Guion de prácticas

Práctica Final

Abril de 2018

DECSAI

Metodología de la Programación

Curso 2017/2018

Contents

1 Introducción

Los objetivos de este guion de prácticas son los siguientes:

1. Practicar con un problema en el que es necesaria la modularización. Para desarrollar los programas de esta práctica, el alumno debe crear distintos archivos, compilarlos, y enlazarlos para obtener los ejecutables.
2. Practicar con el uso de la memoria dinámica. El alumno deberá usar estructuras de datos que se alojan en memoria dinámica.
3. Profundizar en los conceptos relacionados con la abstracción de tipos de datos.
4. Practicar con el uso de clases como herramienta para implementar los tipos de datos donde se requiera encapsular la representación.
5. Usar los tipos que ofrece C++ para el uso de ficheros.

Los requisitos para poder realizar esta práctica son:

1. Saber manejar punteros, memoria dinámica, clases y ficheros.
2. Conocer el diseño de programas en módulos independientes, así como la compilación separada, incluyendo la creación de bibliotecas y de archivos makefile.
3. El alumno debe realizar esta práctica una vez que haya asimilado los conceptos básicos sobre clases, incluyendo constructores, destructores y sobrecarga del operadores.
   Podrá observar que parte del contenido de esta práctica coincide con prácticas anteriores. Por lo que podrá reutilizar material ya elaborado y depurado, así el tiempo para la elaboración de la práctica final se verá reducido de forma significativa.

2 Problemas a resolver

En esta práctica vamos a desarrollar un conjunto de aplicaciones sobre ficheros de texto que nos permitan averiguar automáticamente el idioma en el que está escrito un determinado texto. El idioma de un texto puede averiguarse considerando las frecuencias de los pares de caracteres consecutivos, bigramas[1] y compararlos con los bigramas de textos cuyos idiomas son conocidos. Así, el bigrama “th” altamente frecuente, es un buen indicativo de que el idioma es inglés, o “la” de que es español. Más adelante daremos más detalles de cómo llevarlo a cabo (ver Sección 4).

Para realizar la práctica se deben implementar dos programas (aplicaciones) que se podrán ejecutar de forma independiente.

[1] O bien, podrían considerarse la frecuencia de simples caracteres, o bien de ternas de caracteres, trigramas... ngramas, lo que daría una mayor precisión a la predicción.

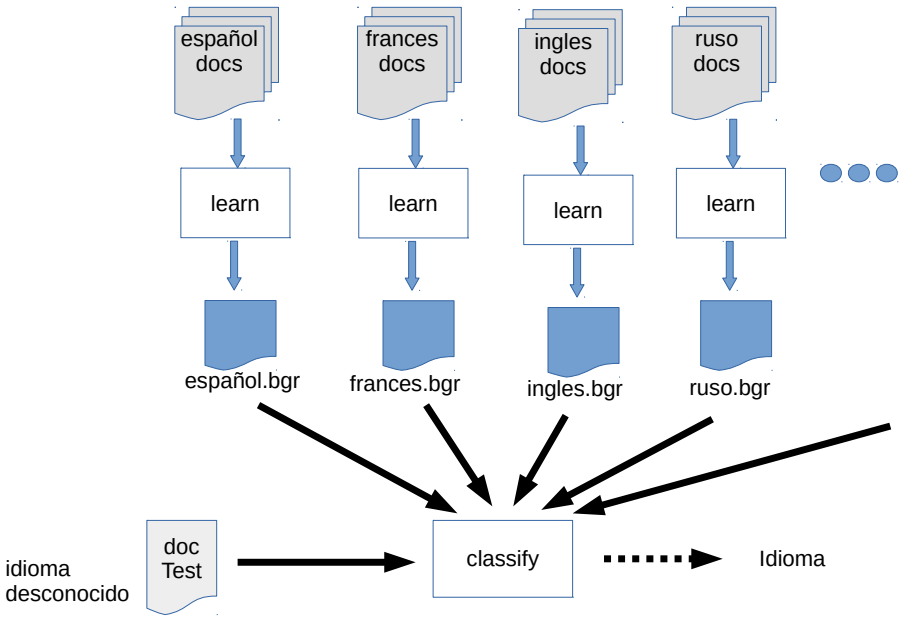

Figure 1: Uso de de los programas

En la figura 1 se ilustra el uso normal del software a desarrollar. Así, en un primer paso deberemos ejecutar el programa, que llamaremos `learn`. Dicha aplicación será la encargada de aprender el conjunto de bigramas asociados a un idioma en concreto. Este tomará como entrada un conjunto de ficheros de texto, todos ellos escritos en un mismo idioma y como salida nos generará un fichero `nombreIdioma.bgr` que contiene información sobre el idioma particular en que están escritos los documentos. Este programa lo deberemos de ejecutar una vez para cada uno de los idiomas que queramos aprender.

La segunda aplicación, que llamaremos `classify`, tomará como entrada un documento de texto con idioma desconocido y un conjunto de ficheros con extensión `bgr` que representan los idiomas candidatos. Como salida obtendremos un echo por pantalla que nos permitirá determinar el idioma más plausible en el que está escrito nuestro documento de test.

Veamos con más detalle cada uno de estos programas:

2.1 Aprendizaje de un idioma: Programa learn

Programa que dado un conjunto de ficheros de texto escritos en un mismo idioma, genera un fichero de salida que contiene la lista de los bigramas diferentes hallados en los todos los ficheros de entrada de entrenamiento.

Como sabemos, un bigrama consiste en dos caracteres que aparecen juntos dentro de una palabra en un texto de entrada. Asociado a cada bigrama podemos utilizar un valor entero que representará la frecuencia acumulada, esto es, el número de ocurrencias de dicho bigrama en todos y cada uno de los ficheros de entrada. La sintaxis del comando learn es

como sigue:

```
learn {-c|-a} [-l nombreIdioma] [-f ficheroSalida] texto1.txt {
   texto2.txt texto3.txt}
```

Los argumentos son:

`-c|-a` parámetro obligatorio, debe de figurar uno de los dos necesariamente:
`-c` crea un nuevo fichero de idioma, en caso de que el fichero salida exista lo destruye y crea uno nuevo
`-a` actualiza la frecuencia de aparición de cada uno de los bigramas presentes en los documentos de entrada a un idioma existente previamente. En esta opción, se presupone que el fichero con nombre `ficheroSalida.bgr` deberá existir antes de ejecutar el programa y debe ser un fichero de idioma válido. Al terminar el programa el fichero `ficheroSalida.bgr` se verá actualizado con la nueva información.

parámetros optativos:
`-l` indica el nombre del idioma del que se aprende (unknown por defecto)
`-f` indica el nombre del fichero de salida `<.bgr>` (`out.bgr` por defecto)

El comando learn acepta un número variable de ficheros de entrada `<.txt>`, aunque al menos uno será obligatorio[2]. El fichero de salida, que representa al idioma, contiene una lista formada por cada bigrama y el número de veces (frecuencia) que aparece en los ficheros de entrada. La lista, se guarda de forma ordenada por frecuencia de mayor a menor. Un ejemplo de ejecución podría ser el siguiente:

```
prompt% learn -c -l spanish -f ficheroSalidaSpanol.bgr
   ficheroSpanol1.txt ficheroSpanol2.txt
```

El resultado de esta ejecución es un fichero en disco ("ficheroSalidaSpanol.bgr" en el ejemplo), que contendrá la unión y el recuento de los bigramas que aparecen en los ficheros de entrada proporcionados. El formato del fichero `<.bgr>` consiste en una cabecera y la lista de bigramas como se detalla en la sección 5

2.2 Programa classify. Asigna el texto a un idioma

El programa classify tiene como objetivo averiguar automáticamente el idioma en el que está escrito un fichero de texto, comparándolo con los idiomas para los que hayamos realizado previamente el cálculo de la lista de sus bigramas. La sintaxis del comando learn es como sigue:

```
classify ficheroDesconocido.txt ficheroIdioma1.bgr {
   ficheroIdioma2.bgr ficheroIdioma2.bgr}
```

[2]Un caso particular, es cuando el entrenamiento se hace utilizando un único fichero.

La salida será un conjunto ordenado de pares `<idioma, distancia>`. Por ejemplo, podríamos utilizar

```
prompt% classify ficheroIdiomaDesconocido.txt
    ficheroIdiomaEspanol.bgr  ficheroIdiomaIngles.bgr
    ficheroIdiomaAleman.bgr ficheroIdiomaPortugues.bgr
```

y obtener como salida:

```
español 0.2
portugues 0.3
ingles 0.7
aleman 0.9
```

El fichero de idioma desconocido será clasificado como aquel idioma con una distancia menor respecto a los ficheros `<bgr>` con los que se compara.

3 ¿Cómo realizar el aprendizaje del Idioma?

Para realizar el aprendizaje debemos abrir, uno a uno, los ficheros de texto que contienen el documento de entrada (es importante leer la nota que aparece al final de esta sección). Una vez abierto, pasamos a recorrerlo identificando los pares de caracteres consecutivos en el mismo. Estos caracteres pueden ser letras, dígitos, separadores (blanco, tabulador, salto de línea,) o símbolos de puntuación (comas, puntos, puntos y comas, etc.)

Una vez que tenemos un par de caracteres, deberemos chequear si forman un bigrama válido. En este sentido, se dice que un bigrama es válido si se encuentra compuesto de dos caracteres que pertenecen a un conjunto de caracteres predefinidos, por ejemplo $validos = \{\texttt{abcdefghij}\}$. Así, si el contenido del fichero es el siguiente:

```
gafa: –fachada–, hija.
```

identificaremos los bigramas: `ga af fa fa ac ch .... ja`

Para cada bigrama en el fichero (par de caracteres válidos) debemos de calcular la frecuencia de aparición. Para facilitar este proceso debemos almacenar las frecuencias en una matriz bidimensional, `F`, de tamaño $n \times n$, siendo n el número de caracteres válidos. En la tabla siguiente podemos ver un esquema de la matriz asociada a nuestro ejemplo, con $n = 10$. Inicialmente, todas las posiciones de la matriz tendrán el valor cero.

En la posición $F[i][j]$ se almacenará la frecuencia con la que aparece el bigrama que tiene como primer carácter el elemento (i-1)-ésimo y como

segundo carácter el elemento (j-1)-ésimo en el conjunto $validos$, respectivamente. Así, el bigrama `ab` se almacena en la posición $F[0][1]$ y el bigrama `ch` se almacena en la posición $F[2][7]$. Por tanto, dado un nuevo bigrama, podemos actualizar su frecuencia fácilmente si conocemos las posiciones de los caracteres en el conjunto $valido$ e incrementamos en uno su valor.

F	a 0	b 1	c 2	d 3	e 4	f 5	g 6	h 7	i 8	j 9
a 0	1	2	3	0	3	2	6	8	9	0
b 1	3	2	6	8	9	0	3	0	3	2
c 2	1	2	0	0	3	0	4	1	0	0
..	.....									
j 9	4	0	0	2	6	8	5	2	0	0

Una vez que se han procesado todos los ficheros de entrada, podemos salvar el fichero idioma salida transformando esta tabla en un vector de bigramas, que será ordenado por frecuencia de aparición de los bigramas y los volcamos en el fichero de salida correspondiente.

3.1 ¿Cómo actualizar un fichero de idioma?

Cuando llamamos al programa `learn` con la opción de actualizar, por ejemplo

```
learn -a -l ingles -f ingles.bgr  documentoIngles1.txt
   documentoIngles2.txt
```

el objetivo es actualizar la frecuencias considerando las frecuencias de los bigramas que aparecen en `documentoIngles1.txt documentoIngles2.txt`. En este caso, el proceso es ligeramente distinto, por lo que lo detallaremos a continuación:

En un primer paso debemos abrir el fichero de idioma `ingles.bgr` y cargar dicho idioma en memoria, en un objeto `idioma`. A partir de dicho idioma debemos de obtener la matriz bidimensional de frecuencias F, donde $F[i][j]$ representa el valor actual del bigrama asociado a la secuencia de caracteres (i-1)-ésimo, (j-1)-ésimo del conjunto de caracteres válidos.

A partir de aquí se sigue el mismo procesamiento que en el caso anterior. Notar que de forma abstracta, lo único que hacemos es inicializar el contador de bigramas, F, con los datos obtenidos a partir de un fichero Idioma.

3.2 Notas a tener en cuenta:

De forma genérica, indicar que los ficheros de texto que vamos a considerar, están todos en la misma codificación. En concreto recomendamos usar la codificación ISO 8859-15 (también conocida como Alfabeto Latino n.º 1 o ISO Latín 1.) que usa un byte por cada carácter y que define la codificación del alfabeto latino, incluyendo los diacríticos (como letras

acentuadas, ñ, ç). Podemos aprender mas sobre este tema en la sección 7 de esta memoria.

En cualquier caso, se **recomienda** que inicialmente se consideren como caracteres válidos únicamente los caracteres:

$validos = \{$`abcdefghijklmnopqrstuvwxyzABCDE...XYZ`$\}$

Este conjunto de caracteres válidos será fácilmente extensible en etapas posteriores

En la plataforma **decsai** podrán encontrar una serie de ficheros de texto de ejemplo en el formato indicado escritos en 3 idiomas diferentes.

- `aliceWonder.Unix.ISO-8859-15.txt`,
- `fortunata.Unix.ISO-8859-15.txt`,
- `lesMiserables.UNIX.ISO-8859-15.txt`,
- `quijote.txt`,
- `test_learn.txt`.

4 ¿Cómo realizar la clasificación?

La forma de proceder es como sigue. En primer lugar debemos de obtener el conjunto de bigramas del documento IdiomaDesconocido, junto con su frecuencia asocidada. Para ello, basta con aprender un nuevo idioma (unknown) asociado al fichero (debemos aprovechar el código que hemos desarrollado para la fase de aprendizaje).

Una vez que tenemos el idioma asocidado (unknown) debemos de calcular su distancia con respecto a cada uno de los idiomas conocidos. A menor distancia, mayor parecido. Una distancia cero indica que los idiomas son el mismo.

Para la obtención de esta medida global, necesitamos conocer la posición que ocuparía un bigrama cuando consideramos todos los bigramas en un idioma dado ordenados en orden decreciente de frecuencia.

Supongamos que `idioma.ranking(bg)` nos devuelve dicha posición para un objeto `idioma` dado, empezando en el valor 0 para el primer bigrama hasta `idioma.size()-1`, definiendo `size()` como el número de bigramas en el idioma. En el caso en que `bg` no se encuentre en `idioma` se le asigna el máximo valor posible, esto es, la posición siguiente al último, `idioma.size()`. Entonces, formalmente podemos definir la distancia a un idioma `unknown`, U, de un idioma `candidato`, C, como:

$$distancia(U,C) = \sum_{bg \in U} \frac{abs(U.ranking(bg) - C.ranking(bg))}{U.size() * U.size()}$$

Esto es, la distancia es la suma de las distancias parciales (normalizadas) asignadas a cada uno de los bigramas del fichero de idioma desconocido. El idioma seleccionado será aquél que obtenga una menor puntuación de entre los comparados.

Nota: A efectos de implementación, la función distancia se debe implementar como miembro de la clase Idioma, esto es

```
class Idioma {
  double  distancia( const Idioma & otro);
  ...
};
```

5 Formato fichero de conjunto de bigramas

Cabecera

- Una cadena mágica “MP-BIGRAMAS-%-1.0”.
  Esta cadena se usa para distinguir entre archivos de distintos tipos. Para archivos de texto la cadena es: “MP-BIGRAMAS-T-1.0”
- Un separador. Es decir, un carácter “blanco”, normalmente salto de línea.
- Una palabra que describe el idioma correspondiente a los bigramas. En el ejemplo de ejecución anterior, sería la palabra “spanish”. Como cadena, el usuario podría utilizar cualquier palabra. Por ejemplo, para un texto cuyo idioma es desconocido podría usar la palabra “unknow”.
- Un separador, normalmente el carácter blanco.
- Un entero que indica el número total de bigramas distintos encontrados, esto es, la longitud del conjunto de bigramas que viene a continuación en el fichero.
- Un separador, normalmente un salto de línea.

Conjunto de bigramas

Un conjunto que contiene tantos datos como bigramas haya, que aparecerán en orden de mayor a menor frecuencia. Cada dato contiene los 2 caracteres del bigrama y un número entero que indica la frecuencia de aparición en los ficheros de entrenamiento.
Al tratarse de **ficheros de texto**, el conjunto se guarda intercalando separadores entre los datos, normalmente el carácter blanco, entre bigrama y frecuencia.

6 Diseño propuesto

Se propone crear las siguientes clases para resolver el problema:

Bigrama : clase base formada por el par cadenaC (c-string), contiene 2 caracteres hábiles + `'\0'` y su frecuencia absoluta.

ContadorBigramas : matriz 2D dinámica de enteros, con tantas filas y tantas columnas como caracteres válidos tengamos. Cada entero guarda la frecuencia del bigrama determinado por los dos caracteres que representa la fila y columna correspondiente. Clase necesaria para llevar a cabo el cómputo de las frecuencias de cada bigrama. Necesita: un string con el conjunto de caracteres válidos, esto es $validos$. Se proporcionará su valor en tiempo de ejecución.

En cualquier caso, se **recomienda** que inicialmente se consideren como caracteres válidos los siguientes caracteres:

$validos = \{$`abcdefghijklmnopqrstuvwxyzABCDE...XYZ`$\}$

este conjunto lo podemos codificar mediante un c-string o string

```
string validos = "
    abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

Usar este conjunto de caracteres válidos nos ayudará a resolver nuestro problema sin grandes preocupaciones (se explican los posibles problemas con mas detalle en la sección 7). Por tanto, si implementamos correctamente la aplicación, el conjunto de caracteres válidos será fácilmente extensible en etapas posteriores, lo cual es altamente deseable si deseamos obtener una mejora considerable en la clasificación. Esto es debido a que debemos ser conscientes de que considerar un carácter como la `ñ` puede ser muy indicativo de que el texto está en castellano o una cedilla, `Ç`, lo es de que está escrito en francés.

Aunque formalmente estamos hablando de caracteres, nos podemos encontrar con problemas con el estándar utilizado para la codificación de los caracteres. En la Sección 7 hablaremos mas sobre el tema.

Idioma : Contiene por un lado el identificador del idioma, por defecto será Unknown y la lista de bigramas ordenados por frecuencia. Esta clase será utilizada para la lectura y escritura en ficheros. Una vez calculada la matriz ContadorBigramas se puede elaborar la lista de bigramas, el vector dinámico de bigramas, y guardarlo en un fichero fácilmente. Este contiene unicamente los bigramas hallados en el texto, esto es, aquellos cuyos valores en la matriz bidimensional sea distinto de 0. Para leer un fichero `<bgr>` debemos usar esta clase.

```
class Bigrama{
private:
    char _bigrama[3];    ///< dos caracteres bigrama + \0
    int _frecuencia;     ///< numero de veces que aparece
```

```
class ContadorBigramas {
private:
    int** _bigramas;        ///< Matriz 2D con la frecuencia para
    cada bigrama
    std::string _caracteresValidos;    ///< La longitud de esta
    cadena va a determinar el tamaño de la matriz _bigramas
```

```
class Idioma {
private:
    string _idioma;    ///< indica el nombre del idioma, default:
    unknown
    Bigrama* _ranking;     ///< array dinámico de bigramas (
    ordenado de mayor a menor frecuencia
    int _nBigramas;    ///< numero de bigramas de la lista
```

7 Codificación de Caracteres

Podemos encontrar distintos estándares para realizar la codificación de los caracteres, como por ejemplo ASCII, ISO8859-x o UTF-8, UTF-16, etc.

El primer esfuerzo lo hizo ASCII que consideraba únicamente 128 caracteres (los provenientes del inglés) y por tanto permitía codificar un caracter con un byte. Estos 128 caracteres se vieron insuficientes para trabajar con idiomas como el español o frances, que incluyen caracteres acentuados como la ñ, é, ç, ü, etc. Por ello nace el ISO-8859-15 (llamado ASCII extendido) que utiliza un byte para codificar hasta 256 caracteres. En cualquier caso, ISO-8859-15 no permite representar caracteres en otros idiomas como el chino, ruso, etc. Por lo que aparece el estándar UNICODE (del que es parte UTF-8) que se ha convertido en el estándar actual. UTF-8 puede utilizar desde 1 hasta 4 bytes para codificar un caracter

Así podemos encontrar que estos tres estándares utilizan un único byte (con igual valor) para representar los caracteres

```
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Sin embargo, si consideramos cartacteres como la 'ñ', nos encontramos que:

- ASCII no esta representada
- ISO8859-15 la representa con un único byte con valor en hexadecimal 0xF1
- UTF-8 (caracter unicode U-00F1) la representa con dos bytes con valor en hexadecimal (0xC3 0xB1) (ver https://es.wikipedia.org/wiki/UTF-8).

La codificación de caracteres está por todos lados. Así, la propia consola trabaja con una codificación (por defecto en Linux UTF-8), un fichero de texto también tendrá su codificación (lo podemos ver con el comando `file`) y los editores de texto trabajan con una codificación, aunque en este caso pueden adaptarse a la codificación de los ficheros de texto sobre los que estemos trabajando.

Desde Linux existe una utilidad para poder convertir un fichero de texto de un código a otro, `iconv`.

Veamos todo esto con un pequeño ejemplo. Con nuestro editor (por ejemplo emacs o gedit) editemos un texto con el siguiente contenido: `Hola, camión, ñoño` y lo salvamos con el nombre `prueba.txt`. Podemos consultar su codificación con `file` y el número de caracteres con `wc`

```
prompt$ file prueba.txt
prueba.txt: UTF-8 Unicode text
prompt$ wc prueba.txt
1  3 22 prueba.txt
prompt$ cat prueba.txt
Hola, camión, ńoño
```

Como vemos es un fichero en formato UTF-8, con 1 línea, 3 palabras y 22 bytes. Sin embargo, si hacemos un `cat` nos muestra los 19 símbolos de forma correcta, aunque internamente se almacenen 22 caracteres, como se muestra a continuación.

```
1234567890123456789012345678 90
Hola, camixxn, yyoxxor
```

donde xx e yy representan los codigos unicode para el caracter 'ó' y el caracter 'ñ' y r representa el código del retorno de newline.

Convirtamos ahora ese fichero a código ISO-8859-15. Lo guardaremos en el fichero pruebaIso.txt

```
prompt$ iconv prueba.txt -t ISO885915 -o pruebaIso.txt
prompt$ file pruebaIso.txt
pruebaIso.txt: ISO-8859 text
prompt$ wc pruebaIso.txt
1  3 19 pruebaIso.txt
prompt$ cat pruebaIso.txt
Hola, cami?n, ?o?o
```

Ahora tenemos un total de 19 bytes, pero al hacer un `cat` la consola, que espera una codificación UTF-8, no los sabe interpretar y por tanto no los muestra de forma correcta. Sin embargo, si abrimos el fichero con `gedit` veremos que el contenido es el correcto (gedit reconoce la codificación y la interpreta correctamente).

En Linux podemos cambiar la codificación utilizada por la consola. Para ello, podemos abrinos una `konsole`, terminal de kde, y pulsando con el ratón derecho sobre ella podemos encontrar la opción

`establer codificacion -> europeo occidental -> ISO-8859-15`.

Si lo hacemos, veremos que nos muestra el correctamente el contenido del fichero pruebaIso.txt y no el de prueba.txt. El contenido es el mismo, la diferencia es la codificación utilizada por la consola para mostrar la salida.

7.1 Codificación y C++

Esto puede parecer un lio, pero los lenguajes de programación intentan nos ayudan a ocultar estos problemas, aunque deben considerar una codificación por defecto, por ejemplo C++ o Java utilizan UTF-8. Esto lo podemos ver si intentamos definir la siguiente cadena de caracteres con la palabra ”ñoño”:

```
char cadena[5] = "ńoño";
```

nos muestra el siguiente mensaje de error

```
main.cpp:37:19: error: initializer-string for array of chars is
    too long [-fpermissive]
       char cadena[5]="ńoño";
```

Para poder hacer la inicialización correcta deberemos reservar espacio para los 7 bytes necesarios para almacenar el literal, incluyendo el `'\0'`. De igual forma, si hacemos

```
string cad = "ńoño";
cout << cad << ":" << cad.size() << endl;
```

Nos mostrará que existen 6 caracteres en la cadena `ñoño`. La cadena, está en UTF-8, codificación que es interpretada correctamente por la consola y por tanto nos muestra los símbolos correctos (asumimos que la consola usa dicha codificación, pues el el valor por defecto). En cualquier caso, si realizamos la comparación

```
if (cad[0]=='ń') cout << "iguales";
else cout << "distintos"
```

nos imprimirá `distintos` pues `cad[0]` representa únicamente el primer byte de la cadena y no el símbolo 'ñ'. Recordad que lo que realmente tenemos es `cad[0] == '\xC3'` y `cad[1]=='\xB1'`.

7.2 Trabajaremos con formato ISO 8859-15

En nuestro caso, y ante la necesidad de leer ficheros de texto para extraer los bigramas y aislarnos de estos problemas cuando queramos comparar caracteres especiales (como ñ, á, Á, ü, ç, Ç, ß, etc. que se codifican con más de byte), forzaremos a que los ficheros de datos (documentos o idiomas) estén codificados con código ISO-8859-15. Los documentos

que os pasamos como entrada lo están. Con esto nos garantizamos que cada carácter va a ocupar siempre un único byte y la comparación será simplemente la igualdad entre códigos.

La pregunta que nos hacemos es: ¿cómo podemos incluir dichos caracteres en el string? Es simple, podemos utilizar su código ISO 8859-15 en hexadecimal en el literal, utilizando el carácter de escape. Así, la palabra "ñoño" (recordemos que 'ñ' tiene el código 0xF1), se representa mediante el literal:

```
char cadenaIso[5]= "\xF1o\xF1o"; // representa ñoño
string cadIso = "\xF1o\xF1o";    // representa ñoño
```

Ambas cadenas tendrán logitud 4 y para determinar si la posición i-ésima de un string o c-string con codificación ISO coincide con un determinado carácter espacial es simple, basta con compararlo directamente con el código ISO correspondiente,

```
if (aux[i]=='\xF1')
  cout << "Es ñ: n con tilde en ISO";
```

Para finalizar, indicar que para la práctica vamos a considerar únicamente como caracteres válidos aquellos que representan letras, LETTER, en la codificación ISO 8859-15 (ftp://ftp.unicode.org/Public/MAPPINGS/ISO8859/8859-15.TXT). Dichos carácteres, en su formato impreso, los podemos encontrar en https://www.compart.com/en/unicode/charsets/ISO-8859-15#CS_111

Así el literal de caracteres válidos se puede representar como

```
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ\xE0\xE1\
   xE2\xE3\xE4\xE5\xE6\xE7\xE8\xE9\xEA\xEB\xEC\xED\xEE\xEF\xF0\
   xF1\xF2\xF3\xF4\xF5\xF6\xF8\xF9\xFA\xFB\xFC\xFD\xFE\xFF"
```

8 Evaluación y prueba

El contenido del fichero `test_learn.txt` es el siguiente:

```
Dónde digo: "digo", digo: ¿ Diego ?
```

```
MP-BIGRAMAS_IDIOMA-T-1.0
unknown
9
di 4
go 4
ig 3
de 1
dó 1
```

```
eg 1
ie 1
nd 1
ón 1
```

9 Material a entregar

Cuando esté todo listo y probado el alumno empaquetará la estructura de directorios anterior en un archivo con el nombre **practica6.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 **practica6.zip** debe contener la siguiente estructura:

```
./
├── makefile
├── include
│   ├── Bigrama.h
│   ├── ContadorBigramas.h
│   └── Idioma.h
├── src
│   ├── Bigrama.cpp
│   ├── ContadorBigramas.cpp
│   ├── Idioma.cpp
│   ├── learn.cpp
│   └── classify.cpp
├── bin
├── obj
├── lib
├── data
│   ├── alice.bgr
│   ├── fortunata.bgr
│   ├── lesMiserables.bgr
│   ├── quijote.bgr
│   ├── aliceWonder.Unix.ISO-8859-15.txt
│   ├── fortunata.Unix.ISO-8859-15.txt
│   ├── lesMiserables.UNIX.ISO-8859-15.txt
│   └── quijote.txt
└── zip
```

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

```
unzip practica6.zip
make
bin/learn
bin/classify
bin/learn -c -l spanish -f idiomaSpanol1.bgr fortunata.Unix.ISO-8859
bin/learn -a -l spanish -f idiomaSpanol2.bgr idiomaSpanol1.bgr quijo
```