

UNIVERSIDAD NACIONAL DE INGENIERÍA

Introducción a la Computación

Decisiones y lógica

`if`, `switch`, lógica booleana y compuertas

Gerson Garrido

Semana 4

Hoy

1. Elegir un camino: pseudocódigo y diagramas de flujo
2. Comparar: operadores relacionales
3. `if`, `else` y `else if`. Revisar `scanf` por fin
4. `switch`, `case` y `break`
5. Lógica: `&&`, `||`, `!`, cortocircuito y De Morgan
6. Compuertas y un sumador de un bit

> **Lectura**
>
> Apuntes de la semana 4. Código en `code/week-04`.

SECCIÓN 01

Elegir un camino

01

Una regla de una tienda

pseudocódigo

```
algoritmo precio_con_descuento()
    leer total
    si total ≥ 10000 entonces
        total ← total - total div 10
    fin si
    escribir total
fin algoritmo
```

- Desde 100 soles, 10 % de descuento.
- El dinero va en céntimos: 100 soles son 10000.
- La línea del `si` tiene una **condición**: una pregunta que solo puede ser cierta o falsa.

La seguimos a mano

Paso	S/ 120.50	S/ 80.00
`leer total`	12050	8000

La seguimos a mano

Paso	S/ 120.50	S/ 80.00
`leer total`	12050	8000
¿`total` $\geq$ `10000`?	sí	no

La seguimos a mano

Paso	S/ 120.50	S/ 80.00
`leer total`	12050	8000
¿`total` $\geq$ `10000`?	sí	no
`total` $\leftarrow$ `total - total div 10`	12050 − 1205 = 10845	no se ejecuta

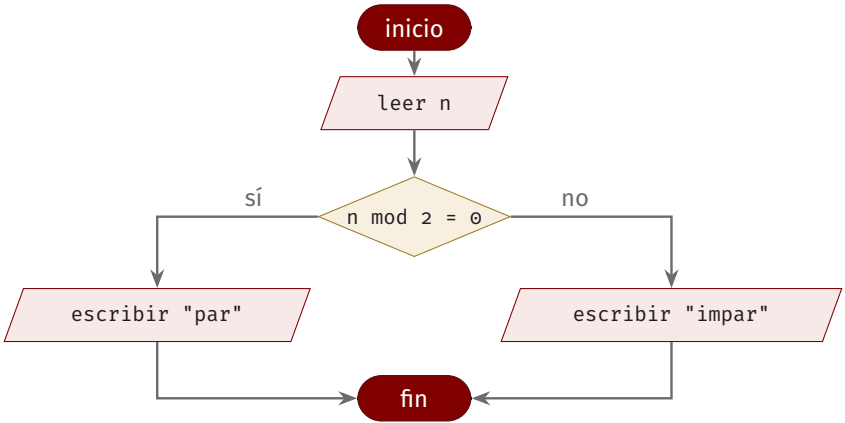
La seguimos a mano

Paso	S/ 120.50	S/ 80.00
`leer total`	12050	8000
¿`total` $\geq$ `10000`?	sí	no
`total` $\leftarrow$ `total - total div 10`	12050 − 1205 = 10845	no se ejecuta
`escribir total`	10845	8000

Idea clave

En cada ejecución se recorre un solo camino. Para probar un programa con decisiones hay que probar **cada camino**.

En un diagrama de flujo: el rombo



El camino se abre en el rombo y se cierra más abajo: eso marcan `si` y `fin si`.

SECCIÓN 02

Comparar

02

Operadores relacionales

Pseudo.	C
$=$	`==`
$\neq$	`!=`
$<$	`<`
$\leq$	`<=`
$>$	`>`
$\geq$	`>=`

terminal

```
$ build/relational
a > b  vale 1
a < b  vale 0
a == b vale 0
a != b vale 1
a >= 5 vale 1
a <= 4 vale 0
```

Con `a = 5` y `b = 3`. Una comparación es una expresión: da un `int`, 1 o 0.

Qué cuenta como verdadero

truthy.c

```c
if (n) {
    printf("%d cuenta como verdadero\n
     ", n);
} else {
    printf("%d cuenta como falso\n", n
     );
}
```

terminal

```
$ echo -7 | build/truthy
-7 cuenta como verdadero
$ echo 0 | build/truthy
0 cuenta como falso
```

Regla de C

0 es falso. Cualquier otro número es verdadero, también los negativos.

`echo -7 | programa` le pasa `-7` a la entrada, como si se escribiera con el teclado.

SECCIÓN 03

if y else

03



El descuento en C

discount.c

```c
int main(void) {
    int total;
    if (scanf("%d", &total) != 1) {
        printf("Entrada inválida\n");
        return 1;
    }
    if (total < 0) {
        printf("El total no puede ser negativo\n");
        return 1;
    }
    if (total >= 10000) {
        total = total - total / 10;
    }
    printf("A pagar: %d.%02d soles\n", total / 100,
      total % 100);
    return 0;
}
```

- **Primer `if`:** ¿se pudo leer?

El descuento en C

discount.c

```c
int main(void) {
    int total;
    if (scanf("%d", &total) != 1) {
        printf("Entrada inválida\n");
        return 1;
    }
    if (total < 0) {
        printf("El total no puede ser negativo\n");
        return 1;
    }
    if (total >= 10000) {
        total = total - total / 10;
    }
    printf("A pagar: %d.%02d soles\n", total / 100,
      total % 100);
    return 0;
}
```

- **Primer `if`:** ¿se pudo leer?
- **Segundo:** ¿el dato tiene sentido?

El descuento en C

discount.c

```c
int main(void) {
    int total;
    if (scanf("%d", &total) != 1) {
        printf("Entrada inválida\n");
        return 1;
    }
    if (total < 0) {
        printf("El total no puede ser negativo\n");
        return 1;
    }
    if (total >= 10000) {
        total = total - total / 10;
    }
    printf("A pagar: %d.%02d soles\n", total / 100,
      total % 100);
    return 0;
}
```

- **Primer `if`:** ¿se pudo leer?
- **Segundo:** ¿el dato tiene sentido?
- **Tercero:** la regla.

El descuento en C

discount.c

```c
int main(void) {
    int total;
    if (scanf("%d", &total) != 1) {
        printf("Entrada inválida\n");
        return 1;
    }
    if (total < 0) {
        printf("El total no puede ser negativo\n");
        return 1;
    }
    if (total >= 10000) {
        total = total - total / 10;
    }
    printf("A pagar: %d.%02d soles\n", total / 100,
      total % 100);
    return 0;
}
```

- **Primer `if`:** ¿se pudo leer?
- **Segundo:** ¿el dato tiene sentido?
- **Tercero:** la regla.
- `%02d`: al menos 2 cifras, con ceros.

Revisar `scanf`, desde hoy siempre

- `scanf` devuelve cuántos datos leyó.
- `abc` da 0; sin datos da un negativo.
- Por eso se pregunta `!= 1`.
- `return 1` termina con un **estado de salida** distinto de 0: algo falló.
- Sin ciclos no se puede volver a pedir: avisamos y terminamos.
- 9999 y 10000: **prueben los bordes**.

terminal

```
$ echo 12050 | build/discount
A pagar: 108.45 soles
$ echo 9999 | build/discount
A pagar: 99.99 soles
$ echo 10000 | build/discount
A pagar: 90.00 soles
$ echo abc | build/discount; echo $?
Entrada inválida
1
```

Lo que la revisión no atrapa

terminal

```
$ echo 15.5 | build/parity
15 es impar
$ echo 12abc | build/parity
12 es par
$ echo 99999999999 | build/parity
1215752191 es impar
```

- `%d` lee cifras y deja el resto sin leer.
- Un número que no cabe en `int` es comportamiento indefinido: ese último valor depende del sistema.
- Por ahora nos basta con revisar `scanf`. Más adelante leeremos con más cuidado.

if y else

parity.c

```c
if (n % 2 == 0) {
    printf("%d es par\n", n);
} else {
    printf("%d es impar\n", n);
}
```

terminal

```
$ echo -4 | build/parity
-4 es par
```

if y else

parity.c

```c
if (n % 2 == 0) {
    printf("%d es par\n", n);
} else {
    printf("%d es impar\n", n);
}
```

terminal

```
$ echo -4 | build/parity
-4 es par
```

Error común: `n % 2 == 1`

En C el resto lleva el signo del dividendo: `-3 % 2` vale `-1`. Con negativos esa pregunta falla, y `gcc` no avisa.

El error clásico: = en lugar de ==

errors/assign_in_if.c

```c
int x = 5;
if (x = 0) {
    printf("cero\n");
} else {
    printf("otro\n");
}
printf("x vale %d\n", x);
```

El error clásico: = en lugar de ==

errors/assign_in_if.c

```c
int x = 5;
if (x = 0) {
    printf("cero\n");
} else {
    printf("otro\n");
}
printf("x vale %d\n", x);
```

1. x empieza en 5.

El error clásico: = en lugar de ==

errors/assign_in_if.c

```c
int x = 5;
if (x = 0) {
    printf("cero\n");
} else {
    printf("otro\n");
}
printf("x vale %d\n", x);
```

1. `x` empieza en 5.
2. `x = 0` guarda 0 en `x`.

El error clásico: = en lugar de ==

errors/assign_in_if.c

```c
int x = 5;
if (x = 0) {
    printf("cero\n");
} else {
    printf("otro\n");
}
printf("x vale %d\n", x);
```

1. `x` empieza en 5.
2. `x = 0` guarda 0 en `x`.
3. La expresión vale 0: falso.

El error clásico: = en lugar de ==

errors/assign_in_if.c

```c
int x = 5;
if (x = 0) {
    printf("cero\n");
} else {
    printf("otro\n");
}
printf("x vale %d\n", x);
```

1. `x` empieza en 5.
2. `x = 0` guarda 0 en `x`.
3. La expresión vale 0: falso.
4. Se ejecuta el `else`: `otro`.

El error clásico: = en lugar de ==

errors/assign_in_if.c

```c
int x = 5;
if (x = 0) {
    printf("cero\n");
} else {
    printf("otro\n");
}
printf("x vale %d\n", x);
```

1. `x` empieza en 5.
2. `x = 0` guarda 0 en `x`.
3. La expresión vale 0: falso.
4. Se ejecuta el `else`: `otro`.
5. Y `x` ya no es 5. La condición no preguntó nada.

El error clásico: = en lugar de ==

errors/assign_in_if.c

```c
int x = 5;
if (x = 0) {
    printf("cero\n");
} else {
    printf("otro\n");
}
printf("x vale %d\n", x);
```

1. `x` empieza en 5.
2. `x = 0` guarda 0 en `x`.
3. La expresión vale 0: falso.
4. Se ejecuta el `else`: `otro`.
5. Y `x` ya no es 5. La condición no preguntó nada.

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/assign_in_if.c -o build/assign_in_if
errors/assign_in_if.c:7:9: warning: suggest parentheses around assignment used as truth value [-Wparentheses]
$ build/assign_in_if
otro
x vale 0
```

Las llaves importan

errors/misleading_indent.c

```c
if (grade < 0)
    printf("Nota negativa\n");
    printf("Rechazada\n");
```

terminal

```
$ echo 15 | build/misleading_indent
Rechazada
```

errors/semicolon_if.c

```c
if (n > 0);
printf("%d es positivo\n", n);
```

terminal

```
$ echo -3 | build/semicolon_if
-3 es positivo
```

Regla del curso: sin llaves, el `if` gobierna una sola instrucción. Siempre llaves.

Una variable vive en su bloque

errors/block_scope.c

```c
if (n > 0) {
    int twice = n * 2;
    printf("%d\n", twice);
}
printf("%d\n", twice);
```

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/block_scope.c -o build/block_scope
errors/block_scope.c:11:20: error: ‘twice’ undeclared (first use in this function)
```

Al cerrar la llave, `twice` deja de existir. Si la necesitan después, declárenla antes del `if`.

SECCIÓN 04

Varias opciones

04

`else if`: una cadena de preguntas

pseudocódigo

```
si promedio ≥ 17 entonces
    escribir "Excelente"
sino si promedio ≥ 14 entonces
    escribir "Bueno"
sino si promedio ≥ 11 entonces
    escribir "Regular"
sino
    escribir "Insuficiente"
fin si
```

mention.c

```c
} else if (average >= 17) {
    printf("Excelente\n");
} else if (average >= 14) {
    printf("Bueno\n");
} else if (average >= 11) {
    printf("Regular\n");
} else {
    printf("Insuficiente\n");
}
```

La escala es un ejemplo inventado. El programa completo rechaza primero lo que no está entre 0 y 20.

Gana la primera verdadera

Sigamos el promedio **15**:

Condición	¿Se cumple?	Qué pasa
`average < 0 || average > 20`	no	sigue

Gana la primera verdadera

Sigamos el promedio **15**:

Condición	¿Se cumple?	Qué pasa
`average < 0 || average > 20`	no	sigue
`average >= 17`	no	sigue

Gana la primera verdadera

Sigamos el promedio **15**:

Condición	**¿Se cumple?**	**Qué pasa**
`average < 0 || average > 20`	no	sigue
`average >= 17`	no	sigue
`average >= 14`	sí	escribe `Bueno` y sale

Gana la primera verdadera

Sigamos el promedio **15**:

Condición	**¿Se cumple?**	**Qué pasa**
`average < 0 || average > 20`	no	sigue
`average >= 17`	no	sigue
`average >= 14`	sí	escribe `Bueno` y sale
`average >= 11`	no se evalúa	

El orden importa

Si `average >= 11` va primero, un 18 sale `Regular`: las demás nunca se alcanzan. Y `gcc` no avisa.

Decisiones anidadas

max_of_three.c

```c
if (a > b) {
    if (a > c) {
        printf("%d\n", a);
    } else {
        printf("%d\n", c);
    }
} else {
    if (b > c) {
        printf("%d\n", b);
    } else {
        printf("%d\n", c);
    }
}
```

Con `4 9 2`:

Pregunta	Respuesta

Decisiones anidadas

max_of_three.c

```c
if (a > b) {
    if (a > c) {
        printf("%d\n", a);
    } else {
        printf("%d\n", c);
    }
} else {
    if (b > c) {
        printf("%d\n", b);
    } else {
        printf("%d\n", c);
    }
}
```

Con `4 9 2`:

Pregunta	Respuesta
a > b: 4 > 9	no

Decisiones anidadas

max_of_three.c

```c
if (a > b) {
    if (a > c) {
        printf("%d\n", a);
    } else {
        printf("%d\n", c);
    }
} else {
    if (b > c) {
        printf("%d\n", b);
    } else {
        printf("%d\n", c);
    }
}
```

Con `4 9 2`:

Pregunta	Respuesta
`a > b`: `4 > 9`	no
`b > c`: `9 > 2`	sí: se escribe `b`

Decisiones anidadas

max_of_three.c

```c
if (a > b) {
    if (a > c) {
        printf("%d\n", a);
    } else {
        printf("%d\n", c);
    }
} else {
    if (b > c) {
        printf("%d\n", b);
    } else {
        printf("%d\n", c);
    }
}
```

Con `4 9 2`:

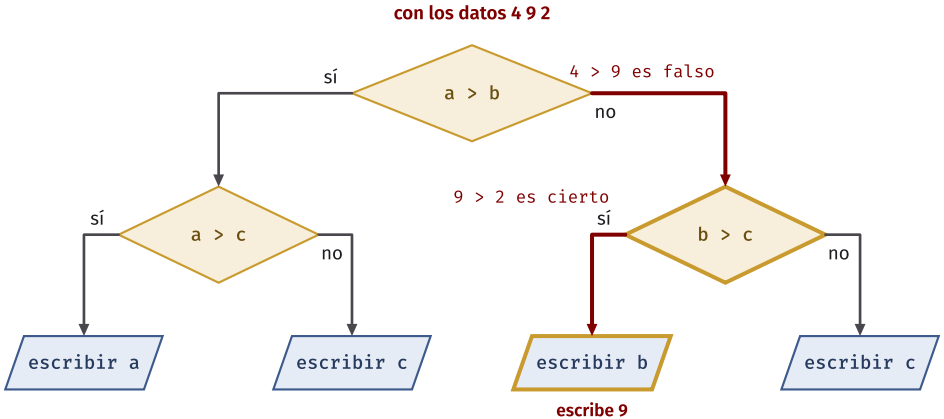
Pregunta	Respuesta
`a > b`: `4 > 9`	no
`b > c`: `9 > 2`	sí: se escribe `b`

terminal

```
$ echo 4 9 2 | build/max_of_three
9
```

Hay cuatro caminos: al menos cuatro pruebas, y también empates como `5 5 1`.

Cuatro caminos, uno por ejecución



¿De quién es este `else`?

errors/dangling_else.c

```c
if (a > 0)
    if (b > 0)
        printf("Ambos positivos\n");
else
    printf("a no es positivo\n");
```

Sin llaves, el `else` va con el `if` más cercano sin `else`. Aquí, con `if (b > 0)`. La sangría engaña.

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/dangling_else.c -o build/dangling_else
errors/dangling_else.c:12:8: warning: suggest explicit braces to avoid ambiguous ‘else’ [-Wdangling-else]
$ echo 5 -1 | build/dangling_else
a no es positivo
```

Comparar `double`: lo de la semana pasada

compare_double.c

```c
double sum = 0.1 + 0.2;

if (sum == 0.3) {
    printf("Con ==: iguales\n");
} else {
    printf("Con ==: distintos\n");
}

if (fabs(sum - 0.3) < 0.000001) {
    printf("Con tolerancia: iguales\n");
} else {
    printf("Con tolerancia: distintos\n");
}
```

terminal

```
$ build/compare_double
Con ==: distintos
Con tolerancia: iguales
```

No se pregunta si son iguales, sino si están **muy cerca**. La tolerancia depende del problema.

SECCIÓN 05

switch

05



`switch`: elegir según un valor

combo.c

```c
switch (option) {
    case 1:
        price = 1200;
        break;
    case 2:
        price = 1500;
        break;
    case 3:
        price = 1800;
        break;
    default:
        printf("Opción inválida\n");
        return 1;
}
```

- Calcula el valor de `option`.
- **Salta** al `case` con ese valor.
- Sigue línea por línea hasta un `break`.
- Si ninguno coincide, salta a `default`.

terminal

```
$ echo 2 | build/combo
Precio: 1500 céntimos
$ echo 9 | build/combo; echo $?
Opción inválida
1
```

Olvidar el `break`: un `case` es solo una etiqueta

errors/combo_no_break.c

```c
switch (option) {
    case 1:
        price = 1200;
    case 2:
        price = 1500;
    case 3:
        price = 1800;
        break;
```

Con `option = 1`:

Línea	price

Olvidar el `break`: un `case` es solo una etiqueta

errors/combo_no_break.c

```c
switch (option) {
    case 1:
        price = 1200;
    case 2:
        price = 1500;
    case 3:
        price = 1800;
        break;
```

Con `option = 1`:

Línea	price
`price = 1200;`	1200

Olvidar el `break`: un `case` es solo una etiqueta

errors/combo_no_break.c

```c
switch (option) {
    case 1:
        price = 1200;
    case 2:
        price = 1500;
    case 3:
        price = 1800;
        break;
```

Con `option = 1`:

Línea	price
`price = 1200;`	1200
`price = 1500;`	1500

Olvidar el `break`: un `case` es solo una etiqueta

errors/combo_no_break.c

```c
switch (option) {
    case 1:
        price = 1200;
    case 2:
        price = 1500;
    case 3:
        price = 1800;
        break;
```

Con `option = 1`:

Línea	`price`
`price = 1200;`	1200
`price = 1500;`	1500
`price = 1800;`	1800

Olvidar el `break`: un `case` es solo una etiqueta

errors/combo_no_break.c

```c
switch (option) {
    case 1:
        price = 1200;
    case 2:
        price = 1500;
    case 3:
        price = 1800;
        break;
```

Con `option = 1`:

Línea	`price`
`price = 1200;`	1200
`price = 1500;`	1500
`price = 1800;`	1800
`break;`	1800

Olvidar el `break`: un `case` es solo una etiqueta

errors/combo_no_break.c

```c
switch (option) {
    case 1:
        price = 1200;
    case 2:
        price = 1500;
    case 3:
        price = 1800;
        break;
```

Con `option = 1`:

Línea	price
`price = 1200;`	1200
`price = 1500;`	1500
`price = 1800;`	1800
`break;`	1800

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/combo_no_break.c -o build/combo_no_break
errors/combo_no_break.c:15:19: warning: this statement may fall through [-Wimplicit-fallthrough=]
$ echo 1 | build/combo_no_break
Precio: 1800 céntimos
```

Sin `break`, la ejecución sigue de largo

```
switch (option) {
   case 1:
      price = 1200;
   case 2:
      price = 1500;
   case 3:
      price = 1800;
      break;
   default:
      printf("Opción inválida\n");
      return 1;
}
```



option

1

price

1200

salta a case 1 y guarda 1200

Sin `break`, la ejecución sigue de largo

```
switch (option) {
   case 1:
      price = 1200;
   case 2:
      price = 1500;
   case 3:
      price = 1800;
      break;
   default:
      printf("Opción inválida\n");
      return 1;
}
```



option

1

price

1500

case 2 es solo una etiqueta: sigue y guarda 1500

Sin `break`, la ejecución sigue de largo

```
switch (option) {
   case 1:
      price = 1200;
   case 2:
      price = 1500;
   case 3:
      price = 1800;
      break;
   default:
      printf("Opción inválida\n");
      return 1;
}
```

option

1

price

1800

case 3 tampoco la detiene: guarda 1800

Sin `break`, la ejecución sigue de largo

```
switch (option) {
   case 1:
      price = 1200;
   case 2:
      price = 1500;
   case 3:
      price = 1800;
      break;
   default:
      printf("Opción inválida\n");
      return 1;
}
```

sale del switch

option

1

price

1800

el break, por fin, la saca.
Precio: 1800 céntimos

Etiquetas seguidas: a propósito

month_days.c

```c
switch (month) {
    case 4:
    case 6:
    case 9:
    case 11:
        days = 30;
        break;
    ...
    default:
        printf("Mes inválido
      \n");
        return 1;
}
```

terminal

```
$ echo 4 | build/month_days
30 días
$ echo 13 | build/month_days
Mes inválido
```

Sin instrucciones entre etiquetas

Todas llevan al mismo código, y `gcc` no avisa: está claro que es a propósito.

Este programa supone un año no bisiesto.

`switch` con caracteres

vowel.c

```c
if (scanf(" %c", &letter) != 1) {
```

vowel.c

```c
switch (letter) {
    case 'a':
    case 'e':
    ...
    case 'U':
        printf("Vocal\n");
        break;
    default:
        printf("No es vocal\n");
}
```

- `char` es un número pequeño: `switch` funciona.
- El espacio antes de `%c` salta espacios y saltos de línea. `%c` solo no lo hace.

terminal

```
$ echo á | build/vowel
No es vocal
```

`á` ocupa 2 bytes en UTF-8, y `%c` lee solo el primero.

¿`switch` o `if`?

Situación	Conviene
Un entero o `char` contra varias constantes	`switch`
Rangos	`if`
Valores `double`	`if`
Varias variables	`if`

Límites de `switch`

Solo enteros. Cada `case` es una constante. `case >= 11:` no existe.

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/switch_double.c -o build/switch_double
errors/switch_double.c:7:13: error: switch quantity not an integer
```

SECCIÓN 06

Lógica booleana

06



Y, o, no

C	Verdadero cuando
`&&`	**ambos** lados lo son
`||`	**al menos uno** lo es
`!`	el valor es falso

En pseudocódigo: `y`, `o`, `no`.
Y el "o" es **inclusivo**: vale también cuando los dos lados son ciertos.

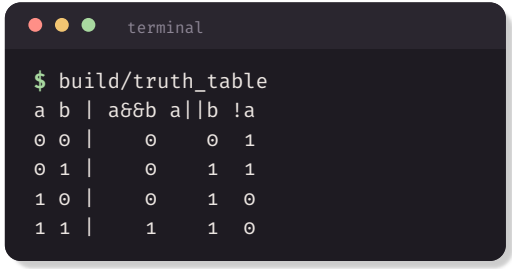


Tabla de verdad
Lista todas las combinaciones de entradas y el resultado de cada una.

Precedencia

Orden	Operadores
1	`!`
2	`* / %`
3	`+ -`
4	`< <= > >=`
5	`== !=`
6	`&&`
7	`||`
8	`=`

Con `x = 5` y `y = 2`:

Expresión	Pasos

Precedencia

Orden	Operadores
1	`!`
2	`* / %`
3	`+ -`
4	`< <= > >=`
5	`== !=`
6	`&&`
7	`||`
8	`=`

Con `x = 5` y `y = 2`:

Expresión	Pasos	
`x + 1 > y * 3`	`6 > 6`	0

Precedencia

Orden	Operadores
1	`!`
2	`* / %`
3	`+ -`
4	`< <= > >=`
5	`== !=`
6	`&&`
7	`||`
8	`=`

Con `x = 5` y `y = 2`:

Expresión	Pasos	
`x + 1 > y * 3`	`6 > 6`	0
`x > 3 && x < 10`	`1 && 1`	1

Precedencia

Orden	Operadores
1	`!`
2	`* / %`
3	`+ -`
4	`< <= > >=`
5	`== !=`
6	`&&`
7	`||`
8	`=`

Con `x = 5` y `y = 2`:

Expresión	Pasos	
`x + 1 > y * 3`	`6 > 6`	0
`x > 3 && x < 10`	`1 && 1`	1
`!(x > 3)`	`!1`	0

Precedencia

Orden	Operadores
1	`!`
2	`* / %`
3	`+ -`
4	`< <= > >=`
5	`== !=`
6	`&&`
7	`||`
8	`=`

Con `x = 5` y `y = 2`:

Expresión	Pasos	
`x + 1 > y * 3`	`6 > 6`	0
`x > 3 && x < 10`	`1 && 1`	1
`!(x > 3)`	`!1`	0

Primero las cuentas, luego comparar, al final combinar.
Ante la duda, paréntesis: `gcc` los pide para `&&` dentro de `||`.

Parece bien, pero no: `0 <= grade <= 20`

- Se agrupa así: `(0 <= grade) <= 20`.
- Lo de adentro vale 0 o 1, y los dos son menores que 20. Siempre es verdadero.
- Lo correcto: `grade >= 0 && grade <= 20`.

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/chained_compare.c -o build/chained_compare
errors/chained_compare.c:12:20: warning: comparison of constant ‘20’ with boolean expression is always true [-Wbool-compare]
errors/chained_compare.c:12:11: warning: comparisons like ‘X<=Y<=Z’ do not have their mathematical meaning [-Wparentheses]
$ echo 25 | build/chained_compare
En rango
```

Parece bien, pero no: `option == 1 || 2`

- Cada lado de `||` es una condición completa.
- El lado derecho es solo `2`, que no es cero: verdadero.
- Lo correcto: `option == 1 || option == 2`.

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/or_constant.c -o build/or_constant
$ echo 7 | build/or_constant
Opción 1 o 2
```

Peor aún

Aquí `gcc` no avisa nada. Solo una prueba lo descubre.

Cortocircuito *(short-circuit)*

- Siempre se evalúa primero la izquierda.
- `A && B`: si `A` es falso, `B` no se evalúa.
- `A || B`: si `A` es verdadero, `B` no se evalúa.
- C lo garantiza: se puede usar a favor.

terminal

```
$ echo 0 | build/short_circuit
No se cumple
```

short_circuit.c

```
1  if (n != 0 && 10 / n > 1) {
```

n	n != 0	10 / n > 1
5	1	2 > 1: 1
20	1	0 > 1: 0
0	0	no se evalúa

Si se invierte: `10 / n > 1 && n != 0`

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=undefined errors/unsafe_division.c -o build/unsafe_division
$ echo 0 | build/unsafe_division
errors/unsafe_division.c:12:12: runtime error: division by zero
```

Comportamiento indefinido

Dividir un entero entre 0 no tiene resultado definido en C. Qué pasa después depende del procesador: en ARM puede seguir; en x86-64 lo habitual es que termine con error.

Idea clave: primero la condición que protege.

Leyes de De Morgan

A	B	`!(A && B)`	`!A || !B`
0	0	1	1
0	1	1	1
1	0	1	1
1	1	0	0

Las columnas coinciden en **todas** las filas: son equivalentes.

Al negar, se intercambian

`!(A && B)` es `!A || !B`

`!(A || B)` es `!A && !B`

Simplificar una condición

Comparación	Negación
`a == b`	`a != b`
`a < b`	`a >= b`
`a > b`	`a <= b`

Lo contrario de < no es >: la igualdad tiene que caer en algún lado.

```
!(grade >= 0 && grade <= 20)
```

Simplificar una condición

Comparación	Negación
`a == b`	`a != b`
`a < b`	`a >= b`
`a > b`	`a <= b`

Lo contrario de < no es >: la igualdad tiene que caer en algún lado.

```
!(grade >= 0 && grade <= 20)
grade < 0 || grade > 20
```

Simplificar una condición

Comparación	Negación
`a == b`	`a != b`
`a < b`	`a >= b`
`a > b`	`a <= b`

Lo contrario de < no es >: la igualdad tiene que caer en algún lado.

```
!(grade >= 0 && grade <= 20)
grade < 0 || grade > 20
```

terminal

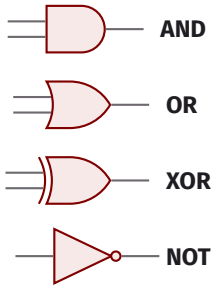
```
$ echo 5 12 | build/de_morgan
!(x >= 0 && y < 10) vale 1
x < 0 || y >= 10    vale 1
```

SECCIÓN 07

Compuertas

07

La lógica como circuito



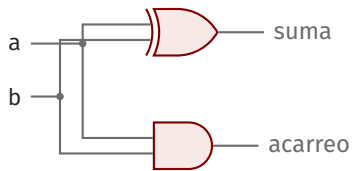
- Un bit es una señal: voltaje alto es 1, bajo es 0.
- Una **compuerta** *(logic gate)* es un circuito de unos pocos transistores: recibe bits y da un bit.
- AND, OR y NOT son `&&`, `||` y `!`.
- **XOR**: 1 si los bits son distintos. Con 0 y 1, es `a != b`.

Sumar dos bits

a	b	suma	acarreo
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Sumar dos bits

a	b	suma	acarreo
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



La suma es XOR. El acarreo es AND.

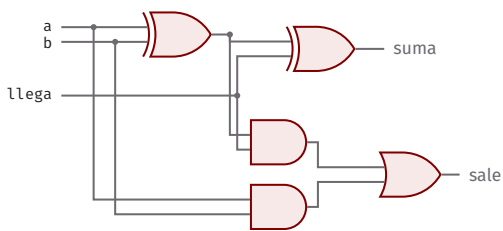
Sumador de un bit, con el acarreo que llega

full_adder.c

```c
int partial = a != b;
int sum = partial != carry_in;
int carry_out = (a && b) || (partial &&
    carry_in);
```

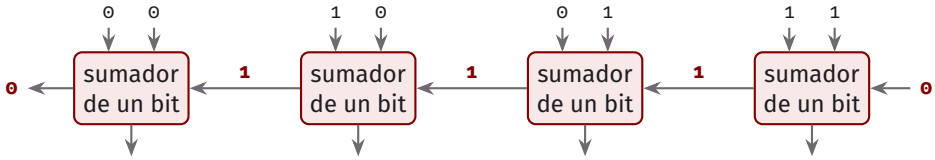
terminal

```
$ echo 1 1 1 | build/full_adder
suma 1, acarreo 1
```



Sale acarreo si `a` y `b` ya eran 1, o si eran distintos y llegó uno.

Cuatro bits en cadena: `0101 + 0011`

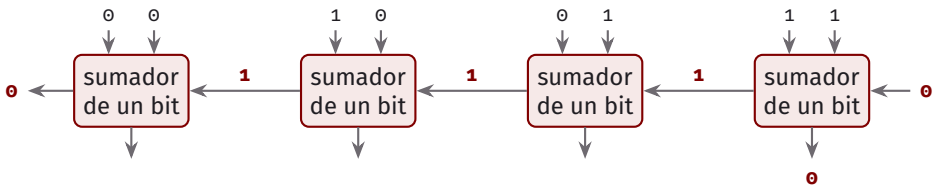


terminal

```
$ echo 0101 0011 | build/adder4
acarreos: 0 1 1 1
suma:     1000
```

5 + 3 = 8. Sin ciclos, `adder4.c` repite la misma cuenta cuatro veces. La próxima semana, ciclos.

Cuatro bits en cadena: `0101 + 0011`

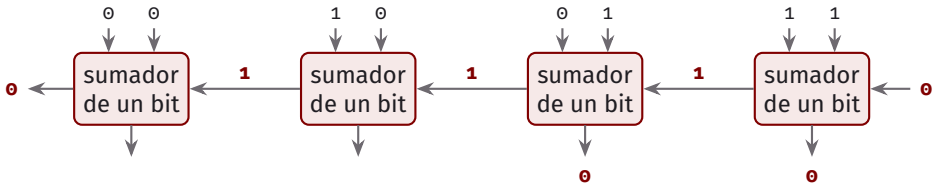


terminal

```
$ echo 0101 0011 | build/adder4
acarreos: 0 1 1 1
suma:     1000
```

5 + 3 = 8. Sin ciclos, `adder4.c` repite la misma cuenta cuatro veces. La próxima semana, ciclos.

Cuatro bits en cadena: `0101 + 0011`

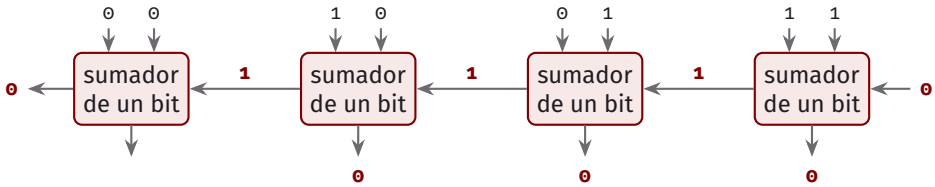


terminal

```
$ echo 0101 0011 | build/adder4
acarreos: 0 1 1 1
suma:     1000
```

5 + 3 = 8. Sin ciclos, `adder4.c` repite la misma cuenta cuatro veces. La próxima semana, ciclos.

Cuatro bits en cadena: `0101 + 0011`

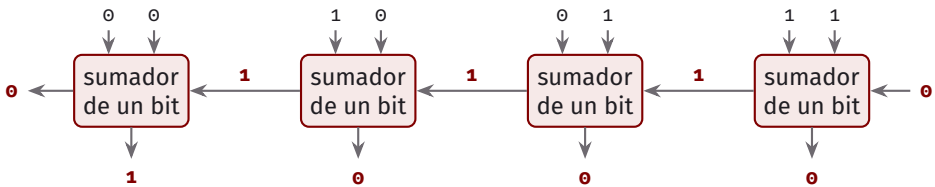


terminal

```
$ echo 0101 0011 | build/adder4
acarreos: 0 1 1 1
suma:     1000
```

5 + 3 = 8. Sin ciclos, `adder4.c` repite la misma cuenta cuatro veces. La próxima semana, ciclos.

Cuatro bits en cadena: `0101 + 0011`



terminal

```
$ echo 0101 0011 | build/adder4
acarreos: 0 1 1 1
suma:     1000
```

5 + 3 = 8. Sin ciclos, `adder4.c` repite la misma cuenta cuatro veces. La próxima semana, ciclos.

Cuando no cabe

terminal

```
$ echo 1000 1000 | build/adder4
acarreos: 1 0 0 0
suma:     0000
$ echo 1111 0001 | build/adder4
acarreos: 1 1 1 1
suma:     0000
```

- Sin signo: 8 + 8 no cabe en 4 bits. Es **overflow**.
- En complemento a dos: $-1 + 1 = 0$ es correcto, aunque salga acarreo.
- `0111 + 0001` da `1000`: 7 + 1 da -8, overflow sin acarreo final.

No cambia la regla de C

El circuito no se queja. Pero para un `int`, el overflow es comportamiento indefinido.

SECCIÓN 08

Laboratorio

08

Laboratorio: tres programas

`grade.c`	Lee una nota de 0 a 20 y dice si aprobó (aquí, con 11 o más). Rechaza lo que está fuera de rango y lo que no es número.
`leap_year.c`	Dice si un año es bisiesto.
`calculator.c`	Calculadora con `switch`, que avisa si se divide entre 0.

Antes de programar

Escriban la lista de datos de prueba: un dato para cada camino y los bordes. Comprueben con `make test-lab`.

Un año bisiesto: cada dato prueba una parte de la regla

- Divisible entre 4: bisiesto.
- **Salvo** si es divisible entre 100: no lo es.
- **Salvo** si es divisible entre 400: sí lo es.

Año	/4	/100	/400	¿Bisiesto?
2026	no	no	no	no
2024	sí	no	no	sí
1900	sí	sí	no	no
2000	sí	sí	sí	sí

La calculadora: `b == 0` con `double`

- Formato: `"%lf%c%lf"`, y se revisa que `scanf` devuelva 3.
- `switch (op)` con `case '+':`, `'-'`, `'*'`, `'/'` y `default`.
- Antes de dividir, un `if` revisa el divisor.

¿No era que `==` con `double` estaba mal?

Está mal para resultados de cuentas que "deberían" ser iguales. Aquí la pregunta es si el divisor es **exactamente** 0, y eso `==` sí lo responde.

Para la próxima clase

- En el laboratorio: **Práctica 1**. Los tres programas son trabajo autónomo.
- Leer los apuntes de la semana 5: ciclos.
- Compilen siempre con `-Wall -Wextra` y lean las advertencias.

Preguntas

¿Qué escribe `if (x = 0) printf("cero"); else printf("otro");`?
Simplifiquen `!(a > 3 && b != 2)`.
Sumen en binario `0110 + 0111`, con el acarreo de cada columna.