Bitcoin Exchange

A continuación se presenta la traducción del enunciado junto con una explicación detallada del ejercicio, su propósito de aprendizaje y una guía completa sobre cómo afrontarlo en C++98, teniendo en cuenta las reglas específicas del módulo.

Traducción y descripción del ejercicio

Ejercicio: Bitcoin Exchange

- **Objetivo del programa:**

- Se debe crear un programa llamado **btc** que, a partir de dos bases de datos en formato CSV, calcule y muestre el valor de una cierta cantidad de bitcoin en una fecha dada.
    - Una base de datos (proporcionada con el enunciado) contiene el historial de precios de bitcoin a lo largo del tiempo.
    - El programa recibirá como entrada un segundo archivo que almacena diversas fechas y valores a evaluar.
- **Formato y reglas de la entrada:**
    - El programa se ejecuta pasando un archivo como argumento.
    - Cada línea del archivo de entrada debe tener el formato:

"date | value"

    - La **fecha (date)** debe estar en formato **Año-Mes-Día (Year-Month-Day)**.
    - El **valor (value)** debe ser un número (puede ser float o entero positivo) comprendido entre 0 y 1000.
- **Cálculo y validación:**
    - El programa debe leer el archivo de precios y, para cada línea del archivo de entrada, buscar la tasa de cambio correspondiente a la fecha indicada.
    - Si la fecha exacta no existe en la base de datos, se utilizará la fecha **más cercana inferior** a la indicada (es decir, se toma la fecha anterior más próxima).
    - Se deben manejar errores en los datos de entrada, mostrando mensajes apropiados para:
        - Archivos que no se puedan abrir.
        - Valores no positivos o fuera del rango permitido.
        - Formatos de fecha incorrectos, entre otros.
- **Ejemplo de archivo de entrada (input.txt):**

```
date | value
2011-01-03 | 3
2011-01-03 | 2
2011-01-03 | 1
2011-01-03 | 1.2
2011-01-09 | 1
```

2012-01-11 | -1
2001-42-42
2012-01-11 | 1
2012-01-11 | 2147483648

- **Formato de salida esperado:**

El programa mostrará en la salida estándar el resultado del cálculo, por ejemplo:

2011-01-03 => 3 = 0.9
2011-01-03 => 2 = 0.6
2011-01-03 => 1 = 0.3
2011-01-03 => 1.2 = 0.36
2011-01-09 => 1 = 0.32
Error: not a positive number.
Error: bad input => 2001-42-42
2012-01-11 => 1 = 7.1
Error: too large a number.

Propósito de aprendizaje

El ejercicio tiene como finalidad que el estudiante:

- **Manejo de archivos y parsing de datos:**

Aprenda a leer archivos de texto (utilizando, por ejemplo, std::ifstream de <fstream>) y a separar y validar cadenas de texto (empleando std::istringstream de <sstream>).

- **Uso de la STL (Standard Template Library):**

Aunque el ejercicio se podría resolver sin ella, **el objetivo es practicar y aplicar los contenedores (containers) y algoritmos (algorithms) estándar** de C++:

    - Utilización de contenedores como std::map, std::vector, etc.
    - Aplicación de algoritmos de <algorithm> para búsquedas y manipulaciones, lo que resulta esencial para escribir código claro y eficiente.
- **Validación y manejo de errores:**

Implementar una gestión robusta de errores (como entradas mal formateadas, valores fuera del rango, etc.) que garantice la fiabilidad del programa.

- **Diseño modular:**

Estructurar el código en varios ficheros (por ejemplo, main.cpp, BitcoinExchange.cpp, BitcoinExchange.hpp) y aprender a organizar la implementación de templates en headers o archivos .tpp según el estándar C++98.

Enfoque del ejercicio en C++98

1. Manejo de archivos (File Handling)

- **Lectura de ficheros:**

Emplea std::ifstream (del header <fstream>) para abrir y leer tanto la base de datos de precios como el archivo de entrada.

- **Validación de apertura:**

Comprueba que el archivo se abra correctamente; si no es posible abrirlo, se debe emitir el mensaje de error "Error: could not open file.".

2. Análisis de cadenas (Parsing) y validación

- **Uso de <sstream>:**

Para extraer y convertir datos, se puede usar std::istringstream para separar los elementos de cada línea usando delimitadores (como la barra vertical | y el guion - en las fechas).

- **Validación de formato:**
  - La **fecha (date)** debe cumplir el formato YYYY-MM-DD (Año-Mes-Día).

Nota: Al estar en formato ISO, en ocasiones se puede comparar la cadena directamente para determinar el orden cronológico, aunque una conversión a enteros (para año, mes y día) aportaría mayor robustez.

  - El **valor (value)** debe ser un número (float o entero) positivo y menor o igual a 1000.
- **Gestión de errores:**

Se deben detectar y reportar errores específicos, como:

  - Números negativos: "Error: not a positive number."
  - Fechas mal formateadas: "Error: bad input => <entrada>"
  - Números excesivamente grandes: "Error: too large a number."

3. Uso de contenedores de la STL

- **Contenedores recomendados:**
  - **std::map (mapa):**

Ideal para almacenar la relación entre fechas y tasas de cambio.

    - **Clave (key):** La fecha (puede ser un std::string o una estructura definida para la fecha).
    - **Valor (value):** La tasa de cambio (tipo float o double).
  - **std::vector (vector):**

Puede utilizarse para almacenar las entradas del archivo de entrada o para cualquier otra lista de datos que se requiera recorrer secuencialmente.

- **Búsqueda de fechas:**

Para encontrar la tasa de cambio correspondiente a una fecha o, en caso de que no exista, la fecha inmediatamente inferior, se puede usar el método lower_bound del std::map.

Término en inglés: **lower_bound (límite inferior)**.

4. Uso de algoritmos de la STL

- **Cabecera <algorithm>:**

Permite el uso de funciones como std::find_if, entre otros, que pueden facilitar la búsqueda y validación de datos en los contenedores.

- **Aplicación práctica:**

Aunque se pueda resolver sin estos algoritmos, se recomienda su uso para demostrar la correcta aplicación de la STL y mejorar la eficiencia y legibilidad del código.

5. Organización modular del código

- **Ficheros requeridos:**
  - Makefile: Para compilar el programa.
  - main.cpp: Contendrá la función principal y la lógica de ejecución.
  - BitcoinExchange.cpp y BitcoinExchange.hpp: Encapsulan la lógica relacionada con la lectura de la base de datos, el parsing y el cálculo del valor.
- **Templates en C++98:**

Si decides utilizar templates, recuerda que en C++98 es habitual definirlos en los archivos de cabecera, aunque es posible separar la declaración (en el header) y la implementación (en un archivo .tpp). Los archivos de cabecera son obligatorios, mientras que los .tpp son opcionales.

6. Conceptos adicionales relevantes en C++98

- **Iteradores (iterators):**

Son esenciales para recorrer los contenedores de la STL sin necesidad de manipular punteros directamente.

- **Excepciones (exceptions):**

Aunque no es obligatorio, el uso de bloques try-catch para capturar errores durante el parsing o la lectura de archivos es una buena práctica.

- **Formateo de salida:**

La salida del programa debe seguir el formato exacto especificado, mostrando la fecha, el valor de entrada y el resultado del cálculo.

- **Comparación de fechas:**

Al tratarse de fechas en formato ISO (YYYY-MM-DD), se pueden comparar como cadenas, pero una solución robusta implicaría descomponer la fecha en componentes numéricos y compararlos secuencialmente.

7. Consideración sobre las reglas específicas del módulo

- **Uso obligatorio de la STL:**

Aunque es posible resolver el ejercicio sin usar los contenedores y algoritmos estándar, se debe **utilizarlos en la medida de lo posible**. Esto no solo es una práctica recomendada, sino que también es parte de la evaluación, ya que se penalizará la falta de uso de estas herramientas.

- **Buena práctica:**

Aprovecha la modularidad y la reutilización de código que ofrecen los contenedores y algoritmos, lo que te ayudará a escribir un código más claro, conciso y mantenible.

Resumen

El ejercicio **Bitcoin Exchange** te reta a desarrollar un programa en C++98 que, usando dos ficheros CSV (uno con el historial de precios de bitcoin y otro con fechas y valores a evaluar), calcule y muestre el valor resultante de multiplicar la cantidad de bitcoin por la tasa de cambio de la fecha indicada. Para ello, es fundamental:

- Manejar la lectura y validación de archivos.
- Realizar un correcto parsing y verificación de formatos.
- Utilizar contenedores y algoritmos de la STL (como std::map, std::vector y funciones de <algorithm>) para gestionar los datos y realizar búsquedas (por ejemplo, utilizando **lower_bound (límite inferior)**).
- Organizar el código de forma modular y aplicar buenas prácticas de programación en C++98.

Estos conceptos no solo te permitirán solucionar el ejercicio, sino también sentar las bases para el desarrollo de aplicaciones más complejas, haciendo un uso eficiente de las herramientas que ofrece el lenguaje.

Recursos en video recomendados (en español)

Para complementar y profundizar en los conceptos necesarios para afrontar este ejercicio, se recomiendan los siguientes videos:

1. **Curso C++. ¿Por qué aprender C++? Vídeo 1 - Píldoras Informáticas**
   Descripción: Introducción al curso de C++, donde se explican las razones para aprender este lenguaje y se presentan sus características principales.
   Ver video
2. **Curso de C++ Avanzado. STL. Algoritmos de la STL - Píldoras Informáticas**
   Descripción: Video tutorial que aborda los algoritmos disponibles en la Standard Template Library (STL) de C++, con ejemplos prácticos de su uso.
   Ver video
3. **[Básicos] MAP STL C++ - YouTube**
   Descripción: Introducción al contenedor map de la STL en C++, aplicado a problemas de programación competitiva.
   Ver video
4. **Curso C++. Disección del primer programa. Vídeo 5 - Píldoras Informáticas**
   Descripción: Análisis detallado de la estructura de un programa básico en C++, útil para comprender la sintaxis y organización del código.
   Ver video
5. **Curso de C++ Básico Gratis - CódigoFacilito**
   Descripción: Curso introductorio a C++ que cubre desde los conceptos básicos hasta temas más avanzados, incluyendo el uso de la STL.
   Ver curso

Estos recursos proporcionan una base sólida para comprender los conceptos clave necesarios para desarrollar el programa "Bitcoin Exchange" utilizando C++98 y la Standard Template Library.

El código

A continuación se ofrece una explicación completa y detallada de lo que hace el código, dividiéndolo en sus partes principales y describiendo el funcionamiento de cada una de ellas.

1. Archivo de cabecera: BitcoinExchange.hpp

a. Inclusión de librerías y definición de macros

- **Librerías estándar:**
- Se incluyen varias cabeceras estándar para permitir funciones de entrada/salida (<iostream>, <fstream>), manejo de cadenas (<sstream>, <string>), contenedores (<map>) y formateo de la salida (<iomanip>), entre otros.
- **Macros para colores:**

Se definen macros como BLUE, GREEN, PURPLE, RED y RESET para dar formato de color a la salida por consola, lo que facilita la identificación de mensajes (por ejemplo, errores se muestran en rojo).

- **Macros para mensajes de error y ruta del archivo:**

Se definen constantes con mensajes preestablecidos (por ejemplo, DATE_ERR, YEAR_ERR, etc.) y la ruta del archivo de base de datos (FILE "./data.csv").

b. Definición de la clase BitcoinExchange

La clase encapsula la lógica principal para:

- **Almacenar la base de datos:**

Utiliza un contenedor std::map<std::string, float> llamado _bitcoinPrice, donde la clave es una cadena que representa la fecha y el valor es el precio de bitcoin en esa fecha.

- **Constructores y operador de asignación:**

Se declara un constructor por defecto, un constructor de copia, un operador de asignación y un destructor para gestionar correctamente la vida del objeto.

- **Métodos públicos:**
  - readData(std::ifstream &file): Lee y procesa el archivo CSV de la base de datos, almacenando cada par fecha-precio en el mapa.
  - getBitcoinPrice(const std::string date): Devuelve el precio del bitcoin para una fecha dada. Si la fecha no se encuentra exactamente en el mapa, busca la fecha anterior más cercana (utilizando el método lower_bound).
  - Métodos de validación:
    - isDateFormatValid(const std::string &date): Comprueba que la cadena de fecha no esté vacía.

- isValidDate(const std::string &date): Verifica que la fecha cumpla el formato “YYYY-MM-DD” y que cada componente (año, mes y día) sea correcto y esté en un rango aceptable.
- isDataInRange(const std::string &rate): Revisa si la cadena que representa un número está en un rango y formato válido, comprobando además que no contenga caracteres indebidos o sea un número negativo o demasiado grande.

2. Archivo de implementación: BitcoinExchange.cpp

a. Constructores, operador de asignación y destructor

- **Constructor por defecto y destructor:**

Permiten la creación y destrucción correcta del objeto.

- **Constructor de copia y operador de asignación:**

Se encargan de copiar la estructura interna (el mapa _bitcoinPrice) de otro objeto BitcoinExchange.

b. Función auxiliar ft_stof

- **ft_stof:**

Esta función recibe una cadena y utiliza un std::stringstream para convertirla a float. Es equivalente a una versión simplificada de std::stof (que no existía en C++98) y se utiliza para convertir los valores numéricos que se leen desde los archivos.

c. Método getBitcoinPrice

- **Validación de la fecha:**

Antes de buscar el precio, se valida la fecha con isValidDate. Si no es válida, se retorna un valor (en este caso, 1) que puede usarse para detectar un error.

- **Búsqueda en el mapa:**
  - Si la fecha existe en _bitcoinPrice, se retorna directamente el precio.
  - Si no existe, se usa lower_bound para encontrar el primer elemento cuya clave no sea menor que la fecha dada.
    - Si la fecha consultada es anterior a la primera registrada, se emite un mensaje de error y se retorna -1.
    - Si la iteración llega al final del mapa, se devuelve el último precio disponible.
    - En el caso general, se decrementa el iterador (con --it) para obtener la fecha inmediatamente anterior, cumpliendo así la regla de usar la fecha inferior más cercana.

d. Métodos de validación de fecha y datos

- **isDateFormatValid:**

Verifica que la cadena no esté vacía, mostrando un error si lo está.

- **isValidDate:**

Comprueba el formato de la fecha (longitud, posiciones de guiones) y que cada parte (año, mes y día) contenga únicamente dígitos. Luego convierte estas partes a enteros y verifica:

- Que el año no sea menor que 2009 (año en que se creó Bitcoin).
- Que el mes esté entre 1 y 12.
- Que el día sea válido, considerando los meses que tienen 30 o 31 días y las particularidades de febrero (incluyendo años bisiestos).

- **isDataInRange:**

Revisa que el string numérico no contenga caracteres no permitidos y se encuentre dentro de límites razonables (por ejemplo, no negativo, y no mayor a cierto valor).

e. Método readData

- **Lectura y procesamiento de la base de datos:**

Este método lee el archivo CSV (la base de datos de precios) línea por línea.

- Primero, se intenta detectar si la primera línea es un encabezado (buscando palabras como "date" o "value").
- Para cada línea, se separa el contenido en dos partes usando la coma (,). Se extrae la fecha y el valor.
- Se valida la fecha mediante isValidDate y se convierte el valor a float (usando un stringstream).
- Si todo es correcto, se inserta el par (fecha, precio) en el mapa _bitcoinPrice.

3. Archivo principal: main.cpp

a. Comprobación de argumentos y apertura de archivos

- **Número de argumentos:**

Se verifica que el programa reciba exactamente un argumento (además del nombre del ejecutable). Si no, se muestra un error (BAD_ARG) y se termina la ejecución.

- **Apertura de archivos:**
  - Se abre el archivo de entrada (cuya ruta se pasa por línea de comandos).
  - Se abre el archivo interno de la base de datos (data.csv definido en la macro FILE).

Si alguno de estos archivos no se puede abrir, se muestra el mensaje de error correspondiente.

b. Procesamiento de la entrada y uso de la base de datos

- **Instanciación y carga de datos:**

Se crea un objeto BitcoinExchange y se llama a readData para que lea y almacene en el mapa la base de datos de precios.

- **Procesamiento del archivo de entrada:**
  - Se lee la primera línea, que normalmente es el encabezado, y luego se procesan las siguientes líneas.
  - Para cada línea:
    - Se busca el carácter '|' que separa la fecha del valor.

- Se extrae la **fecha** y se valida usando los métodos isDateFormatValid e isDataInRange (para asegurarse de que la cadena tiene el formato correcto y está dentro de un rango esperado).
- Se extrae el valor de adquisición (buys) que se encuentra después del delimitador y se valida de forma similar.
- Se convierte el valor de la cadena a float mediante ft_stof.
- Se comprueba que el valor esté entre 0 y 1000; en caso contrario, se muestra un error (BUYS_ERR).
- Se obtiene el precio del bitcoin para la fecha mediante getBitcoinPrice. Si este método devuelve un valor distinto de 1 (el cual se usa aquí como indicador de que la fecha es válida y se encontró un precio), se calcula el resultado multiplicando el valor de compra por el precio del bitcoin.
- Se muestra en pantalla la fecha, el valor de compra y el resultado (usando formateo fijo y precisión de dos decimales, además de aplicar colores para diferenciar los datos).

c. Cierre de archivos y fin de la ejecución

- Finalmente, se cierran ambos archivos abiertos y se retorna EXIT_SUCCESS para indicar que la ejecución finalizó correctamente.

Resumen general

El código implementa un programa que:

1. Carga una base de datos interna de precios de bitcoin (en formato CSV) en un contenedor std::map, lo que permite búsquedas rápidas basadas en fechas.
2. Lee un archivo de entrada que contiene líneas con el formato "fecha | valor", validando que las fechas y los valores sean correctos.
3. Para cada entrada, busca en la base de datos el precio del bitcoin correspondiente a la fecha (o la fecha inmediatamente anterior en caso de no encontrar una coincidencia exacta).
4. Calcula y muestra el resultado de multiplicar el valor indicado por el precio del bitcoin, aplicando formateo de salida y mostrando mensajes de error cuando se detectan entradas inválidas.

El uso de la STL (en particular el std::map y los std::stringstream para la conversión y validación de datos) es fundamental para lograr una solución limpia y eficiente conforme a las reglas del ejercicio. Además, la separación en distintos archivos (cabecera, implementación y función principal) favorece la modularidad y el mantenimiento del código.

Esta estructura y lógica permite al programador cumplir con los requisitos del ejercicio "Bitcoin Exchange", garantizando la validación rigurosa de entradas y la correcta búsqueda y utilización de datos históricos de bitcoin para el cálculo requerido.