Práctica 5. Creación de la clase `Coleccion` con operadores sobrecargados

1. Introducción

Las colecciones o arrays de elementos basados en punteros tienen varios problemas. Por ejemplo, un programa puede *salirse* fácilmente de cualquier extremo de un array, ya que C++ no comprueba si los subíndices están fuera del rango de este (aunque sí se puede hacer esto explícitamente). Los arrays de tamaño n deben enumerar sus elementos así: $0, ..., n-1$; no se permiten —por ejemplo— rangos de subíndices alternados. Un array no puede recibirse ni enviarse todo a la vez; se debe leer o escribir cada elemento de este de manera individual.

Dos arrays no pueden compararse significativamente con los operadores de igualdad o relacionales debido a que los nombres de los arrays son simplemente punteros a la dirección en la que empiezan estos en memoria y, desde luego, dos arrays siempre estarán en distintas ubicaciones de memoria. Cuando se pasa un array a una función de propósito especial diseñada para manejar arrays de cualquier tamaño, el tamaño del array debe pasarse como argumento adicional. Un array no puede asignarse a otro con el(los) operador(es) de asignación debido a que los nombres de los arrays son punteros `const`, y un puntero constante no se puede utilizar del lado izquierdo de un operador de asignación.

Como cabría pensar, éstas herramientas que describimos —entre otras— deberían ser, sin duda, la opción *natural* para tratar con los arrays, pero los arrays basados en punteros no proporcionan dichas herramientas. Sin embargo, C++ proporciona los medios para implementar dichas herramientas de los arrays a través del uso de las clases y la sobrecarga de operadores.

2. Descripción

En esta práctica, vamos a crear una clase tipo array de enteros que ofrezca todas las funcionalidades que se han descrito. Esta clase realizará la comprobación de rangos para asegurar que los subíndices permanecen dentro de los límites del objeto `Coleccion`. Tamb ién debe permitir asignar un objeto `Coleccion` a otro mediante el operador de asignación. Además, los objetos de la clase `Coleccion` deben conocer su tamaño, por lo que éste no se necesitará pasar —en ningún momento— como argumento al pasar un objeto `Coleccion` a una función. Por otro lado, deberán poderse enviar o recibir objetos `Coleccion` completos mediante los operadores de flujo. Y, por último, también se podrán realizar comparaciones entre objetos `Coleccion` mediante los operadores de igualdad: == y ! =.

Para cumplir con todo lo anterior, cada objeto `Coleccion` contendrá dos atributos: `size`, que indicará el número de elementos en el objeto `Coleccion`, y un puntero `int (ptr)` que apunta a la colección de enteros. Además de esta consideración, veremos a continuación algunos otros detalles de implementación.

2.1. Sobrecarga de operadores de flujo como funciones `friend`

El operador de inserción de flujo sobrecargado y el operador de extracción de flujo sobrecargado se deben declarar como funciones `friend` de la clase `Coleccion`. Así, cuando el compilador vea una expresión del tipo `cout << objetoColeccion`, invocará a la función `operator<<` con la llamada `operator<< (cout, objetoColeccion)`. Del mismo modo, cuando el compilador vea una expresión del tipo `cin >> objetoColeccion`, este invocará a la función `operator>>` con la llamada `operator>> (cin, objetoColeccion)`.

Recuerda que estas funciones de operador de inserción de flujo y operador de extracción de flujo no pueden ser miembros de la clase `Coleccion`, ya que el objeto `Coleccion` siempre se menciona del lado derecho del operador de inserción de flujo y del operador de extracción de flujo. Si estas funciones de operador fuesen miembros de la clase `Coleccion`, tendrían que utilizarse las siguientes instrucciones para enviar y recibir un objeto Array, cuya sintaxis resultaría muy confusa para cualquier programador de C++: `objetoColeccion << cout;` y `objetoColeccion >> cin;`.

La función `operator<<` deberá imprimir el número de elementos indicados por `size` a partir de la colección de enteros a la que apunta `ptr`. La función `operator>>` introduce los datos directamente en el array al que apunta `ptr`. Cada una de estas funciones de operador devuelve una referencia apropiada para permitir instrucciones de salida o entrada en casca-

da, respectivamente.

Estas funciones, al ser declaradas como funciones `friend`, tienen acceso a los datos private de un objeto `Coleccion`. Y también pueden acceder a las funciones `getSize()` y `operator[]`, aunque para esto no haría falta que fuesen funciones `friend`.

2.2. Constructor predeterminado de `Coleccion`

Una buena costumbre es implementar un constructor predeterminado. En este caso, se implementará un constructor predeterminado para la clase, en el que se especificará un tamaño predeterminado de 10 elementos. Así, cuando el compilador vea una declaración del tipo `Coleccion integers1 (7);`, este invocará al constructor predeterminado de `Coleccion`, ya que el constructor predeterminado en este caso recibe un solo argumento tipo `int`, que tiene un valor predeterminado de 10.

Por simplicidad, se ha considerado que el valor pasado por parámetro será mayor que 0, pero podría ser que no, en cuyo caso habría que implementar un mecanismo de excepciones (que ya veremos más adelante) que maneje tal situación. Para la reserva de memoria de la colección (que recordemos parte de un simple puntero) se usará `new`; una vez hecha la reserva, se asigna el puntero devuelto por esta reserva al atributo `ptr`. Después, el constructor empleará una instrucción `for` para establecer todos los elementos de la colección en 0.

Sería posible tener una clase `Coleccion` que no inicializara sus atributos si, por ejemplo, estos atributos se van a leer más adelante en algún momento; pero esto se considera, por razones obvias, una mala práctica de programación. Los objetos `Coleccion` (y los objetos en general) deben inicializarse de manera apropiada y mantenerse en un estado consistente.

2.3. Constructor de copia de `Coleccion`

Además del constructor predeterminado, se implementará también un constructor de copia. Este iinicializará un objeto `Coleccion` a partir de un objeto `Coleccion` existente. Para realizar dicha copia, hay que tener la precaución de no dejar a ambos objetos `Coleccion` apuntando a la misma memoria asignada en forma dinámica. Este problema ocurriría si no implementásemos este constructor y, en su lugar, usásemos el constructor de copia predeterminado (el de por defecto) para esta clase.

Los constructores de copia se invocan cada vez que se necesita una copia de un objeto, como cuando se pasa un objeto por valor a una función, se devuelve un objeto por valor

de una función, o se inicializa un objeto con una copia de otro objeto de la misma clase. El constructor de copia, en ese caso, se llamará en una declaración cuando se instancia un objeto de la clase `Coleccion` y se inicializa con otro objeto de la clase `Coleccion`, como por ejemplo, en la instrucción de declaración `Coleccion integers3 (integers1);`.

Otros aspectos interesantes a tener en cuenta son los siguientes. Por un lado, para permitir que se copie un objeto `const`, el argumento para un constructor de copia debe ser una referencia `const` también. Por otro lado, un constructor de copia debe recibir su argumento por referencia, no por valor; en caso contrario, la llamada al constructor de copia produce recursividad infinita (un error lógico fatal), ya que para recibir un objeto por valor, el constructor de copia tiene que realizar una copia del objeto que se usa como argumento. Piensa que, cada vez que se requiere una copia de un objeto, se hace una llamada al constructor de copia de la clase; si el constructor de copia recibe su argumento por valor, ¡se llamaría a sí mismo de manera recursiva para realizar una copia de su argumento!

Y una última consideración. Si lo que hace el constructor de copia es simplemente copiar el puntero del objeto de origen al puntero del objeto de destino, entonces ambos objetos apuntarían a la misma memoria asignada en forma dinámica. De esta forma, el primer destructor en ejecutarse eliminaría la memoria asignada en forma dinámica, y el `ptr` del otro objeto quedaría indefinido. Esto probablemente produciría un grave error en tiempo de ejecución, como la terminación anticipada del programa, a la hora de utilizar este puntero.

3. Implementación

El esqueleto que debería conformar la clase `Coleccion` (`Coleccion.h`) debería ser como se muestra a continuación:

```
/* Coleccion.h
   Practical exercise 5.
   Coleccion class definition with overloaded operators.
*/

#ifndef ARRAY_H
#define ARRAY_H

#include <iostream>

class Coleccion {
  friend std::ostream &operator<< (std::ostream &, const Coleccion &);
```

```
  friend std::istream &operator>> (std::istream &, Coleccion &);

public:
  explicit Coleccion (int = 10); // default constructor
  Coleccion (const Coleccion &); // copy constructor
  ~Coleccion(); // destructor

  size_t getSize() const; // return size

  const Coleccion &operator= (const Coleccion &); // assignment operator
  bool operator== (const Coleccion &) const; // equality operator

  // inequality operator; returns opposite of == operator
  bool operator!= (const Coleccion &right) const {
    return !(*this == right); // invokes Coleccion::operator==
  } // end function operator!=

  // subscript operator for non-const objects returns modifiable lvalue
  int &operator[] (int);

  // subscript operator for const objects returns rvalue
  int operator[] (int) const;

private:
  size_t size; // pointer-based array size
  int *ptr; // pointer to first element of pointer-based array
}; // end class Coleccion

#endif
```

Por su parte, el fichero fuente `Coleccion.cpp` debería contener la implementación de todas las funciones. A continuación se muestra la implementación del constructor y destructor:

```
/* Coleccion.cpp
   Practical exercise 5.
   Coleccion class member -and friend- function definitions.
*/

#include <iostream>
#include <iomanip> // I/O manipulation library
```

```
#include "Coleccion.h" // Coleccion class definition

using namespace std;

// default constructor for class Coleccion (default size 10)
// considering arraySize > 0, otherwise an exception should be thrown
Coleccion::Coleccion (int arraySize)
  : size (arraySize),
    ptr (new int [size]) {
  for (size_t i = 0; i < size; ++i)
    ptr [i] = 0; // set pointer-based array element
} // end Coleccion default constructor

// copy constructor for class Coleccion; must receive a reference to a Coleccion
Coleccion::Coleccion (const Coleccion &arrayToCopy)
  : size (arrayToCopy.size),
  ptr (new int [size]) {
    for (size_t i = 0; i < size; ++i)
      ptr [i] = arrayToCopy.ptr [i]; // copy into object
} // end Coleccion copy constructor

// destructor for class Coleccion
Coleccion::~Coleccion() {
  delete [] ptr; // release pointer-based array space
} // end destructor
```

Por último, para probar todos los flecos de la clase `Coleccion`, necesitamos un programa principal que haga uso de las funciones que ofrece esta. A continuación se muestra la implementación de un programa `main.cpp`. Sírvase como ejemplo de batería de pruebas del programa, al cual, por supuesto, se puede añadir toda la funcionalidad que se considere oportuna.

```
/* main.cpp
   Practical exercise 5.
   Coleccion class test program.
*/
#include <iostream>
#include "Coleccion.h"

using namespace std;
```

```
int main() {
  Coleccion integers1 (7); // seven-element Coleccion
  Coleccion integers2; // 10-element Coleccion by default

  // print integers1 size and contents
  cout << "Size of Coleccion integers1 is " << integers1.getSize()
    << "\nColeccion after initialization:\n" << integers1;

  // print integers2 size and contents
  cout << "\nSize of Coleccion integers2 is "
    << integers2.getSize()
    << "\nColeccion after initialization:\n" << integers2;

  // input and print integers1 and integers2
  cout << "\nEnter 17 integers:" << endl;
  cin >> integers1 >> integers2;

  cout << "\nAfter input, the Coleccions contain:\n"
    << "integers1:\n" << integers1
    << "integers2:\n" << integers2;

  // use overloaded inequality (!=) operator
  cout << "\nEvaluating: integers1 != integers2" << endl;

  if (integers1 != integers2)
    cout << "integers1 and integers2 are not equal" << endl;

  // create Coleccion integers3 using integers1 as an
  // initializer; print size and contents
  Coleccion integers3 (integers1); // invokes copy constructor

  cout << "\nSize of Coleccion integers3 is "
    << integers3.getSize()
    << "\nColeccion after initialization:\n" << integers3;

  // use overloaded assignment (=) operator
  cout << "\nAssigning integers2 to integers1:" << endl;
  integers1 = integers2; // note target Coleccion is smaller
```

```
  cout << "integers1:\n" << integers1
    << "integers2:\n" << integers2;

  // use overloaded equality (==) operator
  cout << "\nEvaluating: integers1 == integers2" << endl;

  if (integers1 == integers2)
    cout << "integers1 and integers2 are equal" << endl;

  // use overloaded subscript operator to create rvalue
  cout << "\nintegers1[5] is " << integers1 [5];

  // use overloaded subscript operator to create lvalue
  cout << "\n\nAssigning 1000 to integers1[5]" << endl;
  integers1 [5] = 1000;
  cout << "integers1:\n" << integers1;
} // end main
```