

UNIVERSIDAD NACIONAL DE INGENIERÍA

Introducción a la Computación

Ciclos anidados y números aleatorios

Filas y columnas, semillas y una simulación con dados

Gerson Garrido

Semana 6

Hoy

1. Un ciclo dentro de otro: rectángulos y tablas
2. Cuántas veces corre cada línea
3. Patrones y divisores
4. Números pseudoaleatorios: `rand` y `srand`
5. Un número en un rango
6. Simular: estimar la probabilidad de sumar 7 con dos dados

Lectura

Apuntes de la semana 6. Código en `code/week-06`.

SECCIÓN 01

Un ciclo dentro de otro

01

Un rectángulo: primero una fila

pseudocódigo

```
algoritmo rectangulo()
    leer filas, columnas
    para fila desde 1 hasta filas hacer
        para columna desde 1 hasta columnas hacer
            escribir "*" sin saltar de línea
        fin para
        saltar de línea
    fin para
fin algoritmo
```

- Una fila ya sabemos dibujarla: es un ciclo.
- El rectángulo repite eso, una vez por fila.
- **Ciclos anidados** *(nested loops)*: un ciclo exterior y uno interior.

Por cada vuelta del exterior, el interior corre **completo**, desde el principio. Como el minutero: da toda su vuelta por cada hora.

En C: la posición dice cuántas veces

rectangle.c

```c
for (int row = 1; row <= rows; row++) {
    for (int col = 1; col <= cols; col++) {
        printf("*");
    }
    printf("\n");
}
```

terminal

```
$ echo 2 5 | build/
    rectangle
*****
*****
```

- `printf("*")` está en los dos ciclos: una vez por **casilla**.
- `printf("\n")` está solo en el exterior: una vez por **fila**.
- `row++` suma 1 a `row`. Un `int` declarado en el `for` solo existe en ese ciclo.

Lo seguimos a mano: 2 filas, 3 columnas

`row`	`col`	`¿col <= 3?`	Qué hace
1	1	sí	escribe `*`
1	2	sí	escribe `*`
1	3	sí	escribe `*`
1	4	no	termina el interior, salta de línea
2	1	sí	escribe `*` en la fila nueva
2	2	sí	escribe `*`
2	3	sí	escribe `*`
2	4	no	termina el interior, salta de línea

`col` **vuelve a 1**: el `for` interior ejecuta otra vez `int col = 1` cada vez que se llega a él.

Por cada fila, el interior corre completo

rows = 2, cols = 3

```
for (int row = 1; row <= rows; row++) {
    for (int col = 1; col <= cols; col++) {
        printf("*");
    }
    printf("\n");
}
```

CONTADORES

row 2 col 4

PANTALLA

```
***
***
```

col 4: salta de línea; luego row pasa a 3 y termina el exterior

La tabla de multiplicar

times_table.c

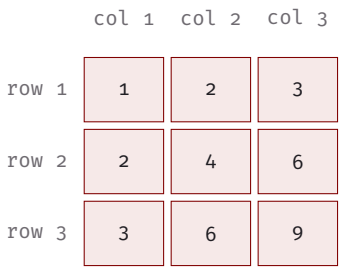
```c
for (int row = 1; row <= n; row++) {
    for (int col = 1; col <= n; col++) {
        printf("%4d", row * col);
    }
    printf("\n");
}
```

terminal

```
$ echo 3 | build/times_table
   1   2   3
   2   4   6
   3   6   9
$ echo 13 | build/times_table
n debe estar entre 1 y 12
```

`%4d`: **ancho mínimo** de 4 caracteres. Si el número es más corto, se completa con espacios a la izquierda. Por eso las columnas quedan alineadas.

En qué orden se llenan las casillas



- La pantalla se escribe de izquierda a derecha y de arriba abajo.
- Por eso, el **exterior recorre las filas** y el interior, las columnas.
- No se puede volver a una fila ya terminada.
- Cada casilla vale `row * col`: depende de los dos ciclos.

SECCIÓN 02

Cuántas veces corre cada línea

02

Contemos, no adivinemos

count_runs.c

```c
for (int row = 1; row <= n; row++) {
    outer_runs++;
    for (int col = 1; col <= n; col++) {
        square_runs++;
    }
}
```

terminal

```
$ echo 10 | build/count_runs
n = 10
Línea del ciclo exterior: 10
Línea interior, cuadrado: 100
Línea interior, triángulo: 55
$ echo 20 | build/count_runs
n = 20
Línea del ciclo exterior: 20
Línea interior, cuadrado: 400
Línea interior, triángulo: 210
```

Exterior: `n` vueltas. Interior: `n` vueltas **por cada una**. Línea interior: $n \times n$.
Si `n` se duplica, la línea interior corre 4 veces más. Si `n` se multiplica por 10, corre 100 veces más.

Las partes del `for` también se cuentan

Parte	Veces	Con $n = 3$
`int row = 1`	1	1
`row <= n`	$n + 1$	4
`row++`	n	3
`int col = 1`	n	3
`col <= n`	$n(n + 1)$	12
cuerpo del interior	n^2	9
`col++`	n^2	9

- La condición corre una vez más que las vueltas: la última da falso.
- `int col = 1` corre una vez **por fila**.
- Un contador también tiene límite: pasarse en un `int` es comportamiento indefinido. Por eso `count_runs` acepta `n` hasta 5000.

Cuando el interior depende del exterior

count_runs.c

```c
for (int row = 1; row <= n; row++) {
    for (int col = 1; col <= row; col++) {
        triangle_runs++;
    }
}
```

Total: $1 + 2 + \cdots + n = \dfrac{n(n+1)}{2}$

La suma	1	2	3	4	5
Al revés	5	4	3	2	1
Juntas	6	6	6	6	6

5 columnas de 6 es 30, el doble de la suma: la suma es 15.

Crece casi igual

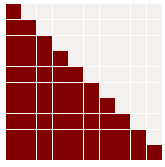
terminal

```
$ echo 100 | build/count_runs
Línea interior, triángulo: 5050
$ echo 200 | build/count_runs
Línea interior, triángulo: 20100
```

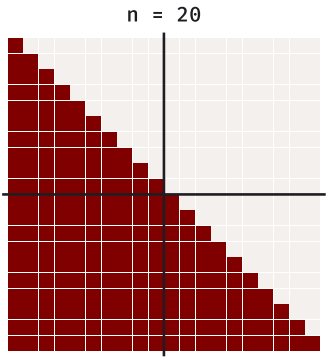
Idea clave

Es casi la mitad del cuadrado, pero al duplicar `n` se multiplica casi por 4. Ser la mitad de un cuadrado sigue siendo crecer como un cuadrado.

Al duplicar n, caben cuatro cuadrados



cuadrado: 100
triángulo: 55



cuadrado: 400
triángulo: 210

Cada casilla es una vez que corre la línea interior.

n se duplica.
El cuadrado se multiplica por 4: caben cuatro cuadrados de n = 10.

- `el triángulo: col <= row`

Es casi la mitad, y crece casi igual.

SECCIÓN 03

Patrones y divisores

03



¿Qué tiene la fila número `row`?

terminal

```
$ echo 4 | build/staircase
1
1 2
1 2 3
1 2 3 4
```

staircase.c

```c
for (int row = 1; row <= n; row++) {
    for (int col = 1; col <= row; col++) {
        printf("%3d", col);
    }
    printf("\n");
}
```

Fila	Primero	Último	Vueltas
1	1	1	1
2	1	2	2
row	1	row	row

La última fila de la tabla se traduce a C. Hagan esta tabla antes de programar un patrón.

Error común: el mismo contador en los dos ciclos

errors/same_counter.c

```c
for (i = 1; i <= 3; i++) {
    for (i = 1; i <= 3; i++) {
        printf("*");
    }
    printf("\n");
}
```

terminal

```
$ build/same_counter
***
```

- Esperábamos 3 filas y sale una.
- Al salir del interior, `i` vale 4; el `i++` la deja en 5.

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/same_counter.c -o build/same_counter
```

`gcc` no avisa: el programa es válido. Cada ciclo con su contador, declarado en su `for`.

Error común: con `while`, el contador no se reinicia solo

errors/no_reset.c

```c
while (row <= 3) {
    while (col <= 4) {
        printf("*");
        col++;
    }
    printf("\n");
    row++;
}
```

terminal

```
$ build/no_reset
****
```

- Las filas 2 y 3 salen vacías.
- `col` llegó a 5 en la fila 1, y ahí se queda.
- Falta `col = 1;` antes del `while` interior.

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/no_reset.c -o build/no_reset
```

Un ciclo que cuenta dentro de otro

divisors.c

```c
for (int n = 1; n <= limit; n++) {
    int count = 0;

    printf("%d:", n);
    for (int d = 1; d <= n; d++) {
        if (n % d == 0) {
            printf(" %d", d);
            count++;
        }
    }
    printf(" (%d)\n", count);
}
```

Un acumulador se inicia **justo antes del ciclo que acumula**.

terminal

```
$ echo 7 | build/divisors
1: 1 (1)
2: 1 2 (2)
3: 1 3 (2)
4: 1 2 4 (3)
5: 1 5 (2)
6: 1 2 3 6 (4)
7: 1 7 (2)
```

Con exactamente 2 divisores: 2, 3, 5, 7. Eso lo usarán en el laboratorio.

SECCIÓN 04

Números pseudoaleatorios

04

¿De dónde sale el azar?

- Una computadora sigue instrucciones exactas: mismos datos, mismo resultado.
- En la mayoría de los programas, los números "al azar" se **calculan**.

Pseudoaleatorio *(pseudorandom)*

Número que parece al azar, pero que sale de una fórmula. Cada número sale de un estado que el generador guarda y actualiza.

Semilla *(seed)*

El valor inicial del que sale toda la secuencia.

Un generador de juguete

$x \leftarrow (5x + 3)$ mód 16, con semilla 7

Paso	x	$5x + 3$	mód16
1	7	38	6
2	6	33	1
3	1	8	8
4	8	43	11
5	11	58	10

toy_generator.c

```c
for (int i = 1; i <= 18; i++) {
    x = (5 * x + 3) % 16;
    printf("%d ", x);
}
```

Tres propiedades

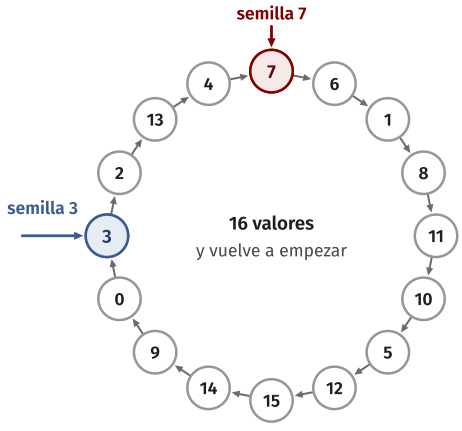
terminal

```
$ echo 7 | build/toy_generator
6 1 8 11 10 5 12 15 14 9 0 3 2 13 4 7 6 1
$ echo 3 | build/toy_generator
2 13 4 7 6 1 8 11 10 5 12 15 14 9 0 3 2 13
```

- **Es determinista:** con la semilla 7 siempre sale lo mismo.
- **La semilla elige el punto de partida:** otra semilla, otro comienzo.
- **Tarde o temprano se repite:** el 6 vuelve en la posición 17. El **período** aquí es 16.

`rand` sigue la misma idea, con una fórmula más elaborada, que cambia de una implementación a otra, y un período mucho más largo.

Un anillo de 16 valores



LA FÓRMULA

```
x ← (5 * x + 3) mod 16
```

CON SEMILLA 7

```
6 1 8 11 10 5 12 15 14
9 0 3 2 13 4 7 6 1
```

CON SEMILLA 3

```
2 13 4 7 6 1 8 11 10
5 12 15 14 9 0 3 2 13
```

Es el mismo anillo. La semilla solo elige dónde se entra; el primer número que se escribe es el siguiente a la semilla.

`rand` y `RAND_MAX`

del entorno del curso

```
$ build/rand_max
RAND_MAX = 2147483647
1804289383
846930886
1681692777
1714636915
1957747793
```

- Está en `stdlib.h`.
- Devuelve un `int` entre 0 y `RAND_MAX`, ambos incluidos.
- `RAND_MAX` depende de la implementación. El estándar exige al menos 32767. Usen el nombre, no el número.
- Sin `srand`, se comporta como con semilla 1: salen **los mismos números** cada vez.

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/no_stdlib.c -o build/no_stdlib
errors/no_stdlib.c:5:20: error: implicit declaration of function ‘rand’ [-Wimplicit-function-declaration]
```

SECCIÓN 05

La semilla

05

`srand` elige la semilla

seed.c

```c
srand((unsigned) seed);

for (int i = 1; i <= 5; i++) {
    printf("%d\n", rand());
}
```

- Recibe un `unsigned int`.
- Se llama **una sola vez**, antes de `rand`.
- La igualdad vale en una misma implementación. Otro sistema puede dar otros números.

del entorno del curso

```
$ echo 42 | build/seed
71876166
708592740
1483128881
907283241
442951012
$ echo 42 | build/seed
71876166
708592740
1483128881
907283241
442951012
```

¿Cuándo fijar la semilla?

Conviene

Para **repetir un resultado**. Si un programa falla con ciertos números al azar, con semilla fija el error aparece igual cada vez y se puede buscar.

No conviene

Cuando cada corrida debe ser distinta. Un juego cuyo dado saca siempre lo mismo no es un juego.

En nuestras pruebas

No se comparan números exactos entre sistemas. Se comprueban propiedades: el rango, y que dos corridas iguales den lo mismo.

Sembrar con la hora

time_seed.c

```c
srand((unsigned) time(NULL));
```

- `time`, de `time.h`, da la hora como un número: aquí, los segundos desde 1970.
- `NULL`: “no hay dónde más guardarla”. Solo interesa lo que devuelve.
- `sleep 1` espera un segundo. `;` ejecuta una orden tras otra.

cambia en cada corrida

```
$ build/time_seed; build/time_seed; sleep 1; build/time_seed
6 6 3 1 3 1 2 5 5 4
6 6 3 1 3 1 2 5 5 4
1 5 5 3 1 5 3 1 4 1
```

Las dos primeras son iguales: empezaron en el mismo segundo, con la misma semilla.

Error común: `srand` dentro del ciclo

errors/srand_in_loop.c

```c
for (int i = 1; i <= 10; i++) {
    srand((unsigned) time(NULL));
    printf("%d ", 1 + rand() % 6);
}
```

cambia en cada corrida

```
$ build/srand_in_loop
1 1 1 1 1 1 1 1 1 1
```

No lo hace más aleatorio

Cada `srand` reinicia la secuencia. Dentro del mismo segundo, cada vuelta recibe la misma semilla y el dado da el mismo número. `srand` va una vez, antes del ciclo.

Tampoco sirve para seguridad: quien conoce la fórmula y observa algunos números puede calcular los siguientes.

SECCIÓN 06

Un número en un rango

06

De `rand` a un dado

Expresión	Da
`rand()`	de 0 a `RAND_MAX`
`rand()% 6`	de 0 a 5
`1 + rand()% 6`	de 1 a 6

errors/die_off_by_one.c

```
$ build/die_off_by_one
Menor: 0
Mayor: 5
```

Sin el `1 +`: un “dado” que da 0 y nunca da 6, con 100 000 lanzamientos.

Fórmula general: entre `min` y `max`, ambos incluidos

`min + rand()% (max - min + 1)`

Hay `max - min + 1` valores. De 0 a 20 son 21, no 20: salirse por uno.

La fórmula en un programa

range.c

```c
printf("%d ", min + rand() % (max - min + 1));
```

del entorno del curso: lo que vale siempre es el rango

```
$ echo 1 6 42 | build/range
1 1 6 4 5 5 2 1 6 6
$ echo 0 20 42 | build/range
12 9 17 18 7 1 10 3 11 14
$ echo -5 5 42 | build/range
5 4 -2 2 4 0 -2 -3 3 4
```

En pseudocódigo, `azar(a, b)` será un entero al azar entre `a` y `b`, ambos incluidos.

¿Todas las caras salen igual?

del entorno del curso

```
$ echo 42 60 | build/die_counts
Cara 1: 13
Cara 2: 6
Cara 3: 6
Cara 4: 13
Cara 5: 8
Cara 6: 14
```

Esperábamos unos 10 por cara. Una muestra pequeña varía mucho.

del entorno del curso

```
$ echo 42 600000 | build/die_counts
Cara 1: 99986
Cara 2: 99434
Cara 3: 100661
Cara 4: 99990
Cara 5: 100184
Cara 6: 99745
```

Esperábamos unos 100 000. Todas quedan a menos de 1 %.

Son seis contadores y un `switch`: una buena razón para querer arreglos, en la semana 9.

Una desviación pequeña: el sesgo

Imaginemos un `rand` que solo diera del 0 al 7:

`rand()`	0	1	2	3	4	5	6	7
`rand()% 6`	0	1	2	3	4	5	0	1

- El 0 y el 1 salen por dos caminos. Es un **sesgo** *(bias)*: 8 no se reparte exacto entre 6.
- Con nuestro `RAND_MAX`: dos caras reciben 357913942 números y las otras cuatro, 357913941.
- Uno en cientos de millones: para un dado, no importa. Importaría si el rango fuera comparable con `RAND_MAX`.

SECCIÓN 07

Simular

07

La respuesta exacta: recorrer los 36 casos

dice_exact.c

```c
for (int target = 2; target <= 12; target++) {
    int ways = 0;

    for (int d1 = 1; d1 <= 6; d1++) {
        for (int d2 = 1; d2 <= 6; d2++) {
            if (d1 + d2 == target) {
                ways++;
            }
        }
    }
    printf("Suma %2d: %d de 36, %.4f\n", target
      , ways, (double) ways / 36);
}
```

terminal

```
$ build/dice_exact
Suma  2: 1 de 36, 0.0278
Suma  3: 2 de 36, 0.0556
Suma  4: 3 de 36, 0.0833
Suma  5: 4 de 36, 0.1111
Suma  6: 5 de 36, 0.1389
Suma  7: 6 de 36, 0.1667
Suma  8: 5 de 36, 0.1389
Suma  9: 4 de 36, 0.1111
Suma 10: 3 de 36, 0.0833
Suma 11: 2 de 36, 0.0556
Suma 12: 1 de 36, 0.0278
```

Sumar 7: 6 de 36 = 1/6.

Tres ciclos anidados: se leen igual. El `if` corre $11 \times 6 \times 6 = 396$ veces.

La respuesta por simulación

pseudocódigo

```
algoritmo estimar_suma_7(intentos)
    aciertos ← 0
    para i desde 1 hasta intentos hacer
        d1 ← azar(1, 6)
        d2 ← azar(1, 6)
        si d1 + d2 = 7 entonces
            aciertos ← aciertos + 1
        fin si
    fin para
    escribir aciertos / intentos
fin algoritmo
```

Simulación

Repetir un experimento muchas veces con números aleatorios y mirar con qué frecuencia ocurre el resultado.

Se usa cuando hay demasiados casos para recorrerlos, o reglas demasiado complicadas para hacer la cuenta.

Cinco intentos a mano

i	d1	d2	Suma	aciertos
1	3	4	7	1
2	6	2	8	1
3	5	2	7	2
4	1	1	2	2
5	6	1	7	3

- Dados elegidos para el ejemplo; no salen de un programa.
- Estimación: $3/5 = 0.6$, muy lejos de 0.1667.
- Con 5 intentos solo son posibles 0, 0.2, 0.4, 0.6, 0.8 y 1.

En C

dice_seven.c

```c
srand((unsigned) seed);

for (int i = 1; i <= trials; i++)
    {
    int d1 = 1 + rand() % 6;
    int d2 = 1 + rand() % 6;

    if (d1 + d2 == 7) {
        hits++;
    }
}
```

del entorno del curso

```
$ echo 42 100 | build/dice_seven
100 intentos, 22 sumaron 7, estimación 0.2200
$ echo 42 100 | build/dice_seven
100 intentos, 22 sumaron 7, estimación 0.2200
$ echo 7 100 | build/dice_seven
100 intentos, 23 sumaron 7, estimación 0.2300
```

- Misma semilla, lo mismo. Otra semilla, otra estimación.
- Las dos, bastante lejos de 0.1667.

Error común: dividir dos enteros

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/int_probability.c -o build/int_probability
$ build/int_probability
Estimación: 0.0000
```

errors/int_probability.c

```
1  double estimate = hits / trials;
```

Un millón de intentos, y responde cero

`hits / trials` es una división entera: si `hits` es menor que `trials`, da 0. El `double` llega tarde. Lo correcto es `(double) hits / trials`.

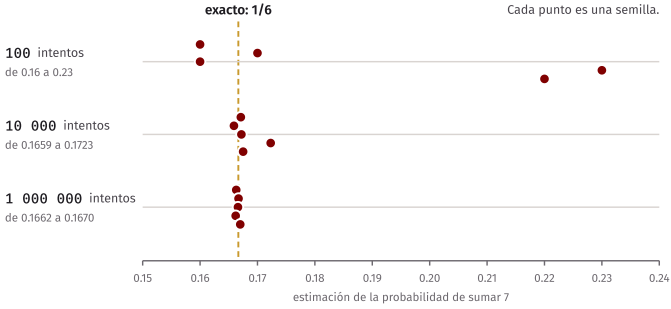
Cinco semillas, tres cantidades de intentos

Semilla	100	10 000	1 000 000
1	0.1600	0.1671	0.1663
2	0.1700	0.1659	0.1667
3	0.1600	0.1672	0.1666
7	0.2300	0.1723	0.1662
42	0.2200	0.1675	0.1670
Peor error	0.0633	0.0056	0.0005

Valores del entorno del curso; en otro sistema serán otros, con el mismo comportamiento. Peor error: la mayor distancia a 1/6.

Para reducir el error típico (su tamaño habitual) a la décima parte, hacen falta cerca de **100 veces más** intentos.

La estimación varía menos con más intentos



- Cada punto es una semilla de la tabla anterior. Una estimación **no es un valor exacto**.
- Con 100 intentos solo salen múltiplos de 0.01: nunca podría dar 1/6.
- El peor error de la tabla no es un límite: otra semilla podría dar uno mayor.
- Se compara con una tolerancia, nunca con ==. La tolerancia depende de los intentos.

SECCIÓN 08

Laboratorio

08

Laboratorio: tres programas y una pregunta

Archivo	Qué hace
`triangle.c`	Triángulo de asteriscos de altura `N`. Hagan primero la tabla del patrón.
`primes.c`	Los primos hasta `N`, uno por línea: exactamente dos divisores.
`dice_estimate.c`	Lee una semilla y estima sumar 7 con 100, 10 000 y 1 000 000 de intentos, con su error.

La pregunta

Corran `dice_estimate` dos veces con la misma semilla y una con otra. ¿Qué cambia? Escríbanlo en un comentario. Comprueben con `make test-lab`.

Pistas para el tercer programa

- El ciclo de `dice_seven.c` va dentro de otro, que recorre las tres cantidades:

```
for (int trials = 100; trials <= 1000000; trials *= 100)
```

- `trials *= 100` abrevia `trials = trials * 100`.
- `hits` vuelve a 0 para cada cantidad: ¿en qué nivel va?
- El error es la distancia sin signo a `1.0 / 6`: `fabs`, de `math.h`.
- `srand` va una sola vez, al principio.

Para la próxima clase

- Hoy, en el laboratorio: `triangle`, `primes` y `dice_estimate`.
- Leer los apuntes de la semana 7: funciones y pruebas.

Preguntas

Dos ciclos anidados de *N* vueltas: ¿cuántas veces corre la línea interior? ¿Y si *N* se duplica?
¿Por qué dos corridas con la misma semilla dan lo mismo? ¿Cuándo conviene y cuándo no?
¿Cómo obtendrían un número entre 1 y 6 con `rand`?
Estiman una probabilidad con 100 y con 10 000 intentos. ¿En cuál confían más?