

Práctica 1. Programación con TAD

Estructuras de datos y algoritmos

Grado en Ingeniería Informática – Curso académico 2026-2027

Alfonso López Ruiz

 alfonso.lopezr@unizar.es

 Despacho junto a salón de actos (Ada Byron)

 Horario de tutorías:

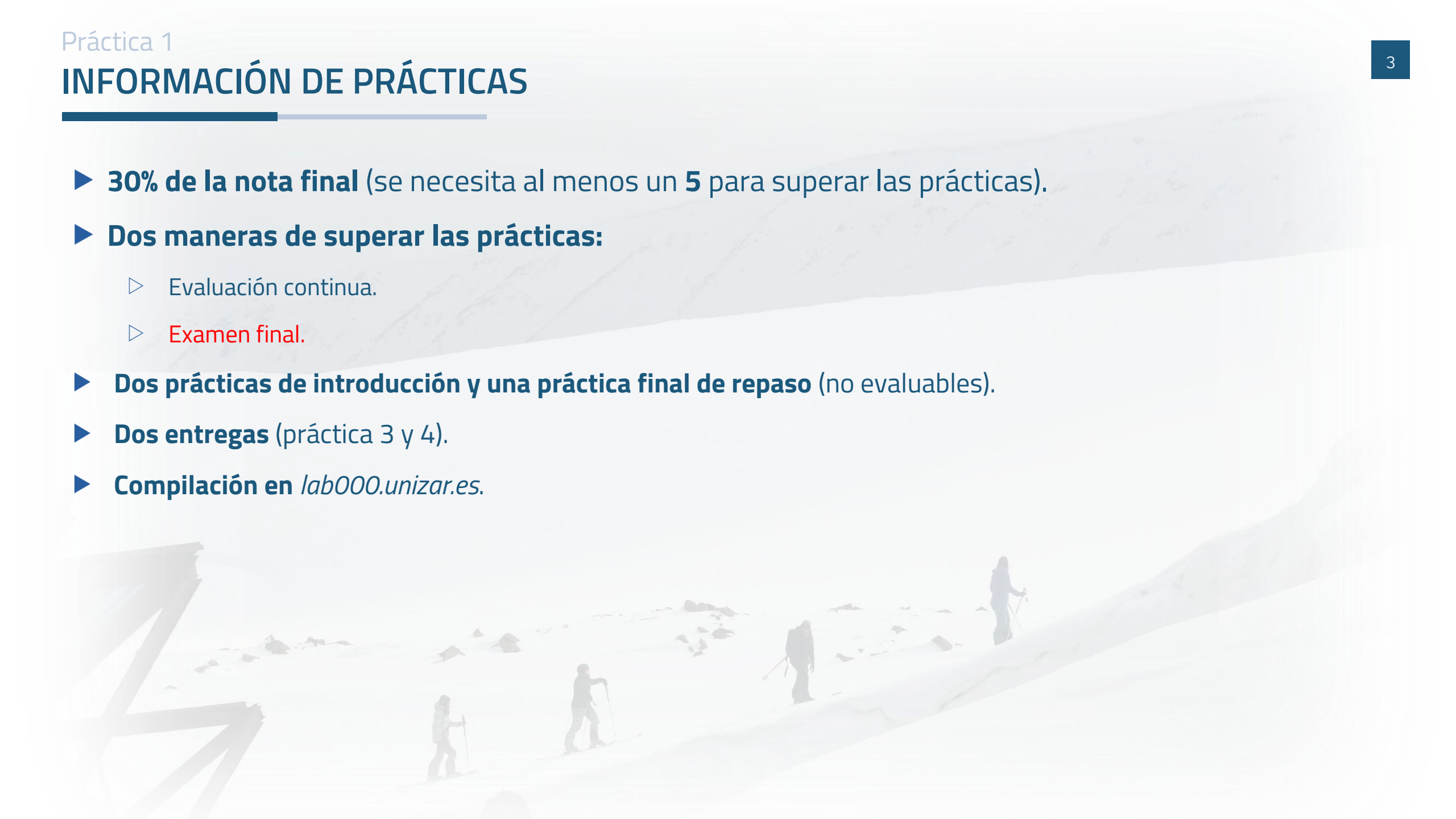
- ▷ Lunes: 15:00 – 18:00
- ▷ Martes: 15:00 – 18:00

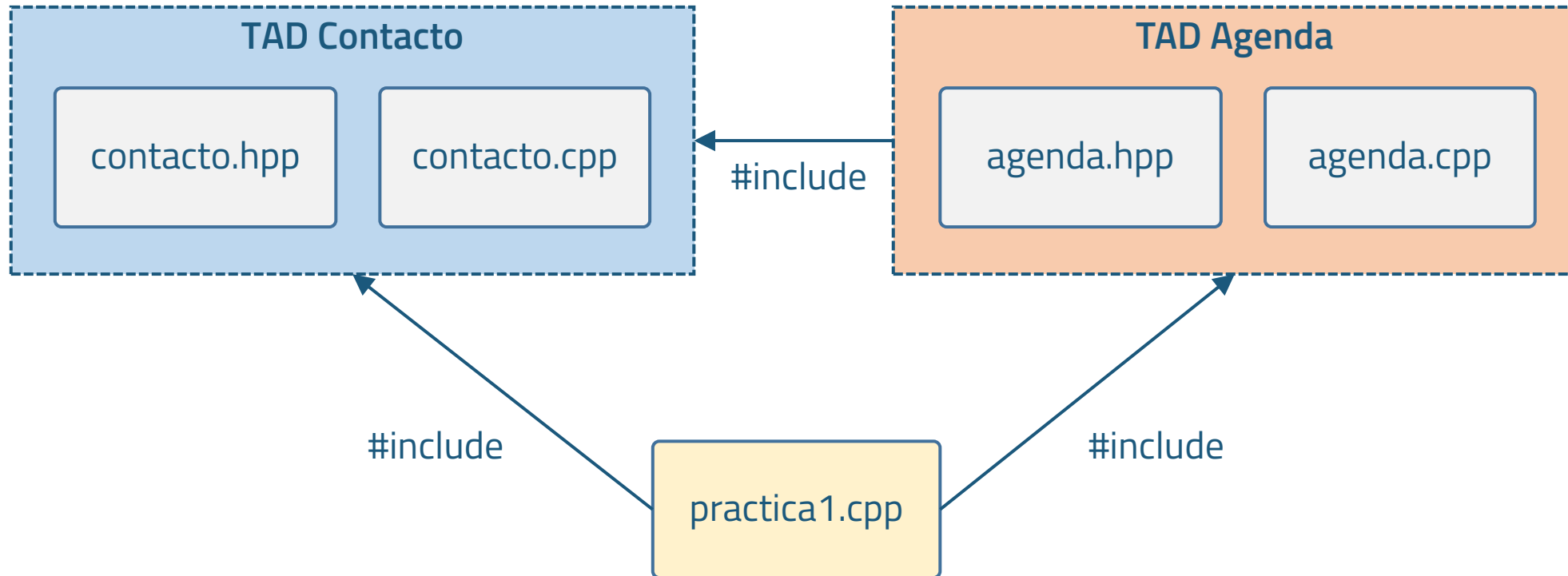
Directorio UNIZAR

<https://directorio.unizar.es/#/tutoria?colectivo=PDI&codCentro=110>

INFORMACIÓN DE PRÁCTICAS

- ▶ **30% de la nota final** (se necesita al menos un **5** para superar las prácticas).
- ▶ **Dos maneras de superar las prácticas:**
 - ▷ Evaluación continua.
 - ▷ **Examen final.**
- ▶ **Dos prácticas de introducción y una práctica final de repaso** (no evaluables).
- ▶ **Dos entregas** (práctica 3 y 4).
- ▶ **Compilación en** *lab000.unizar.es*.





ESPECIFICACIÓN DEL TAD *CONTACTO*

espec contactos

usa cadenas, enteros, booleanos

género contacto

{Los valores del TAD representan personas, que llamamos contactos, para las que se tiene información de su nombre, su dirección y su número de teléfono}

operaciones

crear: cadena nom, cadena dir, entero tel -> contacto

{Dada una cadena nom, una cadena dir, y un entero tel, se obtiene un contacto con nombre nom, dirección dir y número de teléfono tel}

nombre: contacto c -> cadena

{Dado un contacto c, se obtiene la cadena correspondiente al nombre de c}

dirección: contacto c -> cadena

{Dado un contacto c, se obtiene la cadena correspondiente a la dirección de c}

...

Constructor de un contacto

Acceso a propiedades (privadas)

ESPECIFICACIÓN DEL TAD *CONTACTO*

teléfono: contacto c -> entero

{Dado un contacto c, se obtiene el entero correspondiente al teléfono de c}

iguales: contacto c1, contacto c2 -> booleano

{Devuelve verdad si y sólo los contactos c1 y c2 tienen el mismo nombre}

fespec

IMPLEMENTACIÓN DEL TAD *CONTACTO*

```
// contacto.hpp
class Contacto {
public:
    Contacto(std::string n);
    std::string nombre() const;

private:
    std::string nombre_;
};

// contacto.cpp
Contacto::Contacto(std::string n) : nombre_(n) {}

std::string Contacto::nombre() const {
    return nombre_;
}

// main.cpp
int main() {
    Contacto c("Ana");

    std::string n = c.nombre();
}
```

```
// contacto.hpp
struct Contacto {
public:
    Contacto(std::string n);
    std::string nombre() const;

private:
    std::string nombre_;
};

// contacto.cpp
Contacto::Contacto(std::string n) : nombre_(n) {}

std::string Contacto::nombre() const {
    return nombre_;
}

// main.cpp
int main() {
    Contacto c("Ana");

    std::string n = c.nombre();
}
```

IMPLEMENTACIÓN DEL TAD *CONTACTO*

```
// contacto.hpp
struct Contacto {
    public:
        Contacto(std::string n);
        std::string nombre() const;

    private:
        std::string nombre_;
};

// contacto.cpp
Contacto::Contacto(std::string n) : nombre_(n) {}

std::string Contacto::nombre() const {
    return nombre_;
}

// main.cpp
int main() {
    Contacto c("Ana");

    std::string n = c.nombre();
}
```

```
// contacto.hpp
struct Contacto;

void crear(Contacto& c, std::string n);
std::string nombre(const Contacto& c);

struct Contacto {
    friend void crear(Contacto& c, std::string n);
    friend std::string nombre(const Contacto& c);

    private:
        std::string nombre_;
};

// contacto.cpp
void crear(Contacto& c, std::string n) {
    c.nombre_ = n;
}

std::string nombre(const Contacto& c) {
    return c.nombre_;
}

// main.cpp
int main() {
    Contacto c;
    crear(c, "Ana");
    std::string n = nombre(c);
}
```

IMPLEMENTACIÓN DEL TAD *CONTACTO*

```
// PREDECLARACION DEL TAD contacto (inicio INTERFAZ)
```

```
struct contacto;
```

```
void crear(string nombre, string direccion, int telefono, contacto& c);
string nombre(const contacto& c);
string direccion(const contacto& c);
int telefono(const contacto& c);
bool operator==(const contacto& c1, const contacto& c2);
```

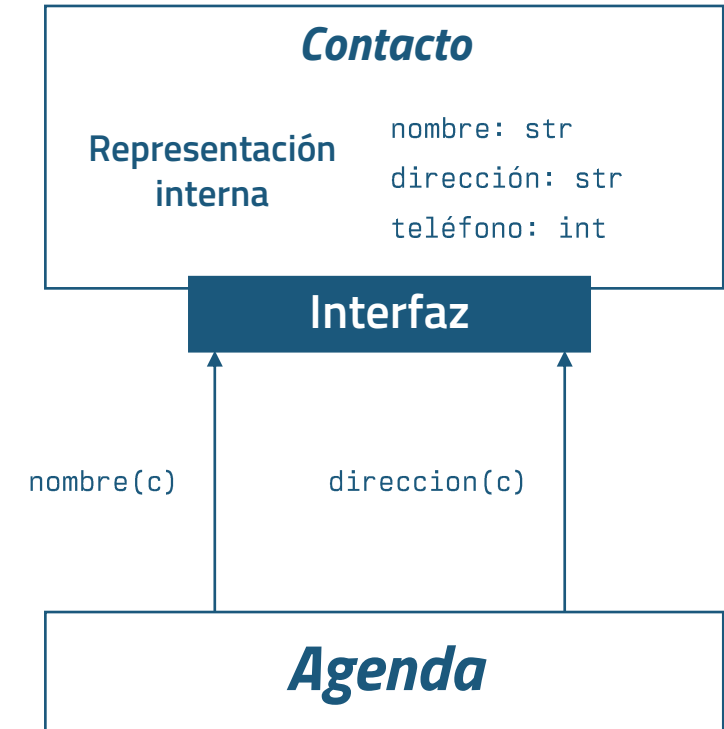
```
struct contacto {
    friend void crear(string nombre, string direccion, int telefono, contacto& c);
    friend string nombre(const contacto& c);
    friend string direccion(const contacto& c);
    friend int telefono(const contacto& c);
    friend bool operator==(const contacto& c1, const contacto& c2);
```

```
private:
```

```
    string nombre;
    string direccion;
    int telefono;
```

```
};
```

Interfaz



Representación interna

IMPLEMENTACIÓN DEL TAD *CONTACTO*

```
// PREDECLARACION DEL TAD contacto (inicio INTERFAZ)
```

```
struct contacto;
```

```
void crear(string nombre, string direccion, int telefono, contacto& c);
string nombre(const contacto& c);
string direccion(const contacto& c);
int telefono(const contacto& c);
bool operator==(const contacto& c1, const contacto& c2);
```

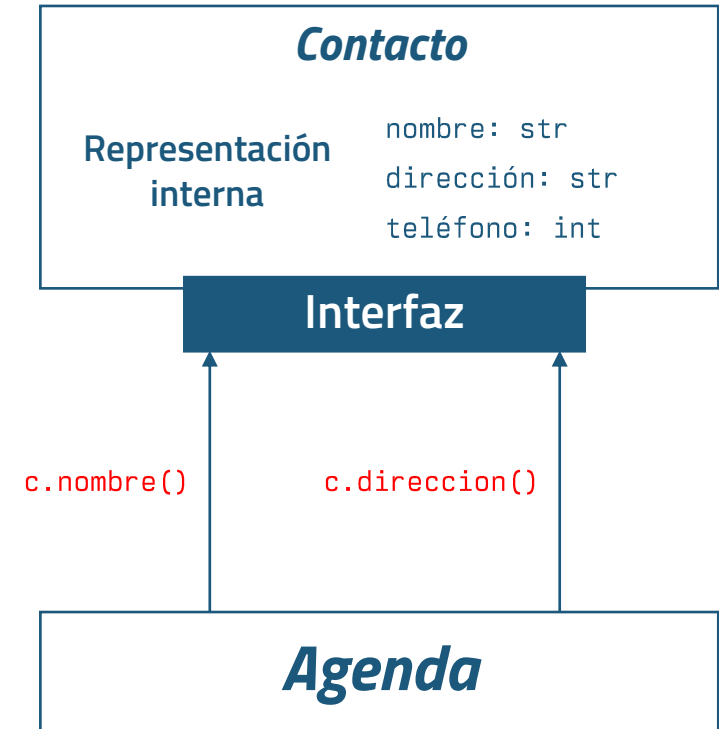
```
struct contacto {
    friend void crear(string nombre, string direccion, int telefono, contacto& c);
    friend string nombre(const contacto& c);
    friend string direccion(const contacto& c);
    friend int telefono(const contacto& c);
    friend bool operator==(const contacto& c1, const contacto& c2);
```

```
private:
```

```
    string nombre;
    string direccion;
    int telefono;
```

```
};
```

Interfaz



Representación interna

IMPLEMENTACIÓN DEL TAD *CONTACTO*

► Especificación:

crear: cadena nom, cadena dir, entero tel -> contacto

{Dada una cadena nom y otra dir y un entero tel, se obtiene un contacto de nombre nom, dirección dir y con número de teléfono tel}

► Implementación C++:

```
void crear(string nom, string dir, int tel, contacto& c);
```

```
void crear(string nom, string dir, int tel, contacto c);
```

```
contacto crear(string nom, string dir, int tel);
```

```
// Documentación
```

```
void crear(string nom, string dir, int tel, contacto& c) {
```

```
    c.nombre = nom;
```

```
    c.dirección = dir;
```

```
    c.telefono = tel;
```

```
}
```

```
// main.cpp
```

```
Contacto c ("Ana", "Zaragoza", 999, c);
```

```
Contacto c;
```

```
crear("Ana", "Zaragoza", 999, c);
```

IMPLEMENTACIÓN DEL TAD *CONTACTO*

► Especificación:

nombre: contacto c -> cadena

{Dado un contacto c, se obtiene la cadena correspondiente al nombre del contacto}

► Implementación C++:

```
string nombre(const contacto& c);
```

```
string nombre(const contacto c);
```

```
// Documentación
```

```
string nombre(const contacto& c) {  
    return c.nombre;  
}
```

```
// main.cpp
```

```
c.nombre();
```

```
nombre(c);
```

IMPLEMENTACIÓN DEL TAD *CONTACTO*

► Especificación:

iguales: contacto c1, contacto c2 -> booleano

{Devuelve verdad si y sólo los contactos c1 y c2 tienen el mismo nombre}

► Implementación C++:

```
friend bool operator==(const contacto& c1, const contacto& c2);
```

Sobrecarga de operadores

Common operators						
assignment	increment decrement	arithmetic	logical	comparison	member access	other
<pre>a = b a += b a -= b a *= b a /= b a %= b a &= b a = b a ^= b a <<= b a >>= b</pre>	<pre>++a --a a++ a--</pre>	<pre>+a -a a + b a - b a * b a / b a % b ~a a & b a b a ^ b a << b a >> b</pre>	<pre>!a a && b a b</pre>	<pre>a == b a != b a < b a > b a <= b a >= b a <=> b</pre>	<pre>a[...]</pre> <pre>*a</pre> <pre>&a</pre> <pre>a->b</pre> <pre>a.b</pre> <pre>a->*b</pre> <pre>a.*b</pre>	function call <pre>a(...)</pre> comma <pre>a, b</pre> conditional <pre>a ? b : c</pre>

IMPLEMENTACIÓN DEL TAD *AGENDA*

espec agendas
usa contactos cadenas, booleanos **#include "contacto.hpp"**
género agenda

{Los valores del TAD representan colecciones de contactos a las que se pueden añadir elementos de tipo contacto, y de las que se pueden eliminar sus contactos de uno en uno, eliminándose siempre el último contacto añadido de todos los que contenga la agenda}

operaciones

iniciar: -> agenda

{Devuelve una agenda vacía, sin contactos}

añadir: agenda a, contacto c -> agenda

{Devuelve una agenda igual a la resultante de añadir un contacto c a la agenda a.}

vacía: agenda a -> booleano

{Devuelve verdad si y sólo si la agenda a está vacía}

...

IMPLEMENTACIÓN DEL TAD *AGENDA*

► Especificación:

añadir: agenda a, contacto c -> agenda

{Devuelve una agenda igual a la resultante de añadir un contacto c a la agenda a.}

► Implementación C++:

```
// Documentación
bool anyadir (agenda& a, const contacto& c)
{
    bool sePuede = a.total < MAX_AGENDA;
    if (sePuede) {
        a.datos[a.total] = c;
        a.total++;
    }
    return sePuede;
}
```

Parámetro inout



IMPLEMENTACIÓN DEL TAD *AGENDA*

► Especificación:

añadir: agenda a, contacto c -> agenda

{Devuelve una agenda igual a la resultante de añadir un contacto c a la agenda a.}

► Implementación C++:

```
bool anyadir (agenda& a, const contacto& c);  
void anyadir (agenda& a, const contacto& c, bool& error);  
void anyadir (agenda& a, const contacto& c, int& codError);  
int  anyadir (agenda& a, const contacto& c);
```

IMPLEMENTACIÓN DEL TAD *AGENDA*

...

borrarUltimo: agenda a -> agenda

{Si a no está vacía, devuelve la agenda igual a la resultante de eliminar de a el último contacto añadido a ella. Si a está vacía, devuelve la agenda vacía}

está: agenda a, contacto c -> booleano

{Dada una agenda a y un contacto c, devuelve verdad si y sólo si en a hay algún contacto igual a c (en el sentido de la función iguales del TAD contacto), falso en caso contrario}

fespec

IMPLEMENTACIÓN DEL TAD *AGENDA*

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

```
const int MAX = 40;
```

Conjunto estático de contactos

```
struct agenda;
```

```
void iniciar(agenda& a);  
bool anyadir(agenda& a, const contacto& c);  
bool vacia(const agenda& a);  
void borrarUltimo(agenda& a);  
bool esta(const agenda& a, const contacto& c);
```

Interfaz

```
struct agenda {  
    friend void iniciar(agenda& a);  
    friend bool anyadir(agenda& a, const contacto& c);  
    friend bool vacia(const agenda& a);  
    friend void borrarUltimo(agenda& a);  
    friend bool esta(const agenda& a, const contacto& c);
```

```
private:  
    contacto datos[MAX];  
    int total;  
};
```

Representación interna

IMPLEMENTACIÓN DEL TAD *AGENDA*

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

```
const int MAX = 40;
```

```
struct agenda;
```

```
void iniciar(agenda& a);
```

```
bool anyadir(agenda& a, const contacto& c);
```

```
bool vacía(const agenda& a);
```

```
void borrarUltimo(agenda& a);
```

```
bool esta(const agenda& a, const contacto& c);
```

```
struct agenda {
```

```
    friend void iniciar(agenda& a);
```

```
    friend bool anyadir(agenda& a, const contacto& c);
```

```
    friend bool vacía(const agenda& a);
```

```
    friend void borrarUltimo(agenda& a);
```

```
    friend bool esta(const agenda& a, const contacto& c);
```

```
private:
```

```
    contacto datos[MAX];
```

```
    int total;
```

```
};
```

```
// Documentación
```

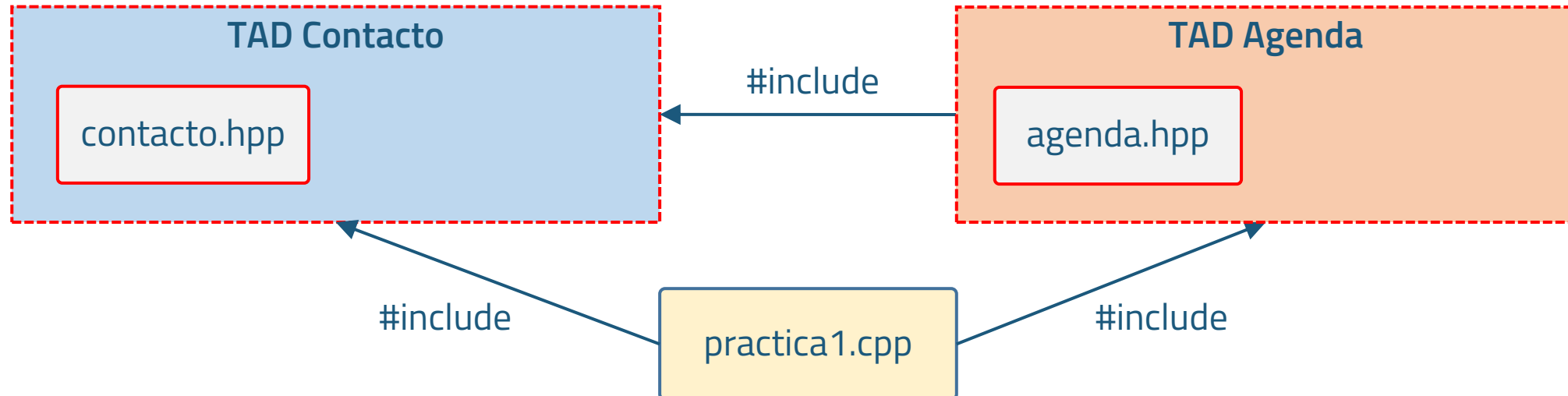
```
void iniciar (agenda& a) {
```

```
    a.total = 0;
```

```
}
```

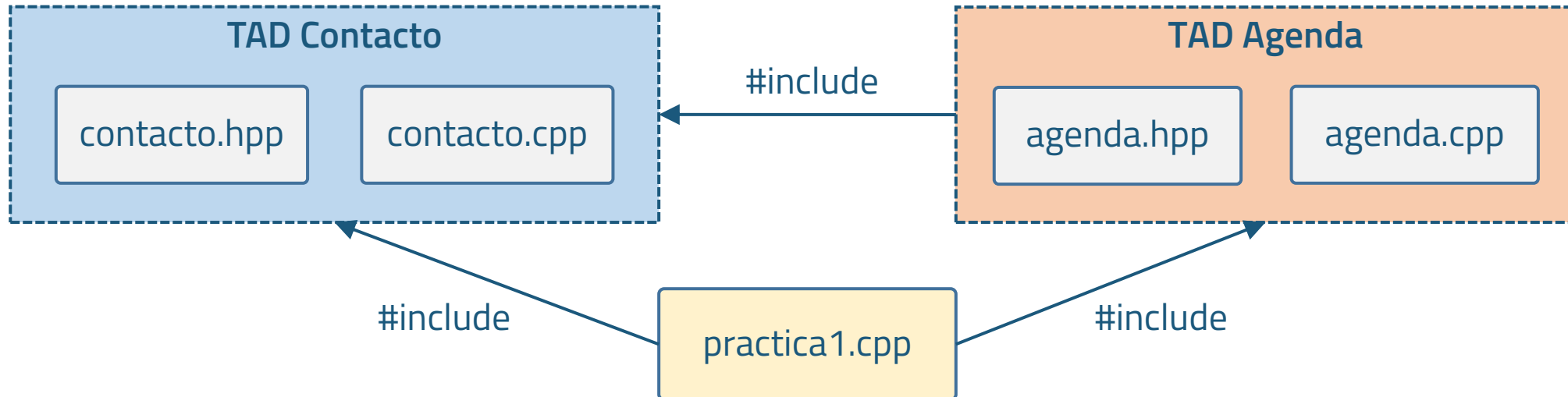
OTROS DETALLES DE IMPLEMENTACIÓN

- ▶ Un iterador nos permite **recorrer un conjunto de elementos**.
 - ▷ De manera temporal, podemos crear una función ***mostrarTodos***.
 - ▷ Itera por todos los elementos y muestra su contenido.
- ▶ **Compilación de programa:** `g++ -std=c++11 practica1.cpp ... -o nombrePrograma`



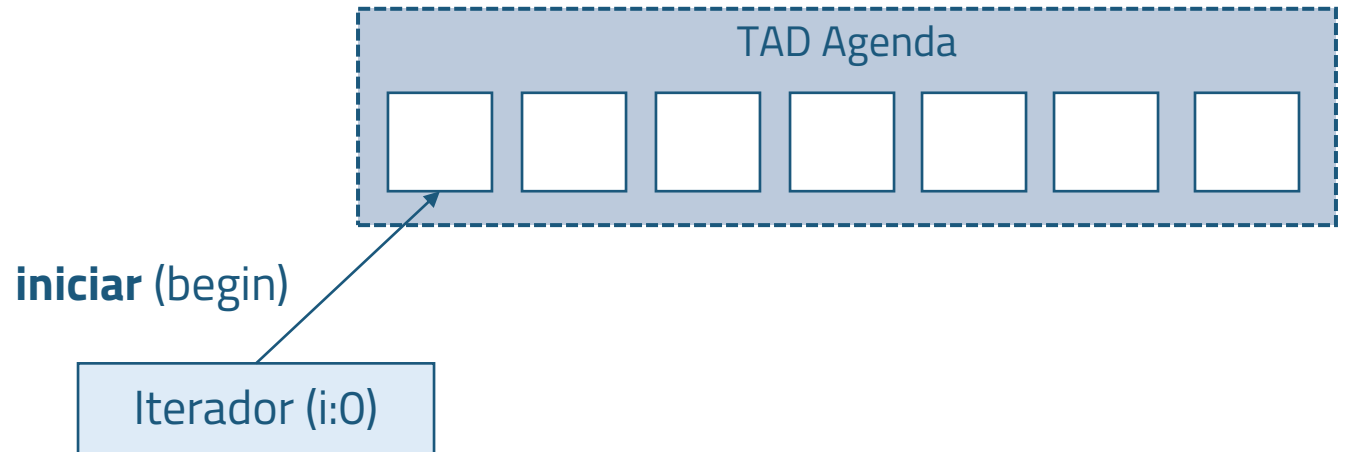
OTROS DETALLES DE IMPLEMENTACIÓN

- ▶ Un iterador nos permite **recorrer un conjunto de elementos**.
 - ▷ De manera temporal, podemos crear una función ***mostrarTodos***.
 - ▷ Itera por todos los elementos y muestra su contenido.
- ▶ **Compilación de programa:** `g++ -std=c++11 practica1.cpp contacto.cpp agenda.cpp ... -o nombrePrograma`



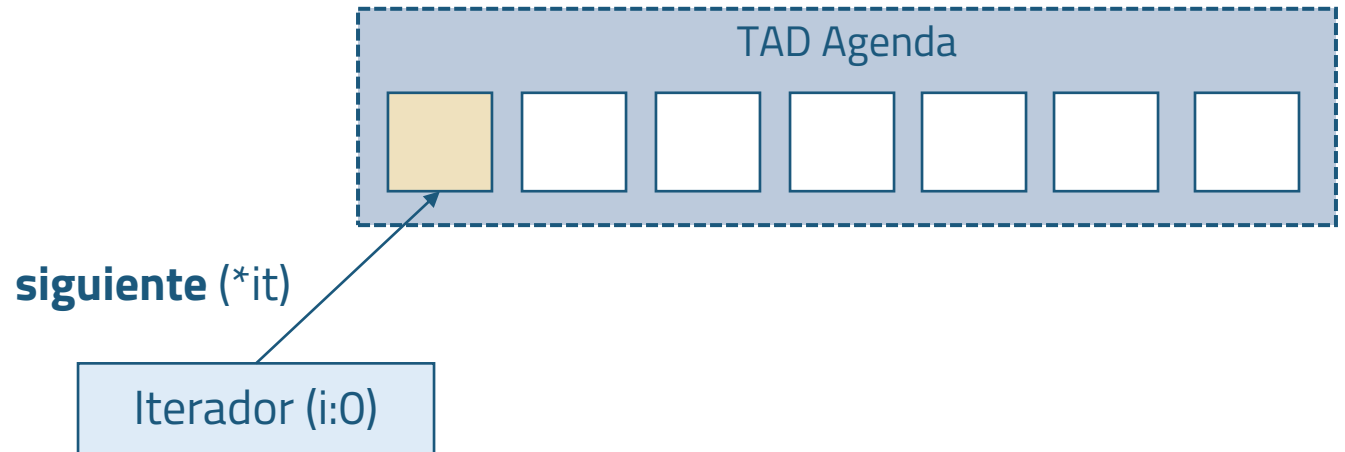
ITERADORES

- ▶ 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.



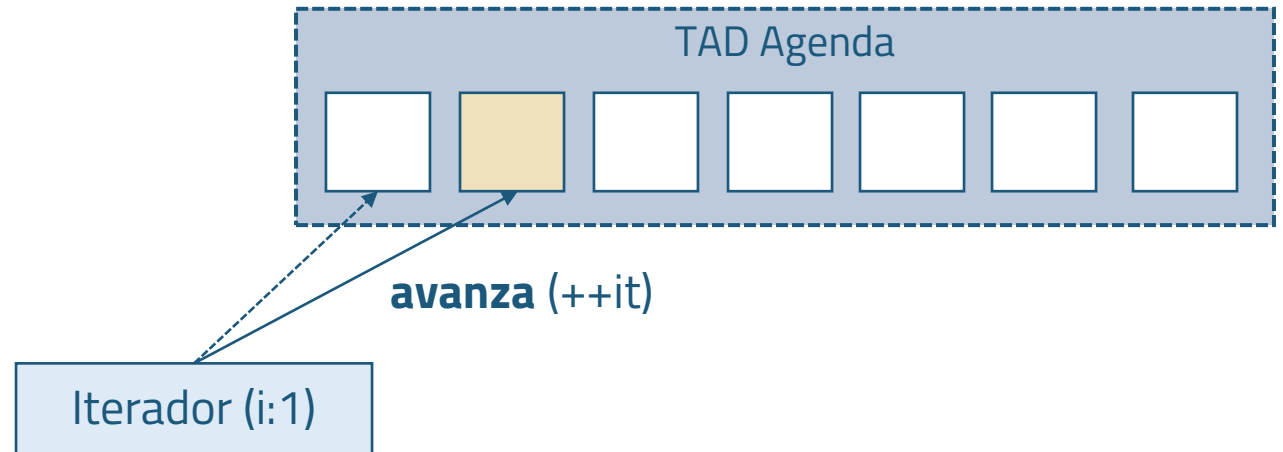
ITERADORES

- ▶ 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

- ▶ 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.

practica1.cpp

```
...  
int main() {  
    agenda miagenda;  
    ...  
    iniciarIterador(miagenda);  
    while (existeSiguiete(miagenda)) {  
        ok = siguienteYavanza(miagenda, c);  
        // Tratar el contacto siguiente obtenido en c o tratar el error  
    }  
    ...  
}
```

ITERADORES

espec agendas

usa contactos, cadenas, booleanos

género agenda

...

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}

parcial siguiente: agenda a → contacto

{Devuelve el siguiente contacto a visitar. Parcial: la operación no está definida si no quedan contactos por visitar (no existeSiguiente?)}

parcial avanza: agenda a → agenda

{Prepara el iterador para que se pueda visitar el siguiente contacto.

Parcial: la operación no está definida si no quedan contactos por visitar (no existeSiguiente?(a)}

fespec

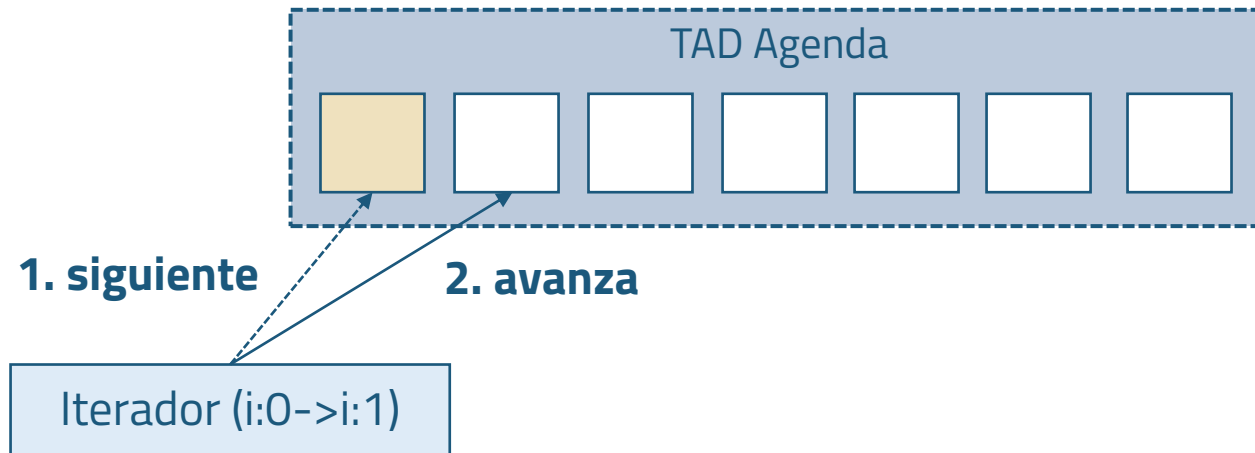
► Especificación:

`parcial` siguiente: agenda a -> contacto

`parcial` avanza: agenda a -> agenda

► Implementación C++:

```
bool siguienteYavanza (agenda& a, contacto& e);
```



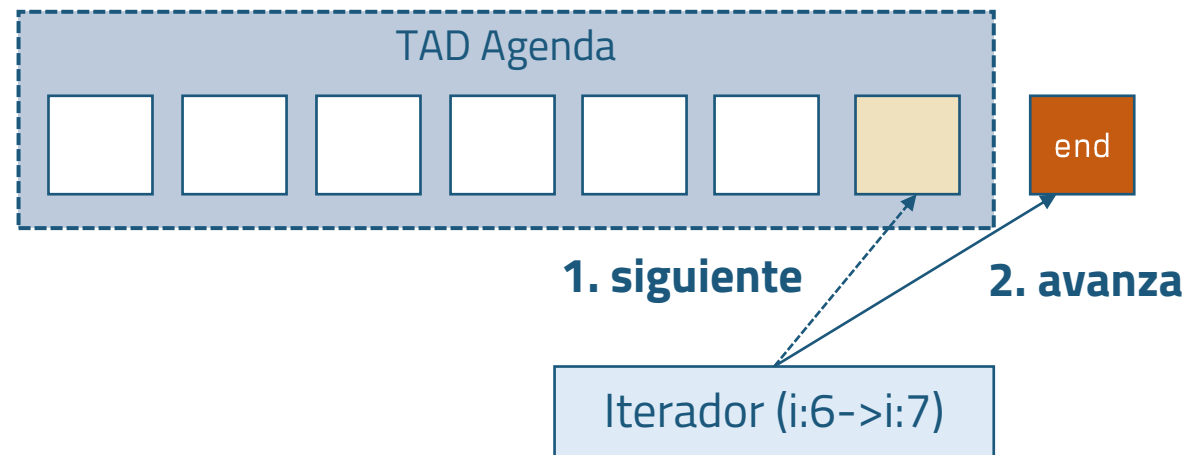
► Especificación:

`parcial` siguiente: agenda a -> contacto

`parcial` avanza: agenda a -> agenda

► Implementación C++:

```
bool siguienteYavanza (agenda& a, contacto& e);
```



```
#include "contacto.hpp"

// Interfaz
void iniciarIterador(agenda& a);
bool existeSiguiete(const agenda& a);
bool siguieteYavanza(agenda & a, contacto& c);
```

```
struct agenda {
```

```
...
```

```
friend void iniciarIterador (agenda& a);
friend bool existeSiguiete (const agenda& a);
friend bool siguieteYavanza (agenda & a, contacto& c);
```

```
private:
```

```
// Representación interna del tipo
```

```
contacto datos[MAX_AGENDA];
```

```
int total;
```

```
...
```

```
};
```

```
#endif
```

Interfaz del iterador

Representación interna del iterador

practica1.cpp

```
...
int main() {
    agenda miagenda;
    ...
    iniciarIterador(miagenda);
    while (existeSiguiete(miagenda)) {
        ok = siguieteYavanza(miagenda, c);
        // Tratar el contacto siguiete obtenido en c o tratar el error
    }
    ...
}
```

► **Las operaciones del iterador solo tienen sentido desde fuera del TAD.**

▷ Consulta de contactos en la agenda.

► **Es posible, pero no recomendable modificar un contenedor mientras se itera.**

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

► Normalmente, se trata de una operación de búsqueda + inserción/borrado.

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

