Guion de prácticas

Práctica Final

Mayo de 2017

DECSAI

Metodología de la Programación

Curso 2016/2017

Índice

1. Definición del problema

En el guion anterior se utilizó un array dinámico para representar internamente la clase *Imagen*, evitando los problemas de desperdicio o falta de memoria producido por el uso de arrays automáticos.
Si bien la clase se puede usar, aún es necesario incorporar un conjunto de métodos para considerarla completa.

En esta práctica se completará la funcionalidad de la clase incorporando el constructor de copia, el destructor y la sobrecarga del operador de asignación. Posteriormente, se repetirá el ejercicio, cambiando la representación interna de la Imagen a una matriz dinámica. Finalmente, se implementará un nuevo programa para realizar un fundido de dos imágenes en un número de pasos dado utilizando, para ello, una lista de celdas enlazadas tal y como se ha explicado en teoría. En todos los casos, se deberá considerar si es necesario reimplementar los posibles métodos afectados dentro de cada una de las clases.

Regla de tres

Cuando nos veamos obligados a definir alguno de los siguientes métodos:

- destructor,
- constructor de copia,
- operador de asignación,

Probablemente tengamos que definir los tres. Si no los definimos, el compilador generar estos métodos por defecto. Si alguno de estos métodos por defecto, no cumple adecuadamente su función, probablemente tampoco lo hagan el resto.

2. Objetivos

El desarrollo de esta práctica pretende servir a los siguientes objetivos:

- Continuar trabajando con memoria dinámica.
- Completar la implementaciones de la clase *Imagen*, con una nueva representación interna, dándose cuenta de que, al no cambiar el interfaz, no es necesario modificar los programas o funciones que usan la clase.
- Realizar la implementación de la clase Lista, según la declaración contenida en el fichero `lista.h` para almacenar en memoria dinámica una secuencia de imágenes.
- Realizar programas que lean parámetros desde la línea de órdenes.
- Realizar un programa nuevo que transforme una imagen en otra, utilizando como base las clases `Imagen` y `Lista` construidas anteriormente.

3. Tareas a realizar

A partir del trabajo especificado en la práctica 5 deberán realizar las siguientes tareas.

```
class Imagen{
private:
   Byte ** datos;    ///< datos de la imagen
   int nfilas;       ///< número de filas de la imagen
   int ncolumnas;    ///< número de columnas de la imagen
```

1. Completar la clase *Imagen*:

   a) Cree una copia de la práctica anterior.

   b) Reimplementar *Imagen* cambiando la representación interna a una matriz dinámica, implementando el destructor, el constructor de copia y la sobrecarga del operador de asignación. Repase los apuntes de teoría. Considere si cambia la implementación de otros métodos cuando la clase está completa. Utilice la representación que mantiene una estructura de punteros al comienzo de las filas con todas las filas almacenadas como una zona continua de memoria. Dicha estructura se muestra en la Fig. 1.

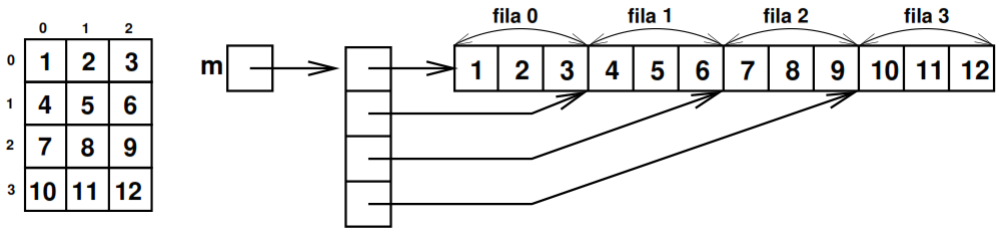

Figura 1: Estructura de la matriz.

Tal como ocurre en la implementación del array dinámico, el constructor de la clase debe reservar memoria y el destructor, liberarla. Adapte las implementaciones del constructor, constructor de copia, destructor, y la sobrecarga del operador de asignación. Repase los apuntes de teoría para implementar la reserva/liberacion de memoria asociada a la representacion utilizada de la matriz. Tenga especial cuidado con los siguientes aspectos:

- Reimplementar los métodos de acceso/consulta a la **Imagen** (set/get, setPos/getPos).
- Reimplementar las funciones de lectura/escritura de la **Imagen** a ficheros sin alterar las funciones de `pgm.cpp`. En esta práctica se considerará la lectura y la escritura de imágenes tanto de texto como binarias.

c) Pruebe la nueva implementación sobre los programas *testimagen*, *testplano* y *testarteASCII* de las práctica anteriores. Recuerde eliminar la llamada (o llamadas) al método *destruir* dentro de cada *main*. Utilice el programa `Valgrind` para comprobar el uso correcto de la memoria dinámica.

2. Reimplemente el programa `testarteASCII` para que lea todos sus parámetros desde la línea de órdenes.

```
ubuntu:$ bin/testarteASCII
Número de parámetros incorrecto
  testarteASCII <imagen> <grises> <salida>
ubuntu:$ bin/testarteASCII data/icon_BIN.pgm data/grises.txt
    data/salida_ascii.txt
Imagen data/icon_BIN.pgm leida correctamente
ubuntu:$
```

donde la salida ASCII correspondiente se guarda en el fichero ASCII de salida, el cual debe tener el mismo contenido que el mostrado en la Práctica 4 y 5 por pantalla.

3. Implementar la clase *Lista* a partir de la siguiente declaración (siguiendo los apuntes de teoría). Esta lista de celdas enlazadas contendrá una imagen en cada celda de la lista:

```
class Celda {
private:
  Imagen *img; /// Contenedor para la imagen
  Celda * sig; /// Puntero a la siguiente celda (se usará
    dentro de la Lista)
public:
  /** @brief Constructor de la clase */
  Celda();
  /** @brief Destructor de la clase */
  ~Celda();
  /** @brief Inserta UNA COPIA de la imagen que se pasa por
    parámetro en la celda (usar operador = sobrecargado)
      @param img La imagen cuya copia se pretende insertar
     */
  void setImagen(const Imagen &img);
  /** @brief Devuelve un puntero a la imagen contenida en la
    celda
      @return La dirección de memoria de la imagen contenida
       */
  Imagen * getImagen() const;
  /** @brief Actualiza el puntero a la siguiente celda con
    el puntero que recibe como argumento
      @param next Puntero a la siguiente celda    */
  void setSiguiente(Celda *next);
  /** @brief Devuelve el puntero a la siguiente celda
     @return Puntero a la siguiente celda    */
  Celda * getSiguiente() const;
};
```

```
/** @brief Una lista enlazada de celdas según los apuntes de
    teoría */
class Lista {
private:
  Celda *lista;
public:
  /** @brief Constructor de la clase */
  Lista();
  /** @brief Destructor de la clase */
  ~Lista();
  /** @brief Libera toda la lista */
  void destruir();
  /** @brief Inserta UNA COPIA de la imagen en la primera
   posición de la lista, desplazando el resto, si lo
   hubiese.
     @param img La imagen cuya copia se pretende insertar
    */
  void insertarPrincipio(const Imagen &img);
  /** @brief Inserta UNA COPIA de la imagen en la última
   posición de la lista.
     @param img La imagen cuya copia se pretende insertar
    */
  void insertarFinal(const Imagen &img);
  /** @brief Devuelve la longitud de la lista
     @return El número de celdas válidas que contiene la
   lista   */
  int longitud() const;
  /** @brief Consulta una celda de la lista
     @param pos Posición ordinal de la celda que se quiere
   consultar
     @return La dirección de memoria de la Imagen en la
   celda que ocupa la posición @a pos dentro de la lista.
   Si @a pos es mayor o igual que la longitud de la lista,
   entonces devuelve 0   */
  Imagen * get(int pos) const;
};
```

4. Implementar el programa `testMorphing` que hace un fundido en un número dado de pasos entre dos imágenes siguiendo el siguiente método.

   a) El programa lee todos los parámetros desde la línea de órdenes

```
ubuntu:$ bin/testMorphing
Número de parámetros incorrecto
  testMorphing <imagen1> <imagen2> <npasos> <salida>
ubuntu:$
```

   `imagen1` se deberá transformar en `imagen2` en `npasos` pasos, donde cada paso es una imagen intermedia, y guardar cada imagen intermedia con el prefijo indicado por `salida` y añadiéndole un número que indique el paso intermedio en que se encuentra

```
ubuntu:$ bin/testMorphing data/rajoy.pgm data/iglesias.
    pgm 10 data/morph
Imagen data/rajoy.pgm leida correctamente
Imagen data/iglesias.pgm leida correctamente
data/morph0.pgm
data/morph1.pgm
data/morph2.pgm
data/morph3.pgm
data/morph4.pgm
data/morph5.pgm
data/morph6.pgm
data/morph7.pgm
data/morph8.pgm
data/morph9.pgm
data/morph10.pgm
ubuntu:$
```

   de forma que el paso 0 se corresponde exactamente con `imagen1`, el último paso se corresponde exactamente con la imagen `imagen2` y cada paso `k` intermedio compone la imagen intermedia con la siguiente fórmula

$$pixel_{i,j}^{k} = pixel_{i,j}^{imagen1} + k * \frac{pixel_{i,j}^{imagen2} - pixel_{i,j}^{imagen1}}{npasos}$$

$$k = 0, \ldots, npasos$$

Figura 2: Secuencia de imágenes de salida en el ejemplo anterior

4. Material a entregar

Cuando esté todo listo y probado el alumno empaquetará la estructura de directorios anterior en un archivo con el nombre **practica5.zip** y lo entregará en la plataforma **decsai** en el plazo indicado. No deben entregarse archivos objeto (**.o**) ni ejecutables. Para asegurarse de esto último conviene ejecutar **make zip** antes de proceder al empaquetado.

El fichero **practica5.zip** debe contener la siguiente estructura:

```
./
├── makefile
├── include
│   ├── byte.h
│   ├── imagen.h
│   ├── pgm.h
│   └── lista.h
├── src
│   ├── byte.cpp
│   ├── imagen.cpp
│   ├── pgm.cpp
│   ├── lista.cpp
│   ├── testimagen.cpp
│   ├── testplano.cpp
│   ├── testarteASCII.cpp
│   └── testMorphing.cpp
├── bin
├── obj
├── lib
├── data
│   ├── lena.pgm
│   ├── giotexto.pgm
│   ├── gio.pgm
│   ├── grises.txt
│   ├── rajoy.pgm
│   └── iglesias.pgm
└── zip
```

El alumno debe asegurarse de que ejecutando las siguientes órdenes se compila y ejecuta correctamente su proyecto:

```
unzip practica6.zip
make
bin/testimagen
bin/testplano
bin/testarteASCII <imagen> <grises> <salida>
bin/testMorphing <imagen1> <imagen2> <npasos> <salida>
```