Implementación de TADs (no genéricos) en C++

Práctica 1

Objetivo

- Hacer un programa para gestionar un grupo de contactos.
  - Necesitamos los TADs *contacto* y *agenda*

TAD Contacto: especificación

espec contactos
  usa cadenas, enteros, booleanos
  género contacto *{Los valores del TAD contacto representan personas, que llamamos contactos, para los 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 de nombre nom, dirección dir y con número de teléfono tel}

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

  direccion: contacto c → cadena
  {Dado un contacto c, se obtiene la cadena correspondiente a la dirección del contacto c}
...

TAD Contacto: especificación

…

teléfono: contacto c → entero
{Dado un contacto c, se obtiene el entero correspondiente al número de teléfono del contacto c}

iguales: contacto c1, contacto c2 → booleano
{Devuelve verdad si y sólo si los contactos c1 y c2 tienen el mismo nombre}

fespec

Implementación: fichero *contacto.hpp*

```
#ifndef CONTACTO_HPP
#define CONTACTO_HPP
#include<iostream> //para utilizar el tipo de dato string
using namespace std;

// Inicio interfaz del TAD contacto.
// Pre-declaración:

struct contacto; //…(ver en el fichero)

void crear(string nom, string dir, int tel, 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); //…

// Fin interfaz del TAD contacto.

... (sigue)
```

Implementación: fichero *contacto.hpp*

```
// Declaración:
struct contacto {
    friend void crear(string nom, string dir, int tel,
                      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: //declaración de la representación interna del tipo:
      //campos de contacto (privados => encapsulación)
      //…documentación sobre la representación interna……
      string nombre;
      string dirección;
      int telefono;
};
#endif
```

Observaciones

- 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);
```

Observaciones

- Especificación:

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

- Implementación C++: los parámetros de entrada (de tipos no básicos) se declararán como referencias constantes.

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

Implementación: fichero *contacto.cpp*

```
#include "contacto.hpp"

// Implementación de las operaciones del TAD

//…documentación sobre la operación…
void crear(string nom, string dir, int tel, contacto& c){
    c.nombre = nom;
    c.dirección = dir;
    c.telefono = tel;
}
//…documentación sobre la operación…
string nombre(const contacto& c){
    return c.nombre;
}

// etc
```

TAD Agenda: Especificación

espec agendas

usa contactos, booleanos

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 la 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}

...

TAD Agenda: Especificación

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 operación iguales del TAD contacto), falso en caso contrario}

fespec

Implementación: fichero *agenda.hpp*

```
#ifndef AGENDA_HPP
#define AGENDA_HPP

#include "contacto.hpp"

// Inicio interfaz del TAD agenda.
//Pre-declaración:

const int MAX_AGENDA = 40; //Límite tamaño de la agenda
                           //en esta implementación.
struct agenda; //… (ver en el fichero)
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); //…

// Fin interfaz del TAD agenda.
```

Implementación: fichero *agenda.hpp*

```
// Declaración:

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: // declaración de la representación interna del tipo:
      //...completar con documentación sobre la representación interna...
      contacto datos[MAX_AGENDA];
      int total;

};

#endif
```

Observaciones

- Especificación:
  - el tamaño de la agenda como colección de contactos no está limitado

- Implementación C++
  - Implementación en memoria estática (vector): limita el tamaño de la colección al tamaño del vector utilizado
  - Debe documentarse para informar a los posibles usuarios de la implementación

```
const int MAX_AGENDA = 40; //Límite del tamaño
                           //de la agenda
```

Observaciones: operaciones parciales

- En la especificación: la operación añadir no es parcial

```
añadir: agenda a, contacto c → agenda
 {Devuelve la agenda igual a la resultante de añadir un
  contacto c a la agenda a.}
```

- Implementación C++ con tamaño limitado para la colección:
  - La operación se implementa como parcial (no siempre se puede añadir)
  - devuelve verdad si NO ha podido añadir el contacto.

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

- Otras formas permitidas:

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

Implementación: fichero *agenda.cpp*

```
#include "agenda.hpp"

//…documentación sobre la operación…
void iniciar (agenda& a) {
    a.total = 0;
}
//…documentación sobre la operación…
bool anyadir (agenda& a, const contacto& c) {
    bool sePuede = a.total < MAX_AGENDA;
    if (sePuede) {
        a.datos[a.total] = c;
        a.total++;
     }
     return (!sePuede);
}

// etc
```

Iteradores

- Un *iterador*, definido para un contenedor, se utiliza para:
  - visitar/recorrer todos los elementos del contenedor (sin dejarse ninguno, ni visitar ninguno más de una vez)
  - Se utiliza fuera de la implementación del TAD contenedor
  - Permite visitar/recorrer todos los datos del contenedor, sin exponer los detalles de la implementación del TAD contenedor
- Operaciones básicas de un iterador:
  - ***iniciarIterador***: prepara el iterador para que el siguiente elemento a visitar sea el primero.
  - ***existeSiguiente?***: devuelve falso si ya se ha visitado el último elemento, devuelve verdad en caso contrario.
  - ***siguiente***: devuelve el siguiente elemento a visitar.

    Parcial: la operación no está definida si ya se ha visitado el último elemento.
  - ***avanza***: prepara el iterador para que se pueda visitar otro elemento.
    Parcial: la operación no está definida si ya se ha visitado el último elemento.

TAD Agenda: Especificación

espec agendas

usa contactos, booleanos

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 la 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}

...

TAD Agenda: Especificación

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 operación iguales del TAD contacto), falso en caso contrario}

. . . *{le añadimos las operaciones del iterador…}*

Agenda con iterador

. . . {operaciones del iterador para agendas:}

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?(a)}
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

Observaciones

- Especificación:

```
parcial siguiente: agenda a → contacto
{Devuelve el siguiente elemento a visitar.
 Parcial: la operación no está definida si no quedan elementos
 por visitar ( no existeSiguiente?(a)}

parcial avanza: agenda a → agenda
{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)}
```

- Implementación C++: ambas operaciones juntas

```
bool siguienteYavanza (agenda& a, contacto& e);
{Si existe algún contacto pendiente de visitar, modifica e
con el siguiente contacto a visitar, y además después avanza
el iterador para que a continuación se pueda visitar otro
contacto, y devuelve true. Si no quedaban contactos
pendientes por visitar, devuelve false.}
```

Implementación: fichero *agenda.hpp*

```
#ifndef AGENDA_HPP
#define AGENDA_HPP
#include "contacto.hpp"
// Inicio interfaz del TAD agenda. Pre-declaración:
. . .
void iniciarIterador (agenda& a);
bool existeSiguiente (const agenda& a);
bool siguienteYavanza (agenda & a, contacto& c);
// Fin interfaz del TAD agenda.
// Declaración:
struct agenda{
    . . .
 friend void iniciarIterador (agenda& a);
 friend bool existeSiguiente (const agenda& a);
 friend bool siguienteYavanza (agenda & a, contacto& c);
 private: // declaración de la representación interna del tipo:
     contacto datos[MAX_AGENDA];
     int total;
     . . . .


};

#endif
```

Añadiremos algo, a la **representación interna** del tipo, **para** gestionar cuál es el ***estado del iterador*** **:**

- Lo que le añadamos, **únicamente lo usarán las operaciones que implementan el iterador** (y **no** deberán usarlo las otras operaciones del contenedor)
- Las operaciones del contenedor **no usarán a las operaciones del iterador** (ni siquiera las que no modifican el contenedor)

Ejemplo de uso del iterador: PRÁCTICA 1

- Las operaciones del iterador **se usarán fuera de la implementación del TAD**
- **Nunca** se debe modificar el contenedor (en este caso, la agenda) mientras se recorre con las operaciones de un iterador

```
// Por ejemplo, en el main de P1:
agenda miagenda;
iniciar(miagenda);
. . . . //crear contactos y añadirlos a la agenda …
contacto c; bool ok;
//Recorrer todos los contactos de la agenda:
//Inicio recorrido: a partir de aquí NO se debe modificar miagenda
iniciarIterador(miagenda);
while (existeSiguiente(miagenda)) {
    ok = siguienteYavanza(miagenda,c);
   //tratar el contacto siguiente obtenido en c,
   //o tratar el error
}
//Fin del recorrido: se puede volver a modificar miagenda
. . .
```

Iteradores

Validez de un iterador:

- *Un iterador tiene validez garantizada* ***mientras no se modifique el contenedor*** *al que referencia*
  - **Sirve para recorrer un contenedor en un estado determinado**
  - El uso de operaciones que modifiquen el contenedor mientras está siendo recorrido, tendrá efectos imprevisibles(*) en el iterador y en general lo harán incorrecto
  - *El que un iterador quede invalidado o no cuando se modifica el contenedor que referencia, depende del tipo de contenedor, de la alteración realizada, y de sus implementaciones.*
    - *Detalles que no debe necesitar conocer el usuario del contenedor o del iterador*
- Los iteradores sirven para implementar (desde fuera del contenedor) **algoritmos de recorrido o búsqueda en el contenedor, pero NO algoritmos que modifiquen el contenedor**

(*) *A menos que se trate de un contenedor especificado de tal forma que todas sus operaciones contemplen el estado del iterador y especifiquen su efecto sobre el.*