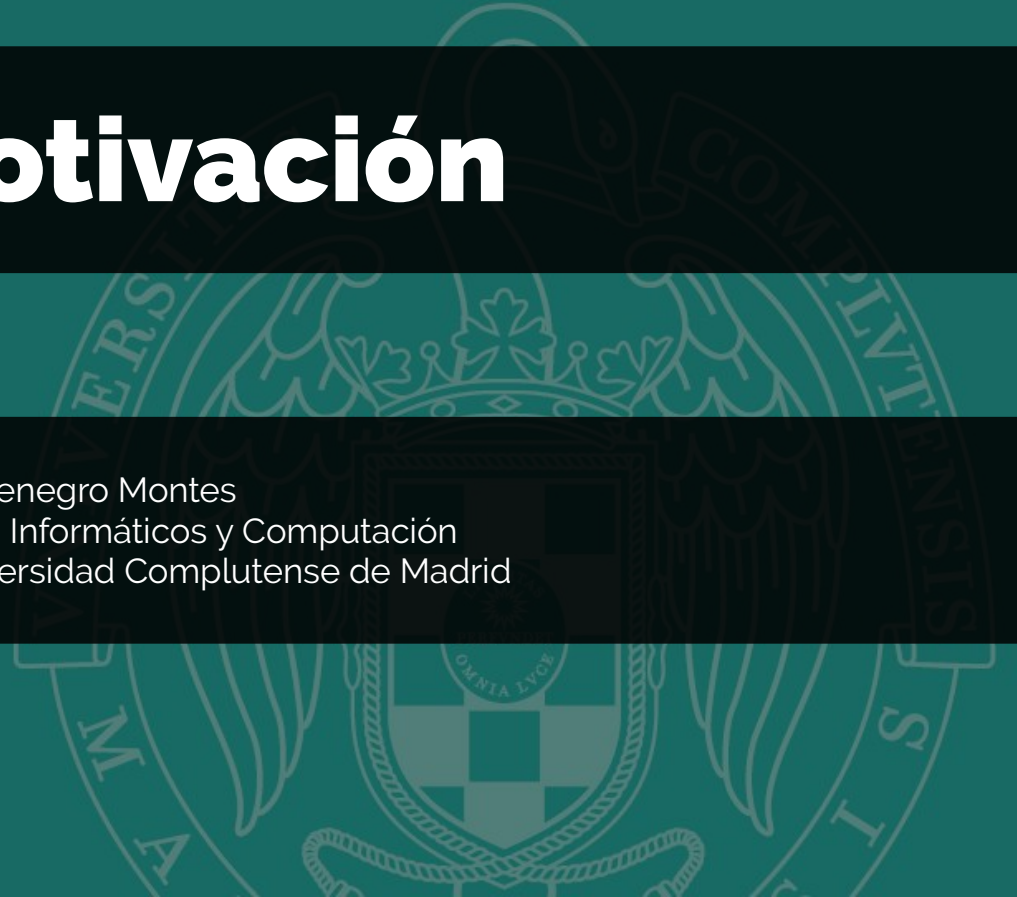


ESTRUCTURAS DE DATOS

INTRODUCCIÓN A LOS TIPOS ABSTRACTOS DE DATOS

TADs: motivación

Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid



Un pequeño juego



Jugador 1

N
S
O
T

E
T
V



Jugadora 2

Un pequeño juego



Jugador 1

N
S
O
T



Jugadora 2

E
T
V

- Para saber si una letra se ha dicho antes, debemos almacenar el conjunto de letras nombradas hasta el momento.

Tipo de datos ConjuntoChar

```
const int MAX_CHARS = 26;
```

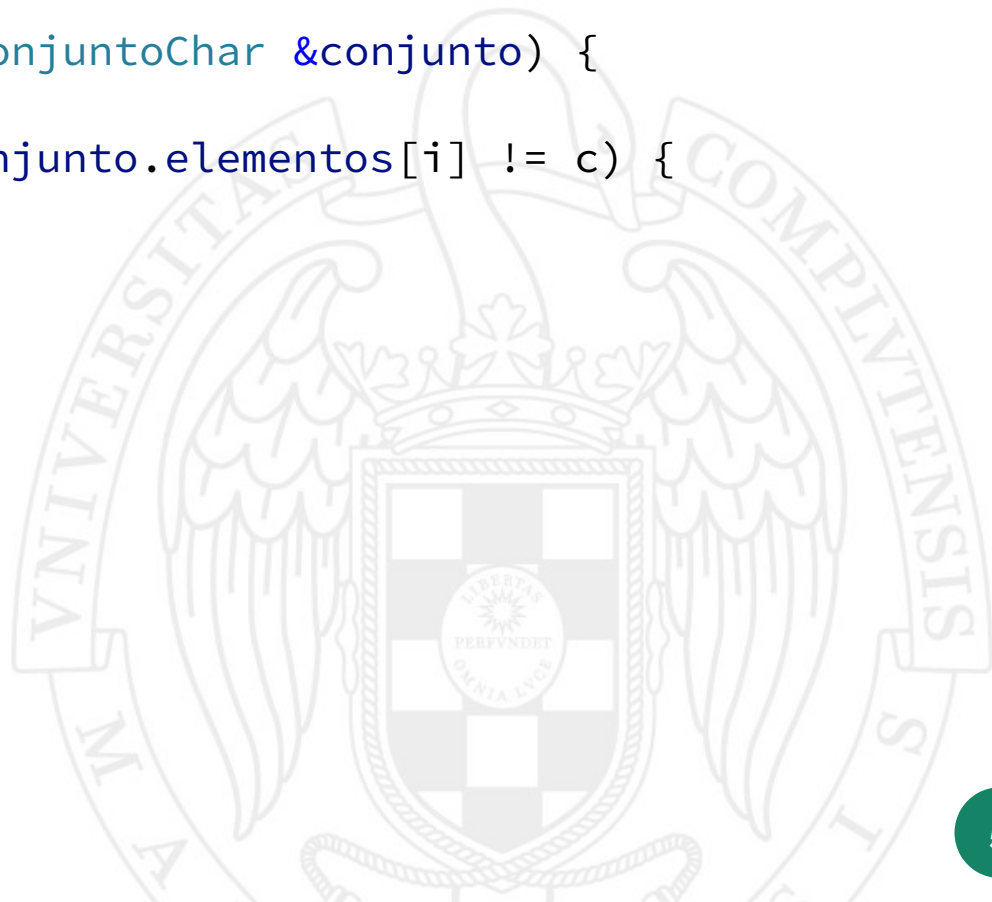
```
struct ConjuntoChar {  
    int num_chars;  
    char elementos[MAX_CHARS];  
};
```

- Suponemos que solo se admiten las letras mayúsculas del alfabeto inglés (A-Z).
 - Son un total de 26 letras.
- Guardamos las letras nombradas hasta el momento en el array **elementos**.
- Las primeras **num_chars** posiciones tienen letras. El resto se consideran posiciones “vacías”.

Función auxiliar: esta_en_conjunto

- Determina si el conjunto contiene la letra c pasada como parámetro.

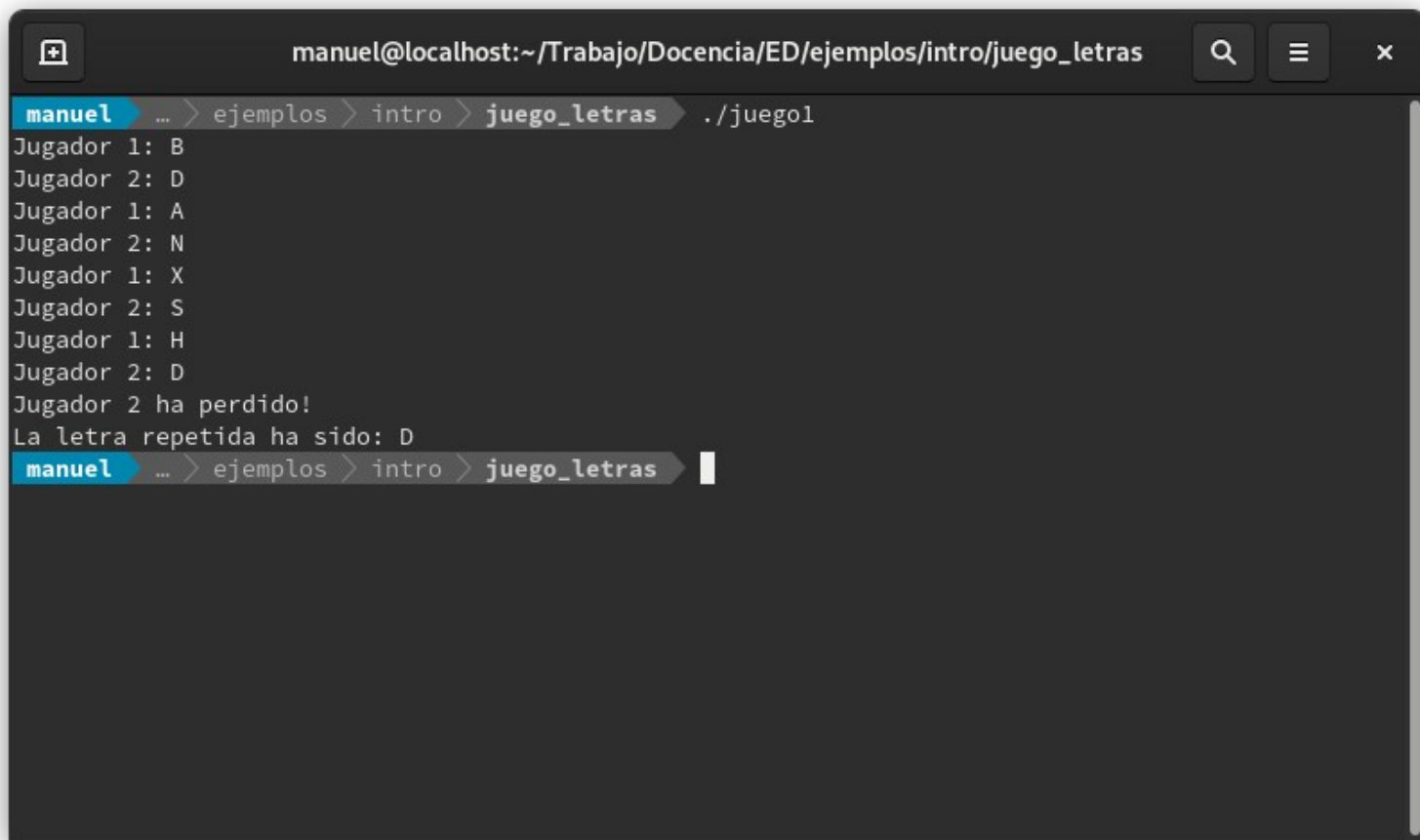
```
bool esta_en_conjunto(char c, const ConjuntoChar &conjunto) {  
    int i = 0;  
    while (i < conjunto.num_chars && conjunto.elementos[i] != c) {  
        i++;  
    }  
    return conjunto.elementos[i] == c;  
}
```



Implementación inicial del juego

```
int main() {  
    int jugador_actual = 1;  
    ConjuntoChar letras_nombradas;  
    letras_nombradas.num_chars = 0;  
  
    char letra_actual = preguntar_letra(jugador_actual);  
  
    while (!esta_en_conjunto(letra_actual, letras_nombradas)) {  
        letras_nombradas.elementos[letras_nombradas.num_chars] = letra_actual;  
        letras_nombradas.num_chars++;  
  
        jugador_actual = cambio_jugador(jugador_actual);  
        letra_actual = preguntar_letra(jugador_actual);  
    }  
  
    std::cout << "Jugador " << jugador_actual << " ha perdido!" << std::endl;  
    std::cout << "La letra repetida ha sido: " << letra_actual << std::endl;  
    return 0;  
}
```

Funcionamiento



A terminal window titled "manuel@localhost:~/Trabajo/Docencia/ED/ejemplos/intro/juego_letras" with search, menu, and close icons. The terminal shows the execution of a word game program. The path bar indicates the current directory is "juego_letras". The program output shows two players, Jugador 1 and Jugador 2, making moves. Jugador 1's moves are B, A, X, H. Jugador 2's moves are D, N, S, D. The game ends with the message "Jugador 2 ha perdido!" and "La letra repetida ha sido: D". The terminal prompt returns to "manuel" in the "juego_letras" directory.

```
manuel@localhost:~/Trabajo/Docencia/ED/ejemplos/intro/juego_letras
manuel ... > ejemplos > intro > juego_letras > ./juego1
Jugador 1: B
Jugador 2: D
Jugador 1: A
Jugador 2: N
Jugador 1: X
Jugador 2: S
Jugador 1: H
Jugador 2: D
Jugador 2 ha perdido!
La letra repetida ha sido: D
manuel ... > ejemplos > intro > juego_letras > |
```

Cambios en la implementación

```
const int MAX_CHARS = 26;
```

```
struct ConjuntoChar {  
    bool esta[MAX_CHARS];  
};
```

- Nuestro conjunto contiene un número limitado de letras.
- Podemos representar el contenido del conjunto como un array de booleanos.
 - Si la letra A está en el conjunto: `esta[0] = true`.
 - Si la letra B está en el conjunto: `esta[1] = true`.
 - ...

Cambios en esta_en_conjunto

```
bool esta_en_conjunto(char c, const ConjuntoChar &conjunto) {  
    return esta[c - (int)'A'];  
}
```



Implementación inicial del juego

```
int main() {  
    int jugador_actual = 1;  
    ConjuntoChar letras_nombradas;  
    letras_nombradas.num_chars = 0; ❌  
  
    char letra_actual = preguntar_letra(jugador_actual);  
  
    while (!esta_en_conjunto(letra_actual, letras_nombradas)) { ✅  
        letras_nombradas.elementos[letras_nombradas.num_chars] = letra_actual; ❌  
        letras_nombradas.num_chars++;  
  
        jugador_actual = cambio_jugador(jugador_actual);  
        letra_actual = preguntar_letra(jugador_actual);  
    }  
  
    std::cout << "Jugador " << jugador_actual << " ha perdido!" << std::endl;  
    std::cout << "La letra repetida ha sido: " << letra_actual << std::endl;  
    return 0;  
}
```



¿Qué ha fallado?

- Cualquier cambio en el tipo de datos `ConjuntoChar` tiene que ser propagado hasta aquellos sitios en los que se utilicen dichos campos.
- La función `main()` menciona explícitamente los campos del tipo `ConjuntoChar`. Por tanto, se ve afectada por el cambio de la definición del tipo.
- Un cambio en la definición de un tipo de datos debe provocar el menor impacto posible en la implementación del resto del programa.
- ¿Cómo delimitamos las operaciones que pueden verse afectadas por este cambio?

Abstracción mediante Tipos Abstractos de Datos (TADs)

ESTRUCTURAS DE DATOS

INTRODUCCIÓN A LOS TIPOS ABSTRACTOS DE DATOS

TADs: definición

Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid



¿Qué hemos hecho mal?

```
int main() {  
    int jugador_actual = 1;  
    ConjuntoChar letras_nombradas;  
    letras_nombradas.num_chars = 0;  
  
    char letra_actual = preguntar_letra(jugador_actual);  
  
    while (!esta_en_conjunto(letra_actual, letras_nombradas)) {  
        letras_nombradas.elementos[letras_nombradas.num_chars] = letra_actual;  
        letras_nombradas.num_chars++;  
  
        jugador_actual = cambio_jugador(jugador_actual);  
        letra_actual = preguntar_letra(jugador_actual);  
    }  
  
    std::cout << "Jugador " << jugador_actual << " ha perdido!" << std::endl;  
    std::cout << "La letra repetida ha sido: " << letra_actual << std::endl;  
    return 0;  
}
```

La lógica del juego utiliza detalles relativos a la implementación de los conjuntos de caracteres

¿Qué hemos hecho mal?

```
int main() {  
    int jugador_actual = 1;  
    ConjuntoChar letras_nombradas;  
    letras_nombradas.num_chars = 0;  
  
    char letra_actual = preguntar_letra(jugador_actual);  
  
    while (!esta_en_conjunto(letra_actual, letras_nombradas)) {  
        letras_nombradas.elementos[letras_nombradas.num_chars] = letra_actual;  
        letras_nombradas.num_chars++;  
  
        jugador_actual = cambio_jugador(jugador_actual);  
        letra_actual = preguntar_letra(jugador_actual);  
    }  
  
    std::cout << "Jugador " << jugador_actual << " ha perdido!" << std::endl;  
    std::cout << "La letra repetida ha sido: " << letra_actual << std::endl;  
    return 0;  
}
```

Sin embargo, aquí sí lo
hemos hecho bien...

Abstrayendo los detalles



Jugador 1

N
S
O
T

E
T
V



Jugadora 2

- El sitio en el que se guardan las letras nombradas hasta el momento se corresponde con la definición matemática de **conjunto**.

$$\text{LetrasNombradas} = \{ 'N', 'E', 'S', 'O', 'T', 'V' \}$$

En un lenguaje ideal...

```
int main() {  
    int jugador_actual = 1;  
    LetrasNombradas = ∅;  
  
    char letra_actual = preguntar_letra(jugador_actual);  
  
    while (letra_actual ∉ LetrasNombradas) {  
        LetrasNombradas = LetrasNombradas ∪ {letra_actual}  
  
        jugador_actual = cambio_jugador(jugador_actual);  
        letra_actual = preguntar_letra(jugador_actual);  
    }  
  
    std::cout << "Jugador " << jugador_actual << " ha perdido!" << std::endl;  
    std::cout << "La letra repetida ha sido: " << letra_actual << std::endl;  
    return 0;  
}
```


¿Qué necesitamos de un conjunto?

- Obtener un conjunto vacío.

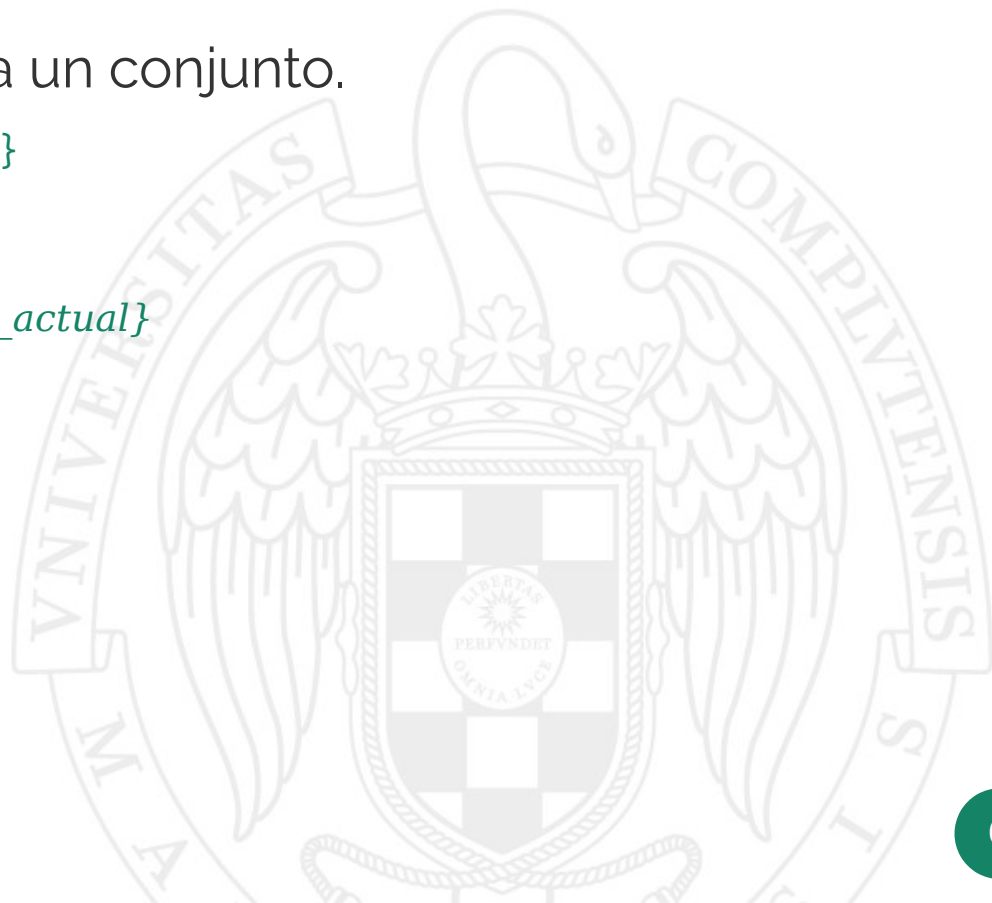
LetrasNombradas = \emptyset ;

- Saber si una letra pertenece (o no) a un conjunto.

while (*letra_actual* \notin *LetrasNombradas*) { ... }

- Añadir una letra a un conjunto.

LetrasNombradas = *LetrasNombradas* \cup {*letra_actual*}



Tipo Abstracto de Datos: definición

- Un **tipo abstracto de datos** (TAD) es un tipo de datos asociado con:
 - Un **modelo** conceptual.
 - Un conjunto de **operaciones**, especificadas mediante ese modelo.



En nuestro ejemplo

Tipo de datos: ConjuntoChar

- **Modelo:** conjuntos de letras, en el sentido matemático del término.

- **Operaciones:**

[true]

vacío() \rightarrow (C: ConjuntoChar)

[C = \emptyset]

[l \in {A,...,Z}]

pertenece(l: char, C: ConjuntoChar) \rightarrow (está: bool)

[está \Leftrightarrow l \in C]

[l \in {A,...,Z}]

añadir(l: char, C: ConjuntoChar)

[C = old(C) \cup {l}]

Implementación del TAD

- Nuestro modelo conceptual admite varias representaciones en C++. Hemos propuesto dos:

- **Representación 1:** array de caracteres.

```
struct ConjuntoChar {  
    int num_chars;  
    char elementos[MAX_CHARS];  
};
```

Cada representación implementa de manera distinta las operaciones mostradas anteriormente

- **Representación 2:** array de booleanos.

```
struct ConjuntoChar {  
    bool esta[MAX_CHARS];  
};
```

La representación determina la eficiencia de las operaciones implementadas

Representación 1

```
void vacio(ConjuntoChar &result) {  
    result.num_chars = 0;  
}
```

```
void anyadir(char letra, ConjuntoChar &conjunto) {  
    assert (conjunto.num_chars < MAX_CHARS);  
    assert (letra ≥ 'A' && letra ≤ 'Z');  
    conjunto.elementos[conjunto.num_chars] = letra;  
    conjunto.num_chars++;  
}
```

```
bool pertenece(char letra, const ConjuntoChar &conjunto) {  
    assert (letra ≥ 'A' && letra ≤ 'Z');  
    int i = 0;  
    while (i < conjunto.num_chars && conjunto.elementos[i] ≠ letra) {  
        i++;  
    }  
    return conjunto.elementos[i] = letra;  
}
```

Representación 2

```
void vacio(ConjuntoChar &result) {  
    for (int i = 0; i < MAX_CHARS; i++) {  
        result.esta[i] = false;  
    }  
}
```

```
void anyadir(char letra, ConjuntoChar &conjunto) {  
    assert (letra ≥ 'A' && letra ≤ 'Z');  
    conjunto.esta[letra - 'A'] = true;  
}
```

```
bool pertenece(char c, const ConjuntoChar &conjunto) {  
    assert (c ≥ 'A' && c ≤ 'Z');  
    return conjunto.esta[c - 'A'];  
}
```



Nuestro programa ideal...

```
int main() {  
    int jugador_actual = 1;  
    LetrasNombradas = ∅;  
  
    char letra_actual = preguntar_letra(jugador_actual);  
  
    while (letra_actual ∉ LetrasNombradas) {  
        LetrasNombradas = LetrasNombradas ∪ {letra_actual}  
  
        jugador_actual = cambio_jugador(jugador_actual);  
        letra_actual = preguntar_letra(jugador_actual);  
    }  
  
    std::cout << "Jugador " << jugador_actual << " ha perdido!" << std::endl;  
    std::cout << "La letra repetida ha sido: " << letra_actual << std::endl;  
    return 0;  
}
```

... y nuestro programa real

```
int main() {  
    int jugador_actual = 1;  
    ConjuntoChar letras_nombradas;  
    vacio(letras_nombradas);  
  
    char letra_actual = preguntar_letra(jugador_actual);  
  
    while (!pertenece(letra_actual, letras_nombradas)) {  
        anyadir(letra_actual, letras_nombradas);  
  
        jugador_actual = cambio_jugador(jugador_actual);  
        letra_actual = preguntar_letra(jugador_actual);  
    }  
  
    std::cout << "Jugador " << jugador_actual << " ha perdido!" << std::endl;  
    std::cout << "La letra repetida ha sido: " << letra_actual << std::endl;  
    return 0;  
}
```


¿Qué hemos ganado?

1) Simplificar el desarrollo

No hemos de preocuparnos de cómo está implementado ConjuntoChar.

2) Reutilización

ConjuntoChar puede utilizarse en otros contextos.

3) Separación de responsabilidades

Podemos reemplazar una implementación de ConjuntoChar por otra sin alterar el resto del programa.

Pero hay personas despistadas

```
int main() {  
    ...  
  
    std::cout << "Jugador " << jugador_actual << " ha perdido!" << std::endl;  
    std::cout << "La letra repetida ha sido: " << letra_actual << std::endl;  
    std::cout << "Se han nombrado " << letras_nombradas.num_chars  
               << " letras." << std::endl;  
    return 0;  
}
```



¿Existe algún mecanismo en el compilador de C++ que impida a las personas despistadas acceder a la representación interna de un TAD?

Encapsulación mediante clases

ESTRUCTURAS DE DATOS

INTRODUCCIÓN A LOS TIPOS ABSTRACTOS DE DATOS

TADs: definición

Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid



ESTRUCTURAS DE DATOS

INTRODUCCIÓN A LOS TIPOS ABSTRACTOS DE DATOS

Encapsulación en TADs

Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid



TAD ConjuntoChar

- Representa conjuntos de caracteres en mayúsculas en el alfabeto inglés (A..Z).

- Operaciones:
 - [true]*
 - vacío()** \rightarrow (C: ConjuntoChar)
 - [C = \emptyset]*

[l \in {A,...,Z}]

pertenece(l: char, C: ConjuntoChar) \rightarrow (está: bool)

[está \Leftrightarrow l \in C]

[l \in {A,...,Z}]

añadir(l: char, C: ConjuntoChar)

[C = old(C) \cup {l}]

Implementación de TADs mediante clases



TADs mediante clases

- Podemos definir tipos abstracto de datos utilizando las clases de C++.

```
class ConjuntoChar {  
public:  
  
    // operaciones públicas  
  
private:  
    // representación interna  
};
```



TADs mediante clases

- Podemos definir tipos abstracto de datos utilizando las clases de C++.

```
class ConjuntoChar {  
public:  
    ConjuntoChar();  
    bool pertenece(char l) const;  
    void anyadir(char l);  
  
private:  
    bool esta[MAX_CHARS];  
};
```

Diagram illustrating the mapping of C++ code to abstract operations:

- `ConjuntoChar();` maps to **vacio()** → (C: ConjuntoChar)
- `bool pertenece(char l) const;` maps to **pertenece**(l: char, C: ConjuntoChar) → bool
- `void anyadir(char l);` maps to **añadir**(l: char, C: ConjuntoChar)

Implementación de las operaciones

```
ConjuntoChar::ConjuntoChar() {  
    for (int i = 0; i < MAX_CHARS; i++) {  
        esta[i] = false;  
    }  
}
```

```
bool ConjuntoChar::pertenece(char l) const {  
    assert (l ≥ 'A' && l ≤ 'Z');  
    return esta[l - (int)'A'];  
}
```

```
void ConjuntoChar::anyadir(char l) {  
    assert (l ≥ 'A' && l ≤ 'Z');  
    esta[l - (int)'A'] = true;  
}
```

¿Cómo comprobar las precondiciones?

[true]

vacío() \rightarrow (C: ConjuntoChar)

[C = \emptyset]

[l \in {A,...,Z}]

pertenece(l: char, C: ConjuntoChar) \rightarrow (está: bool)

[está \Leftrightarrow l \in C]

[l \in {A,...,Z}]

añadir(l: char, C: ConjuntoChar)

[C = old(C) \cup {l}]

- Más sencillo, pero menos flexible: la macro **assert**.
 - Se encuentra en el fichero de cabecera **<cassert>**.
 - Comprueba la condición pasada como parámetro. Si es falsa, el programa aborta.
- Más potente y flexible: manejo de excepciones en C++.

Uso de TAD: lenguaje ideal

```
int main() {  
    int jugador_actual = 1;  
    LetrasNombradas = ∅;  
  
    char letra_actual = preguntar_letra(jugador_actual);  
  
    while (letra_actual ∉ LetrasNombradas) {  
        LetrasNombradas = LetrasNombradas ∪ {letra_actual}  
  
        jugador_actual = cambio_jugador(jugador_actual);  
        letra_actual = preguntar_letra(jugador_actual);  
    }  
  
    std::cout << "Jugador " << jugador_actual << " ha perdido!" << std::endl;  
    std::cout << "La letra repetida ha sido: " << letra_actual << std::endl;  
    return 0;  
}
```

Uso de TAD: sin clases

```
int main() {  
    int jugador_actual = 1;  
    ConjuntoChar letras_nombradas = vacio();  
    vacio(letras_nombradas);  
  
    char letra_actual = preguntar_letra(jugador_actual);  
  
    while (!pertenece(letra_actual, letras_nombradas)) {  
        anyadir(letra_actual, letras_nombradas);  
  
        jugador_actual = cambio_jugador(jugador_actual);  
        letra_actual = preguntar_letra(jugador_actual);  
    }  
  
    std::cout << "Jugador " << jugador_actual << " ha perdido!" << std::endl;  
    std::cout << "La letra repetida ha sido: " << letra_actual << std::endl;  
    return 0;  
}
```

Uso de TAD: con clases

```
int main() {  
    int jugador_actual = 1;  
    ConjuntoChar letras_nombradas;
```

```
    char letra_actual = preguntar_letra(jugador_actual);
```

```
    while (!letras_nombradas.pertenece(letra_actual)) {  
        letras_nombradas.anyadir(letra_actual);
```

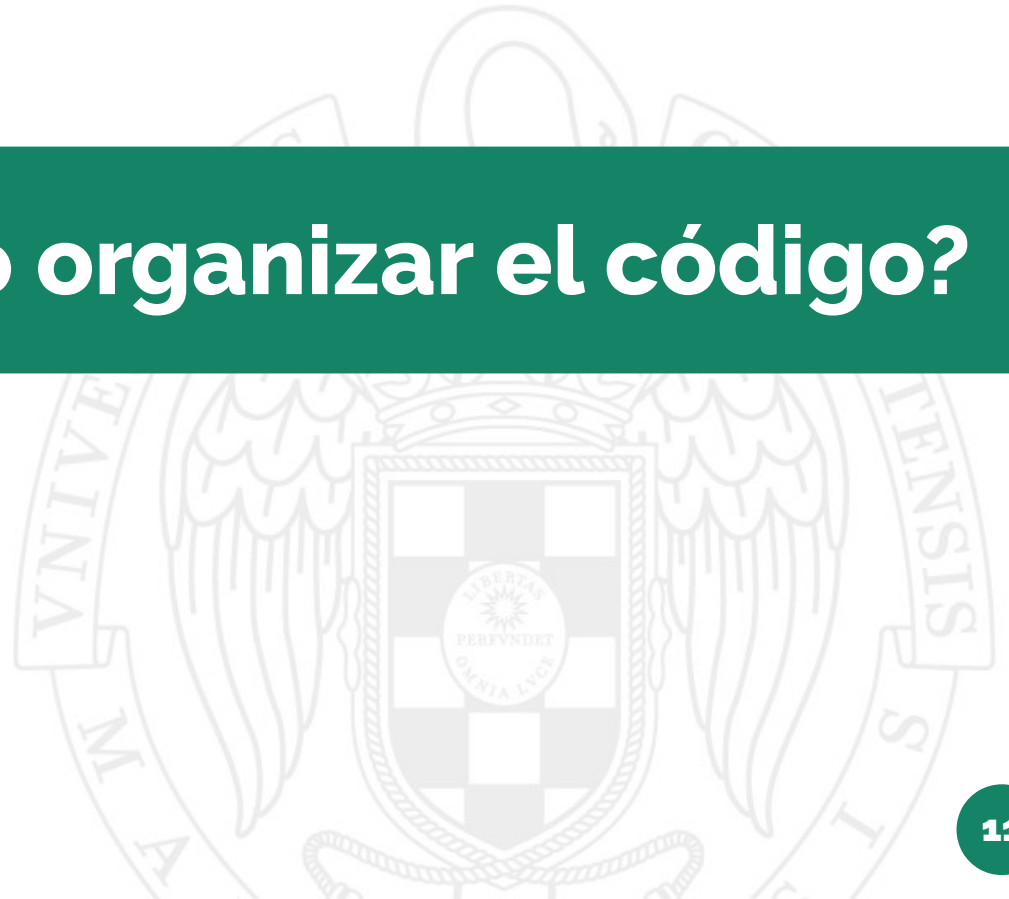
```
        jugador_actual = cambio_jugador(jugador_actual);  
        letra_actual = preguntar_letra(jugador_actual);  
    }
```

```
    std::cout << "Jugador " << jugador_actual << " ha perdido!" << std::endl;  
    std::cout << "La letra repetida ha sido: " << letra_actual << std::endl;  
    return 0;  
}
```

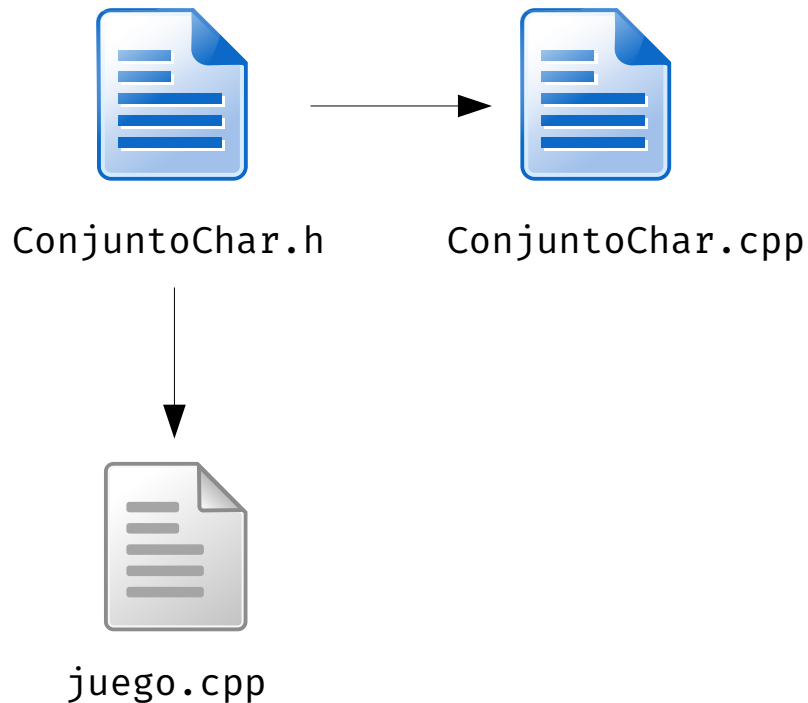


Encapsulación garantizada
por el compilador

Modularidad: ¿Cómo organizar el código?



Alt. 1: interfaz/implementación separadas



- Ventajas:
 - Separación de aspectos de implementación.
 - La implementación puede compilarse por separado.
- Desventajas:
 - No puede utilizarse en combinación con plantillas de C++ (template).

Alt. 2: interfaz e implementación juntas



ConjuntoChar.h



juego.cpp

- Inconvenientes:
 - Los detalles de implementación quedan expuestos.
 - Si cambiamos `ConjuntoChar`, hemos de recompilar `juego.cpp`
- Pero cuando utilicemos `templates` no tendremos más remedio que implementar las operaciones genéricas en el `.h`
... por lo menos hasta C++20

ESTRUCTURAS DE DATOS

INTRODUCCIÓN A LOS TIPOS ABSTRACTOS DE DATOS

Modelo vs. representación

Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid



Modelo vs. representación



TAD ConjuntoChar (representación 1)

Modelo

Conjuntos de letras mayúsculas

$\mathcal{P}(\{A..Z\})$

$\{A, D, Z\}$

$\{G, M\}$

\emptyset

Representación

```
class ConjuntoChar {  
    ...  
private:  
    int num_chars;  
    char elementos[MAX_CHARS];  
};
```

num_chars: 3
elementos: A D Z ...

num_chars: 2
elementos: G M ...

num_chars: 0
elementos: ...

TAD ConjuntoChar (representación 2)

Modelo

Conjuntos de letras mayúsculas

$\mathcal{P}(\{A..Z\})$

$\{A, D, Z\}$

\emptyset

Representación

```
class ConjuntoChar {  
    ...  
private:  
    bool esta[MAX_CHARS];  
};
```

esta:

T	F	F	T	F	...	F	...	F	T
---	---	---	---	---	-----	---	-----	---	---

esta:

F	F	F	F	F	...	F	...	F	F
---	---	---	---	---	-----	---	-----	---	---

TAD de números `int`

Modelo

Elementos de \mathbb{Z}

Representación

32 bits en complemento a 2

25

000...00011001

-7

111...11111001

TAD de números float

Modelo

Elementos de \mathbb{Q}

1.3423121

-0.5

Representación

IEEE 754

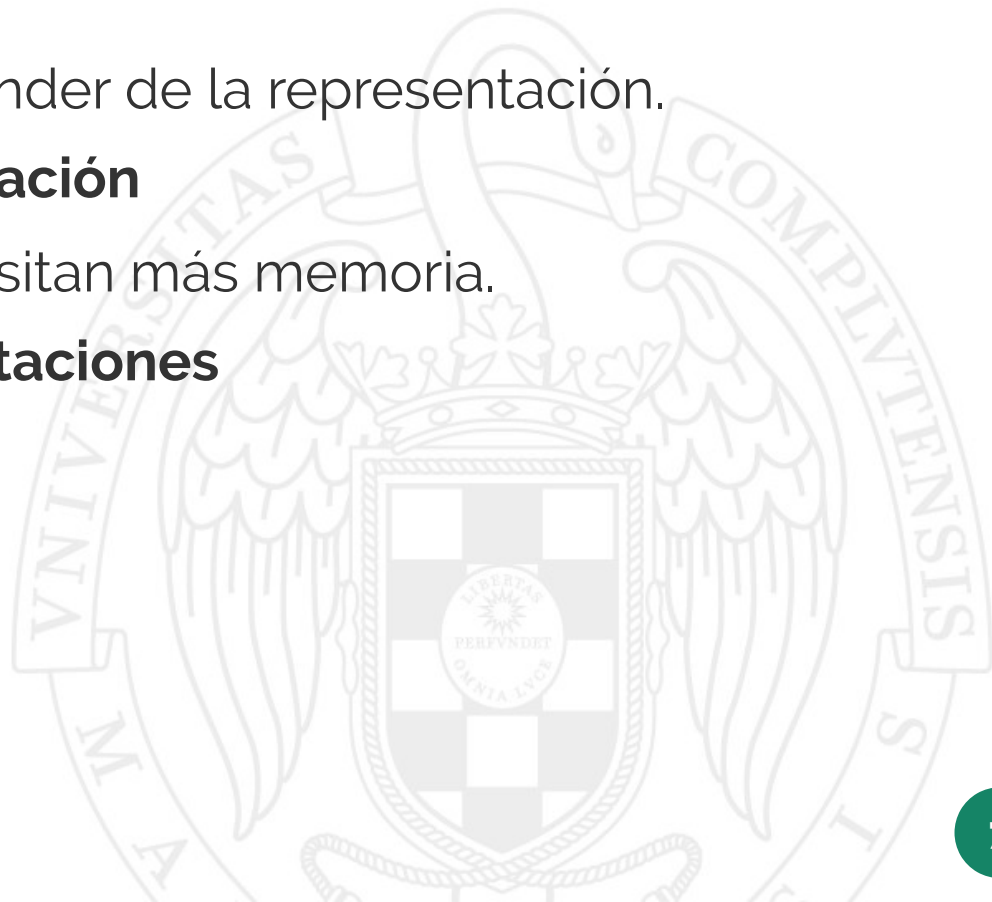
00111111101010111101000011100010

10111111100000000000000000000000

¡Cuidado! La representación es relevante

¿Por qué nos interesa conocer la representación?

- **Eficiencia de las operaciones**
 - El coste en tiempo puede depender de la representación.
- **Coste en memoria de la representación**
 - Algunas representaciones necesitan más memoria.
- **Limitaciones de algunas representaciones**



Limitaciones de algunas representaciones

- Enteros de 32 bits: -2147483648 a 2147483647.
- Coma flotante con float:

0.7

001111111001100110011001100110011

0.6999999881

```
float f = 7.0 / 10;  
std::cout << std::setprecision(10) << f << std::endl;
```


Limitaciones de algunas representaciones

- Enteros de 32 bits: -2147483648 a 2147483647.
- Coma flotante con `float`:

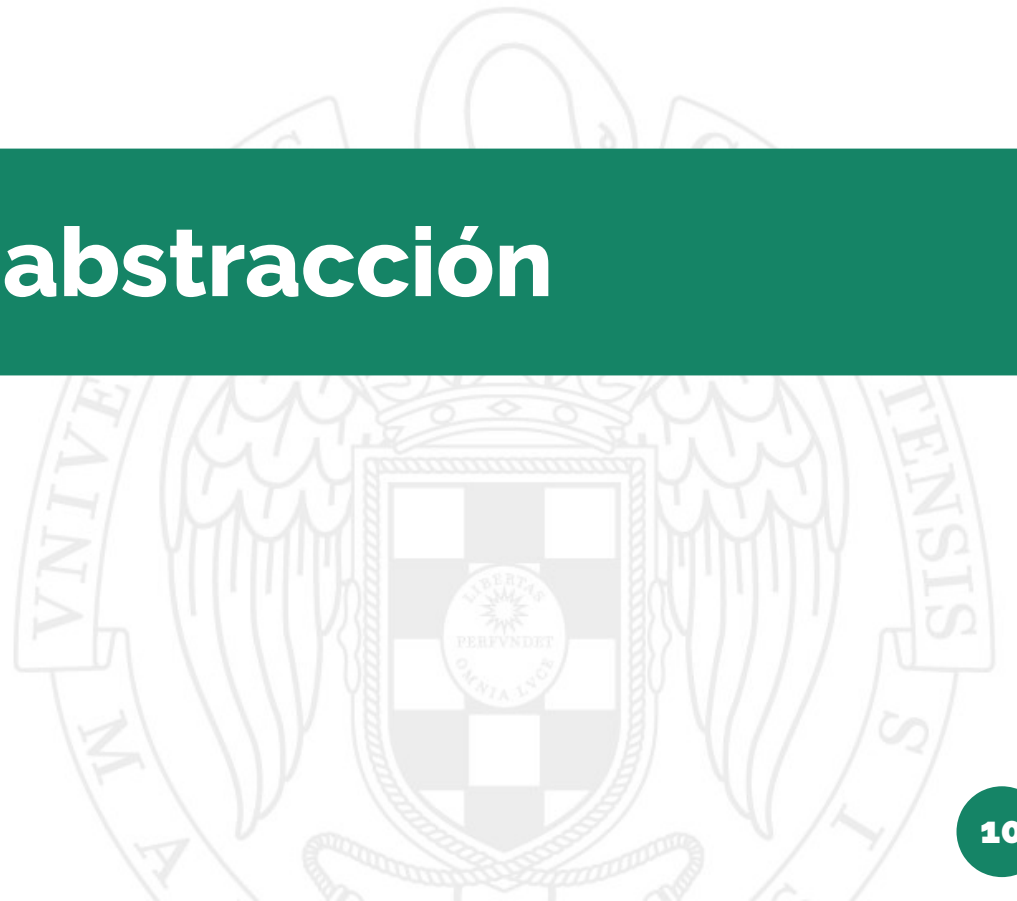
0.7

001111111001100110011001100110011

0.6999999881

- TAD `ConjuntoChar`: capacidad máxima

Función de abstracción



Función de abstracción

- Fijada la representación de un TAD, la función de abstracción asociada a una representación que asocia cada instancia de la representación con el modelo que representa.
- Ejemplo: TAD `int`

$$000\dots00011001 \xrightarrow{f_{int}} 25$$

$$f_{int}(x_{31}x_{30}\cdots x_0) = \begin{cases} \sum_{i=0}^{30} 2^i * x_i & \text{si } x_{31} = 0 \\ -(1 + \sum_{i=0}^{30} 2^i * \overline{x_i}) & \text{si } x_{31} = 1 \end{cases}$$

Función de abstracción

- Fijada la representación de un TAD, la función de abstracción asociada a una representación que asocia cada instancia de la representación con el modelo que representa.
- Ejemplo: TAD ConjuntoChar

```
class ConjuntoChar {  
    ...  
private:  
    int num_chars;  
    char elementos[MAX_CHARS];  
};
```

$$f_{CCI} : \text{ConjuntoChar} \rightarrow \mathcal{P}(\{A..Z\})$$

$$f_{CCI}(x) = \{ x.elementos[i] \mid 0 \leq i < x.num_chars \}$$

Función de abstracción

- Fijada la representación de un TAD, la función de abstracción asociada a una representación que asocia cada instancia de la representación con el modelo que representa.
- Ejemplo: TAD ConjuntoChar

```
class ConjuntoChar {  
    ...  
private:  
    bool esta[MAX_CHARS];  
};
```

$$f_{cc2} : \text{ConjuntoChar} \rightarrow \mathcal{P}(\{A..Z\})$$

$$f_{cc2}(x) = \{ c \in \{A..Z\} \mid x.esta[\text{ord}(c) - \text{ord}('A')] = \text{true} \}$$

Tipos de operaciones



TAD = Modelo + Operaciones

- Las operaciones en un TAD se especifican en función de los modelos.

[true]

vacio() \rightarrow (C: ConjuntoChar)

[C = \emptyset]

[l \in {A,...,Z}]

pertenece(l: char, C: ConjuntoChar) \rightarrow (está: bool)

[está \Leftrightarrow l \in C]

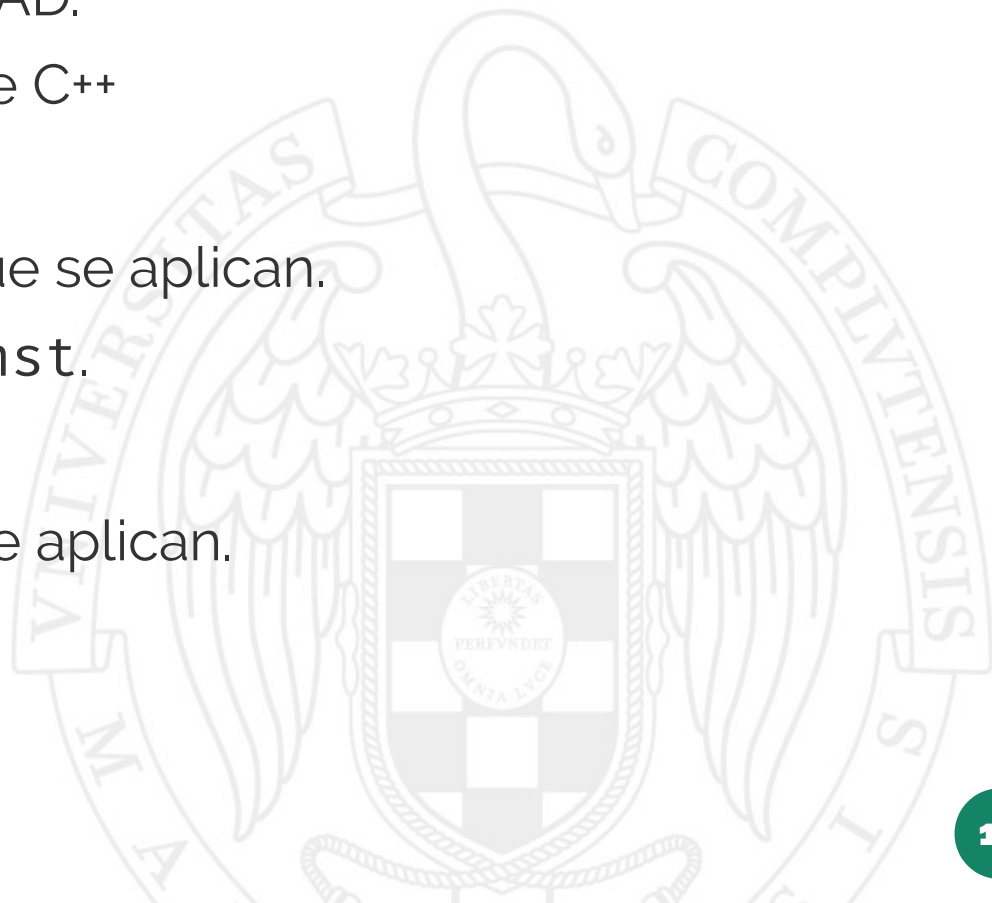
[l \in {A,...,Z}]

añadir(l: char, C: ConjuntoChar)

[C = old(C) \cup {l}]

Tipos de operaciones

- Funciones **constructoras**
 - Crean una nueva instancia del TAD.
 - Equivalen a los constructores de C++
- Funciones **observadoras**
 - No modifican el TAD sobre el que se aplican.
 - En C++ llevan el modificador `const`.
- Funciones **mutadoras**
 - Modifican el TAD sobre el que se aplican.



Ejemplo

[true]

vacio() \rightarrow (C: ConjuntoChar)

[C = \emptyset]

[l \in {A,...,Z}]

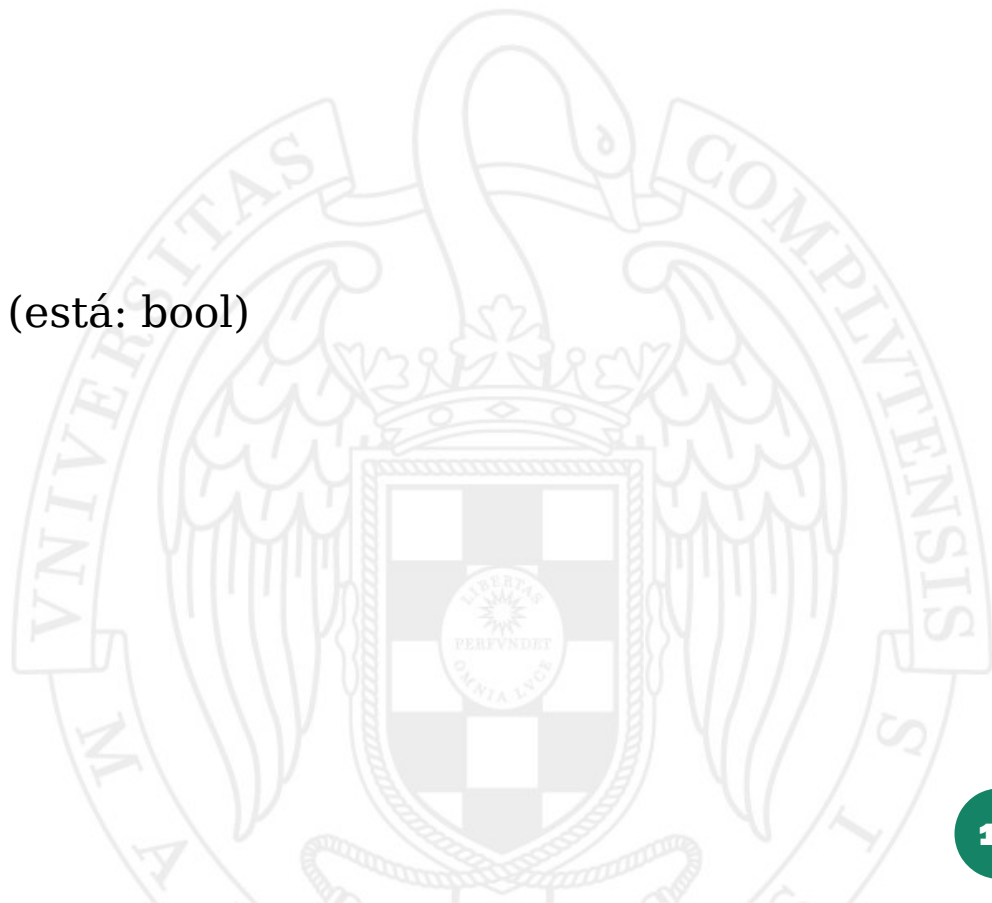
pertenece(l: char, C: ConjuntoChar) \rightarrow (está: bool)

[está \Leftrightarrow l \in C]

[l \in {A,...,Z}]

añadir(l: char, C: ConjuntoChar)

[C = old(C) \cup {l}]



Invariante de la representación



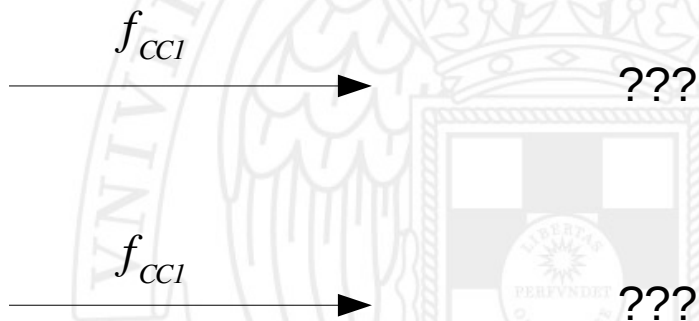
Instancias no válidas

- No todas las instancias de una representación denotan un modelo.

```
class ConjuntoChar {  
    ...  
private:  
    int num_chars;  
    char elementos[MAX_CHARS];  
};
```

num_chars: 2
elementos: 9 M ...

num_chars: -3
elementos: B M ...



Invariante de representación

- Un **invariante de representación** es una fórmula lógica que especifica cuándo una instancia es válida.

```
class ConjuntoChar {  
    ...  
private:  
    int num_chars;  
    char elementos[MAX_CHARS];  
};
```

$$I_{CCI}(x) = \\ 0 \leq x.num_chars \leq MAX_CHARS \wedge \\ \forall i: 0 \leq i < x.num_chars \Rightarrow x.elementos[i] \in \{A..Z\}$$

Invariante de representación

- Un **invariante de representación** (o invariante de clase) es una fórmula lógica que especifica cuándo una instancia es válida.

```
class ConjuntoChar {  
    ...  
private:  
    bool esta[MAX_CHARS];  
};
```

$$I_{cc2}(x) = \textit{true}$$

Invariantes y operaciones

- Las operaciones constructoras deben producir una instancia que cumpla el invariante.
- Las operaciones consultoras pueden asumir que la instancia cumple el invariante.
- Las operaciones mutadoras pueden asumir que la instancia cumple el invariante, y han de preservarlo al final de su ejecución.

[true]

vacio() \rightarrow (C: ConjuntoChar)

[C = \emptyset]

[l \in {A,...,Z}]

pertenece(l: char, C: ConjuntoChar) \rightarrow (está: bool)

[está \Leftrightarrow l \in C]

[l \in {A,...,Z}]

añadir(l: char, C: ConjuntoChar)

[C = old(C) \cup {l}]