Universidad de La Laguna

Grado en Ingeniería Informática

Computabilidad y Algoritmia

PRÁCTICA 3

Operaciones con cadenas y lenguajes

Factor de ponderación: 7

1. Objetivos

El objetivo de la práctica es trabajar algunas operaciones básicas con cadenas así como introducir el concepto de lenguaje. También se pretende poner en práctica algunas cuestiones relacionadas con la programación en C++. En particular, se trabajará la *sobrecarga de operadores* y se utilizará la clase `set` de la *STL* para representar conjuntos. Al igual que en la práctica anterior, recuerde que además de repasar las operaciones con cadenas 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. Algunos de los principales criterios de estilo se describirán brevemente en la sección 3 de esta práctica. Estos criterios son los mismos que los ya introducidos en la práctica anterior.
- **Sets de la STL**: para la definición de conjuntos de elementos [8, 9] se deberá utilizar el contenedor `set` [7] de la STL [10] (librería estándar de C++).
- **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].
- Compilación de programas utilizando `make` [2, 3].

2. Ejercicio práctico

El ejercicio práctico a desarrollar se plantea como una ampliación del programa desarrollado en la práctica anterior. Teniendo en cuenta las nociones básicas introducidas en la práctica anterior sobre *símbolos*, *alfabetos* y *cadenas*, en este caso nos centraremos en las siguientes propiedades y operaciones:

- Dos cadenas podrán ser comparadas, determinando si son iguales, si una de ellas es subcadena de la otra o viceversa, o son simplemente cadenas diferentes:
  - Si $w_1 == w_2$, entonces las dos cadenas tienen exactamente los mismos símbolos en el mismo orden.
  - Si $w_1 != w_2$, entonces las cadenas no son iguales (difieren en al menos un símbolo dentro de la secuencia).
  - Si $w_1 > w_2$, entonces w_2 es una subcadena de w_1.
  - Si $w_1 < w_2$, entonces w_1 es una subcadena de w_1.
- Dadas dos cadenas w_1 y w_2, se podrá calcular la cadena resultante de la concatenación $w_1 \cdot w_2$
- Dada una cadena w, se podrá calcular la cadena resultante de la operación potencia w^n

Aprovecharemos esta práctica para también **introducir el concepto de lenguaje** [1]:

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

Teniendo esto en cuenta, para aquellas operaciones en las que el resultado sea un conjunto de cadenas, deberemos manejar dicho resultado como un lenguaje.

Al igual que en la práctica anterior, el programa recibirá por línea de comandos el nombre del fichero de entrada, el nombre del fichero de salida y un código de operación:

```
./p03_string_operations filein.txt fileout.txt opcode
```

El fichero de entrada tendrá en cada línea la especificación de los símbolos que definen el alfabeto (separados por espacios) seguidos de la cadena. Además tenga en cuenta que podría darse el caso de que no se especifique el alfabeto. En estos caso tendría que obtenerse el alfabeto de forma automática a partir de los símbolos que intervienen en la propia cadena. En el ejemplo siguiente podemos ver una cadena definida sobre el alfabeto $\{a, b\}$, una cadena definida sobre el alfabeto $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, y una cadena definida sobre el alfabeto $\{h, o, l, a\}$:

```
a b abbab
0 1 2 3 4 5 6 7 8 9 6793836
hola
```

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. Por ejemplo, si se escogiera el código asociado al cálculo de la longitud de la cadena, el fichero de salida asociado a la entrada anterior sería el siguiente:

```
5
7
4
```

El formato del fichero de salida podrá variar en función del código de operación especificado. Además de las 5 operaciones definidas en la práctica anterior, en esta práctica incluiremos las tres operaciones enumeradas como 6, 7 y 8, respectivamente:

1. *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
   ```

2. *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
   ```

3. *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 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}
```

4. *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 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}
   ```

5. *Subcadenas*: escribir en el fichero de salida el conjunto de subcadenas para cada cadena de entrada. Dicho conjunto de cadenas también deberá tratarse como un lenguaje.

6. *Comparación de cadenas*: escribir en el fichero de salida la relación existente entre cada una de las cadenas de entrada y una cadena extra que se solicitará al usuario por teclado. Para nuestro ejemplo de entrada, y suponiendo que la cadena auxiliar que se introduzca por teclado sea *ba*, entonces la salida debería ser la siguiente:

   ```
   abbab > ba
   6793836 != ba
   hola != ba
   ```

7. *Concatenación de cadenas*: escribir en el fichero de salida la concatenación de cada una de las cadenas de entrada con una cadena extra que se solicitará al usuario por teclado. Para nuestro ejemplo de entrada, y suponiendo que la cadena auxiliar que se introduzca por teclado sea xxx, entonces la salida debería ser la siguiente:

   ```
   abbabxxx
   6793836xxx
   holaxxx
   ```

8. *Potencia*: escribir en el fichero de salida la potencia n-ésima de cada una de las cadenas de entrada. El valor de n se solicitará al usuario por teclado. Para nuestro ejemplo de entrada, y suponiendo que $n = 3$, entonces el fichero de salida debería contener lo siguiente:

   ```
   abbababbababbab
   679383667938366793836
   holaholahola
   ```

3. 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 [4] 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*) [5] 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 [6] 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 3: Operaciones con cadenas y lenguajes
```

```
// Autor: Nombre y Apellidos
// Correo: aluXXXXX@ull.edu.es
// Fecha: 13/10/2021
//
// Archivo cya-P03-StringsOp.cc: programa cliente.
//         Contiene la función main del proyecto que usa las clases
//         X e Y para ... (indicar brevemente el objetivo)
//         Descripción otras funcionalidades
//
// Referencias:
//          Enlaces de interés
/
// Historial de revisiones
//          13/10/2021 - 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 signficativo *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 [2, 3] 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])

4. 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 planeta.
- 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/1920/mod/resource/view.php?id=20983

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

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

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

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

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

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

[8] STL sets example, https://thispointer.com/stdset-tutorial-part-1-set-usage-details

[9] STL sets example, https://tinyurl.com/cyasetexample

[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