Resolución ejercicio 8, guía I

Materia: Algoritmos y Programación II (75.04/95.12)
Autor: Patricio Tula
Revisión: Ing. Leandro Santi

1. Enunciado

Implementar una clase *arreglo*, con las operaciones necesarias para poder ejecutar este programa:

```
1 int main()
2 {
3     arreglo a(7);
4     a[0] = 3;
5     a[1] = 6;
6     a[2] = a[0] + a[1];
7     arreglo b = a;
8     std::cout << b[2] << std::endl;
9 }
```

2. Análisis del problema y especificaciones de diseño

Previo a lanzarnos en la resolución automática del ejercicio haremos un minucioso análisis del problema planteado.

2.1. Análisis línea por línea

- **3er Línea:** Se crea el objeto 'a' de la clase *arreglo* y aparentemente de tamaño 7. El 7 entre paréntesis podría ser otra cosa, sin embargo, asumiremos que esto es el tamaño, ó sea, la cantidad de elementos que podrá contener 'a'.
- **4a Línea:** Al objeto 'a' se le asigna para almacenar en su primer posición el valor del entero 3.
- **5a Línea:** De nuevo, al objeto 'a' se le asigna en su segunda posición el valor del entero 6 para almacenar.
- **6a Línea:** En la tercer posición de 'a' se asigna la suma de los enteros almacenados en la primer y segunda posición del objeto en cuestión. Ó sea, la tercer posición almacenará $6 + 3 = 9$.
- **7a Línea:** Se crea el objeto clase arreglo 'b' copiándole el contenido del objeto 'a'. Por lo tanto 'b' será una copia de 'a'.
- **8a Línea:** Se imprime por pantalla el valor del contenido de la tercer posición del objeto 'b'. Ó sea, se imprime por la salida estándar el número 9.

En la séptima línea surge una pregunta evidente: *¿Se está creando el objeto y luego asignándole una copia en dos operaciones distintas? ¿Ó bien es todo una sola operación y se está copiando y creando el objeto 'b' simultáneamente?*

La respuesta no es evidente al principio y se trata de una característica de funcionamiento del lenguaje de programación C++. Concretamente lo que sucede es en la función main se está creando una copia de 'a' llamada 'b' en una sola operación y no en dos. El programa llama únicamente al constructor copia de la clase, en vez de llamar al constructor por defecto y luego la operación asignación, como a uno podría ocurrirsele. Por ende, no será necesario diseñar un método de sobrecarga del operador igual.

Ya sabemos lo que hace el programa, ahora necesitamos saber que vamos a requerir para que este funcione apropiadamente.

2.2. Especificaciones de diseño

A partir de los análisis realizados podemos sacar conclusiones e interpretaciones sobre los requerimientos que tiene el programa para funcionar. Tendremos que definir dos aspectos del modelo. El primero es el manejo de datos y el segundo es el comportamiento de la estructura.

En cuanto al manejo de datos tendremos las siguientes consideraciones:

- Primero que nada la estructura debe almacenar la información pretendida. Ó sea, debe ser capaz de almacenar elementos indexados por posición.
- Propondremos que los elementos almacenado sean números enteros, puesto que el programa del ejercicio no especifica su naturaleza. Podrían ser enteros o naturales o racionales o irracionales.

Es sencillo darse cuenta los atributos que vamos a otorgarle a la clase *arreglo*. Uno será un arreglo dinámico de enteros (*int* a_*) y el otro será su correspondiente tamaño (*size_*), ó sea, la cantidad de elemento que será capaz de almacenar .

Por otro lado del diseño, tenemos las siguientes premisas del funcionamiento y comportamiento de la estructura:

- Esta estructura deberá poseer un constructor parametrizable que pueda crear un objeto *arreglo* de tamaño arbitrario.
- Deberá contar con un operador indexación ([ ], corchetes) capaz tanto para asignar información a la estructura como también para pedir información sobre los elementos de la misma sin modificar su estado.
- Otro funcionalidad que debe poseer la estructura es la de poder crear una copia de un objeto *arreglo* determinado.

3. Implementación

3.1. Clase arreglo, prototipos

```
1  class arreglo
2  {
3      private:
4          int *a_;
5          size_t size_;
6      public:
7          arreglo();
8          arreglo(const size_t);
9          arreglo(const arreglo &);
10         ~arreglo();
11         int const &operator[](size_t)const;
12         int &operator[](size_t);
13         arreglo const &operator=(const arreglo&);
14         bool empty()const;
15 };
```

Podemos ver en código en C++ todas las contemplaciones que tuvimos en cuenta previamente en las especificaciones de diseño del modelo.

La clase que diseñamos cuenta con atributos privados y métodos con públicos. Los atributos son solamente un arreglo dinámico de enteros (*int*) y un número natural de amplia representación que almacenará el tamaño de este arreglo. Por convención y para darle mayor claridad a nuestro código hemos agregado un guión bajo al final del nombre de los mismos.

El tipo de dato elegido para representar el tamaño del arreglo fue *size_t* puesto que por definición este tiene el tamaño suficiente para barrer todo el espacio de memoria de la computadora. Por ejemplo, en una arquitectura de 'LP64' este tipo de dato posee 64 bits de representación.

En el siguiente link [1] podrán obtener mayor información acerca de modelos de datos para arquitecturas de 64 bits.

Por otro lado los métodos de la clase son los siguiente: un constructor por defecto, un constructor parametrizado, un constructor copia, un destructor, dos sobrecargas sobre el mismo operador, una sobrecarga sobre el operador igual y un método boleando para determinar si la estructura está vacía.

Hemos optado por implementar un método de sobrecarga del operador igual para darle clase mayor robustez, pero este tendrá una implementación alternativa, dado que el programa del ejercicio no lo requiere implementado.

3.2. Constructores y destructor

A continuación expondremos el código de los métodos de construcción y destrucción de la clase *arreglo*.

```
1  arreglo::arreglo():a_(0), size_(0)
2  {}
3  arreglo::arreglo(const size_t size):size_(size)
4  {
5      a_ = new int[size];
6  }
7  arreglo::arreglo(const arreglo & orig)
8  {
9      if(!orig.empty())
10     {
11         size_ = orig.size_;
12         a_ = new int[size_];
13         for(unsigned int i=0; i < size_; i++)
14             a_[i] = orig.a_[i];
15     }
16 }
17 arreglo::~arreglo()
18 {
19     delete []a_;
20 }
```

Lo primero que nos surge explicar en cuanto al segmento de código que acabamos de arrojar es porqué los métodos tiene a su izquierda un *arreglo::*. Este sufijo que se le agrega a los métodos de la clase mediante al operador :: (doble dos puntos), llamado *resolución de ámbito* (*scope resolution*), indica que este pertenece a la clase en cuestión y a su ámbito. Por ende, puede manipular sus atributos y métodos libremente.

La estructura del constructor por defecto es bastante sencilla. Únicamente, establece los valores iniciales de los atributos mediante al operador : (dos puntos). El operador : es una atribución única del los constructores y sirve para inicializar atributos. El resto de los métodos no pueden utilizarlo.

El constructor parametrizable crea dinámicamente un arreglo del tamaño ingresado por argumento. El operador *new[]* se encarga de pedir la memoria necesaria.

El constructor copia es un poco más complejo. Primero valida que la estructura a copiar no este vacía y luego dinámicamente crea una copia idéntica. El argumento de este constructor es una copia constante del objeto que se desea copiar.

Debido a la simplicidad de la clase y sus atributos, el constructor es muy elemental. Solamente se encarga de liberar la memoria pedida por el atributo *a_* mediante al operador *delete[]*.

Como regla general, en C++ hay que liberar toda la memoria que se pide. Si bien el destructor es automático y se llama cuando se cierra la última llave del ámbito donde el objeto fue creado, este no hace magia.

3.3. Sobrecarga de operadores

3.3.1. Operadores corchetes

```
1 int const & arreglo::operator[](size_t index)const
2 {
3     return a_[index];
4 }
5 int & arreglo::operator[](size_t index)
6 {
7     return a_[index];
8 }
```

Los métodos de sobrecarga de esta clase parecerían ser indistinguibles. A simple vista cualquiera podría pensar que hacen lo mismo.

Sin embargo, el primero devuelve una copia **constante** del elemento almacenado en la posición indicada por el índice *index*. Este método cumple el rol de getter brindando una copia inmodificable del estado de elemento del objeto.

Por otro lado, el segundo devuelve una copia del elemento almacenado en la posición pasada por argumento entre corchetes. Este elemento del objeto podrá ser modificado y el rol de este método será el de un setter.

En las líneas cuatro, cinco y seis vemos como el programa utiliza al segundo método de sobrecarga y en la línea ocho por otro lado el programa utiliza al segundo método de sobrecarga.

La siguiente disyuntiva de diseño vinculada con el constructor por defecto queda abierta respecto de los operadores corchete: *El constructor por defecto pone todo en cero, pero crea un objeto que es válido. Si a ese objeto se le aplicase el operación [], ¿qué sucedería? ¿Sería deseable darle una vuelta de rosca más? ¿Este diseño es completamente robusto?*

3.3.2. Operador igual

```
1 arreglo const &arreglo::operator=(const arreglo &r)
2 {
3     std::cerr << "arreglo::operator=() no implementado" << std::endl
          ;
4     abort();
5     return *this;
6 }
```

Más allá que la resolución del ejercicio no requiera la implementación del método de sobrecarga del operador igual, nosotros hemos agregado uno de todas maneras con el objetivo de hacer más robusto al programa y no incurrir en un error típico de programación.

El código propuesto no funciona como un operador igual debería, este simplemente imprime por el flujo de error (*cerr*) un mensaje de error en caso de ser invocado.

Cabe aclarar que en el lenguaje C++ el operador igual puede llamarse implícitamente, sin que el usuario lo especifique (esto es así por compatibilidad con C) y puesto que la clase *arreglo* es dinámica una buena práctica de programación es afirma lo de esta manera.

4. Código fuente completo

```
1  /**************************************************
2  *Materia: Algoritmos y programacion ii(75.04/95.12) *
3  *GUIA I                                            *
4  *Ejercicio 8                                       *
5  *Autor: Patricio Tula                              *
6  *Revision: Ing. Leandro Santi                      *
7  **************************************************/
8  #include <iostream>
9  #include <stdlib.h>
10 class arreglo
11 {
12     private:
13         int *a_;
14         size_t size_;
15     public:
16         arreglo();
17         arreglo(const size_t);
18         arreglo(const arreglo &);
19         ~arreglo();
20         int const &operator[](size_t)const;
21         int &operator[](size_t);
22         arreglo const &operator=(const arreglo&);
23         bool empty()const;
24 };
25 arreglo::arreglo():a_(0), size_(0)
26 {}
27 arreglo::arreglo(const size_t size):size_(size)
28 {
29     a_ = new int[size];
30 }
31 arreglo::arreglo(const arreglo & orig)
32 {
33     if(!orig.empty())
34     {
35         size_ = orig.size_;
36         a_ = new int[size_];
37         for(unsigned int i=0; i < size_; i++)
38             a_[i] = orig.a_[i];
39     }
40 }
41 arreglo::~arreglo()
42 {
43     delete []a_;
44 }
45 bool arreglo::empty()const
46 {
47     if(!a_)
48         return true;
49     return false;
50 }
```

```
51 int const & arreglo::operator[](size_t index)const
52 {
53     return a_[index];
54 }
55 int & arreglo::operator[](size_t index)
56 {
57     return a_[index];
58 }
59 arreglo const &arreglo::operator=(const arreglo &r)
60 {
61     std::cerr << "arreglo::operator=() no implementado" << std::endl
       ;
62     abort();
63     return *this;
64 }
65
66 int main()
67 {
68     arreglo a(7);
69     a[0] = 3;
70     a[1] = 6;
71     a[2] = a[0] + a[1];
72     arreglo b = a;
73     std::cout << b[2] << std::endl;
74     return 0;
75 }
```

5. Conclusiones

Hemos visto que a partir de un análisis exhaustivo del problema pudimos obtener una solución adecuada y correcta. ¿Esta es la única solución? No, lo más probable es que hayan muchas soluciones alternativas que cumplan con lo pedido. En esta materia nos vamos a topar muchas veces con esta disyuntiva (muchas soluciones para el mismo problema). Sin embargo, lo más importante será tomar siempre la misma actitud que tomamos al principio de este ejercicio, y antes que nada analizar detalladamente el problema, luego interpreta que es lo que tenemos que hacer y luego llevar a cabo las decisiones pertinentes para implementarlo.

6. Bonus track

¿Que pasaría si el programa fuera el siguiente? ¿Como cambiaría el diseño de la estructura de dato? ¿Porqué?

```
1  int main()
2  {
3      arreglo a(7);
4      a[0] = 3;
5      a[1] = 6;
6      a[2] = a[0] + a[1];
7      arreglo b;
8      b = a;
9      std::cout << b[2] << std::endl;
10 }
```

Rta.: Habría que diseñar un método de sobrecarga para el operador igual de la clase, dado que en este caso, el programa no estaría utilizando el constructor copia.

```
1  arreglo const & arreglo::operator=(arreglo & orig)
2  {
3      if(this != &orig)
4      {
5          a_ = new int[orig.size_];
6          size_ = orig.size_;
7          for(size_t i=0; i < size_ ; i++)
8              a_[i] = orig.a_[i];
9      }
10     return *this;
11 }
```