
# Guion de prácticas adicional

## *Gestión de Entrada/Salida con ficheros*

*Abril de 2018*

DECSAI

Metodología de la Programación

Curso 2017/2018

# Índice

# 1. Introducción

Esta sesión de prácticas está dedicada a aprender a utilizar la entrada/salida en ficheros en C++, en especial ficheros de texto, esto es, archivos que contienen únicamente secuencias de caracteres que ocupan un único byte que representa letras, dígitos, signos de puntuación o separadores (espacios en blanco, tabuladores, saltos de línea). Estos ficheros son utilizados en muchas situaciones como mecanismo para almacenar los datos (como por ejemplo, un fichero de código C++ se almacena en un fichero de texto, una base de datos se puede exportar a un fichero .csv), permitiendo el trasvase de información entre muchos programas.

El modo normal en que procesamos un archivo de texto, como por ejemplo cuando consideramos analizamos un documento, es leerlo/escribirlo desde el principio hasta el final, por lo que se habla de flujo de caracteres (stream). El concepto de flujo de entrada/salida permite ocultar detalles sobre cómo funcionan los dispositivos externos. El programador considera el flujo como una secuencia de caracteres que puede leer o escribir. Pero para realizar la conexión con el fichero se hace a través de una zona de memoria conocida como buffer donde se ubican de forma temporal los caracteres a ser tratados. Esto permite minimizar los accesos a disco (mas lentos) ya que en cada lectura física de disco se carga todo lo posible en el buffer de entrada y las escrituras no se producen físicamente hasta que el buffer de salida esté completo. Pero también nos independiza de cómo el sistema operativo realiza dichas operaciones de carga y descarga de los buffers.

En general, cuando leemos un fichero de texto, nos ubicamos en el principio del mismo, y vamos procesando los caracteres en orden hasta llegar al final del archivo. De igual modo, cuando escribimos, se empieza a escribir en el principio del archivo y se vuelcan en orden todos y cada uno de los caracteres al disco. En ambos casos se hace un recorrido secuencial sobre los ficheros, siendo la forma mas habitual para tratar con ficheros de entrada/salida. En ningún momento intentamos posicionarnos de forma directa en posiciones intermedias del archivo, por ejemplo en el carácter 1526.

# 2. Primeros Pasos

El estándar C++ permite hacer la entrada/salida de datos de una forma muy similar a como se realiza la entrada/salida desde consola con `cin` y `cout`. Recordemos que `cin` es un objeto de la clase `istream` (input stream, flujo de entrada) y `cout` es un objeto de la clase `ostream` (output stream, flujo de salida). Tanto cin como cout son objetos globales creados en el estándar C++, por lo que los tenemos a disposición del programador al incluir `iostream`.

Cuando trabajos con archivos de texto tenemos definidas dos clases con un comportamiento similar: la clase `ifstream` (flujo de archivos de entrada) hereda de istream, y la clase de `ofstream` (flujo de archivos de salida) hereda de ostream. Por lo tanto, todas las funciones miembro y

operadores que se pueden utilizar con objetos istream u ostream también se pueden aplicar a los objetos de la clase ifstream y ofstream. Pero hay una primera diferencia: Es el propio programador el que se debe encargar de crear el objeto y asociarlo con un determinado fichero de entrada/salida, lo que nos permite controlar qué archivos se utilizan y el propósito con el que los utilizamos.

Una segunda diferencia es que para poder trabajar con archivos disponemos de algunas funciones adicionales e información interna (atributos) que permiten controlar la *conexión* del flujo de entrada con los archivos físicos en disco.

## 2.1. Abriendo y cerrando ficheros

En primer lugar, debemos de declarar tantos objetos del tipo `fstream` como archivos queramos utilizar de forma simultánea. En el siguiente fragmento de código usamos dos objetos, uno para realizar la entrada (lectura) de datos y otro de salida (escritura). Pero un programa puede tener y usar tantos archivos como desee, para ello debemos declarar un objeto para cada archivo.

```
#include <iostream>
#include <fstream> // las declaraciones de clase para los
    archivos de flujo de texto

usando namespace std;
...
int main ()
{
ifstream mi_fichero_entrada; //  flujo de entrada
ofstream mi_fichero_salida; //   flujo de salida
...
}
```

En este caso sólo hemos declarado los objetos, indicando si se utilizarán como flujos entrada o de salida. Pero para trabajar con los ficheros debemos de poder asociarle a cada flujo el fichero concreto en disco sobre el que queremos trabajar. Para ello, debemos indicarle al objeto de flujo de datos el nombre del archivo (mediante un string o cstring) y el objeto se encargará de gestionar con el Sistema Operativo las conexiones necesarias para su lectura/escritura. Continuando con el ejemplo

```
#include <iostream>
#include <fstream> // las declaraciones de clase para los
    archivos de flujo de texto

usando namespace std;
...
int main ()
{
ifstream mi_fichero_entrada; //  flujo de entrada

mi_fichero_entrada.open("datosEntrada.txt");
```

```
ofstream mi_fichero_salida; //    flujo de salida

mi_fichero_salida.open("fSalida.txt");
...
}
```

Esta apertura de ficheros puede estar sujeta a problemas, como puede ser el caso en el que el fichero que queremos abrir para lectura no exista (por ejemplo, nos hemos equivocado al escribir el nombre del fichero). En estos casos es conveniente estar seguros de que la apertura ha sido correcta antes procesar los ficheros. Para ello podemos utilizar el método `is_open`, que devuelve `true` si la operación de apertura se pudo realizar de forma correcta.

```
if (mi_fichero_entrada.is_open()){
   // podemos continuar con el procesamiento
}
```

Gestionar los datos de entrada/salida puede ser sencillo, nos basta con utilizar la sobrecarga de operadores `operator>>` y `operator<<`, definida para los flujos de entrada y salida, respectivamente.

```
string a;
int x;
char y;

if (mi_fichero_entrada.is_open()){
   mi_fichero_entrada >> a; // lee hasta encontrar separador
   mi_fichero_entrada >> x >> y ; // lee entero  char
}

if (mi_fichero_salida.is_open()){
   mi_fichero_salida << "El numero de lineas es" << n_lineas <<
   endl;
   for (int i=0; i<n_lineas;i++)
    mi_fichero_salida << "linea "<< i << endl;
}
```

El contenido del fichero de entrada se procesa tal y como lo haría cin cuando leemos desde consola. Igualmente, en el fichero de salida se incluiría un contenido idéntico al que obtenemos al realizar un cout.

Una vez que hemos terminado de leer/escribir en un archivo, es necesario cerrar el archivo (`close`). Cerrar un archivo implica desconectar el objeto fstream del archivo físico y decirle al Sistema Operativo que realice las tareas necesarias para guardar su estado de forma correcta. En el caso de un fichero de salida, su cierre nos garantiza que toda la información que podría estar guardada temporalmente en el buffer de memoria, pero no volcada físicamente al fichero en disco, sea guardada de forma correcta. Recordemos que para ahorrar accesos a discos (bastante costosos), se realizan sólo cuando el buffer esta lleno. Cerrar el fichero, garantiza que el proceso finaliza de forma correcta.

El cierre se realizar al llamar al método `close()` de forma explícita o bien cuando se llama al destructor del objeto, que cierra el archivo. En cualquier caso, es conveniente cerrar el fichero cuando ya no sea necesario, pues nos libera los recursos utilizados.

```
mi_fichero_entrada.close();
mi_fichero_salida.close();
```

Para leer un archivo dos veces con el mismo objeto, léalo una vez hasta el final de archivo, borre el estado de fin de archivo con `clear()`, para finalizar cerrándolo. Entonces, podemos volver a abrirlo y leerlo nuevamente.

## 2.2. Escritura en archivo: Ejemplo simple

Pretendemos diseñar una función que escriba en un fichero la tabla de multiplicar de un número dado. Así, para el número 5, creará el fichero con nombre `multiplicar5.txt` en el directorio de trabajo con la siguiente información:

```
Tabla multiplicar del 5
5 x 0 = 0
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50
```

Para obtener dicho fichero basta con utilizar el siguiente código:

```
void tablaMultiplicar(int numero){
    string nombreFichero="multiplicar"+to_string(numero)+".txt";
    ofstream fsal;
    fsal.open(nombreFichero);
    if (fsal.is_open()){
      fsal << "Tabla multiplicar del "<< numero << endl;
      for (int i = 0 ; i<=10 ; i++){
        fsal << numero << " x " << i<< " = "<< numero*i << endl;
      }
      fsal.close();
    } else {
        cout << "problemas con fichero "<< nombreFichero<< endl;
    }
}
```

En este caso, cuando abrimos el archivo para salida puede ocurrir dos cosas: La primera, que el fichero `multiplicar5.txt` no existe. En este

caso, el Sistema Operativo crea el archivo vacío con ese nombre, almacena la información y lo cierra. Esta suele ser la operación mas común cuando trabajamos con archivos. Sin embargo, la situación es diferente cuando el archivo `multiplicar5.txt` ya existe con anterioridad. En este caso, el Sistema Operativo borrará el archivo, y crea uno nuevo sobre el que se realizarán las operaciones de escritura. En este caso, debemos asegurarnos de que ese sea el comportamiento que deseamos, o actuar en consecuencia para evitarlo utilizando los métodos de la clase.

## 2.3. Lectura de archivo: Ejemplo simple

En este ejemplo, trataremos de leer un fichero que ha sido generado por nuestra función `tablaMultiplicar` y trataremos de chequear que los datos almacenados en el fichero son correctos. Abrimos el fichero de entrada, por ejemplo `multiplicar5.txt` utilizando una cadena que contiene el nombre del archivo. Asumimos que el programador, o usuario, proporciona un nombre de archivo correcto, en caso contrario nos dará un mensaje de error.

```
void chequeatablaMultiplicar(const string & nombreFichero){

    ifstream fent;
    fent.open(nombreFichero);
    if (fent.is_open()){
        string cadena;
        int numero;
        // Leemos primera linea
        fent >> cadena;  //lee: Tabla
        fent >> cadena;  //lee: multiplicar
        fent >> cadena;  //lee: del
        fent >> numero;
        for (int i = 0; i<=10;i++){
            int m,j,res;
            string operador,igual;
            fent >>  m >> operador >> j >> igual >> res;
            // m es multiplicando
            // operador es "x"
            // j es multiplicador
            // igual es "="
            // res es el resultado, m*j

            if (!( (m==numero) && (j==i) && (res==m*j) ) ){
                cout << "error en linea i" << endl;
            }
        }
        cout << "fichero correcto"<< endl;
        fent.close();
    } else {
        cout << "problemas con fichero "<< nombreFichero<< endl;
    }
}
```

El Sistema Operativo buscará el archivo en el mismo directorio donde está el ejecutable (salvo que le pasemos la ruta que se debe seguir

para llegar al archivo desde el directorio de trabajo o el ruta completa desde el directorio raíz). La ruta serán las mismas cadenas que se usan todo el tiempo en DOS, Linux o Unix. Por ejemplo, si el fichero `multiplicar5.tex` esta en el directorio de trabajo es suficiente con llamar a la función

```
int main(){
chequeatablaMultiplicar("multiplicar5.txt");
}
```

Pero si asumimos que el ejecutable está en un directorio `bin` y los datos en el directorio `data` deberíamos llamarlo como

```
int main(){
chequeatablaMultiplicar("../data/multiplicar5.txt");
}
```

En esta práctica asumiremos la estructura anterior, por lo que los archivos se encuentran en el subdirectorio `data` del proyecto Si abre un archivo de entrada y el sistema operativo no puede encontrar un archivo con ese nombre, se trata de un error, que analizaremos con mas detalle posteriormente.

# 3. Lectura de Ficheros. Otras funciones para entrada de datos

Además de usar el `operator>>` para la lectura de datos del flujo de entrada, la clase `ifstream` dispone de otros métodos que son de utilidad en múltiples casos. Estos métodos ya los encontrábamos en (los hereda de) la clase `istream` por lo que también los podemos utilizar con `cin`.

- Lectura de caracteres del flujo:

```
(1) int get();
(2) istream & get (char & c);
```

  Lee el siguiente carácter, sin omitir nada, y lo devuelve como un entero (opción 1) o lo asigna como valor del argumento (opción 2).

  La opción 1, si se encuentra eof (End Of File-fin de fichero), devuelve el valor especial definido como EOF. La necesidad de probar EOF representa un inconveniente para su uso. Por eso, se recomienda utilizar el resto de las opciones que devuelven una referencia al objeto `ifstream`; esta referencia se puede usar para comprobar el estado del flujo.

  Veamos el siguiente ejemplo, que muestra el contenido de un fichero por pantalla.

```
#include <iostream>      // std::cin, std::cout
#include <fstream>       // std::ifstream
using namespace std;
...
ifstream is("../data/fichero.txt");
char c;
while (is.get(c)) //lee c y chequea estado del objeto is
   cout << c;

is.close();                  // cerramos fichero
```

- Leer c-string:

```
(3) istream& getline (char* s, int n );
(4) istream& getline (char* s, int n, char delim );
```

Leer un c-string (opciones 3 y 4) hasta que se han leído n caracteres (incluyendo el terminador `'\0'`) o se encuentra el carácter delimitador (`'\n'` en la opción 3) o el especificado por el usuario (`delim` en la opción 4). Termina la cadena de caracteres con `'\0'` para garantizar que como resultado de la lectura obtenemos un c-string válido, independientemente de cómo fue leído. Debemos asegurar que la longitud del array apuntado por `s` tenga un tamaño suficiente para alojar n caracteres, en caso contrario se producirá un error de desbordamiento.

El carácter `'\n'` no se incluye en la cadena `s`, lo cual no suele ser problemático, por ser el comportamiento mas lógico cuando procesamos la entrada desde un fichero, este es, normalmente deseamos eliminar dicho carácter.

El valor devuelto es una referencia al objeto istream, que se puede utilizar para comprobar el estado de la secuencia. Si se rellena el array sin encontrar el `'\n'` (esto es, no llegamos a leer el final de la línea), la operación de lectura falla (para indicar que no se ha leído correctamente el fichero de entrada). Veamos un ejemplo, donde suponemos que tenemos el siguiente fichero de entrada:

```
Si se rellena el array sin encontrar el
El valor devuelto es una referencia al objeto istream,
donde suponemos que tenemos el siguiente fichero de entrada
de lectura falla (para indicar que no se ha leido
```

```
void prueba_getline(int n){
 ifstream fe("data/lineasLargas.txt");
 cout << "Leemos lineas de "<< n<< " caracteres" << endl;
 if (fe.is_open()){
    char * cad  = new char[n];
```

```
    int k = 0;
    while (fe.getline(cad,n))
        cout <<"␣"<< ++k <<":"<<cad << endl;

    cout << "tras␣bucle:␣"<<cad << endl;
    fe.close();
    }
    else cout << "error␣de␣apertura␣"<< endl;

int main(){
 prueba_getline(300);
 prueba_getline(20)
}
```

La salida de las dos llamadas es la que se muestra en el cuadro siguiente, donde en el primer caso obtenemos la salida esperada (es posible leer todas las líneas), mientras que en el segundo caso, al poner un máximo de 19 caracteres, (n-1, por el '\0'), al realizar la primera operación de lectura y ser incorrecta, se detecta el error en el `ifstream` y se impiden las siguientes lecturas.

```
Leemos lineas de 300 caracteres
 1:Si se rellena el array sin encontrar el
 2:El valor devuelto es una referencia al objeto istream,
 3:donde suponemos que tenemos el siguiente fichero de
   entrada
 4:de lectura falla (para indicar que no se ha leí
 tras bucle:

Leemos lineas de 20 caracteres
 tras bucle: Si se rellena el ar
```

- Leer std::string

```
istream & getline (istream &, std :: string &);
```

Esta función global, declarada en `<string>`, permite leer una línea en un `std::string`. Dado que string ajusta el tamaño automáticamente, según sea necesario, leer una línea en un `std::string` con esta función no produce desbordamiento, siendo la mejor manera de procesar un archivo de entrada cuando queremos leer línea a línea.

El istream que devuelve la función puede ser utilizado para chequear la correcta lectura, como muestra el siguiente ejemplo:

```
#include <string>
...
void prueba_getline(){
 ifstream fe("../data/lineasLargas.txt");
 if (fe.is_open()){
    string cads;
```

```
    int k=0;
    while (getline(fe,cads) ){
       cout << ++k<< ":";
       cout << cads << endl;
    }
    fe.close();
} else cout << "error en fichero"<< endl;
```

# 4. Detectando errores

Como hemos comentado, un `ifstream` dispone de un conjunto de bits internos que permiten controlar el estado del mismo. Cuando se produce un error en la transmisión de datos (como en el ejemplo anterior al no encontrar un `'\n'`), se establece una alerta (se encienden los bits de error) y por seguridad cualquier operación de E/S sobre ese archivo deja de realizarse. El programador puede borrar el estado de alerta (mediante la función miembro `clear()`), pero debemos estar seguros de lo que estamos haciendo.

Veamos un ejemplo, donde en primer lugar intentamos abrir un fichero con un nombre incorrecto. En este caso, como es lógico se detecta el error, activando el bit correspondiente, y si posteriormente intentamos realizar la apertura correcta no nos dejará, salvo que borremos las señales de error mediante un `clear()`..

```
ifstream fe;
fe.open("NombreMal.txt");
if (!fe.is_open()){
 cout << "Error de apertura " << endl;
 fe.clear();   //Reseteamos
 fe.open("correcto.txt");

 }
```

Existen diversas funciones miembro para chequear el estado del flujo como:

- `bool is_open()`. Devuelve verdadero si el flujo está abierto para lectura o escritura, false en caso contrario.
- `bool good()` Devuelve verdadero si todo está correcto, esto es, ninguno de los bits de error está activado y el flujo de datos se puede usar (está abierta).

```
if ( stream_object.good() ){   // equv if (stream_object)
   procesar
}
```

Preguntar directamente por el flujo en una condición, como por ejemplo `if (stream_object) {...}`, nos da el mismo valor true-false que devuelve la función `good()`. Fijaros que ya hemos utilizado esta comprobación para leer el flujo hasta el final, como muestra el siguiente ejemplo.

```
while (getline(fe,cads)){ // while (getline(fe,cads).good())
       cout << ++k<< ":";
       cout << cads << endl;
    }
```

- `bool bad()`. Devuelve true si hay algún problema hardware en la entrada/salida de datos (no es el opuesto de `good()`).
- `bool fail()` Es true si hay algún error lógico en la entrada/salida de datos (entrada inválida) o detecta error hardware. Por ejemplo, el error lógico se produce cuando intentamos leer un entero y en el fichero hay almacenado un carácter. Si nos encontramos con una entrada inválida, el programa puede limpiar (`clear`) la entrada e intentar continuar.
- `bool eof()` Devuelve verdadero si se ha intentado leer después del final del archivo. En este sentido, solo tras intentar realizar un intento de lectura fallido se activa la señal de EOF. Si en nuestro programa alcanzamos el EOF de forma inesperada (por ejemplo, cuando creemos que el fichero tiene 30 líneas y sin embargo en realidad sólo tiene 10) determinaremos que algo va mal y deberemos actuar en consecuencia, por ejemplo dando un mensaje de alerta al usuario.

El siguiente código nos muestra cómo podemos actuar tras leer un dato de entrada, asumimos un entero, de un fichero que ha sido abierto correctamente. Distinguiremos dos situaciones, la primera en la cual el valor leído es correcto, y por tanto puede utilizarse, y la segunda en la que hemos detectado un error o hemos llegado al final del fichero y debemos parar.

```
int valor;

do{

fichero_entrada >> valor;  // Leemos valor

if (fichero_entrada.good()) { // Todo OK
  valor_correcto = true;
  podemos_continuar = true;
}
else if (fichero_entrada.eof()) {  // Hemos alcanzado EOF
  valor_correcto = false;  // No se ha obtenido ningún valor
  podemos_continuar = false; // No podemos seguir leyendo
}
else if (fichero_entrada.bad()) { // Error harware E/S
  valor_correcto = false;
```

```
  podemos_continuar = false;
}
else if (fichero_entrada.fail()) { // Entrada no valida
  cout << "Invalida (esperaba entero) salto a sig linea" <<endl;
  valor_correcto = false;

  fichero_entrada.clear(); //Limpiamos el stream para poder leer
  char c;
  while (fichero_entrada.get(c) && c != '\n'); // Leemos
  if (fichero_entrada.good()) //leido resto linea correctamente
     podemos_continuar = true;
  else {
     podemos_continuar = false;
     cout << "alcanzado EOF o error al procesar linea"<< endl;
    }
}

if (valor_correcto)
  cout << "Leido "<< valor << endl;

} while (podemos_continuar);
```

Así, al ejecutar este código asumiendo el siguiente fichero

```
23
18
2xfs
8
ho3la
33
14.5
12
```

obtenemos la siguiente salida:

```
Leido 23
Leido 18
Leido 2
Invalida (esperaba entero) salto a sig linea
Leido 8
Invalida (esperaba entero) salto a sig linea
Leido 33
Leido 14
Invalida (esperaba entero) salto a sig linea
Leido 12
```

# 5. Práctica a entregar

Continuando con la gestión de los bigramas, podemos caracterizar un idioma si calculamos la frecuencia de aparición de cada uno de los posibles Bigramas en un idioma y los ordenamos por de mayor a menor fre-

cuencia. Así, por ejemplo, en la siguiente tabla se muestran los 10 Bigramas mas frecuentes de los idiomas Español, Francés e Inglés (calculados a partir distintas obras clásicas). La frecuencia absoluta de los bigramas no es relevante por el momento, pero si podemos considerar el orden de los mismos en los distintos idiomas, así en el subdirectorio `data` se pueden encontrar ficheros de bigramas aprendidos a partir de otras fuentes.

| Español | Francés | Inglés |
|---|---|---|
| en=32618 | ai=9522 | he=4084 |
| de=32550 | es=9393 | th=3930 |
| la=29088 | le=9298 | in=2339 |
| es=28903 | en=8813 | er=2149 |
| ue=28493 | it=8754 | an=1850 |
| er=27384 | re=8436 | ou=1738 |
| qu=26242 | de=8163 | it=1535 |
| ra=24083 | nt=7583 | nd=1470 |
| as=22108 | on=7398 | at=1405 |
| os=21746 | ou=7200 | re=1392 |

Para gestionar la colección de Bigramas en un determinado idioma se propone el uso de la clase Idioma (definida en el fichero `idioma.h`) que representa el conjunto los bigramas asociados a un idioma junto con el identificador del mismo. En particular, esta clase dispone de métodos que permite la lectura/escritura de un conjunto de bigramas en un fichero.

Para ello, en primer lugar debemos definir claramente la estructura de los ficheros de idioma, que tendrán la extensión `.bgr`. En el siguiente cuadro mostramos un ejemplo de fichero, llamado `ejemplo.bgr` (lo encontramos en el directorio `data`) bien formado, donde se distinguen:

- Línea 1: Contiene la palabra clave `MP-BIGRAMAS_IDIOMA-T-1.0`.
- Línea 2: Contiene el nombre del idioma (en minúscula), con el término `unknown` para indicar que es un idioma desconocido. Por ejemplo, podemos considerar `español, ingles, frances`, etc.
- Línea 3: El número de bigramas, $n$ que contienen el fichero, en nuestro caso 5
- Restantes $n$ líneas: cada una contiene dos ítems, el primero dos caracteres que representan el bigrama y el segundo su frecuencia. Se asume que en un fichero de idioma bien formado no puede haber dos líneas que representen el mismo bigrama.

```
MP-BIGRAMAS_IDIOMA-T-1.0
unknown
5
de 4
no 2
an 1
ar 1
br 1
```

Repasaremos las funciones más importantes que tenemos que considerar

- `bool Idioma::salvarAFichero(const char *fichero) const;` Serializa y guarda un idioma en un archivo de texto, devolviendo true si la operación se ha realizado con éxito o false en caso contrario.
- `bool cargarDeFichero(const char *fichero);` Recupera una serialización de un idioma desde un fichero y lo almacena en un objeto de tipo idioma. Devuelve true si la operación se ha realizado con éxito y false en caso contrario.
- `void setPosicion(int p, const Bigrama & bg);` Hace que en la posición $p$ del idioma se almacene el bigrama $bg$. Se asume que $p$ es una posición correcta en el idioma (en otro caso, el comportamiento queda indefinido).
- `bool insertarBigrama(const Bigrama & bg);` Busca el Bigrama representado por `bg.getBigrama()` en el idioma. Si lo encuentra actualiza su frecuencia a la de `bg.getFrecuencia()`. En caso contrario, añade un nuevo Bigrama al idioma, aumentando su tamaño (número de Bigramas distintos) en 1. Como hemos visto, con este método puede ocurrir que el número de bigramas en el idioma aumente, por lo que puede ser necesario liberar/reservar memoria para permitir aumentar el tamaño del idioma correctamente, en cuyo caso devuelve true. Si el bigrama $bg$ ya está en el idioma devuelve falso.

Para comprobar la correcta ejecución de nuestro programa deberemos utilizar el siguiente código, que compilaremos con el nombre ejecutable pruebaldioma. Un ejemplo de llamada al mismo será

```
> ./bin/pruebaldioma an ./data/ejemplo.bgr
```

En este caso buscará el bigrama asociado a la cadena `"an"` en el fichero `ejemplo.bgr`. La salida será

```
> El bigrama an tiene una frecuencia de 1 en el idioma unknown
```

Además, modificará el fichero `ejemplo.bgr` para contener:

```
MP-BIGRAMAS_IDIOMA-T-1.0
unknown
10
de 4
no 2
an 1
ar 1
br 1
xa 100
xb 100
xc 100
xd 100
```

```
xe 100
```

```
/**
 * @file main.cpp
 * @author DECSAI
*/
#include <iostream>
#include <cstring>
#include "Bigrama.h"
#include "Idioma.h"
using namespace std;
int main(int narg, char * args[]) {
    char cadena[3];
    Idioma id;
    int frec, pos;

    if (narg < 3) {
        cerr << "Error en la llamada.\n
    pruebaIdioma <bigrama> <fich1.bgr> <fich2.bgr> ..."<<endl;
        return 1;
    }
    strncpy(cadena, args[1],2);
    cadena[2]='\0';
    for(int i=2; i<narg; i++) {
      if (id.cargarDeFichero(args[i])) {
         pos = id.findBigrama(cadena);
         if (pos >= 0)
           cout << endl << "El bigrama "<<cadena << " tiene una
    frecuencia de " << id.getPosicion(pos).getFrecuencia() << "
    en el idioma " << id.getIdioma() << endl;
         else
           cerr << endl << "El bigrama "<<cadena << " no aparece
    en el idioma " << id.getIdioma() << endl;


        Bigrama bg;
        cadena[0]='x';
        bg.setFrecuencia(100);
        for (int i=0;i<5;i++){
           cadena[1]='a'+i; cadena[2]='\0';
           bg.setBigrama(cadena);
           id.insertarBigrama(bg);
        }

        id.salvarAFichero(args[i]);
        }
    }
    id.liberarMemoria();
    return 0;
}
```

## 5.1. Algunas notas sobre el programa anterior

El programa anterior recibe los argumentos desde línea de comandos. En esta sección veremos como tratar estos argumentos considerando dos posibilidades

### 5.1.1. Llamando al programa desde un terminal

En este caso, asumimos que el directorio de trabajo es el directorio donde está el ejecutable. Entonces, podremos ejecutar el programa como:

```
> pruebaIdioma an PATH_TO/ejemplo.bgr PATH_TO/spanish.bgr
```

donde `PATH_TO` representa el camino hasta llegar a los ficheros `ejemplo.bgr` y `spanish.bgr`. En cualquier caso, al ejecutar el programa dichas variables son recibidas como parámetros de la función

```
int main( int narg, char * args[])
```

En concreto:

- `narg` representa el número de argumentos introducidos, incluyendo el nombre del programa, en nuestro ejemplo 4. Por tanto, la variable `narg` como mínimo valdrá 1, pues el valor que obtiene al considerar el propio nombre del programa como primer argumento.
- `args` es un array de tamaño `narg` de c-string (array de puenteros a char), donde en nuestro caso tendremos que:
  - `args[0]` es: `pruebaIdioma`
  - `args[1]` es: `an`
  - `args[2]` es: `PATH_TO/ejemplo.bgr`
  - `args[3]` es: `PATH_TO/spanish.bgr`

### 5.1.2. Ejecutamos el programa desde Netbeans

Cuando ejecutemos nuestro programa desde el entorno NetBean, será necesario modificar las propiedades del proyecto para indicar cuales serán los argumentos que se les pasa al programa. Para ello, nos situamos sobre el proyecto en la ventana `Projects->Properties`. Seleccionamos la categoría `run` y modificamos la opción `Run command` de forma que quede como se indica

```
"${OUTPUT_PATH}" an ./data/ejemplo.bgr ./data/spanish.bgr
```

En este caso, el `PATH_TO` hasta los ficheros de datos debe ser el path relativo que se necesita para alcanzar los ficheros desde el directorio raíz del proyecto, en nuestro caso dichos ficheros se ubican en el subdirectorio `data`. Indicar que `"${OUTPUT_PATH}"` es una constante que utiliza NetBeans para referenciar el nombre completo del ejecutable creado por el entorno.