**Programación Orientada a Objetos**

Parcial 3

Notas de curso

Dr. Ezequiel Arceo May

6 de mayo de 2019

# Índice

**Índice** **1**

**1. Concepto básicos** **2**

**2. Herencia** **3**

2.1. Herencia en C++ usando ': public' . . . . . 3

**3. Polimorfismo** **6**

3.1. Polimorfismo en C++ usando 'virtual' . . . . . 6

**4. Sobrecarga de operadores** **9**

4.1. Sobrecarga de `<<` . . . . . 11

4.2. Sobrecarga de operadores aritméticos `+,-,*,/` . . . . . 11

**5. Proyecto Final** **12**

«Un lenguaje de programación es una forma de expresarnos»
— Ezequiel Arceo —

# 1. Concepto básicos

Los conceptos más básicos que debemos entender en la Programación Orientada a Objetos (POO) son:

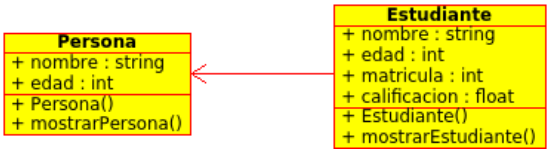
- Clase
- Objeto
- Abstracción
- Encapsulación
- Herencia
- Polimorfismo

**Jueves 2 de Mayo de 2019**

# 2. Herencia

La **herencia** sucede cuando una clase nueva se crea a partir de una clase existente, obteniendo (heredando) todos sus atributos y métodos.

A la clase de la cual se hereda se le llama *clase padre* o *súper clase*. A la clase que recibe la herencia se le llama *clase hija* o *subclase*.




Con la herencia nos ahorramos volver a definir los atributos y métodos de una clase. Basta con heredarlos de la clase padre y ya.

## 2.1. Herencia en C++ usando ‘: public‘

Para aprender a usar herencia en C++ seguiremos un pequeño ejemplo.

Primero **declaramos y definimos a la clase padre**, en esta caso será la clase **Persona**, con los atributos **nombre** y **edad**, y los métodos **Persona** y **mostrarPersona**:

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

class Persona{
  private:
    string nombre;
    int edad;
  public:
    Persona(string, int);
    void mostrarPersona();
};
Persona::Persona(string _n, int _e){
  nombre = _n;
  edad = _e;
}
void Persona::mostrarPersona(){
  cout << "Nombre: " << nombre << endl;
  cout << "Edad: " << edad << endl;
}
```

Seguidamente, **declaramos una clase hija que herede de la clase padre**, en este caso la clase **Estudiante**, con los atributos **nombre**, **edad**, **matricula** y **calificacion**, y los métodos **Estudiante** y **mostrarEstudiante**.

Especificamos la herencia escribiendo `:public Persona` entre el nombre de la clase **Estudiante** y el corchete de apertura de su declaración. Este fragmento de código indica que la clase **Estudiante** puede acceder a todo público de la clase **Persona**, sin necesidad de anteponer `Persona::`

```
class Estudiante: public Persona {
  private:
    int matricula;
    float calificacion;
  public:
    Estudiante(string,int,int,float);
    void mostrarEstudiante();
};
```

Ahora es el momento de usar la herencia, accediendo a los miembros de la clase padre desde la clase hija.

El constructor de la clase **Estudiante** tiene 4 atributos, dos de ellos establecidos por el constructor de la clase **Persona**, y solamente tenemos que establecer los atributos específicos de la clase **Estudiante**:

```
Estudiante::Estudiante(string _n, int _e, int _m, float _c): Persona(_n, _e){
  matricula = _m;
  calificacion = _c;
}
```

En el método **mostrarPersona** solamente tenemos que mostrar manualmente los atributos **matricula** y **calificacion**, y delegamos mostrar los demás a la clase padre:

```
void Estudiante::mostrarEstudiante(){
  mostrarPersona(); // uso método de la clase padre
  cout << "Matricula: " << matricula << endl;
  cout << "Calificacion: " << calificacion << endl;
}
```

Listo!, ahora ya podemos usar a la clase **Estudiante**

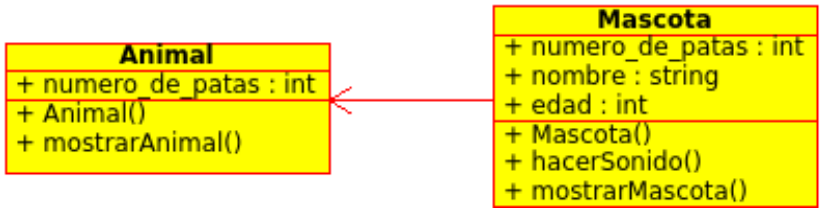
```cpp
int main(){
  Estudiante e1("Ezequiel Arceo May",34,123456,8.5);
  e1.mostrarEstudiante();
  system("pause");
  return 0;
}
```

### Actividad 1.

Aplica la herencia usando como clase padre a la clase **Persona** y como clase hija a la clase **Fanatico** con atributos **gustoPor** (de qué ámbito soy fanático) y **preferido** (cuál es mi entidad preferida), y los métodos **Fanatico** y **mostrarFanatico**.

**Trabajo B1. Entrega Antes de: Jueves 2 de Mayo, a las 23:59 horas.**

Implementa el siguiente diagrama de clases usando herencia




- Enviar por correo a `ezequiel_arceo@my.uvm.edu.mx`
- El asunto del correo será `POO_TrabajoB1_NOMBRE_DEL_AUTOR`
- El nombre de cada archivo adjunto será `POO_TrabajoB1_NOMBRE_DEL_AUTOR.*`

Ejemplo:
ASUNTO: `POO_TrabajoB1_JUAN_PEREZ_FERNANDEZ`
ADJUNTO: `POO_TrabajoB1_JUAN_PEREZ_FERNANDEZ.cpp`

# 3. Polimorfismo

El **polimorfismo** es la cualidad que tiene los objetos de responder de distintas formas al mismo mensaje.

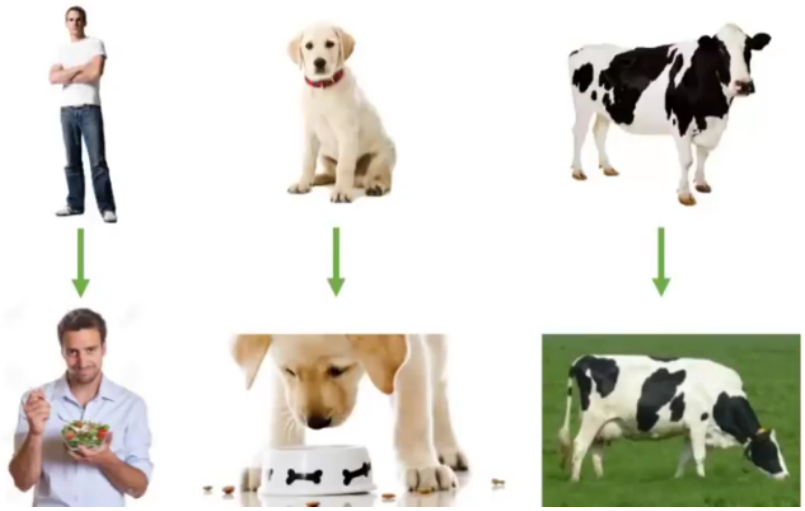


**Figura 1:** Considere objetos de las clases **Persona**, **Perro** y **Vaca**, cada uno de ellos puede **comer**, pero lo hacen de forma distinta.


## 3.1. Polimorfismo en C++ usando 'virtual'

En al sección anterior vimos que todas las clases implementadas tenían un método llamado **mostrarNombreDeLaClase**. Si quisiéramos indicar a cada elemento de una clase que simplemente se muestre, usando su método **mostrar** estaríamos ante la necesidad de aplicar polimorfismo.

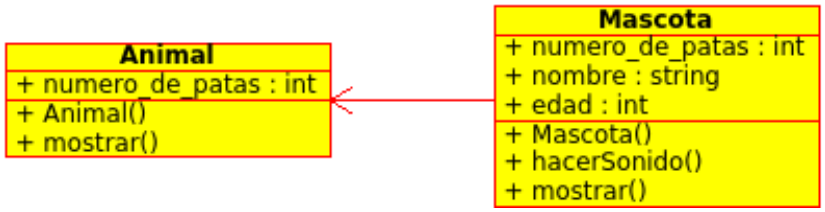
```cpp
#include <iostream>
#include <string>
using namespace std;

class Persona{
  private:
    string nombre;
    int edad;
  public:
    Persona(string, int);
    virtual void mostrar(); // Polimorfismo
};
Persona::Persona(string _n, int _e){
  nombre = _n;
  edad = _e;
}
void Persona::mostrar(){
  cout << "Nombre: " << nombre << endl;
  cout << "Edad: " << edad << endl;
}
class Estudiante: public Persona {
  private:
    int matricula;
    float calificacion;
  public:
    Estudiante(string,int,int,float);
    void mostrar(); // Polimorfismo
};
Estudiante::Estudiante(string _n, int _e, int _m, float _c): Persona(_n, _e){
  matricula = _m;
  calificacion = _c;
}
void Estudiante::mostrar(){
  Persona::mostrar(); // Polimorfismo
  cout << "Matricula: " << matricula << endl;
  cout << "Calificacion: " << calificacion << endl;
}
int main(){
  Persona persona("Chabelo",100);
  persona.mostrar();
  Estudiante alumno("Ezequiel Arceo May",34,123456,8.5);
  alumno.mostrar();
  return 0;
}
```

Como hemos visto en el ejemplo anterior (el mismo que en la sección previa), para usar polimorfismo anteponemos la palabra `virtual` a cada función polimórfica en la clase padre. Las clases hijas declaran funciones con el mismo nombre (pero sin `virtual`). Las clases hijas pueden llamar a las funciones polimórficas de sus clases padres usando el ámbito de clase, es decir `NombreDeLaClasePadre::nombreDelMetodo`.

**Trabajo B2. Entrega Antes de: Jueves 2 de Mayo, a las 23:59 horas.**

Implementa el siguiente diagrama de clases usando herencia y polimorfismo




- Enviar por correo a `ezequiel_arceo@my.uvm.edu.mx`
- El asunto del correo será `POO_TrabajoB2_NOMBRE_DEL_AUTOR`
- El nombre de cada archivo adjunto será `POO_TrabajoB2_NOMBRE_DEL_AUTOR.*`

Ejemplo:
ASUNTO: `POO_TrabajoB2_JUAN_PEREZ_FERNANDEZ`
ADJUNTO: `POO_TrabajoB2_JUAN_PEREZ_FERNANDEZ.cpp`

**Lunes 6 de Mayo de 2019**

# 4. Sobrecarga de operadores

Se habrá dado cuenta que los *tipos de datos primitivos* como **int**, **float**,**double**,**char**,**string** pueden ser impresos de forma trivial usando:

```
int mi_entero = 2;
cout << mi_entero;
float mi_decimal = 3.1416;
cout << mi_decimal;
string mi_cadena = "Hola";
cout << mi_cadena;
```

Sin embargo, los tipos de datos que definimos nosotros (nuestras clases) no pueden ser impresas de la misma forma, y hemos tenido que escribir un método **mostrarX** para cada clase **X**. Esto mejora un poco si usamos polimorfismo, ya que cada clase **X** puede tener un método **mostrar**.

```
#include <iostream>

using namespace std;

class Punto
{
  private:
    int x, y;
  public:
    Punto(int _x, int _y); // Constructor
    void mostrarPunto(); // QUEREMOS DEJAR DE HACER ESTO!
    int get_x(); // getter de x
    int get_y(); // getter de y
};

// Constructor por defecto
Punto::Punto(int _x, int _y) {
  x = _x;
  y = _y;
}
```

```cpp
// Getters
int Punto::get_x(){
  return x;
}
int Punto::get_y() {
  return y;
}
void Punto::mostrarPunto(){// QUEREMOS DEJAR DE HACER ESTO!
  cout << "(" << x << ", " << y << ")" << endl;
}

int main(){
  Punto p1 = Punto(2,4);
  p1.mostrarPunto(); // funciona
  cout << p1 << endl; // NO FUNCIONA, PERO QUISIÉRAMOS QUE SÍ
  return 0;
}
```

Resulta que `<<` (*operador de inserción de flujo*) y `>>` (*operador de extracción de flujo*) llamados **operadores**, no son más que funciones.

Si los operadores `<<` y `>>` son funciones, entonces se pueden sobrecargar para recibir como argumentos a instancias de nuestras clases. De igual forma se puede sobrecargar los operadores aritméticos (`+,-,*,/`), los operadores de comparación (`<,<=,>,>=,==,!=`), los operadores lógicos (`&&, ||, !`), entre muchos otros.

En esta sección aprenderemos con ejemplos como sobrecargar los operadores `<<,+,-,*`, y se dejará como ejercicio la sobrecarga de `/`.

Los operadores son funciones cuyo nombre empieza por la palabra reservada `operator` seguida del símbolo del operador. En el siguiente ejemplo se muestra al operador `+` usado como operador y como función:

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

int main(){
  string s1 = "Hola";
  string s2 = " mundo";
  cout << (s1 + s2) << endl;         // + como operador
  cout << operator+(s1,s2) << endl; // + como función
  return 0;
}
```

## 4.1. Sobrecarga de <<

Suponga que queremos hacer que la clase **Tipo** pueda ser impresa con `cout`, necesitamos escribir una función de la forma:

```
ostream& operator <<(ostream& output, Tipo mi_tipo){
  output << ... ;
  return output;
}
```

donde `...` es una secuencia de impresiones en función de los atributos de la clase `Tipo`.

Por ejemplo, para la clase `Punto` mostrada anteriormente podemos definir

```
ostream& operator <<(ostream& output, Punto punto){
  output << "(" << punto.get_x() << ", " << punto.get_y() << ")";
  return output;
}
```

Ahora ya podemos imprimir instancias de la clase **Punto** con `cout`!

```
int main(){
  Punto p1 = Punto(2,4);
  p1.mostrarPunto(); // funciona
  cout << p1 << endl; // FUNCIONA!
  return 0;
}
```

**Trabajo B3. Entrega Antes de: Martes 7 de Mayo de 2019 a las 23:59.**
Crea una clase llamada **Mascota** con los atributos **especie** y **nombre**, y el método **Mascota**. Sobrecargue el operador `<<` para poder imprimir instancias de la clase **Mascota** en la forma: `<especie:nombre>`.

NOTA: será necesario crear métodos getters para cada atributo.

- Enviar por correo a `ezequiel_arceo@my.uvm.edu.mx`
- El asunto del correo será `POO_TrabajoB3_NOMBRE_DEL_AUTOR`
- El nombre de cada archivo adjunto será `POO_TrabajoB3_NOMBRE_DEL_AUTOR.*`

Ejemplo:
ASUNTO: `POO_TrabajoB3_JUAN_PEREZ_FERNANDEZ`
ADJUNTO: `POO_TrabajoB3_JUAN_PEREZ_FERNANDEZ.cpp`

## 4.2. Sobrecarga de operadores aritméticos +,-,*,/

# 5. Proyecto Final

En la sección anterior aprendimos cómo sobrecargar los operadores `<<,+,-,*,/` con ejemplos de aritmética entera (manual y paso a paso) en nuestro sistema posicional.

En esta sección tomaremos los resultados de la sección anterior y representaremos los niveles enteros para realizar las 4 operaciones aritméticas en **notación maya de base 10**.

Las siguiente tabla muestra la asignación de alumnos y operaciones por equipo:

| EQUIPO | ALUMNO | ASIGNACIÓN | OP |
|---|---|---|---|
| 1 | ORTIZ AYALA CARLOS ROBERTO<br>ARRIAGA HERNANDEZ ESAU<br>TEJEDA VALENZUELA ROMULO MAGDIEL | suma maya | + |
| 2 | MARTINEZ VILLANUEVA JOSE ALFREDO<br>CASTAÑEDA MORENO JAVIER<br>RAKITA LUCAS | resta maya | + |
| 3 | MAY CUEVAS MARIA VALERIA<br>CARDENAS MAY DANIEL FERNANDO<br>CENICEROS CEBALLOS RAFAEL | multiplicación maya | * |
| 4 | MORALES MITRE RUBEN FRANCISCO<br>MARMOLEJO ZETINA ALLAN ABISAI<br>PEREZ CHAN BERENICE DEL CARMEN<br>SANCHEZ GONZALEZ ALEJANDRO TEODORO | división maya | / |

**Intrucciones:**

1. Modificará la clase **Nivel** para:
   a) contener dos atributos enteros: **barritas** y **piedritas**.
   b) cada barrita se vale 5 unidades, y cada piedrita vale 1 unidad.
2. Modificará la clase **Entero** para:
   a) tener un método que implemente la operación aritmética asignada a su equipo. Por ejemplo, a quien le tocó "suma maya" implementará el método **suma(Entero otro_entero)**.
   b) sobrecargará el operador asignado para llamar al método aritmético asignado. Por ejemplo, a quien le tocó "suma maya" sobrecargará el operador `+`.
3. Sobrecargará `<<` para que los objetos de la clase **Entero** se impriman con `cout` en niveles apilados de forma vertical, con las unidades en la parte inferior.

4. La operación se imprimirá paso a paso en pantalla.

**NOTA:El Proyecto Final se entrega el Lunes 20 de Mayo de 2019 en horario de clases.**

Use como referencia las siguientes lecturas:

Matemáticas Mayas en base 10
The lucid and powerful Mayan mathematics for teaching
L. F. Magaña

Matemáticas Mayas en base 10
TO LEARN MATHEMATICS: MAYAN MATHEMATICS IN BASE 10
L. F. Magaña

Matemáticas Mayas
MATEMÁTICA MAYA
Las fascinantes, rápidas y divertidas
matemáticas de los mayas.
L. F. Magaña. Marzo 2006.