

---

# Relación de ejercicios tema 4: Clases en C++ (ampliación) y tema 5: Sobrecarga de operadores

Metodología de la programación, 2017-2018

---

## Contenido:

1	Introducción	1
2	Clases, constructores y destructores, métodos	1
3	Clases y sobrecarga de operadores	3

---

## 1 Introducción

Los ejercicios propuestos están relacionados con los conceptos de constructor, destructor, métodos de acceso a datos miembro, modularización, etc. Los ejercicios deben implementarse de forma completa, lo que implica que para cada uno de ellos debe existir:

- Un archivo **.h** con las declaraciones.
- Un archivo **.cpp** con la implementación.
- Un archivo **makefile** para generar el ejecutable.
- Una estructura de directorios similar a la usada en las prácticas, para que todos los elementos que constituyen el programa queden organizados de forma clara.

## 2 Clases, constructores y destructores, métodos

1. Implementad la clase Racional, destinada a soportar el trabajo con números fraccionarios de la forma  $\mathbf{a/b}$ , siendo **a** y **b** números enteros. Proponed una representación para la clase e incluid los siguientes métodos:
  - (a) Constructor sin argumentos, para construir un objeto que represente al valor 0.
  - (b) Constructor para crear un racional a partir de un número entero.
  - (c) Constructor para crear un racional a partir de dos enteros: numerador y denominador.
  - (d) Métodos para devolver los valores de numerador y denominador.
  - (e) Piensa si es necesario implementar el constructor de copia y el destructor, e implementálos en caso positivo.
2. Ampliad el ejercicio 15 del tema 2, sobre la clase **VectorSD** (array dinámico de enteros). En primer lugar añadiremos los siguientes métodos:
  - (a) Constructor de copia.
  - (b) Destructor.
  - (c) Método para modificar el valor en la posición **i**. Si la posición no es correcta (no está entre 0 y el número de elementos menos 1) se lanzará una aserción.

- (d) Método para insertar un elemento en una posición determinada. Si la posición no es correcta se lanzará una aserción.
- (e) Método para borrar un elemento de una determinada posición. Si la posición no es correcta se lanzará una aserción.

Implementa también una función externa a la clase (`ordenar()`) que ordene de menor a mayor los elementos de un array dinámico de la clase anterior.

Modifica el programa `main` del ejercicio 15 del tema 2, para que se cree una copia del array leído, y luego sea ordenada con la función anterior. Fíjese que ahora no hay que llamar a la función `liberar()`:

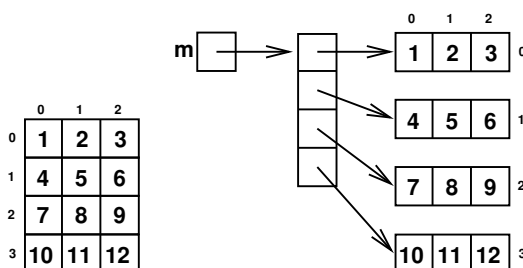
```
#include <iostream>
#include <fstream> // ifstream
using namespace std;

int main(int argc, char* argv[])
{
    VectorSD v;
    if (argc==1)
        v.leer(cin);
    else {
        ifstream flujo(argv[1]);
        if (!flujo) {
            cerr << "Error: Fichero " << argv[1] << " no válido." << endl;
            return 1;
        }
        v.leer(flujo);
    }
    VectorSD vCopia(v); // creamos una copia de v
    ordenar(vCopia); // Ordenamos la copia

    cout << "Array original:" << endl;
    v.mostrar(cout);

    cout << "\nArray copia:" << endl;
    vCopia.mostrar(cout);
}
```

### 3. Ampliad el ejercicio 17 del tema 2, sobre matrices bidimensionales dinámicas.



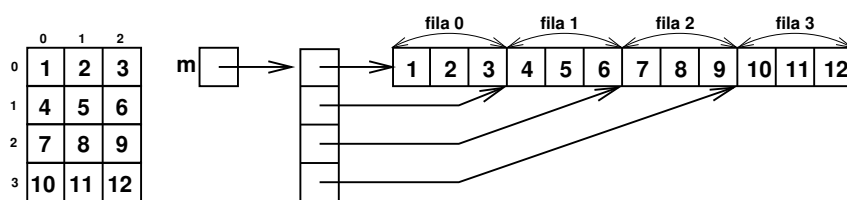
En primer lugar añadiremos los siguientes métodos:

- (a) Constructor de copia.
- (b) Destructor.

Implementa también una función externa a la clase (`buscarMaximo()`) que devuelva la fila y columna donde está el valor máximo de la matriz.

Haz un programa que lea una matriz y escriba la fila y columna donde está el valor máximo, así como tal valor máximo.

- 4. Igual que en el ejercicio anterior, pero con otra estructura interna para almacenar los valores de la matriz (en este caso la clase será `Matriz2D_2`), tal y como se indica a continuación:



5. Implementad la clase **Lista** para trabajar con listas dinámicas (de tamaño arbitrario, y no definido a priori) de datos de tipo **TipoBase**. Cada nodo de la **Lista** estará enlazado con el siguiente. Se debe aportar la siguiente funcionalidad:
  - (a) Constructor sin argumentos para crear una lista vacía.
  - (b) Constructor de copia.
  - (c) Destructor.
  - (d) Método para devolver el número de elementos en la lista.
  - (e) Método para devolver el elemento de una posición dada.
  - (f) Método para modificar el elemento de una posición dada.
  - (g) Método para insertar un elemento al principio.
  - (h) Método para insertar un elemento en una posición dada.
  - (i) Método para insertar un elemento al final.
  - (j) Método para borrar un elemento en una posición dada.
6. Implementad la clase **Pila**: estructura de datos donde los elementos se acceden siguiendo una política **LIFO** (last in, first out). La clase debe proporcionar la siguiente funcionalidad:
  - (a) Constructor sin argumentos, creando una pila vacía.
  - (b) Constructor de copia.
  - (c) Destructor.
  - (d) Método para añadir un valor a la pila.
  - (e) Método para extraer un valor de la pila (consultará el valor del elemento ubicado en la cima y lo elimina de la pila).
  - (f) Método para consultar si la pila está vacía.
  - (g) Método para consultar el valor del elemento ubicado en la cima de la pila sin borrarlo.
7. Igual que en el ejercicio anterior, pero en este caso para la clase **Cola** (la política de acceso es ahora **FIFO** (first in, first out)).

### 3 Clases y sobrecarga de operadores

Los ejercicios propuestos están relacionados con los anteriores, pero centrados ahora en los conceptos de constructor de copia y sobrecarga de operadores. Se trata aquí de añadir funcionalidad adicional a los ejercicios incluidos en la primera parte de la relación.

1. Ampliad la clase **Racional** con las siguientes operaciones:
  - (a) Sobrecarga de los operadores unarios  $+$  y  $-$ .
  - (b) Sobrecarga de los operadores aritméticos binarios  $+$ ,  $-$ ,  $*$ ,  $/$ , con el objeto de poder operar entre dos racionales y racionales con enteros (en cualquier orden).
  - (c) Sobrecarga de los operadores aritméticos binarios  $+=$ ,  $-=$ ,  $*=$  y  $/=$ .
  - (d) Sobrecarga de los operadores relacionales binarios  $==$ ,  $!=$ ,  $<$ ,  $>$ ,  $>=$  y  $<=$  para poder comparar racionales con racionales y racionales con enteros en cualquier orden.

- (e) Sobrecarga de los operadores `<<` y `>>` para insertar un número racional en un flujo y extraer un número racional de un flujo. La inserción/extracción se realiza de la siguiente forma: sea **r** un dato de la clase **Racional**:

- Si **r** contiene el valor  $3/5$  entonces **cout** mostrará  $3/5$ .
- La ejecución de `cin >> r` hará que se lea una cadena de caracteres y se procese adecuadamente para separar numerador y denominador.

2. Ampliad la clase **VectoSD** del ejercicio 2 de la sección anterior, agregando:

- (a) Sobrecarga del operador de asignación (empleando código reutilizable).
- (b) Reescribid el destructor empleando código reutilizable.
- (c) Sobrecarga del operador `[]` para que sirva de operador de acceso a los elementos del vector, de forma que pueda actuar tanto como **lvalue** como **rvalue**.
- (d) Sobrecarga de los operadores relaciones binarios `==` y `!=` para comparar dos vectores dinámicos. Dos vectores serán iguales si tienen el mismo número de elementos y sus contenidos son idénticos (y ocupan las mismas posiciones).
- (e) Sobrecarga de los operadores relacionales binarios `>`, `<`, `>=` y `<=` para poder comparar dos vectores. Usar un criterio similar al que se sigue en la comparación de dos cadenas de caracteres clásicas (orden lexicográfico).
- (f) Considerad una implementación nueva para redimensionar un vector: emplear los operadores binarios `+`, `-`, `+=` y `-=` de manera que, por ejemplo:
  - Si **v** es un vector, la instrucción `v = v+1` crea un nuevo vector con un elemento más, generado a partir del último sumando 1.
  - Si **v** es un vector, la instrucción `v2 = v+1` crea un vector dinámico con un elemento más que **v**, lo rellena a partir de **v** y agrega un elemento más tal y como se ha indicado en el punto anterior.
  - Si **v** es un vector, la instrucción `v -= 10` crea uno nuevo con 10 casillas menos que **v** (descarta las 10 últimas).
- (g) Sobrecarga los operadores `<<` y `>>` para leer/escribir un vector dinámico. Para la implementación del operador `>>` leerá una secuencia indefinida de valores (separados por espacios), hasta que se introduzca el valor `*`. Los valores se leerán en una cadena de caracteres, y sólo se convertirán al tipo **TipoBase** cuando se verifique que son válidos para su almacenamiento (no se ha introducido el terminador `(*)`).
- (h) Modifica el programa **main** del ejercicio 2 de la sección anterior, para hacer uso del operador de asignación y de los operadores de entrada y salida de un flujo (`>>` y `<<`).

```
#include <iostream>
#include <fstream> // ifstream
using namespace std;

int main(int argc, char* argv[])
{
    VectorSD v;
    VectorSD vCopia;

    if (argc==1)
        cin >> v;
    else {
        ifstream flujo(argv[1]);
        if (!flujo) {
            cerr << "Error: Fichero " << argv[1] << " no válido." << endl;
            return 1;
        }
        flujo >> v;
    }
    vCopia = v; // se usa operador de asignación
    ordenar(vCopia); // Ordenamos la copia

    cout << "Array original:" << endl;
    cout << v;
```

```

        cout << "\nArray copia:" << endl;
        cout << vCopia;
    }

```

3. Construye una clase **Poligono** que permitirá almacenar polígonos convexos, con un número de vértices cualquiera. Para cada vértice, usaremos la clase **Punto**:

```

class Punto{
    double x;
    double y;
public:
    Punto(){x=0; y=0;};
    Punto(int x, int y){this->x=x; this->y=y;};
    double getX(){return x;} const;
    double getY(){return y;} const;
    double setXY(int x, int y){this->x=x; this->y=y;};
};

```

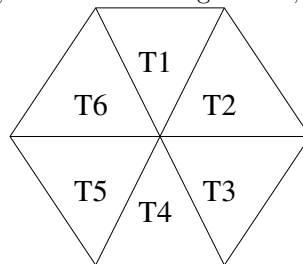
La clase **Poligono** guarda los vértices en un array dinámico de objetos **Punto** del tamaño justo para contener los vértices que tiene el polígono actualmente:

```

class Poligono{
    int nVertices;
    Punto *vertices;
public:
    Poligono();
    Poligono(int n);
    ~Poligono(); // destructor
    Poligono(const Poligono& otro); // constructor copia
    Poligono& operator=(const Poligono& otro); // op asignación
    int getNumeroVertices() const;
    Punto getVertice(int index) const;
    void addVertice(const Punto& v);
    double getPerimetro() const;
};

```

- (a) Implementa los métodos de las clases **Punto** y **Poligono** indicados en las declaraciones anteriores.
- (b) Añade el método **double getArea()** a la clase **Poligono** para calcular el área de un polígono. Para ello, tenga en cuenta que el área de un polígono puede calcularse mediante la suma de las áreas de un conjunto de triángulos que están inscritos en él. Los triángulos se obtienen uniendo los lados del polígono con un punto interior, mediante dos segmentos, tal como muestra la siguiente figura.



Para calcular el área de un triángulo, añade una función externa al módulo **Poligono**:

```
double areaTriangulo(Punto pto1, Punto pto2, Punto pto3).
```

Se usará la siguiente fórmula para calcular el área:

$$area = \sqrt{T(T - S_1)(T - S_2)(T - S_3)}$$

$$T = \frac{S_1 + S_2 + S_3}{2}$$

donde  $S_1$ ,  $S_2$  y  $S_3$  son las longitudes de los lados del triángulo.

Debe implementarse también en la clase **Poligono** el método auxiliar:

```
Punto getPuntoInterior()
```

que devuelve un punto interior cualquiera del polígono. Este método puede implementarse como el punto medio de la recta entre dos vértices opuestos o bien simplemente devolviendo uno de los vértices del polígono, ya que un vértice es también un punto interior al `Poligono`.

- (c) Sobrecarga el operador de salida (`<<`) para mostrar un objeto `Poligono` en un flujo de salida (muestra el número de vértices, y luego las coordenadas x e y de cada vértice). Por ejemplo, a continuación se muestra un polígono de 6 vértices:

```
6
4 12
10 12
14 6
10 0
4 0
0 6
```

- (d) Sobrecarga el operador de entrada (`>>`) para leer un objeto `Poligono` de un flujo de entrada (en el formato del apartado anterior).
  - (e) Haz un programa que lea un Polígono almacenado en un fichero de texto pasado al programa como argumento de `main()` (o de la entrada estándar si no se proporciona el argumento), lo muestre en la salida estándar y calcule su área y perímetro.
4. Haz un programa que lea de un fichero de texto (o entrada estándar), una lista de polígonos, cada uno de ellos especificado según el ejercicio anterior y los vaya guardando en un objeto de la clase `ArrayPoligono`, que es un array dinámico de objetos `Poligono`. Tras leer los polígonos, el programa debe ordenar el array de menor a mayor según el área, y mostrar luego el área y perímetro de cada uno de ellos.

La clase `ArrayPoligono` a utilizar tendrá la siguiente forma:

```
class ArrayPoligono{
    int reservados; // Tamaño reservado actualmente en el array
    int nPoligonos; // Número de polígonos actualmente en el array
    Poligono *poligonos;
public:
    ...
}
```

Construye la clase `ArrayPoligono`, añadiendo los métodos necesarios (constructores, constructor de copia, destructor, operador de asignación, etc). El constructor por defecto reserva espacio para 4 polígonos. En el método para añadir un nuevo polígono, si el array está lleno, se realojará automáticamente un nuevo array del doble de tamaño al que tenga actualmente.

5. Ampliad la clase `Matriz2D_1` con los siguientes métodos:
- (a) Operador de asignación, empleando código reutilizable.
  - (b) Reescribe el destructor en base a la estrategia anterior.
  - (c) Sobrecarga alternativa del operador de asignación, que recibe como argumento un dato de `TipoBase` e inicia toda la matriz al valor especificado.
  - (d) Sobrecarga del operador `()` para que sirva de operador de acceso a los elementos de la matriz y pueda actuar como **lvalue** y **rvalue**.
  - (e) Sobrecarga los operadores unarios `+` y `-`.
  - (f) Sobrecarga los operadores relacionales binarios `==` y `!=` para poder comparar matrices dinámicas: para la igualdad han de contener el mismo número de filas y columnas y mismos valores en cada posición.
  - (g) Sobrecarga el operador `<<` para mostrar el contenido de la matriz.
6. Igual que en el ejercicio anterior, pero para la clase `Matriz2D_2`.
7. Ampliad la clase `Lista` (de datos `TipoBase`) con los siguientes métodos:

- (a) Operador de asignación, empleando código reutilizable. Reescribid el destructor en base a esta estrategia.
  - (b) Sobrecarga del operador `[]` para que sirva de operador de acceso a los elementos de la lista y pueda actuar tanto como **lvalue** como **rvalue**. El índice hace referencia a la posición, de tal manera que 1 indica el primer nodo, 2 el segundo, etc.
  - (c) Sobrecarga de los operadores `<<` y `>>` para leer/escribir una lista.
    - Para la implementación del operador `>>` leerá una secuencia indefinida de valores, hasta que se introduzca el valor `*`. Los valores se leerán en una cadena de caracteres, y sólo se convertirán al tipo **TipoBase** cuando se verifique que son válidos para su almacenamiento (no se ha introducido el terminador `*`).
    - El nuevo valor siempre se guardará al final.
8. Ampliad la clase **Pila** (de datos **TipoBase**) con los siguientes métodos:
- (a) Operador de asignación, mediante código reutilizable.
  - (b) Reescribe el destructor en base a la estrategia anterior.
  - (c) Sobrecarga el operador `<<`.
9. Ampliad la clase **Cola** (de datos **TipoBase**) con los siguientes métodos:
- (a) Operador de asignación, mediante código reutilizable.
  - (b) Reescribe el destructor en base a la estrategia anterior.
  - (c) Sobrecarga el operador `<<`.
10. Implementad la clase **Conjunto**, de forma que permita manipular un conjunto de elementos de tipo **TipoBase**. Para la representación interna de los elementos del conjunto, usad una lista de celdas enlazadas, que debería mantenerse ordenada para facilitar su tratamiento. Las operaciones con que cuenta son:
- (a) Constructor sin argumentos que crea un conjunto vacío.
  - (b) Constructor con un argumento, de tipo **TipoBase**: crea un conjunto con el elemento proporcionado como argumento.
  - (c) Constructor de copia (empleando código reutilizable).
  - (d) Destructor (empleando código reutilizable).
  - (e) Método para consultar si el conjunto está vacío.
  - (f) Sobrecarga del operador de asignación (empleando código reutilizable).
  - (g) Método que devuelva el número de elementos en el conjunto.
  - (h) Método para determinar si un valor **TipoBase** pertenece al conjunto.
  - (i) Sobrecarga de los operadores `>>` y `<<`.
    - Para `<<` se leerá una secuencia indefinida de valores hasta la introducción del valor `*`. Los valores se leerán en una cadena de caracteres y sólo se convierten a **TipoBase** cuando se haya comprobado su validez (no se ha introducido el terminador `*`).
    - No se permiten elementos repetidos.
  - (j) Sobrecarga de los operadores `==` y `!=` para comparar conjuntos. Dos conjuntos son iguales si tienen el mismo número de elementos y éstos son idénticos (independientemente de su posición).
  - (k) Sobrecarga del operador binario `+` para calcular la unión de dos conjuntos. La operación responde a los siguientes criterios:
    - Si  $a$  y  $b$  son conjuntos,  $a+b$  será otro dato de tipo **Conjunto** que contendrá todos los elementos de ambos, sin los elementos repetidos, o sea  $a \cup b$ .
    - Si  $a$  es un conjunto y  $v$  es un valor de **TipoBase**,  $a+v$  será otro dato de tipo **Conjunto** que contendrá  $a \cup \{v\}$ .

- Si  $a$  es un conjunto y  $v$  es un valor de **TipoBase**,  $v+a$  será otro dato de tipo **Conjunto** que contendrá  $\{v\} \cup a$ .
- (l) Sobrecarga el operador binario  $-$  para calcular la diferencia de conjuntos de forma similar al anterior:
- Si  $a$  y  $b$  son conjuntos,  $a-b$  será otro dato de tipo **Conjunto** que contendrá el resultado de quitar de  $a$  los elementos que están en  $b$ , o sea  $a - b$ .
  - Si  $a$  es un conjunto y  $v$  es un valor de **TipoBase**,  $a-v$  será otro dato de tipo **Conjunto** que contendrá  $a - \{v\}$ .
- (m) Sobrecarga el operador binario  $*$  para calcular la intersección de dos conjuntos. La operación responde a los siguientes criterios:
- Si  $a$  y  $b$  son conjuntos,  $a*b$  será otro dato de tipo **Conjunto** que contendrá  $a \cap b$ .
  - Si  $a$  es un conjunto y  $v$  es un valor de **TipoBase**,  $a*v$  será otro dato de tipo **Conjunto** que contendrá  $a \cap \{v\}$ .
  - Si  $a$  es un conjunto y  $v$  es un valor de **TipoBase**,  $v*a$  será otro dato de tipo **Conjunto** que contendrá  $\{v\} \cap a$ .