinco zonas, tres colores, y dos vecinas no pueden tener el mismo.

- A, B y C son vecinas entre sí: 3 × 2 × 1 = 6 formas.
- D es vecina de B y de C: le queda un solo color.

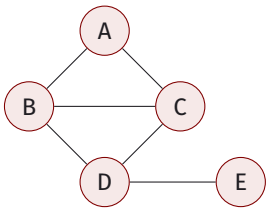
Comprobar a mano



Cinco zonas, tres colores, y dos vecinas no pueden tener el mismo.

- A, B y C son vecinas entre sí: 3 × 2 × 1 = 6 formas.
- D es vecina de B y de C: le queda un solo color.
- E solo es vecina de D: le quedan 2.

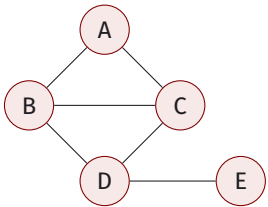
Comprobar a mano



Cinco zonas, tres colores, y dos vecinas no pueden tener el mismo.

- A, B y C son vecinas entre sí: 3 × 2 × 1 = 6 formas.
- D es vecina de B y de C: le queda un solo color.
- E solo es vecina de D: le quedan 2.
- En total, 6 × 1 × 2 = **12**.

Comprobar a mano



Cinco zonas, tres colores, y dos vecinas no pueden tener el mismo.

- A, B y C son vecinas entre sí: 3 × 2 × 1 = 6 formas.
- D es vecina de B y de C: le queda un solo color.
- E solo es vecina de D: le quedan 2.
- En total, 6 × 1 × 2 = **12**.

con el motor del laboratorio

```
$ build/lab/coloring
12 soluciones, 34 nodos
```

Los números

n	Soluciones	Nodos	Hasta la 1.ª
4	2	17	9
5	10	54	6
6	4	153	32
7	40	552	10
8	92	2057	114

Tablero	Vacías	Soluciones	Nodos
`facil.txt`	36	1	69
`medio.txt`	45	1	848
`dificil.txt`	52	1	26019
`sin_solucion.txt`	45	0	689
`repetido.txt`	35	0	0

Los números

n	Soluciones	Nodos	Hasta la 1.ª
4	2	17	9
5	10	54	6
6	4	153	32
7	40	552	10
8	92	2057	114

Tablero	Vacías	Soluciones	Nodos
`facil.txt`	36	1	69
`medio.txt`	45	1	848
`dificil.txt`	52	1	26019
`sin_solucion.txt`	45	0	689
`repetido.txt`	35	0	0

- Es el mismo `backtrack.c`, sin cambiar una línea, y no menciona ningún problema.

Los números

n	Soluciones	Nodos	Hasta la 1.ª
4	2	17	9
5	10	54	6
6	4	153	32
7	40	552	10
8	92	2057	114

Tablero	Vacías	Soluciones	Nodos
`facil.txt`	36	1	69
`medio.txt`	45	1	848
`dificil.txt`	52	1	26019
`sin_solucion.txt`	45	0	689
`repetido.txt`	35	0	0

- Es el mismo `backtrack.c`, sin cambiar una línea, y no menciona ningún problema.
- El programa de sudoku busca hasta 2 soluciones, para saber si es única.

Lo que cuesta

- Una función de 15 líneas pasó a ser un `struct` con cinco callbacks por problema.

Lo que cuesta

- Una función de 15 líneas pasó a ser un `struct` con cinco callbacks por problema.
- Para un solo problema no vale la pena: escriban el backtracking directo.

Lo que cuesta

- Una función de 15 líneas pasó a ser un `struct` con cinco callbacks por problema.
- Para un solo problema no vale la pena: escriban el backtracking directo.
- Todo el estado pasa por `void *`, sin revisión de tipos.

Lo que cuesta

- Una función de 15 líneas pasó a ser un `struct` con cinco callbacks por problema.
- Para un solo problema no vale la pena: escriban el backtracking directo.
- Todo el estado pasa por `void *`, sin revisión de tipos.
- Elegir la celda con menos opciones **mientras se busca** no cabe: el orden de los pasos es fijo.

Lo que cuesta

- Una función de 15 líneas pasó a ser un `struct` con cinco callbacks por problema.
- Para un solo problema no vale la pena: escriban el backtracking directo.
- Todo el estado pasa por `void *`, sin revisión de tipos.
- Elegir la celda con menos opciones **mientras se busca** no cabe: el orden de los pasos es fijo.

> **Diseño**
>
> Una interfaz con demasiados callbacks es tan difícil de usar como el código repetido. Decidir qué entra es la parte difícil.

SECCIÓN 07

Laboratorio

07

Laboratorio: un solucionador genérico

Sesión 1

1. Tres comparadores de `Trip`, sin restar.
2. `array_map`, `array_filter` y `array_fold`, con pruebas.
3. `array_count_if`, con contexto, y `count_above`, sin globales.

Sesión 2

4. `backtrack.c`: se prueba con `coloring`, ya hecho.
5. Los callbacks de N reinas.
6. Los callbacks del sudoku, sin cambiar `backtrack.c`.

Antes de empezar

Lean `lab/README.md`. `make test-lab` falla hasta que terminen.

Resumen

- El nombre de una función, sin paréntesis, es su dirección. `typedef int (*BinaryOp)(int, int);`
- Un **callback** separa cómo se recorre de qué se hace: map, filter y fold.
- `qsort` y `bsearch` reciben `void *`, un tamaño y un comparador. Para hacer aritmética, a `unsigned char *`.
- El comparador recibe `const void *`, y no resta: `(a > b) - (a < b)`. `qsort` no es estable.
- **C no tiene clausuras**. El patrón es un `void *context` junto al callback.
- Con el contexto se pierde la revisión de tipos: es una promesa de ustedes.
- Una tabla de funciones elige con datos, y pierde el aviso de `-Wswitch`.
- Un backtracker genérico es un `struct` de callbacks y un contexto que nunca mira.

Para la próxima clase

- Terminen el laboratorio: `make test-lab`, sin informes de ASan.
- Resuelvan los ejercicios de los apuntes, sobre todo el 2, el 4 y el 6.
- Respondan las preguntas de repaso de la guía de la semana 12.
- La semana 13 empieza el intérprete: datos recursivos y árboles de expresión.

Antes de escribir un callback

¿Qué parte es el cómo, y qué parte es el qué? ¿Qué datos necesita, y quién es su owner?