Universidad de Costa Rica

Escuela de Ciencias de la Computación e Informática

CI-0113 Programación II

**Tarea Programada 2**

Documentación externa

Estudiantes:

* Sebastián Cruz Chavarría – B72458
* Roy Andrés Rojas Angulo – B76629

Prof. Edgar Casasola Murillo

I – 2018

Índice

Descripción del problema …………………………………………………………. 3

Descripción de la solución ………………………………………………………… 4

Diagrama de clases ………………………………………………………………... 6

Descripción del programa …………………………………………………………. 7

Compilación del programa …………………………………………………………12

Casos de prueba …………………………………………………………………….12

Formato de los casos de prueba …………………………………………………..12

Salida esperada y obtenida ……………………………………………………….

Lista de archivos entregados ………………………………………………………13

Descripción del problema

El problema que se plantea en el enunciado de la tarea consiste básicamente en la separación de las palabras que conforman un *hashtag* de opinión, dado que éstos por lo general se componen de varias palabras unidas sin distinción del principio y del final de cada una en medio del comentario. La tarea de separar estas palabras a través de un programa puede facilitar su lectura y comprensión para el usuario final, quien no tendrá que hacer mayor esfuerzo en tratar de darle sentido al comentario de palabras unidas.

No obstante, la resolución de este problema mediante un programa, y en la forma que se especificó en el enunciado, implica más problemas pequeños cuya solución converge en la solución del problema inicial. Para citar algunos de esos problemas, se encuentra el más profundo que, a nuestro parecer, corresponde a la creación de palabras a través de vectores de punteros a Nodo, en donde el camino de punteros y el puntero de la última posición que apunta a sí mismo conforma e indica el final de una determinada palabra en el árbol de vectores de punteros a Nodo. Por otro lado, los otros problemas que devienen en la programación de la clase Árbol, Palabra y Diccionario están fuertemente relacionados con el problema de los vectores de punteros a Nodo, por lo que, una vez resuelto éste, los demás no presentarán mayor conflicto en el proceso de creación del programa.

Otro problema fuera de éste ámbito, pero no menos importante, es el de la creación de un archivo .SVG dinámico e interactivo que debe desplegar el camino de vectores de punteros a Nodo para una palabra especificada por el usuario, para verificar que la creación a través de los punteros es la que se especificó en el enunciado.

Un problema más que, aunque parece insignificante, podría llevar mucho trabajo resolver, consiste en la inclusión al Diccionario de palabras con caracteres como *á, é, í, ó, ú, ü* y *ñ,* dado que manejar adecuadamente éstos en C++ puede ser complicado.

Descripción de la solución

Como se mencionó anteriormente, la solución del problema global se dividió en varios problemas más pequeños, iniciando por el más básico cuya solución consiste en la base de la tarea: la formación de palabras mediante un camino conformado por vectores de punteros a Nodo. Se planteó que fueran punteros a Nodo puesto que únicamente con vectores de punteros a punteros se torna más difícil representar la información necesaria para determinar el camino de una palabra. Como esta clase es la que ocupa el nivel más bajo en nuestra solución, no conoce de los miembros superiores en la jerarquía de clases del programa, como la clase Arbol, Palabra, Diccionario y Texto, y tampoco conoce de caracteres, puesto que los vectores de punteros representan cada palabra únicamente con el camino de punteros que la describe a través de los índices de cada vector.

Más arriba, la clase Arbol tiene como miembro privado un puntero a un Nodo raíz, en donde ambas conforman la estructura base de la tarea que contiene las palabras en forma de árbol de punteros a Nodos. Esta clase conoce la clase Palabra, puesto que las necesita para agregarlas al árbol de vectores de punteros a Nodos haciendo uso de la clase Nodo.

En la clase Palabra se llevan a cabo los procesos de creación de palabras propiamente a partir de un *string,* una cadena de caracteres e incluso con otra instancia de Palabra, además de realizar algunos procesos necesarios para la inclusión de cada palabra al Árbol de Nodos.

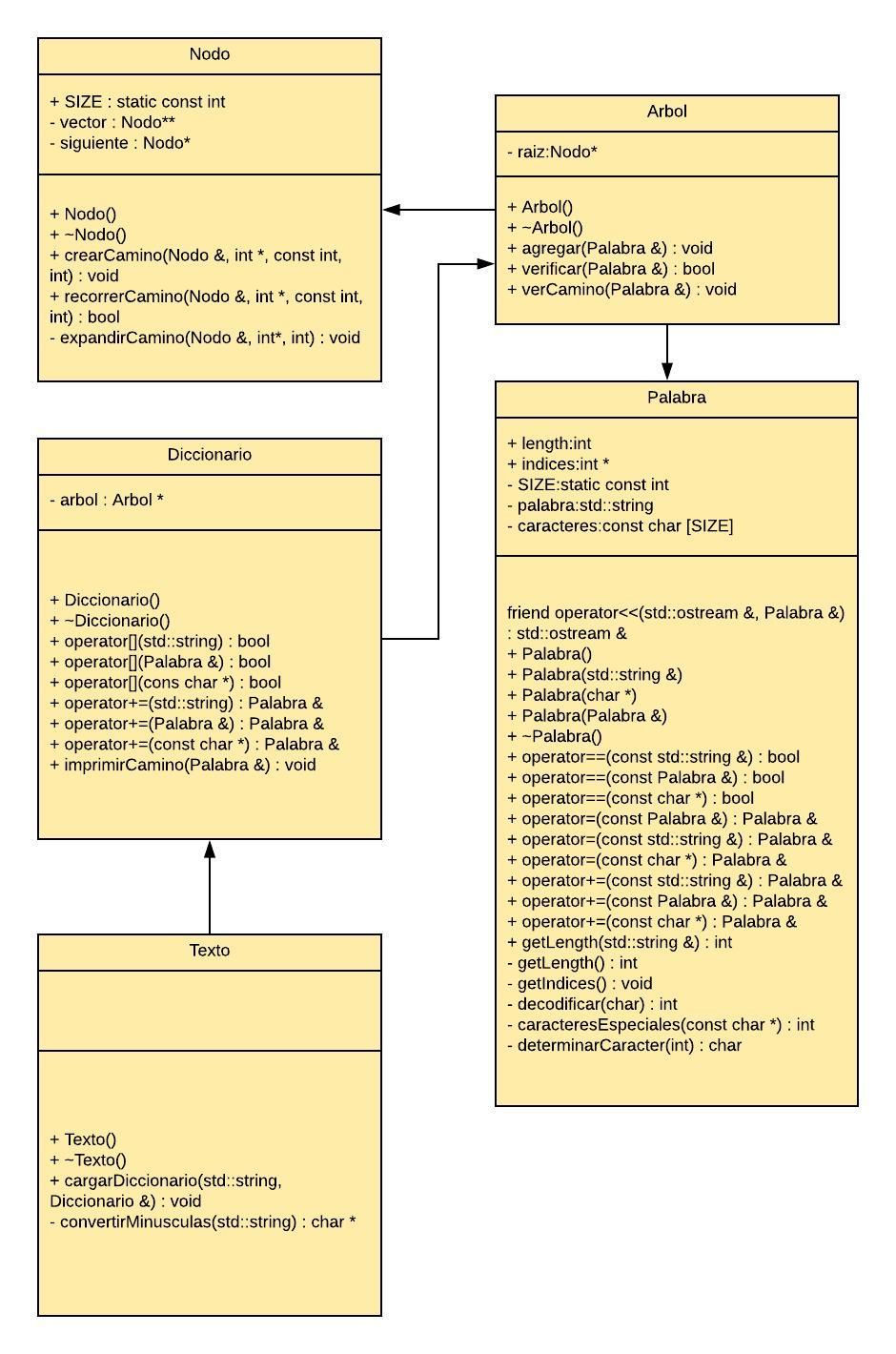
Siguiendo la jerarquía hacia arriba, la clase Diccionario modela al diccionario propiamente dicho, que se compone de las Palabras almacenadas en el Árbol que a su vez tiene el Nodo raíz de todas las palabras que se incluyan en él. De esta manera, cada nivel más arriba en la jerarquía de clases envuelve las clases inmediatamente debajo, de modo que la clase Diccionario no conoce de Nodos ni de los vectores de punteros a Nodos que almacena el Árbol.

En ese mismo nivel, se encuentra la clase Texto que agrupa y utiliza directa e indirectamente las demás clases que forman parte del programa solución para el problema planteado. Esta clase es la que se encarga de recibir los archivos como el *diccionario.txt* y los archivos de comentarios de opinión para procesarlos como se solicitó. La clase Texto lee y divide cada palabra del diccionario suministrado y las añade al contenedor de la clase Diccionario, para tenerlas disponibles cuando se necesiten consultas de palabras al momento de ejecutar la división de palabras en los comentarios de opinión.

La división de palabras de los comentarios acompañados por un *hashtag* se programó –en la forma más genérica– para leer carácter por carácter el comentario, formando una Palabra con los n caracteres que lee desde el inicio para buscar su existencia en el Diccionario en cada iteración. Cuando se encuentra con una palabra existente en el Diccionario, la añade al archivo de salida que corresponde a las palabras del comentario separadas, y vuelve a empezar el proceso a partir de la última palabra que encontró en el Diccionario, hasta llegar a la última.

Cabe destacar que el buen funcionamiento del programa está supeditado a la inexistencia de faltas de ortografía en los comentarios de opinión, a menos que el diccionario con el que se usa el programa también incluya las palabras erróneas de los comentarios.

Diagrama de clases UML



*Figura 1. Diagrama de clases UML del programa.*

Descripción del programa

Clase Nodo

De acuerdo con la estructura esbozada anteriormente del programa, la clase Nodo es la que se encarga de crear los vectores de punteros a Nodo para cada palabra que se desee agregar al Diccionario, así como también se encarga de recorrer un camino de vectores de punteros a Nodo para determinar si una palabra existe, e imprime, a petición del usuario, el camino de punteros que conforma una palabra en la salida estándar de consola, mediante sus funciones *crearCamino(), recorrerCamino* e *imprimirCamino().*

La función *crearCamino(),* como su nombre bien lo describe, crea el camino de vectores de punteros a Nodo para cada palabra que se le ingrese. Sin embargo, como la clase Nodo es lo más interno del programa que resuelve el problema global, no conoce de Diccionario, ni de Palabras, ni de Arbol, ni siquiera de caracteres; los argumentos que requieren sus funciones le son proporcionados desde las clases que hacen uso de ella, por lo que para cada Palabra que se quiere agregar al árbol de Nodos, es traducida en un conjunto de elementos que representan a la Palabra desde un nivel más bajo, como lo son el vector de índices que es la “traducción” de cada carácter de la palabra a número entero, según el subíndice del carácter que corresponde en el vector de caracteres que contiene los caracteres permitidos para una Palabra en esta tarea. Así mismo, otro elemento que se le proporciona a *crearCamino()* es el largo de la Palabra, y un entero para representar el carácter actual para el que se está creando el camino de vectores de punteros a Nodo.

La función *recorrerCamino()* tiene como tarea saltar de puntero en puntero entre los vectores para verificar la existencia de una palabra. Recibe los mismos argumentos que *crearCamino()* para completar su tarea y uno entero adicional con el valor predeterminado de 0, que se le proporciona únicamente cuando el usuario solicita ver el camino para una determinada palabra. Para recorrer el camino de punteros, toma el Nodo por referencia que se le proporciona como argumento y verifica en una cláusula condicional, entre otras cosas, que exista un puntero a Nodo en el subíndice del vector de punteros a Nodo del Nodo raíz; que ese Nodo tenga un puntero al siguiente, que el campo índice de ese Nodo corresponda al índice que se le proporcionó en el llamado actual (para verificar que el camino corresponde a la palabra que se está buscando) y que el entero *caracterActual* sea menor que el largo de la palabra. De cumplirse todas las anteriores condiciones, verifica si se encuentra en el final de la palabra que se está buscando para retornar el valor booleano *true*, para indicar que la palabra existe. En la rama del *else*, toma como nuevo Nodo raíz el Nodo al que apunta *siguiente*, y vuelve a llamarse a la función con los nuevos argumentos, hasta llegar al último carácter de la palabra o hasta encontrar inconsistencias en el camino de punteros que indiquen que la palabra no existe.

Finalmente, la función *verCamino()* imprime en la salida estándar de consola una representación gráfica del vector de punteros del Nodo raíz que se le pasa como argumento, junto con otros elementos que se le proporcionan para determinar si un puntero a Nodo es parte de la palabra que se está buscando. Se imprime una fila de números desde 0 hasta 33, seguido de una flecha “->” y un carácter que tiene tres variantes: ‘X’ cuando el puntero a Nodo forma parte del camino de la palabra que se está buscando; ‘x’ cuando existe un puntero en ese vector que no forma parte de la palabra que se está buscando, y ‘o’ cuando no existe ningún puntero en la posición del vector de punteros a Nodo, para indicar que el espacio está vacío (no existen palabras por ese camino).

Esta función es privada en la clase Nodo, dado que quien la utiliza es la función *recorrerCamino()* cuando se le confirma mediante los argumentos que el usuario desea ver el camino para la palabra que se quiere verificar.

Se proporcionó la visualización de una palabra a través de la salida estándar de consola como sustitución de la visualización mediante un archivo SVG, debido a los vacíos de conocimiento del equipo de trabajo que no permitieron realizar lo solicitado en dicho formato.

Clase Palabra

Mediante esta clase se trabajan los objetos tipo Palabra, que son los únicos aptos para agregar al Árbol de Nodos. Cuenta con varios constructores, que facilitan la creación de una Palabra mediante un *string*, una cadena de caracteres e incluso con otra Palabra. Se trabaja con esta clase porque proporciona elementos necesarios para agregar cada palabra al árbol de vectores de punteros, como lo son el vector de índices que representa a cada carácter de la palabra con un número, también cada Palabra conoce su largo y a lo interno, la “tira” por sí misma es un *string*.

La clase implementa la sobrecarga de los operadores +=, el de subíndice “[ ]”, el de asignación “=” y el de igualdad “==”, para facilitar la manipulación de instancias de esta clase. Se enlistan tres funciones prácticamente iguales para cada sobrecarga, para poder utilizar como *rvalue* un *string*, una cadena de caracteres, y otra Palabra, para prescindir de las tediosas conversiones explícitas de un tipo a otro.

En esta clase se guarda la representación de cada carácter permitido, para efectos de esta tarea, en cualquier palabra que se ingrese al diccionario según lo especificado en el enunciado. Estos caracteres se guardan en el atributo privado *caracteres[SIZE],* donde *SIZE* es un entero constante de valor 34, para almacenar en ese vector los caracteres [a-z], á, é, í, ó, ú, ü, ñ. La función privada *decodificar()* recibe un carácter como parámetro y lo compara con cada uno de los caracteres del vector ya mencionado, para devolver el subíndice del carácter al encontrarlo, o lanzar una excepción *invalid\_argument* para indicar que la Palabra tiene un carácter no permitido. Como el lenguaje de programación C++ es poco amigable con caracteres no-anglosajones (vocales tildadas, ñ y ü para efectos de esta tarea), se incluyeron dos funciones que junto con *decodificar()* permiten el manejo adecuado de estos caracteres en el programa. La función *caracteresEspeciales()* analiza cada Palabra en búsqueda de los caracteres especiales ya mencionados, para lo cual promueve su valor *char* a *int* y cuenta la cantidad de veces que aparezca en el vector de caracteres algún número negativo, ya que esta característica es la que los distingue del resto de caracteres “comunes”. Este conteo es necesario para determinar el largo correcto del vector de índices, ya que, cuando una Palabra incluye caracteres especiales, al convertir cada carácter en un valor numérico, cada carácter especial representa dos valores numéricos negativos, produciendo un excedente en la cantidad de campos del vector de índices que la representará al crearse. Para evitar este inconveniente, a la declaración del vector de índices se le resta la cantidad de caracteres especiales.

La función *determinarCaracter()* recibe un valor entero al que es promovido su argumento en forma natural de *char* y lo decodifica identificando a cuál de los posibles siete valores de la tabla ASCII corresponde, para que *decodificar()* no tenga problemas en identificar el subíndice del carácter que recibe como argumento al buscarlo en el vector constante de caracteres que contiene todos los posibles caracteres admisibles en una Palabra, para efectos de esta tarea.

Clase Arbol

Siguiendo la analogía de un árbol real, la clase Arbol contiene un solo atributo privado: un puntero a Nodo raíz. Esta clase es el único cliente directo de la clase Nodo, y es mediante la misma que se añaden palabras a la estructura de árbol que forma las palabras, a través de su Nodo raíz. Se encarga apenas de obtener los atributos necesarios de cada Palabra para que el Nodo raíz pueda agregarla a su estructura de vectores de punteros. Sus tres funciones *agregar(), verificar()* y *verCamino()* reciben únicamente un objeto Palabra por referencia, obtienen su representación como vector de índices y su largo mediante estos campos públicos de cada Palabra, y se lo proporcionan a las funciones del Nodo raíz *crearCamino()* y *recorrerCamino().* Cabe destacar que la función *verCamino()* primero verifica la existencia de la Palabra que se le indica entre los vectores de punteros; imprime su camino en la salida estándar de consola si existe, y si no imprime un mensaje de error indicando que la palabra solicitada no existe.

Clase Diccionario

La clase Diccionario es utilizada para modelar un diccionario de palabras que, más que brindar una definición de cada palabra, sólo verifica su existencia dentro del Arbol de vectores de punteros que tiene como único atributo privado.

Cuenta con los operadores sobrecargados de subíndice “[ ]” y “+=”, cuyas funciones son buscar una palabra indicada entre los corchetes, y agregar una palabra al diccionario respectivamente. De la misma forma que la clase Palabra, Diccionario incorpora tres distintas versiones de sobrecarga para cada operador, para poder utilizar como *rvalue* tanto un *string* como una cadena de caracteres y un objeto Palabra, aunque a lo interno, cuando se recibe un *string* o una cadena de caracteres, estos elementos son convertidos en una Palabra, ya que la estructura del Árbol sólo conoce de objetos Palabra. También cuenta con la función *imprimirCamino(),* que delega su trabajo a *verCamino()* de Árbol con el mismo argumento con el que se llamó.

Clase Texto

Esta clase representa la funcionalidad total del programa, así como el sentido de la realización de esta tarea: la división de comentarios de *hashtag*, para lo cual se necesita un trabajo en conjunto con las demás clases anteriormente desglosadas: Nodo, Árbol, Palabra y Diccionario. Pero antes de realizar este trabajo, necesita de un Diccionario con las palabras válidas almacenadas, para poder reconocerlas en los comentarios y dividirlas basándose en ese criterio. Entonces, la clase Texto implementa la función *cargarDiccionario()*, la cual recibe como parámetros un *string* con el nombre del archivo de texto donde se guarda el diccionario a cargar, y un objeto Diccionario que toma por referencia al que le carga las palabras, que debe ser proporcionado desde el *main*, ya que el Diccionario es independiente de la clase Texto y sus funciones deben estar disponibles para cualquier otra clase.

Para cargar las palabras al Diccionario, se apoya en la función privada *convertirMinusculas()* para no introducir palabras con caracteres ilegales que puedan afectar la ejecución del programa. La función *cargarDiccionario()* lanzará una excepción del tipo *invalid\_argument* si el archivo de texto que trata de cargar no existe, o su ruta no está bien especificada.

Compilación del programa

La compilación no posee ningún requisito especial, únicamente que se compilen todos los archivos de tipo .cpp que estén en la carpeta donde se encuentren todas las clases del programa.

La ejecución del programa se basaría en el main.cpp, el cuál se iniciaría con la correcta compilación mencionada anteriormente. En el main se mostrará el menú especificado en el enunciado de esta tarea.

Las entradas que se le deben de brindar al programa son únicamente los nombres de los archivos de texto. Tanto del archivo que contiene al diccionario, como el archivo de los comentarios (hashtags). Cabe destacar que estas entradas se pedirán una vez el usuario esté dentro del menú del programa, no a la hora de compilación.

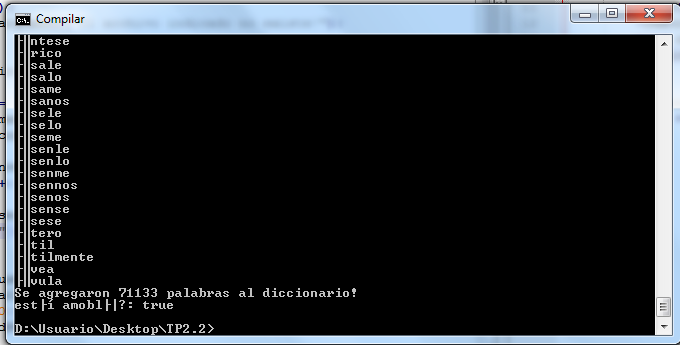
Casos de prueba

Formato de casos de prueba

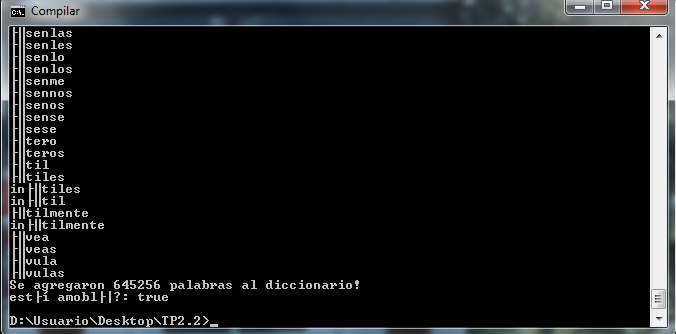
Como se mencionó anteriormente las únicas entradas que se le deben brindar al programa para que funcione correctamente, osea, para que cargue el diccionario y el archivo con los comentarios, son los nombres de los mismos.

Salida esperada y obtenida

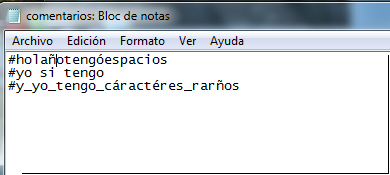
En la prueba que se realizó en la siguiente captura de pantalla se quería verificar que el método agregarPalabras de la clase Diccionario agregaba todo el diccionarioLimpio, además se buscaba con el operador [] sobrecargado de Diccionario la palabra amoblé, la cual fue encontrada.

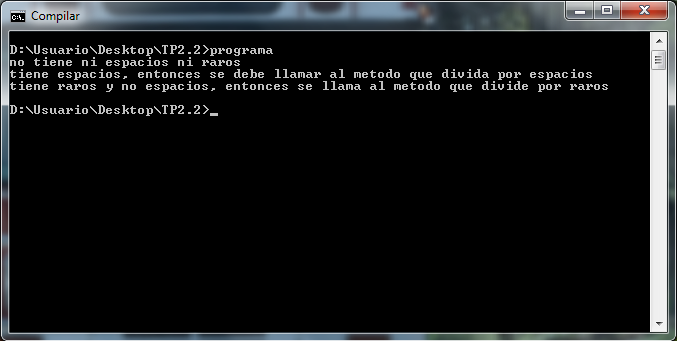


Aquí se quería comprobar que el mismo método agregaba todo el diccionarioGrande, y se volvía a buscar la palabra amoblé.

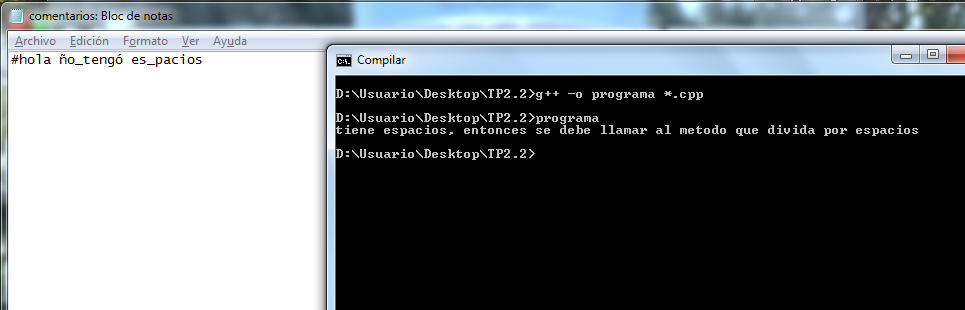


Se prueba el método separar de la clase Texto, el cuál recibiendo un archivo con 3 hashtags debía clasificarlos e imprimir cuál debería ser el método por el que se dividan. Se recibió la salida esperada ya que los clasificó correctamente.



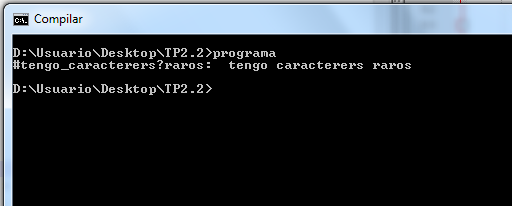


Aquí otra prueba de lo mismo.



En la siguiente captura de ejecución se está imprimiendo el string que retorna el método divPorRaros de la clase Texto, que se encarga de dividir un comentario por aquellos caracteres que no se encuentran en el vector de caracteres.

Como se observa imprime primeramente el hashtag que se recibió y seguidamente las divisiones del mismo.



Lista de archivos entregados

Lista de archivos en la carpeta comprimida:

-main.cpp

-Diccionario.h

-Diccionario.cpp

-Arbol.h

-Arbol.cpp

-Palabra.h

-Palabra.cpp

-Nodo.h

-Nodo.cpp

-Texto.h

-Texto.cpp

-DiagramTP2.jpeg

-Documentacion\_Externa.docx