Universidad de La Laguna

Grado en Ingeniería Informática

**Computabilidad y Algoritmia**

# PRÁCTICA 2

## Cadenas y lenguajes

Factor de ponderación: 8

Semana del 22 al 26 de septiembre 2025

## 1. Objetivos

El objetivo de la práctica es trabajar **conceptos básicos sobre símbolos, alfabetos, cadenas y lenguajes**, a través del diseño de las clases correspondientes en C++. Además de repasar estos conceptos teóricos, se propone que el alumnado utilice este ejercicio para poner en práctica aspectos generales relacionados con el desarrollo de programas en C++.

Algunos de estos principios que enfatizaremos en esta práctica serán los siguientes:

- Programación orientada a objetos: es fundamental identificar y definir clases y objetos que permitan modelar adecuadamente el escenario de trabajo que se plantea.
- Diseño modular: el programa debiera escribirse de modo que las diferentes funcionalidades que se precisen sean encapsuladas en métodos concretos cuya extensión textual se mantenga acotada.
- Pautas de estilo: es imprescindible ceñirse al formato para la escritura de programas en C++ que se propone en esta asignatura. Los principales criterios de estilo que seguiremos en la asignatura ya se describieron brevemente en la primera práctica.
- **Sets de la STL**: para la definición de conjuntos de elementos se deberá utilizar el contenedor `set` [8, 9] de la STL [10].

- **Sobrecarga de operadores**: para comparación entre cadenas, así como para las realización de algunas de las operaciones con cadenas, se deberán aplicar los principios de la sobrecarga de operadores en C++ [11, 12].
- **Operadores de entrada/salida**: para leer o escribir símbolos, alfabetos, cadenas o lenguajes, se deberán definir los correspondientes operadores de entrada y salida estándar [13].
- Compilacion de programas utilizando `make` [3, 4].

## 2. Conceptos fundamentales

Antes de comenzar a describir el programa a desarrollar es importante tener claramente identificados los conceptos principales introducidos en el primer tema de la asignatura [1]:

- Un *alfabeto* es un conjunto no vacío y finito de símbolos. En general, un símbolo no tiene por qué corresponder intrínsecamente a un carácter ASCII [2]. Sin embargo, y con el objetivo de simplificar, en este contexto supondremos que los símbolos estarán asociados a caracteres ASCII.
- Una *cadena* es una secuencia finita de símbolos de un determinado alfabeto. Por lo tanto, a nivel teórico una cadena se define o se asocia a un determinado alfabeto. Además, hemos definido una serie de propiedades así como un conjunto de operaciones que pueden realizarse sobre cadenas. En esta práctica nos centraremos en las siguientes propiedades y operaciones:
  - Toda cadena tiene asociada una determinada longitud. Una cadena que no tiene símbolos tendrá longitud 0 y se le denominará cadena vacía. Para especificar la cadena vacía vamos a utilizar el símbolo &. Por lo tanto, para evitar confusiones, el símbolo & no podrá utilizarse como elemento de los alfabetos que se utilizarán en este programa.
  - A partir de una cadena se podrá calcular su cadena inversa.
  - A partir de una cadena se podrá obtener el conjunto de prefijos de la cadena, el conjunto de sufijos, o bien, el conjunto de todas las posibles subcadenas.
- Un *lenguaje* es un *conjunto de cadenas*. Un lenguaje puede ser vacío, puede contener un número finito de cadenas, o bien, puede ser infinito. En el caso del lenguaje vacío, lo denotaremos como {}. En el caso de un lenguaje finito, relacionaremos sus cadenas entre llaves y separadas por comas. Por ejemplo:

$$L = \{w_1, w_2, w_3, w_4, w_5\}$$

# 3. Ejercicio práctico

Teniendo en cuenta las propiedades de las cadenas y el comportamiento de las mismas, se propone desarrollar un programa en C++ que dado un fichero de entrada con la especificación de cadenas junto con sus respectivos alfabetos, realice un conjunto de comprobaciones básicas sobre ellas. Para ser coherentes con las definiciones anteriores, cabe destacar que en aquellas operaciones sobre cadenas en las que el resultado obtenido sea un conjunto de cadenas, deberíamos manejar dicho resultado como un lenguaje.

El programa recibirá por línea de comandos el nombre del fichero de entrada, el nombre del fichero de salida y un código de operación:

```
./p02_strings filein.txt fileout.txt opcode
```

El comportamiento del programa al ejecutarse en línea de comandos debiera ser similar al de los comandos de Unix. Así por ejemplo, si se ejecuta el programa sin parámetros, se debería obtener información sobre el uso correcto del programa:

```
$ ./p02_strings
Modo de empleo: ./p02_strings filein.txt fileout.txt opcode
Pruebe './p02_strings --help' para más información.
```

La opción `--help` en línea de comandos ha de producir que se imprima en pantalla un breve texto explicativo del funcionamiento del programa. Una información que puede ser de especial ayuda para los usuarios del programa sería precisamente el formato de los ficheros así como el significado de los códigos de operación.

El fichero de entrada tendrá en cada línea la especificación de una cadena de entrada. Cada cadena de entrada vendrá seguida por una secuencia de símbolos que representa el alfabeto sobre el que se ha definido la cadena correspondiente. En el ejemplo siguiente tenemos la cadena *abbab* definida sobre el alfabeto $\{a,\ b\}$, la cadena 6793836 definida sobre el alfabeto $\{1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9\}$, y la cadena *hola* definida sobre el alfabeto $\{a,\ h,\ l,\ o\}$:

```
abbab ab
6793836 123456789
hola ahlo
```

En función del código de operación, se aplicará una determinada operación a cada una de las cadenas de entrada, escribiendo el resultado en el fichero de salida. A continuación, se describe el comportamiento para cada uno de los códigos de operación, enumerados desde el 1 hasta el 5.

1. *Alfabeto*: escribir en el fichero de salida el alfabeto asociado a cada una de las cadenas de entrada. El fichero de salida asociado a la entrada del ejemplo anterior sería el siguiente:

```
{a, b}
{1, 2, 3, 4, 5, 6, 7, 8, 9}
{a, h, l, o}
```

2. *Longitud*: escribir en el fichero de salida la longitud de cada cadena de entrada. Es decir, si se escogiera el código 1, asociado al cálculo de la longitud de la cadena, el fichero de salida asociado a la entrada del ejemplo anterior sería el siguiente:

```
5
7
4
```

3. *Inversa*: escribir en el fichero de salida la inversa de cada cadena de entrada. En este caso, la salida sería la siguiente:

```
babba
6383976
aloh
```

4. *Prefijos*: escribir en el fichero de salida el conjunto de cadenas que son prefijos de la cadena de entrada correspondiente. Tal y como hemos mencionado, dichos conjuntos de cadenas conformarán un lenguaje y, por tanto, se representarán y se visualizarán como tales. Para la entrada de ejemplo, la salida sería similar a la siguiente:

```
{&, a, ab, abb, abba, abbab}
{&, 6, 67, 679, 6793, 67938, 679383, 6793836}
{&, h, ho, hol, hola}
```

5. *Sufijos*: escribir en el fichero de salida el conjunto de cadenas que son sufijos de cada cadena de entrada correspondiente. Al igual que en el caso anterior, dichos conjuntos de cadenas conformarán un lenguaje y, por tanto, se representarán y se visualizarán como tales. Para la entrada de ejemplo, la salida sería similar a la siguiente:

```
{&, b, ab, bab, bbab, abbab}
{&, 6, 36, 836, 3836, 93836, 793836, 6793836}
{&, a, la, ola, hola}
```

   En relación a los resultados anteriores, téngase en cuenta que las cadenas de los conjuntos se están mostrando de forma ordenada en relación a la longitud de las mismas. El resultado sería diferente si se hubiera optar por hacer una ordenación basada en la comparación alfanumérica de las cadenas.

# 4. Estilo y formato del código

Esta relación no pretende ser exhaustiva. Contiene una serie de requisitos y/o recomendaciones cuyo cumplimiento se impulsará a la hora de evaluar las prácticas de la asignatura.

- Estudien la finalidad de los especificadores de visibilidad (public, private, protected) de atributos y métodos. Una explicación breve y simple es ésta [5] aunque es fácil hallar muchas otras. Se debe restringir los atributos y métodos públicos a aquellos que son estrictamente necesarios para que un programa cliente utilice la clase en cuestión.
- El principio de responsabilidad única (*Single responsibility principle, SRP*) [6] es uno de los conocidos como principios "SOLID" y es posiblemente uno de los más fáciles de comprender. Dejando a un lado el resto de principios SOLID, que pueden ser más complejos, merece la pena que lean algo sobre el SRP. Ese principio es una de las razones por las que en sus programas no debiera haber una única clase "*que se encarga de todo*". En esta referencia [7] tienen un sencillo ejemplo (el de la clase `Person` que puede servir para ilustrar el SRP, pero no es difícil encontrar múltiples ejemplos del SRP. Tengan en cuenta que es un principio de orientación a objetos, y por lo tanto independiente del lenguaje de programación: pueden encontrar ejemplos de código en C++, Java, Pythos u otros lenguajes.
- Un programa simple en C++ debiera incluir al menos 3 ficheros:
  - `X.h` para la definición de la clase X
  - `X.cc` para la implementación de la clase X
  - `ClienteX.cc` para el programa "cliente" que utiliza la clase X
- Para evitar la doble inclusión de un mismo fichero (habitualmente un fichero de cabecera, *header*) en un código fuente hay básicamente dos posibilidades: las *include guards* [17] y el uso de la directiva *#pragma once* [18]. Es conveniente que conozcan ambas alternativas, el funcionamiento de cada una y que tengan en cuenta que la directiva `#pragma once`, aunque soportada por la mayor parte de los compiladores de C++, no es estándar. Para más información sobre este asunto pueden leer esta discusión de StackOverflow [19].
- Ejemplo de comentario de cabecera (inicial) para (todos) los ficheros de un proyecto (práctica) de la asignatura:

```
// Universidad de La Laguna
// Escuela Superior de Ingeniería y Tecnología
// Grado en Ingeniería Informática
// Asignatura: Computabilidad y Algoritmia
// Curso: 2º
// Práctica 2: Cadenas y lenguajes
// Autor: Nombre y Apellidos
// Correo: aluXXXXX@ull.edu.es
// Fecha: 16/09/2025
// Archivo cya-P02-strings.cc: programa cliente.
//         Contiene la función main del proyecto que usa las clases X e Y
//         para ... (indicar brevemente el objetivo)
// Referencias:
//          Enlaces de interés
/
// Historial de revisiones
//          16/09/2025 - Creación (primera versión) del código
```

- Además de la información anterior sobre el autor, la fecha y la asignatura, la cabecera también debería contener al menos una breve descripción sobre lo que hace el código incluyendo los objetivos del proyecto en general, las estructuras de datos utilizadas, así como un listado de modificaciones (bug fixes) que se han ido introduciendo en el código. El fichero de cabecera sería básicamente el mismo para todos los ficheros (`*.cc`, `*.h`) del proyecto, con la salvedad de que varía la descripción del contenido del fichero y su finalidad.

- El utilizar una estructura de directorios con nombres estándar (`bin`, `lib`, `src`, `config`, etc.) para los proyectos de desarrollo de software es una buena práctica cuando se trata de proyectos que involucran un elevado número de ficheros. No obstante, para la mayoría de prácticas de esta asignatura, que no contienen más de una decena de ficheros, colocar todos los ficheros de cada práctica en un mismo directorio (con nombre significativo *CyA/practicas/p02_strings*, por ejemplo) es posiblemente más eficiente a la hora de localizar y gestionar los ficheros de cada una de las prácticas (proyectos).

- En el directorio de cada práctica, como se ha indicado anteriormente, debería haber un fichero `Makefile` de modo que el comportamiento del programa `make` fuera tal que al ejecutar:

```
1 make clean
```

  el programa dejara en el directorio en que se ejecuta solamente los ficheros conteniendo código fuente (ficheros `*.h` y `*.cc`) y el propio `Makefile`. En estos breves tutoriales [3, 4] se explica de forma incremental cómo construir un fichero Makefile para trabajar con la compilación de proyectos simples en C++.

- A continuación enumeraremos algunas cuestiones relativas al formato del código:

  1. A ambos lados de un operador binario han de escribir un espacio:

     ```
     a + b
     ```

     En lugar de:

     ```
     a+b
     ```

  2. SIEMPRE después de una coma, ha de ir un espacio.
  3. Se debe indentar el código usando espacios y NO tabuladores. Cada nivel de indentación ha de hacerse con 2 espacios.
  4. TODO identificador (de clase, de método, de variable, ...) ha de ser significativo. No se pueden usar identificadores de un solo caracter salvo para casos concretos (variable auxiliar para un bucle o similar).
  5. TODO fichero de código de un proyecto ha de contener un prólogo con comentarios de cabecera donde se indique al menos: Autor, datos de contacto, Fecha, Asignatura, Práctica, Finalidad del código. Veáse el ejemplo de cabecera proporcionado en este documento.
  6. Asimismo todos los métodos y clases han de tener al menos un mínimo comentario que documente la finalidad del código.
  7. No comentar lo obvio. No se trata de comentar por comentar, sino de aclarar al lector la finalidad del código que se escribe [15].
  8. Como regla de carácter general, el código fuente de los programas no debiera contener de forma explícita constantes. En lugar de escribir

     ```
     double balance[10];
     ```

     Es preferible:

     ```
     const int SIZE_BALANCE = 10;
     ...
     double balance[SIZE_BALANCE];
     ```

     Para el caso de las constantes numéricas, las únicas excepciones a esta regla son los valores 0 y 1. Tengamos en cuenta que las constantes no son solamente numéricas: puede haber constantes literales, de carácter o de otros tipos. Así en lugar de escribir:

     ```
     myString.find(‘ ’);
     ```

Es preferible:

```
const char SPACE = ' ';
...
myString.find(SPACE);
```

9. Cuando se programa un código encadenando múltiples sentencias if-else, ello suele ser síntoma de una mala práctica y, en efecto, debería buscarse alguna alternativa. En este sentido les recomendamos que revisen las diferentes sugerencias que se realizan al hilo de esta discusión [16].

10. Como regla general, se espera que todo el código fuente siga la guía de estilo de Google para C++ [14]. Dentro de esa guía, y para comenzar, prestaremos particular atención a los siguientes aspectos:

    - Formateo del código (apartado *Formatting* en [14])
    - Comentarios de código de diverso tipo (apartado *Comments* en [14])
    - Nominación de identificadores, clases, ficheros, etc. (apartado *Naming* en [14])

# 5. Criterios de evaluación

Se señalan a continuación los aspectos más relevantes (la lista no es exhaustiva) que se tendrán en cuenta a la hora de evaluar esta práctica:

- Se valorará que el alumnado haya realizado, con anterioridad a la sesión de prácticas, y de forma efectiva, todas las tareas propuestas en este guión. Esto implicará que el programa compile y ejecute correctamente.
- También se valorará que, con anterioridad a la sesión de prácticas, el alumnado haya revisado los documentos que se enlazan desde este guión.
- Paradigma de programación orientada a objetos: se valorará que el alumnado haya identificado clases y objetos que permitan modelar adecuadamente el escenario de trabajo que se plantea.
- Paradigma de modularidad: se valorará que el programa se haya escrito de modo que las diferentes funcionalidades que se precisen hayan sido encapsuladas en métodos concretos cuya extensión textual se mantuviera acotada.
- Sobrecarga de operadores: se valorará que el alumnado haya aplicado los principios básicos para la sobrecarga de operadores en C++.
- Uso de contenedores de la librería estándar: se valorará el uso de contenedores estándar ofrecidos en la STL, especialmente el uso de `sets`.
- Se valorará que el código desarrollado siga el formato propuesto en esta asignatura

para la escritura de programas en C++.

- Capacidad del programador(a) de introducir cambios en el programa desarrollado.

Si el alumnado tiene dudas respecto a cualquiera de estos aspectos, debiera acudir al foro de discusiones de la asignatura para plantearlas allí. Se espera que, a través de ese foro, el alumnado intercambie experiencias y conocimientos, ayudándose mutuamente a resolver dichas dudas. También el profesorado de la asignatura intervendrá en las discusiones que pudieran suscitarse, si fuera necesario.

## Referencias


[1] Transparencias del Tema 1 de la asignatura: Alfabetos, cadenas y lenguajes, https://campusvirtual.ull.es/2526/ingenieriaytecnologia/mod/resource/view.php?id=11848

[2] Código ASCII: https://es.wikipedia.org/wiki/ASCII

[3] Makefile Tutorial: http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor

[4] C++ Makefile Tutorial: https://makefiletutorial.com

[5] Difference between private, public and protected inheritance: https://stackoverflow.com/questions/860339/difference-between-private-public-and-protected-inheritance

[6] Single-Responsability Principle (SRP): https://en.wikipedia.org/wiki/Single_responsibility_principle

[7] Example of the SRP: https://stackoverflow.com/questions/10620022/what-is-an-example-of-the-single-responsibility-principle

[8] STL sets, http://www.cplusplus.com/reference/set/set

[9] STL sets tutorial, https://www.w3schools.com/cpp/cpp_sets.asp

[10] Using the C++ Standard Template Libraries, https://link.springer.com/book/10.1007%2F978-1-4842-0004-9

[11] Sobrecarga de operadores, https://es.cppreference.com/w/cpp/language/operators

[12] C++ Operator Overloading Example, https://www.programiz.com/cpp-programming/operator-overloading

[13] Input/Output Operators Overloading in C++, https://www.tutorialspoint.com/cplusplus/input_output_operators_overloading.htm

[14] Google C++ Style Guide, https://google.github.io/styleguide/cppguide.html

[15] Commenting code: https://www.cs.utah.edu/~germain/PPS/Topics/commenting.html

[16] If-else-if chains or multiple-if: https://stackoverflow.com/questions/25474906/which-is-better-if-else-if-chain-or-multiple-if

[17] Include guard: https://en.wikipedia.org/wiki/Include_guard

[18] Pragma once: https://en.wikipedia.org/wiki/Pragma_once

[19] Pragma once vs. Include guards: https://stackoverflow.com/questions/1143936/pragma-once-vs-include-guards