Taller I – Curso 2018 - 2019
Trabajo de Laboratorio

Los equipos de trabajo tienen que ser cooperativos, es decir, todos y cada uno de sus integrantes tendrán que participar para lograr una auténtica integración y cada uno se enriquecerá con la colaboración de los demás.

Planteo del problema

Se desea desarrollar un sistema de software que permita a estudiantes de bachillerato trabajar con ***polinomios con coeficientes enteros***. La aplicación permitirá definir polinomios con coeficientes enteros por parte del estudiante y ejecutar de un conjunto de funcionalidades sobre ellos, destacándose principalmente las operaciones que permiten chequear raíces enteras y evaluar un polinomio en un valor entero dado, así como otras operaciones adicionales.

Si bien la cantidad de polinomios que se manejarán en memoria será finita, el sistema deberá permitir tener en memoria tantos polinomios como se desee. Además, no existirá cota para el grado de cada polinomio. Es decir, cada polinomio podrá ser de cualquier grado.

Para interactuar con la aplicación, el usuario digitará una serie de comandos por teclado que le indicarán al programa las diferentes acciones a realizar. El programa deberá interpretar el comando ingresado por el usuario y luego realizar la acción correspondiente. Los comandos disponibles son los siguientes:

- `crear`: crea un nuevo polinomio especificando todos sus coeficientes
- `sumar`: realiza la suma de dos polinomios, creando uno nuevo como resultado
- `multiplicar`: realiza la multiplicación de dos polinomios, creando uno nuevo como resultado
- `evaluar`: realiza la evaluación para un polinomio en un valor entero dado
- `esraiz`: muestra por pantalla si un número entero dado es raíz de un polinomio
- `mostrar`: muestra por pantalla todos los polinomios existentes en memoria
- `guardar`: guarda en archivo un polinomio existente en memoria
- `recuperar`: recupera a memoria un polinomio previamente guardado en archivo
- `salir`: abandona la aplicación

Se muestran a continuación dos ejecuciones de ejemplo para familiarizarse con el uso de los comandos anteriores:

Ejecución 1

```
Ingrese comando:  crear pepe 2 0 0 -1 –8
Resultado:        pepe = +2x4 –x -8
Ingrese comando:  crear lolo 1 0 1
Resultado:        lolo = +x2 +1
Ingrese comando:  sumar maria pepe lolo
Resultado:        maria = +2x4 +x2 –x -7
Ingrese comando:  multiplicar megalolo lolo lolo
Resultado:        megalolo = +x4 +2x2 +1
Ingrese comando:  evaluar lolo -3
Resultado:        10
```

```
Ingrese comando:  esraiz lolo -1
Resultado:        no es raiz
Ingrese comando:  mostrar
Resultado:        lolo = +x2 +1
                  maria = +2x4 +x2 –x -7
                  megalolo = +x4 +2x2 +1
                  pepe = +2x4 –x -8
Ingrese comando:  guardar pepe pepe.txt
Resultado:        polinomio almacenado correctamente en pepe.txt
Ingrese comando:  salir
Resultado:        hasta la próxima
```

Ejecución 2

```
Ingrese comando:  recuperar pepe pepe.txt      (el mismo de la ejecución 1)
Resultado:        pepe = +2x4 –x -8
Ingrese comando:  crear negapepe -2 0 0 1 8
Resultado:        negapepe = -2x4 +x +8
Ingrese comando:  sumar nulo pepe negapepe
Resultado:        nulo = 0
Ingrese comando:  crear raicesdist 1 0 -1
Resultado:        raicesdist = +x2 -1
Ingrese comando:  esraiz raicesdist 1
Resultado:        es raiz
Ingrese comando:  esraiz raicesdist -1
Resultado:        es raiz
Ingrese comando:  recuperar doble raizdoble.txt   (asumimos ya existía en disco)
Resultado:        doble = +x2 -2x +1
Ingrese comando:  esraiz doble 1
Resultado:        es raiz
Ingrese comando:  mostrar
Resultado:        doble = +x2 -2x +1
                  negapepe = -2x4 +x +8
                  nulo = 0
                  pepe = +2x4 –x -8
                  raicesdist = +x2 -1
Ingrese comando:  salir
Resultado:        hasta la próxima
```

En cada paso de la ejecución, la aplicación solicitará un comando. El usuario digitará un comando y el programa emitirá el resultado de su ejecución, el cual puede ser exitoso o erróneo (en las ejecuciones de ejemplo todos los resultados mostrados fueron exitosos). En caso de ser erróneo, puede ser por diferentes motivos, como ser un error de sintaxis en el comando ingresado o bien un error de funcionamiento (por ejemplo, pretender recuperar un polinomio que no existe en disco). Se valorará positivamente un sofisticado control de sintaxis en los comandos ingresados así como la emisión de mensajes de error al usuario que sean lo más adecuados posibles.

El programa siempre dará la opción de respaldar polinomios en disco para luego ser utilizados en futuras ejecuciones. Cada vez que un polinomio vaya a ser desplegado en pantalla, sus términos se mostrarán siempre ordenados de mayor a menor ordenados según su exponente. Cada exponente debe mostrarse tan solo una vez (y no deben mostrarse aquellos términos que tienen 0 como coeficiente, excepto únicamente en el caso del polinomio nulo). Por ejemplo, para mostrar en pantalla el polinomio resultado de hacer la multiplicación `(x + 1) * (x - 1)`, se debe mostrar como resultado `+x2 -1` (y no cosas como `+x2 + 0x1 - 1`).

Los polinomios que maneje la aplicación en memoria se identificarán mediante un nombre alfanumérico. El nombre será *case-sensitive* (esto es, distingue minúsculas de mayúsculas) y no deberá contener espacios ni otros símbolos no alfanuméricos. Por ejemplo, los siguientes son nombres válidos: `pepe`, `lolo`, `HaWaIi`, `hawaii`, `poli125`, mientras que los siguientes son nombres inválidos: `_poli`, `#soypolinomio`, `nombre con espacios`.

Se presenta a continuación una descripción detallada de cada comando:

Sintaxis del comando:	`crear nombre coef1 coef2 … coefN`
Descripción:	Crea en memoria un nuevo polinomio identificado con el nombre digitado en primer lugar, con los N coeficientes luego del nombre. El primer coeficiente será el del término de mayor grado, el segundo coeficiente será del término con un grado menor al primero y así sucesivamente. El último coeficiente será el correspondiente al término independiente.
Errores a controlar:	Que se digite exactamente un nombre de polinomio luego del comando, que dicho nombre sea alfanumérico, que los coeficientes digitados después del nombre sean efectivamente números enteros, que se digite al menos un coeficiente luego del nombre, que no exista otro polinomio en memoria con el mismo nombre, que el primer coeficiente sea distinto de cero en caso de que se digiten al menos dos coeficientes. Se debe emitir un mensaje de error adecuado en cada caso.

Sintaxis del comando:	`sumar nombreNuevo nombre1 nombre2`
Descripción:	Crea en memoria un nuevo polinomio, cuyo nombre será el primer nombre digitado, el cual será la suma de los polinomios cuyos nombres son el segundo y el tercer nombre digitado.
Errores a controlar:	Que se digiten exactamente tres nombres de polinomios luego del comando, que el nombre del nuevo polinomio sea alfanumérico, que no exista otro polinomio en memoria con el mismo nombre, que el segundo y el tercer nombre correspondan efectivamente a polinomios existentes en memoria. Se debe emitir un mensaje de error adecuado en cada caso.

Sintaxis del comando:	`multiplicar nombreNuevo nombre1 nombre2`
Descripción:	Crea en memoria un nuevo polinomio, cuyo nombre será el primer nombre digitado, el cual será la multiplicación de los polinomios cuyos nombres son el segundo y el tercer nombre digitado.
Errores a controlar:	Que se digiten exactamente tres nombres de polinomios luego del comando, que el nombre del nuevo polinomio sea alfanumérico, que no exista otro polinomio en memoria con el mismo nombre, que el segundo y el tercer nombre correspondan efectivamente a polinomios existentes en memoria. Se debe emitir un mensaje de error adecuado en cada caso.

Sintaxis del comando:	`evaluar nombre numero`
Descripción:	Despliega en pantalla el valor que se obtiene como resultado de evaluar el polinomio cuyo nombre es digitado en primer lugar con el número digitado en segundo lugar. Es decir, el resultado de sustituir las x's del polinomio por el número digitado y realizar las operaciones correspondientes.
Errores a controlar:	Que se digite exactamente un nombre y un valor numérico luego de dicho nombre, que el nombre corresponda efectivamente a un polinomio existente en memoria, que el valor digitado luego de él sea efectivamente un número entero. Se debe emitir un mensaje de error adecuado en cada caso.

Sintaxis del comando:	`esraiz nombre numero`
Descripción:	Determina si el número ingresado en segundo lugar es o no una raíz del polinomio cuyo nombre es ingresado en primer lugar. Es decir, si el resultado de evaluar el polinomio con dicho valor es cero o no.
Errores a controlar:	Que se digite exactamente un nombre y un valor numérico luego de dicho nombre, que el nombre corresponda efectivamente a un polinomio existente en memoria, que el valor digitado luego de él sea efectivamente un número entero. Se debe emitir un mensaje de error adecuado en cada caso.

Sintaxis del comando:	`mostrar`
Descripción:	Muestra por pantalla todos los polinomios existentes en memoria, ordenados alfabéticamente por nombre, de menor a mayor. Cada polinomio se mostrará en una nueva línea, mostrando primero el nombre y a continuación el polinomio correspondiente, con sus términos ordenados de mayor a menor grado.
Errores a controlar:	Que exista al menos un polinomio en memoria para mostrar.

Sintaxis del comando:	`guardar nombre nombreArchivo.txt`
Descripción:	Almacena el polinomio identificado por el nombre digitado en primer lugar en el archivo cuyo nombre es digitado en segundo lugar. Si el archivo con ese nombre ya existe en el disco, preguntará al usuario si desea sobre-escribirlo.
Errores a controlar:	Que se digite exactamente un nombre de polinomio y un nombre de archivo luego de él, que el primer nombre corresponda efectivamente a un polinomio ya existente en memoria, que el segundo nombre sea alfanumérico y tenga solamente la extensión `.txt`. Se debe emitir un mensaje de error adecuado en cada caso.

Sintaxis del comando:	`recuperar nombre nombreArchivo.txt`
Descripción:	Crea en memoria un nuevo polinomio, cuyo nombre será el primer nombre digitado, el cual será el polinomio recuperado del archivo cuyo nombre es el segundo nombre digitado.
Errores a controlar:	Que se digite exactamente un nombre de polinomio y un nombre de archivo luego de él, que el nombre del nuevo polinomio sea alfanumérico, que no exista otro polinomio en memoria con el mismo nombre, que el segundo nombre sea alfanumérico y tenga solamente la extensión `.txt`, que efectivamente exista un archivo en disco con ese nombre. Se debe emitir un mensaje de error adecuado en cada caso.

Sintaxis del comando:	`salir`
Descripción:	Finaliza la ejecución de la aplicación, liberando **toda** la memoria dinámica pedida durante la ejecución de la aplicación.
Errores a controlar:	Ninguno.

Una aproximación a la solución

Como se dijo anteriormente, no existe cota para la cantidad de polinomios que se pueden almacenar en memoria. Además, varias de operaciones requieren realizar búsquedas para saber si un nombre de polinomio existe o no. También se tiene un comando para listar todos los polinomios existentes en memoria, ordenados alfabéticamente de menor a mayor. Tomando en cuenta estas consideraciones, se deberá elegir una estructura de datos adecuada que permita almacenar a los polinomios en memoria.

Por otra parte, se deberá elegir otra estructura de datos adecuada para almacenar a un polinomio, teniendo en cuenta que se debe almacenar tanto su nombre como sus términos con coeficientes distintos de cero (excepto únicamente en el caso del polinomio nulo). Cada término tiene un coeficiente entero y un exponente natural. Un polinomio almacenará una cantidad no acotada de términos, los cuales deberán almacenarse ordenados de mayor a menor grado.

También serán eventualmente necesarias otras estructuras de datos auxiliares no evidentes a simple vista. Se debe analizar cuidadosamente cada comando a efectos de detectar cualquier otra estructura de datos que resulte útil y/o necesaria.

En relación a la lectura e interpretación de los comandos, todo comando será inicialmente leído desde teclado y cargado en un string. Luego se realizará un procesamiento (habitualmente se llama ***parsing*** a dicho procesamiento) de ese string para determinar de qué comando se trata, si es sintácticamente correcto, y posteriormente ejecutar las acciones que correspondan al comando en cuestión. Cuanto más sofisticado sea el *parsing*, más positivamente será valorado el trabajo. Se sugiere particionar el proceso de *parsing* en las siguientes etapas (definiendo oportunamente los procedimientos y funciones que sean necesarios en cada caso):

- Lectura del comando en un string.
- Validación del comando ingresado, devolviendo como resultado si es o no sintácticamente erróneo, el tipo de comando en cuestión y los parámetros que correspondan.
- Procesamiento del comando luego de haber sido validado, manipulando adecuadamente las estructuras de datos que sean pertinentes.

Además del *parsing*, también se deberá definir los procedimientos y funciones necesarios para el procesamiento de todos los comandos, así como los módulos necesarios, distribuyendo adecuadamente todos los subprogramas entre ellos, respetando los criterios usuales de modularización

Primera Entrega - *Análisis y Diseño*

Fecha de entrega: **Lunes 18 de Febrero de 2019**. Se deberá entregar un documento en formato **PDF** a la dirección de correo indicada por el docente tutor. También se deberá entregar una copia impresa del documento al docente tutor en la siguiente tutoría. El documento entregado deberá contener, como mínimo, las siguientes secciones:

1. Elección de todas las estructuras de datos más apropiadas para representar el problema, justificando la elección. Escritura en C++ de todos los tipos de datos correspondientes.
2. Diagrama de módulos conteniendo los módulos identificados a partir de los tipos de datos definidos, junto con las inclusiones correspondientes.
3. Seudocódigo de la ejecución de cada comando de la aplicación. Dicho seudocódigo debe ser lo más detallado posible, a efectos de permitir luego definir, a partir de él, todos los procedimientos y funciones que sean necesarios.
4. Cabezales sintácticos de todos los procedimientos y funciones a incorporar en cada uno de los módulos, los cuales surgirán a partir del seudocódigo realizado para cada comando.
5. Cronograma de implementación y testing de los módulos indicando, para cada módulo a implementar y testear, fechas estimadas de inicio de fin.
6. Explicación de todas las decisiones de diseño que haya ido tomando el equipo al realizar el diseño de la solución.

Segunda Entrega - *Implementación*

Fecha de entrega: **Martes 12 de Marzo de 2019**. Se deberá entregar un archivo comprimido (en formato **zip**) a la dirección de correo indicada por el docente tutor, el cual debe contener el ***project*** en Code::blocks, incluyendo **todos** los archivos (**.h**) y (**.cpp**) correspondientes a los módulos implementados. Se deben **eliminar** los archivos compilados (**.o**) de la subcarpeta obj y el archivo ejecutable (**.exe**) de la subcarpeta bin.

Metodología de Trabajo

Se deberá trabajar en forma ordenada, elaborando el documento solicitado para la primera entrega y haciendo un programa de prueba por cada módulo implementado. Además de los ítems especificados anteriormente, el grupo también deberá entregar cualquier otra documentación que el tutor considere oportuna durante el desarrollo de las distintas etapas del taller.