Universidad de San Andrés

Trabajo Práctico I
Paradigmas de Programación

Lenguaje: C++

Estudiante: Inés Mestres

Año: 2025

Informe TP1

El Trabajo Práctico 1 fue una gran oportunidad para aplicar conceptos fundamentales de la Programación Orientada a Objetos (OOP), tales como clases abstractas, los conceptos de “herencia” y “composición”, y polimorfismo. Realicé tres ejercicios conectados entre sí, construyendo a través de pasos progresivos una red de clases que modelan un juego que dicta una batalla entre personajes (mágicos o guerreros) que utilizan armas (mágicas o de combate).

Ejercicio 1: Modelo de clases

El primer ejercicio pedía la implementación de dos interfaces llamadas “Arma” y “Personaje”. A partir de estas, construí una jerarquía de clases que representan armas y personajes de diferentes tipos en un juego. Este diseño incluye aplicaciones de herencia, composición y polimorfismo. Cada clase fue definida en un archivo Header (.hpp) e implementada en un archivo de código (.cpp) donde se desarrollan los métodos.

Archivo “Armas.hpp”:

En este archivo, declaré la interfaz “Arma”, que contiene únicamente métodos virtuales puros para cumplir con esta definición en C++. Estos métodos son: “get_nombre()”, “usar_arma()” y el destructor “~Arma()”. Además le incorporé un atributo protegido “nombre” y un constructor: “Arma(nombre)”. Aunque no es común armar un constructor de un interfaz o asignar atributos protegidos a un interfaz, es válido y me resultó útil poder inicializar un atributo “nombre” común para todas las subclases. El atributo es protegido ya que será utilizado, como mencioné, en las clases derivadas (no puede ser privado).

De la interfaz “Arma” derivan dos clases abstractas: “ItemMagico” y “armaDeCombate”. “ItemMagico” representa las armas mágicas (bastón, libro de hechizos, poción y amuleto), y “armaDeCombate” aquellas físicas (hacha simple, hacha doble, espada, lanza y garrote). Ambas clases heredan atributos de “Arma”, pero agregan los nuevos atributos protegidos. “ItemMagico” agrega los siguientes: elemento (tierra, aire, fuego, agua), rareza (si es común, raro, o muy raro), color, y la destrucción que genera en un personaje. Además, debe incorporar los respectivos métodos #getters” que devuelven los valores de los atributos; por ejemplo, get_nombre_itemMagico() que retorna el nombre del tipo de arma, como “Bastón”. Mientras, “armaDeCombate” agrega los atributos protegidos: material, rareza, destrucción y precisión, junto a los métodos “get” que permiten acceder a ellos. Finalmente, ambas clases redefinen el método “usar_arma()” y tienen destructores: “~ItemMagico()” y “~armaDeCombate()”

Las subclases concretas de estas clases abstractas definen tipos específicos de personajes. Las subclases de “ItemMagico” son: “baston”, “libroDeHechizos”, “pocion” y “amuleto”, que heredan de “ItemMagico” y definen un constructor con valores fijos para cada atributo. Cada subclase agrega dos atributos privados propios y sus respectivos getters. Asimismo, “armaDeCombate” también tiene clases derivadas que generan cada tipo concreto de arma de combate (ej: lanza). De nuevo, cada una inicializa los atributos generales y también agrega los dos específicos de su tipo de arma concreta. En todos los casos, se sobreescribe el método “usar_arma()” para imprimir tanto los atributos generales como los específicos de cada arma concreta.

La consigna pedía que incorporemos cinco atributos y métodos específicos de cada clase derivada concreta, pero solo pude incluir dos para cada una, lo cual representa una futura mejora. Por ejemplo la clase “baston” tiene atributos específicos a sí misma como: peso y largo, pero podría tener más como por ejemplo: “ancho”.

Archivo “Armas.cpp”:

Este archivo contiene las implementaciones de los métodos declarados en “Armas.hpp”. Como mencioné, organicé las funciones en su orden “jerárquico”, comenzando por aquellas de la interfaz “Arma”, y terminando en las pertenecientes a las subclases concretas.

Los métodos de la interfaz “Arma” son los primeros del archivo. El constructor se implementa como: “Arma::Arma(const std::string& nombre): nombre(nombre){}”. Con este constructor se inicializa el nombre del tipo de arma y el método “get_nombre()” retorna un string del valor del nombre.

Siguiente, están las implementaciones de los métodos de “ItemMagico”. Su constructor recibe los parámetros: nombre, elemento, nivel de rareza, color, y nivel de destrucción del arma mágica. Además, tiene métodos “getters” que retornan todos esos atributos. El método “usar_arma()” imprime un mensaje genérico por consola como aviso de que se está utilizando el arma (aunque esto nunca se verá porque en la práctica, este métodos es sobreescrito por las subclases). Finalmente se implementa el destructor como: “ItemMagico::~ItemMagico(){}” que permite destruir esta clase y sus derivadas.

La segunda clase derivada de “Arma” es “armaDeCombate” y también tiene implementaciones de funciones. Su constructor recibe: nombre, rareza, destrucción, y precisión. Luego, la clase tiene los métodos “getters” necesarios para retornar cada uno de estos atributos. Además, implementa la función “usar_arma()” que imprime sus datos en la consola, y el destructor: “~armaDeCombate()”.

Las subclases concretas de las armas mágicas como “amuleto” por ejemplo, implementan métodos adicionales que retornan los atributos específicos que se les añadió, como en el caso de “amuleto”: “get_tipo_piedra” y “get_poder”. Además, cada una de estas clases sobreescribe el “usar_arma()” que muestra por consola que se está utilizando esta arma. Este método imprime los atributos generales heredados de “ItemMagico” y aquellos propios y únicos (ej:tipo_piedra y poder). El acto de sobrescribir trata de polimorfismo, ya que se utiliza un método definido en la interfaz, y se personaliza las subclases según las características de las mismas. Las subclases concretas de arma de combate funcionan de manera igual; al agregar nuevos atributos a su tipo específico de arma deben implementar los métodos que devuelven estos atributos particulares (“getters”). También sobreescriben el “usar_arma()”.

Archivo Personaje.hpp:

La segunda interfaz creada fue “Personaje”, que define una estructura común para los personajes combatientes del juego. De esta interfaz, derivan dos clases abstractas: “mago” (define tipos de personajes mágicos como hechicero, conjurador, brujo y nigromante) y “Guerreros” (personajes guerreros como bárbaro, paladín, caballero, mercenario y gladiador).

La interfaz “Personaje” recibe a través de agregación elementos del interfaz “Arma”, ya que en el juego pido que personajes puedan portar armas. Uso el término de “agregación” a pesar de que la consigna pedía “composición” porque utilicé punteros smart “shared” en vez de “unique”, que es lo más común cuando se trata de “composición”. Opté por usar punteros “shared” porque me pareció más flexible y más fácil de entender. Esta opción me permitió utilizar instancias de armas sin necesidad de mover los punteros constantemente, facilitando el trabajo en general. A pesar de que no cumple esta regla de “composición” creo que mantiene una estructura similar y se vé un código claro y firme. Si bien algunos objetos pueden persistir aunque otros se eliminen, logré formar vínculos entre dichos objetos para seguir la lógica pedida.

La clase “Personaje” posee atributos protegidos: “tipo_personaje”, un vector de armas (representa composición entre personaje y armas) y vida_persona (la cantidad de vida que tiene el personaje). Nuevamente me pareció muy útil asignarle atributos para generalizar: todos los personajes deben tener un tipo, la posibilidad de utilizar armas, y una cantidad puntual de vida (que se modifica mientras combate con otros jugadores). Su constructor tiene forma: “Personaje(tipo_personaje, camtdad_vida)” y recibe el tipo de personaje y la cantidad de vida inicial, mientras que no se le comunica el vector ya que este se inicializa como vacío. Los métodos virtuales puros que implementa son variados: “get_tipo_personaje()”, “esta_presente()”, “get_vida()” (get), “agregar_arma(arma)”, “perder_vida(vida)”, “atacar_personaje(atacado)”, y “virtual ~Personaje() {}”, que son definidos, y luego implementados en el “.cpp”.

La primera clase abstracta que deriva de “Personaje” es “mago” que define todo tipo de combatiente mágico en el juego (hechicero, conjurador, brujo y nigromante). Recibe el atributo del tipo de personaje y de la cantidad de vida del personaje a través de herencia de la clase “Personaje” y además tiene los atributos protegidos: “nivel_experiencia_mago”, “especialidad_mago” y “energia_mago”. Luego, se incorporan los métodos; el constructor recibe todos los parámetros de los atributos mencionados y hay métodos “getters” que acceden a cada uno de los atributos. Además, tiene otros métodos: “esta_presente()”, “~mago()” los cuales se definen luego en el “.cpp”. Por otro lado, de “Personaje” también deriva la clase abstracta “Guerrero” que define a los combatientes guerreros en el juego (bárbaro, paladín caballero, mercenario y gladiador). También hereda el tipo de personaje y si cantidad de vida, y tiene los siguientes atributos protegidos: “nivel_experiencia_guerrero”, “cualidad_guerrero”, “energia_guerrero”. Sus métodos son: el constructor (recibe como parámetros todos los atributos), “esta_presente()”, y los “getters” de atributos: “get_nivel_experiencia_guerrero()”, “get_cualidad_guerrero()”, “get_energia_guerrero()” y el destructor “ ~guerrero()”, definidos en el “.cpp”.

Las subclases concretas de “mago” (“hechicero”, “conjurador”, “brujo” y “nigromante”) y de “Guerrero” reciben los atributos e incorporan dos atributos propios y los métodos para acceder a ellos. Además, redefinen el “usar_arma()” para que en cada caso, se impriman los datos personalizados a cada tipo de personaje, incluyendo a los atributos generales como a los específicos de la subclase.

Archivo Personaje.cpp:

El archivo implementa los métodos que fueron definidos en el archivo “Personajes.hpp”, pertenecientes a la interfaz Personaje y clases derivadas. Se forma el comportamiento de los personajes del juego.

Los métodos de la interfaz “Personaje” configuran la conducta de todos los personajes. Su constructor: “Personaje(tipo_personaje, cantidad_vida)” inicializa el personaje con sus atributos: tipo y vida. Luego, hay métodos virtuales puros como: “get_tipo_personaje() que retorna el tipo del personaje creado (ej: hechicero), “esta_presente()” imprime información del personaje para indicar que está presente, “get_vida()” retorna la cantidad de vida actual que tiene el personaje,, “agregar_arma(arma)” agrega un arma al vector de armas del personaje (que inicialmente tiene 0 armas), “perder_vida(cantidad_vidas)” le resta puntos de vida al personaje, y “atacar_personaje(atacado)” permite a un personaje atacar a otro. Finalmente, tenemos el destructor: “~Personaje() {}” que asegura una destrucción adecuada, teniendo en cuenta la jerarquía.

Las clases abstractas “mago” u “Guerrero” también están implementadas en este archivo. El constructor de “mago” recibe los atributos necesarios: aquellos heredados de “Personaje” como tipo de personaje y cantidad de vida, y nuevos atributos como nivel_experiencia_mago, especialidad_mago, y energía_mago. Sus métodos “getters”: “get_nivel_experiencia_mago()”, “get_especialidad_mago()”, y “get_energia_mago()” devuelven los valores. El método “esta_presente()” le informa al usuario a través de la consola, que el personaje pedido ha sido inicializado en el juego. Finalmente, se implementa un destructor: “mago::~mago() {}”. La segunda clase abstracta “Guerrero” tiene métodos similares a “mago”. El constructor también recibe los atributos necesarios de “Personaje” y propios. Tienen los métodos “gets”, que retornan lo pedido en su nombre: “get_nivel_experiencia_guerrero()”, “get_cualidad_guerrero()”, “get_energia_guerrero”. Finalmente, esta clase también tiene el método “esta_presente()” que realiza la misma función que en “mago”, y un destructor: “guerrero::~guerrero() {}”

Las subclases de “mago” (“hechicero”, “conjurador”, “brujo”, “nigromante”) y de “Guerrero” (“bárbaro”, “paladín”, “caballero”, “mercenario” y “gladiador”), tienen atributos específicos a ellas, implementan los “getters” para acceder a estos nuevos valores. Después, redefinen “esta_presente()” para imprimir toda su información.

Archivo “Ejercicio1-Main.cpp” :

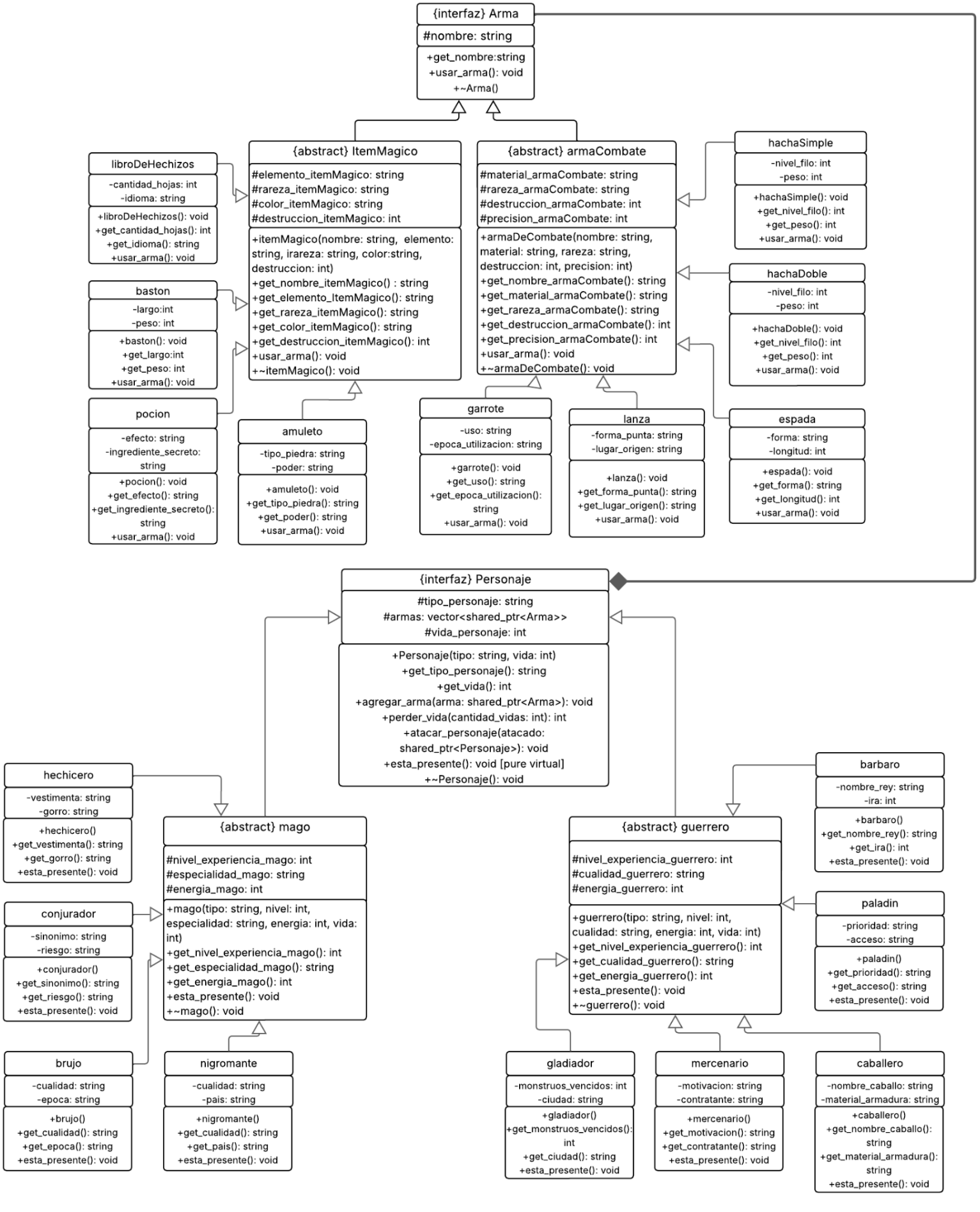
Este archivo contiene la función main(). Permite crear objetos e imprimir sus atributos para verificar el funcionamiento correcto del programa.

En el main, realicé lo siguiente: Creé dos armas, utilizando “std::make_shared<Arma>. Elegí de tipo de arma: el “amuleto” (arma mágica), y la “espada” (arma de combate). Luego, llamé al método “usar_arma()” para imprimir los atributos de ambas armas. Luego, creé dos personajes utilizando “make_shared<Personaje>. Un personaje es de tipo “nigromante” (mago), y el otro de tipo “caballero” (guerrero). Ejecuté un “esta_presente()” en ambos casos para que se impriman los datos de los personajes.

Así, verifiqué que funcione correctamente el programa requerido para completar el Ejercicio 1.

Cómo ejecutar el archivo main de este ejercicio:
Para compilación: “g++ Ejercicio1-Main.cpp Funciones-Arma.cpp Funciones-Personaje.cpp -o main -std=c++17”
Para correr el main: ./main

Este informe incluye un **Diagrama UML** de la estructura implementada en el Ejercicio 1 como una herramienta visual: visual:



Ejercicio 2: Generación Automática de Personajes con Armas

En este segundo ejercicio, el objetivo era automatizar la generación de personajes que portan armas. El objetivo era aprender a utilizar la función “std::rand()”, de la librería “cstdlib” y “ctime”. Para esto, generé dos números aleatorios entre 3 y 7, para indicar la cantidad de personajes de tipo Mago y la cantidad de tipo Guerreros que deberían de ser creados. Luego, volví a generar números aleatorios entre 0 y 2 que indican la cantidad de armas que porta cada personaje creado.

Para automatizar este proceso, creé una clase llamada “PersonajeFactory”, que contiene las funciones encargadas de crear: objetos de tipo personaje , objetos de tipo arma, y objetos de personajes armados (personajes que llevan armas). Esta clase está definida en un archivo Header (.hpp) e implementada en un archivo de código (.cpp).

Archivo “PersonajeFactory.hpp”:

Este archivo contiene las definiciones de la clase “PersonajeFactory”, que permite la creación dinámica de armas y personajes. Esta “fábrica” utiliza elementos de las clases “Arma” y “Personaje”, y los conecta con punteros inteligentes. La relación se llama “composición”: un personaje puede tener armas en su estructura. Todos los métodos de esta clase son estáticos; pertenecen a la clase, no a una instancia. Pueden ser llamados sin crear un objeto de la clase “PersonajeFactory”.

“PersonajeFactory” posee métodos privados estáticos que describen instancias específicas que podrían tener personajes y armas. Estos métodos se utilizan para crear cada tipo de personaje o arma concreta. Por ejemplo: dentro de los tipos de personajes posibles, uno consiste de crear un hechicero, y devuelve un puntero a un hechicero.

Los métodos se dividen en aquellos que crean las instancias de distintos tipos de personajes concretos y los que crean los tipos de armas concretas:

Métodos de creación de personajes concretos: “crear_hechicero()”, “crear_conjurador()”, “crear_brujo()”, “crear_nigromante()”, “crear_barbaro()”, “crear_paladin()”, “crear_caballero()”, “crear_mercenario()”, “crear_gladiador()”.

Métodos de creación de armas concretas: “crear_baston()”, “crear_libroDeHechizos()”, “crear_pocion()”, “crear_amuleto()”, “crear_hachaSimple()”, “crear_hachaDoble()”, “crear_espada()”, “crear_lanza()”, “crear_garrote()”.

Luego, “PersonajeFactory” tiene sus métodos públicos estáticos, que son aquellos se utilizan para crear los objetos concretos fuera de la clase en sí, y que devuelven punteros inteligentes a los objetos pedidos: “crear_personaje(tipo_personaje)”, “crear_arma(tipo_arma)”, “crear_personaje_armado(tipo_personaje, cantidad_personajes)”.

Archivo “PersonajeFactory.cpp”:

El archivo de código de “PersonajeFactory” implementa los métodos que fueron definidos en el header. Los métodos más importantes de la clase son los públicos mencionados.

El método “crear_personaje(tipo_personaje)” recibe una referencia a un string del tipo de personaje que se desea crear como parámetro, y retorna un puntero inteligente a ese personaje. Si le paso la palabra “caballero”, me devuelve un puntero inteligente a un caballero. Luego, “crear_arma(tipo_arma)” recibe como parámetro una referencia a un string del tipo de arma que el usuario quiere crear, y retorna un puntero inteligente a un arma del tipo especificado. Finalmente, el método “crear_personaje_armado(tipo_personaje, cantidad_armas)” recibe dos parámetros: una referencia a un string que especifica el tipo de personaje que se desea crear y la cantidad de armas que tiene dicho personaje. Genera el tipo de personaje descrito y le asigna armas de manera aleatoria.

Archivo “Ejercicio2-Main.cpp”:

Este archivo contiene la función main que permite verificar si la creación automática de objetos personajes, objetos armas y objetos de personajes armados funciona correctamente. Como pide el ejercicio, se utiliza el generador de números aleatorio “std::rand()” y se inicializa como: “std::rand(std::time(nullptr))”. Este inicializador asegura la variabilidad en cada instancia en la cual se ejecuta.

La cantidad de personajes magos y de personajes guerreros se decide con valores aleatorios entre 3 y 7. La cantidad de armas que posee cada personaje también se asigna de manera aleatoria con valores entre 0 y 2.

Como se pide la creación de magos y de guerreros por separado, creé dos vectores que separan los tipos concretos de cada grupo. De estos, se selecciona un tipo de personaje del vector de manera aleatoria y se crea el objeto buscado.

Cómo ejecutar el archivo main de este ejercicio:
Para compilación: “g++ Ejercicio2-Main.cpp Personaje-Factory.cpp ../Ejercicio1/Funciones-Arma.cpp ../Ejercicio1/Funciones-Personaje.cpp -I../Ejercicio1 -o main -std=c++17”
Para correr el main: “./main”

Ejercicio 3: Juego de Batalla

En este ejercicio programé una batalla entre dos personajes del que se arman utilizando el "PersonajeFactory" del ejercicio 2. La batalla utiliza la lógica de "piedra-papel-tijera" utilizando las opciones: "Golpe Fuerte", "Golpe Rápido" y "Defensa y Golpa".

Archivo "Funciones-JuegoBatalla.hpp":

Este archivo contiene la declaración de funciones utilizadas durante la batalla. Se utiliza para "avisarle" al compilador acerca de la existencia de estas funciones, para que puedan ser utilizadas en el main. Estas funciones son: "crear_personajes()", "describir_estado_personajes(jugador1, tipo_personaje1, cantidad_armas1, jugador2, tipo_personaje2, cantidad_armas2)","jugador1_eleccionAtaque()";"jugador2_ataqueRandom()", "combate_comparacionAtaques(jugador1, ataque1, jugador2, ataque2), y "resultados(jugador1)".

Archivo "Funciones-JuegoBatalla.hpp"

Este archivo contiene la implementación de las funciones declaradas en el header, y utilizadas en el juego, para simular la batalla. El método "crear_personajes()" genera los personajes aleatorios con su cantidad de armas "random". Luego, "describir_estado_personajes(jugador1, tipo_personaje1, cantidad_armas1, jugador2, tipo_personaje2, cantidad_armas2)" imprime por consola la información de ambos personajes en el formato: "El jugador 1 tiene XXX HP y el jugador 2 tiene YYY HP.". También, "jugador1_eleccionAtaque()" permite que el jugador 1 elija su ataque, mientras que "jugador2_ataqueRandom()" le asigna un ataque al jugador 2. Finalmente, "combate_comparacionAtaques(jugador1, ataque1, jugador2, ataque2) compara los ataques entre los jugadoresy le resta puntos al personaje que pierda, y "resultados(jugador1)" imprime el ganador del juego (del combate).

Archivo "Ejercicio3-Main.cpp":

La función main ejecuta la lógica del juego para poder simular una batalla entre los jugadores (creados de manera aleatoria, tanto su tipo de personaje, como la cantidad y los tipos de armas).

El main va a ejecutar el siguiente programa para replicar una lucha: Primero, llama la función "crear_personajes()" para crear los jugadores. Luego, el jugador q elige su ataque con la función "jugador1_eleccionAtaque()" y se genera el ataque del jugador 2 de manera aleatoria con la función "jugador2_ataqueRandom()". Se comparan los ataques mediante "combate_comparacionAtaques()". Mientras ninguno de los personajes pierde toda su vida, el juego continúa en un bucle y se ejecutan batallas. La función "resultados()" muestra el ganador.

Para ejecutar el código:
Compilar: "g++ Ejercicio3-Main.cpp Funciones-JuegoBatalla.cpp ../Ejercicio2/Personaje-Factory.cpp ../Ejercicio1/Funciones-Arma.cpp ../Ejercicio1/Funciones-Personaje.cpp -I../Ejercicio1 -I../Ejercicio2 -o main -std=c++17"
Correr el main: "./main"