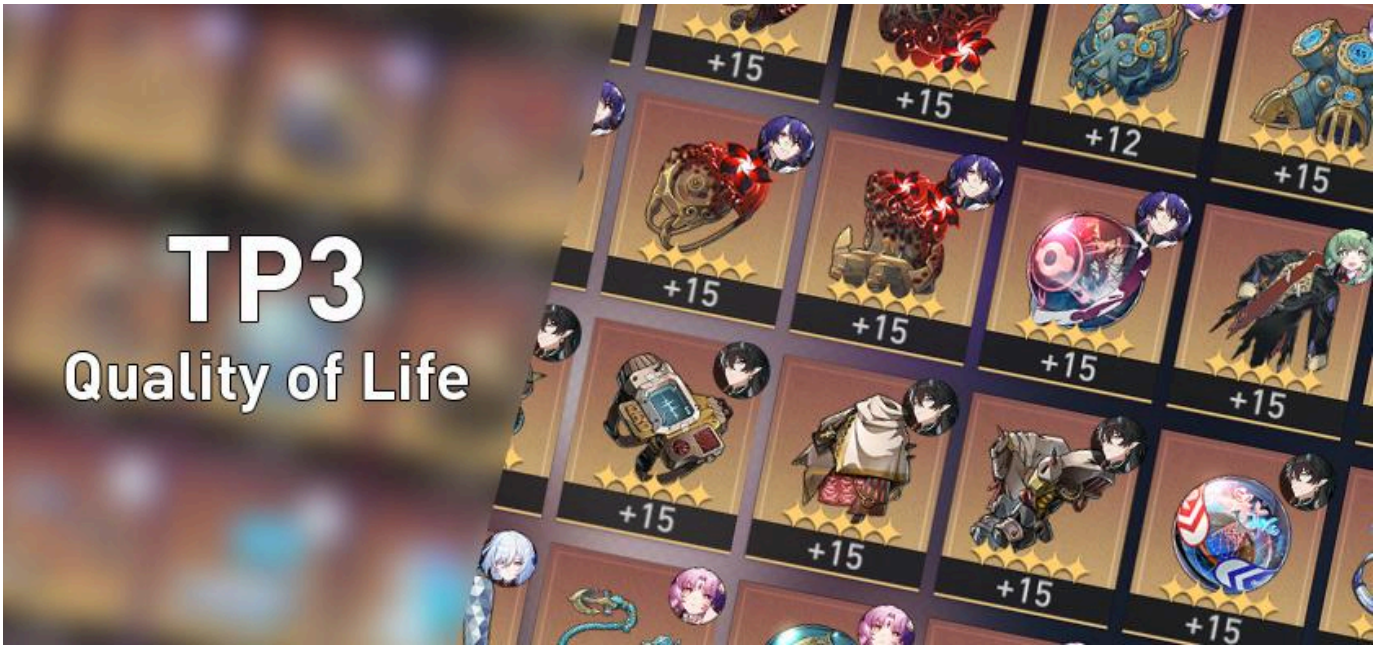

El contexto:

Siguiendo el desarrollo de la versión 1.0 de *Honkai Star Rail*, sus siguientes tareas están relacionadas a sistemas menores pero altamente importantes para los jugadores ya que son mejoras *Quality of Life*, o *QoL* por sus siglas. Estas mejoras de “calidad de vida” apuntan a quitar procesos tediosos a la hora de jugar como clics innecesarios, navegación por menús tediosos, entre otros ejemplos.

De la experiencia del desarrollo de *Genshin Impact*, se sabe que estas mejoras son las más pedidas y las mejores recibidas por la comunidad, por lo que se apuntará a incluir la mayor cantidad posible en la versión de salida del juego.

En esta línea, su tarea será desarrollar dos sistemas: los **Armamentos de Artefactos** y los **Combates Múltiples**.

Imágenes 1 y 2: A la izquierda, los **Armamentos de Artefactos**. A la derecha, los **Combates Múltiples**.

Sobre los Combates Múltiples:

Como en la entrega anterior, los jugadores gastarán su *Poder de Trazacamino* (material que se recarga con el tiempo) para poder iniciar **Combates** que, al finalizarlos exitosamente, darán recompensas como experiencia, materiales de ascensión, entre otros.

Como iniciar **Combates** individuales puede ser excesivamente tedioso (considerando que es una actividad diaria), se desea poder *encadenar hasta seis* ***Combates*** para así pelear de forma ininterrumpida. Estos **Combates** se pelearán *de forma sucesiva, empezando por el* ***primero*** *agregado*. Un **Combate Múltiple** *no puede ser interrumpido y deben pelearse todos los* ***Combates*** para obtener todas las recompensas.

Imágen 3: Ejemplo de un **Combate Múltiple** en el juego, indicando el número de **Combate** actual.

Sobre los Armamentos de Artefactos:

Una de las funcionalidades más pedidas en la entrega anterior es *poder crear **Armamentos**, conjuntos de **Artefactos*** para luego poder equiparlos e intercambiarlos rápidamente entre distintos personajes.
Estos **Armamentos** tendrán, por lo tanto, un *nombre definido por el usuario*, el *UUID* del usuario creador y *hasta seis **Artefactos** diferentes*.
Adicionalmente, se desea poder visualizarlos por pantalla, mostrando la información del **Artefacto** actual y dando la posibilidad de mostrar *tanto el siguiente **Artefacto** como el anterior, en todo momento*. Por ejemplo, mostrar el anterior al *casco* nos muestra la *pista de luz*, como se ve en la siguiente imagen.

Imágen 4: Ejemplo de la visualización de **Armamentos**.

Finalmente, como a los jugadores les interesa compartir los diferentes **Armamentos** en los foros web del juego, se desea poder *exportarlos* a un archivo **.csv** con el nombre del **Armamento** y el **usuario que lo creó** como nombre del archivo, así como *importar* un **Armamento** a partir de un archivo.

Funcionalidades a implementar, parte 1:

Concretamente, la tarea será implementar las clases **armamento_artefactos** y **combate_multiple**, aplicando el paradigma orientado a objetos.
Los métodos a implementar en la clase **armamento_artefactos** son:

1. ***armamento_artefactos****(std::string path_archivo)*:
   Constructor sobrecargado que permitirá crear un Armamento a partir de un archivo.
2. *void* ***agregar_artefacto****(artefacto artefacto_a_agregar)*:
   Este método agregará el artefacto al armamento en una posición arbitraria. No se puede agregar el mismo artefacto dos veces. No se pueden agregar más de seis artefactos.
3. *void* ***mostrar_artefacto_actual****()*:
   Este método mostrará el artefacto actual por pantalla.
4. *void* ***mostrar_artefacto_siguiente****()*:
   Este método mostrará el artefacto siguiente al actual por pantalla.
5. *void* ***mostrar_artefacto_anterior****()*:
   Este método mostrará el artefacto anterior al actual por pantalla.
6. *void* ***mostrar_artefactos****()*:
   Este método mostrará todos los artefactos por pantalla. El primer artefacto mostrado es arbitrario.
7. *void* ***quitar_artefacto****()*:
   Este método quitará el artefacto actual del armamento.
8. *void* ***exportar_armamento****()*:
   Este método guardará el armamento actual, con el formato especificado a continuación.

NOTA: Considerar, para todos los métodos, qué pasa si el armamento está vacío.

Formato de los archivos exportados/importados:

El nombre del archivo será una combinación del UUID del usuario y el nombre del armamento. El UUID es un número identificador único de 8 dígitos, generado para cada usuario:

UUID-NOMBRE_ARMAMENTO.csv

En el archivo, se guardará cada artefacto de la siguiente forma:

ID,SET,TIPO,NIVEL,RAREZA

NOTA: El formato de los archivos importados (incluyendo la cantidad máxima de artefactos) **siempre es correcto**.

Funcionalidades a implementar, parte 2:

Los métodos a implementar en la clase **combate_multiple** son:

1. *void* ***agregar_combate****(combate combate_a_agregar)*:
   Este método agregará el combate. No se pueden agregar más de seis combates.
2. *size_t* ***pelear****()*:
   Este método resolverá todos los combates en el orden especificado anteriormente, mostrando en el proceso la información de cada uno, y devolverá la cantidad total de *Poder de Trazacamino* gastado.
   NOTA: Este método **debe ser implementado recursivamente**, clasificando en la documentación el tipo de recursividad.

Finalmente, se deberá implementar una aplicación sencilla pero **totalmente funcional**, que permita ver y usar todas las funcionalidades desarrolladas.

Enlace al repositorio base del Trabajo Práctico

Aclaraciones y criterios de corrección:

Para que el trabajo tenga buena recepción por nuestros superiores en Hoyoverse, se deberán cumplir las siguientes pautas:

1. Debe implementar las clases de manera totalmente original y propia. La detección de copias resultará en la **pérdida de la regularidad de los involucrados**.
2. El trabajo debe compilar con las flags **-Wall -Werror -Wconversion**. Un trabajo que no compila es un trabajo que *no funciona* (y por lo tanto no pasa ninguna prueba).
3. El código **debe pasar** los tests de la cátedra y también respetar sus firmas (es decir, no se puede *modificar* las pruebas).
4. El código **debe estar escrito** en *snake_case*.

Adicionalmente:

1. **No se puede utilizar** la librería STL de C++.
2. Se pueden utilizar otras librerías que **no sean de estructuras de datos**.

Los criterios de evaluación y corrección por parte de la cátedra son:

1. **Funcionalidad** (50% de la nota):
   Compilación: sin warnings ni errores.
   Funcionalidad: que el código pase las pruebas.
   Memoria dinámica: el código no debe perder memoria. Tampoco debe haber errores de acceso a memoria.
   Manejo de archivos.
2. **Estilo de código** (50% de la nota):
   Uso del paradigma.
   Eficiencia espacial.
   Eficiencia temporal.
   Modularización.
   Precondiciones y postcondiciones.
   Buenas prácticas de programación propuestas por la cátedra.

Formato de entrega:

Se deberá subir el código a la branch *main* del repositorio de GitHub. Adicionalmente, se deberá completar el *README.md* con la información del estudiante. Se corregirá el **último commit** del *main* del repositorio remoto.

Como formalidad, se deberá subir al campus un único archivo comprimido **.zip** en la sección TPs. En la carpeta comprimida, se deberá entregar el repositorio de GitHub. Esto se puede hacer descargándolo de forma manual desde la página o generando una Release.

El nombre del archivo **debe tener** el siguiente formato:

tp3-1c2024-USERNAME-PADRON.zip

El código entregado en el **.zip** *no* será revisado (salvo casos borde).

El plazo de entrega vence el día **MIÉRCOLES 22 DE MAYO 23:59 hrs**. No se aceptarán entregas fuera de término.

Puntaje: **50** puntos:

- Funcionalidad clase **cola**: **5** puntos.
- Funcionalidad clase **lista_circular**: **15** puntos.
- Memoria dinámica: **5** puntos.
- Manejo de archivos: **5** puntos.
- Estilo de código: **20** puntos.

Buena suerte con los 50/50