Laboratorio 5 de Programación 3

Leandro Rabindranath León

- **Este laboratorio es evaluado y la calificación que obtengas cuenta para tu calificación definitiva.**
- **Fecha de entrega: lunes 24 de noviembre de 2014 hasta las 6 pm**
- **Tu entrega consiste de un archivo llamado `word-find.H`, Por favor, al inicio de este archivo, en comentarios, pon los apellidos, nombres y números de cédula de los miembros de la pareja que está sometiendo el archivo.**

1. Introducción

El fin de este laboratorio es continuar el desarrollo de destrezas algorítmicas en búsqueda y backtracking. Ya tuviste una experiencia con este tipo de algorítmica cuando resolviste el sudoku. En este laboratorio la diferencia será que el backtracking es asistido por el uso de una estructura de dato, concretamente, un árbol de prefijos como el que desarrollaste en la práctica previa. Para cumplir este propósito en esta laboratorio instrumentarás un algoritmo que resuelve el juego de la sopa de letras[1].

Para la realización de este laboratorio requieres:

1. La suite clang
2. La biblioteca $\aleph_\omega$ (Aleph-ω); especialmente debes estar familiarizado con el tipo `DynList` y sus distintos métodos.

1.1. Sopa de letras

El juego de la sopa de letras consiste en una matriz $n \times m$ de letras en la cual se desean encontrar palabras. Probablemente este pasatiempo te sea familiar.

En el pasatiempo tradicional, se buscan palabras "lineales" respecto a la matriz. Es decir, se verifica si una secuencia horizontal, vertical o diagonal respecto a la matriz se corresponde con una palabra.

En esta versión de la sopa de letras buscaremos palabras en secuencias no lineales de la matriz. Esto quiere decir que una palabra puede cruzar en las casilla de la matriz cuantas veces se pueda, limitado por supuesto a su longitud.

Como ejemplo, consideremos las siguientes palabras:

[1]Este problema está inspirado en uno parecido desarrollado en el programa de algoritmos de l'École Polytechnique, Francia.

```
rabindranath hacker informatica computacion ingenieria
sistemas murcielago inteligente laboratorio tagore instrumentacion
certificacion algoritmo
```

Y ahora veamos una sopa de letras que las contiene:

	0	1	2	3	4	5	6	7	8	9	10	11
0	e	i	n	d	a	n	t	h	a	o	r	o
1	b	f	c	h	r	a	t	b	l	o	i	p
2	i	a	r	a	k	g	o	e	a	t	i	o
3	t	d	c	c	e	g	o	r	u	a	r	l
4	r	i	o	n	a	r	u	t	e	a	e	o
5	e	c	n	t	l	i	n	i	n	e	i	p
6	k	n	i	e	t	g	e	c	g	o	n	o
7	m	u	i	u	a	p	a	l	i	r	i	m
8	r	c	p	c	i	c	a	o	t	v	u	t
9	m	a	s	m	o	o	r	a	f	m	s	r
10	e	t	s	e	n	n	m	c	e	n	n	t
11	h	e	s	i	y	o	i	n	a	t	i	i

La primera fila y la primera columna son índices de columnas y filas respectivamente. No pertenecen a la sopa de letras. Se colocan para referir fácilmente las casillas de la matriz mediante un par de índices (fila, columna).

Ahora veamos algunas dónde se encuentran las palabras anteriores.

```
instrumentacion i(11,11)n(10,10)s(9,10)t(10,11)r(9,11)u(8,10)m(9,9)e(10,8)
                n(10,9)t(11,9)a(11,8)c(10,7)i(11,6)o(11,5)n(10,4)
certificacion c(5,1)e(5,0)r(4,0)t(3,0)i(2,0)f(1,1)i(0,1)c(1,2)a(2,1)
              c(3,2)i(4,1)o(4,2)n(4,3)
rabindranath r(2,2)a(2,1)b(1,0)i(0,1)n(0,2)d(0,3)r(1,4)a(0,4)n(0,5)a(1,5)t(1,6)h(0,7)
laboratorio l(1,8)a(0,8)b(1,7)o(2,6)r(3,7)a(2,8)t(2,9)o(1,9)r(0,10)i(1,10)o(0,9)
inteligente i(6,2)n(5,2)t(5,3)e(6,3)l(5,4)i(5,5)g(6,5)e(6,6)n(5,6)t(4,7)e(4,8)
computacion c(8,5)o(9,4)m(9,3)p(8,2)u(7,3)t(6,4)a(7,4)c(8,3)i(8,4)o(9,5)n(10,5)
informatica i(11,10)n(10,9)f(9,8)o(8,7)r(9,6)m(10,6)a(9,7)t(8,8)i(7,8)c(6,7)a(7,6)
murcielago m(7,0)u(7,1)r(8,0)c(8,1)i(7,2)e(6,3)l(5,4)a(4,4)g(3,5)o(3,6)
ingenieria i(5,7)n(5,8)g(6,8)e(5,9)n(6,10)i(5,10)e(4,10)r(3,10)i(2,10)a(3,9)
algoritmo a(8,6)l(7,7)g(6,8)o(6,9)r(7,9)i(7,10)t(8,11)m(7,11)o(6,11)
sistemas s(10,2)i(11,3)s(11,2)t(10,1)e(10,0)m(9,0)a(9,1)s(9,2)
tagore t(0,6)a(1,5)g(2,5)o(2,6)r(3,7)e(2,7)
hacker h(1,3)a(2,3)c(3,3)k(2,4)e(3,4)r(4,5)
```

Como entrenamiento para la programación de la solución del pasatiempo, verifica tú mismo algunas de las palabras.

Del mismo modo, es importante que aprehendas que una sopa de letras puede contener varias veces la misma palabra. Por ejemplo, el apellido “Tagore” y el adjetivo “inteligente” se encuentran en las siguientes secuencias:

```
inteligente (6,2)(5,2)(5,3)(6,3)(5,4)(5,5)(6,5)(6,6)(5,6)(4,7)(4,8)
```

```
inteligente (4,1)(5,2)(5,3)(6,3)(5,4)(5,5)(6,5)(6,6)(5,6)(4,7)(4,8)
tagore (5,3)(4,4)(3,5)(3,6)(4,5)(3,4)
tagore (5,3)(4,4)(3,5)(3,6)(3,7)(4,8)
tagore (5,3)(4,4)(3,5)(3,6)(3,7)(2,7)
tagore (5,3)(4,4)(3,5)(2,6)(3,7)(4,8)
tagore (5,3)(4,4)(3,5)(2,6)(3,7)(2,7)
tagore (1,6)(1,5)(2,5)(3,6)(4,5)(3,4)
tagore (1,6)(1,5)(2,5)(3,6)(3,7)(4,8)
tagore (1,6)(1,5)(2,5)(3,6)(3,7)(2,7)
tagore (1,6)(1,5)(2,5)(2,6)(3,7)(4,8)
tagore (1,6)(1,5)(2,5)(2,6)(3,7)(2,7)
tagore (0,6)(1,5)(2,5)(3,6)(4,5)(3,4)
tagore (0,6)(1,5)(2,5)(3,6)(3,7)(4,8)
tagore (0,6)(1,5)(2,5)(3,6)(3,7)(2,7)
tagore (0,6)(1,5)(2,5)(2,6)(3,7)(4,8)
tagore (0,6)(1,5)(2,5)(2,6)(3,7)(2,7)
```

Nota que las soluciones que se puedan encontrar dependen de las palabras que se busquen. Por ejemplo, otras palabras que contiene la sopa anterior, entre muchísimas otras, son:

```
aroma (8,6)(9,6)(9,5)(10,6)(9,7)
aromatica (8,6)(9,6)(9,5)(10,6)(9,7)(8,8)(7,8)(6,7)(7,6)
nitrogeno (5,8)(5,7)(4,7)(3,7)(3,6)(3,5)(3,4)(4,3)(4,2)
nitrogeno (5,8)(5,7)(4,7)(3,7)(3,6)(2,5)(3,4)(4,3)(4,2)
nitrogeno (5,8)(5,7)(4,7)(3,7)(2,6)(3,5)(3,4)(4,3)(4,2)
nitrogeno (5,8)(5,7)(4,7)(3,7)(2,6)(2,5)(3,4)(4,3)(4,2)
nitrogeno (5,6)(5,7)(4,7)(3,7)(3,6)(3,5)(3,4)(4,3)(4,2)
nitrogeno (5,6)(5,7)(4,7)(3,7)(3,6)(2,5)(3,4)(4,3)(4,2)
nitrogeno (5,6)(5,7)(4,7)(3,7)(2,6)(3,5)(3,4)(4,3)(4,2)
nitrogeno (5,6)(5,7)(4,7)(3,7)(2,6)(2,5)(3,4)(4,3)(4,2)
originario (6,9)(7,9)(7,8)(6,8)(5,7)(5,8)(4,9)(3,10)(2,10)(2,11)
originario (6,9)(7,9)(7,8)(6,8)(5,7)(5,8)(4,9)(3,10)(2,10)(1,9)
tricentenario (8,8)(7,9)(7,8)(6,7)(6,6)(5,6)(4,7)(4,8)(5,8)(4,9)(3,10)
              (2,10)(2,11)
tricentenario (8,8)(7,9)(7,8)(6,7)(6,6)(5,6)(4,7)(4,8)(5,8)(4,9)(3,10)
              (2,10)(1,9)
virgen (8,9)(7,10)(7,9)(6,8)(5,9)(6,10)
virgen (8,9)(7,10)(7,9)(6,8)(5,9)(5,8)
virgen (8,9)(7,8)(7,9)(6,8)(5,9)(6,10)
virgen (8,9)(7,8)(7,9)(6,8)(5,9)(5,8)
virgencita (8,9)(7,10)(7,9)(6,8)(5,9)(5,8)(6,7)(7,8)(8,8)(9,7)
virgencito (8,9)(7,10)(7,9)(6,8)(5,9)(5,8)(6,7)(7,8)(8,8)(8,7)
virgencito (8,9)(7,10)(7,9)(6,8)(5,9)(5,8)(6,7)(5,7)(4,7)(3,6)
virgencito (8,9)(7,8)(7,9)(6,8)(5,9)(5,8)(6,7)(5,7)(4,7)(3,6)
```

Hay muchas, pero muchas más, palabras contenidas en esta sopa.

1.2. Generando sopas de letras

Con el propósito de que puedas diseñar tus casos de prueba, se te provee un programa que genera sopas de letras. Su sintaxis es:

```
./gen-word-find nfilas mcols [seed]
```

En programa acepta como entrada las palabras que se desean poner en una matriz `nfilas` ×`ncols`. Las palabras son puestas en casillas elegidas al azar (según semilla `seed`). El orden de colocación es descendente según la longitud de la palabra.

La sopa de letras ejemplo fue generada mediante:

```
./gen-word-find 12 12 < words.txt
```

Donde `words.txt` contiene las palabras del ejemplo.

1.3. Esquema general de solución

Una manera de resolver el problema es mediante el uso de un árbol de prefijos que contenga un diccionario de palabras. Por cada casilla se revisa su letra y se busca en un árbol de prefijos. Si la letra existe, entonces se exploran recursivamente las casillas adyacentes, pero tomando como raíz prefija el nodo encontrado previamente. Por cada casilla que revise se prueba si el nodo actual es un fin de palabra.

1.4. Estructuras de datos

Para definir una sopa de letras emplearemos el siguiente tipo:

```
using Grid = DynMatrix<char>;
```

Con el fin de que recursivamente evitemos regresar a casillas ya exploradas, que contienen un prefijo, usaremos una matriz paralela que nos marcará las casillas que actualmente están explorándose. Tal matriz se denomina traza y se define del siguiente modo:

```
using Trace = DynMatrix<bool>;
```

Sea `trace` una matriz de este tipo. Una entrada `trace(i, j) == true` significa que la casilla `i,j` está siendo explorada y que ella es parte de un prefijo. De este modo, las casillas que en una búsqueda recursiva, en paralelo con el árbol de prefijos, hayan sido exploradas estarían marcadas como `true`. Las casillas adyacentes, con valor de traza `false`, son las casillas por donde se puede continuar la exploración.

Las celdas de la matriz se edulcoran sintácticamente mediante:

```
using Cell = tuple<int, int>;
```

Finalmente, vamos orientar a objetos la solución. Para ello se define la siguiente clase (con algunos de sus métodos):

```
struct WordFind
{
   WordFind(const Grid & g);

   Solution solve(istream & in);
};
```

El constructor recibe una matriz correspondiente a la sopa de letras y el método `solve()` la resuelve con las palabras obtenidas a través del stream `in`.

La intención de la orientación a objeto no es meramente de moda o modelizado, es práctica computacional: es más fácil y eficiente guardar el estado de cálculo dentro del estado del objeto. Esta técnica, aparte de que para algunos puede resultarles más fácil de entender, permite ahorrar pase de parámetros en la recursión.

Nota que al declarar `struct` todos los miembros de la clase `WordFind` devienen públicos. Aunque esto va contra algunos principios de ocultamiento de información, aquí es necesario tener acceso completo a la clase para que el evaluador pueda corregir tu laboratorio. Por consiguiente, **no privatices ningún miembro de la clase, porque si no privarás al evaluador de hacer su trabajo y tu programa no compilara**.

La solución se define mediante:

```
using Move = tuple<char, Cell>;

using WordMove = tuple<string, DynList<Cell>>;

using Solution = DynList<WordMove>;
}
```

`move` de tipo `Move` define una parte temporal de la solución, a saber: la letra `get<0>(move)` se encuentra en la celda `get<1>(move)`. Puede ser útil para construir una solución parcial una vez que se ha detectado una palabra.

`move` de tipo `WordMove` representa una solución parcial; la palabra `get<0>(move)` se encuentra en la secuencia de casillas `get<1>(move)`.

`sol` de tipo `Solution` representa la solución completa: una lista de pares palabra-secuencia-de-celdas.

El estado "sugerido" para la clase `WordFind` es:

- El árbol de prefijos, el cual puede llamarse `prefix_root`.
- La sopa de letras, la cual podríamos llamar `grid`.
- La traza de casillas visitadas, la cual podríamos llamar `trace`.

Eventualmente, puede convenirte como parte del objeto:

- `Solution sol`: acumulado de la solución. A modificarse cada vez que se descubra una palabra.

- Una pila que guarde el camino desde la raíz del árbol hasta el nodo actual. Nota que aunque la palabra está contenida en los nodos, también necesitas las coordenadas de sus letras. Por esta razón, lo recomendable es que esta pila sea de pares letra-celda (tipo `Move`) o nodo-celda.

2. Laboratorio

Tu trabajo en este laboratorio es resolver la sopa de letras. Para ello instrumentarás los métodos de la clase `WordFind`.

2.1. Constructor

Implementa el constructor:

```
WordFind(const Grid & g)
```

El cual recibe una sopa de letras `g` y prepara el estado interno para la solución.

Según el estado que uses, este es el lugar dónde inicializar el árbol de prefijos, la matriz interna de la sopa y, eventualmente, según hayan sido tus decisiones de diseño, la matriz traza y la pila.

Atención: `word-find.C` no compilará hasta que logres definir este constructor.

2.2. Movimientos legales

Instrumenta el siguiente método helper:

```
DynList<Cell> legal_moves(const int row, const int col)
```

Este método recibe como parámetro una celda actual que se está procesando de la matriz `grid` y retorna una lista de celdas adyacente y posibles por donde se pueden continuar una búsqueda o exploración.

Nota que si `legal_moves()` es invocado sobre una celda que está comenzando a explorarse, entonces todas las celdas adyacentes son válidas o legales. Sin embargo, si la celda es la continuación de una exploración, entonces las celdas previas que han sido vistas, es decir, que ya se han explorado y tienen asociado un nodo del árbol, no son válidas; por consiguiente no deberían de retornarse en este método. Aquí te puede ser útil usar la matriz `trace`, en cuyo caso, el resultado de este método son todas las celdas adyacentes menos aquellas que estén marcadas como `true` en `traces`.

Ten cuidado con los casos frontera, los cuales aquí no son estrictamente figurados sino literales, pues se corresponden con los bordes de la matriz.

2.3. Opcional: calcular la palabra de la pila

Cuando se detecten palabras, podría serte útil el siguiente método:

```
void extract_word_from_stack()
```

El cual examina la pila, calcula la palabra y la secuencia de casillas, y la adjunta a la lista solución.

Por supuesto, este método tiene sentido si usas una pila para guardar la secuencia de casillas explorada. Recuerda que lo sugerido es que empiles pares (letra,celda); de este modo, la pila contiene directamente la secuencia que caracteriza la palabra y la secuencia de casillas.

El manejo de la pila es responsabilidad de la búsqueda de palabras. Así que ten cuidado de no modificar la pila en este paso.

Este método es opcional; no será evaluado.

2.4. Búsqueda de palabra

Instrumenta:

```
void search(const int row, const int col, Cnode * root)
```

El cual busca todas las palabras que se pueden construir a partir de la celda `row`,`col` y desde el nodo del árbol `root`.

Probablemente esta sea la parte más compleja de este laboratorio. La estructura sugerida de este algoritmo es la siguiente:

1. Si `root` no tiene un hijo con el valor `grid(row,col)`, entonces retornar (no se hace nada).
2. De lo contrario
   - *a*) Si el hijo es fin de palabra, entonces `extract_word_from_stack()`
   - *b*) Para todo movimiento legal $\Longrightarrow$
     1) Recursionar con el movimiento legal y el hijo

Si usas una pila para guardar el camino desde la raíz hasta `root` entonces a la estructura anterior debes añadirle el manejo de la pila.

2.5. Solución

Programa:

```
Solution solve(istream & in)
```

El cual recibe un stream de palabras a buscar en la sopa de letras y retorna una lista ordenada lexicográficamente de pares palabra,lista-celdas.

Recuerda que el orden lexicográfico es por palabra y, si hay empate (son iguales), entonces se desempata por la lista de celdas. Análogamente el desempate de la lista de celdas es lexicográfico.

El stream `in` puede contener palabras con símbolos inválidos (números, símbolos especiales o errores de codificación). Si te encuentras este caso, entonces debes detectarlo e ignorarlo; es decir, una palabra con símbolos inválidos no se buscará. Ten cuidado con este requerimiento; las palabras válidas que vienen después de una inválida deben ser incluidas en la búsqueda.

El esquema general de este método es como sigue:

1. Cargar el árbol de prefijos con todas las palabras válidas del stream.
2. Por cada celda de la sopa, invocar a `search()`
3. Retornar la lista de palabras; eventualmente, según haya sido tu implementación, requerirás ordenarla.

En este laboratorio no se evaluará el desempeño. Sin embargo, vale la pena una meditación que plantea compromisos entre la facilidad de implementación y desempeño y que se presenta muy a menudo en el diseño de algoritmos y sistemas programados.

La lista `sol` podría ser un árbol binario de búsqueda. En este caso, cada inserción costaría $\mathcal{O}(\lg n)$, pero luego, cuando corresponda retornar el resultado, éste se obtendría en $\mathcal{O}(n)$, pues bastaría recorrer el árbol binario en infijo. Por el contrario, con una lista enlazada cada inserción de palabra será $\mathcal{O}(1)$, pero después para ordenarla requerirás pagar $\mathcal{O}(n \lg n)$. Aunque en ambos esquemas las complejidades son equivalentes, $n \times \mathcal{O}(\lg n) + \mathcal{O}(n)$ para los árboles binarios, y $n \times \mathcal{O}(1) + \mathcal{O}(n \lg n)$, es bastante probable que uno sea más rápido (en un factor constante) que el otro. Además, según la cosmovisión de programación de cada quien uno será más fácil de implementar que el otro.

Si se usase un árbol binario de búsqueda, entonces, ¿cuál tipo de árbol sería preferible emplear? ¿uno tradicional o uno que efectúe acciones para mantenerlo equilibrado?

2.6. Destructor

Implementa el destructor de `WordFind`.

El manejo de memoria será evaluado. Por consiguiente, debes liberar la memoria ocupada por tus estructuras de datos.

2.7. Desafíos (sin calificación)

Para tu enriquecimiento, cuando menos vale la pena que reflexiones acerca de estos dos desafíos.

2.7.1. Palabras lineales

En este laboratorio las palabras no son lineales. ¿Cómo se haría para sólo buscar palabras lineales?; es decir, estrictamente horizontales, verticales o diagonales.

2.7.2. Retorno a las casillas

¿Cómo se haría para buscar palabras con regreso? Por ejemplo, en el siguiente caso:

```
es
tx
```

Puede encontrarse la palabra “este”.

3. Evaluación

La fecha de entrega de este laboratorio es el lunes 24 de noviembre 2014 hasta la 6 pm. Puedes entregar antes, recibir corrección e intentar cuantas veces prefieras (limitado por supuesto a nuestra capacidad computacional).

Para evaluarte debes enviar el archivo `word-find.H` a la dirección:

`leandro.r.leon@gmail.com`

Antes de la fecha y hora de expiración. El "subject" debe **obligatoria y exclusivamente** contener el texto **PR3-LAB-05**. Si fallas con el subject entonces probablemente tu laboratorio no será evaluado, pues el mecanismo automatizado de filtrado no podrá detectar tu trabajo.

En el mensaje y en el inicio del fuente del programa debes colocar los nombres, apellidos y números de cédula de la pareja que somete el laboratorio. **Por favor, no uses otros medios para enviar la solución ni escribas otros comentarios adicionales en el email**.

La evaluación esta dividida en tres partes. La primera parte es ejecutada automáticamente por un programa evaluador compilado con tu fuente. El programa contiene casos de prueba y se coteja con una implementación de referencia. Para cada una de tus rutinas se ejecuta una serie de pruebas (aproximadamente 10) y se comparan con las pruebas y salidas de la implementación de referencia. La más mínima desavenencia acarrea penalidades. Por cada test que falle, se te reportará la entrada que causó la falla. La evaluación de la primera parte representa aproximadamente 90 % de la calificación.

Atención: si tu programa no compila, entonces el evaluador no compila. Si una de tus rutinas se cae, entonces el evaluador se cae. Si una de tus rutinas cae en un lazo infinito o demora demasiado, entonces el evaluador cae en un lazo infinito o se demora demasiado. Exactamente sucedería lo mismo si tu programa fuese parte de un sistema de aviónica: ¡se estrella el avión!. Por esa razón, en todos estos casos será imposible darte una nota, lo que simplemente se traduce en que tienes cero. Si una de tus rutinas dispara una excepción inesperada, el evaluador hará lo posible por capturarla y reportártela; sin embargo, esto no es un compromiso; podría estrellarse el avión. En todo caso, si el evaluador no logra capturar la excepción, no se te dará nota pero se te reportará la excepción.

No envíes programas que no compilan o se caen. Hacen perder los tiempos de red, de cpu, el tuyo y el mío. Si estás al tanto de que una rutina se te cae y no la puedes corregir, entonces trata de aislar la falla y disparar una excepción cuando ésta ocurra. De este modo no tumbarás al evaluador y eventualmente éste podría darte nota para algunos casos (o todos si tienes suerte). Si no logras aislar la falla, entonces deja la rutina vacía (tal como te fue dada). De este modo, otras rutinas podrán ser evaluadas.

La segunda parte es subjetiva y manual. Consiste en revisar tu fuente con la intención de evaluar tu estilo de codificación. Es recomendable que sigas las reglas de estilo comentadas en clase, pues estas armonizan con la persona que subjetivamente te evaluará. Esta parte aporta aproximadamente el 10 % de la nota.

La tercera y última parte es verificar si tu programa maneja correctamente la memoria. Para ello, se someterá a un test con `valgring`. Esta parte representa

aproximadamente el 10 %. Saber administrar bien y correctamente la memoria es una habilidad que debes desarrollar.

4. Recomendaciones

1. Lee enteramente este enunciado antes de proceder al diseño e implementación.

2. Desarrolla tus rutinas en el orden dado.

3. *"¡El tiempo no espera a nadie!"*[2]. Así que comienza este laboratorio lo más pronto posible. Aprovecha la evaluación automática y los múltiples intentos y somete tu solución cuando menos 48 horas antes del plazo de entrega. De este modo, tendrás tiempo para corregir y no meramente ser evaluado. Recuerda que una evaluación no tiene mucho sentido de aprendizaje si no hay corrección.

4. Esta práctica no tiene casos de prueba. Es a ti el diseñarlos. Sin embargo, se te proveen programas para generar sopas de letras y resolverlos. Construye caso de prueba rigurosos y completos. Ataca tu propio programa como si hubiese sido escrito por tu gran adversario. Esta es una actitud esencial para un programador de élite; aunque ciertamente es difícil de cultivar, pues en cierto modo es un ataque a uno mismo.

5. Usa el foro para plantear tus dudas de comprensión. Pero de ninguna manera compartas código, pues es considerado **plagio**.

   Por favor, **no uses el correo electrónico ni el correo del facebook para plantear tus dudas. ¡Usa el foro!**

6. No incluyas headers en tu archivo `word-find.H` (algo como `# include ...`), pues puedes hacer fallar la compilación. Si requieres un header especial, el cual piensas no estaría dentro del evaluador, exprésalo en el foro para así incluirlo.

[2] Maquiavelo.