UNIVERSIDAD NACIONAL DEL LITORAL
FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS

Tecnicatura en diseño y programación de videojuegos

UNL VIRTUAL

Modelos y algoritmos para videojuegos I

Capítulo ½
Orientación a objetos

Docentes
Sebastián Rojas Fredini
Patricia Schapschuk

CONTENIDOS

CAPÍTULO ½
ORIENTACIÓN A OBJETOS

½. 1. Sobre la representación abstracta del mundo

En materias precedentes se vieron conceptos básicos de programación. En particular, se utilizó el paradigma de la programación estructurada, donde un programa se compone de un conjunto de funciones.

En un principio, dicho paradigma era suficiente para la complejidad de aplicaciones que se podían implementar, pero, con el correr de los años, mostró problemas e inconvenientes. Esto propició el terreno para el surgimiento de un nuevo paradigma que se denominó *Programación Orientada a Objetos* (POO). Según el mismo, nuestro programa ya no se compone de un conjunto de funciones, sino que es un conjunto de objetos que interactúan entre sí.

Veamos, primero, un ejemplo práctico que nos ayude a comprender los conceptos abstractos que rodean a la POO.

Si en nuestro programa necesitamos sumar dos números, tenemos tipos de datos que representan dichos números, por ejemplo el tipo *float*. Para realizar la operación, simplemente definimos dos variables y les aplicamos la operación. En este caso, lo que queremos hacer tiene una correspondencia directa con el código y los tipos de datos nativos del compilador.

Si, por ejemplo, nuestro problema es contar la cantidad de vocales de una frase, entonces utilizamos el tipo de dato de C++ string y realizamos las operaciones sobre dicho tipo. Acá también podemos observar una correspondencia directa entre el problema y los tipos de datos usados.

Esta correspondencia sólo aparece en casos particulares y generalmente no podemos expresar nuestro problema del mundo real en función de los tipos de datos nativos del compilador. Para subsanar dicha cuestión, C++ permite al usuario definir tipos de datos propios, que describan mejor los objetos del mundo real que participan en la resolución del problema.

Imaginemos que queremos hacer un videojuego de aventuras, donde uno de los elementos que interviene es el personaje principal, que llamaremos *Caballero.* ¿Cómo hacemos para representar un Caballero con los tipos de datos de C++? C++ no provee nativamente ningún tipo de dato que describa un Caballero. Para ello, debemos definir un tipo de datos propio. En la POO, estos tipos de datos propios se denominan *clases* y son similares a los conceptos de structs que ya se han visto.

Para definirlo, entonces, lo primero que debemos hacer es preguntarnos qué atributos posee un Caballero en nuestro videojuego. Probablemente, una de estas características sea la cantidad de salud de la que dispone. Otra, tal vez, la cantidad de maná que le queda para realizar hechizos. Tomemos como ejemplo dichos atributos, que denominaremos *propiedades,* y definamos nuestro tipo de datos en C++:

```
class Caballero{

	int salud;
	int mana;
};
```

Para definir el tipo de datos se utiliza la palabra clave `class,` seguida del nombre de dicho tipo de datos. Luego, entre llaves, se definen los atributos de nuestro Caballero. Como se puede observar, las propiedades se construyen utilizando —ahora sí— tipos de datos nativos de C++. Particularmente, Caballero posee salud, que es un entero, al igual que maná.

Además de las propiedades, el Caballero en nuestro videojuego puede realizar algunas acciones, como por ejemplo saltar y recibir un golpe. Estas acciones las materializamos en nuestro tipo de datos mediante métodos, que no son más que funciones que pertenecen a una clase, que modifican o dan información sobre el Caballero. En nuestro ejemplo:

```
class Caballero{
	//propiedades
	int salud;
	int mana;
	//métodos
	void saltar();
	void recibirGolpe();
} ;
```

El método `saltar()` no devuelve nada ni recibe argumentos. Este método deberá encargarse de hacer que nuestro personaje salte en el mundo virtual. Por otro lado, `recibirGolpe()` tampoco recibe argumentos ni devuelve nada, y debería encargarse de modificar el estado de Caballero para reflejar que recibió un golpe.

En nuestro ejemplo sencillo, lo que deberíamos hacer es decrementar la salud para reflejar que ahora tiene un punto menos de esa propiedad.

Si agregamos dicho cambio a nuestro Caballero, obtenemos lo siguiente:

```
class Caballero{
	//propiedades
	int salud;
	int mana;
	//métodos
	void saltar();
	void recibirGolpe(){
		salud--;
	}
};
```

Aquí vemos que el método de recibir un golpe fue implementado y hace lo que dijimos previamente: decrementa la salud del Caballero en un punto.

Para tener una clase un poco más completa, vamos a agregar un método que permita al Caballero restaurar salud. Siguiendo las mismas ideas expuestas anteriormente, nuestra clase queda:

```
class Caballero{
	//propiedades
	int salud;
	int mana;
	//métodos
	void saltar();
	void recibirGolpe(){
		salud--;
	}
	void recibirCura(int puntos){
		salud+=puntos;
	}
};
```

Aquí podemos observar el método `recibirCura()`, que recibe un parámetro que indica cuántos puntos de salud recuperar. Por ejemplo, si el Caballero junta un ítem que le recupera la salud un punto, se llama el método con el valor puntos en uno, mientras que si el Caballero junta un ítem que le recupera 10 puntos la salud, será ése el valor que le pasamos.

Ya tenemos nuestra clase Caballero medianamente funcional. Ahora veamos cómo hacer para crear una variable de este nuevo tipo que definimos. Vamos a crear un Caballero que llamaremos *lancelot.* Para ello, en nuestro punto de entrada a la aplicación (main), hacemos lo siguiente:

```
void main(int argc, char* argv[])
{
	Caballero lancelot;

	return;
}
```

lancelot es una variable de tipo Caballero. En la POO, estas variables se denominan *objetos*. Esto significa que un objeto es una variable cuyo tipo es una clase. Se suele decir que Caballero es como un molde, y cuando creamos objetos a partir de ese molde, se crean instancias del molde. En nuestro caso, *lancelot* es una instancia de la clase Caballero. Así como declaramos a *lancelot*, podemos declarar otro Caballero *percival*.

```
void main(int argc, char* argv[])
{
	Caballero lancelot;
	Caballero percival;

	return;
}
```

De esta forma, ya poseemos nuestro tipo de datos, pero nos falta todavía ver un detalle en cuanto a cómo accedemos a los miembros de Caballero. Si realizamos lo siguiente, para hacer que lancelot pierda un punto de salud, veremos que el compilador nos da un error:

```
void main(int argc, char* argv[])
{
	...

	lancelot.recibirGolpe();

	...
}
```

Los miembros de una clase pueden o no ser accesibles desde afuera. Si no se indica nada, las propiedades y métodos son privadas de la clase y no pueden ser accesibles desde afuera, tal como sucede en nuestro ejemplo; `recibirGolpe()` se considera privado a la clase, entonces desde el main no lo podemos utilizar.

Así como se encuentra, nuestra clase Caballero es inutilizable porque todos sus miembros son privados. Por lo tanto, para indicar que un grupo de miembros debe ser accesible desde afuera, utilizamos la palabra clave *public*.

```
class Caballero{

	int salud;
	int mana;
public:
	void saltar();
	void recibirGolpe(){
		salud--;
	}
	void recibirCura(int puntos){
		salud+=puntos;
	}
   };
```

De esta forma, ahora sí podemos invocar el método, para recibir un golpe o para curarse, desde nuestro main.

```
void main(int argc, char* argv[])
{
	...

	lancelot.recibirGgolpe();
	percival.recibirCura(10);

	...
}
```

En el próximo parágrafo vamos a ver ya con más formalismo las definiciones de la POO y los conceptos que intervienen.

½. 2. Clases y objetos

Como vimos en el ejemplo, la POO se basa en la idea de utilizar tipos de datos que se corresponden con los elementos reales que intervienen en nuestro problema. La idea es generar abstracciones de los elementos de la realidad que nos sirvan para nuestro propósito. Dichas abstracciones se materializan en la definición de clases en nuestro código. Es decir, una clase es:

Clase
Abstracción de un objeto. Se materializa en una definición de un nuevo tipo de datos en C++.

Dichas clases son, justamente, definiciones. Esto es, como un molde a partir del cual se crean variables de ese tipo. A estas variables se les llama objetos, que son instancias de dicha clase.

Objeto
Instancia de una clase. Variable cuyo tipo es una clase.

Para decirlo de otro modo, el objeto es lo que obtenemos al usar el molde. La clase es sólo la definición, mientras que el objeto es una entidad concreta que responde a esa definición.

Por ejemplo, si estamos modelando una aplicación de una universidad para hacer un seguimiento de los alumnos, cursos y docentes, seguramente vamos a tener las clases *alumno, profesor* y *curso*. Y en el caso de los cursos, contaremos con objetos para representar a cada uno de ellos, a saber: *matemática aplicada, introducción a la programación,* etc.

Siempre que pensamos en clases, lo hacemos en algo abstracto que define un tipo. En cambio, cuando hablamos de objetos, nos referimos a cosas concretas.

Si, por el contrario, estamos modelando un videojuego de carreras, seguramente tendremos las siguientes clases: *vehículo, circuito, piloto,* etc. Y en el caso de los vehículos, contaremos con objetos como *VW Golf, Ford Focus,* etc.

Recapitulando, la idea de la POO es representar nuestro programa en función de descripciones abstractas de los objetos reales que intervienen en nuestro problema.

Dichas clases poseen miembros, a los cuales dividimos en métodos y propiedades.

Propiedades
Variables miembro de una clase que definen las características del objeto al cual la clase representa.

Métodos
Funciones miembro de una clase que operan sobre las propiedades de la misma.

Los métodos se comportan exactamente igual que una función, en el sentido de que pueden recibir argumentos y devolver resultados.

Generalmente, en C++ la declaración de una clase y su definición suelen dividirse en archivos .h y .cpp, respectivamente. El .h es la declaración de la clase, es decir, un listado de las propiedades y métodos que contiene, pero sin detalles sobre cómo son implementados esos métodos.

Retomando el ejemplo de nuestra clase Caballero, su declaración es la siguiente:

Caballero.h

```
class Caballero{

      int salud;
      int mana;
public:
      void saltar();
      void recibirGolpe();
      void recibirCura(int puntos);
   };
```

Nótese el ";" al final de la declaración, al cerrar la llave. Esto es de uso obligatorio, y si no lo ponemos, veremos errores de compilación un poco difíciles de identificar.

Por otro lado, en el .cpp colocaremos la definición de la siguiente manera:

Caballero.cpp

```
#include "Caballero.h"

void Caballero::saltar(){


}

void Caballero::recibirGolpe(){
            salud--;
}

void Caballero::recibirCura(int puntos){
            salud+=puntos;
}
```

Al tener separados los archivos de declaración (.h) y definición (.cpp), debemos indicarle a nuestro .cpp cuál es el archivo que contiene las declaraciones de los métodos que implementaremos. Esto se traduce en una cláusula `#include`. La misma hace que, a la hora de compilar Caballero.cpp, se incluyan las definiciones de Caballero.h. El .h no se compila, ya que es el archivo de declaración de clase y debe ser incluido en todos los archivos de código fuente donde deseemos utilizar la clase Caballero.

Los nombres de los métodos y atributos son locales a la clase. Esto quiere decir que podríamos tener dos clases diferentes con los mismos nombres de métodos, sin que una guarde relación con la otra. Es por ello que, a la hora de implementar un método, debemos indicar a qué clase pertenece. Esto lo hacemos utilizando el operador de *scope* "::". Por ejemplo, `Caballero::saltar()` indica que vamos a implementar la función *saltar()* de la clase Caballero.

½. 3. Control de acceso

Las propiedades de un objeto siempre pueden ser accedidas por sus métodos. Basándonos en nuestro ejemplo, esto significa que salud y maná pueden ser accedidos por *saltar(), recibirGolpe(), recibirCura(int puntos)*.

Ahora bien, la pregunta lógica es si un método que no es miembro de la clase puede leer o escribir las variables maná y salud. Para definir este comportamiento se utiliza el control de acceso.

El control de acceso consiste en indicarle a las propiedades desde dónde pueden ser accedidas. Si una propiedad sólo puede ser accedida por métodos de la clase a la que pertenece y no por métodos de otras clases o funciones externas, entonces se declara con la palabra clave *private*. En nuestro ejemplo, maná y salud no deben ser modificadas por nadie externamente, por lo que deberíamos definirlas como private. Cabe destacar que si no le indicamos nada, todas las variables y métodos son privados por defecto.

Veamos cómo queda nuestra clase si indicamos explícitamente que nuestras propiedades son privadas:

Caballero.h

```
class Caballero{

private:
	int salud;
	int mana;
public:
	void saltar();
	void recibirGolpe();
	void recibirCura(int puntos);
};
```

Ahora veamos qué efectos tiene sobre nuestro main.cpp. Supongamos que queremos hacer lo siguiente:

main.cpp

```
void main(int argc, char* argv[])
{
	Caballero lancelot;

	lancelot.salud++;

	...

}
```

Para acceder a los miembros de una clase se utiliza el operador ".", como vimos anteriormente. En este caso, declaramos un objeto de tipo Caballero y luego intentamos acceder a la propiedad salud para incrementarla en uno. Pero como salud está declarada como private, este código no compilará, ya que sólo las funciones miembro de Caballero pueden acceder a dicha variable. Es decir, ninguna función externa a Caballero puede realizar cambios sobre sus propiedades. Esto se conoce como *encapsulamiento*, una forma de evitar que usuarios potenciales de nuestro código cambien variables que no deberían y así generar el mal funcionamiento de nuestros métodos. La idea es que sólo puedan cambiar los datos miembros mediante los métodos del objeto.

Así como las propiedades de un objeto pueden ser privadas, también los métodos pueden serlo, esto es, no ser llamados desde el exterior de nuestra clase. Por ejemplo, si definimos Caballero de la siguiente manera, todos nuestros métodos son privados y sólo pueden ser invocados por otros miembros de la clase, pero no por funciones externas:

Caballero.h

```
class Caballero{

private:
	int salud;
	int mana;
	void saltar();
	void recibirGolpe();
	void recibirCura(int puntos);
};
```

Si quisiéramos utilizar la función *recibirCura()* desde el main, no podríamos compilar el código debido a que estamos invocando una función de Caballero que es privada:

main.cpp

```
void main(int argc, char* argv[])
{
	Caballero lancelot;

	lancelot.recibirCura(1);

	...

}
```

Para indicarle que un miembro de la clase debe poder ser accedido externamente, utilizamos la palabra clave *public*. (¿original, no?). Entonces, podemos hacer:

Caballero.h

```
class Caballero{

private:
	int salud;
	int mana;
public:
	void saltar();
	void recibirGolpe();
	void recibirCura(int puntos);
};
```

De esta manera, los tres métodos pueden ser accedidos desde afuera.

Por lo tanto, ahora sí podemos compilar el main sin problemas y tendrá el comportamiento deseado:

main.cpp

```
void main(int argc, char* argv[])
{
	Caballero lancelot;

	lancelot.recibirCura(1);

	...

}
```

No obstante, uno podría preguntarse: ¿Por qué no hicimos públicas las variables directamente, en lugar de definir un método público para modificar las variables privadas? A decir verdad, lo podríamos haber hecho, pero hubiéramos estado

rompiendo con una de las bases de la POO, el encapsulamiento, que mencionamos anteriormente.

Definamos con un poco más de rigor en qué consiste el encapsulamiento:

Encapsulamiento
Ocultamiento del estado. Es decir, los datos miembros de un objeto sólo pueden ser accedidos por los métodos del mismo.

El encapsulamiento permite concentrarse en los métodos a utilizar y no en cómo los mismos trabajan sobre los datos. Al que utiliza la clase no debe interesarle cómo ésta maneja internamente el estado, sino qué métodos llamar para conseguir su propósito. Por ejemplo, un usuario de Caballero debe saber que *recibirCura()* hace que el Caballero recupere un punto de energía, pero no debe –necesariamente– conocer cómo hace la clase para llevar cuenta de la energía.

El encapsulamiento, además, nos permite realizar validaciones sobre las propiedades. Es decir, podríamos validar que la salud no se incremente más que el máximo posible. Si hiciéramos público el atributo salud, estaríamos a merced de que el usuario de la clase la incremente indiscriminadamente o, en el caso contrario, que el personaje tenga una salud negativa. Veamos, a continuación, ejemplos de ambos casos:

Caballero.cp

```
#include "Caballero.h"

void Caballero::saltar(){


}

void Caballero::recibirGolpe(){
              if(salud>0)
                     salud--;
}

void Caballero::recibirCura(int puntos){
              if( (salud+puntos)> 100)
                     salud=100
       else
                     salud+=puntos;

}
```

Recapitulando, los especificadores de acceso que disponemos son:

a) *Private* Pueden ser accedidos solamente por otros miembros de la clase y por clases amigas.

b) *Public* Pueden ser accedidos por cualquier función.

c) *Protected* Pueden ser accedidos por otros miembros de la clase, clases amigas y clases derivadas. (Ver parágrafo ½. 6).

½. 4. Te construyo, te destruyo y te vuelvo a construir

En una clase existen dos tipos de métodos especiales: los constructores y los destructores.

Un constructor es un método que se invoca automáticamente cuando el objeto es creado en memoria. Esto nos da la posibilidad de inicializar variables miembro. El constructor siempre se ejecuta antes de cualquier otro miembro y no devuelve ningún tipo de datos.

El compilador identifica qué método es el constructor, debido a que su nombre es idéntico al de la clase y no especifica tipo de retorno. Por ejemplo, si queremos que, cuando creamos nuestro objeto Caballero, éste arranque con los puntos de salud en 100, debemos hacer:

Caballero.h

```
class Caballero{

private:
	int salud;
	int mana;
public:
	Caballero();
	void saltar();
	void recibirGolpe();
	void recibirCura(int puntos);
};
```

Caballero.cpp

```
#include "Caballero.h"

Caballero::Caballero(){
	salud=100;
}
...
...
```

En la declaración (.h) podemos ver que definimos un método con el mismo nombre de la clase y no devuelve nada.

Por otro lado, en su implementación (.cpp) podemos observar que lo que hacemos es inicializar la propiedad salud en 100. De esta forma, siempre que creemos un objeto Caballero, éste tendrá la propiedad salud en 100. Un detalle a notar es que obviamente queremos que los constructores sean públicos, ya que el objeto debe poder ser construido desde afuera, por ejemplo, desde el main.

También podemos realizar sobrecarga de constructores, siguiendo las mismas reglas que se emplean para las funciones que ya conocen de la programación estructurada. Esto se traduce en que podemos tener más de un constructor y que éstos pueden recibir argumentos. Es decir, si queremos indicarle al constructor con cuánta salud inicializar al Caballero, podemos definir otro constructor adicional que reciba un entero como parámetro. Nuestro código, ahora, se vería de la siguiente manera:

Caballero.h

```
class Caballero{

private:
	int salud;
	int mana;
public:
	Caballero();
	Caballero(int _saludInicial);
	void saltar();
	void recibirGolpe();
	void recibirCura(int puntos);
};
```

Caballero.cpp

```
#include "Caballero.h"

Caballero::Caballero(){
```

```
	salud=100;
}
Caballero::Caballero(int _saludInicial){
	salud=_saludInicial;
}
...
```

Aquí podemos observar a los dos constructores, donde uno no recibe ningún argumento e inicializa la salud siempre en 100, mientras que el otro recibe como argumento cuál es la salud inicial del objeto. Veamos cómo se utiliza esto desde un main:

main.cpp

```
#include "Caballero.h"


void main(int argc, char* argv[])
{
	Caballero lancelot;
	Caballero percival(50);

	return;
}
```

En este ejemplo creamos dos objetos: *lancelot* y *percival*. Como se puede apreciar, los parámetros se pasan al constructor al momento de declarar el objeto. A *lancelot* no le pasamos ningún parámetro, por lo que se invoca al constructor sin parámetros, inicializando la salud en 100, mientras que a *percival* le pasamos un argumento entero, lo cual hace que se invoque al otro constructor y se inicialice la salud en 50.

En el primer caso, cuando el constructor es vacío, debemos tomar la precaución de no colocar los paréntesis, ya que, si lo hacemos, el compilador creerá que estamos declarando una función.

Si estuviésemos trabajando con punteros, los argumentos al constructor se pasan a la hora de crear el objeto en memoria, es decir, al utilizar el *new*. El mismo ejemplo, pero utilizando punteros, quedaría así:

main.cpp

```
#include "Caballero.h"


void main(int argc, char* argv[])
{
	Caballero *lancelot;
	Caballero *percival;

	lancelot= new Caballero();
	percival= new Caballero(50);

	return;
}
```

Aquí observamos que se invocan a los dos constructores, respectivamente, a la hora de crear el objeto con el *new*.

Por otro lado, como dijimos al principio, existen los denominados *destructores*. Cada objeto posee uno y sólo uno. El destructor se ejecuta justo antes de que el objeto se borre de la memoria. Esto le da al objeto la posibilidad de liberar los recursos que haya adquirido para su funcionamiento y cualquier otra operación necesaria para la lógica de nuestra aplicación. El destructor se declara utilizando el mismo nombre de la clase,

pero anteponiendo un "~", y tampoco devuelve nada. Por ejemplo, en el caso de Caballero, el destructor debe declararse e implementarse de la siguiente manera:

Caballero.h

```
class Caballero{

private:
	int salud;
	int mana;
public:
	Caballero();
	Caballero(int _saludInicial);
	~Caballero();
	...
};
```

Caballero.cpp

```
#include "Caballero.h"

Caballero::~Caballero(){

}
```

En nuestro caso particular, el destructor no realiza ninguna tarea, ya que no hemos alojado memoria ni recursos que deban ser liberados.

½. 4.1. Sobre la obligatoriedad de los constructores y destructores

¿Es acaso necesario declarar siempre un constructor y un destructor? La respuesta es *no*. Si no declaramos un constructor, el compilador generará uno automáticamente que, por supuesto, no hará nada; sería equivalente a declararle uno vacío. Lo mismo sucede con los destructores: si no los declaramos, se sintetizará uno automáticamente, pero el mismo no realizará tarea alguna.

Los constructores y destructores no son necesarios, pero sí muy útiles, ya que nos proveen la posibilidad de inicializar y destruir nuestro objeto de la manera que nos parezca más conveniente; seguramente observarán que en la práctica son tremendamente útiles.

No se debe olvidar que C++ no inicializa por defecto las variables. Es decir, si declaramos un entero sin asignarle un valor, ese entero contendrá basura, esto es, datos no útiles, lo cual significa que no valdrá 0, como sí sucede con otros lenguajes. Por ello, en C++ siempre es necesario inicializar las variables en algún valor, y es aquí donde entran en juego los constructores y destructores.

Toda clase que maneje memoria dinámica (*new*) deberá liberarla (*delete)* en los destructores, ya que, si no lo hacemos, estaremos produciendo *memory leaks*, que es memoria no liberada que ocupa recursos, aunque no utilicemos más la clase que creó dichos bloques de memoria.

½. 5. Miembros, punteros y otros rituales (o breve historia de cosas enredadamente sencillas)

Hasta aquí hemos estado utilizando el operador "." para acceder a los miembros de un objeto, pero cuando trabajamos con punteros, la sintaxis cambia sutilmente y tenemos dos formas de acceder a un miembro.

La primera es utilizando el operador de desreferencia "*"; para obtener el objeto al que apunta un puntero y luego a sus miembros, se los puede acceder como si se tratase de un objeto normal, utilizando el operador "."

Por ejemplo:

main.cpp

```
#include "Caballero.h"


void main(int argc, char* argv[])
{
	Caballero *percival;
	percival= new Caballero(50);

	(*percival).recibirCura(10);

	return;
}
```

Aquí, `percival` es un puntero a un objeto de tipo Caballero, pero, al aplicarle el operador "*", obtenemos el objeto al cual apunta. Es decir, `(*percival)` representa ya al objeto, por lo que podemos tratarlo como un objeto tradicional y acceder a sus miembros. Esta sintaxis es un poco tediosa y por ello se introdujo luego una forma más sencilla de escribir lo mismo: utilizando el operador "->". El mismo se emplea para acceder a los miembros de objetos mediante su puntero. Así, el ejemplo anterior quedaría de la siguiente manera:

main.cpp

```
#include "Caballero.h"


void main(int argc, char* argv[])
{
	Caballero *percival;
	percival= new Caballero(50);

	percival->recibirCura(10);

	return;
}
```

Como vemos, ya no se necesita utilizar los paréntesis ni el operador de indirección. De esta forma, estamos accediendo a un miembro de Caballero utilizando un puntero a Caballero. Y por ser un puntero para acceder a un miembro, en vez de ".", debemos utilizar "->".

½. 6. Herencia

Herencia es otro de los conceptos fundamentales que intervienen en la POO.

En los puntos anteriores ya hemos definido una clase Caballero con ciertos métodos y propiedades.

Ahora, imaginemos que en nuestro juego tenemos dos tipos de Caballero: Arqueros y Paladines. Claramente, sendos personajes comparten los atributos de un Caballero (maná y salud), pero cada uno tiene otros atributos que los diferencia entre sí.

Supongamos que el Arquero posee dos métodos, *disparar()* y *recibirFlechas(int cantidad),* adicionales a los métodos de Caballero, y posee otra propiedad que indica la cantidad de flechas que le quedan.

Por otro lado, Paladin posee un método, *golpear()* –porque por supuesto los paladines tienen espadas gigantes–, y un método para elegir la espada que lleva actualmente,

cambiarArma(string nombre). Además, posee un vector con los nombres de los dioses que le han dado su favor.

Si no conociéramos nada más que lo que hemos visto hasta ahora, tendríamos que definir las dos clases repitiendo los atributos que tienen en común y sus métodos. Esto nos generaría el siguiente código:

Arquero.h

```
class Arquero
{

private:
	int salud;
	int mana;

	//propiedades especificas de arquero
	int flechas
public:
	//mismo metodos de caballero
	Arquero();
	Arquero(int _saludInicial);
	~Arquero();
	void saltar();
	void recibirGolpe();
	void recibirCura(int puntos);

	//metodos especificos de arquero
	void Disparar();
	void recibirFlechas(int cantidad);

};
```

Paladin.h

```
#include <iostream>
using namespace std;

class Paladin
{
private:
	int salud;
	int mana;
	string dioses[10];

public:
	//mismos metodos que caballero
	Paladin();
	Paladin(int _saludInicial);
	~Paladin();
	void saltar();
	void recibirGolpe();
	void recibirCura(int puntos);

	//metodos propios de paladin
	void golpear();
	void cambiarArma(string nombre);

};
```

Como se puede observar, ambos repiten los métodos que tiene Caballero, ya que tanto Arquero como Paladin son Caballeros, pero además definen propiedades y métodos que los diferencian de aquél.

Esta repetición de código hace que el mismo se vuelva imposible de mantener e ilegible. Por ello, para solucionar este tipo de casos, surgió la herencia.

La idea es muy sencilla: si ambos objetos son Caballeros, no repetimos los atributos en los dos, sino que los definimos una vez y, de *alguna manera*, les indicamos que, tanto Arquero como Paladin, también son Caballeros, y definimos solamente los atributos adicionales que posee cada uno. Esto se denomina herencia. En otras palabras, podemos hacer que una clase herede las propiedades y métodos de otra, la cual se suele denominar *superclase* o *clase padre/madre*. De esta forma, la clase que hereda tiene todo lo mismo que el padre, más lo que defina ella misma.

Veamos, por ejemplo, como quedaría Paladin si hacemos que herede de Caballero:

Paladin.h

```
#include <iostream>
#include "Caballero.h"
using namespace std;

class Paladin : public Caballero
{
private:

	string dioses[10];

public:
	//metodos propios de paladin
	void golpear(){};
	void cambiarArma(string nombre){};
};
```

Para indicar que esta clase es hija o heredera de Caballero, utilizamos el operador ":" a continuación del nombre de clase. Luego, señalamos cuál es la clase de la que hereda (en nuestro caso, Caballero).

Como vemos, también indicamos si los métodos y propiedades que heredamos de Caballero son públicos o privados, es decir, si pueden ser accedidos desde afuera de la clase Paladin o sólo por los métodos de la misma.

Como a nosotros nos interesa que Paladin se comporte igual que Caballero y podamos utilizar desde afuera sus métodos, entonces declaramos la clase como *public*. A los fines prácticos, esta declaración equivale a la acción de repetir todos los miembros, tal como lo hicimos anteriormente. Es decir, además de los miembros que definimos en Paladin.h, Paladin tendrá, como si fueran propios, todos los miembros de Caballero.h.

Ahora bien, en Caballero.h, los atributos salud y maná son privados. Todos los atributos privados de la superclase *no son accesibles* desde la subclase. Esto significa que nuestro Paladin no tendrá acceso a los enteros salud y maná.

Entonces, para que un atributo sea privado pero se pueda usar en clases heredadas, el mismo debe ser declarado con la palabra clave *protected,* cuyos efectos son iguales a *private,* pero además permite que una clase hija tenga acceso a dicho atributo. Realizando este cambio en Caballero.h, obtenemos:

Caballero.h

```
class Caballero{

protected:
	int salud;
	int mana;
public:
	Caballero();
	Caballero(int _saludInicial);
	~Caballero();
	void saltar();
	void recibirGolpe();
	void recibirCura(int puntos);
};
```

Con esta declaración, sí podemos hacer que, tanto Arquero como Paladin, hereden todos los atributos de Caballero.

En función de lo que hemos visto, repasemos ahora algunas definiciones.

Herencia
La herencia es uno de los mecanismos de la POO por medio del cual una clase se deriva de otra, llamada entonces superclase, de manera que extiende su funcionalidad.

Para definir qué miembros de la superclase son visibles en las clases heredadas se utilizan los especificadores de acceso:

Private Todos los miembros *protected* y *public* de la superclase son *privados* en la clase derivada.

Protected Todos los miembros *protected* y *public* de la superclase son *protected* en la clase derivada.

Public Todos los miembros *protected* y *public* conservan su especificador de acceso original.

Como dijimos anteriormente, todos los miembros *private* de la superclase no son heredados por la clase derivada.

½. 6.1. Herencia de constructores y destructores

Los constructores y destructores de una superclase *no son heredados* automáticamente por las clases derivadas. Sin embargo, es posible utilizar los constructores/destructores (ctors/dtors) desde una clase derivada, pero debemos invocarlos explícitamente. Si lo hacemos, debemos tener en cuenta dos cuestiones fundamentales:

- El constructor de la superclase se invoca antes que el de la clase derivada.
- El destructor de la clase derivada se invoca antes que el de la superclase.

Veamos cómo invocar al constructor de una superclase, retomando nuestro ejemplo.

En función de lo que hemos dicho, Paladin, que hereda de Caballero, debería tener su propio constructor, ya que no hereda el de Caballero.

Ahora, si suponemos que un Paladin posee siempre 500 de salud, podríamos utilizar el constructor de Caballero que recibe el parámetro para setear la salud.

Entonces, declaremos –primero– el constructor en Paladin.h:

Paladin.h

```
#include <iostream>
#include "Caballero.h"
using namespace std;

class Paladin : public Caballero
{
private:

	string dioses[10];

public:
	//metodos propios de paladin
	Paladin();
	void golpear(){};
	void cambiarArma(string nombre){};
};
```

Y luego, en la definición:

Paladin.cpp

```
#include "Paladin.h"

Paladin::Paladin():Caballero(500){


}
```

Así, indicamos que el constructor de Paladin debe primero el constructor de Caballero, pasándole como argumento la salud. Es decir, utilizamos el operador ":", seguido del nombre de la clase, cuyo constructor queremos llamar acompañado de los argumentos que le proveemos a dicho constructor. En este caso, siempre construimos a Caballero con 500 porque nuestros Paladines siempre poseen esa cantidad de salud inicial.

Si no especificamos argumentos para el constructor de la superclase, la misma debe tener sí o sí un constructor vacío.

½. 6.2. Sobrecarga de métodos

En una clase derivada podemos utilizar los métodos de la superclase (si así lo permiten los modificadores de acceso) y también sobrecargarlos o volver a implementarlos con otra funcionalidad. En el próximo parágrafo veremos más en detalle estas ideas.

½. 7. Polimorfismo

El polimorfismo, junto con el encapsulamiento y la herencia, son los tres grandes conceptos claves de la POO.

El concepto de polimorfismo es amplio y se puede dividir en muchos tipos. En particular, nosotros veremos el polimorfismo de subtipo, que es al que comúnmente se hace referencia cuando se habla de este tema.

Polimorfismo
Posibilidad de definir subclases diferentes que tienen métodos o atributos denominados de forma idéntica, pero que se comportan de manera distinta.

En particular, en los lenguajes fuertemente tipados, esto se suele traducir en una jerarquía de herencia, donde la superclase define los métodos y las clases hijas pueden cambiar su comportamiento.

Vamos a retomar nuestro ejemplo del Caballero, Arquero y Paladin, para ver de manera más concreta qué significa el polimorfismo.

Nuestro Caballero posee un método *recibirGolpe()*, el cual es heredado por el Arquero y el Paladin. Pero está claro que un arquero, al no llevar una armadura pesada, recibe mayor daño que el Paladin. Por lo tanto, no podemos implementar el método en Caballero y después utilizarlo en Arquero y Paladin, porque estaríamos restando la misma cantidad de energía a ambos.

Veamos esto en código; supongamos que implementamos el método en Caballero:

Caballero.cpp

```
...
void Caballero::recibirGolpe(){
            salud--;
}
...
```

Entonces, si en el main declaramos un Arquero y un Paladin e invocamos dicho método:

main.cpp

```
#include "Caballero.h"
#include "Arquero.h"
#include "Paladin.h"


void main(int argc, char* argv[])
{
	Arquero *robin_hood;
	Paladin *sir_lancelot;

	robin_hood= new Arquero();
	sir_lancelot= new Paladin();

	robin_hood->recibirGolpe();
	sir_lancelot->recibirGolpe();


	return;
}
```

En ambos casos, *recibirGolpe()* les va a restar un punto de salud porque se llamará al método de la superclase Caballero. Pero, como dijimos anteriormente, esto no tendría que ser así, ya que el arquero debería sufrir un daño mayor.

Entonces, ¿cómo hacemos para que el método se comporte de diferente manera? En primer lugar, debemos redefinirlo en las clases que deseemos que se comporte de distinta manera. Nosotros solo lo redefiniremos, en principio, en Arquero:

Arquero.h

```
class Arquero : public Caballero
{

private:

	int flechas;
public:

	Arquero();
	void recibirGolpe();
	void Disparar();
	void recibirFlechas(int cantidad);

};
```

Arquero.cpp

```
...

Arquero::recibirGolpe(){
	salud-=5;
}
...
```

De esta manera, hicimos que, al recibir un golpe, Arquero pierda 5 puntos de energía en vez de uno solo. Si agregamos el siguiente método a Caballero.h:

```
int saludRestante(){
		return salud;
	}
```

para poder obtener la salud restante y ejecutamos el siguiente main:

main.cpp

```
#include "Caballero.h"
#include "Arquero.h"
#include "Paladin.h"
#include <iostream>

using namespace std;

void main(int argc, char* argv[])
{
	Arquero *robin_hood;
	Paladin *sir_lancelot;

	robin_hood= new Arquero();
	sir_lancelot= new Paladin();

	robin_hood->recibirGolpe();
	sir_lancelot->recibirGolpe();

	cout<<robin_hood->saludRestante()<<endl;
	cout<<sir_lancelot->saludRestante()<<endl;

	cin.get();


	return;
}
```

veremos que, efectivamente, esto sucede.

Esta es una de las bases del polimorfismo, es decir, que un método se comporte de manera distinta dependiendo del objeto. Esto permite que tengamos objetos con los mismos métodos heredados, pero que se comportan de distinta manera en cada clase hija.

½. 7. 1. Virtual

Cuando tenemos herencia de clases, podríamos definir un objeto de tipo Arquero y usar un puntero de tipo Arquero, que es lo más tradicional:

main.cpp

```
#include "Caballero.h"
#include "Arquero.h"
#include "Paladin.h"


void main(int argc, char* argv[])
{
	Arquero *robin_hood;

	robin_hood= new Arquero();

	robin_hood->recibirGolpe();

	return;
}
```

Pero también podríamos definir un objeto de tipo Arquero, apuntándolo con un puntero de tipo Caballero. Esto es posible porque Arquero es una clase derivada de Caballero. Es decir, es un Caballero pero, además, posee otros miembros.

Si realizamos esto:

main.cpp

```
#include "Caballero.h"
#include "Arquero.h"
#include "Paladin.h"

void main(int argc, char* argv[])
{
	Caballero *robin_hood;

	robin_hood= new Arquero();

	robin_hood->recibirGolpe();

	return;
}
```

el programa compilará sin errores ni *warnings*. En este caso, *robin_hood* es un puntero a un Caballero, pero, al hacer un *new,* estamos creando un Arquero. Lo que tenemos en memoria es efectivamente un Arquero con todos sus miembros, pero, a través del puntero que tenemos, sólo podemos acceder a los miembros que heredó de Caballero, no a los que agregó Arquero. Es como si sólo pudiéramos ver la parte del objeto que es Caballero y no los demás miembros que, sin embargo, están en memoria. Es, simplemente, mirar un objeto a través de otra perspectiva; mirar un Arquero como si sólo fuera un Caballero. Si, por ejemplo, intentásemos llamar el método *Disparar(),* el compilador se quejaría, ya que un Caballero no posee dicho método, a pesar de que el objeto sea, en realidad, una subclase de Caballero y sí lo implemente.

Lo que podríamos hacer ahora es volver a mirar al objeto como lo que es: un Arquero, y ahí sí llamar a los métodos de Arquero:

main.cpp

```
#include "Caballero.h"
#include "Arquero.h"
#include "Paladin.h"

void main(int argc, char* argv[])
{
	Caballero *robin_hood;
	Arquero *rb;

	robin_hood= new Arquero();
	rb= (Arquero*)robin_hood;

	rb->Disparar();


	return;
}
```

En este fragmento simplemente convertimos el puntero *robin_hood,* que es de tipo Caballero, al tipo Arquero, utilizando un *cast* de tipos. Por lo tanto, estamos con *rb* mirando al mismo objeto que antes, pero como lo que es, de forma que podemos invocar el método que define Arquero, pero no su clase madre.

Siempre se puede convertir un puntero de una clase hija a un puntero de la clase madre. Se trata meramente de ver el mismo objeto mediante dos punteros distintos: con el puntero a la clase madre vemos sólo la parte que heredó y no la nueva que agregó la subclase.

Teniendo esto en claro, volvamos a nuestro ejemplo:

main.cpp

```
#include "Caballero.h"
#include "Arquero.h"
#include "Paladin.h"


void main(int argc, char* argv[])
{
	Caballero *robin_hood;

	robin_hood= new Arquero();

	robin_hood->recibirGolpe();

	return;
}
```

Aquí estamos apuntando a un Arquero con un puntero del tipo de su superclase Caballero. Como el método *recibirGolpe()* es un método que está definido en Caballero, lo podemos invocar con el puntero *robin_hood*.

Ahora, la pregunta lógica que surge es: ¿Qué método se invoca realmente?, ¿el de la clase madre o el de la clase hija?, dado que la función se implementó en las dos y se comporta de distinta manera en ambas.

El método que se invoca depende de cómo fue definida la función en la superclase. Nosotros la habíamos definido de la siguiente manera:

Caballero.h

```
class Caballero{
...
public:
	...
	void recibirGolpe();
	...
};
```

De este modo, el método que se invocará es el de la clase correspondiente al puntero, es decir, el de la clase madre (Caballero).

Si, en cambio, queremos que, a pesar de que estemos utilizando un puntero a la clase madre, se llame el método dependiendo del tipo verdadero del objeto, debemos utilizar la cláusula *virtual.*

Si redefinimos el método de la manera que se observa a continuación, mas allá de que usemos un puntero a la clase madre, se llamará el método de la clase hija. En nuestro ejemplo, se nominará *recibirGolpe()* de la clase Arquero:

Caballero.h

```
class Caballero{
...
public:
	...
	virtual void recibirGolpe();
	...
};
```

Entonces, si en main hacemos lo siguiente, estaremos restando 5 puntos de salud y no sólo uno, que es lo que sucedería si no utilizáramos *virtual:*

main.cpp

```
#include "Caballero.h"
#include "Arquero.h"
#include "Paladin.h"


void main(int argc, char* argv[])
{
	Caballero *robin_hood;

	robin_hood= new Arquero();

	robin_hood->recibirGolpe();

	return;
}
```

½. 7. 2. Clases abstractas

Una clase abstracta es una clase de la cual heredaremos otras clases, pero nunca será instanciada. Es decir, nunca tendremos un objeto del tipo de la superclase.

En nuestro caso, si declaramos Caballero como abstracta, eso se traduce en que nunca podríamos crear un objeto de tipo Caballero, sino sólo clases derivadas de la misma, como Arquero y Paladin.

Para que una clase se considere abstracta, al menos uno de sus métodos debe ser declarado como *virtual puro*.

Virtual puro significa que sólo se realiza la declaración del método y no su implementación, por lo que las clases derivadas deben proveer implementación. Para declarar un método como virtual puro, simplemente igualamos a cero su declaración. Esto hace que no se pueda crear una instancia de Caballero, convirtiéndose en una clase abstracta, la cual sólo se puede utilizar para actuar como superclase:

Caballero.h

```
class Caballero{
...
public:
	...
	virtual void recibirGolpe()=0;
	...
};
```