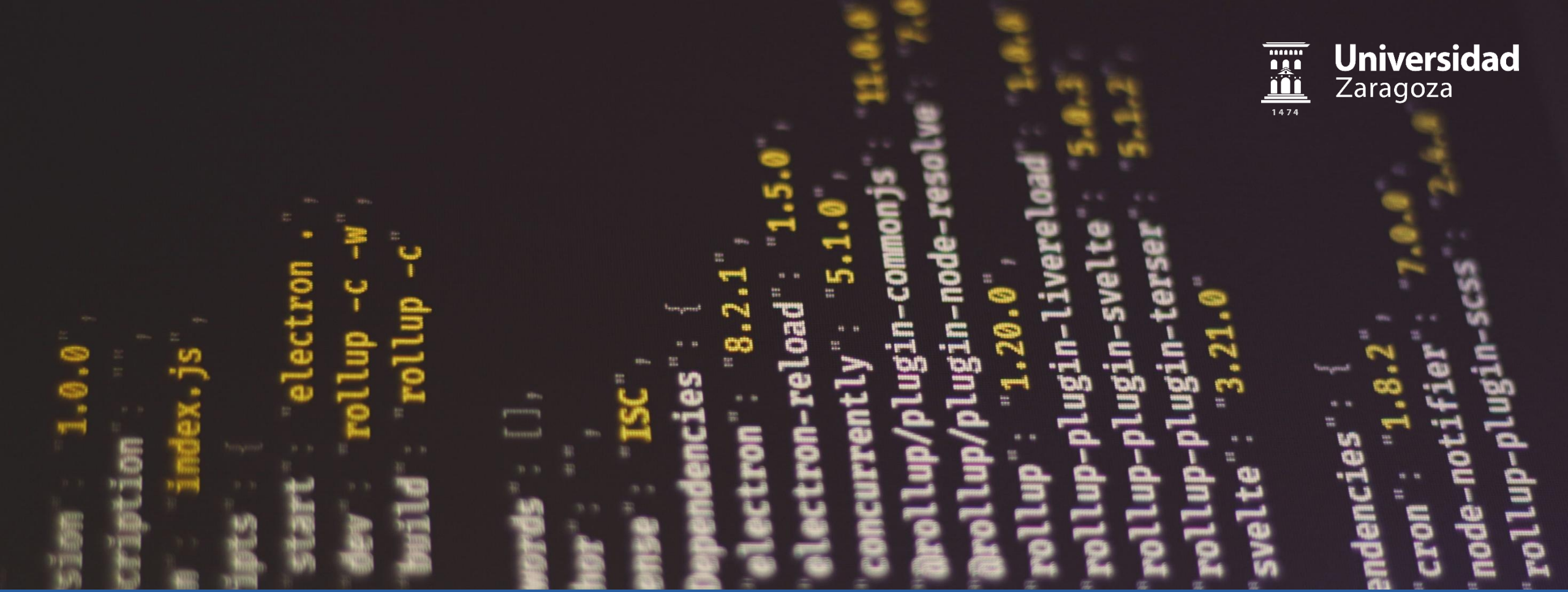


Universidad Zaragoza 1474

Práctica 2. Implementación de TAD genéricos en C++

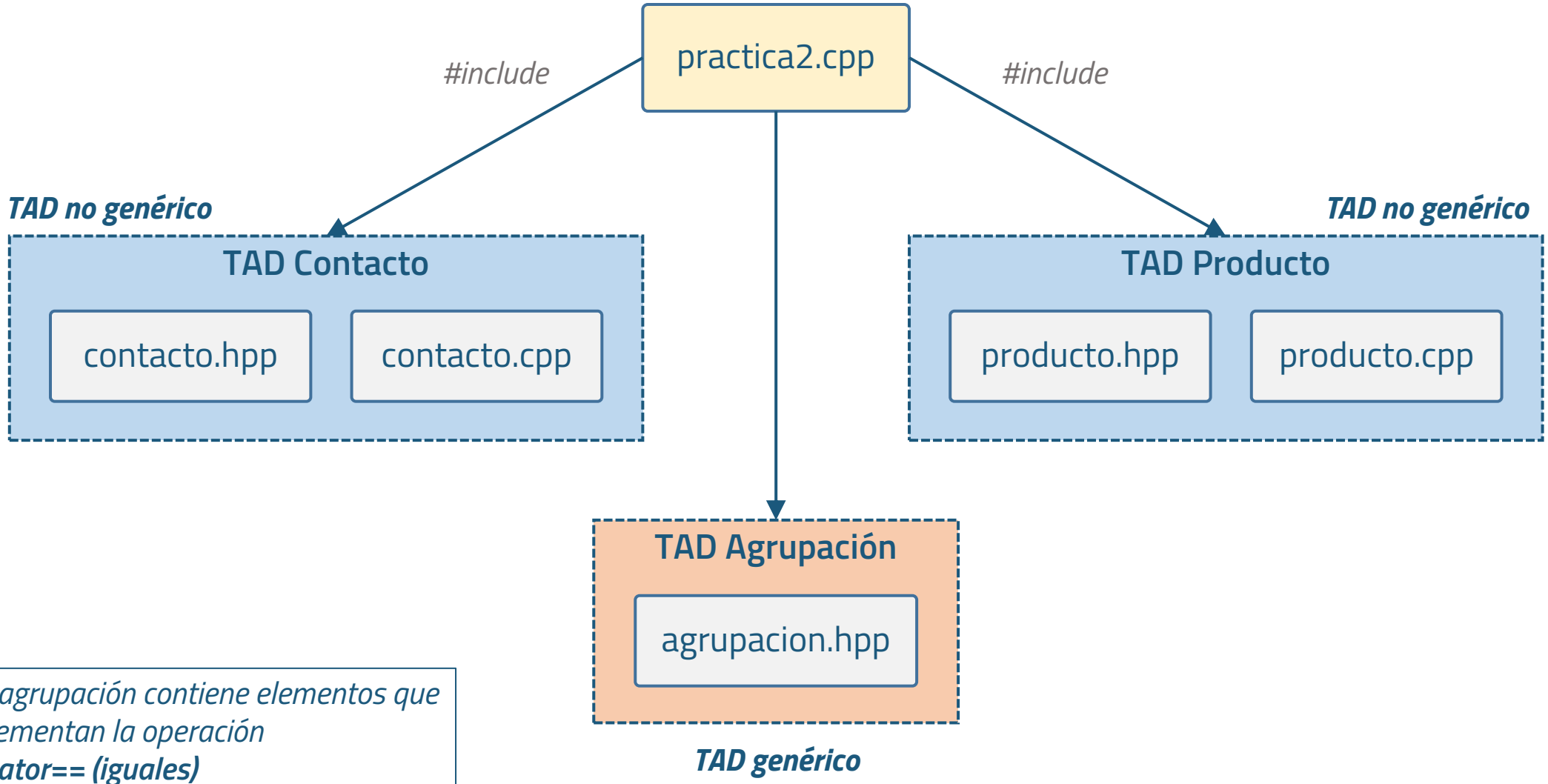
Estructuras de datos y algoritmos

Grado en Ingeniería Informática – Curso académico 2025-2026

OBJETIVOS DE LA PRÁCTICA

- Modificar práctica 1 para incluir **TAD genéricos.**
  - Implementar un programa con una **agrupación de contactos.**
- Incluir un **nuevo elemento, productos.**
  - Implementar un programa con una **agrupación de productos.**
- Combinar **iteradores y TAD genéricos.**

ESQUEMA DE PRÁCTICA



Una agrupación contiene elementos que implementan la operación ***operator== (iguales)***

IMPLEMENTACIÓN DEL TAD *AGRUPACIÓN*

practica2.cpp

```
#include "contacto.hpp"

int main()
{
}
```

contacto.hpp

```
#ifndef CONTACTO_HPP
#define CONTACTO_HPP

struct contacto {};

#endif
```

g++ -std=c++11

practica2.cpp

```
struct contacto {};

int main()
{
}
```

practica2.cpp

```
#include "contacto.hpp"
#include "agrupacion.hpp"

int main()
{
   agrupacion<contacto> a;
  ...
}
```

agrupacion.hpp

```
#ifndef AGRUPACION_HPP
#define AGRUPACION_HPP

template<typenameT>
struct agrupacion
{
  private: T datos[MAX];
};

#endif
```

g++ -std=c++11

practica2.cpp

```
struct agrupacion_contacto
{
  private: contacto datos[MAX];
};

int main()
{
  agrupacion_contacto a;
  ...
}
```

ESPECIFICACIÓN DEL TAD GENÉRICO *AGRUPACIÓN*


```
espec agrupaciones
 usa booleanos
 parámetro formal
   género elem
   operación
      iguales: elem e1, elem e2 -> booleano
      {Dados dos datos de tipo elem e1 y e2, devuelve verdad si y sólo si se consideran iguales}
 fpf
 género agrupación
 {Los valores del TAD representan colecciones ...}

 operaciones
   iniciar: -> agrupacion
   {Devuelve una agrupación vacía, sin elementos}

   añadir: agrupacion a, elemento e -> agrupacion
   {Devuelve una agrupación igual a la resultante de añadir un elemento e a la agrupación a.}

 ...
fespec
```

ESPECIFICACIÓN DEL TAD GENÉRICO *AGRUPACIÓN*


```
espec agrupaciones
 ...

 operaciones

   ...

   vacía: agrupacion a -> booleano
   {Devuelve verdad si y sólo si la agrupación a está vacía}

   borrarUltimo: agrupacion a -> agrupacion
   {Si a no está vacía, devuelve la agrupación igual a la resultante de eliminar de a el
    último elemento añadido a ella. Si a está vacía, devuelve la agrupación vacía}

   está: agrupacion a, elemento e -> booleano
   {Dada una agrupación a y un elemento e, devuelve verdad si y sólo si en
    a hay algún elemento igual a e (en el sentido de la función iguales).}

fespec
```

IMPLEMENTACIÓN DEL TAD *AGRUPACIÓN*


```
const int MAX = 40;

template<typename T> struct agrupacion;

template<typename T> void iniciar(agrupacion<T> &a);
template<typename T> bool anyadir(agrupacion<T> &a, const T& e);
template<typename T> bool vacia(agrupacion<T>& a);
template<typename T> void borrarUltimo(agrupacion<T>& a);
template<typename T> bool esta(const agrupacion<T>& a, const T& e);

template<typename T>
struct agrupacion {
    friend void iniciar<T>(agrupacion<T>& a);
    friend bool anyadir<T>(agrupacion<T>& a, const T& e);
    friend bool vacia<T>(agrupacion<T>& a);
    friend void borrarUltimo<T>(agrupacion<T>& a);
    friend bool esta<T>(const agrupacion<T>& a, const T& e);
private:
     T datos[MAX];
     int total;
};
```

Interfaz

IMPLEMENTACIÓN DEL TAD *AGRUPACIÓN*


```
#ifndef AGRUPACION_HPP
#define AGRUPACION_HPP

...
template<typename T> void iniciar(agrupacion<T> &a);
...

template<typename T>
struct agrupacion {
    friend void iniciar(agrupacion<T>& a);
    ...
private:
    ...
};

template<typename T>
void iniciar(agrupacion<T> &a) {
   a.total = 0;
}

...
#endif
```

IMPLEMENTACIÓN DEL TAD *AGRUPACIÓN*


```
...
template<typename T> bool esta(const agrupacion<T>& a, const T& e);

template<typename T>
struct agrupacion {
    friend bool esta(const agrupacion<T>& a, const T& e);
    ...
private:
     ...
};

template<typename T>
bool esta(const agrupación<T>& a, const T& e)
{
   bool pertenece = false;
   for (int i = 0; i < a.total && !pertenece; ++i)
      pertenece = a.datos[i] == e;

   return pertenece;
}
...
#endif
```

practica2.cpp

```
...
int main() {
  producto p1;
  ...
  agrupacion<producto> a;
  anyadir(a, p1);
  esta(a, p1);
  ...
}
```

practica2.cpp

```
...
int main() {
  producto p1;
  ...
  agrupacion<producto> a;
  anyadir(a, p1);
  ...
}
```

ESPECIFICACIÓN DEL TAD GENÉRICO *PRODUCTO*


```
espec productos
 usa cadenas, enteros, booleanos
 género producto
 {Los valores del TAD producto representan productos para los que se tiene información de su nombre y cantidad}

 operaciones
   crear: cadena nom, entero c -> producto
   {Dada una cadena nom y un entero c, se obtiene un producto de nombre nom y con cantidad de producto c}

   nombre: producto p -> cadena
   {Dado un producto p, se obtiene la cadena correspondiente al nombre del producto p}

   cantidad: producto p -> entero
   {Dado un producto p, se obtiene el entero correspondiente a la cantidad del producto p}

   iguales: producto p1, producto p2 -> booleano
   {Devuelve verdad si y sólo si los productos p1 y p2 tienen el mismo nombre}

fespec
```

IMPLEMENTACIÓN DEL TAD *PRODUCTO*


```
...

struct producto;
```

Interfaz

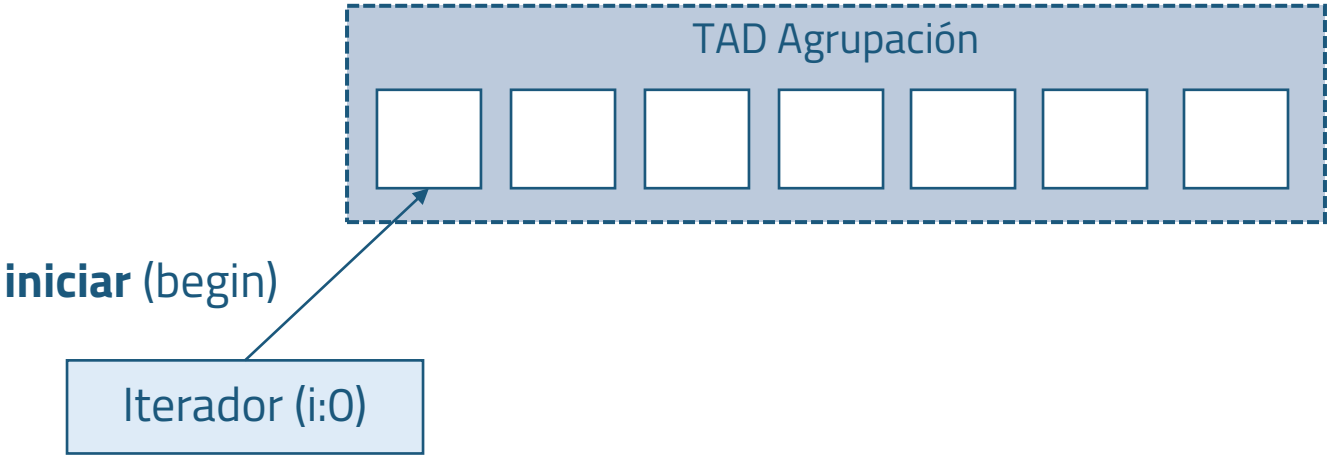
```
void crear(string nom, int can, producto& p);
string nombre(const producto& p);
int cantidad(const producto& p);
bool operator==(const producto& p1, const producto& p2);
```

```
struct producto {
    friend void crear(string nom, int can, producto& p);
    friend string nombre(const producto& p);
    friend int cantidad(const producto& p);
    friend bool operator==(const producto& p1, const producto& p2);

private:
    string nombre;
    int cantidad;
}

...
```

```
...
int main() {
   contacto c1;
   crear("pepe", "calle 1", 976555555, c1);

   agrupacion<contacto> agenda;
   iniciar(agenda);
   anyadir(agenda, c1);

   producto nar;
   crear("naranjas", 2, nar);

   agrupacion<producto> compra;
   iniciar(compra);
   anyadir(compra, nar);
}
```

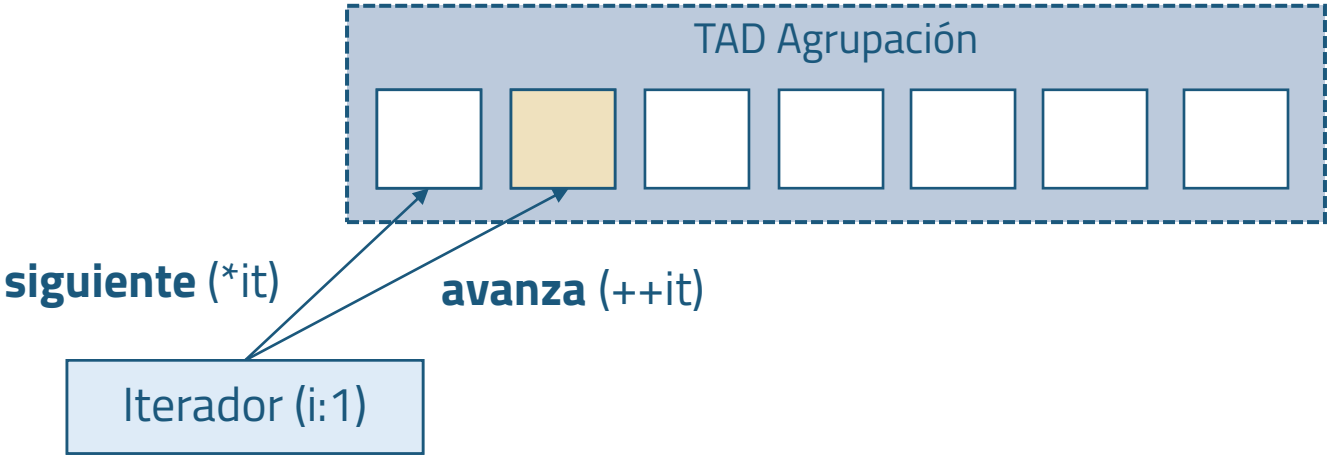
ITERADORES

- Un iterador nos permite **recorrer un conjunto de elementos**.
  - De manera **completa** (todos los elementos).
  - **Sin visitar un mismo elemento más de una vez**.
- Se utiliza desde fuera del TAD al que afecta el iterador.
- No expone detalles de implementación.
- Se utilizarán para recorrer y buscar objetos del TAD.
  - No haremos ninguna modificación.



ITERADORES

- Un iterador nos permite **recorrer un conjunto de elementos**.
  - De manera **completa** (todos los elementos).
  - **Sin visitar un mismo elemento más de una vez**.
- Se utiliza desde fuera del TAD al que afecta el iterador.
- No expone detalles de implementación.
- Se utilizarán para recorrer y buscar objetos del TAD.
  - No haremos ninguna modificación.



ITERADORES


```
espec agrupaciones
 usa booleanos
 parámetro formal
   género elem
   operación
      iguales: elem e1, elem e2 -> booleano
      {Dados dos datos de tipo elem e1 y e2, devuelve verdad si y sólo si se consideran iguales}
 fpf
 género agrupación

 ...

 iniciarIterador: agenda a -> agenda
 {Prepara el iterador para que el siguiente contacto a visitar sea el primero (situación de no haber
  visitado ningún contacto)}
 existeSiguiente?: agenda a -> booleano
 {Devuelve verdad si queda algún contacto por visitar, devuelve falso si ya se ha visitado el último
  contacto}
 ...
fespec
```

ITERADORES


```
espec agrupaciones
 usa booleanos
 parámetro formal
   género elem
   operación
      iguales: elem e1, elem e2 -> booleano
      {Dados dos datos de tipo elem e1 y e2, devuelve verdad si y sólo si se consideran iguales}
 fpf
 género agrupación

 ...

 parcial siguiente: agrupacion a -> elemento
 {Devuelve el siguiente elemento a visitar. Parcial: la operación no está definida si no quedan
  elementos por visitar (no existeSiguiente?)}

 parcial avanza: agrupacion a -> agrupacion
 {Prepara el iterador para que se pueda visitar el siguiente elemento.
  Parcial: la operación no está definida si no quedan elementos por visitar ( no existeSiguiente?(a)}
fespec
```

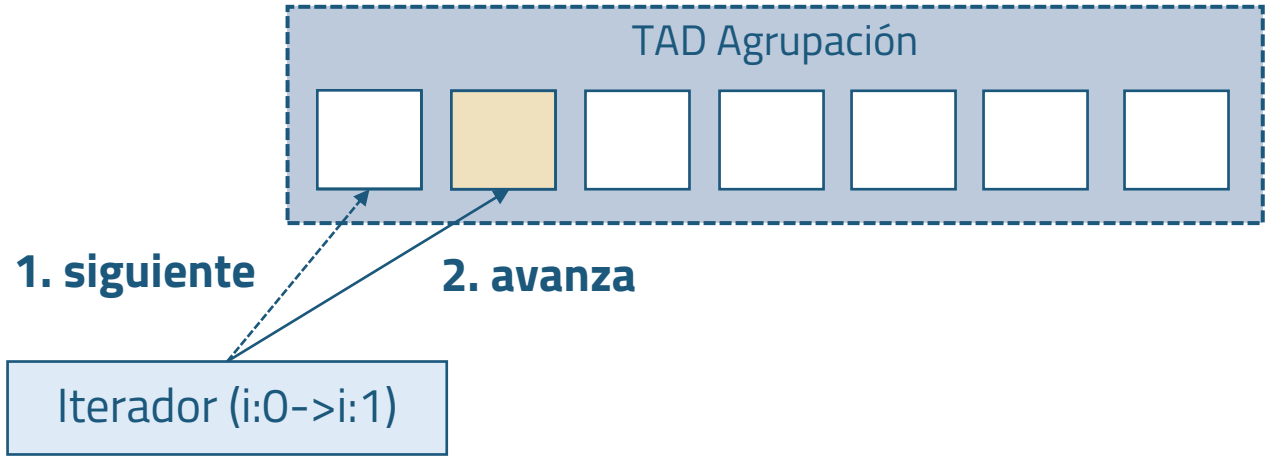
ITERADORES

- **Especificación:**

```
parcial siguiente: agrupacion a -> elemento
parcial avanza: agrupacion a -> agrupacion
```

- **Implementación C++:**

```
bool siguienteYavanza(agrupacion<T>& a, T& e);
```



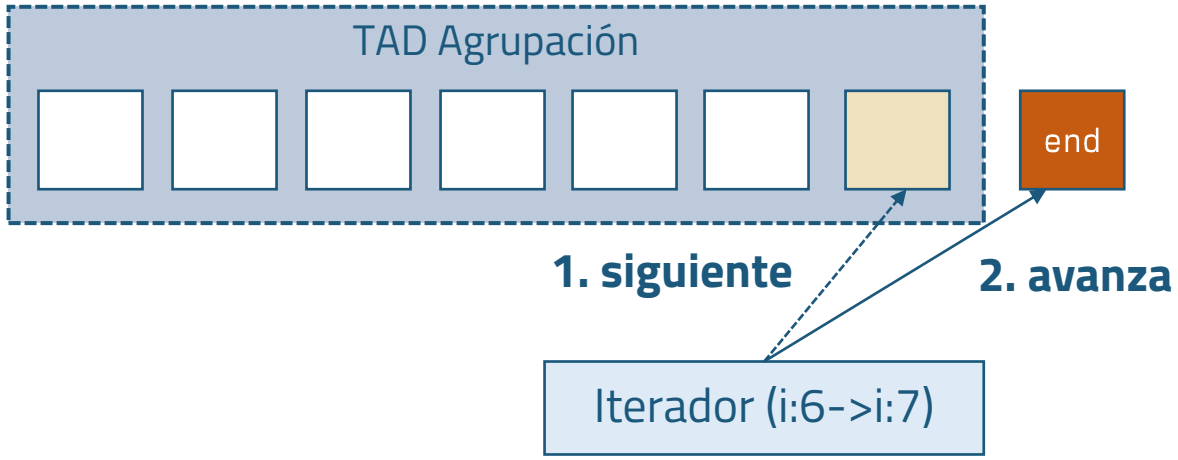
ITERADORES

- **Especificación:**

```
parcial siguiente: agrupacion a -> elemento
parcial avanza: agrupacion a -> agrupacion
```

- **Implementación C++:**

```
bool siguienteYavanza(agrupacion<T>& a, T& e);
```



ITERADORES


```
...
// Interfaz
template<typename T> void iniciarIterador(agrupacion<T>& a);
template<typename T> bool existeSiguiente(const agrupacion<T>& a);
template<typename T> bool siguienteYavanza(agrupacion<T>& a, T& e);

template<typename T>
struct agrupacion {
  ...
  friend void iniciarIterador<T>(agrupacion<T>& a);
  friend bool existeSiguiente<T>(const agrupacion<T>& a);
  friend bool siguienteYavanza<T>(agrupacion<T>& a, T& e);

private:
  // Representación interna del tipo
  T datos[MAX];
  int total;
  ...
};
#endif
```

Interfaz del iterador

Representación interna del iterador

practica2.cpp

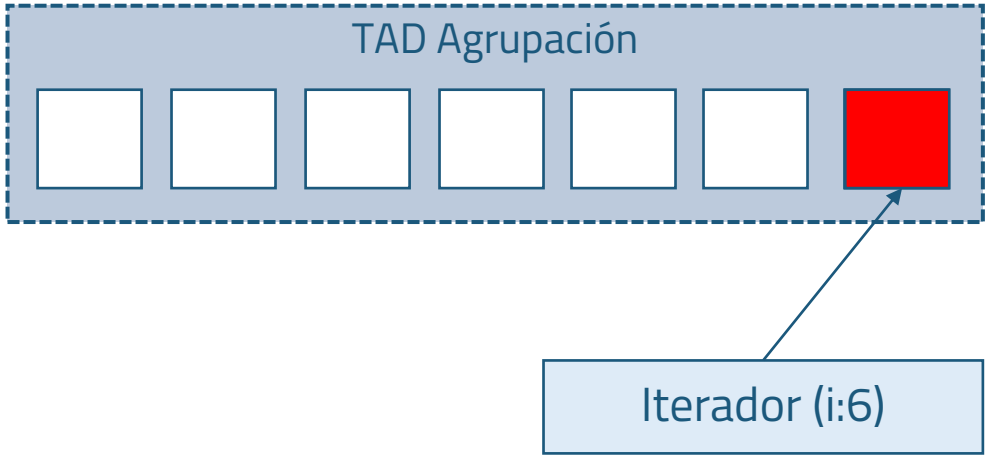
```
...
int main() {
  agrupacion<producto> micompra;
  iniciar(micompra);
  ...

  producto p; bool ok;
  iniciarIterador(micompra);
  while (existeSiguiente(micompra)) {
     ok = siguienteYavanza(micompra, p);
     // Tratar el producto siguiente obtenido en p o tratar el error
  }
  ...
}
```

ITERADORES

- **Las operaciones del iterador solo tienen sentido desde fuera del TAD.**
  - Consulta de contactos en la agrupación.
- **Es posible, pero no recomendable modificar un contenedor mientras se itera.**

```
it = vector.erase(it)
```

- **El iterador queda invalidado cuando se producen modificaciones.**



1. siguienteYavanza

2. borraUltimo

3. existeSiguiente?