UNIVERSIDAD DE GRANADA

ETSIIT Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

DECSAI

Metodología de la Programación

DGIM, GI-C y DGADE

Curso 2022/2023

Guion de prácticas

Language4

Memoria Dinámica dentro de la clase Language

Mayo de 2023

Contents

1 Definición del problema

La implementación de la clase Language utilizada hasta ahora, almacena los objetos `BigramFreq` en un vector cuyo tamaño se fija en tiempo de compilación. En la definición, hasta el momento, de la clase tenemos lo siguiente:

```
class Language {
private:

   static const int DIM_VECTOR_BIGRAM_FREQ = 2000; ///< The capacity of the array _vectorBigramFreq
    std::string _languageId; ///< language identifier
    BigramFreq _vectorBigramFreq[DIM_VECTOR_BIGRAM_FREQ]; ///< array of BigramFreq
    int _size; ///< Number of elements in _vectorBigramFreq


    static const std::string MAGIC_STRING_T; ///< A const string with the magic string for text files
public:...
```

Como consecuencia de esto, cada vez que se instancia un objeto de la clase Language, se "crea" un vector de tamaño `DIM_VECTOR_BIGRAM_FREQ`, capacidad máxima que puede tener nuestro vector, aunque el language actual contenga unos pocos bigramas ocupados.

Una manera de evitar este problema de desperdicio de memoria es reservar, en tiempo de ejecución, solamente la memoria que se va a necesitar [1]. Se hace necesario un cambio mayor en la clase Language: el vector de objetos `BigramFreq` va a ser dinámico, ajustándose así el tamaño del vector al número de bigramas que va a contener el objeto language.

```
class Language {
private:
    std::string _languageId; ///< language identifier
    BigramFreq* _vectorBigramFreq; ///< Dynamic array of BigramFreq
    int _size; ///< Number of elements in _vectorBigramFreq
    static const std::string MAGIC_STRING_T; ///< A const string with the magic string for text files

public:...
```

Se trata por tanto, de redefinir la parte interna de la clase para gestionar memoria dinámica dentro de la clase. Esto implica la definición de métodos específicos como: constructor de copia, operador de asignación, destructor y también de una serie de métodos privados para una mayor modularidad.

Si los métodos no privados han sido implementados adecuadamente (maximizando la reutilización de los métodos básicos de la clase) el esfuerzo de refactorización se reduce notablemente, limitándose a constructores y algunos métodos modificadores.

El programa principal de la práctica que aquí nos ocupa, realiza las mismas tareas que el de la práctica anterior para la predicción del idioma de un language determinado en función de la distancia calculada a cada uno de los language con los que se compara. El programa principal que utiliza todas las clases anteriores (esto es, que ve las clases desde fuera) no sufre ningún cambio. Lo que significa que, todas las ejecuciones de `language4` son idénticas en sintaxis y resultados a las utilizadas con `language3`, salvo por el cambio de nombre a `language4`. Las ejecuciones de uno y otro deben arrojar los mismos resultados.

[1] Conocemos el tamaño del vector de pares-frecuencia a reservar para un objeto language cuando se va a leer ya que, se encuentra en la cabecera de cualquier fichero `.bgr`.

Por último, al igual que ocurriera en la práctica anterior, los distintos languages que van a intervenir en el proceso de predicción, también se van a almacenar en memoria dinámica.

2 Arquitectura de las prácticas

Como ya se indicó en la práctica anterior, la práctica `Language` se ha diseñado por etapas, las primeras contienen estructuras más sencillas, sobre las cuales se asientan otras estructuras más complejas y se van completando con nuevas funcionalidades.

En `Language4` se cambia la componente privada de la clase `Language` para alojar el vector de objetos `BigramFreq` de forma eficiente en memoria dínámica, bloque **C'** de la Figura 1, el resto de clases permanecen iguales.

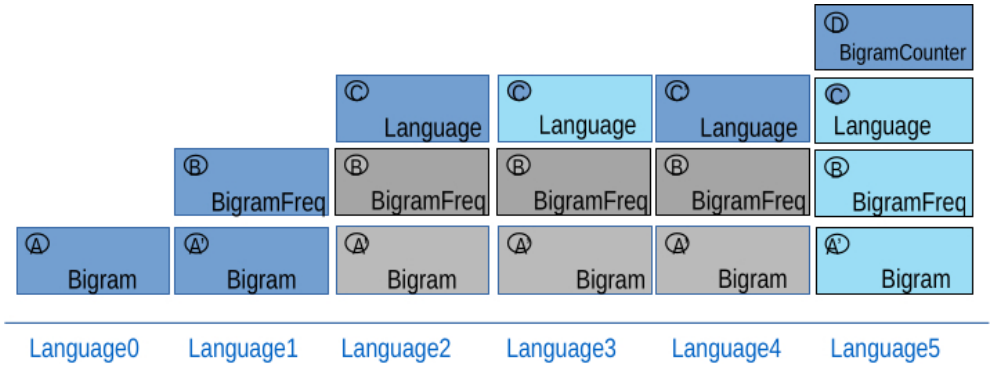

Figura 1: Arquitectura de las prácticas de MP 2023. Los cambios esenciales en las clases (cambio en estructura interna de la clase) se muestran en azul intenso; los que solo incorporan nuevas funcionalidades en azul tenue. En gris se muestran las clases que no sufren cambios en la evolución de las prácticas.

C' Language.cpp
: Manteniendo la interfaz previa, al cambiar la parte privada de la clase con memoria dínámica, se necesita refactorizar algunos métodos de la clase y se han de incluir otros nuevos para su correcto funcionamiento. Esto supone cambios mayores dentro de la clase.

3 Objetivos

El desarrollo de esta práctica `Language4` persigue los siguientes objetivos:

- Practicar con punteros y memoria dinámica dentro de una clase.
- Aprender a desarrollar los métodos básicos de una clase que contenga memoria dinámica: constructor de copia, destructor y operador de asignación.

- Practicar con herramientas como el depurador de NetBeans y `valgrind` para rastrear errores en tiempo de ejecución.
- Modificar la clase Language de prácticas anteriores para incluir gestión de memoria dinámica.
- Refactorizar la clase Language para reducir al mínimo el número de cambios a realizar en los métodos anteriores de la clase.

4 Memoria dinámica en la clase Language

A partir de los ficheros de código de la práctica anterior, realizar los siguientes cambios.

- Refactorizar la clase `Language` utilizando como estructura interna un vector dinámico.
  - Ahora, el constructor de la clase debe reservar memoria (Repase los apuntes de teoría para implementar la reserva de memoria asociada a un vector). Más concretamente, el constructor sin parámetros debe crear un objeto language vacío (0 componentes BigramFreq, y sin memoria reservada).
  - El constructor con parámetros debe reservar la memoria para el language y ajustar el número de componentes, `size`, para mantener la consistencia del objeto.
  - Por último tenga en cuenta que el método de lectura debe crear el language **antes** de leer los datos.
- Implementar constructor de copia, destructor y operador de asignación.
- Revisar el resto de métodos de Language por si necesitan ser refactorizados.
- Se recomienda implementar métodos privados como: `allocate()`, `deallocate()`, `copy()` etc., para evitar repetir código, lo cual sería fuente de futuros errores.

5 Práctica a entregar

Para la elaboración de la práctica, puede reutilizar todos los ficheros de código utilizados en la práctica anterior. Tan solo requiere el nuevo fichero `Language.h`. La práctica deberá ser entregada en Prado, en la fecha que se indica para la entrega, con el nombre `Language4.zip`