UNIVERSIDAD DE GRANADA

ETSIIT Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

DECSAI

Metodología de la Programación

Curso 2023/2024

Guion de prácticas

Kmer0

class Kmer

Marzo de 2024

Índice

1. Las prácticas del curso, una visión general

Todas las prácticas que se van a desarrollar en este curso tienen como objeto principal trabajar con secuencias de ADN (*DNA*) y ARN (*RNA*) más o menos largas, procedentes de una variedad de genomas de distintas especies. Así pues, vamos a desarrollar un conjunto de aplicaciones propias del dominio de la bioinformática, cuyo componente básico son los nucleóticos, también conocidos como bases nitrogenadas: adenina (A), citosina (C), guanina (G), timina (T) y uracilo (U). La timina es específica del ADN y el uracilo específico del ARN. Un ejemplo de una secuencia de ADN es la que se muestra a continuación:

Ejemplo 1 *Ejemplo de una secuencia de ADN*

```
ACAAACGCCTCCCATTATAAGTTTATACTTCAcctcccaccactataacaacc
cagaatccatgagggcattatcaggagtgagtggaagagtaagtttgccaatg
tgaAATGTGC
```

Un concepto clave en las prácticas de la asignatura es el de k-mero [1]: secuencia corta de nucleótidos consecutivos de longitud k (siendo k un número natural) que aparece dentro de una secuencia genómica más larga. Un ejemplo de k-mero de longitud 3 (3-mero) sería “CCT” que podemos ver que puede encontrarse en varios lugares de la secuencia de ADN del anterior ejemplo. A través del estudio de las distribuciones de los k-meros en una serie de genomas de especie conocida, en la práctica final de la asignatura se buscará predecir o clasificar el genoma de un individuo de especie desconocida. Este proceso se conoce como aprendizaje supervisado (Abrir →). Una aproximación al problema del aprendizaje supervisado se da a continuación.

En primer lugar, dado el genoma perteneciente a una determinada especie (por ejemplo Drosophila melanogaster –mosca de la fruta–, Homo sapiens –humano–, Pan troglodyte –chimpacé–, rattus norvegicus –rata– o SARS-CoV-2 –virus del COVID-19–) procederemos a **aprender** un modelo para tal especie. Esto es, se aprende un modelo para cada especie que se va a considerar: Drosophila melanogaster, Homo sapiens, Pan troglodite, SARS-CoV-2, etc.

En segundo lugar, procederemos con la **clasificación**. A la llegada del genoma de un individuo desconocido, trataremos de hallar el modelo al que más se ajusta ese nuevo genoma para emitir nuestra predicción de a qué especie pertenece. La especie a la que pertenece un genoma puede averiguarse comparando las frecuencias de los k-meros (*bímeros*), ternas (*3-meros*, etc) del individuo desconocido con las de los k-meros de las especies conocidas. Así, por ejemplo una alta frecuencia del 7-mero “AAACATA” puede ser indicativo de la especie *Escherichia-coli* mientras que una alta frecuencia del 7-mero “GAGTATA”puede ser indicativo de la especie *Drosophila melanogaster*.

[1] Del griego meros (parte), que se podría traducir como *una parte o porción de longitud* k.

2. Arquitectura de las prácticas

La práctica final del `Kmer` se ha diseñado por etapas con productos operativos desde la primera etapa. Las primeras etapas contienen clases más sencillas, sobre las cuales se asientan otras más complejas y en el progreso se van incorporando nuevas funcionalidades a clases anteriores. La Figura 1 muestra el diseño de la arquitectura que se va a emplear y que se va a ir desarrollando progresivamente en esta y las siguientes sesiones de prácticas. Indicar aquí, que las estructuras de datos que se van a emplear van a sufrir varias modificaciones y van a evolucionar conforme se van adquiriendo nuevos conocimientos (los impartidos en teoría que se van a aplicar) o bien conforme llegan nuevas especificaciones (nuevos problemas para resolver). Esta situación no es excepcional sino que, la modificación y evolución de clases es un proceso habitual en cualquier desarrollo de software.

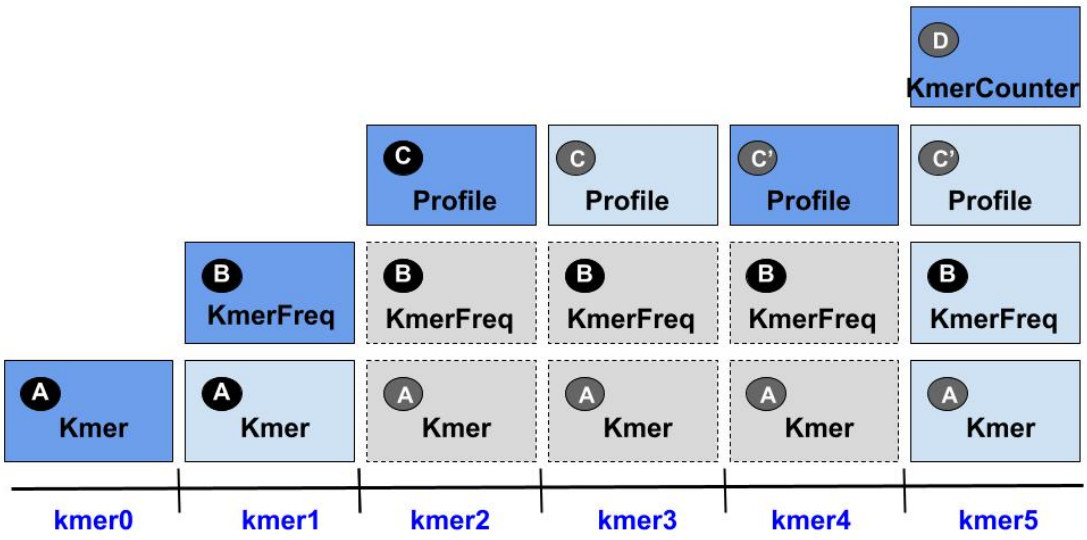

Figura 1: Arquitectura de las prácticas de MP 2024. Los cambios esenciales en las clases (cambio en estructura interna de la clase) se muestran en azul intenso, mientras que los que solo incorporan nuevas funcionalidades se muestran en azul tenue. En gris se muestran las clases que no sufren cambios durante el desarrollo de las prácticas.

A Kmer.cpp
: Implementa la clase Kmer, una clase que se define inicialmente con un string de longitud k que servirá para almacenar un k-mero (de longitud k), para la que se van a desarrollar los primeros métodos. En etapas posteriores, se revisita la clase, incorporando algunos métodos adicionales.

B KmerFreq.cpp
: Implementa la clase KmerFreq, una composición formada por un ob-

jeto de la clase anterior y un entero para el registro de la frecuencia de un k-mero. En sucesivas etapas, se revisita la clase, incorporando nuevos métodos con funcionalidades adicionales.

C Profile.cpp
: Implementa la clase Profile, una estructura para almacenar las frecuencias de un conjunto de k-meros. Será usado para definir un modelo para cada una de las especies. Tiene el registro ordenado por frecuencias de los k-meros presentes en un genoma o un conjunto de genomas. En una primera aproximación se usará un vector estático de objetos de la clase KmerFreq.

C' Profile.cpp
: Manteniendo la interfaz previa, se cambia la estructura de la clase para alojar el vector de KmerFreq, de forma más eficiente, en memoria dinámica.

D KmerCounter.cpp
: Implementa la clase KmerCounter, la estructura que va a permitir alojar una matriz bidimensional en memoria dinámica; nos será de utilidad a la hora del recuento de los k-meros durante el aprendizaje del modelo de una especie.

Este trabajo progresivo, con la incorporación de nuevos métodos, se ha planificado en hitos sucesivos con entregas en Prado. Cada práctica requerirá además de la implementación de alguna(s) funcion(es) externa(s) y de una función main() propia para cubrir sus objetivos específicos.

3. Objetivos

El desarrollo de la práctica Kmer0 persigue los siguientes objetivos:

- repasar conceptos sobre clases y **calificadores** de métodos,
- repasar conceptos básicos de funciones,
- practicar el paso de objetos por referencia,
- practicar la devolución por referencia,
- distinguir entre la devolución por valor y la devolución por referencia,
- reforzar la comprensión de los conceptos de compilación separada.

4. Práctica a entregar

En esta práctica vamos a comenzar con el primer procesamiento del genoma necesario para la obtención de un modelo de la especie a la que pertenece. Se trata de la obtención de los k-meros (donde k puede ser

un valor cualquiera) de una secuencia genómica que queremos analizar y algún procesamiento adicional.

Así, el programa a desarrollar en esta práctica tiene como objetivo obtener los k-meros y sus complementarios de una secuencia de nucleótidos leída de la entrada estándar.

Ejemplo 2 *Dado el genoma siguiente:*

```
ACTGGGTCGTAAC
```

El conjunto de 7*-meros que se obtiene es:*

```
1 ACTGGGT
    CTGGGTC
3     TGGGTCG
        GGGTCGT
5         GGTCGTA
            GTCGTAA
7             TCGTAAC
```

Para la extracción de los k-meros vamos a considerar **únicamente** la lista de nucleótidos: {A, C, G, T} (los que pueden formar parte de una secuencia de ADN). Para que esta lista sea fácilmente modificable, tal conjunto de caracteres aparece como una constante local (`const string VALID_NUCLEOTIDES = "ACGT"`) de la función `main()` en el fichero `main.cpp`. Para la obtención del k-mero complementario a uno dado, hay que tener en cuenta que cada nucleótido tiene su complementario correspondiente. El complementario de A es T (y viceversa) y el de C es G (y viceversa). De nuevo, para que sea fácilmente modificable, se ha definido dentro de la función `main()`, la constante local `const string COMPLEMENTARY_NUCLEOTIDES = "TGCA"` que contiene el nucleótido complementario de cada uno de los nucleótidos incluidos en la constante `VALID_NUCLEOTIDES`.

Nuestro programa debe guardar los k-meros de la secuencia genómica de entrada en un array. A continuación, debe convertir todos los caracteres de cada k-mero a mayúscula [2] y los caracteres no válidos (los que sean distintos a {A, C, G, T}) se sustituirán por el carácter definido con la constante `Kmer::MISSING_NUCLEOTIDE` (carácter _). Estas dos conversiones de un k-mero constituyen lo que denominaremos *normalización*. Seguidamente el programa guardará en otro array los k-meros complementarios convertidos a minúscula. Finalmente el programa mostrará ambos arrays en la salida estándar.

4.1. Módulos del proyecto

Para la elaboración de la práctica dispone de una serie de ficheros (`Kmer.h, main.cpp`) con código C++, donde se encuentran las especificaciones de lo que tiene que implementar.

Analice con cuidado las cabeceras de los métodos y los comentarios de cada método, pues en ellos están detalladas sus especificaciones. En las declaraciones de prototipos, *el número de argumentos* y *los tipos* han

[2] Mediante la función *toupper()* : (Abrir →). También existe su homóloga *tolower()*.

sido establecidos y **no se han de cambiar**. Sin embargo, **debe revisar** la forma en que se hace el paso de argumento para cada parámetro (por valor, por referencia o por referencia constante) y el calificador **const** o (no const) para argumentos y métodos con el propósito de que sean los más adecuados para su cometido. También debe tener en cuenta las **excepciones** que debe lanzar cada método cuando así se indica en la especificación (ver sección 4.3).

En esta práctica debe tener en cuenta las especificaciones indicadas en los siguientes módulos:

- El fichero **Kmer.h** (ver detalles en sección 8.1) contiene la declaración de la clase `Kmer` y la declaración de algunas funciones externas (`IsValidNucleotide()`, `ToLower()` y `ToUpper`). La clase `Kmer` contiene únicamente el dato miembro `string _text` que se usará para guardar un k-mero de cualquier longitud.

```
class Kmer {
...

private:
    /**
     * A string with a list of characters representing the
    nucleotides in
     * this Kmer.
     */
    std::string _text;
}; // end class Kmer
```

  En esta primera práctica, el fichero `Kmer.h` ya contiene correctamente definidos los prototipos de cada método y función, por lo que no necesita modificarlos. En próximas prácticas sí que tendrá que revisar los nuevos métodos y funciones que vayan apareciendo en este u otros ficheros. Añada al proyecto el fichero **Kmer.cpp** e implemente en él los métodos de la clase `Kmer` y las funciones externas anteriores.

- El módulo **main.cpp** (ver detalles en sección 8.2) tiene por objetivo leer de la entrada estándar un entero k y una cadena de caracteres `genome`, donde k es la longitud de los k-meros que se van a montar a partir de la cadena `genome`, y entonces debe obtener los k-meros y sus complementarios de `genome`.

  La función `main()` declara los arrays `kmers` y `complementaryKmers`, dos arrays de k-meros de la misma longitud, los cuales podrán contener k-meros repetidos.

  - El array `kmers` servirá para guardar los k-meros extraídos de `genome`, donde se insertarán los k-meros desde la posición 0 en adelante, en el mismo orden en que se hallan en la cadena (en la forma descrita en el ejemplo 2).
  - El array `complementaryKmers` servirá para guardar los k-meros complementarios de `kmers`, de manera que cada nucleótido de `kmers` en la posición `i` tiene su complementario en la misma posición `i` de `complementaryKmers`.

Nota: El código del main que elabore en esta ocasión, no es necesario que esté correctamente modularizado por falta de los conocimientos necesarios sobre paso de vectores a funciones.

4.2. Ejemplos de ejecución

Ejemplo 3 *Un ejemplo de llamada al programa desde un terminal podría ser:*

```
linux> dist/Debug/GNU-Linux/kmer0 < data/shortDNA5_missing.k0in
```

El contenido del fichero `data/shortDNA5_missing.k0in` *es el siguiente:*

```
1 5 GCGCCCCGuutGGACAGCCATGCGCTAACCCTGGCT0
```

Como puede verse, los datos que necesita leer el programa de la entrada estándar son:

- *Un número entero para definir el valor de* k*.*
- *Una cadena de caracteres que contiene una secuencia genómica.*

La salida obtenida al ejecutar el programa sería la siguiente:

```
1  33
2  GCGCC<—>cgcgg
3  CGCCC<—>gcggg
4  GCCCC<—>cgggg
5  CCCCG<—>ggggc
6  CCCG_<—>gggc_
7  CCG__<—>ggc__
8  CG__T<—>gc__a
9  G__TG<—>c__ac
10 __TGG<—>__acc
11 _TGGA<—>_acct
12 TGGAC<—>acctg
13 GGACA<—>cctgt
14 GACAG<—>ctgtc
15 ACAGC<—>tgtcg
16 CAGCC<—>gtcgg
17 AGCCA<—>tcggt
18 GCCAT<—>cggta
19 CCATG<—>ggtac
20 CATGC<—>gtacg
21 ATGCG<—>tacgc
22 TGCGC<—>acgcg
23 GCGCT<—>cgcga
24 CGCTA<—>gcgat
25 GCTAA<—>cgatt
26 CTAAC<—>gattg
27 TAACC<—>attgg
28 AACCC<—>ttggg
29 ACCCT<—>tggga
30 CCCTG<—>gggac
31 CCTGG<—>ggacc
32 CTGGC<—>gaccg
33 TGGCT<—>accga
34 GGCT_<—>ccga_
```

La salida anterior puede verse también en el fichero `data/shortDNA5_missing.k0out`*.*

Ejemplo 4 *Dado el contenido del fichero* `easyDNA5.k0in`

```
1 5 GCgCcCCGtG
```

se muestra a continuación la salida del programa `kmer0`

```
1 6
2 GCGCC<—>cgcgg
3 CGCCC<—>gcggg
4 GCCCC<—>cgggg
5 CCCCG<—>ggggc
6 CCCGT<—>gggca
7 CCGTG<—>ggcac
```

Ejemplo 5 *Dado el contenido del fichero* `shortDNA20_missing.k0in`

```
20 GCGCCCCGuutGGACAGCCa
```

la salida es:

```
1
GCGCCCCG_TGGACAGCCA<—>cgcggggc__acctgtcggt
```

Ejemplo 6 *Un caso particular es cuando la longitud del k-mero solicitada es mayor que la longitud de la cadena suministrada. Un ejemplo lo encontramos en el fichero* `shortDNA25_no.k0in`*.*

```
25 GCGCCCCGACtGGACAGCC
```

la salida es:

```
0
```

4.3. Las especificaciones sobre excepciones

Las excepciones son una forma de actuar (en tiempo de ejecución) ante circunstancias excepcionales, como cuando no se cumplen los precondiciones de un módulo, lo que podría producir un mal funcionamiento de nuestro programa que, bien pasase inadvertido, o que hiciese que este terminase de forma anómala. Por ejemplo, situaciones que podrían provocar estos problemas serían una división por cero, acceso a una posición fuera del rango dentro de un vector, intento de escribir en un fichero para el que no tenemos permiso de escritura, etc. En esos casos podrían usarse excepciones para que nuestro programa tenga controladas este tipo de situaciones de error.

Cuando se lanza una excepción, si esta no es capturada, esta es enviada a la función llamante. En la función llamante, si la excepción no es capturada, se envía de nuevo a su función llamante y así sucesivamente hasta que la excepción llega a `main()`. Si la excepción tampoco es capturada en `main()`, el programa termina y se muestra su descripción por la salida estándar.

A continuación, vamos a analizar las especificaciones sobre excepciones que nos podemos encontrar en esta asignatura en las cabeceras de métodos o funciones.

En primer lugar, veamos el constructor con un string como parámetro, `Kmer(const string &text)`, extracto de `Kmer.h`:

```
/**
  * @brief It builds a Kmer object with the characters in the
    string @p text
  * representing the list of nucleotides of the new Kmer.
  *
  * @throw std::invalid_argument Throws an std::invalid_argument
    exception
  * if the given text is empty
```

```
  * @param text a string with the characters representing the
    nucleotides for
  * the kmer. It should be a string with at least one character.
  */
     Kmer(const std::string& text);
```

Como puede verse, se indica que el método lanza la excepción `std::invalid_argument` si el texto suministrado es vacío.

En la especificación del método de consulta `at()` de la clase Kmer, se especifica una excepción diferente: `std::out_of_range`, como puede verse a continuación:

```
/**
  * @brief Gets a const reference to the character (nucleotide)
    at the given
  * position
  * @param index the position to consider
  * @throw std::out_of_range Throws a std::out_of_range
    exception if the
  * index is not in the range from 0 to k-1.
  * @return A const reference to the character at the given
    position
  */
     const char& at(int index) const;
```

Indicar que por simplicidad, en esta asignatura, nuestro propósito será únicamente detectar estas situaciones de error y especificar dónde o cuál es la situación de error que produjo la excepción, mediante el uso de una cadena de texto que describa tal error y que cada uno podrá definir como quiera. Esta descripción acompañará al nombre de la excepción como puede verse en los ejemplos siguientes. El texto definido se mostrará automáticamente en la salida estándar, en caso de que se lance la excepción durante la ejecución del programa, ya que según acabamos de decir, en caso de que se lance una excepción el programa terminará y se mostrará por la salida estándar la descripción de la excepción.

Veamos cómo vamos a proceder en la implementación al encontrarnos con una excepción en la especificación. En primer lugar, se identifica la excepción especificada. En los ejemplos anteriores tenemos especificada la excepción `std::invalid_argument`, o bien `std::out_of_range`. En segundo lugar, se lanza la excepción indicada con la palabra reservada `throw` **si se cumple la condición especificada** en la cabecera. En nuestro caso debe lanzarse la excepción si la longitud de `text` es 0, o cuando el valor del índice no se encuentra en el rango $[0..k[$, respectivamente. A continuación se muestran las definiciones de los dos métodos, completa el primero y parcialmente incompleta el segundo.

```
Kmer::Kmer(const std::string& text) {
   if(text.size()==0){     // text is empty
       throw std::invalid_argument( // composed string
           string("Kmer(const std::string& text): ") +
                 "text is an empty string ");
```

```
    }
    _text = text;
}
```

```
const char& Kmer::at(int index) const{
    if(index<0 || index>=getK()){//Precondition violation
        throw std::out_of_range( //composed string
            string("const char& Kmer::at(int index) const: ") +
                "invalid position " + to_string(index));
    }
    else{
        return ... // the correct value
    }
}
```

Indicar de nuevo que, si se capturase una excepción se podría arreglar la situación de error, pero por simplicidad, en la asignatura Metología de la Programación, tan solo vamos a usar la parte `throw` sin la componente `catch`; o sea, nunca vamos a capturar excepciones en esta asignatura. Si quiere saber más: (Abrir →).

5. Configuración de las prácticas

Para la elaboración de la práctica dispone de una serie de ficheros que se encuentran en `Kmer0_nb.zip` de prado.ugr.es o el repositorio de `github` según la distribución del profesor. Una vez descomprimido el anterior fichero, va a encontrar entre otros, los ficheros `Kmer.h`, `main.cpp`, `shortDNA25_missing.k0in`, `shortDNA25_missing.k0out`, `documentation.doxy`, etc. Para montar la primera práctica, tendrá que crear un proyecto nuevo según se especifica en la sección 6. Pero antes, veamos cómo definir el espacio de trabajo.

El espacio de trabajo

ProyectosNetBeans : Carpeta raíz en la que se van a crear todos los proyectos de la asignatura. Tendremos un directorio por cada Kmer, que se genera en el momento de crear un proyecto NetBeans. La estructura que deberíamos de tener es la que se muestra a continuación:

```
NetbeansProjects
├── Debugger*
├── HelloWorld
├── Kmer0
├── Kmer1
├── *
├── MPGeometry*
├── MPTools
├── MyVector
└── Scripts
```

```
└── ValgrindShowcase
```

Genomes: Un conjunto de ficheros que contienen trozos de genomas. En muchos ejemplos solo se da un trozo, ya que la longitud de un genoma completo podría ir desde los 580,000 pares de bases de ADN de una bacteria como Mycoplasma genitalium a los 3 mil millones de pares de bases de ADN del ser humano repartidos por sus 23 pares de cromosomas. Dispondremos de numerosos ficheros, ya sea en su formato textual (*.dna) o bien en su formato de profile, lista ordenada de k-meros y frecuencias (*.prf).

Scripts: Una serie de scripts Bash de apoyo a las funciones de NetBeans.

```
Kmer
├── build
│   ├── Debug
│   └── Release
├── data
├── dist
│   ├── Debug
│   └── Release
├── doc
│   └── documentation.doxy
├── include
│   └── *h
├── Makefile
├── nbproject
├── scripts
├── src
│   └── *cpp
├── tests
│   └── *test
└── zip
```

Figura 2: Estructura interna de cada proyecto Kmer

Para tener bien ordenado el contenido de cada proyecto Kmer, se va a organizar en las siguientes carpetas:

build Es una carpeta temporal que contiene código precompilado y pendiente de enlazar.

data Lo usaremos en las primeras prácticas para contener ficheros con ejemplos de entrada al programa y la correspondiente salida esperada al ejecutarlo.

dist Contiene los binarios ejecutables. Vamos a generar dos versiones por cada programa:

- **Release** Es la versión más eficiente y que menos espacio ocupa. Es la que debería entregarse al cliente.
- **Debug** Es la versión sobre la que se depuran errores y se corrigen defectos. Ocupa más espacio que la anterior ya que contiene información extra para poder usar el depurador. Puede llegar a ser el doble de grande.

doc Aquí se almacena la documentación del proyecto, tanto del código, como la generada por los tests.

include Contiene los ficheros de cabeceras **.h**.

nbproject Es la carpeta que utiliza NetBeans para almacenar la configuración del proyecto.

scripts En esta carpeta colocaremos los scripts de apoyo a Netbeans que se han desarrollado en esta asignatura y se puede ampliar con otros más. Se utilizarán en prácticas posteriores. Por ahora se hace todo manual.

src Contiene los ficheros fuente del proyecto **.cpp**.

tests Contiene los tests de integridad de cada proyecto. Son ficheros con extensión **.test**.

zip Carpeta para dejar las copias de seguridad del proyecto.

6. Configurar el proyecto en NetBeans

Para crear un proyecto Kmer desde cero.

1. Crear el proyecto en NetBeans como `C++ Application` con nombre Kmer0 para esta práctica. Crear las carpetas: `include, src, data, doc, script, zip`, dentro del directorio Kmer0. Esto se puede hacer desde un terminal usando el comando `mkdir <nombreDir>` o bien desde el entorno de NetBeans, a través de la pestaña **File**. Es necesario añadir las carpetas una a una.

   Carpetas como `build, dist, nbproyect` son creadas y gestionadas automáticamente por NetBeans, por lo que no deben tocarse.

2. Colocar cada uno de los ficheros en su sitio, el/los ficheros `*.h` en `include`, los `*.cpp` en `src`, etc. Estas acciones y las realizadas en el paso anterior se han llevado a cabo a través del S.O.. Compruébalo con la instrucción unix: `ls`. La estructura debe ser como la indicada en la figura 3.

3. Propiedades de proyecto. Compilador

   a) Poner el estándar a C++14.

   b) Añadir en `Include Directories` la carpeta con los ficheros de cabecera del propio proyecto (**./include**).

   c) En `Additional Options` es interesante añadir las opciones `-Wall` (activar todos los mensajes de advertencia del compilador) y `-pedantic` (da mensajes de advertencia si nuestro programa no sigue las normas ISO C e ISO C++). Por ejemplo, el uso de `-pedantic` hará que nos aparezca una advertencia si declaramos un array de un tamaño que no sea una constante, algo prohibido en esta asignatura:

```
int main(){
    int variable=10;
    double array[variable];
}
```

4. Desde la vista lógica del proyecto.

   a) En `Header Files`, `Add existing item`: fichero `Kmer.h`.

   b) En `Source Files`, `Add existing item`: ficheros `Kmer.cpp` y `main.cpp`.

5. Sobre el nombre del proyecto, botón derecho `Set As Main Project` o desde el menú principal `Run` – `Set Main Project` y seleccionar este proyecto.

6. Con esto ya se podría ejecutar el proyecto normalmente. Obviamente, no hace nada porque todavía está vacío. A partir de aquí empezaría el desarrollo del proyecto para el que cada programador seguirá un orden determinado y, probablemente, diferente a todos los demás.

   Nota: No deje para el final la comprobación de todos los métodos y no compruebe solo los métodos utilizados en `main()`. **Todos los métodos han de ser testados** para comprobar que cada uno funciona como se espera, aunque no sea utilizado en el `main()` solicitado para la práctica [3]. Para ello deberá realizar una batería de pruebas.

La práctica deberá ser entregada en Prado, durante el periodo habilitado en la propia actividad de entrega, y consistirá en un fichero ZIP del proyecto. Se puede montar el zip desde NetBeans, a través de **File** → **Export project** → **To zip**. El nombre Kmer0.zip, sin más aditivos, es el mismo nombre para todos los estudiantes. Compruebe que lo entregado es compilable y operativo.

7. Uso de scripts

En primer lugar, hemos de completar nuestro espacio de trabajo con la carpeta **Scripts**, también disponible en el repositorio de la asignatura:

```
NetbeansProjects
├── Debugger*
├── HelloWorld
├── Kmer0
├── Kmer1
├── *
├── MPGeometry*
├── MPTools
├── MyVector
├── Scripts
└── ValgrindShowcase
```

[3] Considere por ejemplo todos los constructores que se han declarado en la clase *Kmer* y, seguramente no se emplearán todos en la función `main()` de kmer0 pero sí en sucesivas prácticas... Cuando se define una clase, es cuando esta se depura, incluso antes.

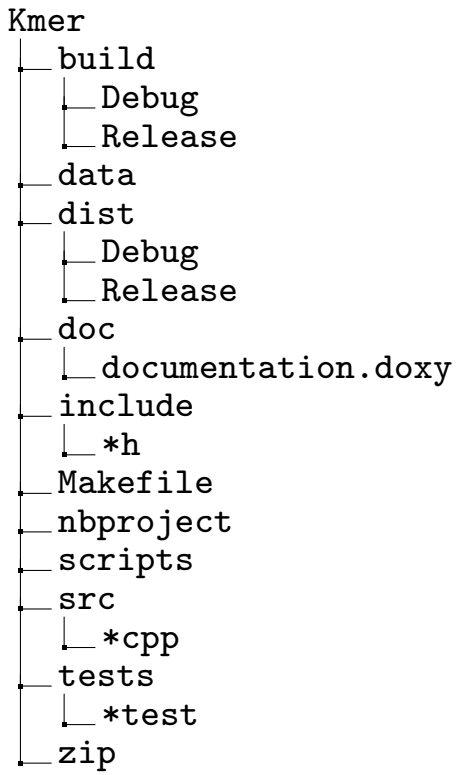

Figura 3: Estructura interna de cada proyecto Kmer

La carpeta **Scripts** contiene una serie de scripts bash de apoyo a las funciones de NetBeans. Es única para todos los proyectos NetBeans, y como vemos, se encuentra al mismo nivel que las carpetas de los proyectos Kmer.

Sin embargo, existe una carpeta `scripts` (con s minúscula) por cada uno de los proyectos Kmer (ver figura 3). No confundir con la carpeta `Scripts` anterior. La carpeta `scripts` es la que contiene los scripts que nosotros vamos a ejecutar manualmente [4].

Como se ha indicado, en la carpeta `scripts` se encuentran una serie de utilidades bash, ficheros con extensión `*.sh` tales como:

- `runUpdate.sh` que actualiza los scripts de la carpeta `scripts` con los scripts actualizados que hayamos descargado del repositorio en la carpeta `Scripts`.
- `runDocumentation.sh` que va a facilitar el proceso de generación de la documentación con doxygen y su visualización en un navegador.
- `runZipProject.sh` que facilita el proceso de elaborar un zip, eliminando previamente todos los archivos binarios que no tiene sentido exportar como `*.bin, *.o,` etc.
- `runTests.sh` que ejecuta los tests de integración disponibles en la carpeta `tests` del proyecto proporcionado en el repositorio.

[4] Estos hacen uso de las utilidades en Scripts; por ello, para un correcto funcionamiento, es importante que `Scripts` esté correctamente ubicada en su lugar.

Para ejecutar un script desde NetBeans, basta con situarse con el cursor del ratón sobre el script, pulsar botón derecho del ratón y elegir `Run`. Por ejemplo, en el caso del script `runZipProject.sh`, tras ejecutar el script, se obtendrá un fichero con extensión `.zip` en el directorio `zip`.

7.1. El script runTests.sh

Con cada proyecto Kmer se proporcionan varios tests de integración, cada uno de ellos en un fichero con extensión `.test`, y que deben estar colocados en la carpeta `tests`. Un test de integración es una prueba de nuestro programa, en la que se definen los datos que nuestro programa lee y la salida que debería producir con esos datos.

Por ejemplo, el fichero `easyDNA3.test` del proyecto Kmer0, tiene el siguiente contenido:

```
%%%CALL < data/easyDNA3.k0in
%%%DESCRIPTION Easy test with a few nucleotides. K=3
%%%OUTPUT
8
GCG<-->cgc
CGC<-->gcg
GCC<-->cgg
CCC<-->ggg
CCC<-->ggg
CCG<-->ggc
CGT<-->gca
GTG<-->cac
```

En la asignatura no es imprescindible entender el formato de estos ficheros de tests, pero lo comentamos aquí brevemente. En el test anterior, se indica que si el programa `Kmer0` se llama redirigiendo la entrada estándar desde el fichero `data/easyDNA3.k0in`, entonces la salida que debería producir el programa es la que se muestra tras la línea `%%%OUTPUT`. El contenido del fichero `data/easyDNA3.k0in` es:

```
3 GCgCcCCGtG
```

O sea, el test anterior hace que nuestro programa lea los datos 3 (un número entero) y `GCgCcCCGtG` (un string).

Para ejecutar los tests de integración, usaremos el script `runTests.sh`. Hay varias formas de hacerlo:

1. Desde NetBeans podemos ejecutar todos los tests de integridad de la siguiente forma: seleccionamos la vista física del proyecto (*Files*), situamos el cursor del ratón sobre el script, pulsamos botón derecho del ratón y elegimos `Run`.

2. Desde un terminal de Netbeans o de nuestro sistema operativo Linux podemos ejecutar todos los tests de integridad situándonos dentro de la carpeta de nuestro proyecto y ejecutando:

```
Linux> script/runTests.sh
```

3. Desde un terminal de Netbeans o de nuestro sistema operativo Linux podemos ejecutar uno solo de los tests situándonos dentro de la carpeta de nuestro proyecto y ejecutando:

```
Linux> script/runTests.sh test/nombreTestAEjecutar.sh
```

Al hacerlo con cualquiera de las opciones anteriores, nos aparecerá en pantalla la lista de tests ejecutados y podremos ver si nuestro programa los ha pasado o no. Por ejemplo, en Kmer0, si ejecutamos desde un terminal el script de la siguiente forma:

```
Linux> script/runTests.sh test/easyDNA3.test
```

obtendremos por la salida (en caso de que nuestro programa pase el test) lo siguiente:

```
Loading ansiterminal...

VALIDATION TOOL v2.0
Checking tests folder           OK
Setting everything up           OK
Test #1  Easy test with a few nucleotides. K=3

Testing tests/easyDNA3.test             Generating fresh
 ↪  binaries
make[2]: 'dist/Debug/GNU-Linux/kmer0teacher' está actualizado.
[  OK  ] Test #1 [tests/easyDNA3.test] (
 ↪  dist/Debug/GNU-Linux/kmer0teacher  < data/easyDNA3.k0in)
End of tests
```

8. Código para la práctica

8.1. Kmer.h

```
/*
 * Metodología de la Programación: Kmer0
 * Curso 2023/2024
 */

/**
 * @file Kmer.h
 * @author Silvia Acid Carrillo <acid@decsai.ugr.es>
 * @author Andrés Cano Utrera <acu@decsai.ugr.es>
 * @author Luis Castillo Vidal <L.Castillo@decsai.ugr.es>
 * @author Javier Martínez Baena <jbaena@ugr.es>
 *
 * Created on 24 October 2023, 14:00
 */

#ifndef KMER_H
#define KMER_H

#include <iostream>
#include <string>

/**
 * @class Kmer
 * @brief It represents a list of k consecutive nucleotides of a DNA or RNA
 * sequence. Each nucleotide is represented with a character like
```

```
 * 'A', 'C', 'G', 'T', 'U'.
 */
class Kmer {
public:
    /**
     * A static const character representing an unknown nucleotide. It is used
     * when we do not known which nucleotide we have in a given position of
     * a Kmer
     */
    static const char MISSING_NUCLEOTIDE = '_';

    /**
     * @brief It builds a Kmer object using a string with @p k characters
     * (nucleotides). Each character will be set to the value
     * @p MISSING_NUCLEOTIDE.
     * @throw std::invalid_argument Throws an std::invalid_argument exception
     * if @p k is less or equal than zero
     * @param k the number of nucleotides in this Kmer. It should be an integer
     * greater than zero. Input parameter
     */
    Kmer(int k=1);

    /**
     * @brief It builds a Kmer object with the characters in the string @p text
     * representing the list of nucleotides of the new Kmer.
     * @throw std::invalid_argument Throws an std::invalid_argument exception
     * if the given text is empty
     * @param text a string with the characters representing the nucleotides for
     * the kmer. It should be a string with at least one character. Input parameter
     */
    Kmer(const std::string& text);

    /**
     * @brief Returns the number of nucleotides in this Kmer.
     * Query method
     * @return the number of nucleotides in this Kmer
     */
    int getK() const;

    /**
     * @brief Returns the number of nucleotides in this Kmer.
     * Query method
     * @return the number of nucleotides in this Kmer
     */
    int size() const;

    /**
     * @brief Returns a string with a list of characters, each one representing
     * a nucleotide of this Kmer.
     * Query method
     * @return The text of this Kmer as a string object
     */
    std::string toString() const;

    /**
     * @brief Gets a const reference to the character (nucleotide) at the given
     * position.
     * Query method
     * @param index the position to consider. Input parameter
     * @throw std::out_of_range Throws an std::out_of_range exception if the
     * index is not in the range from 0 to k-1 (both included).
     * @return A const reference to the character at the given position
     */
    const char& at(int index) const;

    /**
     * @brief Gets a reference to the character (nucleotide) at the given
     * position.
     * Modifier method
     * @param index the position to consider. Input parameter
     * @throw std::out_of_range Throws an std::out_of_range exception if the
     * index is not in the range from 0 to k-1 (both included).
     * @return A reference to the character at the given position
     */
    char& at(int index);

    /**
     * @brief Normalizes this Kmer. That is, it converts all the characters to
     * uppercase. Then, invalid characters are replaced by the
     * MISSING_NUCLEOTIDE value.
     * Modifier method
     * @param validNucleotides a string with the list of characters
     * (nucleotides) that should be considered as valid. Input parameter
     */
    void normalize(const std::string& validNucleotides);

    /**
     * @brief Returns the complementary of this Kmer. For example, given the Kmer
     * "TAGAC", the complementary is "ATCTG" (assuming that we use.
     * @p nucleotides="ATGC" and @p complementaryNucleotides="TACG").
     * If a nucleotide in this object is not in @p nucleotides, then that
     * nucleotide remains the same in the returned kmer.
     * Query method
     * @param nucleotides A string with the list of possible nucleotides. Input parameter
     * @param complementaryNucleotides A string with the list of complementary
     * nucleotides. Input parameter
     * @throw std::invalid_argument Throws an std::invalid_argument exception if
     * the sizes of @p nucleotides and @p complementaryNucleotides are not
     * the same
     * @return The complementary of this Kmer
```

```
     */
    Kmer complementary(const std::string& nucleotides,
        const std::string& complementaryNucleotides) const;

private:
    /**
     * A string with a list of characters representing the nucleotides in
     * this Kmer.
     */
    std::string _text;
}; // end class Kmer

/**
 * @brief Checks if the given nucleotide is contained in @p validNucleotides.
 * That is, if the given character can be considered as part of a genetic
 * sequence.
 * @param nucleotide The nucleotide (a character) to check. Input parameter
 * @param validNucleotides The set of characters that we consider as possible
 * characters in a genetic sequence. Input parameter
 * @return true if the given character is contained in @p validNucleotides;
 * false otherwise
 */
bool IsValidNucleotide(char nucleotide, const std::string& validNucleotides);

/**
 * @brief Converts to lowercase the characters (nucleotides) of the given Kmer
 * @param kmer A Kmer object. Output parameter
 */
void ToLower(Kmer& kmer);

/**
 * @brief Converts to uppercase the characters (nucleotides) of the given Kmer
 * @param kmer A Kmer object. Output parameter
 */
void ToUpper(Kmer& kmer);

#endif /* KMER_H */
```

8.2. main.cpp

```
/*
 * Metodología de la Programación: Kmer0
 * Curso 2023/2024
 */

/*
 * File:   main.cpp
 * @author Silvia Acid Carrillo <acid@decsai.ugr.es>
 * @author Andrés Cano Utrera <acu@decsai.ugr.es>
 * @author Luis Castillo Vidal <L.Castillo@decsai.ugr.es>
 * @author Javier Martínez Baena <jbaena@ugr.es>
 *
 * Created on 24 October 2023, 13:58
 */

#include <iostream>
#include <string>

#include "Kmer.h"

using namespace std;

/**
 * This program first reads from the standard input an integer k (length of Kmer)
 * and a string with a genetic sequence. Then, it obtains from the genetic
 * sequence, the list of kmers (of length k) and saves them in the array kmers.
 * Then, the kmers are normalized. After that, the complementary kmers,
 * converted to lowercase, are saved in the array complementaryKmers. Finally
 * the kmers in the arrays kmers and complementaryKmers are shown in the
 * standard output.
 * See the next example:
 *
 * Running example:
 * > kmer0 < data/easyDNA5_missing.k0in
6
GCGCC<—>cgcgg
CGCCC<—>gcggg
GCCC_<—>cggg_
CCC_G<—>ggg_c
CC_G_<—>gg_c_
C_G_G<—>g_c_c
 */
int main(int argc, char* argv[]) {
    // This string contains the list of nucleotides that are considered as
    // valid within a genetic sequence. The rest of characters are considered as
    // unknown nucleotides
    const string VALID_NUCLEOTIDES = "ACGT";

    // This string contains the list of complementary nucleotides for each
    // nucleotide in validNucleotides
    const string COMPLEMENTARY_NUCLEOTIDES = "TGCA";

    // This is a constant with the dimension of the array kmers
    const int DIM_ARRAY_KMERS = 100;
```

```
    // This is the array where the kmers of the input genetic sequence will be
    // saved
    Kmer kmers[DIM_ARRAY_KMERS];

    // This is the array where the complementary kmers will be
    // saved
    Kmer complementaryKmers[DIM_ARRAY_KMERS];

    // Read K (integer) and a string with the input nucleotides list

    // Obtain the kmers: find the kmers in the input string and put them in an array of Kmers

    // Normalize each Kmer in the array

    // Obtain the complementary kmers and turn them into lowercase

    // Show the list of kmers and complementary kmers as in the example

    return 0;
}
```