# Tecnológico de Monterrey

Evidencia 1 | Proyecto Integrador


**Participantes:**

Miguel Ángel Monroy Posada - A01739891

**Profesor:**

Leon Felipe Guevara Chavez


**Materia:**

Programacion Orientada a Objetos

**Campus:**

CNAL

**Grupo:**

851

INTRODUCCION

Se presenta una situación problema de forma simulada, en la que el programador debe de hacer bajo C++ una batalla simulada, en ella se encuentran personajes, como elfos, magos, guerreros donde cada uno en su simulación se podrá ver el ataque de cada uno conforme a los atributos, siendo este una cercanía a videojuegos tipo RPG o también llamado juego de turnos, en los cuales consisten en batallas de personajes pero en turnos que cada uno atacara al contrincante hasta que todos los personajes de un lado queden derrotados, marcando el fin del contrincante.

Durante este proyecto se plasma la creatividad al hacer un código versión videojuego, poniéndonos el reto de que debería de tener estos personajes, como deberían de comportarse y cual seria nuestro lore, o mas bien dicho, los nombres y rasgos de dicha persona, en este proceso se tenia la pequeña ventaja donde teníamos un poco mas de conocimiento acerca de esto dado que en esta época lo mas moderno es jugar videojuegos, del cual proviene esta situación problema.

Se ocuparon conceptos como polimorfismo, sobrecarga, excepciones, vectores, entre muchas otras cosas, pero sobre todo se ocupo la herencia, dado que estos personajes tendrían características que los hacen ser lo que son, pero tendrán cosas en común uno sobre otro. Todos estos conceptos fueron vistos dentro de clase y reforzados tanto en ejercicios, como en quizzes y en libros o paginas web relacionado a los temas para estar mas comprendidos, todo con el objetivo de poder aplicar correctamente en nuestro proyecto de código al igual que en el examen, además de ello, se intenta respetar las reglas básicas de la programación orientada a objetos las cuales son el encapsulamiento, herencia, abstracción y polimorfismo.

USO DE LOS CONCEPTOS

Antes comenzar a explicar los conceptos que se ocuparon dentro del proyecto quisiera aclarar un par de cosas.

Comenzando con la inspiración para poner todos estos atributos de cada uno, aunque un gran par del código es base de las recomendaciones y criterios que pidió el profesor, como el atacar de mago que podría modificar los estados de los demás personajes o tener dos atributos de salud y vida, una que se modifica independiente de la otra, mientras que el otro solo funciona como una forma de imprimir la barra. La otra gran parte de código fue inspirado en un juego en especifico llamado “Expedition 33”, un videojuego RPG con mecánicas avanzadas como un contrataque, objetos que puedan revivir a los compañeros, curarlos, además de tener un medidor de energía que hace que puedan tener la posibilidad de hacer un combo de ataques y genere más daño. Una breve historia acerca de como me inspire fue que durante 2do semestre este juego salió y decidí jugarlo después de ver un par de reseñas, aunque al inicio parecía aburrido dado que no sabia mucho acerca de RPG, yo soy mas de FPS, la historia que tenían además de las peleas y sus mecánicas eran interesantes, actualmente ya tengo mas de 100 horas jugadas, ya pasado 3 veces el juego y sigo descubriendo nuevas mecánicas que no sabía desde un inicio.

Dicho juego me hizo inspirarme un poco en lo que podría yo colocar en mi programa, por ejemplo, aunque se pedía como requisito intente darle un poco mas de peso al crítico, siendo este no solo un modificador temporal que puede o no pasar, sino también un atributo general que tendrían como probabilidad cada personaje, también la experiencia fue un atributo inspirado del videojuego, siendo este un punto importante para ver un poco mas de desequilibrio entre los personajes. Los atributos de que se encuentran en sus clases derivadas (mago, elfo, guerrero) fueron inspiradas de la pelicular “El señor de los anillos” y “El hobbit” creo que esto se le podría imaginar a cualquiera porque los tres tipos de combatientes tienen todo referente a poder apoyarse con estas películas que son casi cultura general, como Star Wars, DC o Marvel. Dicho todo esto, comenzare explicando en que son y donde explican estos conceptos que vimos en clase además del porque hacerlo de esa manera.

- Herencia

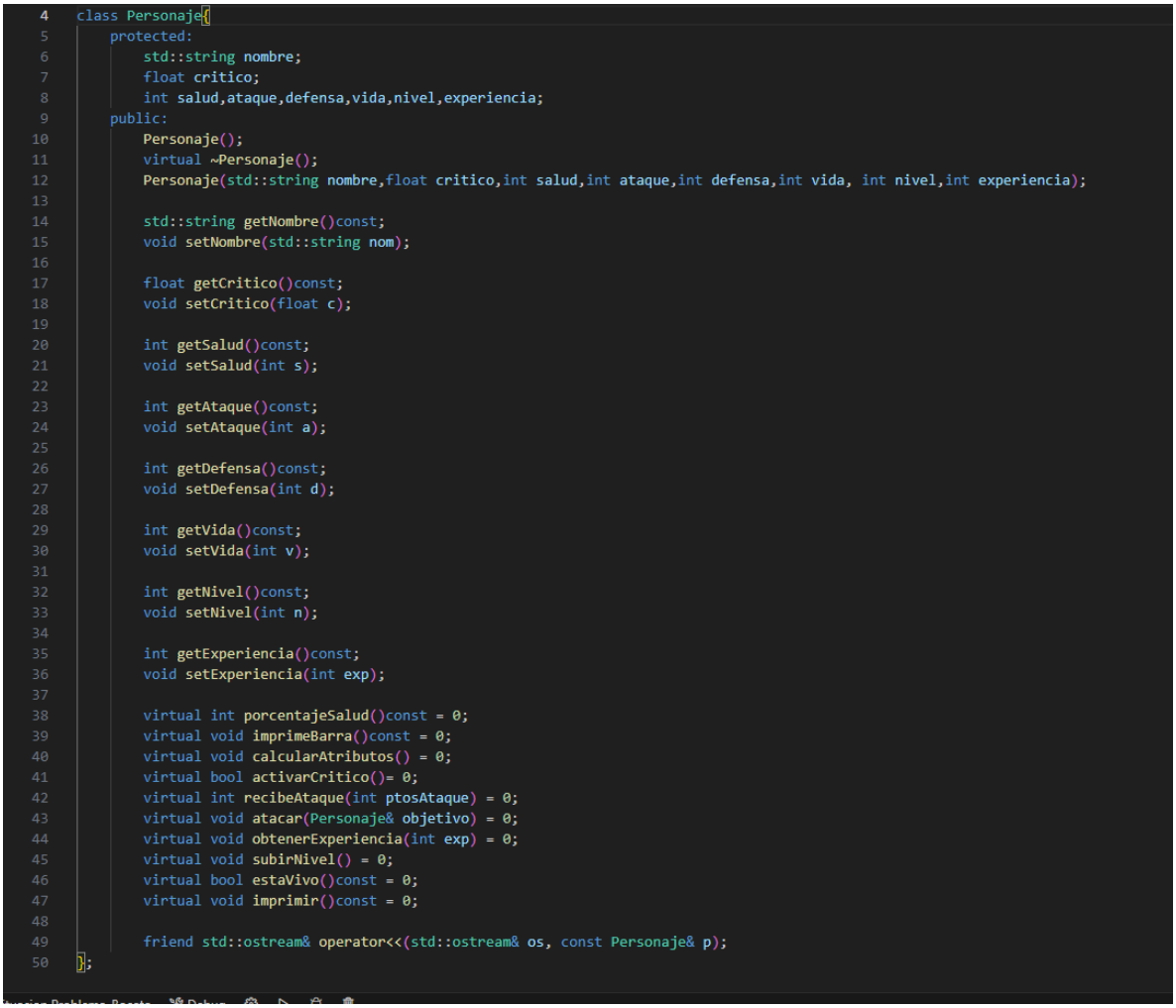
La herencia es uno de los pilares fundamentales de C++, permite crear una clase, la cual será llamada clase base, esta permite heredar ciertos atributos a demás clases, estas llamadas clases derivadas, sirve para reutilizar código y organizar jerarquías lógicas. Se pueden ocupar 3 tipo de herencia:

- Public: Los miembros públicos conservan su visibilidad

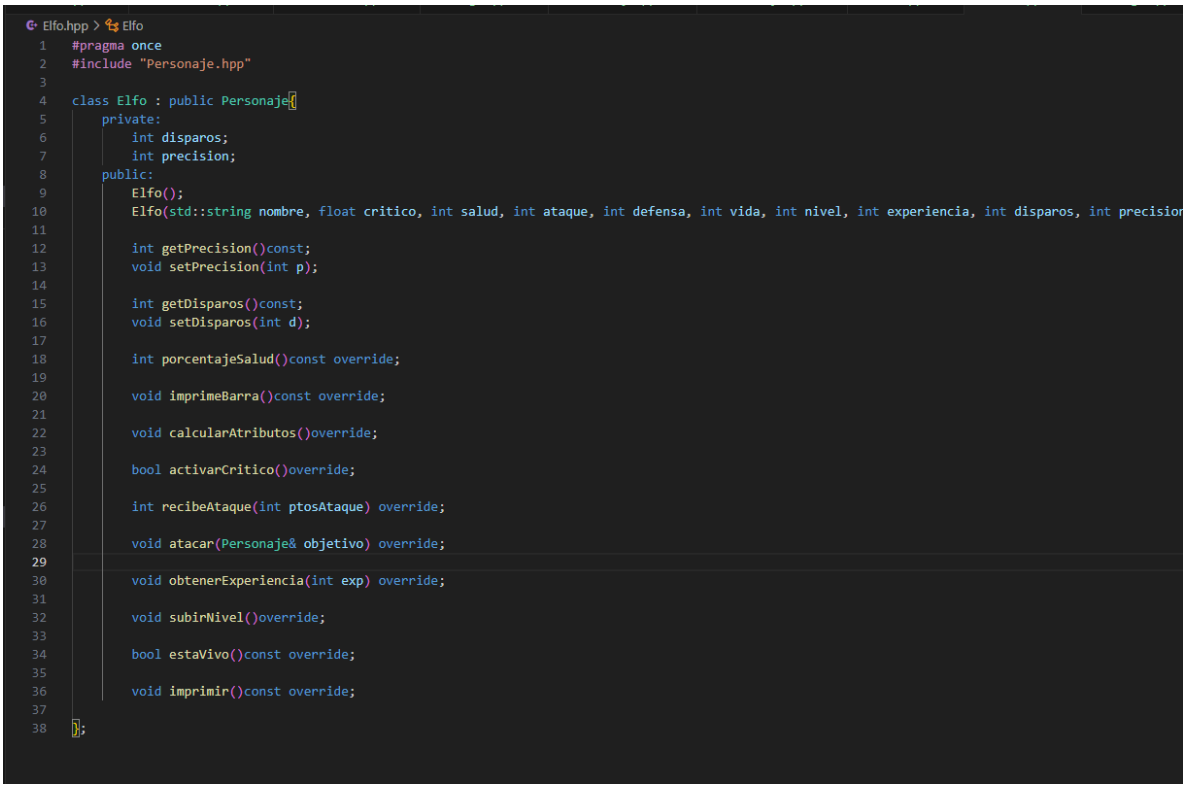
- Protected: Los miembros públicos están protegidos para la clase hija

- Private: Se vuelven privados los miembros, siendo no accesibles a menos que sean dentro de la misma clase.

Aquí tenemos un ejemplo de herencia dentro de nuestro código del proyecto:



```
class Personaje{
    protected:
        std::string nombre;
        float critico;
        int salud,ataque,defensa,vida,nivel,experiencia;
    public:
        Personaje();
        virtual ~Personaje();
        Personaje(std::string nombre,float critico,int salud,int ataque,int defensa,int vida, int nivel,int experiencia);

        std::string getNombre()const;
        void setNombre(std::string nom);

        float getCritico()const;
        void setCritico(float c);

        int getSalud()const;
        void setSalud(int s);

        int getAtaque()const;
        void setAtaque(int a);

        int getDefensa()const;
        void setDefensa(int d);

        int getVida()const;
        void setVida(int v);

        int getNivel()const;
        void setNivel(int n);

        int getExperiencia()const;
        void setExperiencia(int exp);

        virtual int porcentajeSalud()const = 0;
        virtual void imprimeBarra()const = 0;
        virtual void calcularAtributos() = 0;
        virtual bool activarCritico()= 0;
        virtual int recibeAtaque(int ptosAtaque) = 0;
        virtual void atacar(Personaje& objetivo) = 0;
        virtual void obtenerExperiencia(int exp) = 0;
        virtual void subirNivel() = 0;
        virtual bool estaVivo()const = 0;
        virtual void imprimir()const = 0;

        friend std::ostream& operator<<(std::ostream& os, const Personaje& p);
};
```




Aquí tenemos dos imágenes donde la primera es nuestra clase base, tenemos atributos generales como la vida, el ataque, los niveles, el nombre, etc. Si bien nuestra clase Personaje tiene lo que es una persona normal, nuestro objetivo es crear personajes que sean característicos por destacar en ciertos aspectos. Por ejemplo, podemos ver que en Elfo contiene todo lo que tiene la clase Personaje, pero tiene tres atributos extras, las cuales son velocidad, disparos y precisión, características muy destacadas para los elfos dado que los conocemos más como excelentes arqueros.

Cabe aclarar que tenemos que elegir aquí nuestra protección de atributos, se declara como protected los datos de personaje dado que recibirán modificaciones, no necesariamente todos deben de ser protected, solo los que se en clases hijas recibirán algún cambio, los demás se pueden mantener como private, aunque otra solución es colocar todo como private y llamar los atributos por medio de los getters.

- Polimorfismo

El polimorfismo permite que un puntero o referencia de la clase base, puede ocuparse los métodos a que se comportan de manera diferente según la clase derivada.

En nuestra clase base(Personaje) se tiene un ejemplo de polimorfismo, estos están declarados como virtual, estos al ser declarados como virtual se debe de declarar su comportamiento cuando se hable conforme al tipo de combatiente.

```
virtual int porcentajeSalud()const = 0;
virtual void imprimeBarra()const = 0;
virtual void calcularAtributos() = 0;
virtual bool activarCritico()= 0;
virtual int recibeAtaque(int ptosAtaque) = 0;
virtual void atacar(Personaje& objetivo) = 0;
virtual void obtenerExperiencia(int exp) = 0;
virtual void subirNivel() = 0;
virtual bool estaVivo()const = 0;
virtual void imprimir()const = 0;
```

Esta forma esta justificada dado que tenemos diferentes modificadores para los personajes de cada uno, ejemplo, para imprimir cada uno mostrara la mayoría de los mismo atributos, algunos a lo mejor cambiando en ciertas cosas como ataque, defensa, nivel, pero también mostrando sus atributos característicos como ejemplo en mago se mostrara el elemento y el mana

```
void Mago::imprimir() const {
    cout << "Nombre: " << getNombre() << endl;
    cout << "Nivel: " << getNivel() << endl;
    cout << "Salud: " << getSalud() << "/" << vida << endl;
    cout << "Ataque: " << getAtaque() << endl;
    cout << "Defensa: " << getDefensa() << endl;
    cout << "Porcentaje de crítico: " << getCritico() << endl;
    cout << "Experiencia: " << getExperiencia() << endl;
    cout << "Maná: " << mana << endl;
    cout << "Elemento: " << elemento << endl;
    cout << "Hechizo: " << hechizo << endl;
    cout << "Porcentaje de salud: ";
    imprimeBarra();
    cout << "Estado actual: " << (estaVivo() ? "Vivo" : "Muerto") << endl;
}
```

Claro esta que para esto no solo se necesita poner virtual y ya, también se necesita una declaración en nuestro .hpp como override, esto hace que podamos sobre escribir el método antes establecido sin tener ningún tipo de problema

```
28          int porcentajeSalud()const override;
29          void imprimeBarra()const override;
30          void calcularAtributos()override;
31          bool activarCritico()override;
32          int recibeAtaque(int ptosAtaque) override;
33          void atacar(Personaje& objetivo) override;
34          void obtenerExperiencia(int exp) override;
35          void subirNivel()override;
36          bool estaVivo()const override;
37          void imprimir()const override;
```

Existen muchos mas ejemplos tanto de mago como de los demás combatientes que cambian en esta sección, dado que tienen atributos extras diferentes a los demás, como otras formas de activar el critico en cada uno, como la forma de recibir el ataque o incluso atacar. Otro claro ejemplo de esta diferencia es en la sección de recibirAtaque y atacar.

En nuestras clases mago y guerrero tenemos la gran diferencia en que mago tiene un comportamiento diferente conforme al atacar basado en recibir el crítico, aunque tengan mucha similitud dado que ambos atacan la clase mago tiene una opción de crear una afectación a guerrero para debilitarlo y en guerrero tenemos un comportamiento totalmente diferente a recibirAtaque, esto debido a que tiene una armadura la cual aumenta su defensa, siendo este un receptor de menor

daño a comparación de los otros dos combatientes que solo se basan en su defensa inicial. Este es uno de los tantos ejemplos que hay en conforme a los cambios de comportamiento debido a los diferentes tipos de combatientes, obviamente algunos tendrán muchas en común como la opción de estarVivo dado que siempre se evaluaría con la vida, u otras como el revivir que solo están en algunos combatientes debido a sus atributos.

```cpp
void Mago::atacar(Personaje& objetivo) {
    if (objetivo.getSalud() <= 0) {
        cout << objetivo.getNombre() << " ya está derrotado. No puedes atacarlo." << endl;
        return;
    }

    int damage = ataque;

    if (activarCritico()) {
        damage = static_cast<int>(ataque * 1.5);
        cout << "Se realizó un ataque crítico." << endl;
    }

    bool estabaVivo = objetivo.getSalud() > 0;

    int danoReal = objetivo.recibeAtaque(damage);

    Guerrero* ptrGuerrero = dynamic_cast<Guerrero*>(&objetivo);
    if (ptrGuerrero != nullptr) {
        if (mana >= 40) {
            ptrGuerrero->debilitarPorMagia(mana);
        } else {
            cout << nombre << " no tiene suficiente maná para debilitar a " << ptrGuerrero->getNombre() << "." << endl;
        }
    }

    cout << nombre << " ataca a " << objetivo.getNombre() << " con " << damage << " puntos de ataque." << endl;
    cout << objetivo.getNombre() << " recibe " << danoReal << " puntos de daño." << endl;

    obtenerExperiencia(25);

    if (estabaVivo && objetivo.getSalud() <= 0) {
        cout << objetivo.getNombre() << " ha sido derrotado." << endl;
        obtenerExperiencia(75);
    }

    generarMana(); // Para que genere mas mana despues de atacar
}
```

```cpp
int Guerrero::recibeAtaque(int ptosAtaque) {
    int danio = ptosAtaque - defensa;
    if (danio < 0) danio = 0;
    salud -= danio;
    if (salud < 0) salud = 0;
    return danio;
}
```

- Clases abstracta

De la mano del polimorfismo tenemos lo que son las clases abstractas, pero para que sirven y como se declaran. Bueno, como es mostrado en una foto anterior, las clases abstractas son cuando se declaran por virtual al método que queramos para que podamos modificarlo en las clases hijas, también existe la clase abstracta pura donde ahora no se puede crear una instancia directa, antes se podían declarar como abstracta para ser modificada y además en la clase base tener una instancia, ahora ya no, solo se puede colocar en clases hijas de forma permanente.

```
private:
    std::vector<Personaje*> ejercito1;
    std::vector<Personaje*> ejercito2;
```

En esta parte aunque no estamos creando aun ningún objeto, se esta ocupando el polimorfismo en base a clases abstractas.

- Sobrecarga de operadores

Es una forma que nos permite definir o personalizar el comportamiento de ciertos operadores cuando los usas con objetos de la clase, en este código se ocupa en la clase Personaje

```
ostream& operator <<(ostream& os, const Personaje& p){
    p.imprimir();
    return os;
```

Esta sobrecarga que tenemos en nuestro Personaje.cpp nos ayuda a poder llamarlo e imprimir los caracteres de cada personaje, es decir sus atributos que les hemos colocado.

```
void Combate::mostrarEjercito(const vector<Personaje*>& ejercito) const {
    for (auto p : ejercito) {
        cout << *p << endl;
    }
}
```

- Excepciones

Como ultimo tenemos las excepciones, que son un mecanismo preventivo donde podemos teorizar que pueda haber un error, podríamos decir que son errores a propósito. No hablamos de un error de compilación específicamente, pueden ser errores como decir que tiene que escribir del 1-5 pero el escribe 6, ahí hay un error y se puede ocupar las excepciones ahí y decir que no esta dentro del rango

```
Personaje* Combate::crearPersonajeDatos(const string& tipo, const vector<string>& d) {
    try {
        if (tipo == "Elfo") {
            return new Elfo(d[0], stof(d[1]), stoi(d[2]), stoi(d[3]), stoi(d[4]),
                            stoi(d[5]), stoi(d[6]), stoi(d[7]), stoi(d[8]), stoi(d[9]));
        } else if (tipo == "Guerrero") {
            return new Guerrero(d[0], stof(d[1]), stoi(d[2]), stoi(d[3]), stoi(d[4]),
                                stoi(d[5]), stoi(d[6]), stoi(d[7]), stoi(d[8]), stoi(d[9]), stoi(d[10]));
        } else if (tipo == "Mago") {
            return new Mago(d[0], stof(d[1]), stoi(d[2]), stoi(d[3]), stoi(d[4]),
                            stoi(d[5]), stoi(d[6]), stoi(d[7]), stoi(d[8]), d[9], stoi(d[10]));
        }
    } catch (const exception& e) {
        cerr << "Error al crear personaje (" << tipo << "): " << e.what() << endl;
    }
    return nullptr;
}
```

Aquí tenemos en el código una excepción que se puede denotar gracias al try que tenemos arriba, en este caso lo coloque como una protección a falta de datos en el txt, es decir tiene que cumplir con rellenar los datos de cada personaje, obviamente tienen que ser en el orden que tendremos nuestro constructor, de esa parte es de donde evaluara el código si esta correcto, donde hay un string deberá de haber un string, al igual que zonas donde haya int y float debe de tener esos mismo caracteres para que no arroje un error al crear personaje.

# CONCLUSION

La verdad es que el proyecto fue algo bonito de hacer, al final se consistía en un videojuego y creo que lo hermoso de esto fue la creatividad que tuve durante el desarrollo de esto, al principio no sabia en que basarme, solo tenia un par de ideas basado en lo que nos recomendaba cada tarea, pero conforme al tiempo empezaron a llegar y creo que eso fue bastante bueno, posiblemente este proyecto hubiera sido algo un poco mas burdo de no ser que se me ocurrieron ideas como estas de querer colocar nuevas cosas como la experiencia que suba con el nivel abarcar un poco mas la zona de daño siendo modificado por lo críticos y muchas cosas más creo que abría sido aburrido presentarlo e incluso explicarlo.

Creo que este proyecto ayudo a ser un poco mas creativo, normalmente soy alguien un poco lineal al momento de hablar sobre temas de la escuela, no intento desviarme tanto de lo que pide uno para estar completamente dentro de lo que se pide en la tarea, pero creo que ahora mismo tanto en la rubrica menciona que seamos algo creativo como yo también tuve el tiempo para pensar en nuevas formas y tuve algo de que inspirar mis ideas que ayudaron a que volvería esa imaginación que antes tenia cuando era chico y pensaba en historias de película cuando agarraba mis juguetes ya sea desde un simple carro de hot wheels hasta un muñeco de max Steel e incluso dibujos que hacia antes.

Para finalizar quiero agradecer por la oportunidad de hacer un tipo de proyecto así de creativo, normalmente las situaciones problema son algo mas complejas, enfocándonos en el sentido de programación, por ejemplo cosas como haz específicamente un programa que sea para plataformas de videos, que no tengan ninguna cosa mas porque no es necesario, lo cual por algo programación es programación, es algo donde debes de imaginar que hacer y no seguir siempre una estructura lineal donde no se pueda hacer nada innovador y este proyecto hace algo diferente porque si tiene su rubrica y todo pero deja ser creativo en ciertos aspectos lo cual ayuda a uno no aburrirse a hacer siempre lo mismo y pues como había dicho, se reanima dicho espíritu que de alguna forma de apaga que es nuestra pequeña creatividad.

REFERENCIAS

*W3Schools.com*. (s. f.). https://www.w3schools.com/cpp