Universidad de La Laguna

Grado en Ingeniería Informática

Computabilidad y Algoritmia

PRÁCTICA 2

Operaciones con lenguajes

Factor de ponderación: 7

1. Objetivos

El objetivo de la práctica es incorporar el concepto de lenguaje así como trabajar con algunas operaciones básicas con lenguajes. 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 lenguajes, 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 continuación 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 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:

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

A pesar de que sabemos que los lenguajes pueden tener infinitas cadenas, en este ejercicio práctico manejaremos únicamente lenguajes con un número finito de cadenas.

El objetivo de esta práctica será desarrollar un programa cliente que, dado unos lenguajes de entrada (especificados en ficheros de texto) y un código de operación, ejecute la operación correspondiente sobre los lenguajes de entrada. El resultado de las operaciones llevadas a cabo se almacenará en un fichero de salida. Al igual que en la práctica anterior, el programa recibirá por línea de comandos todos los parámetros necesarios para poder realizar una ejecución. Teniendo esto en cuenta la ejecucion del programa en línea de comandos debería ser similar a lo siguiente:

```
./p02_languages filein1.txt filein2.txt fileout.txt opcode
```

En los ficheros de entrada encontraremos la definición de los lenguajes con los que vamos a realizar la operación correspondiente. En el fichero de salida escribiremos los lenguajes resultantes de la operación que hayamos indicado realizar, mientras que el código de operación nos indicará precisamente el tipo de operación que habrá de llevarse a cabo.

El formato de los ficheros de entrada serán similar al siguiente:

```
{ a b c } { a abc cab bbcc }
{ a } { a aa aaa }
{ a b } { }
```

```
{ 0 1 } { & }
{ 0 1 } { 0 1 00 01 10 11 }
```

En cada línea encontraremos la especificación de un lenguaje: primero el alfabeto y luego el conjunto de cadenas que conforman el lenguaje. Esto es, cada línea del fichero de entrada contiene, entre llaves, los símbolos que conforman un alfabeto y, a continuación, y también entre llaves, las cadenas que conforman un lenguaje. Para simplificar, en lugar de utilizar las comas como separadores de los elementos de un conjunto, utilizaremos simplemente espacios. También (con el objetivo de simplificar) pueden suponer que inmeditemaente tras (o justo antes de) las llaves siempre habrá un espacio.

En el primero de los lenguajes de entrada del ejemplo anterior se define un alfabeto $\Sigma = \{a, b, c\}$ y un lenguaje $L = \{a, abc, cab, bbcc\}$. Teniendo en cuenta el formato de los ficheros de entrada, las llaves y los espacios no deberían usarse como símbolos de los alfabetos sobre los que se forman los lenguajes. Además, cabe destacar que para especificar la cadena vacía vamos a utilizar el símbolo &. Por lo tanto, para evitar confusiones, el símbolo & tampoco podrá utilizarse como elemento de los alfabetos que se utilizarán en esta práctica. A modo de ejemplo, la cuarta línea del fichero de entrada anterior definiría al lenguaje que contiene únicamente a la cadena vacía. Por último, para especificar el lenguaje vacío se propone utilizar las llaves sin definir ninguna cadena dentro, tal y como se ha hecho en la tercera línea del ejemplo anterior.

El formato de los ficheros de salida será equivalente al de los ficheros de entrada. Estos es, en primer lugar se indicará el alfabeto sobre el que se define el lenguaje y a continuación se especificará el propio lenguaje. Se utilizarán las llaves y los espacios tal y como se ha descrito para el caso de los ficheros de entrada.

En lo que respecta a la especificación de los lenguajes, cabe destacar que en este ejercicio todos los lenguajes de entrada tendrán un número finito de cadenas.

Una vez clarificado como será la especificación de lenguajes en los ficheros de entrada, procederemos a realizar la operación correspondiente en función del valor introducido como código (`opcode`) introducido en la línea de comandos. El código de operación será un número entero que identificará a las operaciones tal y como se indica a continuacion:

1. Concatenación
2. Unión
3. Intersección
4. Diferencia
5. Inversa
6. Potencia

Hay que tener en cuenta que independientemente de la operación seleccionada siempre se obtendrá un único lenguaje resultante. Es por ello, que existe un único fichero de salida donde se escribirán dichos lenguajes resultantes de la operación. Sin embargo, en el caso de los lenguajes de entrada debemos diferenciar dos casos. Para las operaciones binarias (requieren dos operandos), como son la concatenacion, la unión, la intersección y la diferencia, se necesitarán dos lenguajes de entrada. En estos casos, se tomará como primer operando un lenguaje del primer fichero de entrada y como segundo operando el correspondiente lenguaje del segundo fichero de entrada. Esto significa que para poder realizar una operación binaria el número de lenguajes especificados en el primer fichero `filein1.txt` debe coincidir con el número de lenguajes especificados en el segundo fichero `filein2.txt`. Para las operaciones unarias (requieren de un único operando) como la inversa y la potencia bastará con tener un único lenguaje de entrada, por lo tanto, los lenguajes se tomarán únicamente del primer fichero de entrada, ignorando así los lenguajes especificados en el segundo de los ficheros. Además, hay que tener en cuenta que para realizar el cálculo de la potencia se necesitará un dato extra: el propio valor de n para calcular los correspondientes resultados de L^n. Este valor puede pasarse por línea de comandos o solicitarse al usuario por pantalla durante la ejecución del programa. Cualquiera de las opciones será válida.

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 TODAS 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 2: Operaciones con lenguajes
// Autor: Nombre y Apellidos
// Correo: aluXXXXX@ull.edu.es
// Fecha: 10/10/2022
//
// Archivo cya-P02-Languages.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
//          10/10/2022 - 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://campusingenieriaytecnologia2223.ull.es/mod/resource/view.php?id=5914

[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