Metodología de la programación

20 de abril de 2020

ugr | Universidad de Granada

Práctica 4: Clases (ampliación)

Curso 2019-2020

Índice

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 consideréis que se está realizando de forma correcta la práctica. Se debe ofrecer la posibilidad de de poder insertar datos manualmente por parte del usuario.

Se recuerda que el trabajo en estos ejercicios debe ser personal. 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.

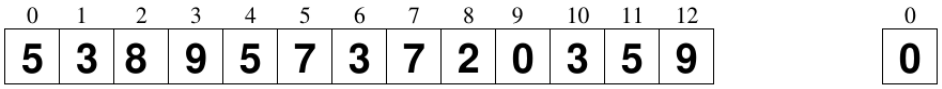
Para todos los ejercicios debes separar el código en módulos independientes para código fuente, archivos de cabecera, archivos de código objeto y ejecutable. Debe proporcionarse un archivo **Makefile** que funcione (con directiva **clean**) y que no contenga elementos o instrucciones no relacionadas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

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):



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 2

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 3

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:

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{matrix}$$

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 4

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 !=

La fecha límite de entrega será el 10 de Mayo.