Metodología de la Programación

DGIM

Curso 2020/2021

Guion de prácticas

Compilación separada. Automatización con ficheros Makefile

Mayo de 2021

Índice

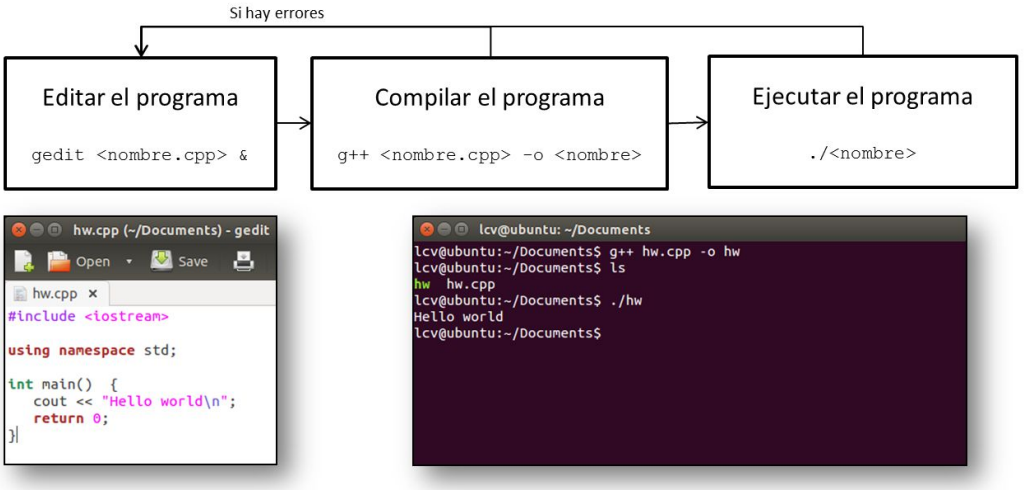

Figura 1: Ciclo de vida de la compilación y ejecución de programas en C++ en Linux

1. Objetivos

Para esta sesión de prácticas, el estudiante deberá entender los conceptos relacionados con la modularización y la compilación separada al tiempo que se revisan los conceptos de clases y vectores de objetos. (ver tema 12 del libro Garrido, A. "**Fundamentos de programación en C++**", Delta Pub., 2005).

Se recuerda que la copia de código no aporta nada al aprendizaje y será considerada como un incumplimiento de las normas de la asignatura con las consecuencias que ello implica, según la normativa de la Universidad de Granada.

2. Un breve apunte sobre la compilación de C++ en Linux

Vamos a seguir el ciclo de edición-compilación-ejecución en una instalación de Linux estándar.

2.1. Edición

Los ficheros con extensión cpp se pueden editar con programas estandar de la consola de Linux como **vi**, **vim** o en modo gráfico (Gnome) con programas como **gedit** (Ver Figura 1).

2.2. Compilación

Una vez guardado el programa en un fichero con extensión **cpp** el siguiente paso es compilarlo con **g++** mediante la orden mostrada en la Figura 1.

Los errores y/o advertencias que pueda generar el compilador son del mismo tipo de las que se obtenían al utilizar tanto **DevC++** como **CodeBlocks**. Dichos entornos utilizan una versión de **g++** para Windows, y por tanto, la gestión de los errores no deberá representar ningún problema[1].

2.3. Ejecución

Si la compilación ha funcionado correctamente, la ejecución del programa se realiza tal y como muestra la Figura 1, observando en la consola el resultado.

3. Un proyecto de ejemplo: Secuencia

El programa a utilizar en esta entrega utiliza dos tipos de datos representados por las clases **SecuenciaCaracteres** y **VectorSecuencias** que van a representar una palabra vista como una secuencia de caracteres y un conjunto de palabras como un vector de palabras respectivamente.

La clase **SecuenciaCaracteres** es una estructura con tres campos y un método privado. El primer campo representa el tamaño máximo para la secuencia de caracteres (**TAMANIO**), el segundo es un vector de caracteres que representará a la secuencia de caracteres (**v**) y el tercero un contador de las posiciones usadas para representar la secuencia (**utilizados**). Por último, veáse el código del cuadro siguiente donde figura la definición de la clase además de un método privado (**contarOcurrencias**) que devuelve el número de ocurrencias en la secuencia del carácter pasado como parámetro:

```
class SecuenciaCaracteres{
private:
    static const int TAMANIO = 5000;
    char v[TAMANIO];
    int utilizados;
    int contarOcurrencias(char caracter){
        int contador=0;
        for(int i=0; i < utilizados; i++){
            if(v[i] == caracter){
                contador++;
            }
        }
        return contador;
    }
```

Además la clase **SecuenciaCaracteres** contiene una serie de métodos públicos disponibles:

```
SecuenciaCaracteres();
int obtenerUtilizados();
int capacidad();
void aniade(char nuevo);
char elemento(int indice);
int primeraOcurrencia(char aBuscar);
SecuenciaCaracteres subsecuencia(int posIni, int posFin);
bool esPalisimetrica();
```

en el que destacaremos el método **esPalisimetrica**[2] que devuelve un dato booleano que indica cuándo la secuencia de caracteres es palisimétrica, es decir, si cuando se parte por la mitad, da como resultado dos se-

[1] Para más detalles sobre el proceso de compilación y ejecución en Linux, consulte el Guión "Introducción a la Compilación de Programas en C++"

[2] Este método es parte de un examen de Fundamentos de la Programación del curso 2020/2021.

cuencias que tienen las mismas letras con las mismas frecuencias. Si la longitud de la secuencia es un número impar, la decisión se toma sin considerar la letra central. Por ejemplo la secuencia de caracteres `gaga` es palisimétrica. Las dos mitades `ga` y `ga` tienen los mismos caracteres con la misma frecuencia. Otros ejemplos de secuencias palisimétricas son `rotor, aabccaba, xyzxy`. La secuencia `abbaab` no es palisimétrica.

La segunda clase **VectorSecuencias** es una estructura con tres campos. El primer campo representa el tamaño máximo para el vector (**TAMA**), el segundo es un vector de secuencias que representará al grupo de palabras (**vector**) y el tercero un contador de las posiciones usadas para representar la secuencia (**utilizados**).

```
class VectorSecuencias{
private:
    static const int  TAMA=20;
    SecuenciaCaracteres vector[TAMA];
    int utilizados;
```

Esta clase cuenta también con varios métodos públicos:

```
VectorSecuencias();
void aniade(SecuenciaCaracteres s);
int capacidad();
SecuenciaCaracteres secuencia(int indice);
int obtenerUtilizados();
```

Además en el fichero `secuencia.cpp` se encuentran varias funciones externas a las clases que son utilizadas en el programa principal main:

```
void pintaSecuencia(SecuenciaCaracteres s);
void pintaVector(VectorSecuencias v);
int cuantasPalisimetricas(VectorSecuencias v);
```

Con el fin de centrarnos en la compilación separada, se proporciona en **prado** la totalidad del código que implementa un programa que comprueba si varias secuencias de caracteres son polisimétricas (ver sección 9.1).

Descargue este programa (fichero **secuencia.zip**) en una carpeta llamada `secuenciaV1` y descomprímalo. En la Figura 2 puede verse la estructura de directorios que resulta al descomprimir el fichero .zip.

```
./
├── bin
├── data
│   └── palisimetrica.dat
├── doc
│   └── secuencia.doxy
└── src
    └── secuencia.cpp
```

Figura 2: Contenido del fichero **secuencia.zip**

A continuación compílelo como sigue.

```
g++ src/secuencia.cpp -o bin/palisimetrica
```

Esta orden creará el ejecutable binario llamado **palisimetrica** en la carpeta `bin`.

A continuación ejecute el programa, para comprobar su correcto funcionamiento.

```
./bin/palisimetrica
```

Puede introducir varios ejemplos como los que están a continuación (4 primeras líneas) para comprobar que el programa proporciona la siguiente salida:

```
gaga
rotor
aabcaba
abbaab
Secuencias Iniciales
gaga
rotor
aabcaba
abbaab

Hay 3 y 1 No Palisimetricas

Secs Palisimetricas
gaga
rotor
aabcaba

Secs No Palisimetricas
abbaab
```

Para parar la ejecución basta indicar con el teclado el indicador `EOF` (end of file), que en el caso de Linux es `CTRL+D`.

Se proporciona también un fichero de validación `polisimetrica.dat` que puede probarse mediante redireccionamiento de la entrada de datos usando el operador < e indicando la carpeta donde están los datos:

```
./bin/palisimetrica < ./data/palisimetrica.dat
```

Esta orden proporcionará la siguiente salida:

```
Secuencias Iniciales
gaga
rotor
aabcaba
abbaab

Hay 3 y 1 No Palisimetricas

Secs Palisimetricas
gaga
rotor
aabcaba

Secs No Palisimetricas
abbaab
```

El zip descargado también incorpora un fichero de documentación automática **secuencia.doxy** que se explicará en un guión de prácticas independiente y requiere tener instalado el programa **doxygen**.

```
sudo apt install doxygen
```

Para generar y visualizar esta documentación de forma automática se procederá de la siguiente forma:

```
doxygen ./doc/secuencia.doxy
firefox doc/html/index.html &
```

La primera orden crea la documentación para ser consultada en formato web o para ser utilizada en LaTeX. La segunda orden abre la documentación en el navegador web Firefox (la Figura 3 muestra parte de esta documentación generada de forma automática).

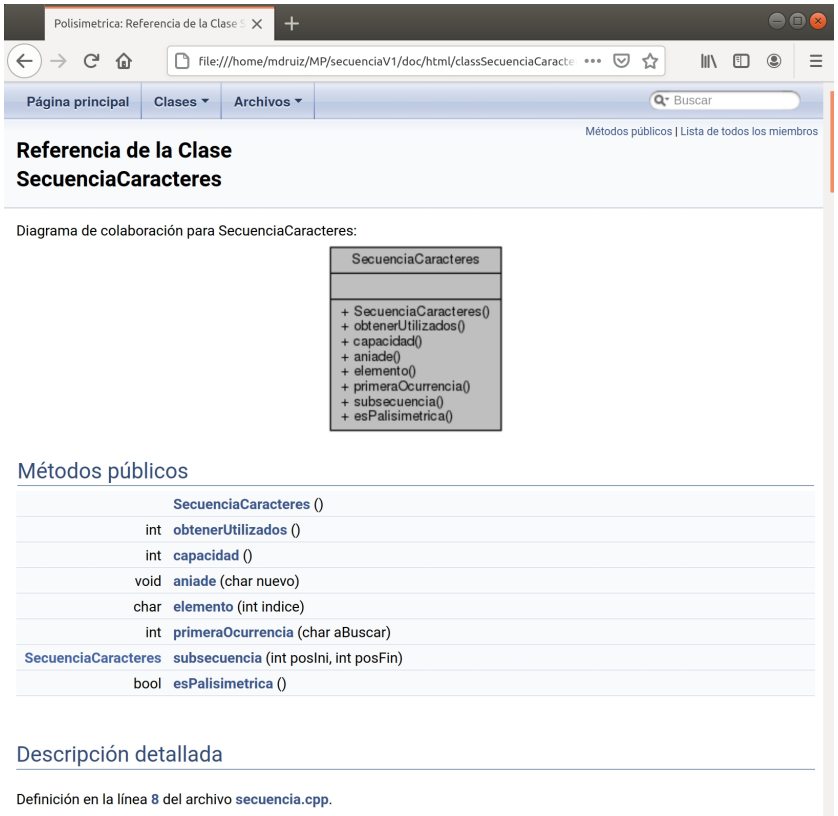

Figura 3: Parte de la documentación automática generada por doxygen

4. Versión 2. Implementación en módulos separados

Cree una nueva carpeta llamada **secuenciaV2** donde se hará una nueva versión del programa. A continuación, deberá dividir el programa que ya ha compilado y ejecutado en la sección anterior en los siguientes módulos distintos (ver estructura de ficheros en la sección 9.2):

1. Módulo *Secuencia*: implementado en *secuencia.h* y *secuencia.cpp*. Contiene el código para manejar el tipo de datos *SecuenciaCaracteres*.

2. Módulo *VectorSecuencias*: implementado en *vectorSecuencias.h* y *vectorSecuencias.cpp*. Contiene el código para manejar el tipo de datos *vectorSecuencias*.

3. Módulo *Prueba*: implementado en *prueba.cpp*. Contiene el código que implementa el programa `main` donde se comprueban si varias secuencias de caracteres son palisimétricas.

Tenga en cuenta que para evitar dobles inclusiones, los ficheros `.h` deben contener directivas del preprocesador de la forma siguiente:

```
#ifndef _FICHERO_H_
#define _FICHERO_H_
...
#endif
```

En los ficheros `.h` se incluirán la definición de las estructuras y los prototipos de las funciones (su cabecera más punto y coma). Por ejemplo, el contenido de `secuencia.h` sería algo así:

```
#ifndef SECUENCIA_H
#define SECUENCIA_H

class SecuenciaCaracteres{
private:
    static const int TAMANIO = 5000;
    char v[TAMANIO];
    int utilizados;
    int contarOcurrencias(char caracter);
public:
    SecuenciaCaracteres(){ utilizados=0; }
    int obtenerUtilizados(){ return utilizados; }
    int capacidad(){ return TAMANIO; }
    ...
    bool esPalisimetrica();
};
void pintaSecuencia(SecuenciaCaracteres s);

#endif
```

Para poder compilar bien los ficheros `cpp` deberán incluirse donde sea necesario, los ficheros `.h` con las directivas:

```
#include "secuencia.h"
```

o

```
#include "vectorSecuencias.h"
```

o ambas.

Cuando tenga todos los módulos, se deberán compilar para obtener los archivos **secuencia.o**, **vectorSecuencias.o** y **prueba.o**.

```
g++ -c secuencia.cpp -o secuencia.o
g++ -c vectorSecuencias.cpp -o secuencia.o
g++ -c prueba.cpp -o prueba.o
```

Una vez que dispone de los dos archivos objeto, deberá enlazarlos para obtener el ejecutable **palisimetrica** de la aplicación propuesta.

```
g++ secuencia.o vectorSecuencias.o prueba.o -o palisimetrica
```

Esto creará el archivo binario palisimetrica que podrá ejecutar para comprobar su funcionamiento y obteniendo la misma salida que en el caso anterior.

```
./palisimetrica
```

Si modificamos el fichero `secuencia.cpp`, ¿qué órdenes sería/n necesarias volver a ejecutar para obtener de nuevo el ejecutable? ¿Y si modificamos `secuencia.h`?

5. Versión 3. Implementación en carpetas separadas

Cree una nueva carpeta llamada `secuenciaV3` donde se hará una nueva versión del programa. Dentro de ella, deberá crear los directorios **include** para los ficheros `.h`, **src** para los ficheros `.cpp`, **obj** para los ficheros `.o`, **lib** para alojar las bibliotecas y **bin** para los ejecutables. Se crearán también las carpetas **doc** para documentación y **data** para ficheros auxiliares y se colocará cada fichero en su sitio (ver Figura 4).

Una vez ordenados los ficheros por carpetas, se procede a compilar, cogiendo cada fichero desde su carpeta y colocando cada fichero de salida en su carpeta correspondiente (Figura 5).

```
g++ -c src/secuencia.cpp -o obj/secuencia.o -Iinclude
g++ -c src/vectorSecuencias.cpp -o obj/vectorSecuencias.o -Iinclude
g++ -c src/prueba.cpp -o obj/prueba.o -Iinclude
g++ obj/secuencia.o obj/vectorSecuencias.o obj/prueba.o -o bin/palisimetrica
```

Compruebe de nuevo que el binario se ha creado correctamente ejecutándolo:

```
./bin/palisimetrica < ./data/palisimetrica.dat
```

```
./
├── bin
├── data
│   └── palisimetrica.dat
├── doc
│   └── secuencia.doxy
├── include
│   ├── secuencia.h
│   └── vectorSecuencias.h
├── lib
├── obj
└── src
    ├── prueba.cpp
    ├── secuencia.cpp
    └── vectorSecuencias.cpp
```

Figura 4: Estructura básica de las carpetas de un proyecto con múltiples módulos antes de compilar y enlazar

6. Versión 4. Creación y uso de una biblioteca

Cree una nueva carpeta llamada `secuenciaV4` donde se hará una nueva versión del programa. En esta sesión deberá crear una biblioteca sobre la estructura creada en la Sección 5. Concretamente, debe realizar las siguientes tareas:

1. Compile los ficheros sin enlazar el binario aún:

```
g++ -c src/secuencia.cpp -o obj/secuencia.o -Iinclude
g++ -c src/vectorSecuencias.cpp -o obj/vectorSecuencias.o -Iinclude
g++ -c src/prueba.cpp -o obj/prueba.o -Iinclude
```

2. Cree una biblioteca con los archivos **secuencia.o** y **vectorSecuencias.o** con nombre **libsecuencias.a**.

```
ar rvs lib/libsecuencias.a obj/secuencia.o  obj/vectorSecuencias.o
```

   Esto dará la siguiente salida:

```
ar: creando lib/libsecuencias.a
a - obj/secuencia.o
a - obj/vectorSecuencias.o
```

3. Ejecute la orden para crear el ejecutable utilizando la biblioteca que acabamos de crear, es decir, sin enlazar directamente con los archivos objeto.

```
g++ obj/prueba.o -lsecuencias -o bin/palisimetrica -Llib
```

```
./
├── bin
│   └── palisimetrica
├── data
│   └── palisimetrica.dat
├── doc
│   └── secuencia.doxy
├── include
│   ├── secuencia.h
│   └── vectorSecuencias.h
├── lib
├── obj
│   ├── prueba.o
│   ├── secuencia.o
│   └── vectorSecuencias.o
└── src
    ├── prueba.cpp
    ├── secuencia.cpp
    └── vectorSecuencias.cpp
```

Figura 5: Estructura básica de las carpetas de un proyecto con múltiples módulos después de compilar y enlazar

Después, ejecute el binario creado para comprobar que se ha creado correctamente.

Indicar también que, cuando se va a usar una biblioteca, es necesario incorporar los **.h** propios de la biblioteca, en nuestro proyecto.

4. Indique de nuevo qué debemos hacer si modificamos el fichero `secuencia.h`, ¿qué órdenes serían necesarias volver a ejecutar para obtener de nuevo el ejecutable? ¿Y si modificamos `secuencia.cpp`?

```
./
├── bin
│   └── palisimetrica
├── data
│   └── palisimetrica.dat
├── doc
│   └── secuencia.doxy
├── include
│   ├── secuencia.h
│   └── vectorSecuencias.h
├── lib
│   └── libsecuencias.a
├── obj
│   ├── prueba.o
│   ├── secuencia.o
│   └── vectorSecuencias.o
└── src
    ├── prueba.cpp
    ├── secuencia.cpp
    └── vectorSecuencias.cpp
```

Figura 6: Estructura básica de las carpetas de un proyecto con múltiples módulos después de crear y usar una biblioteca

7. Versión 5. Automatización de la compilación con ficheros Makefile

Cree una nueva carpeta llamada `secuenciaV5` donde haremos una nueva versión del programa a partir de la versión 4 (misma estructuración en carpetas) copiando los ficheros en las mismas carpetas.

En esta versión construiremos un fichero **Makefile** para gestionar automáticamente la compilación de la aplicación. El archivo **Makefile** se construirá con un editor de texto, y se colocará en la carpeta padre de `src` (ver Figura 7).

Para facilitar el desarrollo de la práctica, así como el estudio de las dependencias que se reflejan en el archivo **Makefile**, se proporciona en la Figura 7 un esquema de los archivos y sus relaciones.

Puesto que conocemos las dependencias entre los ficheros de nuestro proyecto, tras hacer modificaciones en un módulo no necesitamos reconstruir el proyecto completo. Cuando modificamos algún fichero solo es necesario reconstruir los ficheros que dependen de él, lo que hace que a su vez sea necesario reconstruir los ficheros que dependan de los nuevos ficheros reconstruidos, y así sucesivamente. El problema es que, dependiendo del módulo al que afecten las modificaciones, lo que tenemos que reconstruir podrá variar. Por ejemplo, si modificamos `secuencia.cpp` debemos reconstruir `secuencia.o`, `libsecuencias.a`, y el ejecutable `palisimetrica`. Sin embargo, si modificamos `prueba.cpp` solo tendremos que reconstruir `prueba.o` y el ejecutable.

```
./
├── Makefile
├── bin
│   └── palisimetrica
├── data
│   └── palisimetrica.dat
├── doc
│   └── secuencia.doxy
├── include
│   ├── secuencia.h
│   └── vectorSecuencias.h
├── lib
│   └── libsecuencias.a
├── obj
│   ├── prueba.o
│   ├── secuencia.o
│   └── vectorSecuencias.o
├── src
│   ├── prueba.cpp
│   ├── secuencia.cpp
│   └── vectorSecuencias.cpp
└── zip
```

Figura 7: Estructura básica de las carpetas de un proyecto con múltiples módulos

Para evitar acordarnos de qué partes se han de reconstruir en función de cuales se han modificado, disponemos de herramientas que automatizan todo el proceso. En nuestro caso usaremos la utilidad **make**. Su cometido es leer e interpretar un fichero en el que se almacenan una serie de reglas que reflejan el esquema anterior de dependencias. Lo habitual es que a este fichero de reglas se le llame `Makefile` o `makefile`, aunque podría cambiar. El aspecto de una regla es el siguiente:

```
1 Objetivo : Lista de dependencias
2     Acciones
```

Donde:

- `Objetivo`: Esto es lo que queremos construir. Habitualmente es el nombre de un fichero que se obtendrá como resultado del procesamiento de otros ficheros. Más adelante veremos que no tiene que ser, necesariamente, un nombre de fichero.
- `Lista de dependencias`: Esto es una lista de ítems de los que depende la construcción del objetivo de la regla. Normalmente los ítems son ficheros o nombres de otros objetivos. Al dar esta lista de dependencias, la utilidad **make** debe asegurarse de que han sido todas satisfechas antes de poder alcanzar el objetivo de la regla.

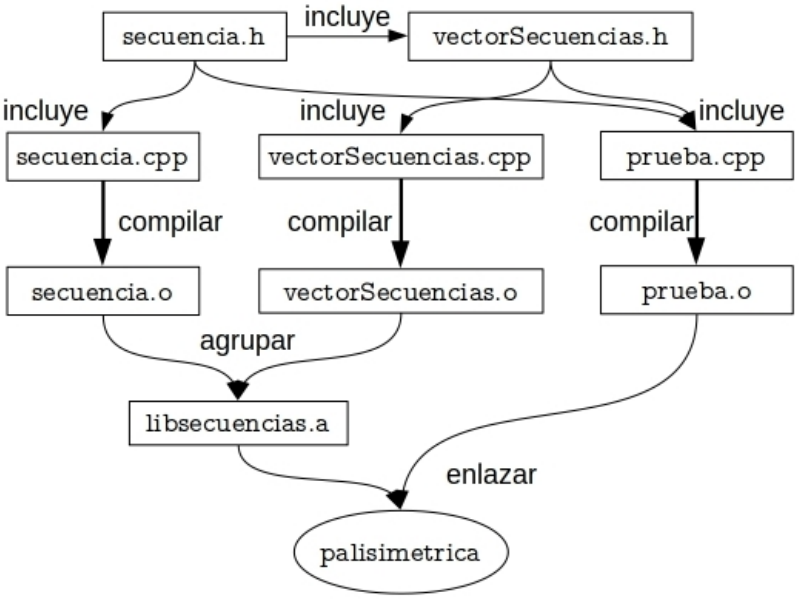

Figura 8: Módulos y relaciones para el programa "palisimetrica". Las flechas indican dependencias: (1) Las dependencias de inclusión se deben añadir tanto si es inclusión directa como indirecta y (2) El resto de dependencias se especifican solo si son directas.

- `Acciones`: Este es el conjunto de acciones que se deben llevar a cabo para conseguir el objetivo. Normalmente serán instrucciones como las que hemos visto antes para compilar, enlazar, etc.

IMPORTANTE: Observe que al escribir estas reglas en el fichero **Makefile**, las acciones se deben sangrar (o tabular) con un carácter tabulador. Si se rellena con espacios, el programa **make** no es capaz de entender la regla.

Por ejemplo, la siguiente sería una regla válida:

```
1 bin/palisimetrica : obj/prueba.o lib/libsecuencias.a
2     g++ -o bin/palisimetrica obj/prueba.o -Llib/ -lsecuencias
```

donde el objetivo es la construcción del ejecutable `bin/palisimetrica`. En la lista de dependencias hemos puesto la lista de ficheros de los que depende, o lo que es lo mismo, estamos diciendo que para poder construir `bin/palisimetrica`, previamente han de haber sido construidos adecuadamente los ficheros `obj/prueba.o` y `lib/libsecuencias.a`. Finalmente la acción que hay que llevar a cabo para generar el objetivo es la ejecución de **g++** con los parámetros que vemos en el ejemplo.

Si al aplicar la regla, se detecta que alguno de los ítems de la lista de dependencias no existe o necesita ser actualizado, entonces se ejecutarán las reglas necesarias para crearlo de nuevo antes de aplicar las acciones de la regla actual.

En realidad, **make** toma uno por uno los ítems de la lista de dependencias e intenta buscar alguna regla que le diga cómo se debe construir ese ítem. De esta forma, si por ejemplo está analizando el ítem

`lib/libsecuencias.a`, se buscará una regla cuyo objetivo sea ese. En nuestro ejemplo, esa regla debería ser la siguiente:

```
lib/libsecuencias.a : obj/secuencia.o obj/vectorSecuencias.o
    ar rsv lib/libsecuencias.a obj/secuencia.o obj/vectorSecuencias.o
```

de esta forma, para saber si `lib/libsecuencias.a` necesita ser actualizada se aplicará la regla de forma análoga a como hicimos con la regla anterior. Al analizar la lista de dependencias, intentará comprobar si alguno de los ítems `obj/secuencia.o obj/vectorSecuencias.o` necesita ser actualizado y buscará reglas con ese objetivo. En nuestro ejemplo las reglas podrían ser estas:

```
obj/secuencia.o : src/secuencia.cpp include/secuencia.h
        g++ -c src/secuencia.cpp -o obj/secuencia.o -I./include

obj/vectorSecuencias.o : src/vectorSecuencias.cpp include/vectorSecuencias.h include/secuencia.h
        g++ -c src/vectorSecuencias.cpp -o obj/vectorSecuencias.o -I./include
```

Ahora, por ejemplo la lista de dependencias para la regla `obj/vectorSecuencias.o` contiene los ítems:
`src/vectorSecuencias.cpp`,
`include/vectorSecuencias.h`
`include/secuencia.h`.
Al tratarse de ficheros de código fuente, no será frecuente que haya reglas cuyo objetivo sea obtener estos, ya que esos ficheros no se van a crear a partir de otros ficheros sino que serán creados manualmente en un editor de textos por parte del programador. En esta situación, para averiguar si es necesario volver a construir el objetivo `obj/secuencia.o`, se comprobará si se da o no alguna de las siguientes situaciones:

- Que el objetivo no exista. Esta situación indica que el código fuente no fue compilado con anterioridad y que, por lo tanto, se ejecutarán las acciones de la regla para compilarlo y generar el código objeto.
- Que el objetivo ya exista pero su fecha de modificación sea anterior a la del fichero de código fuente. En este caso es evidente que, aunque con anterioridad ya fue compilado el código fuente y fue creado el código objeto, en algún momento posterior el programador ha modificado dicho código fuente. Esto implica que el código objeto necesita ser actualizado, es decir, que hay que aplicar las acciones de la regla para volver a compilarlo.
- Si no se da ninguna de las dos situaciones anteriores, entonces es que el objetivo está actualizado y no se aplicarán las acciones.

Misma forma de proceder para el objetivo `obj/prueba.o`.

```
obj/prueba.o: src/prueba.cpp include/secuencia.h include/vectorSecuencias.h
   g++ -c src/prueba.cpp -o obj/prueba.o -Iinclude
```

Por tanto, al aplicar la primera regla (`bin/palisimetrica`), si por ejemplo hemos hecho cambios en `src/secuencia.cpp`, se aplicarán de manera sucesiva (y ordenada) todas las reglas necesarias para ir construyendo todos los ficheros intermedios del proceso de compilación y enlazado. El encadenamiento de reglas lo gestiona **make** de manera automática y no debemos preocuparnos por el orden en el que escribimos las reglas en el fichero `Makefile`; él sabe buscar la regla adecuada en cada momento.

7.1. Ejecutando make

Si ejecutamos **make** *sin ningún parámetro*, intentará aplicar la **primera regla** que se encuentre en el fichero `Makefile`. Si queremos aplicar otra regla, debemos llamar a **make** usando como parámetro el objetivo de dicha regla. Por ejemplo, en el caso anterior, y suponiendo que la regla `bin/palisimetrica` no es la primera que hemos escrito (recuerda que el orden de las reglas en `Makefile` no es importante), debemos ejecutar:

```
make bin/palisimetrica
```

Esta ejecución de **make** construirá la aplicación **palisimetrica** y todos los ficheros necesarios para conseguir dicho objetivo.

La salida obtenida al ejecutar make o la orden anterior cuando aún no hay ningún fichero objeto, ni librerías creadas, sería la siguiente:

```
g++ -c src/prueba.cpp -o obj/prueba.o  -I./include
g++ -c src/secuencia.cpp -o obj/secuencia.o -I./include
g++ -c src/vectorSecuencias.cpp -o obj/vectorSecuencias.o -I./include
ar rsv lib/libsecuencias.a obj/secuencia.o obj/vectorSecuencias.o
ar: creando lib/libsecuencias.a
a - obj/secuencia.o
a - obj/vectorSecuencias.o
g++ -o bin/palisimetrica obj/prueba.o -Llib/ -lsecuencias
```

Si por el contrario ya está todo creado la salida será un mensaje como el siguiente:

```
make: 'bin/palisimetrica' está actualizado.
```

Un `Makefile` puede construir más de una aplicación por ello es frecuente encontrarse como primera regla una entrada llamada `all:`

```
1 all : bin/palisimetrica bin/main2
```

Observe que no tiene acciones asociadas. Al haber sido escrita en primer lugar en `Makefile`, bastará con ejecutar:

```
make
```

para que sea aplicada. Al aplicarla, lo primero que se intenta hacer es verificar si es necesario actualizar alguna de sus dependencias, que son los ejecutables **palisimetrica** y **main2**. Si aún no han sido generados, se crean (aplicando otras reglas). Una vez que ha terminado ese proceso, habría que pasar a aplicar las acciones de esta regla, pero como no tiene, no se hace nada más. El resultado es que ejecutando simplemente **make** hemos conseguido crear todos los ejecutables del proyecto.

7.2. Otras reglas estándar

A veces se hace necesario borrar ficheros que se consideran temporales o ficheros que no se van a necesitar con posterioridad. Por ejemplo, una vez acabado el proyecto definitivamente, no serán necesarios los códigos objeto ni, probablemente, las bibliotecas por lo que podemos borrarlos.

En este sentido se suelen incorporar algunas reglas que hacen estas tareas de limpieza. Es frecuente considerar dos niveles de limpiado del proyecto. Un primer nivel que limpia ficheros intermedios pero deja las aplicaciones finales que hayan sido generadas, `clean_blando`. Y un segundo nivel `clean` que elimina todos los archivos binarios, lo que permite migrar una aplicación a otra máquina.

```
1 clean:
2     @echo "Limpiando..."
3     rm obj/*.o lib/*.a bin/* -r doc/html doc/latex
```

De esta forma, tras acabar de programar el proyecto podemos quedarnos únicamente con el código fuente original y los ejecutables. Observe que esta regla no tiene dependencias por lo que al aplicarla, directamente se pasa a ejecutar sus acciones.

Otras reglas estándar podrían dedicarse a la generación automática de la documentación o al proceso de limpiar y comprimir el proyecto, listo para la entrega.

```
1 doxy:
2         doxygen doc/secuencia.doxy
3
4 zip: clean
5         zip -r zip/compSeparadaMake.zip *
```

Si recuperamos todas las instrucciones utilizadas hasta el momento con sus respectivos objetivos obtenemos:

```
all : bin/palisimetrica

bin/palisimetrica : obj/prueba.o lib/libsecuencias.a
  g++ -o bin/palisimetrica obj/prueba.o -Llib/ -lsecuencias

lib/libsecuencias.a : obj/secuencia.o obj/vectorSecuencias.o
  ar rsv lib/libsecuencias.a obj/secuencia.o obj/vectorSecuencias.o

obj/secuencia.o : src/secuencia.cpp include/secuencia.h
  g++ -c src/secuencia.cpp -o obj/secuencia.o -I./include

obj/vectorSecuencias.o : src/vectorSecuencias.cpp include/vectorSecuencias.h include/secuencia.h
  g++ -c src/vectorSecuencias.cpp -o obj/vectorSecuencias.o -I./include

obj/prueba.o : src/prueba.cpp include/secuencia.h include/vectorSecuencias.h
  g++ -c src/prueba.cpp -o obj/prueba.o  -I./include

clean :
  @echo "Limpiando..."
  rm obj/*.o lib/*.a bin/* -r doc/html doc/latex

doxy :
  doxygen doc/secuencia.doxy

zip : clean
  zip -r zip/compSeparadaMake.zip *
```

Makefile

8. Versión 6. Uso de macros en makefiles

Uno de los problemas que podemos ver en el fichero **makefile** anterior, es que se repiten por todos lados los nombres de los directorios `bin`, `include`, `lib`, `obj` y `src`. Qué pasa ahora si decidimos cambiar el nombre de alguno de ellos? Tendríamos que cambiar una por una todas las ocurrencias de dichos directorios. Mediante el uso de *macros*, que son variables o cadenas que se expanden cuando realizamos una llamada a **make**, conseguimos que los nombres de los directorios solo los tengamos que escribir una vez. Por ejemplo, incluiremos al principio de nuestro **makefile**, una macro para el directorio `bin` de la forma `BIN = bin` y luego siempre que queramos hacer referencia al directorio `bin` escribiremos `$(BIN)`.

Si se modifica el anterior fichero makefile para que use macros de este tipo, en lugar de directamente los nombres de los directorios obtenemos:

```
# Definición de macros para definir las carpetas de trabajo
BIN = ./bin
OBJ = ./obj
SRC = ./src
INC = ./include
LIB = ./lib
ZIP = ./zip
DOC = ./doc
# Opciones de compilación
# -Wall muestra todas las advertencias
# -g compila en modo "depuración"
OPT= -std=c++14 -Wall -g
# Nombre de la práctica
PRJ=compSeparadaMake

# Las macros se usan en las reglas del makefile como si fuesen variables
# que se sustituyen por su valor definido anteriormente

all : $(BIN)/palisimetrica

$(BIN)/palisimetrica : $(OBJ)/prueba.o $(LIB)/libsecuencias.a
  g++ -o $(BIN)/palisimetrica $(OBJ)/prueba.o -L$(LIB)/ -lsecuencias

$(LIB)/libsecuencias.a : $(OBJ)/secuencia.o $(OBJ)/vectorSecuencias.o
  ar -rsv $(LIB)/libsecuencias.a $(OBJ)/secuencia.o $(OBJ)/vectorSecuencias.o

$(OBJ)/secuencia.o : $(SRC)/secuencia.cpp $(INC)/secuencia.h
  g++ $(OPT) -c $(SRC)/secuencia.cpp -o $(OBJ)/secuencia.o -I$(INC)

$(OBJ)/vectorSecuencias.o : $(SRC)/vectorSecuencias.cpp $(INC)/vectorSecuencias.h $(INC)/secuencia.h
  g++ $(OPT) -c $(SRC)/vectorSecuencias.cpp -o $(OBJ)/vectorSecuencias.o -I$(INC)

$(OBJ)/prueba.o : $(SRC)/prueba.cpp $(INC)/secuencia.h $(INC)/vectorSecuencias.h
  g++ $(OPT) -c $(SRC)/prueba.cpp -o $(OBJ)/prueba.o -I$(INC)

clean :
  @echo "Limpiando..."
  rm $(OBJ)/*.o $(LIB)/*.a $(BIN)/* $(ZIP)/* -r $(DOC)/html $(DOC)/latex

doxy :
  doxygen $(DOC)/secuencia.doxy

zip : clean
  zip -r $(ZIP)/$(PRJ).zip *
```

Makefile con macros

9. Apendice 1. Modularización

9.1. Un solo fichero: secuencia.cpp

```
#include <cstring>
#include <cstdlib>
#include <iostream>

using namespace std;

class SecuenciaCaracteres{
```

```
private:
    /**
     * Constante para el TAMANIO maximo de las secuencias
     */
    static const int TAMANIO = 5000;

    /**
     * secuencia de caracteres
     */
    char v[TAMANIO];

    /**
     * Contador de posiciones usadas
     */
    int utilizados;

    /**
     * Devuelve el contador de ocurrencias de un determinado
     * caracter
     * @param caracter caracter objetivo
     * @return contador de ocurrencias
     */
    int contarOcurrencias(char caracter){
        int contador=0;
        for(int i=0; i < utilizados; i++){
            if(v[i] == caracter){
                contador++;
            }
        }

        // devuelve contador
        return contador;
    }

public:

    /**
     * Constructor por defecto
     * Metodo ya implementado
     * @return
     */
    SecuenciaCaracteres(){
        // basta con hacer que utilizados sea 0
        utilizados=0;
    }

    /**
     * Devuelve el numero de posiciones usadas
     * Metodo ya implementado
     * @return
     */
    int obtenerUtilizados(){
        return utilizados;
    }

    /**
     * Devuelve el espacio total en la secuencia
     * Metodo ya implementado
     * @return
     */
    int capacidad(){
        return TAMANIO;
    }

    /**
     * Agrega un nuevo caracter a la secuencia
     * Metodo ya implementado
     * @param nuevo
     */
    void aniade(char nuevo){
        // se agrega si hay espacio
        if(utilizados < TAMANIO){
            v[utilizados]=nuevo;
            utilizados++;
        }
    }

    /**
     * Se accede al caracter de una posicion.
     * Metodo ya implementado
     * NOTA: no se comprueba la validez del indice
     * @param indice indice objetivo
     * @return caracter almacenado en indice
     */
    char elemento(int indice){
        return v[indice];
    }

    /**
     * Localiza la primera ocurrencia del caracter
     * pasado como argumento
     * Metodo ya implementado
     * @param aBuscar caracter a buscar
     * @return posicion de primera ocurrencia del caracter
     */
    int primeraOcurrencia(char aBuscar){
        int posicion=-1;
        for(int i=0; i < utilizados; i++){
            if(v[i] == aBuscar){
                posicion=i;
```

```
            }
        }

        // se devuelve la posicion
        return posicion;
    }

    /**
     * Construye la subsecuencia contenida entre dos posiciones
     * @param posIni
     * @param posFin
     * @return
     */
    SecuenciaCaracteres subsecuencia(int posIni, int posFin){
        SecuenciaCaracteres resultado;

        // solo se trabaja si las posiciones son validas
        if(posFin >= posIni && posIni >= 0 && posIni < utilizados
                && posFin < utilizados){
            // se agregan los caracteres de interes
            for(int i=posIni; i <= posFin; i++){
                resultado.aniade(v[i]);
            }
        }

        // se devuelve el resultado
        return resultado;
    }

    /**
     * Determina si una palabra cumple la condicion de palisimetrica
     * @return resultado de la comprobacion
     */
    bool esPalisimetrica(){
        bool resultado=true;

        // se generan dos secuencias: una para cada mitad
        int mitad = utilizados/2;

        // la primera secuencia va desde 0 hasta mitad
        SecuenciaCaracteres primera;
        for(int i=0; i < mitad; i++){
            primera.aniade(v[i]);
        }

        // se determina si la cadena tiene numero para o
        // impar de caracteres para determinar el comienzo
        // de la segunda cadena
        if(utilizados %2 != 0){
            mitad = mitad+1;
        }

        SecuenciaCaracteres segunda;
        for(int i=mitad; i < utilizados; i++){
            segunda.aniade(v[i]);
        }

        // se genera una secuencia con los caracteres que
        // aparecen en la secuencia, sin repetidos
        SecuenciaCaracteres sinRepetidos;
        for(int i=0; i < utilizados; i++){
            if(sinRepetidos.primeraOcurrencia(v[i]) == -1){
                sinRepetidos.aniade(v[i]);
            }
        }

        // Se recorren ahora los caracteres de la secuencia
        // sin repetidos y se cuentan las apariciones en cada
        // una de las secuencias (para ello se usa un metodo
        // auxiliar, que consideramos privado)
        for(int i=0; i < sinRepetidos.utilizados && resultado == true; i++){
            int ocurrenciasPrimera = primera.contarOcurrencias(sinRepetidos.v[i]);
            int ocurrenciasSegunda = segunda.contarOcurrencias(sinRepetidos.v[i]);

            // se comprueba la igualdad
            if(ocurrenciasPrimera != ocurrenciasSegunda){
                resultado=false;
            }
        }

        // se deveulve el valor de resultado
        return resultado;
    }
};
////////////////////////////////////////////////////////////////////////////////////////////

class VectorSecuencias{
private:
    /**
     * Constante para el TAMANIO maximo de las secuencias
     */
    static const int  TAMA=20;
    SecuenciaCaracteres vector[TAMA];
    int utilizados;
public:
    /* Constructor por defecto
     * crea vector vacio
     */
    VectorSecuencias(){
        utilizados=0;
```

```
    }
    /**
     * Anade una nueva secuencia al vector
     * @param s secuencia a insertar
     */
    void aniade(SecuenciaCaracteres s){
            if(utilizados < TAMA){
            vector[utilizados++]=s;
            }
    }

    /**
     * Numero mÃ¡ximo de secuencias que se pueden almacenar en el vector
     * @return capacidad
     */
    int capacidad(){
        return TAMA;
    }
    /**
     * Devuelva la secuencia en la posicion dada por indice
     * @param indice posicion a consultar
     * @return secuencia en dicha posicion
     * @pre 0 \<= indice \< obtenerUtilizados()
     */
    SecuenciaCaracteres secuencia(int indice){
        return vector[indice];
    }
    /**
     * Numero de secuencias insertadas en el vector
     * @return  numero de secuencias en el vector
     */
    int obtenerUtilizados(){
        return utilizados;
    }


};

// Funciones genericas

/**
 * muestra una secuencia de caracteres en la salidad estandar
 * @param s secuencia a mostrar
 */
void pintaSecuencia(SecuenciaCaracteres s){
    for (int i=0;i<s.obtenerUtilizados();i++)
        cout << s.elemento(i);

}

/**
 * muestra un vector de secuencias en la salidad estandar
 * @param v vector a mostrar
 */
void pintaVector(VectorSecuencias v){
for (int i=0;i<v.obtenerUtilizados();i++){
    pintaSecuencia(v.secuencia(i));
    cout << endl;
}
}

/**
 * Cuenta el numero de secuencias palisimetricas en el vector
 * @param v el vector
 * @return numero de secs palisimetricas en v
 */
int cuantasPalisimetricas(VectorSecuencias v){
    int contador=0;
    for (int i=0; i<v.obtenerUtilizados(); i++){
        if (v.secuencia(i).esPalisimetrica())
            contador++;
    }
    return contador;
}


// Main

int main() {
    // se lee del flujo de entrada
    VectorSecuencias v;
    VectorSecuencias vpal, vnopal;
    bool continuar = true;
    while(continuar){
        string cadena;
        cin >> cadena;
        if(cadena == ""){
            continuar = false;
        }
        else{

            // se crea la secuencia
            SecuenciaCaracteres s;
            for(int i=0; i < cadena.length(); i++){
                s.aniade(cadena[i]);
            }
            v.aniade(s);

        }
```

```
    }
    cout << "Secuencias Iniciales"<< endl;
    pintaVector(v);
    cout << endl;

    for (int i=0; i<v.obtenerUtilizados();i++){
        // se hace la comprobacion
        bool cond = v.secuencia(i).esPalisimetrica();
        if (cond) {
            vpal.aniade(v.secuencia(i));
        } else {
            vnopal.aniade(v.secuencia(i));
        }
    }
    int np = cuantasPalisimetricas(v);
    cout << "Hay "<<  np << " y " << v.obtenerUtilizados()-np << " No Palisimetricas"<<endl;

    cout << endl << "Secs Palisimetricas"<<endl;
    pintaVector(vpal);
    cout << endl;
    cout << "Secs No Palisimetricas"<< endl;
    pintaVector(vnopal);
    cout<< endl;

    return 0;
}
```

secuencia.cpp

9.2. Módulos separados: secuencia.h, secuencia.cpp, vectorSecuencias.h, vectorSecuencias.cpp, prueba.cpp

```
#ifndef SECUENCIA_H
#define SECUENCIA_H

class SecuenciaCaracteres{
private:
    /**
     * Constante para el TAMANIO maximo de las secuencias
     */
    static const int TAMANIO = 5000;

    /**
     * secuencia de caracteres
     */
    char v[TAMANIO];

    /**
     * Contador de posiciones usadas
     */
    int utilizados;

    /**
     * Devuelve el contador de ocurrencias de un determinado
     * caracter
     * @param caracter caracter objetivo
     * @return contador de ocurrencias
     */
    int contarOcurrencias(char caracter);

public:

    /**
     * Constructor por defecto
     * Metodo ya implementado
     * @return
     */
    SecuenciaCaracteres();

    /**
     * Devuelve el numero de posiciones usadas
     * Metodo ya implementado
     * @return
     */
    int obtenerUtilizados();

    /**
     * Devuelve el espacio total en la secuencia
     * Metodo ya implementado
     * @return
     */
    int capacidad();

    /**
     * Agrega un nuevo caracter a la secuencia
     * Metodo ya implementado
     * @param nuevo
     */
    void aniade(char nuevo);

    /**
     * Se accede al caracter de una posicion.
```

```
     * Metodo ya implementado
     * NOTA: no se comprueba la validez del indice
     * @param indice indice objetivo
     * @return caracter almacenado en indice
     */
    char elemento(int indice);

    /**
     * Localiza la primera ocurrencia del caracter
     * pasado como argumento
     * Metodo ya implementado
     * @param aBuscar caracter a buscar
     * @return posicion de primera ocurrencia del caracter
     */
    int primeraOcurrencia(char aBuscar);

    /**
     * Construye la subsecuencia contenida entre dos posiciones
     * @param posIni
     * @param posFin
     * @return
     */
    SecuenciaCaracteres subsecuencia(int posIni, int posFin);

    /**
     * Determina si una palabra cumple la condicion de palisimetrica
     * @return resultado de la comprobacion
     */
    bool esPalisimetrica();
};

//Funciones genericas
/**
 * muestra una secuencia de caracteres en la salidad estandar
 * @param s secuencia a mostrar
 */
void pintaSecuencia(SecuenciaCaracteres s);


#endif /* SECUENCIA_H */
```

secuencia.h

```
#include <iostream>
#include "secuencia.h"

using namespace std;

/**
 * Devuelve el contador de ocurrencias de un determinado
 * caracter
 * @param caracter caracter objetivo
 * @return contador de ocurrencias
 */
int SecuenciaCaracteres::contarOcurrencias(char caracter) {
    int contador = 0;
    for (int i = 0; i < utilizados; i++) {
        if (v[i] == caracter) {
            contador++;
        }
    }

    // devuelve contador
    return contador;
}

/**
 * Constructor por defecto
 * Metodo ya implementado
 * @return
 */
SecuenciaCaracteres::SecuenciaCaracteres(){
    // basta con hacer que utilizados sea 0
    utilizados=0;
}

/**
 * Devuelve el numero de posiciones usadas
 * Metodo ya implementado
 * @return
 */
int SecuenciaCaracteres::obtenerUtilizados(){
    return utilizados;
}

/**
 * Devuelve el espacio total en la secuencia
 * Metodo ya implementado
 * @return
 */
int SecuenciaCaracteres::capacidad(){
    return TAMANIO;
}

/**
 * Agrega un nuevo caracter a la secuencia
 * Metodo ya implementado
 * @param nuevo
 */
```

```
void SecuenciaCaracteres::aniade(char nuevo) {
    // se agrega si hay espacio
    if (utilizados < TAMANIO) {
        v[utilizados] = nuevo;
        utilizados++;
    }
}

/**
 * Se accede al caracter de una posicion.
 * Metodo ya implementado
 * NOTA: no se comprueba la validez del indice
 * @param indice indice objetivo
 * @return caracter almacenado en indice
 */
char SecuenciaCaracteres::elemento(int indice){
    return v[indice];
}

/**
 * Localiza la primera ocurrencia del caracter
 * pasado como argumento
 * Metodo ya implementado
 * @param aBuscar caracter a buscar
 * @return posicion de primera ocurrencia del caracter
 */
int SecuenciaCaracteres::primeraOcurrencia(char aBuscar) {
    int posicion = -1;
    for (int i = 0; i < utilizados; i++) {
        if (v[i] == aBuscar) {
            posicion = i;
        }
    }

    // se devuelve la posicion
    return posicion;
}

/**
 * Construye la subsecuencia contenida entre dos posiciones
 * @param posIni
 * @param posFin
 * @return
 */
SecuenciaCaracteres SecuenciaCaracteres::subsecuencia(int posIni, int posFin) {
    SecuenciaCaracteres resultado;

    // solo se trabaja si las posiciones son validas
    if (posFin >= posIni && posIni >= 0 && posIni < utilizados
            && posFin < utilizados) {
        // se agregan los caracteres de interes
        for (int i = posIni; i <= posFin; i++) {
            resultado.aniade(v[i]);
        }
    }

    // se devuelve el resultado
    return resultado;
}

/**
 * Determina si una palabra cumple la condicion de palisimetrica
 * @return resultado de la comprobacion
 */
bool SecuenciaCaracteres::esPalisimetrica() {
    bool resultado = true;

    // se generan dos secuencias: una para cada mitad
    int mitad = utilizados / 2;

    // la primera secuencia va desde 0 hasta mitad
    SecuenciaCaracteres primera;
    for (int i = 0; i < mitad; i++) {
        primera.aniade(v[i]);
    }

    // se determina si la cadena tiene numero para o
    // impar de caracteres para determinar el comienzo
    // de la segunda cadena
    if (utilizados %2 != 0) {
        mitad = mitad + 1;
    }

    SecuenciaCaracteres segunda;
    for (int i = mitad; i < utilizados; i++) {
        segunda.aniade(v[i]);
    }

    // se genera una secuencia con los caracteres que
    // aparecen en la secuencia, sin repetidos
    SecuenciaCaracteres sinRepetidos;
    for (int i = 0; i < utilizados; i++) {
        if (sinRepetidos.primeraOcurrencia(v[i]) == -1) {
            sinRepetidos.aniade(v[i]);
        }
    }

    // Se recorren ahora los caracteres de la secuencia
    // sin repetidos y se cuentan las apariciones en cada
    // una de las secuencias (para ello se usa un metodo
```

```
    // auxiliar, que consideramos privado)
    for (int i = 0; i < sinRepetidos.utilizados && resultado == true; i++) {
        int ocurrenciasPrimera = primera.contarOcurrencias(sinRepetidos.v[i]);
        int ocurrenciasSegunda = segunda.contarOcurrencias(sinRepetidos.v[i]);

        // se comprueba la igualdad
        if (ocurrenciasPrimera != ocurrenciasSegunda) {
            resultado = false;
        }
    }

    // se deveulve el valor de resultado
    return resultado;
}

//Funciones genÃ©ricas

/**
 * muestra una secuencia de caracteres en la salidad estandar
 * @param s secuencia a mostrar
 */
void pintaSecuencia(SecuenciaCaracteres s){
    for (int i=0;i<s.obtenerUtilizados();i++)
        cout << s.elemento(i);

}
```

secuencia.cpp

```
#ifndef VECTORSECUENCIAS_H
#define VECTORSECUENCIAS_H

#include "secuencia.h"

class VectorSecuencias{
private:
     /**
      * Constante para el TAMANIO maximo de las secuencias
      */
     static const int  TAMA=20;
     SecuenciaCaracteres vector[TAMA];
     int utilizados;
public:
     /* Constructor por defecto
      * crea vector vacio
      */
     VectorSecuencias();

     /**
      * Anade una nueva secuencia al vector
      * @param s secuencia a insertar
      */
     void aniade(SecuenciaCaracteres s);

     /**
      * Numero mÃ¡ximo de secuencias que se pueden almacenar en el vector
      * @return capacidad
      */
     int capacidad();

     /**
      * Devuelva la secuencia en la posicion dada por indice
      * @param indice posicion a consultar
      * @return secuencia en dicha posicion
      * @pre 0 \<= indice \< obtenerUtilizados()
      */
     SecuenciaCaracteres secuencia(int indice);

     /**
      * Numero de secuencias insertadas en el vector
      * @return  numero de secuencias en el vector
      */
     int obtenerUtilizados();

};

//Funciones genericas

/**
 * muestra un vector de secuencias en la salidad estandar
 * @param v vector a mostrar
 */
void pintaVector(VectorSecuencias v);

/**
 * Cuenta el numero de secuencias palisimetricas en el vector
 * @param v el vector
 * @return numero de secs palisimetricas en v
 */
int cuantasPalisimetricas(VectorSecuencias v);

#endif /* VECTORSECUENCIAS_H */
```

vectorSecuencias.h

```
#include <iostream>
```

```
#include "vectorSecuencias.h"

using namespace std;

/* Constructor por defecto
 * crea vector vacio
 */
VectorSecuencias::VectorSecuencias(){
    utilizados=0;
}

/**
 * Anade una nueva secuencia al vector
 * @param s secuencia a insertar
 */
void VectorSecuencias::aniade(SecuenciaCaracteres s){
    if(utilizados < TAMA){
        vector[utilizados++]=s;
    }
}

/**
 * Numero mÃ¡ximo de secuencias que se pueden almacenar en el vector
 * @return capacidad
 */
int VectorSecuencias::capacidad(){
    return TAMA;
}

/**
 * Devuelva la secuencia en la posicion dada por indice
 * @param indice posicion a consultar
 * @return secuencia en dicha posicion
 * @pre 0 \<= indice \< obtenerUtilizados()
 */
SecuenciaCaracteres VectorSecuencias::secuencia(int indice){
    return vector[indice];
}

/**
 * Numero de secuencias insertadas en el vector
 * @return  numero de secuencias en el vector
*/
int VectorSecuencias::obtenerUtilizados(){
    return utilizados;
}

//Funciones genericas

/**
 * muestra un vector de secuencias en la salidad estandar
 * @param v vector a mostrar
 */
void pintaVector(VectorSecuencias v){
    for (int i=0;i<v.obtenerUtilizados();i++){
        pintaSecuencia(v.secuencia(i));
        cout << endl;
    }
}

/**
 * Cuenta el numero de secuencias palisimetricas en el vector
 * @param v el vector
 * @return numero de secs palisimetricas en v
 */
int cuantasPalisimetricas(VectorSecuencias v){
    int contador=0;
    for (int i=0; i<v.obtenerUtilizados(); i++){
        if (v.secuencia(i).esPalisimetrica())
            contador++;
    }
    return contador;
}
```

vectorSecuencias.cpp

```
#include <cstring>
#include <cstdlib>
#include <iostream>
#include "secuencia.h"
#include "vectorSecuencias.h"

using namespace std;

// Main

int main() {
    // se lee del flujo de entrada
    VectorSecuencias v;
    VectorSecuencias vpal, vnopal;
    bool continuar = true;
    while(continuar){
        string cadena;
        cin >> cadena;
        if(cadena == ""){
            continuar = false;
        }
        else{
```

```
            // se crea la secuencia
            SecuenciaCaracteres s;
            for(int i=0; i < cadena.length(); i++){
                s.aniade(cadena[i]);
            }
            v.aniade(s);

        }

    }
    cout << "Secuencias Iniciales"<< endl;
    pintaVector(v);
    cout << endl;

    for (int i=0; i<v.obtenerUtilizados();i++){
        // se hace la comprobacion
        bool cond = v.secuencia(i).esPalisimetrica();
        if (cond) {
            vpal.aniade(v.secuencia(i));
        } else {
            vnopal.aniade(v.secuencia(i));
        }
    }
    int np = cuantasPalisimetricas(v);
    cout << "Hay "<<  np << " y " << v.obtenerUtilizados()-np << " No Palisimetricas"<<endl;

    cout << endl << "Secs Palisimetricas"<<endl;
    pintaVector(vpal);
    cout << endl;
    cout << "Secs No Palisimetricas"<< endl;
    pintaVector(vnopal);
    cout<< endl;

    return 0;
}
```

prueba.cpp