

Metodología de la programación

20 de abril de 2018



Práctica 4: Clases (ampliación)

Curso 2017-2018

Índice

Objetivos	2
Ejercicio 1 (Junio 2014)	2
Ejercicio 2 (Junio 2013)	3
Ejercicio 3 (Junio 2012)	4
Ejercicio 4	5
Ejercicio 5 (Septiembre 2016)	5
Ejercicio 6 (Junio 2016)	6
Ejercicio 7 (Junio 2015)	7
Ejercicio 8 (Septiembre 2013)	8
Entrega	8

Objetivos

El objetivo de esta práctica es trabajar con los conceptos vistos en el tema de teoría de ampliación de clases y servir como punto de partida para los ejercicios relacionados con sobrecarga de operadores. Para ello debéis implementar todos los ejercicios propuestos en esta relación. Para cada uno de ellos se ofrece un conjunto de pruebas concreto que el programa debe tratar de forma correcta para que se considere la evaluación de la práctica.

Se recuerda que el trabajo en estos ejercicios debe ser personal y que se recomienda la asistencia a clase de prácticas. La copia de código no aporta nada al aprendizaje y será considerada como un incumplimiento de las normas de la asignatura con las consecuencias que ello implica, según la normativa de la Universidad de Granada.

Para todos los ejercicios debes separar el código en módulos independientes para código fuente, archivos de cabecera, archivos de código objeto y ejecutable. Debe proporcionarse un archivo **Makefile** que funcione (con directiva **clean**) y que no contenga elementos o instrucciones no relacionadas con el ejercicio correspondiente. Los errores en este archivo se penalizarán en la nota de la práctica.

En todos los ejercicios se incluirá un programa principal que incluya las sentencias necesarias para probar que todos los métodos implementados funcionan de forma correcta. Sería conveniente probar que no hay problemas de gestión dinámica de memoria mediante la herramienta **valgrind**.

Ejercicio 1 (Junio 2014)

Se desea resolver el problema del juego de los barquitos, para lo que se precisa el diseño de la clase **Barquitos** que contiene toda la información necesaria para gestionar el

tablero de un jugador. Debe incluir una matriz de enteros que permita codificar el estado del tablero, mediante el uso de memoria dinámica. En la siguiente figura se presenta una matriz de configuración y muestra indicaciones para interpretarla:

	1	2	3	4	5	6	7	8	9	10	11
A											
B		.					.				
C			×	×	×	.	
D				×			.				
E							×				
F							.				
G											
H		.			.						
I		.			.						
J			.							×	
K							.	.			
L							×	×	.		

	1	2	3	4	5	6	7	8	9	10	11
A	9	9	9	9	9	9	9	9	9	9	9
B	9	-9	1	9	9	9	-9	9	9	9	9
C	9	9	-9	-9	-9	-9	-3	-3	-3	-9	9
D	9	9	9	-4	9	9	-9	9	9	9	9
E	9	1	9	4	9	9	-1	9	9	9	9
F	9	9	9	4	9	9	-9	9	9	2	2
G	9	9	9	4	9	9	9	9	9	9	9
H	9	-9	9	9	-9	9	3	9	9	9	9
I	9	-9	9	9	-9	9	3	9	9	9	9
J	9	9	-9	9	9	9	3	9	9	-1	9
K	9	2	2	9	9	9	9	-9	-9	9	9
L	9	9	9	9	9	9	9	-2	-2	-9	9

Codificación:

Valor negativo: se ha disparado en la casilla

Valor 9: agua

Valor entero i: parte de un barco de i casillas

Ejemplos:

B3: valor 1. Barco de 1 casilla

B2: valor -9. Agua donde se ha disparado

D4: valor -4. Barco de 4 alcanzado

C8: valor -3. Barco de 3 alcanzado

Se pide:

- definir la estructura de la clase **Barquitos**
- implementar constructor que recibe como argumento el número de filas y de columnas del tablero. Recordad que la matriz donde se desarrolla el juego se reserva en memoria dinámica
- constructor de copia
- destructor
- método para comprobar si es posible ubicar un barco en una determinada posición. Recibirá como argumento la fila y columna en que se encuentra, tamaño del barco (número comprendido entre 1 y 9) y carácter indicando la colocación horizontal ('H') o vertical ('V'). El barco puede situarse en la posición indicada si las casillas que ocuparía no están ya ocupadas por otro barco y están dentro del tablero
- operador de asignación

Ejercicio 2 (Junio 2013)

Se desea resolver el problema de sumas de enteros no negativos de gran tamaño (que exceden la capacidad de representación del tipo **int**). Para ello se propone crear la clase **BigInt** que puede almacenar valores enteros de longitud indeterminada.

La clase representará un entero mediante un array (de longitud variable) de enteros, reservado en memoria dinámica (donde cada entero representa uno de los dígitos del entero largo a construir y sólo podrá tomar valores entre 0 y 9). El almacenamiento se produce de forma que el valor menos significativo se guarda en la posición 0 del array. Dos ejemplos de representación de enteros se muestran a continuación (para 9530273759835 y 0):

0	1	2	3	4	5	6	7	8	9	10	11	12		0
5	3	8	9	5	7	3	7	2	0	3	5	9		0

Se pide:

- implementar constructor por defecto (se crea objeto que representa valor 0)
- destructor
- constructor de copia
- método para sumar dos objetos de la clase **BigInt**. El resultado será un nuevo objeto de la clase
- operador `<<`

Ejercicio 3 (Junio 2012)

Se desea crear una clase para poder manejar figuras planas en una aplicación de gráficos 2D. Esta clase, de nombre **Polilinea** se basa en almacenar información sobre puntos en un espacio bidimensional. La información sobre los puntos se gestiona con una clase auxiliar de nombre **Punto**. Las estructuras básicas de estas clases es la siguiente:

```
class Punto{
    int x, y;
    .....
};

class Polilinea{
    Punto *p; // Array de puntos
    int num;  // Numero de puntos
    .....
};
```

Se pide:

- constructor por defecto (para crear una línea poligonal vacía)
- destructor
- constructor de copia
- método agregarPunto para añadir un nuevo punto a una polilínea
- operador de suma (el resultado será una nueva polilínea que contendrá todos los puntos de los objetos sumado)

Ejercicio 4

Dadas las siguientes definiciones de clase:

```
class Celda{
    private:
        double info;
        Celda *sig;
    public:
        .....
};

class Lista{
    private:
        Celda * contenido;
        .....
};
```

donde **info** es una variable de tipo **double** y **sig** es un puntero que apunta a un objeto de tipo **Celda**. Con esta declaración un objeto de la clase **Lista** contiene básicamente un puntero a un objeto de la clase **Celda**, que se corresponde con el primer valor que almacena. En base a estas clases se pide:

- constructor por defecto
- destructor
- constructor de copia
- operador <<

Ejercicio 5 (Septiembre 2016)

Se pretende implementar una clase que permita representar de forma eficiente matrices dispersas, donde sólo un número relativamente bajo de valores son significativos (distintos de cero). Estos valores significativos son los únicos que se almacenan. Cada valor significativo precisa almacenar: fila, columna y valor numérico que representa (de tipo **double**). La información sobre valores significativos se almacena en una clase llamada **Valor**. Asumiendo que ya se dispone de la clase **Valor**, la estructura básica de la clase a crear para representar matrices dispersas sería:

```
class MatrizDispersa{
    private:
        int nfilas;           //numero de filas de la matriz
        int ncolumnas;       // numero de columnas de la matriz
        Valor *valores;      // Array para almacenar los valores significativos
        int numeroValores;   // Numero de valores significativos almacenados
    public:
        .....
};
```

A modo de ejemplo, podemos considerar la siguiente matriz:

1	0	0	0
0	2	0	0
0	0	3	0
0	0	0	4

Un objeto de la clase **MatrizDispersa** para almacenar esta información tendría los siguientes valores de los datos miembro:

- $nfilas = 4$
- $ncolumnas = 4$
- $numeroValores = 4$ (hay 4 valores significativos, en la diagonal principal en este ejemplo)
- valores contendría 4 objetos de la clase **Valor**, con el siguiente contenido:
 - $fila=1$, $columna=1$, $valor=1.0$
 - $fila=2$, $columna=2$, $valor=2.0$
 - $fila=3$, $columna=3$, $valor=3.0$
 - $fila=4$, $columna=4$, $valor=4.0$

Se pide

- constructor por defecto
- destructor
- constructor de copia
- constructor de copia para una matriz determinada. El constructor recibe como argumentos tres arrays: los dos primeros (de tipo **int**) contienen la información de filas y columnas (respectivamente) y el tercero contendrá los valores de tipo **double** asociados. En el caso de la matriz de ejemplo vista antes, habría que pasar como argumento al constructor: (1,2,3,4), (1,2,3,4), (1.0,2.0, 3.0, 4.0)
- operador de suma (para realizar la suma puede usarse el método desarrollado en el ejercicio 2 de los guiones previos)

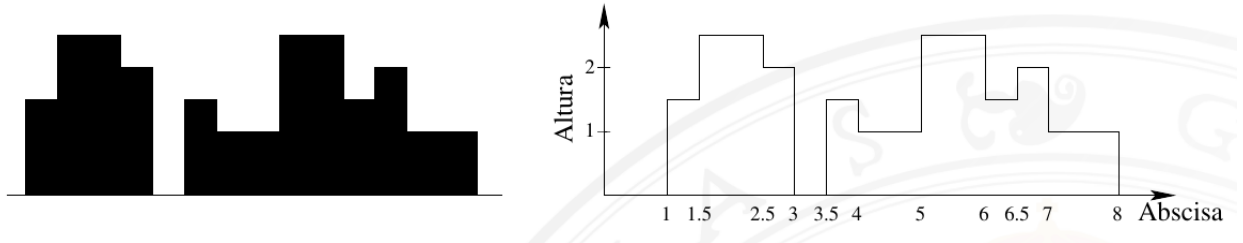
Ejercicio 6 (Junio 2016)

Se define un **Skyline** como la silueta urbana que refleja la vista general de los edificios de una ciudad. Se propone crear una clase para gestionar este tipo de vistas, de forma simplificada, donde los edificios no son más que un rectángulo que se eleva desde una línea base horizontal. La siguiente figura muestra un ejemplo:

La silueta puede codificarse guardando los valores de abscisas y alturas de cada cambio de altura.

Se propone la siguiente representación de la clase:

```
class Skyline {
    private:
        double *abscisas;    // array de abscisas
        double *alturas;     // array de alturas (> 0, la ultima sera
                                0)
```



```

    int n; // tam. de los arrays anteriores
    public:
        .....
};

```

Un objeto representando la información del ejemplo anterior mostraría los siguientes valores de sus datos miembro:

- abscisas: 1, 1.5, 2.5, 3, 3.5, 4, 5, 6, 6.5, 7.8
- alturas: 1.5, 2.5, 2.0, 1.5, 1, 2.5, 1.5, 2, 1, 0
- n: 11

Se pide:

- constructor por defecto
- destructor
- constructor específico para un edificio: recibe como argumento tres valores que representan dos abscisas (comienzo y final) y la altura del edificio
- constructor de copia
- operador []

Ejercicio 7 (Junio 2015)

Se desea crear un programa para calcular el número de repeticiones en una secuencia de números enteros. Por ejemplo, en el caso de la secuencia 939324 existen 4 valores distintos, repitiéndose 9 2 veces, 3 se repite 2 veces, 2 aparece una sola vez y 4 únicamente 1 vez. Se propone crear la siguiente estructura de clases:

```

class Pareja{
    .....
    int dato; // valor
    int nveces; // numero de repeticiones
    .....
};

class Precuencias{
    private:
        Pareja *parejas; // array de objetos de la clase Pareja
        int npares; // numero de objetos almacenados en el array

```

```
public :
    .....
};
```

Se pide:

- constructor por defecto
- destructor
- constructor de copia
- método **agregarValor** que recibe un valor (entero) y lo agrega. Si ya aparece alguna pareja asociada al valor, debe incrementarse su contador. Si no está, se incluirá una nueva pareja
- operadores == y !=

Ejercicio 8 (Septiembre 2013)

Se desea crear una clase **Menu** para simplificar el desarrollo de programas que usan menús en modo texto. Para ello se propone la siguiente representación:

```
class Menu{
    private:
        char * titulo; // Titulo general del menu
        char **opc;    // Cadenas de longitud variable que
                        // describen cada una de las opciones
        int nopc;      // Numero de opciones en el menu
    public:
        .....
};
```

Se pide:

- constructor por defecto
- destructor
- constructor de copia
- métodos **setTitulo** (asigna título al menú), **getNumeroOpciones** (recupera el valor de **nopc**) y **agregarOpcion** (que agrega una nueva opción al objeto)
- operador de asignación

Entrega

La entrega se hará mediante la plataforma **PRADO**. Debéis entregar todo el código desarrollado para los ejercicios. Se recomienda organizar todo el código en un directorio (**guion4**) dentro del que figuran a su vez directorios para cada uno de los ejercicios entregados. A la plataforma se subirá un comprimido (archivo con extensión **tgz** o **zip**) con todo el contenido del directorio base (**guion4**).

La fecha límite de entrega de fijará más adelante.