Implementación TADs en C++

Práctica 0

Índice

1. Implementación C++ TADs no genéricos
   - Implementación operaciones
   - Implementación operaciones parciales
2. Implementación C++ TADs genéricos
   - Operaciones del parámetro formal
3. Iteradores

Ejemplos

- Hacer un programa para gestionar una lista de la compra.
  - Necesitamos los TADs *producto* y *compra*
- Hacer un programa para gestionar un grupo de contactos.
  - Necesitamos los TADs *contacto* y *agenda*

TAD Producto: especificación

```
espec productos
  usa cadenas, enteros
  género producto {Los valores del TAD productos 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}

fespec
```

Implementación: fichero *producto.h*

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

// Interfaz del TAD producto. Pre-declaraciones:
struct producto;
void crear (string nom, int can, producto& p);
string nombre (const producto& p);
int cantidad (const producto& p);
// Declaración:
struct producto {
    friend void crear (string nom, int can, producto& p);
    friend string nombre (const producto& p);
    friend int cantidad (const producto& p);
  private: //declaración de la representación interna del tipo:
     //campos de producto:
     string nombre;
     int cantidad;
};
#endif
```


Observaciones

- Especificación:

  `crear: cadena nom, entero can → producto`

  {Dada una cadena nom y un entero c, se obtiene un producto de nombre nom y con cantidad de producto c}

- Implementación C++

```
void crear (string nom, int can, producto& p);
```


Observaciones

- Especificación:

  `nombre: producto c → cadena`

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

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

```
string nombre(const producto& p);
```


Implementación: fichero *producto.cc*

```
#include "producto.h"

void crear (string nom, int can, producto& p){
    p.nombre = nom;
    p.cantidad = can;
}

string nombre (const producto& p){
    return p.nombre;
}

int cantidad (const producto& p){
    return p.cantidad;
}
```

TAD Compra: Especificación

```
espec compras
  usa productos
  género compra {Los valores del TAD compra representan colecciones
     de productos a las que se pueden añadir elementos de tipo
     producto, y de las que se pueden eliminar sus productos de uno
     en uno eliminándose siempre el último producto añadido de todos
     los que contenga la compra)}
operaciones
   iniciar: → compra
   {Devuelve una compra vacía, sin productos}

   añadir: compra c, producto p → compra
   {Devuelve la compra resultante de añadir un producto p a
    una compra c.}

   parcial borrarUltimo: compra c → compra
   {Devuelve la compra resultante de eliminar de c el último
     producto añadido a c.
    Parcial: la operación no está definida si la compra está
      vacía.}
fespec
```


Implementación: fichero *compra.h*

```
#ifndef COMPRA_H
#define COMPRA_H
#include "producto.h"

// Interfaz del TAD compra. Pre-declaraciones:
const int MAX_COMPRA = 40; //Limite del tamaño de la compra, en
                           // esta implementación.
struct compra;
void iniciar (compra& c);
bool anyadir (compra& c, const producto& p);
bool borrarUltimo (compra& c);
// Declaración:
struct compra{
    friend void iniciar (compra& c);
    friend bool anyadir (compra& c, const producto& p);
    friend bool borrarUltimo (compra& c);
  private://declaración de la representación interna del tipo:
     producto lacompra[MAX_COMPRA];
     int total;
};
#endif
```


Observaciones

- Especificación:
  - el tamaño de la compra como colección de productos 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_COMPRA = 40; //Límite del tamaño
                           // de la compra
```


Observaciones: operaciones parciales

- En la especificación: la operación añadir no es parcial

```
añadir: compra c, producto p → compra
 {Devuelve la compra resultante de añadir un producto p a
 una compra c.}
```

- Implementación C++ con tamaño limitado para la colección:
  - La operación se implementa como parcial (no siempre se puede añadir)
  - devuelve verdad si no hay error.

```
bool anyadir (compra& c, const producto& p);
```

- Otras formas permitidas:

```
void anyadir (compra& c, const producto& p, bool& error);
int anyadir (compra& c, const producto& p);
void anyadir (compra& c, const producto& p, int& cod_er);
```

Observaciones: operaciones parciales

- Especificación

```
parcial borrarUltimo: compra c → compra
{Devuelve la compra resultante de eliminar de c el último
 producto añadido a c.
Parcial: la operación no está definida si la compra está
 vacía.}
```

- Implementación C++: devuelve verdad si se ha podido borrar (no se ha producido error)

```
bool borrarUltimo (compra& c);
```

- Otras formas permitidas:

```
void borrarUltimo (compra& c, bool& error);
int borrarUltimo (compra& c);
void borrarUltimo (compra& c, int& cod_er);
```


Implementación: fichero *compra.cc*

```
#include "compra.h"
void iniciar (compra& c) {
    c.total = 0;
}

bool anyadir (compra& c, const producto& p) {
    bool sePuede = c.total < MAX_COMPRA;
    if (sePuede) {
        c.lacompra[c.total] = p;
        c.total++;
     }
     return sePuede;
}

bool borrarUltimo (compra& c) {
    bool sePuede = c.total > 0;
    if (sePuede) {
        c.total--;
    }
    return sePuede;
}
```


TAD Contacto: especificación

```
espec contactos
  usa cadenas, enteros
  género contacto {Los valores del TAD contactos representan
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 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 del contacto c}
  dirección: contacto c → cadena
  {Dado un contacto c, se obtiene la cadena correspondiente a
  la dirección del contacto c}
  teléfono: contacto c → entero
  {Dado un contacto c, se obtiene el entero cadena
  correspondiente al teléfono del contacto c}
fespec
```


Implementación del TAD contacto

- Similar a la implementación de producto.

- Crear un fichero *contacto.h* con declaración del tipo y las operaciones de *contacto*

- Crear un fichero *contacto.cc* con la implementación de las operaciones de *contacto*.

TAD Agenda: Especificación

```
espec agendas
  usa contactos
  género agenda {Los valores del TAD agenda 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 resultante de añadir un contacto c a
    una agenda a.}

   parcial borrarUltimo: agenda b → agenda
   {Devuelve la agenda resultante de eliminar de b el último
     contacto añadido a b.
    Parcial: la operación no está definida si la agenda está
      vacía.}
fespec
```


Implementación: fichero *agenda.h*

```
#ifndef AGENDA_H
#define AGENDA_H

#include "contacto.h"
// Interfaz del TAD agenda. Pre-declaraciones:
const int MAX_AGENDA = 40; //Límite tamaño de la agenda, en
                           // esta implementación.
struct agenda;
void iniciar (agenda& c);
bool anyadir (agenda& c, const contacto& p);
bool borrarUltimo (agenda& c);
// Declaración:
struct agenda{
    friend void iniciar (agenda& c);
    friend bool anyadir (agenda& c, const contacto& p);
    friend bool borrarUltimo (agenda& c);
  private: //declaración de la representación interna del tipo:
     contacto laagenda[MAX_AGENDA];
     int total;
};
#endif
```


Implementación: fichero *agenda.cc*

```
#include "agenda.h"

void iniciar (agenda& c) {
    c.total = 0;
}

bool anyadir (agenda& c, const contacto& p) {
    bool sePuede = c.total < MAX_AGENDA;
    if (sePuede) {
        c.laagenda[c.total] = p;
        c.total++;
     }
     return sePuede;
}

bool borrarUltimo (agenda& c) {
    bool sePuede = c.total > 0;
    if (sePuede) {
        c.total--;
    }
    return sePuede;
}
```


¿Otra solución?

- La compra y la agenda son casi iguales, incluso en su especificación
- Única diferencia: guardar un producto o un contacto
- Solución: programar con tipos genéricos
  - Una compra es una "agrupación" de datos producto
  - Una agenda es una "agrupación" de datos contacto

Agrupación: TAD genérico

```
espec agrupaciones

  parámetro formal
     género elem
  fpf
  género agrupación {Los valores del TAD agrupaciones representan
     colecciones a las que se pueden añadir elementos de tipo elem,
     y de las que se pueden eliminar sus elementos de uno en uno
     (eliminándose siempre el último elemento añadido de todos los
     que contenga la agrupación)}
operaciones
   iniciar: → agrupación
   {Devuelve una agrupación vacía, sin elementos}
   añadir: agrupación c, elem e → agrupación
   {Devuelve la agrupación resultante de añadir un elemento
    e a una agrupación c.}
   parcial borrarUltimo: agrupación c → agrupación
   {Devuelve la agrupación resultante de eliminar de c el
    último elemento añadido a c.
    Parcial: la operación no está definida si la agrupación
    está vacía.}
fespec
```


Diferencia implementación TAD genérico y no genérico

- Si un **TAD es genérico**, su implementación se hará:
  - en un único fichero *.h* (declaración e implementación de las operaciones)
- Si un **TAD es no genérico,** su implementación se dividirá en:
  - un fichero *.h* (con las declaraciones);
  - y un fichero *.cc* o *.cpp* (con la implementación de las operaciones)

Implementación genéricos: *agrupacion.h*

```
#ifndef AGRUPACION_H
#define AGRUPACION_H
// Interfaz del TAD agrupación genérico. Pre-declaraciones:
const int MAX = 40; //Límite tamaño de la agrupación, en
                    // esta implementación.
template<typename T> struct agrupacion;
//definir operaciones de agrupacion
template<typename T> void iniciar (agrupacion<T>& c);
template<typename T> bool anyadir (agrupacion<T>& c, const T& p);
template<typename T> bool borrarUltimo (agrupacion<T>& c);
// Declaración:
template<typename T>
struct agrupacion{
    friend void iniciar<T> (agrupacion<T>& c);
    friend bool anyadir<T> (agrupacion<T>& c, const T& p);
    friend bool borrarUltimo<T> (agrupacion<T>& c);
  private://declaración de la representación interna del tipo:
     T datos[MAX];
     int total;
};
//ATENCIÓN: para tipos de datos genéricos
//la implementación de las operaciones también estará en agrupacion.h
                            // ...continua agrupacion.h . . .
```


Implementación: continuación *agrupacion.h*

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

template<typename T>
bool anyadir (agrupacion<T>& c, const T& p) {
    bool sePuede = c.total < MAX;
    if (sePuede) {
        c.datos[c.total] = p;
        c.total++;
     }
     return sePuede;
}

template<typename T>
bool borrarUltimo (agrupacion<T>& c) {
    bool sePuede = c.total > 0;
    if (sePuede) c.total--;
    return sePuede;
}
#endif //fin de agrupacion.h
```

Uso de genéricos: concretar parámetros formales

```
#include "agrupacion.h"
#include "producto.h"
#include "contacto.h"

int main() {
   producto nar;
   crear("naranjas", 2, nar);
   producto per;
   crear("peras", 4, per);

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

   contacto c1;
   crear("pepe", "calle 1", 976555555, c1);
   contacto c2;
   crear("ana", "calle 2", 976444444, c2);

   agrupacion<contacto> agenda; iniciar(agenda);
   anyadir(agenda, c1); anyadir(agenda, c2);
}
```


Operaciones sobre el parámetro formal

- Añadir una nueva operación a la agrupación que indique si un elemento está o no en la 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 son iguales, devuelve falso en caso contrario}
  fpf
  género agrupación

operaciones
   . . . .
   está: agrupación c, elem e → booleano
   {Dada una agrupación c y un elemento e, devuelve verdad si y
   sólo si el elemento e está en c, falso en caso contrario}
fespec
```


Implementación: Cuidado!!

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

- Si al concretar el parámetro formal, el tipo de dato no tiene definido el operador ==, error
- **Error en tiempo de compilación**: no existe == para producto

Una implementación correcta

- El tipo de dato concreto debe tener declarado e implementado el operador ==.
- **Para sobrecargar el operador ==**
  Por ejemplo, en *producto.h* podemos hacerlo añadiendo:

```
//Interfaz del TAD. Pre-declaraciones:
. . . . . .
bool operator== (const producto& p1, const producto& p2);

struct producto {
   . . . .
   friend bool operator== (const producto& p1,
                           const producto& p2);
}
```

Y en *producto.cc*, añadiremos su implementación …

Sobrecarga de operadores

- De forma similar, si se necesita, podemos sobrecargar los operadores

<, <=, >, >=

```
bool operator<(const producto& p1, const producto& p2);

bool operator<=(const producto& p1, const producto& p2);

bool operator>(const producto& p1, const producto& p2);

bool operator>=(const producto& p1, const producto& p2);
```


Implementación genéricos: *agrupacion.h*

```
#ifndef AGRUPACION_H
#define AGRUPACION_H
// Interfaz del TAD agrupación genérico.
// Pre-declaraciones:
// El tipo T requerirá tener definida una función:
//       bool operator== (const T& t1, const T& t2);
```

En C++ podremos indicar las restricciones sobre los tipos únicamente en los comentarios (por tanto el compilador no podrá comprobar que el genérico se usa con tipos que las cumplan)

```
const int MAX = 40; //Límite tamaño de la agrupación, en esta implementación.
template<typename T> struct agrupacion;
//definir operaciones de agrupacion
template<typename T> void iniciar (agrupacion<T>& c);
template<typename T> bool anyadir (agrupacion<T>& c, const T& p);
template<typename T> bool borrarUltimo (agrupacion<T>& c);
template<typename T> bool esta (const agrupacion<T>& c, const T& e);
// Declaración:
template<typename T>
struct agrupacion{
    friend void iniciar<T> (agrupacion<T>& c);
    friend bool anyadir<T> (agrupacion<T>& c, const T& p);
    friend bool borrarUltimo<T> (agrupacion<T>& c);
    friend bool esta<T> (const agrupacion<T>& c, const T& e);
  private://declaración de la representación interna del tipo:
     T datos[MAX];
     int total;
}; // . . . continua agrupacion.h . . .
```


Iteradores

- Un iterador se utiliza para recorrer todos los elementos de una colección una única vez.
- 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.

Agrupación con iterador

```
espec agrupaciones
  usa booleanos
  parámetro formal
     género elem
  fpf
  género agrupación
operaciones
    . . . {otras operaciones}
    iniciarIterador: agrupación c → agrupación
    {Prepara el iterador para que el siguiente elemento a visitar sea
     el primero (situación de no haber visitado ningún elemento)}
    existeSiguiente?: agrupación c → booleano
    {Devuelve verdad si queda algún elemento por visitar, devuelve
     falso si ya se ha visitado el último elemento}
    parcial siguiente: agrupación c → elem
    {Devuelve el siguiente elemento a visitar.
     Parcial: la operación no está definida si no quedan elementos
     por visitar ( no existeSiguiente?(c)}
    parcial avanza: agrupación c → agrupación
    {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?(c)}
fespec
```

Agrupación con iterador

```
#ifndef AGRUPACION_H
#define AGRUPACION_H
//Interfaz del TAD. Pre-declaraciones:
. . . .

template<typename T> void iniciarIterador (agrupacion<T>& c);
template<typename T> bool existeSiguiente (const agrupacion<T>& c);
template<typename T> bool siguiente (agrupacion<T>& c, T& p);
//Declaración:
template<typename T>
struct agrupacion{
    . . . .
   friend void iniciarIterador<T> (agrupacion<T>& c);
   friend bool existeSiguiente<T> (const agrupacion<T>& c);
   friend bool siguiente<T> (agrupacion<T>& c, T& p);

  private:
     //campos del struct
     . . . .
};
```


Observaciones

- Especificación:

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

parcial avanza: agrupación c → agrupación
{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?(c)}
```

- Implementación C++: ambas operaciones juntas

```
bool siguiente (agrupacion<T>& c, T& sig);
{Si existe algún elemento pendiente de visitar, modifica sig
con el siguiente elemento a visitar, y además después avanza
el iterador para que a continuación se pueda visitar otro
elemento, y devuelve true. Si no quedaban elementos
pendientes por visitar, devuelve false.}
```


Ejemplo de uso del iterador

```
agrupacion<producto> micompra;
iniciar(micompra);
//crear productos y añadirlos a agrupación …
. . . .
//Recorrer todos los productos de la agrupación:
producto p; bool ok;
iniciarIterador(micompra);
while (existeSiguiente(micompra)) {
    ok = siguiente(micompra, p);
   //tratar el elemento siguiente obtenido en p,
   //o tratar el error
}
```


Observaciones iteradores

- Nunca se debe modificar la colección de datos mientras se recorre con las operaciones de un iterador

```
//a partir de aquí no se modifica micompra
iniciarIterador(micompra);
while (existeSiguiente(micompra)) {
    ok= siguiente(micompra, p);
   //tratar el siguiente elemento p ...
   //NO deben añadirse, modificar o borrar elementos
   // a micompra
   . . . .
} //fin del recorrido
//ya se puede modificar
```