Lenguajes de Programación Orientada a Objetos

Herencia

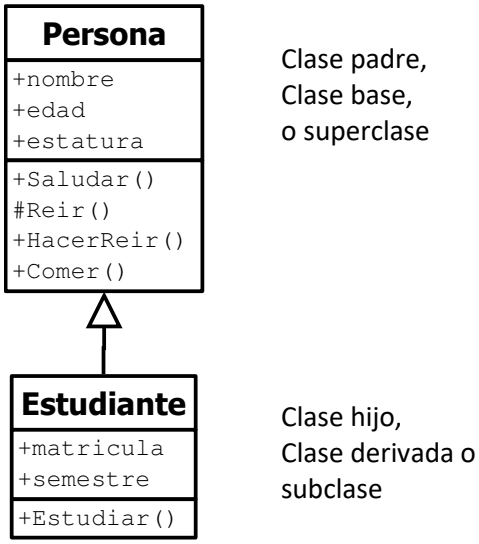
Hemos visto que un objeto es una entidad con características, identidad y estado, o una entidad con atributos y comportamiento. Una clase es un conjunto de objetos con características y semántica en común, y en programación utilizamos una clase para modelar un grupo de objetos, es decir, nosotros definiremos las características que tendrán ciertos objetos por medio de la creación de clases. Por ejemplo, podemos crear una clase que defina las características que tendrán los objetos de tipo Persona, creando una clase llamada Persona y declarando en ella que comportamiento y atributos que consideramos que debe tener una persona y que sean útiles para nuestro programa o solución.

Una de las herramientas principales de la Programación Orientada a Objetos es la herencia. Por medio de herencia podemos generar una clase a partir de una clase que ya existe. La clase nueva hereda las características de la clase existente y puede agregar nuevas características o modificar comportamiento, siempre manteniendo la semántica de la clase original. Tomando como ejemplo la clase Persona, originalmente en esta clase definimos algunos atributos y comportamiento:

Podemos tomar como base la clase Persona y crear una clase a partir de esta. La nueva clase será para objetos que presenten las mismas características que una persona, pero para los cuales sea necesario definir características o comportamiento adicionales, o comportamiento modificado. Por ejemplo, si en nuestro programa manejaremos estudiantes, podemos crear la clase Estudiante a partir de Persona, ya que un estudiante, siendo persona, presenta las mismas características, pero además presenta algunos atributos adicionales. En la clase Persona, declaramos *nombre, edad* y *estatura* como atributos, además, declaramos que una persona puede *saludar, reír, comer* y podemos *hacerla reír*. El estudiante presentara estas mismas características, además, como atributos adicionales todo estudiante tiene un numero de *matricula* y cursa determinado *semestre* de su carrera o plan de estudios. Como comportamiento adicional, un estudiante debe *estudiar*.

En el paradigma de orientación a objetos, a la clase a partir de la cual creamos una clase nueva se le conoce como *clase padre*, y a la nueva clase que hereda las características de la clase padre se le conoce como *clase hijo*. También se les conoce como *clase base* y *clase derivada*, respectivamente (Persona es la clase base Estudiante y Estudiante la clase derivada de persona). Y considerando que las clases son

conjuntos, se les conoce como super clase y sub clase, respectivamente (Persona es la super clase de Estudiante, y Estudiante es una subclase de Persona). En diagramación UML (y en otros lenguajes de diagramación), la herencia se representa por medio de un triángulo cuya punta o cima apunta a la clase padre, y de la base parte una línea que lo une a la clase hijo.



El símbolo de herencia se considera una relación “es un” o “es una”, y se lee de la clase derivada hacia la clase base. En el ejemplo anterior se leería como “Estudiante es una Persona”.

En C++, la sintaxis para declarar una clase que se deriva de una clase base es la siguiente:

```
class ClaseDerivada :[especificador_de_acceso] ClaseBase{
      //atributos y comportamiento
}
```

La clase derivada heredará los elementos de la clase base, pero aún se considera el tipo de acceso declarado para estos elementos. Recordemos que los elementos de tipo *private* solo pueden ser accedidos directamente en la clase en la cual se declararon, por lo tanto, si en la clase base un elemento fue declarado como *private*, este se hereda a la clase derivada pero en el código de esta última clase no se podrá tener acceso directo al elemento, pero podría hacerlo de forma indirecta por medio de una operación (si *nombre* es *private*, se puede modificar con SetNombre()). Los elementos declarados como protected en la clase padre, si podrán ser accedidos directamente en la clase hijo, y no podrán ser accedidos en código ajeno a la estructura de herencia de la clase padre.

Un elemento opcional en la declaración de una clase derivada es el especificador de acceso. Este modificará como se considerara el acceso para los elementos heredados. Si el especificador de acceso

es *private*, en la clase derivada todos los elementos heredados se volverán *private*, por lo que clases derivadas de la subclase recién declarada no podrán tener acceso directo a ellos. Si el especificador de acceso es *protected* todos los elementos heredados en la clase derivada se convertirán en *protected*. Si el especificador de acceso es *public,* en la clase derivada no se modificará el tipo de acceso para los elementos heredados, es decir, los miembros *protected* heredados seguirán siendo *protected*, y los miembros *public* seguirán siendo *public*. Si el tipo de acceso no es especificado al derivar la clase, por default se considera *private*, por lo que hay que poner atención en especificarlo si no queremos convertir todos los elementos heredados en *private.*

También es conveniente notar que, si hacemos una clase la cual planeamos usar como clase base de otras clases, o queremos dejar abierta la posibilidad de usarla como base, es conveniente usar acceso *protected* para elementos que no queremos que estén disponibles para el exterior de la clase.

Suponiendo que en C++ tenemos la declaración de la clase persona como sigue:

```
class Persona
{
protected:
	string nombre;
	int edad;
	float estatura;
	void Reir();
public:
	Persona();
	Persona(string nombre);
	void SetNombre(string nombre);
	void SetEdad(int edad);
	void SetEstatura(float estatura);
	void Saludar();
	void Comer();
	void HacerReir(int motivo);
	~Persona();
};
```

La declaración de la clase Estudiante, que se deriva de Persona queda como sigue:

```
class Estudiante :public Persona{
protected:
	int matricula;
	int semestre;
public:
	Estudiante();
	Estudiante(string nombre, int matricula);
	 int GetMatricula();
	 int GetSemestre();
	void Estudiar();
	~Estudiante();
};
```

Lenguajes de Programación Orientada a Objetos

Constructores en clases derivadas

Cada clase puede tener uno o más constructores, que harán inicialización de los recursos necesarios para el objeto cuando este sea creado. Cuando una clase esta derivada de otra, antes de ejecutarse el código de su constructor se ejecuta automáticamente el código del constructor en la clase padre, si esta cuenta con uno. El llamado al constructor en la clase padre puede ser implícito o explicito. Si no especifica explícitamente el constructor en clase padre a ejecutar, se ejecuta el constructor default o constructor sin parámetros. Si la clase tiene constructores con parámetros pero no tiene un constructor sin parámetros, es obligatorio hacer explicito cual constructor se ejecutara y los valores para sis parámetros.

Para el ejemplo de la clase Estudiante, cuando antes de que se ejecute el código de la función Estudiante() se ejecutara el código de Persona::Persona(). De igual forma, antes de que se ejecute el código de la función *Estudiante(string nombre, int matricula)*, se ejecutara el código de Persona::Persona(). Si la clase persona no contara con el contructor anteriormente mencionado, entonces será necesario indicar explícitamente que se ejecutara su constructor Persona::Persona(string nombre), indicando también el valor de sus argumentos.

El constructor a ejecutar se debe especificar en la implementación de la función. Debe hacerse de la siguiente forma:

```
ClaseDerivada::ClaseDerivada([parámetros]):ClaseBase([parámetros]){
   Código de constructor
}
```

Una vez que se indica la función a implementar, y antes de iniciar su bloque de código (antes de la llave que abre, se deben poner : seguido de la llamada al constructor en clase padre, indicando argumentos si es necesario.

Por ejemplo, en la clase Estudiante existe un constructor que recibe el nombre y la matricula como parámetros. Al ejecutarse ese constructor seria conveniente que se inicializara el nombre por medio del constructor correspondiente en Persona. Entonces, en la implementación de la clase estudiante (Estudiante.cpp) implementaríamos el constructor de la siguiente forma:

```
Estudiante::Estudiante(string nombre,int matricula):Persona(nombre){
    this->matricula=matricula;
}
```

En este ejemplo, se ejecutara Persona::Persona(string nombre) usando como argumento la variable nombre recibida como parámetro en el constructor de Estudiante.

Cuando la clase padre se deriva de otra clase, ocurre lo mismo. Antes de ejecutar el código de su constructor se ejecuta un constructor de su clase base. Entonces, el llamado de constructores se hace recursivamente hasta que se ejecute el constructor de la clase en la raíz de la jerarquía de clases del objeto que se está creando.

Sobreescritura de funciones (métodos y operaciones)

Una clase tiene un comportamiento que esta programado en sus funciones miembro (métodos y operaciones). Al derivar la clase estamos haciendo una versión especializada de la clase base, con atributos adicionales (si son necesarios) y comportamiento adicional.

El comportamiento que ya fue programado en la clase puede ser modificado en una clase derivada. Para hacer eso será necesario implementar nuevamente en la clase derivada dicho comportamiento para esta tenga su propia versión del comportamiento. En C++ es posible hacer esto declarando nuevamente la función o funciones correspondientes e implementándolas. A esta acción se le llama *sobrescritura de funcione*s u *override* (en ingles). También se puede llamar *reimplementación*, aunque este termino casi no se utiliza.

En el ejemplo de la clase Estudiante, esta clase heredo un comportamiento de su clase padre, la clase Persona. Este comportamiento incluye la función Saludar(), todas las personas saludaran utilizando el código implementado en esa función. Sin embargo, podemos considerar que los estudiantes no saludan como las demás personas, pero deben de saludar. Entonces será necesario hacer una sobreescritura de la función Saludar().

```
class Estudiante :public Persona{
protected:
	int matricula;
	int semestre;
public:
	Estudiante();
	Estudiante(string nombre, int matricula);
	 int GetMatricula();
	 int GetSemestre();
	void Estudiar();
	void Saludar(); //<- sobreescritura de la función heredada
	~Estudiante();
};
```

Y será necesario implementar la función Estudiante::Saludar(). Entonces, los objetos de tipo estudiante tendrán su propia versión que sobrescribe a la función Persona::Saludar(). Es importante tener en cuenta que para que sea considerado sobreescritura, la nueva función debe llamarse exactamente igual y debe recibir los mismos tipos de parámetro que en la clase padre, es decir, su prototipo debe ser idéntico. De otra forma, si el nombre es diferente será una función diferente y si el nombre es el mismo

se considerará que es una versión alterna a la función heredada y que se está haciendo sobrecarga, y no sobreescritura.

Otro aspecto importante a considerar es declarar las funciones que podrán ser sobrescritas como "sobrescribibles", para que la sobrescritura se haga correctamente. Para hacer esto, será necesario utilizar el modificador *virtual* en la declaración de la función miembro de clase. Esto indica al compilador que se aplicará el mecanismo de sobrescritura para esa función, si en el objeto para la cual se invoca existe una función que sobrescribe. Si se utiliza el modificador *virtual* y la función nunca es sobrescrita en alguna clase derivada, no tiene ninguna reacción secundaria, es decir, no importa si la función es declarada virtual y nunca se sobrescribe. Entonces, al diseñar una clase debemos tener en cuenta que esa clase se podría usar en un futuro para derivar clases y seria conveniente declarar las funciones que podrían comportarse diferente como virtuales. Si tenemos la seguridad de que alguna función será sobrescrita entonces seria obligatorio declararla virtual para que funcione correctamente nuestro programa. Por ejemplo, en la clase persona, modificar el nombre, edad y estatura no debería ser diferente para ninguna persona, sin importar el tipo, pero si podría comer diferente, saludar diferente o reír diferente dependiendo del tipo de persona que se trate, por lo que es conveniente declarar esas funciones con el modificador *virtual*, como se muestra en el código a continuación. El modificador se utiliza solamente al declarar la clase, no se utiliza en la implementación.

```
class Persona
{
protected:
	string nombre;
	int edad;
	float estatura;
	virtual void Reir();
public:
	Persona();
	Persona(string nombre);
	void SetNombre(string nombre);
	void SetEdad(int edad);
	void SetEstatura(float estatura);
	virtual void Saludar();
	virtual void Comer();
	virtual void HacerReir(int motivo);
	~Persona();
};
```

Al hacer la sobreescritura de una función, podemos caer en el error de escribir de forma incorrecta el prototipo de la función en la clase derivada. Por ejemplo, para la función Persona::Saludar(), la sobreescritura en la clase estudiante es la función Estudiante::Saludar(), pero tenemos la probabilidad de cometer el error de nombrarla Estudiante::saludar(), esta última función tiene el nombre diferente, por lo que la clase Estudiante tendrá una función Saludar() (heredada de Persona) y una función saludar(). Para evitar este tipo de error, podemos usar el especificador *override* al declarar la función

que sobre escribe. Al encontrar este especificador, el compilador verifica si existe una función virtual heredada con el mismo prototipo, y en caso de que no exista dará como resultado un error. Adicionalmente, cuando, posteriormente, veamos la declaración de la clase derivada sabremos que esa función sobre escribe a una heredada al ver este especificador. El uso del especificador *override* no es obligatorio, pero resulta útil y conveniente para el programador. La declaración de la clase Estudiante, considerando este especificador, queda como sigue:

```
class Estudiante :public Persona{
protected:
	int matricula;
	int semestre;
public:
	Estudiante();
	Estudiante(string nombre, int matricula);
	 int GetMatricula();
	 int GetSemestre();
	void Estudiar();
	void Saludar() override; //<- sobreescritura de la función heredada
	~Estudiante();
};
```

Es probable que en una versión sobre escrita de una función, o en la clase que la contiene, sea necesario o conveniente ejecutar la función heredada (de la clase padre), y complementarla con algún código. Esto es posible recurriendo al operador de resolución de ámbito. Si en la clase Estudiante se hace llamado a Saludar(), se estará refiriendo a Estudiante::Saludar(), pero si queremos usar la versión heredada de Persona, debemos hacerlo especificando que es la función perteneciente a Persona mediante el operador de resolución de ámbito (Persona::Saludar()).

```
//implementación de version sobrescrita de Saludar
void Estudiante::Saludar(){
	 Persona::Saludar(); //invocamos la vesion de persona, y agregamos
codigo complementario
     cout<<" y soy estudiante…"<<endl;
}
```

Herencia Múltiple

Es posible crear una clase derivada de más de una clase padre. Al usar varias clases padres, se están heredando características de múltiples clase.

El problema de herencia múltiple es que no se puede establecer una clase base dominante, y se puede prestar a ambigüedades. Si más de una clase padre tiene un método con el mismo nombre, ¿cual es el

que se hereda o se prefiere al invocar el método como miembro en la clase hijo? Por este motivo, algunos lenguajes no implementan herencia múltiple.

En C++ si es posible, indicando las diferentes clases padre separadas por coma.

```
class ClaseDerivada :[especificador_de_acceso] ClaseBase1,
                     [especificador_de_acceso] ClaseBase2{
       //atributos y comportamiento
}
```

Por ejemplo

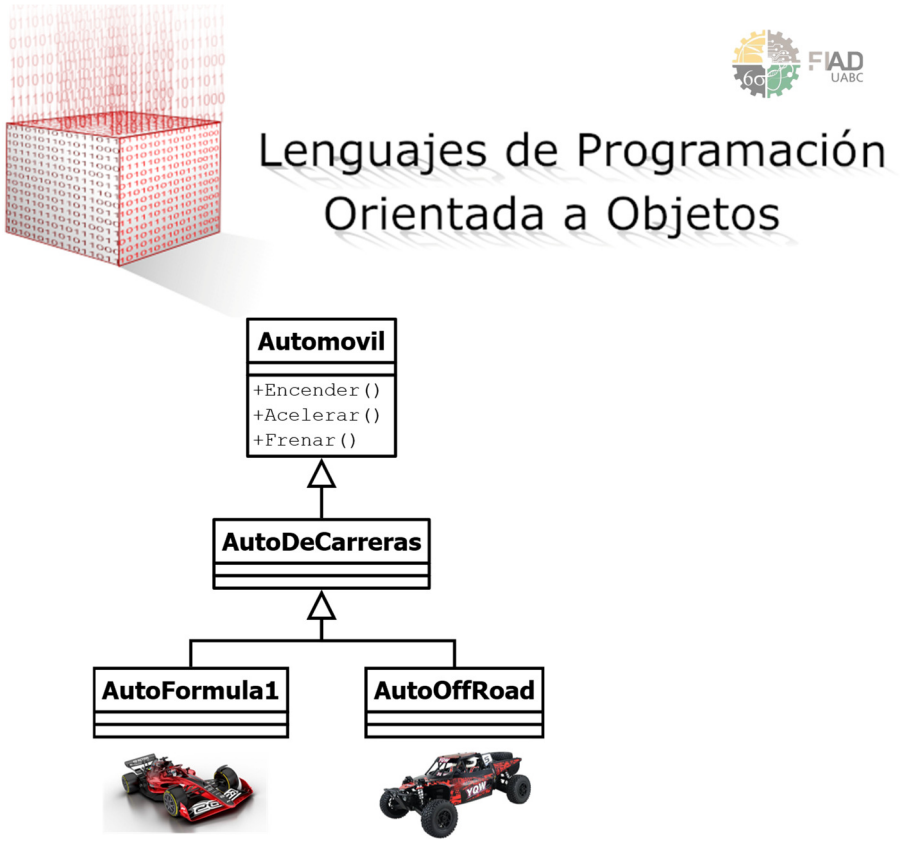
```
class Mexicano:public Persona, public Cantante{

///declaración de características del mexicano, que es persona y cantante.

}
```

Conceptos relacionados con herencia

La herencia esta ligada a varios conceptos o características, las cuales, a su vez, están ligadas entra si. A continuación, se revisan algunos de ellos.

Especialización

Este concepto es una de sus características principales en la herencia. En términos generales, la especialización se refiere partir de lo general hacia lo particular. Básicamente, si tenemos una clase que presenta características generales de un conjunto de objetos, a partir de ella podemos crear subclases que presenten características particulares, es decir, especializas la función o rol de la clase padre. Por ejemplo, si tenemos una clase que represente automóviles (clase Automóvil), esta clase va a tener las características que todo automóvil tiene (atributos y comportamiento), pero pueden existir subclases de automóvil que presentan un comportamiento especializado, por ejemplo, un auto de carreras se caracteriza por alcanzar mayores velocidades que el promedio de los autos. El auto de carreras será una subclase de automóvil y especializa su comportamiento y su función. Dentro de los autos de carreras podríamos tener los autos de carreras Formula 1, que tienen características muy particulares, y por otro lado los autos de carreras Off-road, cuyas características son diferentes al de Formula 1, pero siguen siendo del conjunto de autos de carreras.



Al implementar las clases en el diagrama, es necesario considerar que algunas operaciones se pudieron haber omitido y que se considera que serán reimplementadas en las clases derivadas.

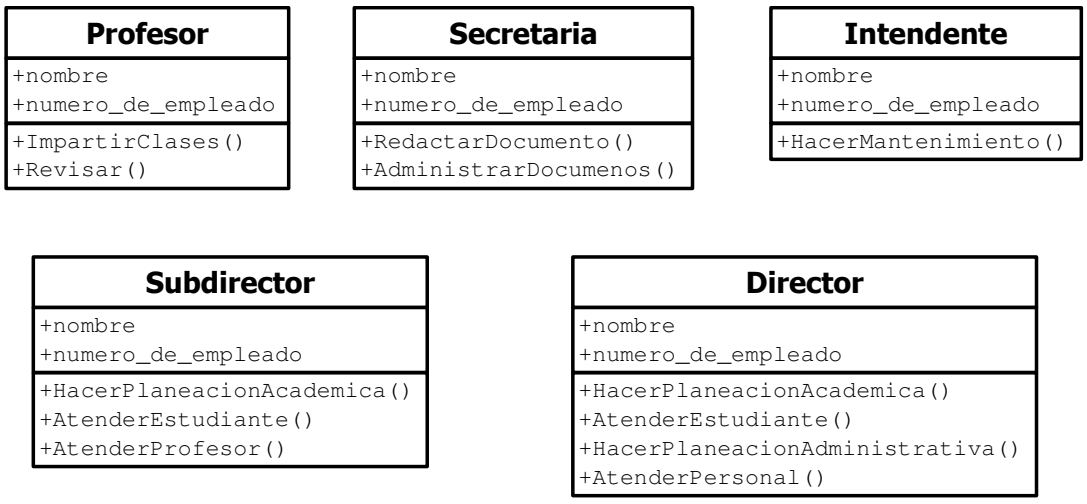
```
class Automovil{

	float velocidad=0.0f;
public:
	virtual void Encender();
	virtual void Acelerar(float cantidad); //Se puede sobrescribir
	virtual void Frenar();
};

class AutoDeCarreras:public Automovil{

public:
	virtual void Encender() override;
	virtual void Acelerar(float cantidad) override;
	virtual void Frenar() override;
};

class AutoFormula1:public AutoDeCarreras{

public:
	void Encender() override;
	void Acelerar(float cantidad) override;
	void Frenar() override;
};
```

```
class AutoOffRoad : public AutoDeCarreras{

public:
	void Encender() override;
	void Acelerar(float cantidad) override;
	void Frenar() override;
};
```

Generalización o abstracción

La abstracción es lo contrario a la especialización. En la abstracción partimos de diferentes clases que tienen algún comportamiento particular, pero que presentan características en común y que semánticamente también tienen cosas en común. Considerando estas cosas que tienen en común, se puede crear una clase que las contenga y que generalice las características comunes de las otras clases. Esta clase sería una clase abstracta, que generaliza a otras. Entonces, las clases con comportamiento particular deben ser derivadas de la clase abstracta, y se pueden omitir los atributos que ya fueron considerados en ella. El comportamiento puede ser generalizado en la clase abstracta, pero implementado de forma especifica en las clases derivadas.

Por ejemplo, consideremos que tenemos las clases Profesor, Secretaria, Intendente, Director y Subdirector.



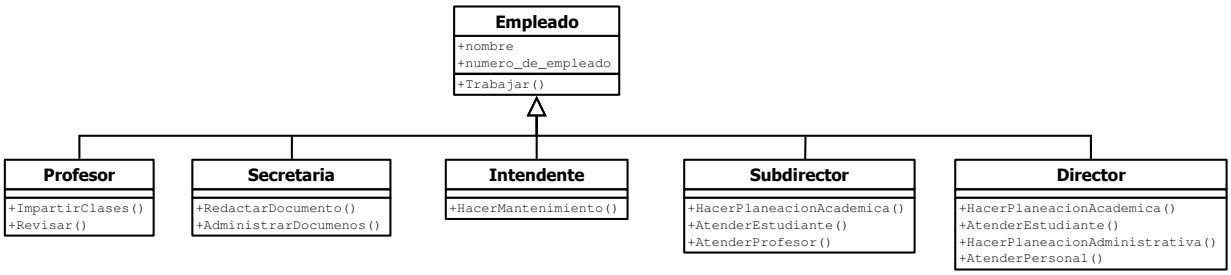
Su declaración podría ser la siguiente, tomando en cuenta funciones “getter” para los atributos, y un constructor para su inicialización, y, aunque el diagrama indica que los atributos son públicos, se declararon privados.


```
class Profesor{
 private:
         string nombre;
         int numero_de_empleado;
 public:
        Profesor(string nombre, int num_empleado);
        string GetNombre();
        int GetNumeroDeEmpleado();
        void ImpartirClases();
        void Revisar();
};

class Secretaria{
private:
        string nombre;
        int numero_de_empleado;
public:
        Secretaria(string nombre, int num_empleado);
        string GetNombre();
        int GetNumeroDeEmpleado();
        void RedacarDocumento();
        void AdministrarDocumntos();
};

class Intendente{
private:
        string nombre;
        int numero_de_empleado;
public:
        Intendente(string nombre, int num_empleado);
        string GetNombre();
        int GetNumeroDeEmpleado();
        void HacerMantenimiento();
};
```

```
class Subdirector{
private:
        string nombre;
        int numero_de_empleado;
public:
        Subdirector(string nombre, int
num_empleado);
        string GetNombre();
        int GetNumeroDeEmpleado();
        void HacerPlaneacionAcademica();
    void AtenderEstudiante();
        void AtenderProfesor();
};

class Director{
private:
        string nombre;
        int numero_de_empleado;
public:
        Director(string nombre, int
num_empleado);
        string GetNombre();
        int GetNumeroDeEmpleado();
        void HacerPlaneacionAcademica();
        void HacerPlaneacionAdministrativa();
        void AtenderEstudiante();
        void AtenderPersonal();
};
```

Los objetos de las clases anteriores tendrán atributos en común (nombre y numero de empleado), y semánticamente comparten una característica: trabajan para la universidad o una escuela. Podemos tomar esa característica compartida para diseñar una clase abstracta o que las generalice. Podemos pensarlos como Empleados de la universidad o escuela, o simplemente Empleados. Todo empleado tendrá nombre y número de empleado. Las acciones que hacen los objetos de las clases anteriores es trabajo que hacen para la universidad, entonces, su rol actividad para la universidad es trabajar. Nuestra clase abstracta (la llamaremos Empleado) entonces tendrá los atributos nombre y numero_de_empleado, y tendrá la operación o método Trabajar(). De esta clase se derivarán las clases que teníamos anteriormente, estas heredaran los atributos de Empleado, por lo que omitiríamos la declaración de los atributos que actualmente están declarados. Heredaran el método Trabajar(), pero cada quien lo hará de forma especial e implementara su propia versión, basada en el comportamiento que ya tienen, por ejemplo, cuando el profesor trabaje, impartirá clases y revisara tareas y trabajos, cuando el intendente trabaje hará mantenimiento de las instalaciones. Un diagrama de clases modificado es el siguiente:

Lenguajes de Programación Orientada a Objetos



La declaración de las clases correspondientes al diagrama anterior es la que se muestra a continuación. Hay que considerar que los atributos serán heredados de Empleado a las clases derivadas, y en esa ocasión se declaran *protected*, para que las subclases de Empleado puedan tener acceso directo a ellos.

```
//Clase abstracta
class Empleado{
protected:
        string nombre;
        int numero_de_empleado;
public:
        Empleado(string nombre, int num_empleado);
        string GetNombre();
        int GetNumeroDeEmpleado();
        virtual void Trabajar();
};
//Subclases
class Profesor:public Empleado{
  public:
        Profesor(string nombre, int num_empleado);
        void Trabajar() override;
        void ImpartirClases();
        void Revisar();
};

class Secretaria:public Empleado{
public:
        Secretaria(string nombre, int num_empleado);
        void Trabajar() override;
        void RedacarDocumento();
        void AdministrarDocumntos();
};

class Intendente:public Empleado{
public:
        Intendente(string nombre, int num_empleado);
        void Trabajar() override;
        void HacerMantenimiento();
};
```

```
//Subclases
class Subdirector :public Empleado{
public:
        Subdirector(string nombre, int
num_empleado);
        void Trabajar() override;
        void HacerPlaneacionAcademica();
    void AtenderEstudiante();
        void AtenderProfesor();
};

class Director :public Empleado{
public:
        Director(string nombre, int
num_empleado);
        void Trabajar() override;
        void HacerPlaneacionAcademica();
        void HacerPlaneacionAdministrativa();
        void AtenderEstudiante();
        void AtenderPersonal();
};
```

Lenguajes de Programación Orientada a Objetos

Polimorfismo

El polimorfismo es un concepto muy recurrido en POO y que también puede ser mal entendido. La palabra polimorfismo significa múltiples formas. A menudo se interpreta como que un objeto puede tener muchas formas o cambiar de forma, esta idea es equivocada ya que un objeto nunca cambia de forma. Una vez que se crea una instancia, el objeto creado no puede cambiar de clase, pertenecerá siempre a la clase en la cual fue creado.

El polimorfismo, o las múltiples formas a la que se refiere la palabra se presenta en 2 sentidos. El primero es en referencia al conjunto de objetos que pertenecerían a una clase. A una clase pertenecen muchos objetos los cuales presentan las características y semántica que se consideraron al modelar la clase, pero dentro de esta clase puede haber múltiples formas diferentes, esto es gracias a la herencia. Dentro de una clase puede haber objetos de la clase con diferente especialización, es decir, las instancias de clases derivadas se consideran elementos del conjunto de objetos que considera la clase padre. Por ejemplo, considerando las clases definidas para el ejemplo de abstracción (Empleado, Profesor, etc.), un profesor, además de ser elemento de la clase Profesor, se considera elemento del conjunto de empleados o de la clase empleado. Lo mismo ocurre con una secretaria, es elemento de la clase Empleados. Así, la clase empleado puede tener múltiples formas (profesores, secretarias, intendentes, etc). Entonces, si hacemos referencia a un objeto de tipo empleado, este objeto puede ser de cualquiera de las subclases de empleado, teniendo claro que el objeto nunca cambia de forma, solo existe la posibilidad de que el objeto empleado sea de alguna de las formas posibles.

En C++, es necesario el uso de apuntadores o referencias para poder hacer el tipo de manejo de objetos. Si tenemos un apuntador de tipo Empleado, a este se le puede asignar la dirección de memoria de una instancia de cualquier clase derivada de Empleado.

```
Empleado *empleado;
Empleado=new Secretaria();
```

Tomando en cuenta esto, se puede hacer una función que reciba un empleado como parámetro y escribir el código en términos de Empleado. Esta función entonces podrá recibir cualquier objeto de clase derivada de Empleado, y no hay necesidad de hacer una versión de la función para cada tipo de empleado.

```
void UtilizarEmpleado(Empleado *empleado){
     empleado->Trabajar();
}

int main(){
     Profesor el_profe("Victor Torres",223111);
     UtilizarEmpleado(&el_profe);
}
```

Es importante considerar que cuando un objeto de subclase es manejado como objeto de superclase, los elementos que especializan la subclase no están accesibles. Si tenemos un objeto manejado como Empleado, sabemos que podrá trabajar por que es lo que hace un empleado, pero no se sabrán detalles específicos de su instancia, si es Profesor no se tendrá acceso a Revisar() u otras funciones propias de Profesor.

El otro sentido, que es uno al cual se hace referencia en bibliografía, es que los objetos pueden reaccionar de forma diferente a un mismo mensaje. Visto de otra manera, los objetos pueden hacer la misma cosa de formas diferentes. La forma en la que lo harán depende de su especialización. Por ejemplo, si tengo un objeto Empleado, y envio un mensaje para que trabaje (invocando la función Trabajar()), la acción que se desarrollara depende de la subclase a la cual pertenece el objeto, es decir, del tipo con el que fue creada de la instancia. Por ejemplo, en la función

```
void UtilizarEmpleado(Empleado *empleado){
     empleado->Trabajar();
}
```

Lo que desarrollara el objeto que se recibe como parámetro dependerá del tipo de objeto del cual se trate. Si es un intendente, hará mantenimiento (si así se programó), si es un profesor impartirá clases y revisará. Es una reacción diferente en base al mismo mensaje (trabajar).

Es importante recordar y que para que funcione correctamente una función sobrescrita es necesario utilizar el modificador *virtual* en la clase padre. Esto permitirá que se aplique la sustitución correspondiente, de acuerdo al principio de Liskow. Si se omite la modificación con *virtual*, en la función utilizar empleado siempre se ejecutara la versión Empleado::Trabajar(), no importa si la instancia recibida como argumento tiene su propia versión.

Funciones virtuales puras y clases abstractas

Cuando se diseña un programa es conveniente considerar el polimorfismo. Este permitirá una flexibilidad en mantenimiento y permitirá que se agreguen nuevos módulos sin necesidad de hacer ajustes mayores a la estructura del programa. En el diseño existirán clases abstractas que ayudaran a la implementar el polimorfismo. En estas clases existirán métodos y operaciones que no necesariamente tienen un comportamiento definido al nivel de generalización al que esta la clase abstracta. Por ejemplo, la clase Empleado declara una operación trabajar, pues todo empleado trabajara, pero no se puede decir a ese nivel de abstracción como trabajara el empleado. Es necesario saber que tipo de empleado es para decir que acciones desarrollara al trabajar. Dado a que no es posible decir que hará para trabajar el empleado, pero es necesario declarar la función con propósitos de polimorfismo, esta función puede declararse como virtual pura. La función virtual pura no tiene código, solo esta declarada pero su código

no existe, y debe ser sobrescrita en una subclase para ejecutar las acciones correctas de acuerdo a la semántica de la subclase. Para declarar la función virtual pura, solo es necesario declararla con el modificador virtual e igualarla a 0, no debe ser implementada.

A continuación, se declara como virtual pura la función Trabajar del empleado.

```
class Empleado{
protected:
      string nombre;
      int numero_de_empleado;
public:
      Empleado(string nombre, int num_empleado);
      string GetNombre();
      int GetNumeroDeEmpleado();
      virtual void Trabajar()=0;
};
```

Cuando una clase tiene al menos una función virtual pura se conoce como clase abstracta, y no se pueden crear instancias de la misma, pero se pueden crear instancias de clases derivadas que si implementen la función.

Si la clase Empleado fue declarada como en el ejemplo anterior, el siguiente código produce error:

```
Empleado el_empleado("Juan",12); //creación estática
Empleado *el_empleado2=new Empleado("Sosimo", 15);//creación dinámica
//Error por que no se pueden crear instancias. Que codigo se ejecutara si se
//invoca el_empleado.Trabajar()?
```

Si la clase Profesor se deriva de la clase Empleado, se debe poner atención en que se debe sobrescribir la función Trabajar() e implementarla. De otra forma, se hereda como virtual pura y Profesor también será clase abstracta. Si Profesor implementa Trabajar(), el siguiente codigo si es valido.

```
Empleado *el_empleado=new Profesor("Victor Torres",12345);
```

Conversión e identificación del tipo de instancias.

Como se mencionó previamente, cuando un objeto de subclase es manejado como objeto de superclase, los elementos que especializan la subclase no están accesibles. Si utilizamos un apuntador de Empleado para manejar un profesor, no tendremos acceso a las operaciones Revisar() e ImpartirClase(), solo a Trabajar() por que esta declarada como miembro de Empleado. Por ejemplo, en el siguiente codigo

```
Empleado *el_empleado=new Profesor("Victor Torres",12345);
el_empleado->ImpartirClases();
el_empleado->Revisar();
```

aunque la instancia asignada a el apuntador el_empleado es un profesor, habrá un error al invocar el_empleado->ImpartirClase(), ya esta función no esta declarada como miembro de Empleado. Esto se puede resolver convirtiendo el apuntador con el cual se maneja la instancia al tipo apropiado, con el operador de conversión de tipo (casting) de C:

```
Profesor *el_profesor=(Profesor *)el_empleado;
el_profesor->Revisar(); //revisar si es miembro de profesor, el codigo es valido
```

Es conveniente hacer la conversión con el operador *dynamic_cast*, para preservar correctamente el polimorfismo.

```
Profesor *el_profesor=dynamic_cast<Profesor *>(el_empleado);
```

Para hacer la conversión es conveniente asegurarse de que la instancia asignada al apuntador corresponde al tipo al cual se convertirá. Por ejemplo, en el siguiente código

```
void UtilizarEmpleado(Empleado *empleado){

     Profesor *prof=(Profesor *)empleado; // se convierte a Profesor
     prof->ImpartirClase();
}
```

¿Qué sucedería si en lugar de una instancia de Profesor, esta asignada una instancia de Secretaria en el apuntador empleado? Si se hace la conversión y se invoca la operación ImpartirClase(), dado a que Secretaria no tiene esa operación, el resultado es indeterminado, y probablemente exista un error en la ejecución por una segmentación de memoria. Se puede verificar el tipo utilizando el operador *typeid*, este devuelve un objeto de clase *type_info*, que tiene la información sobre el tipo del objeto consultado. Uno de sus funciones miembro es name(), que devuelve una cadena con el nombre del tipo consultado. El siguiente código verifica adecuadamente el tipo antes de convertir

```
void UtilizarEmpleado(Empleado *empleado){

      //verificamos el tipo del contenido en el apuntador
     type_info info= typeid(*empleado);
     if(info.name()=="class Profesor"){
      //si es de clase Profesor, hacemos la conversión de tipo y ejecución de función
        Profesor *prof=(Profesor *)empleado; // se convierte a Profesor
        prof->ImpartirClase();
     }
}
```