

Programación 1

Tema 5

Instrucciones simples y estructuradas



Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza





Índice

- ❑ Instrucciones simples
- ❑ Instrucciones estructuradas



Instrucción

```
<instrucción> ::=  
    <instrucciónSimple> |  
    <instrucciónEstructurada>
```



Instrucciones. Instrucciones simples

- Instrucciones simples
 - De declaración
 - De expresión
 - Asignación
 - Incremento y decremento
 - Entrada y salida
 - Instrucción nula
 - De invocación
 - De devolución de valor



Instrucciones.

Instrucciones estructuradas

- Instrucciones estructuradas
 - Bloques secuenciales de instrucciones
 - Instrucciones condicionales
 - Instrucciones iterativas
 - Bucles *while*
 - Instrucciones iterativas indexadas (bucles *for*)

Instrucciones simples de declaración

- `int x;`
- `int i, j, k;`
- `bool b;`
- `int a = 100;`
- `char c1 = 'h';`
- `bool b = true;`
- `double r2 = 1.5e6;`
- `int n = 4 + 8;`
- `char c = char(int('A') + 1);`
- `boolean esDoce = (n == 12);`
- `double r = sqrt(2.0);`
- `const double PI = 3.141592653589793;`

Instrucciones simples de expresión

□ **Asignación**

- `x = 10;`
- `x = x + 10;`
- `x += 10;`

□ **Incremento y decremento**

- `x++;`
- `x--;`

□ **Entrada y salida**

- ```
cout << "Resultado: " << fixed
 << setprecision(2) << setw(8)
 << 2 * PI * r << endl;
```
- `cin >> n1 >> n2;`

# Instrucciones simples

---

- Nula
  - ;
- Invocación a función
  - `presentarTabla(7);`
  - `intercambiar(m, n);`
  - `ordenar(a, b, c);`
- Devolución de valor en una función
  - `return 0;`
  - `return n;`
  - `return 2 * PI * r;`



# Funciones

---

- Grupo de instrucciones a las que se le da un nombre determinado para ser invocadas desde algún otro punto del programa
- Sintaxis:
  - Declaración
  - Definición
  - Invocación

Más adelante en el curso, no ahora

# Funciones. Sintaxis

---

- $\langle \text{definición-función} \rangle ::=$   
     $\langle \text{tipo} \rangle \langle \text{identificador} \rangle$   
         $\text{"(" } [\langle \text{lista-parámetros} \rangle] \text{"}"$   
     $\langle \text{bloque-secuencial} \rangle$
- $\langle \text{lista-parámetros} \rangle ::=$   
     $\langle \text{parámetro} \rangle \{ \text{","} \langle \text{parámetro} \rangle \}$
- $\langle \text{parámetro} \rangle ::=$   
     $\langle \text{tipo} \rangle \langle \text{identificador} \rangle$
- $\langle \text{bloque-secuencial} \rangle ::=$   
     $\text{"{" } \{ \langle \text{instrucción} \rangle \} \text{"}"$

# Funciones. Sintaxis

---

- $\langle \text{invocación-función} \rangle ::=$   
     $\langle \text{identificador} \rangle$   
     $\text{"(" [ } \langle \text{lista-argumentos} \rangle \text{ ] "}"}$
- $\langle \text{lista-argumentos} \rangle ::=$   
     $\langle \text{argumento} \rangle \{ \text{","} \} \langle \text{argumento} \rangle$
- $\langle \text{argumento} \rangle ::= \langle \text{expresión} \rangle$

# Funciones. Sintaxis

---

- Restricciones a la sintaxis:
  - El identificador de la invocación es el mismo que el de la definición.
  - La lista de parámetros (definición) y la de argumentos (invocación) tienen el mismo número de elementos.
  - El tipo del  $i$ -ésimo argumento en la lista de argumentos de la invocación es el mismo (o es compatible) con el  $i$ -ésimo parámetro de la lista de parámetros de la definición.
  - Si el tipo devuelto es distinto de **void**, el cuerpo de la función debe devolver un dato del tipo adecuado a través de la instrucción **return**.

# Funciones. Ejemplo

|         | a  | b  | c  | d  | e  | f  | g  | h  | ← columnas |
|---------|----|----|----|----|----|----|----|----|------------|
| 8       | 57 | 58 | 59 | 60 | 61 | 62 | 63 | 64 |            |
| 7       | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 |            |
| 6       | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 |            |
| 5       | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 |            |
| 4       | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |            |
| 3       | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |            |
| 2       | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 |            |
| 1       | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  |            |
| ↑ filas |    |    |    |    |    |    |    |    |            |

En ajedrez, queremos calcular el número de escaque a partir del entero que identifica la fila y la letra que identifica la columna.

# Funciones. Ejemplo de definición

```
/* Pre: $1 \leq \text{fila} \leq 8$ y
* $'a' \leq \text{columna} \leq 'h'$.
* Post: Ha devuelto el número de escaque
* (entre 1 y 64) que corresponde a
* la fila y columnas establecidas por
* los parámetros de la función.
*/
int numEscaque(int fila, char columna) {
 return (fila - 1) * 8 + columna - 'a' + 1;
}
```

# Funciones. Ejemplos de invocaciones

```
int numEscaque(int fila, char columna) {
 return (fila - 1) * 8 + (columna - 'a') + 1;
}
...

int primero = numEscaque(1, 'a');
int ultimo = numEscaque(8, 'h');

int fila = 3;
char columna = 'd';
int escaque = numEscaque(fila, columna);

cout << numEscaque(fila + 1, columna - 1) << endl;
```

# Funciones. Ejemplos de invocaciones

```
int numEscaque(int fila, char columna) {
 return (fila - 1) * 8 + (columna - 'a') + 1;
}
...

int primero = numEscaque(1, 'a');
int ultimo = numEscaque(8, 'h');

int fila = 3;
char columna = 'd';
int escaque = numEscaque(fila, columna);

cout << numEscaque(fila + 1, columna - 1) << endl;
```



# Funciones. Ejemplos de invocaciones

```
int numEscaque(int fila, char columna) {
 return (fila - 1) * 8 + (columna - 'a') + 1;
}
...

int primero = numEscaque(1, 'a');
int ultimo = numEscaque(8, 'h');

int fila = 3;
char columna = 'd';
int escaque = numEscaque(fila, columna);

cout << numEscaque(fila + 1, columna - 1) << endl;
```

# Funciones. Ejemplos de invocaciones

```
int numEscaque(int fila, char columna) {
 return (fila - 1) * 8 + (columna - 'a') + 1;
}
...

int primero = numEscaque(1, 'a');
int ultimo = numEscaque(8, 'h');

int fila = 3;
char columna = 'd';
int escaque = numEscaque(fila, columna);

cout << numEscaque(fila + 1, columna - 1) << endl;
```

# Funciones. Ejemplos de invocaciones

```
int numEscaque(int fila, char columna) {
 return (fila - 1) * 8 + (columna - 'a') + 1;
}
```

...

64



```
int primero = numEscaque(1, 'a');
```

```
int ultimo = numEscaque(8, 'h');
```

```
int fila = 3;
```

```
char columna = 'd';
```

```
int escaque = numEscaque(fila, columna);
```

```
cout << numEscaque(fila + 1, columna - 1) << endl;
```

# Funciones. Ejemplos de invocaciones

```
int numEscaque(int fila, char columna) {
 return (fila - 1) * 8 + (columna - 'a') + 1;
}
...

int primero = numEscaque(1, 'a');
int ultimo = numEscaque(8, 'h');

int fila = 3;
char columna = 'd';
int escaque = numEscaque(fila, columna);

cout << numEscaque(fila + 1, columna - 1) << endl;
```

3      'd'

# Funciones. Ejemplos de invocaciones

```
int numEscaque(int fila, char columna) {
 return (fila - 1) * 8 + (columna - 'a') + 1;
}
... 20
```

```
int primero = numEscaque(1, 'a');
int ultimo = numEscaque(8, 'h');
```

```
int fila = 3;
char columna = 'd';
int escaque = numEscaque(fila, columna);
```

```
cout << numEscaque(fila + 1, columna - 1) << endl;
```

# Funciones. Ejemplos de invocaciones

```
int numEscaque(int fila, char columna) {
 return (fila - 1) * 8 + (columna - 'a') + 1;
}
... 20
```

```
int primero = numEscaque(1, 'a');
int ultimo = numEscaque(3, 'h');
```

```
int fil = 3;
char col = 'd';
int escaque = numEscaque(fil, col);
```

```
cout << numEscaque(fil + 1, col - 1) << endl;
```

# Funciones. Ejemplos de invocaciones

```
int numEscaque(int fila, char columna) {
 return (fila - 1)* 8 + (columna - 'a') + 1;
}
...

int primero = numEscaque(1, 'a');
int ultimo = numEscaque(8, 'h');

int fila = 3;
char columna = 'd';
int escaque = numEscaque(fila, columna);

cout << numEscaque(fila + 1, columna - 1) << endl;
```

# Funciones. Ejemplos de invocaciones

```
int numEscaque(int fila, char columna) {
 return (fila - 1) * 8 + (columna - 'a') + 1;
}
... 27
```

```
int primero = numEscaque(1, 'a');
int ultimo = numEscaque(8, 'h');
```

```
int fila = 3;
char columna = 'd';
int escaque = numEscaque(fila, columna);
```

```
cout << numEscaque(fila + 1, columna - 1) << endl;
```

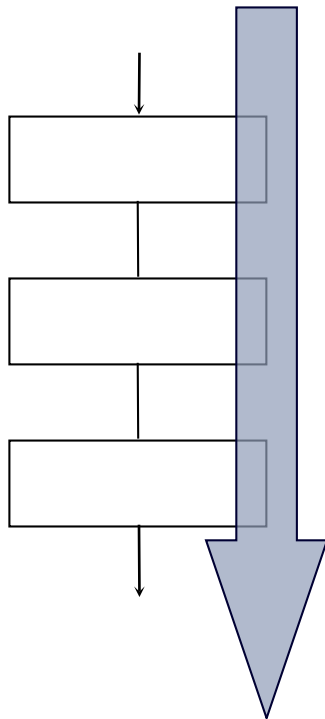


# Instrucciones estructuradas

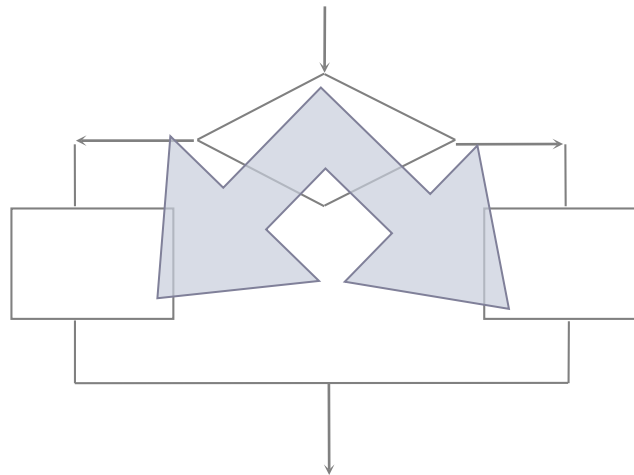
```
<instrucciónEstructurada> ::=
 <bloque-secuencial> |
 <instrucciónCondicional> |
 <instrucciónIterativa> |
 <instrucciónIterativaIndexada>
```

# Instrucciones estructuradas

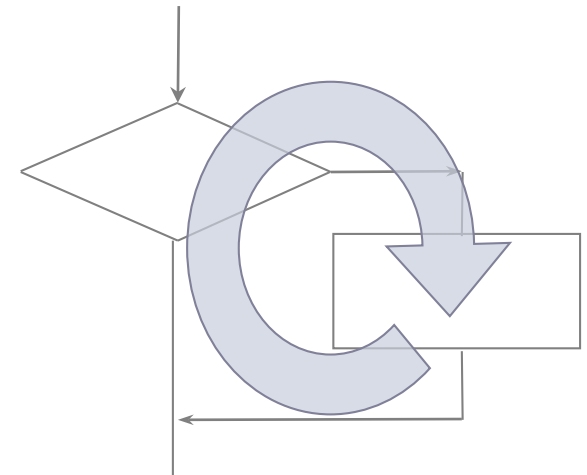
Secuencial



Condicional



Iterativa



# Composición secuencial

```
<bloque-secuencial> ::=
 “{” { <instrucción> } “}”
```

# Composición secuencial

```
int main() {
 // Definición de la constante de cambio
 const double PTAS_POR_EURO = 166.386;

 // Petición y lectura del valor de pesetas
 cout << "Escriba una cantidad en pesetas: " << flush;
 int pesetas;
 cin >> pesetas;

 // Cálculo del equivalente en euros y céntimos
 int centimos = int((pesetas * 100 / PTAS_POR_EURO) + 0.5);
 int euros = centimos / 100;
 centimos = centimos % 100;

 // Escritura de resultados
 cout << "Equivale a " << euros << " euros y " << centimos
 << " céntimos" << endl;
 return 0;
}
```

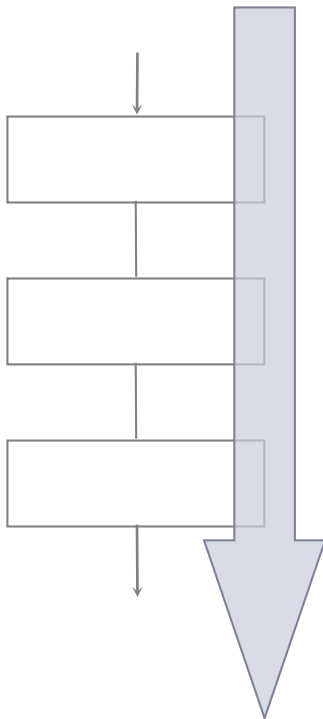
# Composición secuencial

## □ Intercambio de los valores de dos variables

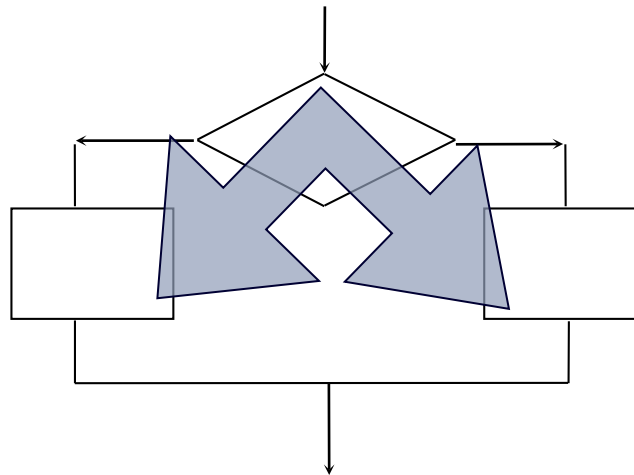
```
{
 int a = ...;
 int b = ...;
 // $a=A_0$ y $b=B_0$
 int temp = a;
 a = b;
 b = temp;
 // $a=B_0$ y $b=A_0$
}
```

# Instrucciones estructuradas

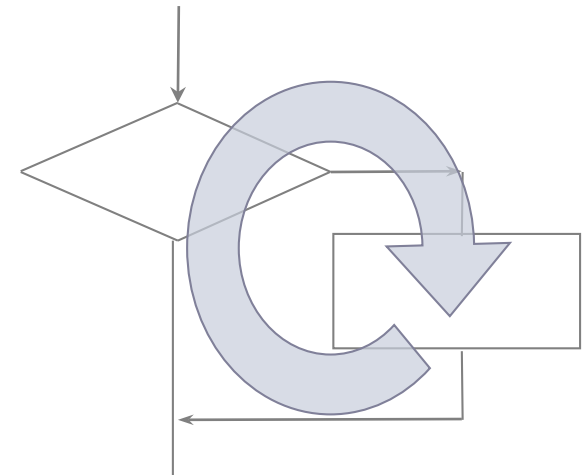
Secuencial



Condicional



Iterativa



# Composición condicional

```
<instrucciónCondicional> ::=
 "if" "(" <condición> ")"
 <instrucción>
 ["else" <instrucción>]
```

```
<condición> ::= <expresión>
```

# Composición condicional

---

## □ Semántica

- Se evalúa la condición.
- Si el valor resultante es *cierto*, se ejecuta únicamente la instrucción que sigue a la condición, una sola vez.
- Si el valor resultante es *falso* y hay una cláusula **else**, se ejecuta únicamente la instrucción de la cláusula **else**, una sola vez.



# Composición condicional

```
if (x >= 0) {
 cout << x << endl;
}
else {
 cout << -x << endl;
}
```

# Programa completo

```
#include <iostream>
using namespace std;

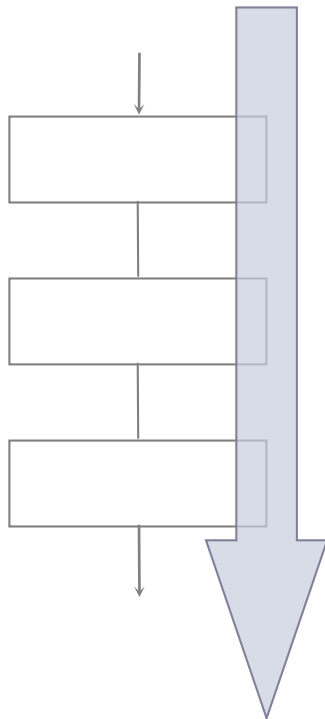
/*
 * Pre: ---
 * Post: Ha pedido al operador por un número y ha escrito en pantalla el
 * valor absoluto del número suministrado por el operador.
 */
int main() {
 cout << "Introduzca un número: " << endl;
 double x;
 cin >> x;

 cout << "Su valor absoluto es: ";
 if (x >= 0) {
 cout << x << endl;
 }
 else {
 cout << -x << endl;
 }

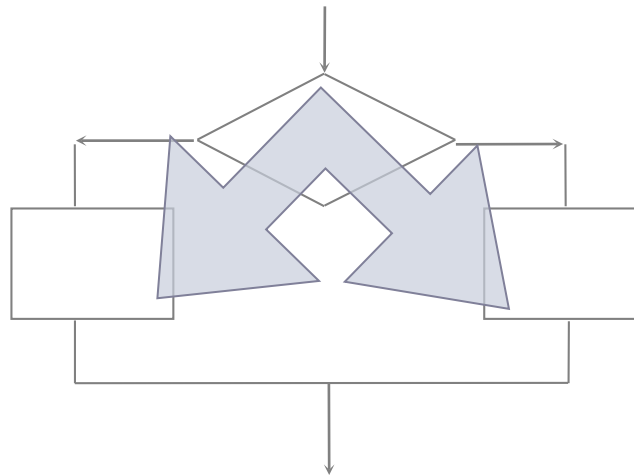
 return 0;
}
```

# Instrucciones estructuradas

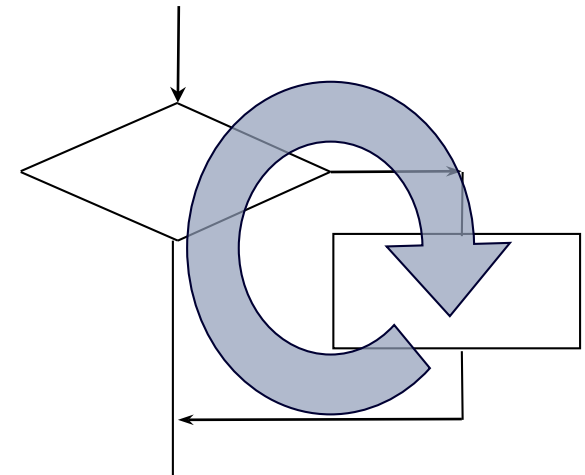
Secuencial



Condicional



Iterativa





# Composición iterativa

```
<instrucciónIterativa> ::=
 “while” “(” <condición> “)”
 <instrucción>
```

# Composición iterativa

---

## □ Semántica

- Se evalúa la condición  
→ resultado *cierto* o *falso*
- Mientras se evalúa como *cierto*, se ejecuta la instrucción que sigue a la condición y se vuelve a evaluar la condición
- Cuando se evalúa como *falso*, concluye la ejecución de la instrucción iterativa

# Ejemplo

```
void presentarTabla(int n) {
 // Escribe la cabecera de la tabla de multiplicar del «n»
 cout << endl;
 cout << "LA TABLA DEL " << n << endl;

 // Escribe las 11 líneas de la tabla de multiplicar del «n»
 int i = 0;
 while (i <= 10) {
 cout << setw(3) << n
 << " x " << setw(2) << i
 << " = " << setw(3) << n * i
 << endl;
 i++;
 }
}
```



# Un problema

---

- Programa que calcule el factorial de un número

Escriba un número natural: 5  
 $5! = 120$

# Factorial

```
#include <iostream>
using namespace std;

/*
 * Pre: n >= 0
 * Post: Ha devuelto n!
 */
int factorial(int n) {
 ...
}

/*
 * Pre: ---
 * Post: Ha pedido al operador un entero, lo ha leído de teclado y ha
 * escrito en pantalla su factorial
 */
int main() {
 ...
}
```



# Factorial

```
/*
 * Pre: ---
 * Post: Ha pedido al operador un entero, lo ha leído de
 * teclado y ha escrito en pantalla su factorial
 */
int main() {
 cout << "Escriba un número natural: " << flush;
 int n;
 cin >> n;

 cout << n << "! = " << factorial(n) << endl;

 return 0;
}
```

# Factorial

```
/*
 * Pre: $n \geq 0$
 * Post: Ha devuelto $n!$
 */
int factorial(int n) {
 // Asigna a «factorial» el valor de « n !», siendo $n \geq 0$
 int i = 1;
 int factorial = 1; // factorial = $i!$
 while (i < n) {
 i++;
 factorial = i * factorial; // factorial = $i!$
 }
 // $i = n$, factorial = $i! \Rightarrow$ factorial = $n!$
 return factorial;
}
```

# Factorial

```
/*
 * Asigna a «factorial» el valor de n!,
 * siendo $n \geq 0$
 */
int i = 1;
int factorial = 1; // factorial = i!
while (i < n) {
 i++;
 factorial = i * factorial; // factorial = i!
}
// $i = n$, factorial = i! \rightarrow factorial = n!
```

n

4

# Factorial

```
/*
 * Asigna a «factorial» el valor de n!,
 * siendo $n \geq 0$
 */
int i = 1;
int factorial = 1; // factorial = i!
while (i < n) {
 i++;
 factorial = i * factorial; // factorial = i!
}
// i = n, factorial = i! \rightarrow factorial = n!
```

n

4

i

1

# Factorial

```
/*
 * Asigna a «factorial» el valor de n!,
 * siendo $n \geq 0$
 */
int i = 1;
int factorial = 1; // factorial = i!
while (i < n) {
 i++;
 factorial = i * factorial; // factorial = i!
}
// $i = n$, factorial = i! \rightarrow factorial = n!
```

|      |   |
|------|---|
| n    | 4 |
| i    | 1 |
| fact | 1 |

# Factorial

```
/*
 * Asigna a «factorial» el valor de n!,
 * siendo $n \geq 0$
 */
int i = 1;
int factorial = 1; // factorial = i!
while (i < n) {
 i++;
 factorial = i * factorial; // factorial = i!
}
// i = n, factorial = i! \rightarrow factorial = n!
```

|      |   |
|------|---|
| n    | 4 |
| i    | 1 |
| fact | 1 |

# Factorial

```
/*
 * Asigna a «factorial» el valor de n!,
 * siendo $n \geq 0$
 */
int i = 1;
int factorial = 1; // factorial = i!
while (i < n) {
 i++;
 factorial = i * factorial; // factorial = i!
}
// i = n, factorial = i! \rightarrow factorial = n!
```

|      |   |
|------|---|
| n    | 4 |
| i    | 1 |
| fact | 1 |

# Factorial

```
/*
 * Asigna a «factorial» el valor de n!,
 * siendo $n \geq 0$
 */
int i = 1;
int factorial = 1; // factorial = i!
while (i < n) {
 i++;
 factorial = i * factorial; // factorial = i!
}
// $i = n$, factorial = i! \rightarrow factorial = n!
```

|      |   |
|------|---|
| n    | 4 |
| i    | 2 |
| fact | 1 |



# Factorial

```
/*
 * Asigna a «factorial» el valor de n!,
 * siendo $n \geq 0$
 */
int i = 1;
int factorial = 1; // factorial = i!
while (i < n) {
 i++;
 factorial = i * factorial; // factorial = i!
}
// $i = n$, factorial = i! \rightarrow factorial = n!
```

|      |   |
|------|---|
| n    | 4 |
| i    | 2 |
| fact | 1 |

# Factorial

```
/*
 * Asigna a «factorial» el valor de n!,
 * siendo $n \geq 0$
 */
int i = 1;
int factorial = 1; // factorial = i!
while (i < n) {
 i++;
 factorial = i * factorial; // factorial = i!
}
// $i = n$, factorial = i! \rightarrow factorial = n!
```

|      |   |
|------|---|
| n    | 4 |
| i    | 2 |
| fact | 2 |

# Factorial

```
/*
 * Asigna a «factorial» el valor de n!,
 * siendo $n \geq 0$
 */
int i = 1;
int factorial = 1; // factorial = i!
while (i < n) {
 i++;
 factorial = i * factorial; // factorial = i!
}
// i = n, factorial = i! \rightarrow factorial = n!
```

|      |   |
|------|---|
| n    | 4 |
| i    | 2 |
| fact | 2 |

# Factorial

```
/*
 * Asigna a «factorial» el valor de n!,
 * siendo $n \geq 0$
 */
int i = 1;
int factorial = 1; // factorial = i!
while (i < n) {
 i++;
 factorial = i * factorial; // factorial = i!
}
// i = n, factorial = i! \rightarrow factorial = n!
```

|      |   |
|------|---|
| n    | 4 |
| i    | 2 |
| fact | 2 |

# Factorial

```
/*
 * Asigna a «factorial» el valor de n!,
 * siendo $n \geq 0$
 */
int i = 1;
int factorial = 1; // factorial = i!
while (i < n) {
 i++;
 factorial = i * factorial; // factorial = i!
}
// i = n, factorial = i! \rightarrow factorial = n!
```

|      |   |
|------|---|
| n    | 4 |
| i    | 3 |
| fact | 2 |

# Factorial

```
/*
 * Asigna a «factorial» el valor de n!,
 * siendo $n \geq 0$
 */
int i = 1;
int factorial = 1; // factorial = i!
while (i < n) {
 i++;
 factorial = i * factorial; // factorial = i!
}
// $i = n$, factorial = i! \rightarrow factorial = n!
```

|      |   |
|------|---|
| n    | 4 |
| i    | 3 |
| fact | 2 |

# Factorial

```
/*
 * Asigna a «factorial» el valor de n!,
 * siendo $n \geq 0$
 */
int i = 1;
int factorial = 1; // factorial = i!
while (i < n) {
 i++;
 factorial = i * factorial; // factorial = i!
}
// i = n, factorial = i! \rightarrow factorial = n!
```

|      |   |
|------|---|
| n    | 4 |
| i    | 3 |
| fact | 6 |

# Factorial

```
/*
 * Asigna a «factorial» el valor de n!,
 * siendo $n \geq 0$
 */
int i = 1;
int factorial = 1; // factorial = i!
while (i < n) {
 i++;
 factorial = i * factorial; // factorial = i!
}
// i = n, factorial = i! \rightarrow factorial = n!
```

|      |   |
|------|---|
| n    | 4 |
| i    | 3 |
| fact | 6 |



# Factorial

```
/*
 * Asigna a «factorial» el valor de n!,
 * siendo $n \geq 0$
 */
int i = 1;
int factorial = 1; // factorial = i!
while (i < n) {
 i++;
 factorial = i * factorial; // factorial = i!
}
// i = n, factorial = i! \rightarrow factorial = n!
```

|      |   |
|------|---|
| n    | 4 |
| i    | 3 |
| fact | 6 |

# Factorial

```
/*
 * Asigna a «factorial» el valor de n!,
 * siendo n>=0
 */
int i = 1;
int factorial = 1; // factorial = i!
while (i < n) {
 i++;
 factorial = i * factorial; // factorial = i!
}
// i = n, factorial = i! → factorial = n!
```

|      |   |
|------|---|
| n    | 4 |
| i    | 4 |
| fact | 6 |

# Factorial

```
/*
 * Asigna a «factorial» el valor de n!,
 * siendo $n \geq 0$
 */
int i = 1;
int factorial = 1; // factorial = i!
while (i < n) {
 i++;
 factorial = i * factorial; // factorial = i!
}
// $i = n$, factorial = i! \rightarrow factorial = n!
```

|      |   |
|------|---|
| n    | 4 |
| i    | 4 |
| fact | 6 |

# Factorial

```
/*
 * Asigna a «factorial» el valor de n!,
 * siendo n>=0
 */
int i = 1;
int factorial = 1; // factorial = i!
while (i < n) {
 i++;
 factorial = i * factorial; // factorial = i!
}
// i = n, factorial = i! → factorial = n!
```

|      |    |
|------|----|
| n    | 4  |
| i    | 4  |
| fact | 24 |

# Factorial

```
/*
 * Asigna a «factorial» el valor de n!,
 * siendo $n \geq 0$
 */
int i = 1;
int factorial = 1; // factorial = i!
while (i < n) {
 i++;
 factorial = i * factorial; // factorial = i!
}
// i = n, factorial = i! \rightarrow factorial = n!
```

false

|      |    |
|------|----|
| n    | 4  |
| i    | 4  |
| fact | 24 |

# Factorial

```
/*
 * Asigna a «factorial» el valor de n!,
 * siendo n>=0
 */
int i = 1;
int factorial = 1; // factorial = i!
while (i<n) {
 i++;
 factorial = i * factorial; // factorial = i!
}
// i = n, factorial = i! → factorial = n!
```

|      |    |
|------|----|
| n    | 4  |
| i    | 4  |
| fact | 24 |

# Composición iterativa indexada

```
<instrucciónIterativaIndexada> ::=
 "for" "(" <inicialización> ";"
 <condición> ";"
 <actualización> ")"
 <instrucción>
```

```
<inicialización> ::= <instrucción>
```

```
<condición> ::= <expresión>
```

```
<actualización> ::= <instrucción>
```

# Composición iterativa indexada

---

- Semántica
  - Se ejecuta la instrucción de inicialización
  - Se evalúa la condición → resultado *cierto* o *falso*
  - Mientras el resultado es *cierto*:
    - Se ejecuta la instrucción del cuerpo del bucle
    - Se ejecuta la instrucción de actualización
    - Se vuelve a evaluar la condición
  - Cuando el resultado es *falso*, concluye la ejecución de la instrucción iterativa indexada



# Ejemplo

```
void presentarTabla(int n) {
 // Escribe la cabecera de la tabla de multiplicar del «n»
 cout << endl;
 cout << "LA TABLA DEL " << n << endl;

 // Escribe las 11 líneas de la tabla de multiplicar del «n»
 for (int i = 0; i <= 10; i++) {
 cout << setw(3) << n
 << " x " << setw(2) << i
 << " = " << setw(3) << n * i
 << endl;
 }
}
```

## Equivalencias bucles *while* y *for*

```
for (<inicialización>; <condición>;
 <actualización>)
 <instrucción>
```

```
<inicialización>;
while (<condición>) {
 <instrucción>;
 <actualización>;
}
```

# Factorial

```
/*
 * Asigna a «factorial» el valor de $n!$, siendo $n \geq 0$
 */
int factorial = 1;
for (int i = 1; i <= n; i++) {
 factorial = i * factorial; // factorial = i!
}
// factorial = $n!$
```



# Índice

---

- ❑ Instrucciones simples
- ❑ Instrucciones estructuradas