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]

vacio() → (C: ConjuntoChar) Inicializar un conjunto al conjunto vacío.

[C = ∅]

[l ∈ {A,...,Z}]

pertenece(l: char, C: ConjuntoChar) → (está: bool) Ver si pertenece o no al conjunto

[está ⇔ l ∈ C]

[l ∈ {A,...,Z}]

añadir(l: char, C: ConjuntoChar) añadir elemento

[C = old(C) ∪ {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
};
```

Como si fuera la parte de una interfaz

para prohibir a las personas a que accedan a los detalles internos de mi clase.

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];
};
```

constructor

vacio() → (C: ConjuntoChar)

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

añadir(l: char, C: ConjuntoChar)

No modifica el estado del objeto y por eso está declarado como constante.

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;
}
```

Comprobar que se cumpla la precondición.

En caso contrario lanza una excepción.

¿Cómo comprobar las precondiciones?

[true]

vacio() → (C: ConjuntoChar)

[C = ∅]

[l ∈ {A,...,Z}]

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

[está ⇔ l ∈ C]

[l ∈ {A,...,Z}]

añadir(l: char, C: ConjuntoChar)

[C = old(C) ∪ {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++.

esto es lo de TP, no se si lo veremos es otra forma de tratar

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;
}
```

antes estaba así sin clases ni nada

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;
}
```

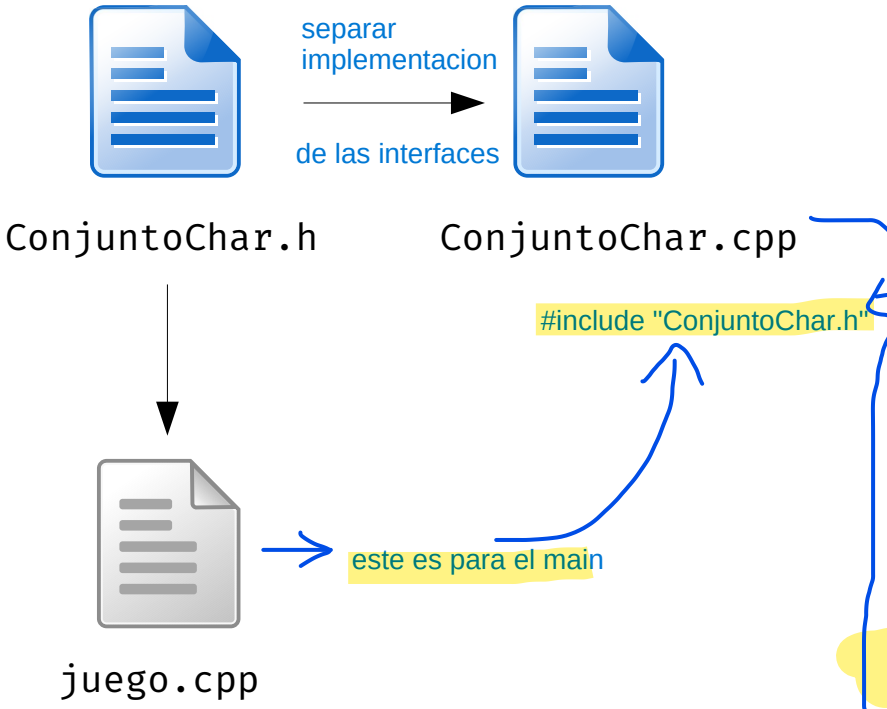
Encapsulación garantizada por el compilador

Garantizamos que el método main no va a acceder al array de booleanos de conjuntoChar. ENCAPSULACIÓN

Modularidad: ¿Cómo organizar el código?

Alt. 1: interfaz/implementación separadas

Lo ideal en teoría sería esto.



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

Ha hablado de genéricos.

Y en esta asignatura vamos a hacer un uso INTESIVO DE TEMPLATES. POR TANTO NO USAREMOS MUCHO ESTA ORGANIZACIÓN

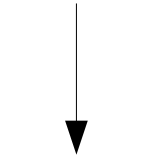
Se podría compilar por separado y finalmente crear el ejecutable, que ya lo hace el visual studio por separado, por defecto.

Alt. 2: interfaz e implementación juntas

NOS TENEMOS QUE CONFORMAR CON UTILIZAR ESTA OPCIÓN .

Definición de la clase e implementación de los métodos.

`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