

Programación Orientada a Objetos

Parcial 3

Notas de curso

Dr. Ezequiel Arceo May

12 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
4.2.1. Representación Interna	12
4.2.2. Suma	13
4.2.3. Resta	14
4.2.4. Multiplicación	15
4.2.5. División	16
5. Proyecto Final	17

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

1. Concepto básicos

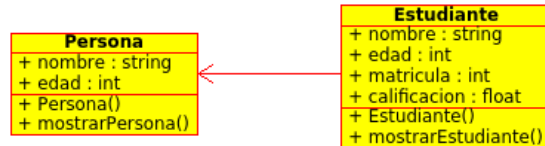
Los conceptos más básicos que debemos entender en la Programación Orientada a Objetos (POO) son:

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

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 este caso será la clase **Persona**, con los atributos `nombre` y `edad`, y los métodos `Persona` y `mostrarPersona`:

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class Persona{
6  private:
7      string nombre;
8      int edad;
9  public:
10     Persona(string, int);
11     void mostrarPersona();
12 };
13 Persona::Persona(string _n, int _e){
14     nombre = _n;
15     edad = _e;
16 }
17 void Persona::mostrarPersona(){
18     cout << "Nombre: " << nombre << endl;
19     cout << "Edad: " << edad << endl;
20 }
```

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::`:

```
1 class Estudiante: public Persona {
2     private:
3         int matricula;
4         float calificacion;
5     public:
6         Estudiante(string,int,int,float);
7         void mostrarEstudiante();
8 };
```

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**:

```
1 Estudiante::Estudiante(string _n, int _e, int _m, float _c): Persona(_n, _e){
2     matricula = _m;
3     calificacion = _c;
4 }
```

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

```
1 void Estudiante::mostrarEstudiante(){
2     mostrarPersona(); // uso método de la clase padre
3     cout << "Matricula: " << matricula << endl;
4     cout << "Calificacion: " << calificacion << endl;
5 }
```

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

```

1  int main(){
2      Estudiante e1("Ezequiel Arceo May",34,123456,8.5);
3      e1.mostrarEstudiante();
4      system("pause");
5      return 0;
6  }

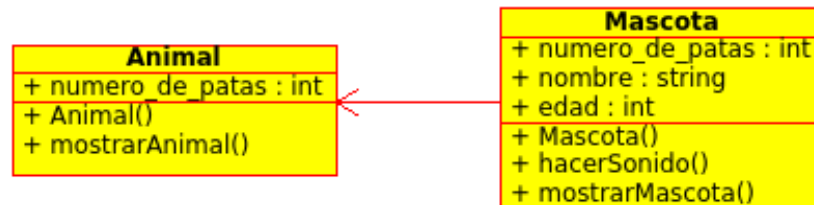
```

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 `P00_TrabajoB1_NOMBRE_DEL_AUTOR`
- El nombre de cada archivo adjunto será `P00_TrabajoB1_NOMBRE_DEL_AUTOR.*`

Ejemplo:

ASUNTO: `P00_TrabajoB1_JUAN_PEREZ_FERNANDEZ`

ADJUNTO: `P00_TrabajoB1_JUAN_PEREZ_FERNANDEZ.cpp`

3. Polimorfismo

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

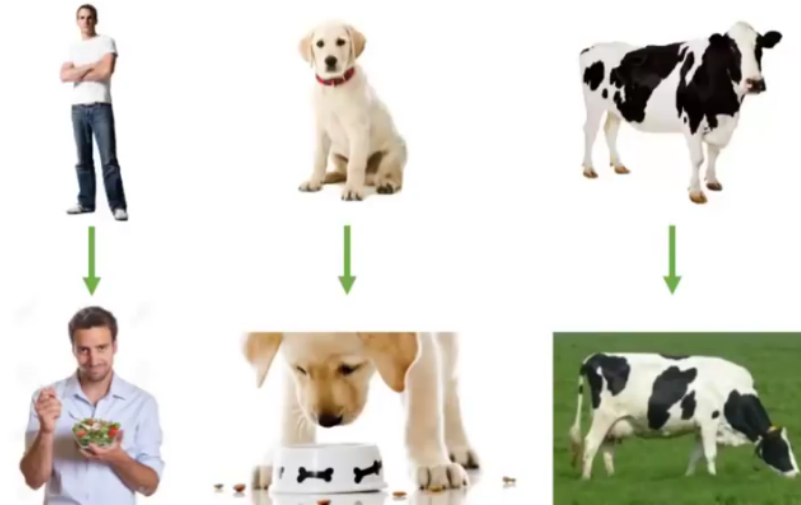


Figura 1: Considere objetos de las clases **Persona**, **Perro** y **Vaca**, cada uno de ellos puede **comer**, pero lo hacen de forma distinta.



3.1. Polimorfismo en C++ usando 'virtual'

En la 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.

```

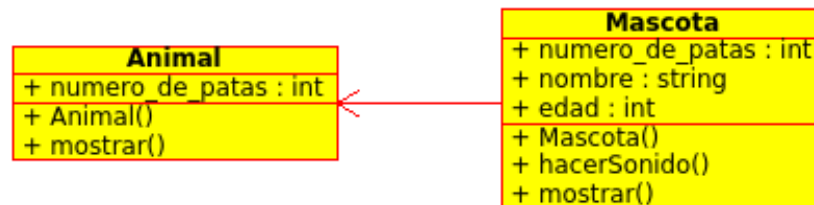
1  #include<iostream>
2  #include<string>
3  using namespace std;
4
5  class Persona{
6      private:
7          string nombre;
8          int edad;
9      public:
10         Persona(string, int);
11         virtual void mostrar(); // Polimorfismo
12 };
13 Persona::Persona(string _n, int _e){
14     nombre = _n;
15     edad = _e;
16 }
17 void Persona::mostrar(){
18     cout << "Nombre: " << nombre << endl;
19     cout << "Edad: " << edad << endl;
20 }
21 class Estudiante: public Persona {
22     private:
23         int matricula;
24         float calificacion;
25     public:
26         Estudiante(string,int,int,float);
27         void mostrar(); // Polimorfismo
28 };
29 Estudiante::Estudiante(string _n, int _e, int _m, float _c): Persona(_n, _e){
30     matricula = _m;
31     calificacion = _c;
32 }
33 void Estudiante::mostrar(){
34     Persona::mostrar(); // Polimorfismo
35     cout << "Matricula: " << matricula << endl;
36     cout << "Calificacion: " << calificacion << endl;
37 }
38 int main(){
39     Persona persona("Chabelo",100);
40     persona.mostrar();
41     Estudiante alumno("Ezequiel Arceo May",34,123456,8.5);
42     alumno.mostrar();
43     return 0;
44 }

```

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 `P00_TrabajoB2_NOMBRE_DEL_AUTOR`
- El nombre de cada archivo adjunto será `P00_TrabajoB2_NOMBRE_DEL_AUTOR.*`

Ejemplo:

ASUNTO: `P00_TrabajoB2_JUAN_PEREZ_FERNANDEZ`

ADJUNTO: `P00_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:

```
1 int mi_entero = 2;
2 cout << mi_entero;
3 float mi_decimal = 3.1416;
4 cout << mi_decimal;
5 string mi_cadena = "Hola";
6 cout << mi_cadena;
```

Sin embargo, los tipos de datos que definimos nosotros (nuestras clases) no pueden ser impresos 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**.

```
1 #include <iostream>
2
3 using namespace std;
4
5 class Punto
6 {
7     private:
8         int x, y;
9     public:
10        Punto(int _x, int _y); // Constructor
11        void mostrarPunto(); // QUEREMOS DEJAR DE HACER ESTO!
12        int get_x(); // getter de x
13        int get_y(); // getter de y
14 };
15
16 // Constructor por defecto
17 Punto::Punto(int _x, int _y) {
18     x = _x;
19     y = _y;
20 }
```

```

1 // Getters
2 int Punto::get_x(){
3     return x;
4 }
5 int Punto::get_y() {
6     return y;
7 }
8 void Punto::mostrarPunto(){// QUEREMOS DEJAR DE HACER ESTO!
9     cout << "(" << x << ", " << y << ")" << endl;
10 }
11
12 int main(){
13     Punto p1 = Punto(2,4);
14     p1.mostrarPunto(); // funciona
15     cout << p1 << endl; // NO FUNCIONA, PERO QUISIÉRAMOS QUE SÍ
16     return 0;
17 }

```

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:

```

1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 int main(){
6     string s1 = "Hola";
7     string s2 = " mundo";
8     cout << (s1 + s2) << endl; // + como operador
9     cout << operator+(s1,s2) << endl; // + como función
10    return 0;
11 }

```

4.1. Sobrecarga de <<

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

```
1 ostream& operator <<(ostream& output, Tipo mi_tipo){
2     output << ... ;
3     return output;
4 }
```

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

```
1 ostream& operator <<(ostream& output, Punto punto){
2     output << "(" << punto.get_x() << ", " << punto.get_y() << ")";
3     return output;
4 }
```

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

```
1 int main(){
2     Punto p1 = Punto(2,4);
3     p1.mostrarPunto(); // funciona
4     cout << p1 << endl; // FUNCIONA!
5     return 0;
6 }
```

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 **P00_TrabajoB3_NOMBRE_DEL_AUTOR**
- El nombre de cada archivo adjunto será **P00_TrabajoB3_NOMBRE_DEL_AUTOR.***

Ejemplo:

ASUNTO: **P00_TrabajoB3_JUAN_PEREZ_FERNANDEZ**

ADJUNTO: **P00_TrabajoB3_JUAN_PEREZ_FERNANDEZ.cpp**

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

Sobrecargaremos los operadores +, -, *, / mediante métodos para el manejo de números de la clase **Entero** que nosotros mismos escribiremos. Pero primero necesitaremos entender nuestro sistema numérico.

4.2.1. Representación Interna

Cuando escribimos un número entero, como 276, estamos escribiendo una versión abreviada

$$\begin{aligned} 276 &= 200 + 70 + 6 \\ &= 2 \cdot 100 + 7 \cdot 10 + 6 \cdot 1 \\ &= 2 \cdot 10^2 + 7 \cdot 10^1 + 6 \cdot 10^0 \end{aligned}$$

Entonces podemos codificar el 276 como un vector de la forma

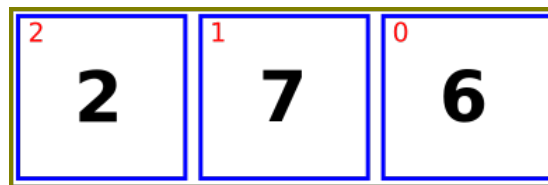


De esta forma la representación interna de *un entero es un vector de niveles*,

- el valor del nivel con índice 0 corresponde a las unidades,
- el valor del nivel con índice 1 corresponde a las decenas,
- el valor del nivel con índice 2 corresponde a las centenas,
- ...

Nosotros modelaremos a los enteros con la clase **Entero**, cuyo único atributo será un vector de la clase **Nivel**.

Por comodidad representaremos en papel a los enteros con los niveles ordenados de izquierda a derecha.



4.2.2. Suma

La suma de dos enteros **e1** y **e2** se lleva a cabo sumando sus valores nivel a nivel

² 2	¹ 7	⁰ 6
+		
² 0	¹ 1	⁰ 2
=		
² 2	¹ 8	⁰ 8

Note que, antes de sumar nivel a nivel, hemos añadido niveles con valor cero al número más *corto* para que alcance la longitud (en niveles) del número más *largo*.

4.2.3. Resta

La resta de dos enteros $e1$ y $e2$ es en realidad la suma de $e1$ y $-e2$

² 2	¹ 7	⁰ 6
--------------------------	--------------------------	--------------------------

-

² 0	¹ 1	⁰ 2
--------------------------	--------------------------	--------------------------

=

² 2	¹ 7	⁰ 6
--------------------------	--------------------------	--------------------------

+

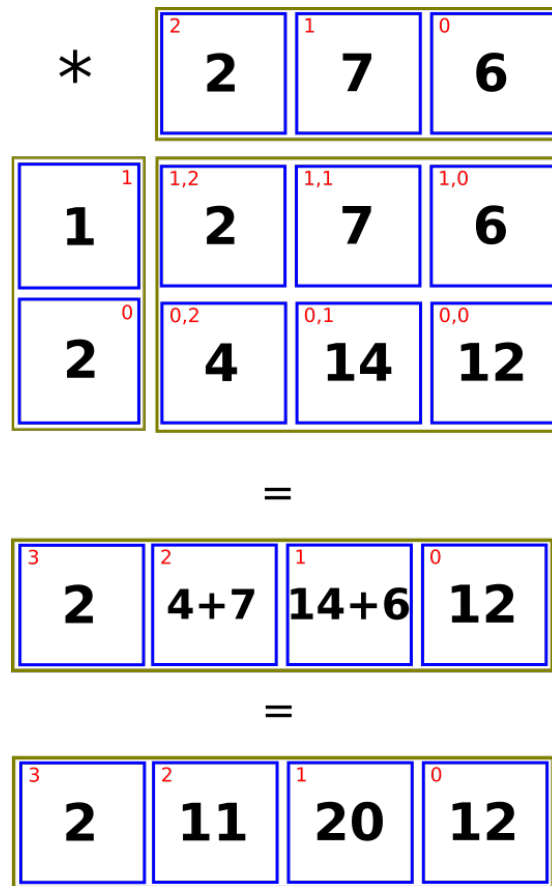
² -0	¹ -1	⁰ -2
---------------------------	---------------------------	---------------------------

=

² 2	¹ 6	⁰ 4
--------------------------	--------------------------	--------------------------

4.2.4. Multiplicación

La multiplicación de dos enteros **e1** y **e2** es en realidad la suma del producto de todas las parejas de niveles

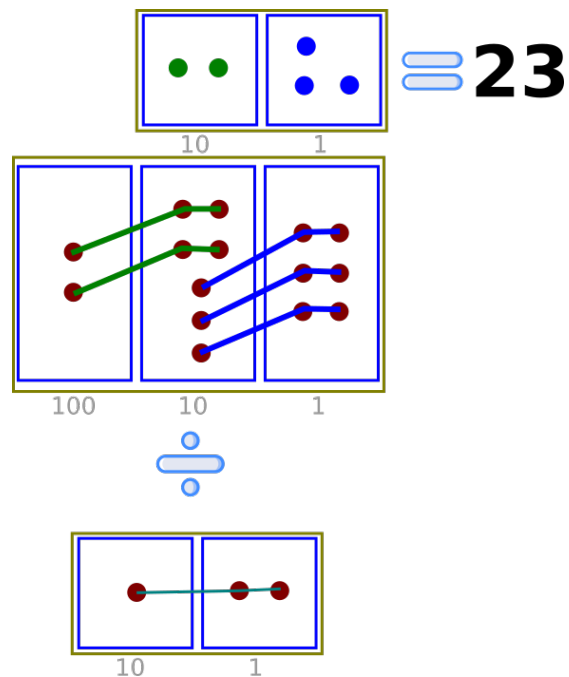


$$\begin{aligned}
 267 \cdot 12 &= (2 \cdot 10^2 + 7 \cdot 10^1 + 6 \cdot 10^0) \cdot (1 \cdot 10^1 + 2 \cdot 10^0) \\
 &= 2 \cdot 10^3 + 7 \cdot 10^2 + 6 \cdot 10^1 + 4 \cdot 10^2 + 14 \cdot 10^1 + 12 \cdot 10^0 \\
 &= 2 \cdot 10^3 + (7 + 4) \cdot 10^2 + (14 + 6) \cdot 10^1 + 12 \cdot 10^0 \\
 &= 2 \cdot 10^3 + 11 \cdot 10^2 + 20 \cdot 10^1 + 12 \cdot 10^0
 \end{aligned}$$

4.2.5. División

La división de dos enteros **e1** y **e2** es en realidad la una resta de sucesiva de **e2** a **e1**

$$\begin{aligned} 276 &= 120 + 120 + 12 + 12 + 12 \\ &= 2 \cdot 12 \cdot 10^1 + 3 \cdot 12 \cdot 10^0 \\ &= (2 \cdot 10^1 + 3 \cdot 10^0) 12 \end{aligned}$$



El método **divide** se rompe si el valor en el último nivel del numerador es menor que el valor en el último nivel de denominador ¿Cómo podemos corregir esto?

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

Instrucciones:

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.