FACULTAD DE INFORMÁTICA

Una lista de objetos basada en array

TEMA

Programación orientada a objetos — Unidad 5

Autor: Luis Hernández Yáñez

FdI UCM

Contenido

FdI UCM

Arrays de objetos

Ahora que ya sabemos crear objetos y usar arrays, podemos combinar ambas cosas para crear listas de objetos basadas en arrays.

Una lista es una secuencia de elementos. Los arrays nos permiten mantener secuencias de elementos. Por tanto, podemos usar un array para implementar una estructura lista.

Ya vimos en la unidad anterior una lista de estructuras.
Ahora vamos a desarrollar una lista de objetos.

Básicamente lo que necesitamos es un array de objetos.
¿Qué tipo de objetos? Podemos usar la clase **Persona** desarrollada en la unidad anterior y crear una lista de **Persona**s.

Declararemos el array de la siguiente forma:

```
Persona _array[MAX];
```

MAX será una constante con el número máximo de objetos que se podrán guardar en el array. Podemos declararla con un enumerado.

FdI UCM

La clase **Lista**

La lista es una estructura de datos y como tal es un elemento de programa independiente que debe estar implementado en una clase.

El array de objetos será un atributo de la clase **Lista**.

Como en el array no tienen por qué estar todas las posiciones ocupadas por objetos que se hayan insertado en la lista, debemos saber cuántas posiciones están ocupadas en cada momento (y cuáles).

Lo mejor es mantener todos los objetos en posiciones contiguas, a partir de la primera (la 0), de forma que lo único que necesitamos saber es cuántos hay. Bastará un contador de elementos.

El contador de elementos será otro atributo de la clase **Lista**.

El contador de elementos indicará la primera posición que está disponible. Lo inicializaremos a cero y no cabrán más objetos en la lista cuando valga **MAX**.

La clase **Lista**

Atributos de la clase **Lista**

No necesitamos más datos, por lo que la lista de atributos será:

```
class Lista {
...
private:
  enum { MAX = 100 };
  Persona _array[MAX];
  int _cont;
};
```

MAX no es un atributo, es una declaración de identificador que equivale al valor que se le asigna (100 en este caso), de forma que cualquier referencia a ese identificador se sustituye por su valor.

La clase **Lista**

Métodos de la clase **Lista**

La clase se ajustará a la FCO, por lo que tendrá definidos constructor predeterminado, constructor de copia, operador de asignación y destructor.

Como la lista es una *máquina de datos*, no se debe tener acceso a sus *interioridades* (atributos), sino que se utilizará exclusivamente a través de las operaciones definidas. No sólo no tiene utilidad conocer los valores de los atributos (el contador y el array), sino que se puede hacer que la *máquina* falle modificando los atributos (*¿el contador indicando más allá de la primera posición vacía?*).

Por tanto, no definiremos en la clase **Lista** ni accedentes ni mutadores.

El resto de los métodos se corresponderán con operaciones específicas de la lista como estructura de datos.

La clase **Lista**

Como se trata de un ejemplo de lista, no vamos a implementar un conjunto completo de operaciones, sino tan sólo algunas que nos permitan probar suficientemente la lista.

- ✓ ¿Caben más objetos?
  El método **llena()** devolverá **true** si no caben más objetos y **false** en caso contrario.
- ✓ ¿Hay algún objeto que se pueda recuperar?
  El método **vacia()** devolverá **true** si no hay objetos y **false** en caso contrario.
- ✓ Nuevo elemento
  El método **insertar()** insertará el objeto que acepta en la lista.
- ✓ Obtener elemento
  El método **recuperar()** devolverá el objeto dada una posición de la lista. Aunque la primera posición del array es la 0, para este método se indicará la primera posición de la lista como la 1.
- ✓ Visualización
  El método **mostrar()** mostrará todos los objetos de la lista.

La clase **Lista**

Métodos que indican su éxito o fracaso

A menudo implementaremos, como métodos, operaciones que pueden fallar. Por ejemplo, si la lista está llena, el método **insertar()** no podrá insertar el objeto en la lista tal como se le solicita. Y si al método **recuperar()** se le pasa una posición en la que no hay objeto también fallará, ya que no habrá objeto que devolver.

Cuando una operación puede fallar, el método debe indicar el éxito o el fallo de la operación. ¿Cómo? Por ejemplo, devolviendo un valor **bool** que lo indique (**true** si éxito y **false** si fallo).

Y lo más sencillo es que se devuelva el valor **bool** como resultado de la función miembro.

Pero, ¿y si hace falta que el resultado de la función sea otro más específico? Si así fuera, el valor **bool** se podría devolver como parámetro por referencia, pero normalmente puede ser el resultado.

La clase **Lista**

El método **recuperar()**

Como resultado de la función no se debe devolver nunca un valor que pueda no existir. Por ejemplo, el método **recuperar()** no debe establecer el objeto a devolver como el resultado de la función:

```
Persona recuperar(int); // posible prototipo
```

Si se hace eso, ¿qué se devuelve en el caso de que se proporcione una posición no válida? No hay objeto que devolver, pero la función está obligada a devolver un objeto **Persona**.

La única posibilidad sería devolver un objeto **Persona** *especial* que sirviera como **Persona** nula. No haremos algo así de extraño.

El objeto se devolverá, si se tiene éxito, por medio de un parámetro por referencia. Y así, nos *queda libre* el resultado de la función para el valor que indique el éxito o el fallo de la operación:

```
bool recuperar(int, Persona&); // prototipo definitivo
```


La clase **Lista**

Con el método **recuperar()** implementado de esa forma, se debe consultar el valor que devuelve antes de intentar acceder al objeto **Persona** que devuelve. A no ser que se esté absolutamente seguro de que la posición es correcta (*no hay que asumir riesgos*).

C++ permite invocar las funciones como si fueran procedimientos de Pascal, descartando el resultado que devuelven. Sin embargo, insisto en que sólo se debe hacer cuando se esté realmente seguro del éxito.

El método **insertar()**

El método **insertar()** no devuelve nada, aparte de la indicación de éxito o fallo (que será el resultado como en el método anterior). Tan sólo acepta el objeto a insertar:

```
bool recuperar(Persona); // prototipo
```


¿Cómo comunicar el fallo de un método?

Los métodos son servicios que se ponen a disposición de los programadores para manipular los objetos de las clases en las que están definidos. En C++ se trata de funciones miembro.

Cuando un método no puede realizar el cometido que se le ha encomendado[1], debe indicarlo de alguna forma, ya que si no, se puede intentar utilizar sus resultados sin que sean correctos, lo que seguramente llevará al programa a estrellarse.

Acabamos de ver una buena forma de comunicar el fallo (o el éxito) de un método: por medio de un resultado **bool**.

Sin embargo, a menudo se tiende a indicar el fallo de una función mediante un mensaje dirigido a la pantalla que indica, con palabras, el fallo que se ha producido. Esta es una elección equivocada.

[1] Si el método está implementado correctamente, no fallará por su *culpa*, sino porque se le proporcionan datos incorrectos u otras circunstancias que escapan a su control. Más adelante veremos el manejo de excepciones, el mecanismo de C++ para hacer frente a los errores de ejecución.

¿Cómo comunicar el fallo de un método?

El fallo de un método se debe comunicar al programador y no al usuario. Por esto, mostrar un mensaje en la pantalla cuando falla el método, como en el siguiente:

```
void Lista::insertar(Persona p)
{
  if(_cont == MAX) cout << "Imposible insertar";
  else {
    _array[_cont] = p;
    _cont++;
  }
}
```

hace que aparezca el mensaje siempre que el método falle. Será el usuario el que verá el mensaje sin posibilidad de emprender ninguna acción mas que intentar realizar la operación de otra forma.

(Y si, por ejemplo, la clase se usase en un programa con interfaz gráfica, ni siquiera se vería el mensaje de ninguna forma.)

FdI UCM ¿Cómo comunicar el fallo de un método?

A menudo el programador podría emprender alguna acción correctiva si se le comunicara el fallo, evitando inconveniencias al usuario.

Por esto, hacer que la función devuelva un valor *ad hoc* que informe al programador del fallo es la mejor opción:

```
bool Lista::insertar(Persona p)
// Devuelve true si se inserta; false si lista llena
{
  if(_cont == MAX) return false;
  _array[_cont] = p;
  _cont++;
  return true;
}
```

Comprobando ese valor, el programador sabrá si el método ha tenido éxito o no, y en caso de que haya fallado decidirá si informa al usuario o emprende alguna otra acción:

```
if(miLista.insertar(unaPersona)) ... else ...
```


FdI UCM Implementación de la clase **Lista**

```
#include <iostream>
#include <string>
using namespace std;

#include "persona.h" // Inclusión de la clase Persona

class Lista {
public:
  Lista() : _cont(0) {}
  Lista(const Lista&);
  Lista& operator=(const Lista&);
  ~Lista() {}
  bool llena() const;
  bool vacia() const;
  bool insertar(Persona);
  bool recuperar(int, Persona&);
  void mostrar() const;
```

(*continúa*)

FdI UCM Implementación de la clase **Lista**

```
private:
  enum { MAX = 100 };
  Persona _array[MAX];
  int _cont;
}; // Fin de la definición de la clase Lista

Lista::Lista(const Lista& otra) {
  _cont = otra._cont;
  for(int i = 0; i < _cont; i++)
    _array[i] = otra._array[i];
}

Lista& Lista::operator=(const Lista& otra) {
  _cont = otra._cont;
  for(int i = 0; i < _cont; i++)
    _array[i] = otra._array[i];
  return *this;
}
```

(*continúa*)

FdI UCM Implementación de la clase **Lista**

```
bool Lista::llena() const { return _cont == MAX; }

bool Lista::vacia() const { return _cont == 0; }

bool Lista::insertar(Persona p) {
// Devuelve true si se inserta; false si lista llena
  if(_cont == MAX) return false;
  _array[_cont] = p;
  _cont++;
  return true;
}

bool Lista::recuperar(int pos, Persona& p) {
// true si tiene éxito; false si posición no válida
  if(pos < 1 || pos > _cont) return false;
  p = _array[pos-1];
  return true;
}
```

(*continúa*)

Implementación de la clase **Lista**

```
void Lista::mostrar() const {
  cout << "Elementos de la lista:" << endl;
  for(int i = 0; i < _cont; i++)
    _array[i].mostrar();
}

// Prueba de la lista de personas
int main() {
  Persona persona;
  Lista miLista;
  int op, pos;

  do {
    cout << "1 - Insertar" << endl;
    cout << "2 - Obtener" << endl;
    cout << "3 - Listar" << endl;
    cout << "0 - Salir" << endl;
    cout << "Opción: "; cin >> op;
```

(*continúa*)

Implementación de la clase **Lista**

```
switch(op) {
  case 1:
    if(miLista.llena())
      cout << "La lista ya está llena" << endl;
   else {
      persona.leer();
      miLista.insertar(persona);
    }
    break;
  case 2:
    cout << "Posición: ";
    cin >> pos;
    if(miLista.recuperar(pos,persona))
      persona.mostrar();
    else
      cout << "No hay elemento en esa posición"
           << endl;
    break;
```

(*continúa*)

Implementación de la clase **Lista**

```
      case 3:
        miLista.mostrar();
        break;
      } // fin del switch
  } while(op != 0);

  return 0;
}
```

Cada clase en su módulo

Como ya venimos haciendo, colocamos el código de cada clase en un módulo aparte que pueda ser utilizado en cualquier programa.

Cada clase ya terminada la convertimos en un módulo de biblioteca; cada módulo de biblioteca se compone de dos archivos:

- ✓ Cabecera: la interfaz, dada básicamente por la estructura **`class`**. Archivo: ***`nombre`*`.h`**
- ✓ Implementación: resto del código (métodos implementados). Archivo: ***`nombre`*`.cpp`**

Uso del módulo de clase:

```
#include "nombre.h"
```

Dado que nuestros módulos de clases los tendremos en la misma carpeta que el programa que los use, debemos colocar comillas, no ángulos, encerrando el nombre del archivo de cabecera.

El manual del compilador explica dónde busca las bibliotecas si se encierra el nombre de la cabecera entre ángulos y si encierra entre comillas.

La clase **Lista** (de **Persona**s) como módulo de biblioteca

```
#ifndef listaper_h // Evitar inclusiones múltiples
#define listaper_h
#include "persona.h"

class Lista {
public:
  Lista();
  Lista(const Lista&);
  Lista& operator=(const Lista&);
  ~Lista() {}
  bool llena() const;
  bool vacia() const;
  bool insertar(Persona);
  bool recuperar(int, Persona&) const;
  void mostrar() const;
private:
  enum { MAX = 100 };
  Persona _array[MAX];
  int _cont;
}; // Fin de la definición de la clase Lista

#endif
```

Implementamos todos los métodos en el archivo de implementación (.cpp)

>>> listaper.h

La clase **Lista** (de **Persona**s) como módulo de biblioteca

```
// Clase Lista (de personas) - Implementación "listaper.cpp"

#include <iostream>
#include <string>
using namespace std;

#include "listaper.h" // Inclusión de la cabecera

Lista::Lista() : _cont(0) {}

Lista::Lista(const Lista& otra) {
  _cont = otra._cont;
  for(int i = 0; i < _cont; i++)
    _array[i] = otra._array[i];
}

Lista& Lista::operator=(const Lista& otra) {
  _cont = otra._cont;
  for(int i = 0; i < _cont; i++)
    _array[i] = otra._array[i];
  return *this;
}
```

(*continúa*)

La clase **Lista** (de **Persona**s) como módulo de biblioteca

```
bool Lista::llena() const { return _cont == MAX; }

bool Lista::vacia() const { return _cont == 0; }

bool Lista::insertar(Persona p) {
// Devuelve true si se inserta; false si lista llena
  if(_cont == MAX) return false;
  _array[_cont] = p;
  _cont++;
  return true;
}

bool Lista::recuperar(int pos, Persona& p) const {
// true si tiene éxito; false si posición no válida
  if(pos < 1 || pos > _cont) return false;
  p = _array[pos-1];
  return true;
}
```

(*continúa*)

La clase **Lista** (de **Persona**s) como módulo de biblioteca

```
void Lista::mostrar() const {
  cout << "Elementos de la lista:" << endl;
  for(int i = 0; i < _cont; i++)
    _array[i].mostrar();
}
```

¿Y si queremos una lista de empleados?

¿Habrá que cambiar muchas cosas?

Prueba a crear una clase **Lista** de **Empleado**s a partir de la de **Persona**s (archivos **listaemp.h** y **listaemp.cpp**)

¿Cuánto tiempo te ha llevado?

>>> listaper.cpp