

## TP2

# Los Saltos

### El contexto:

Su siguiente tarea en el desarrollo de la versión 1.0 de *Honkai Star Rail* es implementar el sistema más crucial (y polémico, con buena razón): el sistema de **saltos** (comúnmente conocido como **gacha**).

Este sistema es con el cual los jugadores deben interactuar para obtener tanto personajes como equipamiento. Cada **salto** recompensa al jugador con un objeto de rareza aleatoria. Se pueden realizar **saltos** utilizando la moneda ficticia del juego o, alternativamente, dinero de la vida real, constituyendo así la principal fuente de ingresos de la compañía.

Dejando de lado la moralidad de este tipo de sistemas, nos concentraremos solamente en implementar el sistema **gacha**, que usarán los jugadores cuando el juego salga al público.

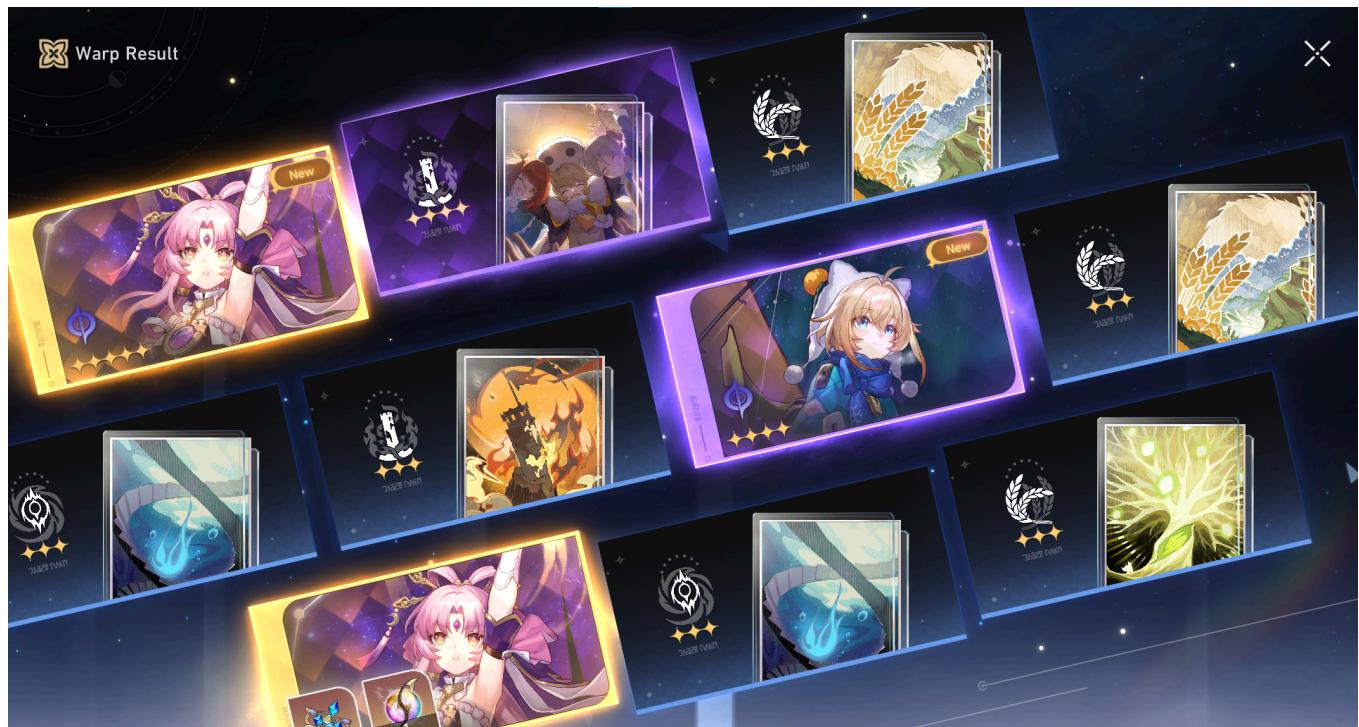


Imagen 1: Ejemplo de un "salto múltiple", con dos saltos de 5 estrellas y dos de 4 estrellas.

# Cómo se genera un salto:

En el juego se pueden obtener saltos de tres tipos de rareza: **5 estrellas**, **4 estrellas** y **3 estrellas**.

Al generar un salto, su rareza estará determinada por los siguientes porcentajes:



Imagen 2: Distribución de rareza. 5 estrellas: 1%, 4 estrellas: 10%, 3 estrellas: 89%.

**NOTA:** La rareza del salto se debe determinar con *un solo número aleatorio*.

## Sistema de Pity:

Para aliviar la mala suerte que pueden tener los jugadores (considerando que el sistema es aleatorio por naturaleza), se implementará adicionalmente un sistema de “Pity”, del inglés *pena*. Este sistema garantiza a los jugadores un salto de mayor rareza en función de cuántos saltos ya se han realizado.

**En el caso de que la rareza determinada sea 3 estrellas**, las siguientes reglas aplicarán:

1. Si el jugador realizó *89 saltos o más* sin conseguir un **5 estrellas**, entonces el salto actual será de **5 estrellas** de forma asegurada.
2. De forma similar, si el jugador realizó *9 saltos o más* sin conseguir un **4 estrellas**, entonces el salto actual será de **4 estrellas** de forma asegurada.
3. En caso de que se deba generar *simultáneamente* un salto de 5 y 4 estrellas, **se priorizará al de 5 estrellas**. Por ejemplo, si los contadores son 89 y 9 (**5 y 4 estrellas** respectivamente), se generará un salto de **5 estrellas** y los contadores pasarán a ser 0 y 10.

## Sobre los contadores:

Para que el sistema de Pity funcione, se deberá llevar la cuenta de cuantos saltos realizó el jugador sin obtener un salto de rareza **5 o 4 estrellas**.

Estos contadores **deben reiniciarse** si el salto generado es de **5 o 4 estrellas** (respectivamente) *independientemente de si se generó por la probabilidad base o por el sistema de Pity*.

Por ejemplo, si los contadores son 30 y 5 (**5 y 4 estrellas** respectivamente) y se genera un **5 estrellas**, los contadores pasarán a ser 0 y 6.

## Sobre la clase “salto” y “generador\_salto”:

Estas clases ya fueron implementadas por otro equipo de desarrollo en *Hoyoverse* y se podrán usar libremente para generar instancias de la clase **Salto**. De esta forma, podrán dedicarse exclusivamente a implementar el sistema descrito anteriormente. *Importante:* Estas clases **NO** deben ser alteradas.

# Funcionalidades a implementar:

Concretamente, la tarea será implementar la clase **gacha**, aplicando el paradigma orientado a objetos.

Los métodos a implementar son:

1. `salto generar_salto()`:

Este método generará y devolverá un salto, de acuerdo con lo planteado anteriormente.

2. `vector<salto> generar_salto_multiple()`:

Este método generará y devolverá *diez* saltos dentro de un vector. Equivale a ejecutar `generar_salto()` diez veces.

3. `vector<salto> generar_salto_multiple(size_t cantidad)`:

Este método generará y devolverá *cantidad* de saltos dentro de un vector. Equivale a ejecutar `generar_salto()` *cantidad* de veces.

4. `void exportar_saltos()`:

Este método guardará, a modo de registro, la información de todos los saltos generados en un archivo `registro_saltos.csv`. No debe sobrescribir información anterior. Cada línea es un salto.

Adicionalmente, se deberá implementar el **TDA template vector**, que luego utilizarán en la clase **gacha** para devolver saltos múltiples.

**Se debe respetar la interfaz pública propuesta por la cátedra:** los métodos a implementar ya están declarados en el [repositorio base del Trabajo Práctico](#), con su documentación y pruebas. Las *pre* y *postcondiciones* de cada método les servirán de guía para programar el TDA.

Finalmente, se deberá implementar un `main.cpp` sencillo que les permita mostrar por pantalla el resultado de los saltos y verificar la salida del archivo, a modo de prueba.

*Algunas observaciones:*

1. El vector debe manejar su memoria **tanto al dar de alta como al dar de baja**. Para esto, pueden implementar un aumento/decremento lineal o, idealmente, [factor of two](#). Por lo tanto, el vector **no puede tener un tamaño fijo**.
2. La información del salto la pueden imprimir o guardar en un archivo de la forma `stream << salto`.
3. Abrir archivos es una operación costosa. Al exportar los saltos, consideren guardar toda la información abriendo una sola vez el archivo `registro_saltos.csv`.
4. ¿Qué pasa si llamamos al método `exportar_saltos()` múltiples veces? ¿Se debería guardar información repetida?
5. Piensen qué [modo de apertura de archivo](#) necesitan usar en este caso.
6. Pueden crear métodos y atributos **privados** y constructores **públicos** en la clase **gacha**, en caso de necesitarlo. También pueden crear más clases.
7. Pueden crear métodos **privados** en la clase **vector**.

# 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++ (y, particularmente, *std::vector*).
2. Se puede utilizar otras librerías que **no sean de estructuras de datos**, como **<random>**.

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:

**tp2-1c2024-USERNAME-PADRON.zip**

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

El plazo de entrega vence el día **DOMINGO 28 DE ABRIL 23:59 hrs**. No se aceptarán entregas fuera de término.

Puntaje: **40** puntos:

- Funcionalidad clase **vector**: **10** puntos.
- Funcionalidad clase **gacha**: **6** puntos.
- Memoria dinámica: **4** puntos.
- Estilo de código: **20** puntos.

