$Date : 2003-11-1017:45:00+01$ $Revision : 1.3$

Ejercicio Final de PS
Normativa y Enunciado

Otoño de 2003

Este documento es largo pero es imprescindible que lo leáis íntegramente y con detenimiento, incluso si sois repetidores, pues se dan las instrucciones y normas que hay que seguir para que el ejercicio final sea evaluado positivamente. El profesorado de la asignatura dará por hecho que todos los alumnos conocen el contenido íntegro de este documento.

Contenidos:

1. Normativa

1. Tal y como se dice en la Guía Docente, para cubrir los objetivos de la asignatura se considera imprescindible el desarrollo por parte del estudiante de un ejercicio final que requiere horas adicionales de trabajo personal, aparte de las clases de laboratorio, donde se realizan los otros ejercicios prácticos que os permiten familiarizaros con el entorno de trabajo y el lenguaje de programación C++.

2. El ejercicio final se realizará en equipos de dos estudiantes. Si uno abandona, uno de los integrantes debe notificarlo a la mayor brevedad via e-mail a `jimenez@lsi.upc.edu`, y continuar el ejercicio en solitario, eventualmente. Por otro lado, solamente se permitirán equipos individuales en casos excepcionales donde se pruebe la imposibilidad de reunión o comunicación con otros estudiantes mediante algún tipo de justificante. Consultad el apartado *Equipos de Prácticas* en las páginas Web de la asignatura (`http://www.lsi.upc.edu/~ps`) donde se dan las instrucciones sobre la formación de equipos.

3. El soporte que utilizaréis para los ejercicios es el lenguaje de programación C++ (específicamente el compilador GNU `g++-4.1.2`) sobre el entorno Linux del LCFIB. Esto no impide el desarrollo previo en PCs o similares. De hecho, existen compiladores de C++ para toda clase de plataformas y debería resultar sencillo el traslado desde vuestro equipo particular al entorno del LCFIB, especialmente si trabajáis con Linux en vuestro PC.

   Atención: Existe la posibilidad de pequeñas incompatibilidades entre algunos compiladores de C++. En cualquier caso es imprescindible realizar al menos una comprobación final de que el programa desarrollado en PC o similar funciona efectivamente sobre el entorno Linux del LCFIB.

4. El ejercicio final será evaluado mediante:

   - su ejecución en el entorno del LCFIB sobre una serie de *juegos de prueba* y
   - la corrección del diseño, implementación y documentación: las decisiones de diseño y su justificación, la eficiencia de los algoritmos y estructuras de datos, la legibilidad, robustez y estilo de programación, etc. Toda la documentación debe acompañar al código; no tenéis que entregar ninguna documentación en papel.

   Existen dos tipos de juegos de pruebas: públicos y privados. Los juegos de pruebas públicos se pondrán a vuestra disposición con antelación suficiente en las páginas Web de la asignatura (`http://www.lsi.upc.edu/~ps`). La nota del ejercicio final se calcula a partir de la nota de ejecución (E) y la nota de diseño (D). La nota de este ejercicio final es:

   $$P = 0.3E + 0.7D,$$

   si ambas notas parciales (E y D) son mayores que 0; $P = 0$ si la nota de diseño es 0.

El documento *Guía y Normas de Programación* describe, entre otras cosas, las situaciones que dan origen a una calificación de 0 en el diseño (y por ello una calificación de 0 en el ejercicio). La nota de ejecución (E) es 3 puntos como mínimo si se superan los juegos de prueba públicos; en caso contrario, la nota es 0. Los juegos de prueba privados aportan hasta 7 puntos más, en caso de haberse superado los juegos de prueba públicos.

5. La fecha límite de la entrega final es el 12 de Diciembre de 2003 a las 12 del mediodía. Si un equipo no ha entregado su ejercicio final entonces su calificación será 0. En las páginas Web de la asignatura (`http://www.lsi.upc.edu/~ps`) se darán más detalles sobre el procedimiento de entrega del ejercicio final.

6. No subestiméis el tiempo que debéis dedicar a cada uno de los aspectos del ejercicio: diseño, codificación, depuración de errores, pruebas, documentación, ...

2. Enunciado de la práctica

En esta práctica habréis de construir una serie de módulos y clases relacionadas con la construcción y resolución de sopas de letras. A continuación se describe el funcionamiento de la función de resolución y el de la generación.

La función de resolución recibe un conjunto de palabras (denominado *diccionario*) y una sopa de letras, y ha de encontrar todas las palabras de cuatro o más letras que aparecen en la sopa de letras.

Por ejemplo, suponed que nuestro diccionario contiene todas las palabras de 4 o más letras que aparecen en un diccionario de bolsillo convencional y que la sopa de letras que nos dan es[1]:

```
M  E  L  O  N  Q  P  E
A  A  T  C  D  O  U  I
A  J  N  A  R  A  N  E
E  O  A  Z  E  R  E  C
I  I  N  U  A  A  L  A
S  M  T  A  O  N  I  G
O  R  E  N  T  D  A  L
P  B  F  H  N  A  G  T
S  U  J  A  C  N  L  V
R  A  S  I  Y  O  E  P
```

Vuestra función habrá de devolver una lista de las palabras que ha encontrado en la sopa de letras, ordenada alfabéticamente, e indicando para cada palabra hallada, la fila y columna de la primera letra (de la palabra) y la orientación. Puesto que una misma palabra puede aparecer repetidas veces en la sopa de letras, las repeticiones deberán aparecer por orden lexicográfico de posición, y en caso de empate, por orden de orientación. El orden de orientación es el inducido por la lista que se da más abajo ($\mathrm{H} < \mathrm{HR} < \ldots < \mathrm{B} < \mathrm{BR}$).

La orientación se representará mediante uno o dos caracteres de la siguiente manera:

- H = horizontal, de izquierda a derecha;
- HR = horizontal, de derecha a izquierda;
- V = vertical, de arriba a abajo;
- VR = vertical, de abajo a arriba;
- D = diagonal de arriba-izquierda a abajo-derecha;
- DR = diagonal de abajo-derecha a arriba-izquierda;
- B = diagonal de abajo-izquierda a arriba-derecha;

[1] Se explicará más adelante cuál es el formato con el que se introducirán las palabras que forman el diccionario y el formato para introducir la sopa de letras.

- BR = diagonal de arriba-derecha a abajo-izquierda.

En el ejemplo que dábamos, la función generará la siguiente lista, suponiendo que todas estas palabras figuran en el diccionario:

```
ARANDANO 3 6 V
CEREZA 4 8 HR
LATA 9 7 DR
MANZANA 1 1 D
MELON 1 1 H
NARANJA 3 7 HR
NATA 5 3 D
PLATANO 10 8 DR
SANDIA 10 3 B
```

En el caso de que aparezcan varias palabras en la sopa de letras con la misma posición de origen y orientación, solamente se listará la más larga. Por otra parte, dos palabras pueden tener un solapamiento de más de una letra. Por ejemplo, para la sopa de letras que aparece a continuación, en la parte izquierda, el resultado ha de ser la lista que aparece a la derecha, suponiendo que todas las palabras figuran en el diccionario (en la sopa de letras se han omitido las letras que no son relevantes para el ejemplo).

```
  |1 2 3 4 5 6 7
--+-------------
 1|      P E R A
 2|      E
 3|  P   R
 4|P E R A
 5|  R
 6|C A S A S
 7|  L A
 8|    C
 9|    O
10|    S
11|    A
```

```
ACOSA 7 3 V
ASAS 6 2 H
CASAS 6 1 H
COSA 8 3 V
OCAS 9 3 VR
PERA 1 4 H
PERA 1 4 V
PERA 4 1 H
PERAL 3 2 V
SACOS 6 3 V
```

En cambio no aparecerán ni `CASA` ni `SACO` por compartir origen y orientación con palabras más largas. Tampoco aparecen las palabras de longitud inferior a 4, como por ejemplo, `ASA`, `ASO`, `ERA`, `OCA` y `OSA`.

La función de generación o construcción de una sopa de letras recibe un diccionario y una lista de los huecos donde se han de colocar las palabras de una sopa de letras. Un hueco viene descrito por una posición de inicio (fila y columna), una orientación y una longitud. Esta función debe decir si existe o no al menos una palabra de ese diccionario que rellene completamente cada uno de los huecos. Si existe, la función devolverá la lista de palabras encontradas con los huecos en los que encajan.

Por ejemplo, si tenemos un diccionario con las palabras:

```
CALABAZA
CARA
CARAMELO
HELADO
PEPINO
PIMIENTO
PINO
```

y una sopa de letras con dos huecos: uno vertical de 6 letras en <1 2 V 6> y otro horizontal de 8 letras en <2 1 H 8> representados en la siguiente sopa:

```
huecos                  |1 2 3 4 5 6 7 8
-------                --+---------------
1 2 V 6                 1|  _
2 1 H 8                 2|_ _ _ _ _ _ _ _
                        3|  _
                        4|  _
                        5|  _
                        6|  _
                        7|
```

hay que encontrar las palabras del diccionario que rellenen los huecos. Tenemos que los dos huecos descritos comparten la segunda posición. En el diccionario dado todas las palabras de longitud 6 tienen una `E` en la segunda posición y las de longitud 8 tienen o una `A` o una `I`. Como no hay ninguna letra en común entre las de longitud 6 y las de 8 en la posición requerida, obviamente no hay solución y la función generará una lista vacía.

Si para el mismo diccionario nos dan los mismos huecos pero en posiciones distintas:

```
huecos                  |1 2 3 4 5 6 7 8
-------                --+---------------
1 2 V 6                 1|  _
4 1 H 8                 2|  _
                        3|  _
                        4|_ _ _ _ _ _ _ _
                        5|  _
                        6|  _
                        7|
```

resulta que ahora la letra que tienen que tener en común es la segunda letra de una palabra de longitud 8 (`{A, I}`) y la cuarta letra de una palabra de longitud 6 (`{A, I}`). En esta ocasión sí que podemos encontrar una palabra de 8 letras y una de 6 que compartan una letra en la posición mencionada. En realidad, podemos generar todas las siguientes sopas de letras:

```
  |1 2 3 4 5 6 7 8        |1 2 3 4 5 6 7 8        |1 2 3 4 5 6 7 8
--+---------------      --+---------------      --+---------------
 1|  H                   1|  H                   1|  P
 2|  E                   2|  E                   2|  E
 3|  L                   3|  L                   3|  P
 4|C A L A B A Z A       4|C A R A M E L O       4|P I M I E N T O
 5|  D                   5|  D                   5|  N
 6|  O                   6|  O                   6|  O
 7|                      7|                      7|
```

La función generará en forma de lista **una** de las tres soluciones posibles:

```
CARAMELO 4 1 H          CALABAZA 4 1 H          PEPINO 1 2 V
HELADO 1 2 V            HELADO 1 2 V            PIMIENTO 4 1 H
```

La lista está ordenada siguiendo los mismos criterios que usa la función de resolución.

Como la idea es generar una sopa de letras que sirva de entrada a la función de resolución, solamente quedará por rellenar el resto de los espacios de la sopa con letras al azar [2].

[2]Se ha relajado la restricción del relleno de espacios (Antes: "Estos espacios vacíos se han de rellenar de forma que se garantice que al resolver la sopa aparezcan como mínimo todas las palabras que ha producido la función de generación y en las mismas posiciones.", Ahora: "el resto de los espacios de la sopa con letras al azar") para que la función de generación os resulte más fácil de implementar.

3. Diseño modular

El diseño modular (Figura 1) muestra los módulos que intervienen en la práctica. Se han omitido las clases error, mem_din y el módulo util de la librería `libeda` por claridad, ya que muchas clases y módulos del subgrafo usan dichos módulos. Las clases error y mem_din están documentadas en la *Guía y Normas de Programación*. El módulo util está documentado *on-line* en `<eda/util>`.

En todos los módulos o clases de esta práctica se puede usar también la clase string de la librería estándar de C++.

Recordad que no se puede utilizar ninguna clase de una librería externa en uno de vuestros módulos o clases, excepto si en esta documentación se indica lo contrario.

En todas las clases hay que implementar los métodos de construcción por copia, asignación y destrucción ante la posibilidad de que uséis memoria dinámica para implementar la clase en cuestión. Si no se hace uso de memoria dinámica, la implementación de estos tres métodos es muy sencilla pues bastará imitar el comportamiento de lo que serían los correspondientes métodos de oficio (la destructora no hace "nada", y las otras hacen copias atributo por atributo).

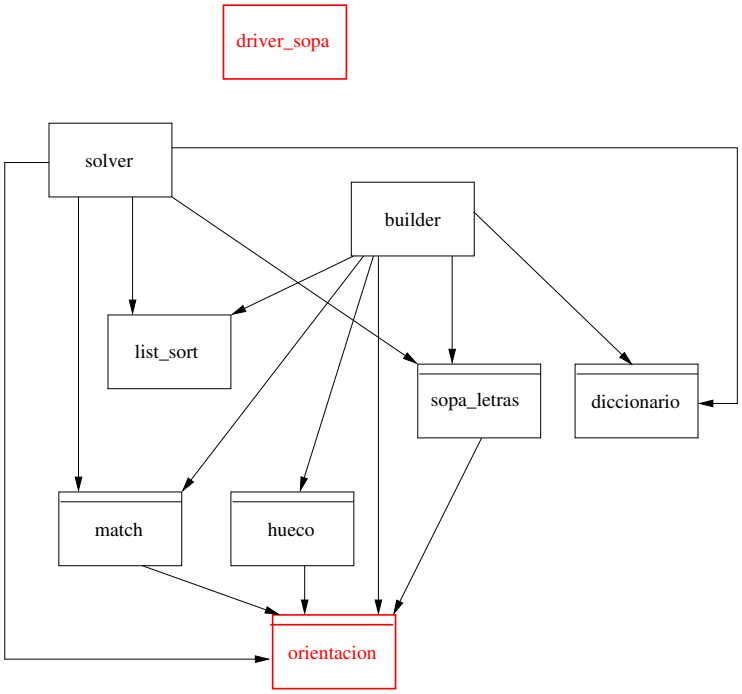

Figura 1: Diseño modular de la práctica

Describimos a continuación qué habréis de hacer en esta práctica. Esencialmente, vuestra tarea será:

- Implementar la clase sopa_letras, una matriz de letras donde aparentemente la secuencialidad de dichas letras en cualquier orientación no representa ninguna palabra.

- Diseñar e implementar la clase diccionario que permitirá almacenar palabras y decidir de manera eficiente si un string es o no una palabra válida (es decir, figura entre las palabras almacenadas).
- Implementar el módulo solver que ofrece una función para la resolución de una sopa de letras dada, y un módulo builder que nos proporciona la función de generación de sopas de letras.
- Implementar las clases hueco y match y una función de ordenación en el módulo list_sort.

4. La clase orientacion

```
#ifndef _ORIENTACION_HPP
#define _ORIENTACION_HPP

#include <string>
```

Esta clase es "trivial" y se os proporciona ya definida. Una variable del tipo `orientacion` puede tomar cualquiera de los valores que aparecen listados en la definición del tipo mediante `enum`. El valor `orientacion::NO_OR` sirve como valor ficticio. El operador de incremento aplicado sobre una variable del tipo hace que ésta tome el valor siguiente en la lista, salvo que su valor fuese `NO_OR`, en cuyo caso no se modifica. Los operadores relacionales (==, <=, ...) pueden usarse entre los valores del tipo (`H` es el menor valor y `NO_OR` el mayor). También puede usarse la asignación entre objetos de la clase en cualquier momento que se necesite. La constructora que recibe un `string` como argumento nos permite crear una orientación a partir de su nombre. Es útil para leer una orientación desde el canal de entrada. Finalmente, `tostring` nos devuelve la representación de un valor en un string lo que permite imprimirlo convenientemente.

Ejemplos de uso:

```
for(orientacion mior = orientacion::H;
    mior != orientacion::NO_OR;       ++mior)
      cout << mior.tostring() << " ";
cout << "Introduce una orientacion:";
string onom;
cin >> onom; orientacion mior(onom);
```

```
class orientacion {
public:
  enum {H, HR, V, VR, D, DR, B, BR, NO_OR};

  orientacion(int n = NO_OR) throw() : _or(n) {}

  orientacion(const string& s) throw() {
    if (s == "H") { _or = H; return; }
    if (s == "HR") { _or = HR; return; }
    if (s == "V") { _or = V; return; }
    if (s == "VR") { _or = VR; return; }
    if (s == "D") { _or = D; return; }
    if (s == "DR") { _or = DR; return; }
    if (s == "B") { _or = B; return; }
    if (s == "BR") { _or = BR; return; }
    _or = NO_OR;
  }

  orientacion& operator++() throw() {
    _or = (_or == NO_OR) ? NO_OR : _or + 1;
    return *this;
```

```
  }

  operator int() const throw() {
    return _or;
  }

  string tostring() const throw() {
    switch (_or) {
    case H : return "H";
    case HR : return "HR";
    case V : return "V";
    case VR : return "VR";
    case D : return "D";
    case DR : return "DR";
    case B : return "B";
    case BR : return "BR";
    default : return "NO_OR";
    }
  }

private:
  int _or;

};
#endif
```

5. La clase match

```
#ifndef _MATCH_HPP
#define _MATCH_HPP

#include <string>
#include <eda/error>
#include <eda/util>

#include "orientacion.hpp"
using util::nat;
```

Un objeto de la clase `match` es una cuádrupla $\langle palabra, fila, columna, orientacion\rangle$; la clase proporciona operaciones elementales para consultar la información y realizar comparaciones (<).

```
class match {

public:
  explicit match(const string& pal = "", nat fil = 0, nat col = 0,
                 orientacion o = orientacion::H) throw(error);
```

Constructora por copia, asignación y destructora.

```
  match(const match& s) throw(error);
  match& operator=(const match& s) throw(error);
  ~match() throw();
```

Las cuatro consultoras permiten examinar las componentes de un match.

```
  string palabra() const throw();
  nat fila() const throw();
  nat columna() const throw();
  orientacion orient() const throw();
```

$s < t$ devuelve cierto si el match s es menor que el match t; la comparación se basa en primer lugar en el orden ascendente alfabético de las palabras; en caso de que sea la misma palabra el orden lo dicta la posición de origen, el par $\langle fila, columna\rangle$; por último se tendrá en cuenta la orientación cuando las dos posiciones de origen son iguales. Dadas dos posiciones $\langle i, j\rangle$ y $\langle i', j'\rangle$, $\langle i, j\rangle < \langle i', j'\rangle$ si $i < i'$ ó $i = i' \wedge j < j'$.

```
  bool operator<(const match& t) const throw();

private:
  #include "match.rep"
```

```
};
#endif
```

6. La clase hueco

```
#ifndef _HUECO_HPP
#define _HUECO_HPP

#include <eda/error>
#include <eda/util>

#include "orientacion.hpp"
using util::nat;
```

Un objeto de la clase `hueco` es una cuádrupla $\langle fila, columna, orientacion, longitud\rangle$; la clase proporciona operaciones elementales para consultar la información y realizar comparaciones (<).

```
class hueco {

public:
  explicit hueco(nat fil = 0, nat col = 0,
                 orientacion o = orientacion::H, nat lon = 0) throw(error);
```

Constructora por copia, asignación y destructora.

```
  hueco(const hueco& h) throw(error);
  hueco& operator=(const hueco& h) throw(error);
  ~hueco() throw();
```

Las cuatro consultoras permiten examinar las componentes de una solución.

```
  nat fila() const throw();
  nat columna() const throw();
  orientacion orient() const throw();
  nat longitud() const throw();
```

$s < t$ devuelve cierto si el hueco s es menor que el hueco t; la comparación se basa en primer lugar en la posición, el par $\langle fila, columna\rangle$; si ambos huecos tienen la misma posición de origen, el orden lo dictan las orientaciones; en caso de empate se usará la longitud. Dadas dos posiciones $\langle i, j\rangle$ y $\langle i', j'\rangle$, $\langle i, j\rangle < \langle i', j'\rangle$ si $i < i'$ ó $i = i' \wedge j < j'$.

```
  bool operator<(const hueco& t) const throw();

private:
  #include "hueco.rep"
};
#endif
```

7. La clase sopa_letras

```
#ifndef _SOPA_LETRAS_HPP
#define _SOPA_LETRAS_HPP

#include <string>
#include <eda/error>
#include <eda/util>

#include "orientacion.hpp"
using util::nat;
```

Una `sopa_letras` almacena una matriz de $M \times N$ caracteres. Todos los caracteres son asteriscos ('*') o letras mayúsculas de la A a la Z, excluídas la Ñ, la Ç y las vocales acentuadas [a]. El número de filas y el de columnas están acotados, respectivamente, por las constantes `MAXFILAS` y `MAXCOLUMNAS` que se definen en la clase.

Para facilitar la implementación, las filas se indexarán de 0 a $M-1$ y las columnas se indexarán de 0 a $N-1$ (nótese que en los ejemplos del enunciado se supone, por el contrario, que los índices van de 1 en adelante).

Si la sopa de letras se ha construído sin especificar el número de filas y columnas (usando la constructora por defecto) entonces éstos vienen determinados por la operación `inserta_fila`. El número de columnas nos lo da la longitud del primer string insertado en la sopa de letras S y el número de filas es el número de strings insertados en S hasta el momento. Si no se han insertado filas entonces tanto el número de filas como el de columnas es 0.

Implementación: La representación de esta clase se encontrará en el fichero `sopa_letras.rep` y la implementación en el fichero `sopa_letras.cpp`.

[a] Dicho de otro modo, sólo contiene caracteres cuyos respectivos códigos ASCII están entre 65 ('A') y 90 ('Z').

```
class sopa_letras {

public:
```

Construye una `sopa_letras` vacía, sin filas ni columnas.

```
  sopa_letras() throw(error);
```

Construye una `sopa_letras` vacía, con num_fil filas y num_col columnas. Da error si $num\_fil >$ `MAXFILAS` o $num\_col >$ `MAXCOLUMNAS`. Todas las posiciones válidas contienen asteriscos.

```
  sopa_letras(nat num_fil, nat num_col) throw(error);
```

Constructora por copia, asignación y destructora.

```
  sopa_letras(const sopa_letras& S) throw(error);
```

```
sopa_letras& operator=(const sopa_letras& S) throw(error);
~sopa_letras() throw();
```

Añade los caracteres del string s como última fila de la `sopa_letras`. Si la sopa de letras ha sido construída sin especificar el número de filas y de columnas entonces se produce un error si la longitud del string es diferente de la de las filas precedentes (si existen), si es la primera y la longitud es excesiva (demasiadas columnas) o si la `sopa_letras` ya tenía el máximo número de filas posible. Por otro lado, si la sopa de letras se ha construído especificando el número de filas y de columnas, se produce un error si la longitud del string no coincide con el número de columnas especificado o si la sopa de letras ya tenía el número de filas especificado.

```
void inserta_fila(string s) throw(error);
```

Devuelve la cadena de caracteres (string) de longitud lon que comienza en la posición (fil, col) en la orientación $orient$; se produce un error si no existe un string de esa longitud partiendo del origen y en la orientación dados.

```
string cadena(nat fil, nat col, nat lon,
              orientacion orient) const throw(error);
```

Añade los caracteres del string s a partir de la posición de origen dada por la fila fil y la columna col, en la `orientacion` $orient$ dada; se produce un error si el string s no cabe en la sopa de letras (con las dimensiones que ésta tenga en ese momento) colocándolo en la posición y orientación dadas.

```
void inserta_cadena(string s, nat fil, nat col,
                            orientacion orient) throw(error);
```

Devuelven el número de filas y columnas que la `sopa_letras` tiene en ese momento, respectivamente.

```
nat num_filas() const throw();
nat num_columnas() const throw();
```

Número de filas y de columnas máximos.

```
static const nat MAXFILAS = 80;
static const nat MAXCOLUMNAS = 80;
```

Gestión de errores e información interna.

```
static const char nom_mod[] = "sopa_letras";

static const int NumeroFilasIncorrecto  = 11;
static const int LongitudFilaIncorrecta = 12;
static const int StringNoExiste         = 13;
static const int StringNoCabe           = 14;
```

```
  static const char MsgNumFilasIncorr[] = "Numero de filas incorrecto.";
  static const char MsgLongFilaIncorr[] = "Longitud de la fila incorrecta.";
  static const char MsgStringNoExiste[] = "No existe el string";
  static const char MsgStringNoCabe[]   = "El string no cabe";

private:
  #include "sopa_letras.rep"
};
#endif
```

8. La clase diccionario

```
#ifndef _DICCIONARIO_HPP
#define _DICCIONARIO_HPP

#include <string>
#include <list>
#include <eda/error>
#include <eda/util>
```

Decisiones sobre los datos: Todas las palabras del diccionario estarán constituidas exclusivamente por letras mayúsculas[a], de la A a la Z. El número de palabras que almacenará un diccionario no es conocido ni se puede hacer ninguna estimación razonable sobre su valor. Puede haber diccionarios pequeños, medianos, grandes y gigantescos.

Comentarios y ejemplos sobre la especificación

La especificación de las operaciones constructoras es sencilla y no parecen requerir más explicaciones. Sin embargo, es conveniente hacer algunos comentarios y dar ejemplos que ayuden a clarificar la especificación de las operaciones `prefijo` y `busca_patron`.

Primero, si la palabra dada p está en el diccionario entonces la operación `prefijo` nos devuelve la palabra entera. Y a la inversa, si `prefijo` no nos devuelve la palabra dada, entonces dicha palabra no forma parte del diccionario.

Si p no es una palabra en el diccionario ni tampoco lo es ninguno de sus prefijos, entonces la operación `prefijo` devolverá el string vacío (que es siempre una palabra del diccionario). Supongamos que p = `PODARFOLGZ`. Si el diccionario contiene palabras válidas en castellano entonces la operación `prefijo` nos devolverá `PODAR`, salvo que este verbo no figurase en el diccionario, en cuyo caso, devolvería `PODA`. Si tampoco figurase ésta última en el diccionario, lo más probable es que se devolviese el string vacío pues, en principio, ni `POD`, ni `PO` ni `P` son palabras válidas.

Ahora bien, el diccionario puede contener palabras cualesquiera y el hecho de que estas palabras sean o no válidas en un cierto idioma es completamente irrelevante desde el punto de vista del diseño e implementación de la clase diccionario. Si por cualquier razón hubiéramos añadido el string `PODARFO` a nuestro `diccionario`, entonces `prefijo`("`PODARFOLGZ`") devolvería `PODARFO`. De manera análoga si el diccionario contuviese el string `POD` (pero no las palabras `PODA` o `PODAR` ni ningun string más largo) entonces `prefijo`("`PODARFOLGZ`") devolvería `POD` aunque este string tiene longitud inferior a 4.

[a] Sin incluir ni Ñ ni Ç ni vocales acentuadas; en definitiva, sólo las letras mayúsculas con códigos ASCII entre 65 y 90.

La operación `satisfacen_patron` recibe un patrón a través de un string y devuelve una lista ordenada con las palabras del diccionario que satisfacen dicho patrón. Un patrón consta de letras mayúsculas y asteriscos; cuando en una determinada posición aparece un asterisco, eso significa que queremos todas las palabras que tengan las letras que sí están especificadas en el patrón pero que nos da igual que letra tenga en el lugar al que corresponde el asterisco. Por ejemplo, para el patrón `A*T**` el resultado podría ser [`ACTUA`, `ALTAR`, `ALTAS`, `ASTRO`, `AUTOR`] (obviamente, dependerá de las palabras que contenga el diccionario). Por otro lado, el patrón `*****` nos devolverá todas las palabras de longitud 5 y el patrón `ASTRO` devolverá una lista conteniendo esta palabra si dicha palabra estuviera presente. En general, la operación devolverá una lista vacía si en el diccionario no hay ninguna palabra que satisfaga el patrón dado.

Implementación: La representación de esta clase se encontrará en el fichero `dicccionario.rep` y la implementación en el fichero `diccionario.cpp`.

```
class diccionario {

public:
```

Construye un diccionario que contiene únicamente una palabra: la palabra vacía

```
  diccionario() throw(error);
```

Constructora por copia, asignación y destructora.

```
  diccionario(const diccionario& D) throw(error);
  diccionario& operator=(const diccionario& D) throw(error);
  ~diccionario() throw();
```

Añade la palabra p al diccionario; si la palabra p ya formaba parte del diccionario, la operación no tiene efecto alguno.

```
  void inserta(const string& p) throw(error);
```

Devuelve el prefijo más largo de p que es una palabra perteneciente al diccionario, o dicho de otro modo, la palabra más larga del diccionario que es prefijo de p.

```
  string prefijo(const string& p) const throw(error);
```

Devuelve la lista de palabras del diccionario que satisfacen el patrón especificado en el string s, en orden alfabético ascendente.

```
  void satisfacen_patron(const string& s,
                         list<string>& L) const throw(error);
```

Gestión de errores e información interna.

```
  static const char nom_mod[] = "diccionario";
```

```
private:
  #include "diccionario.rep"
};
#endif
```

9. El módulo solver

```
#ifndef _SOLVER_HPP
#define _SOLVER_HPP

#include <list>
#include <eda/error>
#include <eda/util>

#include "match.hpp"
#include "sopa_letras.hpp"
#include "diccionario.hpp"
```

Este módulo define el *namespace* `solver` que incluye la función `solve`. Esta función recibe una `sopa_letras` y un `diccionario` y resuelve la sopa de letras, devolviendo una lista con el resultado. Consultad el enunciado para una descripción detallada del comportamiento de esta función.

Implementación: La implementación de este módulo se encontrará en el fichero `solver.cpp`.

```
namespace solver {
  void solve(const sopa_letras& S, const diccionario& D,
             list<match>& L) throw(error);
};
#endif
```

10. El módulo builder

```
#ifndef _BUILDER_HPP
#define _BUILDER_HPP

#include <list>
#include <eda/error>
#include <eda/util>

#include "match.hpp"
#include "hueco.hpp"
#include "sopa_letras.hpp"
#include "diccionario.hpp"
```

Este módulo define el *namespace* `builder` que incluye la función `build`. Esta función recibe un `diccionario`, una lista de `huecos` y una `sopa_letras` de dimensiones $M \times N$. Rellena la sopa de letras dada y devuelve una lista con las palabras que se han usado para rellenar los huecos. Lanza un error si los huecos no pueden disponerse correctamente en la `sopa_letras`. Se asume que no hay dos huecos que compartan posición de origen y orientación. Consultad el enunciado para una descripción detallada del comportamiento de esta función.

Si no existe ninguna solución para rellenar los huecos, la lista L será vacía y la sopa S se deja intacta. Una vez que se han encontrado palabras en el diccionario con las cuales rellenar los huecos, se rellenará el resto de la sopa de letras con letras al azar, pero sin modificar la solución construída. Aunque por lo general será improbable, puede suceder que la resolución de la sopa generada no contenga las mismas palabras con las que se han rellenado los huecos (se "amplian" o se "fusionan" palabras al introducir letras al azar).

Implementación: La implementación de este módulo se encontrará en el fichero `builder.cpp`.

```
namespace builder {
  void build(const diccionario& D, const list<hueco>& Huecos,
             list<match>& L,
             sopa_letras& S) throw(error);
```

Gestión de errores e información interna.

```
  const char nom_mod[] = "builder";
  const int HuecosNoColocables = 21;
  const char MsgHuecosNoColocables[] = "No se pueden colocar huecos";
};
#endif
```

11. El módulo list_sort

```
#ifndef _LIST_SORT_HPP
#define _LIST_SORT_HPP

#include <list>
```

Este módulo define una función genérica `sort` que ordena en orden ascendente una lista (`list`) cuyos elementos son del tipo `T`; se asume que el tipo `T` dispone de una constructora por defecto, constructora por copia, asignación y que está definido `operator<` entre dos objetos del tipo `T`. Naturalmente NO puede usarse ninguna de las funciones de ordenación definidas en la clase `list` de la STL.

```
namespace list_sort {
  template <typename T>
  void sort(list<T>& L) throw();
};
#include "list_sort.t"
#endif
```

12. Documentación

Los ficheros deben estar debidamente documentados. Es muy importante describir con detalle y precisión la representación elegida, justificando dicha elección, así como las operaciones de cada clase. En el caso de los ficheros `.rep` y `.cpp` es especialmente importante explicar en detalle las representaciones y los motivos de su elección frente a posibles alternativas, y los algoritmos empleados. El coste en tiempo y en espacio es frecuentemente el criterio determinante en la elección, por lo cual deberán detallarse estos costes en la justificación, siempre que ello sea posible, para cada alternativa considerada y para la opción finalmente elegida.

Una vez enviados los ficheros por vía electrónica, dichos ficheros serán impresos para su evaluación. No tenéis que imprimirlos vosotros. No tenéis que entregar ninguna otra documentación. A fin de unificar el aspecto visual del código utilizamos una herramienta de *prettyprinting* denominada `astyle`. Podéis comprobar los resultados que produce el *prettyprinter* mediante el comando

```
$ astyle --style=kr -s2 < fichero.cpp > fichero.formateado
```

y a continuación se puede visualizar mediante

```
$ a2ps fichero.formateado -o - | gv -
```

o imprimir el fichero mediante

```
$ a2ps -Pnombre_impresora fichero.formateado
```

Alternativamente, se puede generar un fichero postscript:

```
$ a2ps fichero.formateado -o fichero.ps
$ gv fichero.ps # para visualizarlo
$ lpr -Pnombre_impresora fichero.ps # para imprimirlo
```