Unnecessary violence

Explicación del ejercicio

El objetivo de este ejercicio es implementar un conjunto de clases en C++ que modelan armas y personajes que pueden usarlas para atacar. Este ejercicio está diseñado para enseñar conceptos fundamentales de programación orientada a objetos, como clases, encapsulación, referencias, punteros y la relación entre objetos.

Detalles del ejercicio

1. Clase Weapon:
    a. Debe tener un atributo privado type, que es un std::string.
    b. Implementa las siguientes funciones miembro:
        i. getType(): Devuelve una referencia constante al atributo type.
        ii. setType(): Recibe un nuevo valor y actualiza el atributo type.

2. Clases HumanA y HumanB:
    a. Ambos representan personajes que pueden atacar con un arma.
    b. Ambos tienen un atributo name (nombre) y un arma (Weapon).
    c. Ambos tienen una función miembro attack() que imprime en pantalla: <name> attacks with their <weapon type>

3. Diferencias entre HumanA y HumanB:
    a. HumanA recibe el arma en su constructor, por lo que siempre tiene un arma asignada.
    b. HumanB no recibe el arma en su constructor, por lo que puede no tener un arma inicialmente. Sin embargo, se le puede asignar un arma más adelante mediante un método específico.

4. Caso de prueba:
    a. Se proporciona un código de ejemplo que verifica la implementación.
    b. Se comprueba que, al cambiar el tipo de un arma compartida, el cambio se refleje en las acciones de los personajes.

5. Requisito adicional:
    a. Reflexiona sobre cuándo usar referencias (Weapon&) y punteros (Weapon*) para el

manejo del arma. Esto está relacionado con la gestión de recursos y el diseño de clases.

Cómo afrontar el ejercicio

1. Dividir el problema en partes:
    a. Primero implementa y prueba la clase Weapon.
    b. Luego, crea las clases HumanA y HumanB una a una, asegurándote de comprender y aplicar las diferencias entre ambas.
2. Usar referencias y punteros adecuadamente:
    a. HumanA siempre tendrá un arma, por lo que puedes usar una referencia constante al objeto Weapon. Esto asegura que no pueda ser nulo y evita errores en tiempo de ejecución.
    b. HumanB puede no tener un arma inicialmente. Por lo tanto, un puntero (Weapon*) es apropiado, ya que permite asignar un arma opcionalmente.
3. Pruebas incrementales:
    a. Escribe pequeños programas de prueba para verificar el comportamiento de cada clase y función antes de pasar al siguiente paso.
4. Gestión de memoria:
    a. Aunque este ejercicio no usa memoria dinámica explícita, siempre es buena práctica verificar fugas de memoria usando herramientas como valgrind.
5. Reflexión y comentarios:
    a. Piensa en las ventajas de cada enfoque (referencia vs puntero) y documenta tus decisiones en los comentarios de tu código.

En resumen

El ejercicio consiste en implementar una clase Weapon y dos clases de personajes (HumanA y HumanB) con diferentes enfoques para manejar armas. La clave es comprender cómo las referencias y los punteros afectan la relación entre objetos y cómo reflejar cambios dinámicos en los atributos. Divide el trabajo en partes pequeñas, prueba a medida que avanzas y asegúrate de reflexionar sobre las decisiones de diseño al finalizar.

Uso de referencias (Weapon&) y punteros (Weapon*) en el estándar C++98

1. Cuándo usar referencias (Weapon&)

Las referencias son útiles cuando se garantiza que siempre habrá un objeto válido asociado. No pueden ser nulas y, una vez inicializadas, no pueden cambiar para referirse a otro objeto. Esto

las hace ideales para casos en los que la asociación entre objetos es obligatoria.

En el ejercicio:

- En la clase HumanA, el arma es obligatoria y se pasa por el constructor. Esto asegura que siempre habrá un arma asociada al objeto. Por lo tanto, es apropiado usar una referencia para Weapon.

Ejemplo de uso con referencias:

```
class HumanA {
private:
  std::string name;
  Weapon& weapon; // Referencia al arma, siempre válida.

public:
  HumanA(const std::string& name, Weapon& weapon) : name(name),
weapon(weapon) {}

  void attack() const {
    std::cout << name << " attacks with their " << weapon.getType() << std::endl;
  }
};
```

- La referencia weapon está garantizada por el constructor, lo que significa que HumanA siempre tendrá un arma válida y no requiere verificar su existencia antes de usarla.

2. Cuándo usar punteros (Weapon*)

Los punteros son útiles cuando:

- El objeto asociado es opcional.
- Puede haber momentos en los que el objeto no esté disponible o sea necesario cambiar dinámicamente la asociación durante la vida útil del objeto.

En el ejercicio:

- En la clase HumanB, el arma no se proporciona en el constructor y es opcional. Por lo tanto, se utiliza un puntero para permitir que el arma pueda no estar presente inicialmente y ser asignada más tarde.

Ejemplo de uso con punteros:

```
class HumanB {
private:
  std::string name;
  Weapon* weapon; // Puntero al arma, puede ser nulo.
```

```
public:
  HumanB(const std::string& name) : name(name), weapon(NULL) {}

  void setWeapon(Weapon& weapon) {
    this->weapon = &weapon; // Asignar un arma.
  }

  void attack() const {
    if (weapon) { // Verificar si el puntero no es nulo.
      std::cout << name << " attacks with their " << weapon->getType() << std::endl;
    } else {
      std::cout << name << " has no weapon to attack with!" << std::endl;
    }
  }
};
```

- En este caso, el puntero weapon puede ser NULL (nulo), y se debe verificar antes de usarlo en la función attack() para evitar errores.

Comparación directa con el ejemplo del ejercicio

Referencias (Weapon&) en HumanA:

- Ventaja: Garantiza que siempre haya un arma válida.
- Uso: Adecuado para HumanA, ya que el arma se pasa en el constructor y es obligatoria.
- Implementación destacada: Weapon club = Weapon("crude spiked club");
  HumanA bob("Bob", club); // Se pasa el arma al constructor.
  bob.attack();

Punteros (Weapon) en HumanB:*

- Ventaja: Permite la ausencia de un arma o cambiar el arma asignada dinámicamente.
- Uso: Adecuado para HumanB, ya que el arma no está garantizada en el constructor.
- Implementación destacada: Weapon club = Weapon("crude spiked club");
  HumanB jim("Jim");
  jim.setWeapon(club); // Asignar un arma más tarde.
  jim.attack();

Resumen práctico

Caso	Usar referencia (Weapon&)	Usar puntero (Weapon*)
Objeto obligatorio	Sí	No
Objeto opcional	No	Sí
Reasignación durante la vida	No	Sí
Comprobación de validez	No necesaria	Necesaria (if (ptr != NULL))

En este ejercicio:

- Usa referencias en HumanA porque siempre tiene un arma.
- Usa punteros en HumanB porque el arma es opcional y puede asignarse dinámicamente.

HumanA.hpp & HumanB.hpp

Explicación detallada del código

El código presentado define dos clases, HumanA y HumanB, que representan personajes que pueden atacar con un arma (Weapon). Estas clases tienen diferencias en cómo gestionan las armas, de acuerdo con lo especificado en el ejercicio. El estándar utilizado es C++98, por lo que ciertos conceptos modernos, como constructores delegados o inicialización con llaves, no están disponibles.

1. Clase HumanA

```
#ifndef HUMANA_HPP
# define HUMANA_HPP

#include "Weapon.hpp"

class HumanA {

	public:

		HumanA(std::string name, Weapon &weapon);
		~HumanA(void);
		void	attack(void);

	private:

		std::string	_name;
		Weapon		_weapon;

};

#endif
```

Análisis del código:

1. Directivas de preprocesador:
    a. La macro #ifndef evita múltiples inclusiones del mismo archivo durante la compilación.
    b. #define HUMANA_HPP define la macro de protección para este archivo de encabezado.
2. Constructor:
    a. HumanA(std::string name, Weapon &weapon):
        i. Recibe el nombre del personaje (std::string name) y una referencia a un objeto Weapon.
        ii. La referencia asegura que el personaje siempre tendrá un arma válida y no necesita usar punteros.
3. Destructor:
    a. ~HumanA(void):
        i. Declara un destructor, aunque no realiza ninguna acción explícita (probablemente sea necesario solo para el diseño de la clase o como requisito).
4. Método attack:
    a. void attack(void):
        i. Declaración de un método que probablemente imprime un mensaje como: <name> attacks with their <weapon type>

5. Miembros privados:
    a. _name: Almacena el nombre del personaje como una cadena.
    b. _weapon: Almacena una referencia al arma del personaje. Al ser una referencia, HumanA siempre tendrá un arma válida y no necesita gestión dinámica.

2. Clase HumanB

```
#ifndef HUMANB_HPP
# define HUMANB_HPP

#include "Weapon.hpp"

class HumanB {

        public:

                HumanB(std::string name);
                ~HumanB(void);
                void    attack(void);
                void    setWeapon(Weapon &weapon);

        private:

                std::string     _name;
                Weapon          *_weapon;

};
```

```
#endif
```

Análisis del código:

1. Directivas de preprocesador:
    a. Similar a las de HumanA, previenen múltiples inclusiones del archivo.
2. Constructor:
    a. HumanB(std::string name):
        i. Solo recibe el nombre del personaje.
        ii. No toma un arma inicialmente, lo que permite que el personaje no tenga arma al inicio.
3. Destructor:
    a. ~HumanB(void):
        i. Similar al de HumanA, está definido pero no realiza ninguna acción específica.
4. Método attack:
    a. void attack(void):
        i. Imprime un mensaje similar al de HumanA.
        ii. Debe manejar el caso en el que _weapon sea nullptr, ya que HumanB puede no tener arma asignada.
5. Método setWeapon:
    a. void setWeapon(Weapon &weapon):
        i. Permite asignar un arma al personaje. Usa una referencia (Weapon &) para evitar copias y asegurar que el arma existe.
6. Miembros privados:
    a. _name: Almacena el nombre del personaje como una cadena.
    b. _weapon: Es un puntero a un objeto Weapon. Por defecto, puede ser nullptr, lo que

indica que HumanB no tiene un arma asignada al inicio.

Diferencias principales entre HumanA y HumanB

Aspecto	HumanA	HumanB
Arma	Siempre tiene un arma desde su creación.	Puede empezar sin arma.
Tipo de arma	Referencia constante a Weapon.	Puntero a Weapon (puede ser nullptr).
Gestión del arma	No necesita métodos adicionales.	Usa setWeapon para asignar un arma.

En resumen:

- HumanA representa un personaje que siempre tiene un arma válida, y esta se pasa como referencia en el constructor.
- HumanB puede empezar sin arma y gestiona su arma mediante un puntero que puede ser nulo. Este diseño permite mayor flexibilidad.
- El uso de referencias en HumanA asegura la validez del arma, mientras que el uso de punteros en HumanB permite manejar la ausencia de arma inicial.

HumanA.cpp & HumanB.cpp

A continuación, se explica en detalle el código proporcionado para las clases HumanA y HumanB en base al estándar C++98.

Clase HumanA

```
#include "HumanA.hpp"

HumanA::HumanA(std::string name, Weapon &weapon) : _name(name),
_weapon(weapon) {
   return ;
}

HumanA::~HumanA(){
   return ;
}

void HumanA::attack(void){
   std::cout << this->_name << " attacks with his " << this->_weapon.getType() <<
std::endl;
}
```

1. Constructor HumanA::HumanA

```
HumanA::HumanA(std::string name, Weapon &weapon) : _name(name), _weapon(weapon) {
   return ;
}
```

- Este constructor recibe dos parámetros: un std::string llamado name y una referencia a un objeto Weapon llamado weapon.
- La inicialización de los atributos _name y _weapon se realiza a través de la lista de inicialización del constructor:

    - _name(name) asigna el valor de name al atributo privado _name.
    - _weapon(weapon) asigna la referencia weapon al atributo privado _weapon. Aquí, se usa una referencia porque se espera que el objeto Weapon ya exista antes de la creación de HumanA, y se mantiene una referencia directa a ese objeto.

  La palabra clave return ; es innecesaria en este caso, ya que no se está realizando ninguna acción explícita en el cuerpo del constructor, pero es legal en C++98.

2. Destructor HumanA::~HumanA

```
HumanA::~HumanA(){
  return ;
}
```

- El destructor no realiza ninguna operación especial. En este caso, el destructor vacío indica que no es necesario realizar ninguna limpieza de recursos (ya que no se usa memoria dinámica explícita).
- La palabra clave return ; aquí también es innecesaria, ya que el destructor no necesita retornar ningún valor. Sin embargo, C++98 permite esta sintaxis.

3. Método HumanA::attack

```
void HumanA::attack(void){
  std::cout << this->_name << " attacks with his " << this->_weapon.getType() << std::endl;
}
```

- El método attack imprime en pantalla un mensaje indicando que el personaje (this->_name) ataca con su arma.
- Se accede a los atributos _name y _weapon del objeto actual (this) utilizando el operador ->.
    - _name es un atributo de tipo std::string, que contiene el nombre del personaje.
    - _weapon es un objeto Weapon y se invoca su método getType() para obtener el tipo de arma (de tipo std::string).
- La salida será algo como: Bob attacks with his sword

Clase HumanB

```
#include "HumanB.hpp"

HumanB::HumanB(std::string name) : _name(name){
  return ;
}
```

```
HumanB::~HumanB(void){
  return ;
}

void HumanB::attack(void){
  std::cout << this->_name << " attacks with his " << (*this->_weapon).getType() <<
std::endl;
}

void HumanB::setWeapon(Weapon &weapon){
  this->_weapon = &weapon;
}
```

1. Constructor HumanB::HumanB

```
HumanB::HumanB(std::string name) : _name(name){
  return ;
}
```

- El constructor recibe un std::string llamado name, que es asignado al atributo privado _name de la clase.
- En este caso, no se inicializa el arma en el constructor, lo que significa que HumanB no necesariamente tendrá un arma cuando se crea (es diferente de HumanA, que siempre recibe un arma).
- La palabra clave return ; es nuevamente innecesaria.

2. Destructor HumanB::~HumanB

```
HumanB::~HumanB(void){
  return                                                                    ;
}
```

- El destructor es similar al de HumanA, sin ninguna operación especial. Tampoco hay memoria dinámica que liberar.
- El return ; es innecesario, pero está permitido.

3. Método HumanB::attack

```
void HumanB::attack(void){
  std::cout << this->_name << " attacks with his " << (*this->_weapon).getType() <<
std::endl;
}
```

- Similar a HumanA::attack, este método imprime en pantalla un mensaje indicando que el personaje ataca con su arma.
- La diferencia clave aquí es que HumanB almacena un puntero (Weapon* _weapon) en lugar de una referencia. Por lo tanto, es necesario desreferenciar el puntero usando el operador *:
  - (*this->_weapon).getType() accede al objeto Weapon apuntado por _weapon y luego

invoca el método getType() para obtener el tipo del arma.

- Si _weapon es NULL (es decir, no se ha asignado un arma), esto causará un error en tiempo de ejecución, por lo que siempre se debe asegurar que _weapon apunte a un objeto válido antes de llamar a attack.

4. Método HumanB::setWeapon

```
void HumanB::setWeapon(Weapon &weapon){
  this->_weapon = &weapon;
}
```

- Este método permite asignar un arma a un objeto HumanB después de su creación.
- Se pasa una referencia al objeto Weapon y se guarda la dirección de memoria de ese objeto en el puntero _weapon.
- Esto permite que el objeto HumanB tenga un arma asignada en un momento posterior a su creación.

Resumen de las diferencias clave:

- HumanA:
  - Usa una referencia al arma (Weapon& _weapon).
  - El arma se asigna en el constructor y siempre estará disponible.
- HumanB:
  - Usa un puntero al arma (Weapon* _weapon).
  - El arma se puede asignar más tarde mediante el método setWeapon.

Este enfoque con punteros y referencias es clave para entender cómo se gestionan las relaciones entre objetos en C++ y cómo se pueden modificar después de la creación de un objeto.

Weapon.hpp & Weapon.cpp

El código proporcionado implementa la clase Weapon en C++ siguiendo el estándar C++98. A continuación, explico cada parte de forma clara, concisa y detallada:

Archivo de encabezado: Weapon.hpp

El archivo Weapon.hpp define la estructura de la clase Weapon y actúa como interfaz para su implementación.

1. Directivas de preprocesador:

```
#ifndef WEAPON_HPP
#define WEAPON_HPP
```

   a. Estas directivas previenen la inclusión múltiple del archivo en un mismo proyecto.
   b. Si WEAPON_HPP no está definido, se define, y todo el contenido del archivo queda

protegido.

2. Inclusión de dependencias:

```
#include <iostream> // Para std::cout y std::endl (aunque no se usa en este archivo).
#include <string>   // Para manejar cadenas de texto con std::string.
```

3. Definición de la clase Weapon:

```
class Weapon {
public:
  Weapon(std::string weaponType);  // Constructor
  ~Weapon();                  // Destructor

  std::string const &getType();    // Devuelve una referencia constante al tipo del arma
  void setType(std::string weaponType); // Modifica el tipo del arma

private:
  std::string _type; // Atributo privado que almacena el tipo del arma
};
```

   a. Atributos:
      i. _type: Una cadena de texto que almacena el tipo del arma.
   b. Métodos:
      i. Weapon(std::string weaponType): Constructor que inicializa el arma con un tipo.
      ii. ~Weapon(): Destructor que limpia los recursos (aquí no hace nada, pero está presente por coherencia con C++98).
      iii. getType(): Devuelve una referencia constante al atributo _type.
      iv. setType(std::string weaponType): Permite cambiar el valor del atributo _type.

4. Cierre de las directivas de preprocesador:

```
#endif
```

Archivo de implementación: Weapon.cpp

Este archivo implementa las funciones miembro declaradas en Weapon.hpp.

1. Inclusión del encabezado:

```
#include "Weapon.hpp"
```

2. Constructor:

```
Weapon::Weapon(std::string weaponType) {
  this->setType(weaponType);
  return;
}
```

a. Inicializa el arma asignándole un tipo mediante el método setType.
b. El uso de this-> asegura que estamos manipulando el atributo _type del objeto actual.
c. El return; es redundante en un constructor pero permitido en C++98.

3. Destructor:

```
Weapon::~Weapon(void)                                                    {
  return;
}
```

a. El destructor no realiza ninguna acción específica, ya que no se reserva memoria dinámica ni se manejan recursos externos.

4. Getter:

```
std::string const &Weapon::getType() {
  return (this->_type);
}
```

a. Devuelve una referencia constante al atributo _type.
b. El uso de una referencia evita la copia del valor, lo que mejora el rendimiento.
c. La referencia constante impide modificar _type a través del retorno.

5. Setter:

```
void Weapon::setType(std::string weaponType) {
  this->_type = weaponType;
}
```

a. Actualiza el atributo _type con el valor recibido como argumento.
b. No devuelve nada (void).

Puntos clave del diseño

1. Encapsulación:
   a. El atributo _type es privado, lo que protege su acceso directo. En su lugar, se utilizan los métodos getType y setType.
2. Uso de referencias constantes:
   a. getType devuelve una referencia constante a _type, lo que evita copias innecesarias y asegura que el valor no pueda ser modificado.
3. Conformidad con C++98:
   a. No se usan características modernas (como inicializadores de miembros o delegación de constructores) introducidas en versiones posteriores de C++.
4. Simplicidad y eficiencia:
   a. Aunque es una implementación básica, sigue las mejores prácticas de encapsulación y

claridad en el diseño.

En resumen:

El código define una clase Weapon que representa un arma con un tipo (_type). El arma puede inicializarse con un tipo, y este tipo puede ser consultado o modificado mediante métodos públicos. El diseño es eficiente y seguro, utilizando referencias constantes para evitar copias innecesarias y encapsulando el atributo para protegerlo de accesos directos.

main.cpp

El código está diseñado para probar las clases Weapon, HumanA, y HumanB implementadas en base al estándar C++98. Vamos a analizar cada parte del código, explicando su propósito y cómo funciona.

Encabezados incluidos

```
#include "HumanA.hpp"
#include "HumanB.hpp"
#include "Weapon.hpp"
```

Estos encabezados incluyen las definiciones de las clases Weapon, HumanA, y HumanB. Cada archivo .hpp debe contener las declaraciones de las clases, mientras que sus implementaciones estarán en los archivos .cpp correspondientes.

Función main()

La función principal realiza dos bloques independientes de código. Estos bloques demuestran cómo interactúan las clases Weapon, HumanA y HumanB en diferentes situaciones.

Bloque 1: HumanA

```
{
    Weapon club = Weapon("crude spiked club");
    HumanA bob("Bob", club);
    bob.attack();
    club.setType("some other type of club");
    bob.attack();
}
```

1. Weapon club = Weapon("crude spiked club");
    a. Se crea un objeto Weapon llamado club y se inicializa con el tipo "crude spiked club".
    b. Esto llama al constructor de la clase Weapon.

2. HumanA bob("Bob", club);
    a. Se crea un objeto HumanA llamado bob, con el nombre "Bob" y el arma club.
    b. HumanA utiliza una referencia a Weapon, por lo que el objeto club está vinculado

directamente al arma de Bob.

3. bob.attack();

    a. Llama a la función miembro attack() de la clase HumanA.
    b. La salida esperada es: Bob attacks with their crude spiked club

4. club.setType("some other type of club");

    a. Cambia el tipo del arma club a "some other type of club". Como HumanA utiliza una referencia al objeto club, el cambio se refleja automáticamente en bob.

5. bob.attack();
    a. Llama nuevamente a attack(), mostrando el cambio en el arma.
    b. La salida esperada es: Bob attacks with their some other type of club

Bloque 2: HumanB

```
{
        Weapon club = Weapon("crude spiked club");
        HumanB jim("Jim");
        jim.setWeapon(club);
        jim.attack();
        club.setType("some other type of club");
        jim.attack();
}
```

1. Weapon club = Weapon("crude spiked club");

    a. Se crea otro objeto Weapon llamado club con el tipo "crude spiked club".

2. HumanB jim("Jim");

    a. Se crea un objeto HumanB llamado jim con el nombre "Jim".
    b. A diferencia de HumanA, HumanB no tiene un arma asignada inicialmente, ya que no la recibe en el constructor.

3. jim.setWeapon(club);

    a. Asigna el arma club al objeto jim mediante un método público. Es probable que setWeapon() utilice un puntero (Weapon*) para permitir esta asignación.

4. jim.attack();

    a. Llama a la función miembro attack() de la clase HumanB.
    b. Como jim tiene ahora asignada el arma club, la salida esperada es: Jim attacks with their crude spiked club

5. club.setType("some other type of club");

    a. Cambia el tipo del arma club. Como HumanB utiliza un puntero al arma, el cambio se refleja en jim.

6. jim.attack();

    a. Llama nuevamente a attack(), mostrando el cambio en el arma.
    b. La salida esperada es: Jim attacks with their some other type of club

Comentario sobre la gestión de memoria

- HumanA: Usa una referencia a Weapon. Esto asegura que el objeto siempre esté vinculado a un arma válida y no necesita manejo explícito de memoria.

- HumanB: Usa un puntero a Weapon. Este diseño permite que HumanB no tenga un arma inicialmente. Sin embargo, requiere asegurarse de que el puntero apunte a un objeto válido antes de usarlo, evitando accesos nulos.

En resumen:

El programa demuestra cómo las clases Weapon, HumanA, y HumanB interactúan, mostrando las diferencias en el manejo de armas entre los personajes. HumanA siempre tiene un arma, asignada por referencia en el constructor, mientras que HumanB puede no tener un arma inicialmente y la asigna dinámicamente con un puntero. La conexión entre los objetos permite reflejar cambios en el arma en tiempo de ejecución.