ASPECTOS BÁSICOS DE LA POO

GESTIÓN DE DIETARIO

Índice

Introducción

Objetivos

En esta práctica desarrollaremos un completo sistema en C++ para manejar un dietario, esencial para el seguimiento de la contabilidad personal a través de ingresos y gastos. Cada registro incluirá detalles importantes como la fecha, concepto y cantidad, donde los ingresos son representados por cantidades positivas y los gastos por negativas. El sistema clasificará las anotaciones por día y concepto.

Se estructura alrededor de clases clave: **Anotacion** para los registros individuales, **Fecha** para las fechas, **Dia** que agrupa anotaciones bajo una fecha, **Dietario** como la colección de todos los días registrados, y **Menu** que facilita la interacción del usuario con opciones para añadir, eliminar, o salir.

El enfoque está en mantener al usuario constantemente informado del estado del dietario, mostrando un resumen financiero junto con las opciones disponibles, asegurando así una gestión efectiva y ordenada de las finanzas personales.

Herramientas de Programación Utilizadas

En este apartado, exploraremos las diversas herramientas utilizadas en el desarrollo de la práctica para garantizar la eficiencia del programa:

- **Clases y Objetos:** Pilares de la programación orientada a objetos, permiten modelar elementos del mundo real dentro del software, encapsulando datos y comportamientos.

  En el proyecto la clase **Dia** es un excelente ejemplo. Se define para encapsular todos los detalles y operaciones relacionados con un día específico, incluyendo las anotaciones de ingresos y gastos. Esta clase almacena una colección de objetos **Anotacion**, demostrando la relación entre clases y objetos en un contexto real de aplicación.

```
class Dia {
private:
    Fecha fecha;
    vector<Anotacion> anotaciones;
```

- **Constructores**: Métodos especiales para inicializar objetos de una clase, esenciales para establecer un estado inicial coherente.

  En mi código, el constructor de la clase **Fecha** inicializa un objeto con valores específicos para día, mes y año, garantizando que cada **Fecha** comience con datos coherentes y precisos.

```
Fecha(int dia, int mes, int anyo) : dia(dia), mes(mes), anyo(anyo) {}
```

- **Operadores**: Facilitan la interacción con objetos, permitiendo definir y personalizar el comportamiento de operaciones estándar como suma o comparación.

La sobrecarga de operadores permite a los objetos de tus clases interactuar mediante operadores conocidos como **+**, **-**, **==**. Por ejemplo, sobrecargar el operador **==** para la clase **Fecha**, permitiendo comparar dos objetos **Fecha** directamente.

```
// Sobrecarga del operador ==
bool operator==(const Fecha& otraFecha) const {
    return dia == otraFecha.dia && mes == otraFecha.mes && anyo == otraFecha.anyo;
}
```

- **Memoria Dinámica:** Asignar memoria durante la ejecución del programa, crucial para la gestión eficiente de recursos y la flexibilidad del software.

  Manejar memoria dinámica es crucial para aplicaciones que necesitan adaptarse a una cantidad variable de datos en tiempo de ejecución. En el proyecto, esto se evidencia en el uso de **std::vector** para almacenar anotaciones dentro de **Dia**, lo cual permite agregar o quitar anotaciones dinámicamente.

```
vector<Anotacion> anotaciones;
```

Código Fuente

El código fuente del programa es un reflejo directo de cómo hemos aplicado los conocimientos teóricos de C++ para solucionar el problema de gestión de un dietario digital. La organización del código en clases específicas como `**Menu**`, `**Dietario**`, `**Dia**`, `**Fecha**`, y `**Anotacion**` facilita la comprensión y mantenimiento del mismo. A continuación, se describen las características clave del código.

En primer lugar, tenemos la clase **Anotación**, que representa una anotación con un concepto y una cantidad asociados. Se incluye el archivo de encabezado <*string*> para utilizar la clase *'string'*. Esta clase tiene variables miembro privadas para almacenar el concepto y la cantidad de la anotación, así como funciones miembro públicas para acceder y modificar estos valores. Se define un constructor que inicializa una instancia de **Anotacion** con un concepto y una cantidad específicos, y también se proporcionan funciones *getter* y *setter* para obtener y establecer estos valores respectivamente. Además, se utiliza un bloque de preprocesador para evitar la inclusión múltiple de este archivo de encabezado en el programa.

La segunda clase **Dia** representa un día con una fecha y un conjunto de anotaciones asociadas. Se incluyen los archivos de encabezado <*vector*> para utilizar la clase *'vector'*, así como los archivos de encabezado de las clases **Fecha** y **Anotacion**. La clase tiene variables miembro privadas para almacenar la fecha del día y un vector de anotaciones. Se proporciona un constructor que inicializa una instancia de Dia con una fecha específica, y un método *'agregarAnotacion'* para agregar nuevas anotaciones al día, permitiendo acumular cantidades si el concepto de la anotación ya existe en el día. Además, se utilizan funciones *getter* para obtener la fecha y las anotaciones del día, y se utiliza un bloque de preprocesador para evitar la inclusión múltiple de este archivo de encabezado en el programa.

La tercera clase **Dietario** representa un conjunto de días con anotaciones asociadas. Se incluyen los archivos de encabezado `<*vector*>` para utilizar la clase `*std::vector*`, así como el archivo de encabezado de la clase **Dia**. La clase tiene un vector privado de días para almacenar las entradas del dietario. Se proporciona un método `*agregarAnotacion*` para agregar una nueva anotación a un día específico, asegurándose de mantener los días

ordenados por fecha. Además, se proporciona un método `*borrarDietario*` para eliminar todas las entradas del dietario. Se utiliza un bloque de preprocesador para evitar la inclusión múltiple de este archivo de encabezado en el programa.

La cuarta clase **Fecha** representa una fecha con día, mes y año. La clase incluye variables miembro privadas para almacenar el día, el mes y el año. Se proporciona un constructor que inicializa una instancia de Fecha con los valores proporcionados para día, mes y año. Además, se incluyen funciones *getter* para obtener el día, el mes y el año de la fecha.

Se sobrecargan los operadores de comparación == y < para permitir la comparación de objetos Fecha. El operador == compara si dos fechas son iguales en día, mes y año, mientras que el operador < compara las fechas en orden cronológico, primero por año, luego por mes y finalmente por día.

La última clase es **Menu** proporciona una interfaz de usuario para interactuar con el dietario. Se incluyen los archivos de encabezado necesarios y se utiliza la clase **Dietario** para almacenar los días y anotaciones.

El método `*mostrarMenu*` muestra las opciones disponibles al usuario y procesa la opción seleccionada hasta que el usuario decide salir. Dentro del menú, se llama a `*mostrarDietario*` para mostrar el contenido actual del dietario y se presentan opciones para agregar nuevas anotaciones o borrar el dietario.

El método `*mostrarDietario*` muestra todas las anotaciones almacenadas en el dietario, mientras que `*mostrarOpciones*` presenta las opciones del menú al usuario. El método `*obtenerOpcion*` obtiene la opción seleccionada por el usuario y se asegura de que sea válida.

Se proporcionan funciones privadas adicionales como `*nuevaAnotacion*` para agregar una nueva anotación al dietario, `*esNumero*` para verificar si una cadena es un número válido, `*esFechaValida*` para verificar si una fecha es válida y `*borrarDietario*` para eliminar todas las entradas del dietario.

El código utiliza `*std::chrono*` y `*std::thread*` para realizar pausas en la ejecución del programa y `*std::all_of*` para verificar si todos los caracteres de una cadena son dígitos.

Como resumen de las clases que acabamos comentar podemos decir:

- **Clase Dietario**: Es el núcleo de la aplicación, donde se gestionan las anotaciones. Permite agregar nuevas anotaciones, calcular totales de ingresos y gastos, y mostrar el estado actual del dietario.

- **Clase Menu:** Sirve como punto de entrada para la interacción del usuario con el programa, mostrando las opciones disponibles y gestionando las entradas para realizar las operaciones correspondientes.

- **Clases Fecha, Dia, Anotacion**: Modelan los elementos básicos del dietario, representando las fechas, los días con sus respectivas anotaciones, y los detalles de cada anotación.

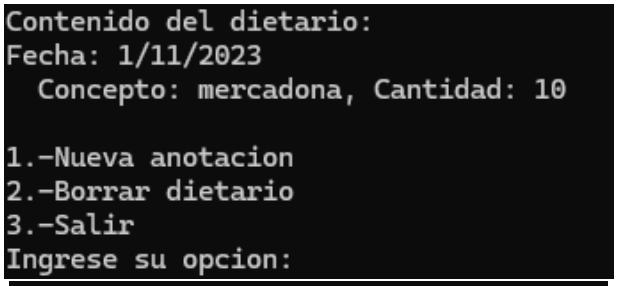
Errores y Tecnologías

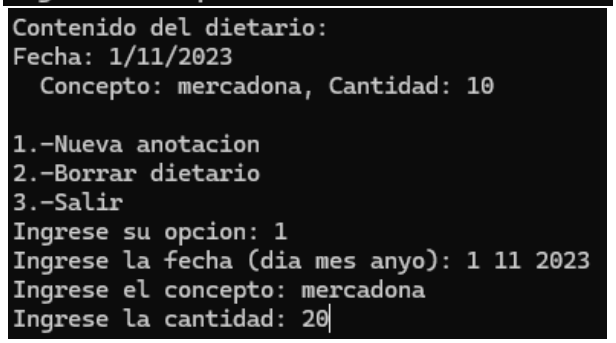
Durante el desarrollo, se identificaron y corrigieron varios errores, especialmente relacionados con la gestión de la memoria y la correcta implementación de las funcionalidades solicitadas. La utilización de tecnologías como vectores de la STL para almacenar dinámicamente las anotaciones y días ha simplificado la gestión de la memoria y la implementación de las funcionalidades requeridas.

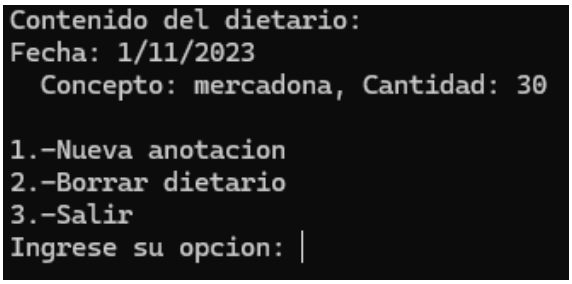
Experimentación y validación

En la fase de experimentación, se llevaron a cabo pruebas concretas para evaluar la robustez y funcionalidad del programa. Algunos experimentos incluyeron:

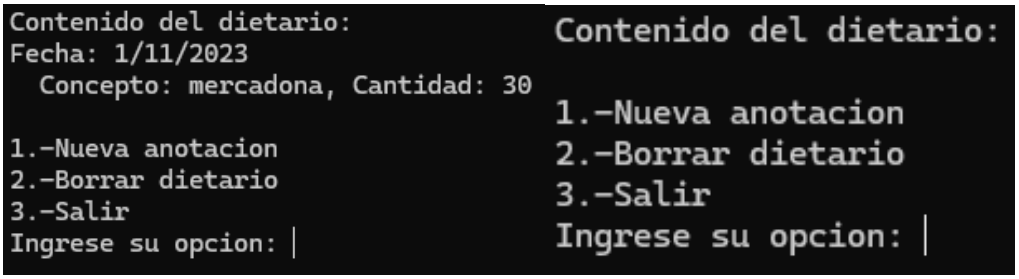
1. **Anotaciones con el mismo nombre:** Se comprobó si al ingresar dos anotaciones con el mismo nombre, sus cantidades se sumaban correctamente, asegurando una gestión adecuada de ingresos o gastos repetidos.



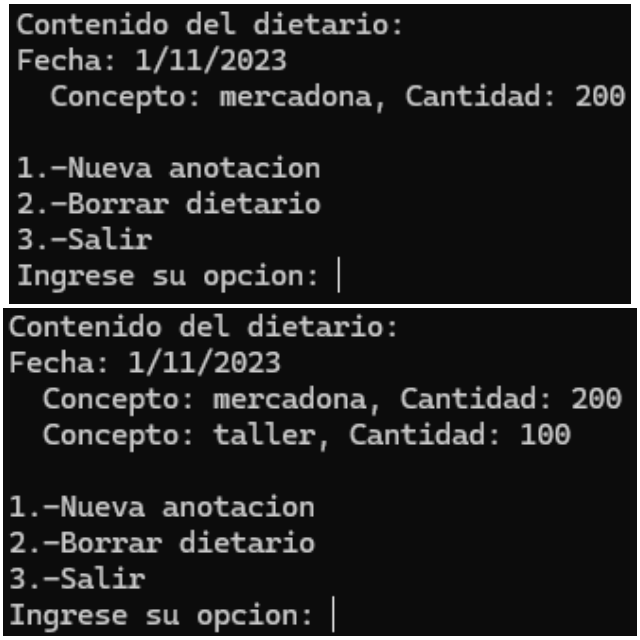




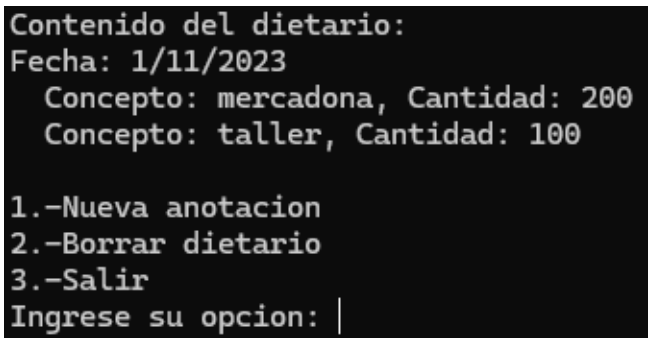
2. **Borrado de anotaciones:** Se validó que, al eliminar una anotación, el programa limpiaba correctamente el registro sin dejar residuos, manteniendo la integridad del dietario.



3. **Inserción de fechas repetidas:** Se verificó que al introducir anotaciones con la misma fecha, estas se integraban en un mismo día, demostrando una correcta organización temporal de los datos.



4. **Varias anotaciones en diferentes fechas:** Se realizaron pruebas para asegurar que el programa ordenaba adecuadamente las anotaciones en pantalla por fecha, facilitando la visualización y el seguimiento del dietario.



```
Contenido del dietario:
Fecha: 1/10/2023
  Concepto: pc, Cantidad: 100
Fecha: 1/11/2023
  Concepto: mercadona, Cantidad: 200
  Concepto: taller, Cantidad: 100

1.-Nueva anotacion
2.-Borrar dietario
3.-Salir
Ingrese su opcion:
```

5. **Control de formato de fecha:** Verificar que la fecha introducida por el usuario tenga el formato correcto (dd mm aaaa) y que los valores proporcionados estén dentro de rangos válidos (día entre 1 y 31, mes entre 1 y 12, año entre 1900 y 2100).

```
Contenido del dietario:

1.-Nueva anotacion
2.-Borrar dietario
3.-Salir
Ingrese su opcion: 1
Ingrese la fecha (dia mes anyo): 1/11/2023
Fecha invalida. Por favor, ingrese una fecha valida.
Ingrese la fecha (dia mes anyo): 32 13 2110
Fecha invalida. Por favor, ingrese una fecha valida.
Ingrese la fecha (dia mes anyo): 1-11-2023
Fecha invalida. Por favor, ingrese una fecha valida.
Ingrese la fecha (dia mes anyo):
```

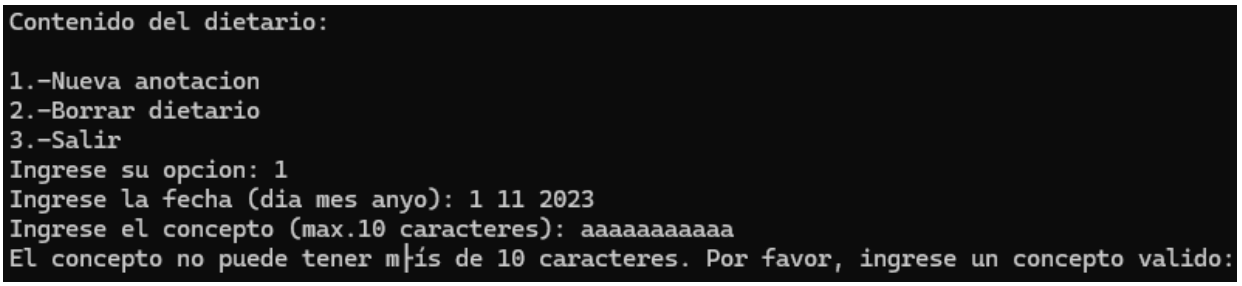
6. **Control de concepto vacío:** Verificar que el usuario no deje el concepto de la anotación vacío al agregar una nueva anotación.

```
Contenido del dietario:

1.-Nueva anotacion
2.-Borrar dietario
3.-Salir
Ingrese su opcion: 1
Ingrese la fecha (dia mes anyo): 11 11 2023
Ingrese el concepto:
El concepto no puede estar en blanco. Por favor, ingrese un concepto valido:
```

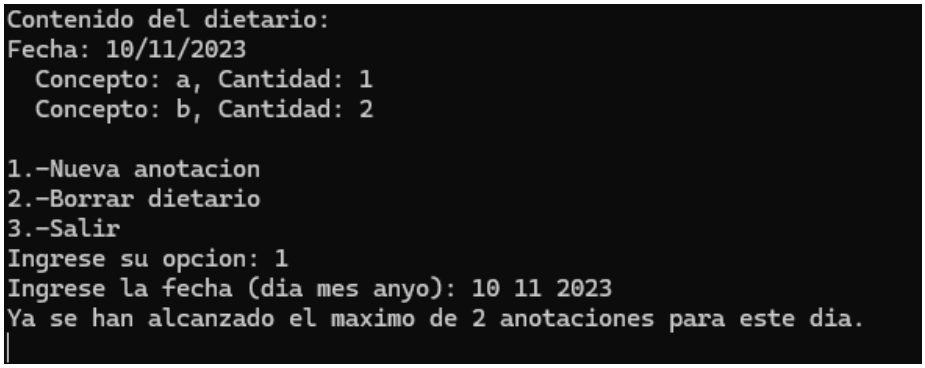
7. **Control de opciones inválida en el menú:** Validar que las opciones introducidas por el usuario en el menú principal sean números válidos y estén dentro del rango de opciones disponibles.

```
Contenido del dietario:

1.-Nueva anotacion
2.-Borrar dietario
3.-Salir
Ingrese su opcion: 5
Opcion invalida. Por favor, ingrese una opcion valida.
```

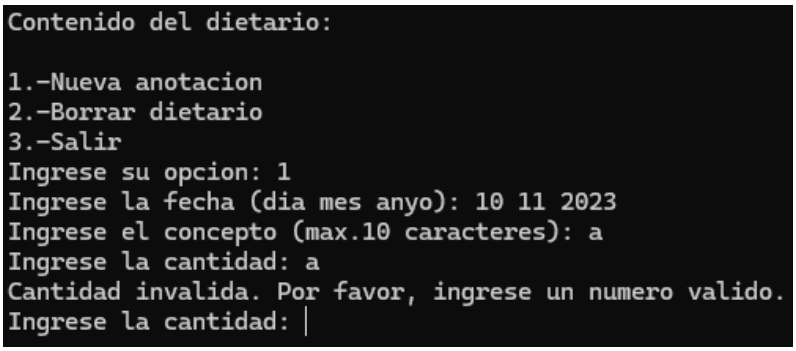
8. **Control de límite de longitud del concepto:** Limitar la longitud del concepto de la anotación para evitar desbordamientos de búfer.



9. **Control de cantidad máxima de anotaciones por día:** Establecer un límite máximo para la cantidad de anotaciones que se pueden agregar en un día específico para evitar sobrecargar el dietario.



10. **Control entrada cantidad:** Verificar que el usuario no pueda introducir letras en cantidad.



Estos experimentos no solo ayudaron a identificar áreas de mejora, sino que también confirmaron la capacidad del programa para manejar escenarios de uso realistas, demostrando su eficacia y robustez. Las pruebas y sus resultados, apoyados por capturas de pantalla incluidas en la documentación, subrayan la funcionalidad y fiabilidad del software desarrollado.

Conclusión

La realización de esta práctica ha sido una experiencia sumamente enriquecedora, que no solo ha permitido aplicar los conocimientos teóricos adquiridos en programación C++ en un proyecto práctico real, sino que también ha desafiado la capacidad de resolver problemas y optimizar soluciones. El proceso de diseño, desarrollo, y experimentación ha facilitado una comprensión más profunda de cómo los conceptos fundamentales de la programación orientada a objetos pueden ser implementados de manera eficaz para desarrollar software robusto y confiable.