

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

El objetivo es que no tengamos que cambiar mucho esta implementación.

este campo ya no está, solo hay un array de booleanos.

¿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;  
}
```

Esto sí que lo hemos hecho bien, no tenemos que modificar nada de esto ya tiene los mismos campos de antes.

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

Abstrayendo los detalles

Pensar en conjuntos no en arrays por ejemplo.



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**.

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

Sin embargo, estos conjuntos matemáticos en c++ no existen, no se pueden expresar de esa manera.

EN VEZ DE EXPRESARLO COMO UN ARRAY EXPRESARLO COMO UN CONJUNTO EN EL SENTIDO MATEMÁTICO

Un conjunto es un, valga la redundancia, un conjunto de datos NO REPETIDOS.

Otro error que hemos realizado es pensar en que los datos deben estar dentro de un array. Verlo como una colección de objetos a la que nosotros podemos ir añadiendo.

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;  
}
```

inicialmente vacío

pertence al de letras nombradas

Claro, esto es en lenguaje matemático pero en c++ esto no existe.

asignarla, pero en c++ no podemos utilizar este lenguaje matemático.

¿Qué necesitamos de un conjunto?

- Obtener un conjunto vacío. inicializar a 0's o a falses

`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}`

Necesitamos hacer estas 3 cosas.

Para ello utilizamos los tipos abstractos de los datos (TAD'S)

Tipo Abstracto de Datos: definición

- Un **tipo abstracto de datos** (TAD) es un tipo de datos asociado con:
 - Un **modelo** conceptual. No necesariamente una entidad matemática.
 - Un conjunto de **operaciones**, especificadas mediante ese modelo.

Las operaciones que hemos visto ahora.

Ahora lo vemos con un ejemplo

En nuestro ejemplo

Tipo de datos: ConjuntoChar

Queremos que represente CONJUNTOS DE LETRAS MAYÚSCULAS.

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

- **Operaciones:**

[true]

PRECONDICIÓN FAL

PARA CUALQUIER ENTRADA

vacio() → (C: ConjuntoChar)

[C = ∅]

POSTCONDICIÓN FAL

DEVUELVA UN CONJUNTO VACÍO.

[l ∈ {A,...,Z}]

UNA LETRA

pertenece(l: char, C: ConjuntoChar) → (está: bool)

[está ⇔ l ∈ C]

DEVUELVE BOOLEANO DE SI LA LETRA ESTA EN EL CONJUNTO C

[l ∈ {A,...,Z}]

CARACTER SEA UNA LETRA MAYÚSCULA DE LA 'A' A LA 'Z'

añadir(l: char, C: ConjuntoChar)

[C = old(C) ∪ {l}]

Y DEVUELVO EL CONJUNTO CON LA NUEVA LETRA.

En este caso 3 OPERACIONES:

Implementación del TAD

ARRAY DE LETRAS CON EL CONTADOR

ARRAY DE BOOLEANOS SIN EL CONTADOR.

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

- Representación 1:** array de caracteres.

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

Nos quedamos con la más eficiente.

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

Las dos hacen lo mismo, pero solo necesitamos una de ellas. Con cual nos quedamos, lo echamos a suertes? PUES NO

Representación 1

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

COSTE CONSTANTE

NUM.CHARS, EL 0 ES LA PRIMERA POSICIÓN

VECTOR ESTÁ INICIALMENTE VACÍO.

```
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++;  
}
```

COSTE CONSTANTE

ACTUALIZA LA PRIMERA POSICIÓN VACÍA DEL

ASSERT ES UNA MACRO QUE, SI LO QUE HAY DENTRO ES CIERTO ABORTA EL PROGRAMA

DESPLAZA EL CONTADOR

```
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;  
}
```

COSTE LINEAL RESPECTO AL NÚMERO DE LETRAS CONTENIDAS EN EL CONJUNTO EN EL CASO PEOR.

PARASABER SI LA LETRA ESTÁ EN EL CONJUNTO

Representación 2

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

COSTE LINEAL RESPECTO AL NÚMERO DE ELEMENTOS. MAXCHARS

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

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

mismas 3 operaciones y en composición ambas tanto rep1 como rep2 tienen el mismo coste.

Nuestro programa ideal...

SI SE PUDIERA UTILIZAR EL LENGUAJE MATEMÁTICO DEL QUE HABLAMOS AL PRINCIPIO.

```
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;  
}
```

No se nombra ninguno de los campos implicito del conjunto char.

Aquí no mencionamos ninguno de los campos conjunto char.

todo está mucho más claro.

¿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.

Main es independiente de la implementación de conjunto char.

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;  
}
```



El código de esta persona puede que al cambiar la representación interna del conjunto char, el código este dejaría de funcionar.

¿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

Este campo en la segunda representación no existe nos daría error. Si cambiamos el código interno de ConjuntoChar.

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

