Programación Orientada a Objetos en C++

Este documento que tienes entre las manos o en el ordenador tiene como finalidad mostrar las características que ofrece C++ en cuanto a la Programación Orientada a Objetos sin entrar en excesivos detalles.

No se explica la sintaxis básica de C/C++, con lo que es requisito que el lector conozca dicha sintaxis. Tampoco se tratan otros aspectos como la Standard Template Library (STL) o el manejo de excepciones.

He intentado que el tutorial fuera todo lo inteligible y claro posible, pero por motivos de tiempo no está todo lo revisado que hubiese deseado. Así que esta es la Versión 1.0 que intentaré ir afinando con el tiempo.

El tutorial posee un ejemplo final donde se aplican todas las características explicadas. Se trata de un programa sencillo para gestión de animales de un Zoológico. Para ello se han definido las clases que nos han sido necesarias.

José Antonio Muñoz Jiménez
jamj2000 at gmail dot com
Almería – IES Al-Ándalus – Mayo 2009 (V. 1.0)

BETA

Índice de contenido

Introducción

A grandes rasgos la historia de la programación de computadoras ha seguido un curso en el cual se pretende crear programas cada vez más grandes y complejos y por otro lado se busca reducir dicha complejidad durante su diseño y codificación.

Básicamente existen tres etapas en toda esta historia:

Programación "espagueti"

En los primeros lenguajes de programación (Ensamblador y las primeras versiones de los lenguajes de alto nivel) los programas se ejecutaban instrucción a instrucción (de forma ***secuencial****) y en ciertas circunstancias, dependiendo de las condiciones, se "saltaba" a otra posición del programa (instrucciones JMP, GOTO, ...) y se continuaba allí la ejecución. Con lo cual seguir el curso del programa era bastante engorroso y hacía que la tarea de depuración y mantenimiento fuera complicada. Esto provocaba que la creación y mantenimiento de programas grandes fuera difícil cuando no imposible.*

Ejemplo de código ensamblador:

```
Comprueba_si_activa:
        CMP     BYTE PTR[DI], 80H           ; 80H : Partición activa
        JZ      Cargar_sector_de_arranque

Comprueba_si_inactiva:
        CMP     BYTE PTR[DI], 00H           ; 00H: Partición inactiva
        JNZ     Error_1
        CMP     DI, 07EEH
        JZ      Error_4
        ADD     DI, +10H                    ; siguiente entrada
        JMP     Comprueba_si_activa


; Tabla de Partición Inválida
Error_1:
        MOV     BP, OFFSET Mensaje1+600H
Llamada_a_rutina:
        CALL    Rutina_Tratamiento_de_Errores
Bucle_infinito:
        JMP     Bucle_infinito
```

Programación estructurada (PE)

Durante la década de los 70, gracias a trabajos anteriores de algunos científicos de la computación (Böhm-Jacopin, Dijkstra, ...), se consideró necesario llevar un mejor seguimiento del curso de ejecución de un programa. Para ello se instó a evitar el uso de las instrucciones de salto anteriores y favorecer el uso de estructuras ***condicionales*** *y* ***repetitivas*** *que ofrecían la misma funcionalidad pero permitían una mayor claridad en el código. Así aparecieron instrucciones como IF y WHILE.*
Además se promocionó el uso de ***procedimientos,*** *en los cuales se guardaban un conjunto de instrucciones y los cuales eran invocados y ejecutados como si de una única instrucción se tratase.*
Por último con el fin de hacer más modulares los programas y de permitir la reutilización de código se crearon ***bibliotecas*** *(o librerías) de procedimientos que podían ser utilizadas por otros programas.*

Ejemplo de código C:

```c
#include <stdio.h>

int contador;

int main () {
      contador = 0;
      while (contador<10)     { /* 10 Repeticiones : 0 .. 9 */
            printf ("\nLínea %d", contador);
            contador = contador + 1;
      }
      return 0;
}
```

Programación orientada a objetos (POO)

A pesar de las grandes ventajas que aportaba la programación estructurada, todavía existía cierta dificultad en la creación de programas muy grandes.
La dificultad provenía principalmente del hecho de trabajar con procedimientos que utilizaban datos a menudo dispersos por toda la biblioteca e incluso otras bibliotecas.
Otra dificultad provenía del hecho de tener nombres de procedimientos distintos para hacer la misma operación, pero con tipos de datos distintos.
Por ejemplo en C para obtener el valor absoluto de un número se utilizan dos funciones, dependiendo de si el número es entero o de punto flotante (contiene decimales):

```
abs (-30);
fabs (-30.2)
```

Ejemplo de código Java:

```
package zoo;


public class Main {
    public static void main(String[] args) {
      Gallina ruperta = new Gallina (Animal.Habitat.BOSQUE, "trigo", 10, 60);

      Gallina golfreda = new Gallina (Animal.Habitat.BOSQUE, "maíz", 10, 60);


      Leon leopoldo = new Leon ();
      leopoldo.setHabitat (Animal.Habitat.ESTEPA);

      leopoldo.setComida ("gacelas");
      leopoldo.setLongevidad (20);
      leopoldo.setPeriodoLactancia (30);

      ruperta.come ();
      golfreda.come ();
      leopoldo.desplaza ();
      leopoldo.come ();
      leopoldo.ruge ();
      leopoldo.setRugido ("GGGGRRR GRRR GRR");
      leopoldo.ruge ();

      System.out.println ("Existen " + Animal.getNumAnimales()
                                    + " animales");
      System.out.println ("Existen " + Ave.getNumAves() + " aves");
    }
}
```

La programación orientada al objeto (POO) pretende un cambio de paradigma a la hora de diseñar y codificar los programas. Frente a la programación orientada a procedimientos, empleada en la PE, se propone la programación orientada a objetos que ofrece una visión más cercana a la forma de pensar del ser humano.

Se trata de programar en un "mundo" de objetos.
Un objeto puede ser cualquier elemento físico (mi coche, la bicicleta del vecino, Juan Ramirez, ...) o abstracto (factura, transacción bancaria, ...).

*Cada **objeto** suele poseer un identificador que lo distingue de forma unívoca.*
*Además, cada objeto poseerá un **estado** y un **comportamiento.***
*El estado de un objeto se representa mediante variables llamadas **propiedades.***
*El comportamiento de un objeto se representa mediante funciones llamadas **métodos.***

estado	*propiedades o atributos*	*(Variables de datos)*
comportamiento	*métodos*	*(Funciones)*

*Los objetos se agrupan en **clases.***
Por ejemplo los objetos mi coche, el coche del vecino y el coche del jefe son todos objetos de la clase Coche. Juan Ramirez, Ana Silva y José Redondo son todos objetos de la clase Persona.

*Las clases a su vez se guardan en **bibliotecas.***
Cada biblioteca define su conjunto propio de clases. Así, por ejemplo, existen bibliotecas para programar interfaces gráficas de usuario que contienen clases como Ventana, Botón, Barra de menú, ...

En POO existen 3 características fundamentales:

- *Encapsulación*
- *Herencia*
- *Polimorfismo*

Entre distintos lenguajes de POO existen pequeñas diferencias a la hora de implementar estas características.

Encapsulación

Clases

La encapsulación consiste en combinar los datos y las funciones que operan sobre dichos datos en un único sitio. Se evita de esta forma tener datos y funciones relacionas repartidos por múltiples archivos de código fuente dentro de la librería o biblioteca.

*Para este fin se ha creado un tipo de dato que se conoce en POO como clase (**class**). Una clase se parece en muchos aspectos a una estructura (**struct**) de C++ **.***

Ej:

```
class Animal {
   private:
      int edad;
      Sexo sexo;
   public:
      void setEdad (int  edad) { this->edad = edad; }
      void setSexo (Sexo sexo) { this->sexo = sexo; }
      int  getEdad ()          { return edad;       }
      Sexo getSexo ()          { return sexo;       }
};
```

```
struct Animal {
   private:
      int edad;
      Sexo sexo;
   public:
      void setEdad (int  edad) { this->edad = edad; }
      void setSexo (Sexo sexo) { this->sexo = sexo; }
      int  getEdad ()          { return edad;       }
      Sexo getSexo ()          { return sexo;       }
};
```

Semejanzas con struct

- *class y struct admiten dos tipos de miembros: variables y funciones.*
- *Para acceder a sus miembros directamente se utiliza el operador punto (.)*
- *Para acceder a sus miembros a través de un puntero se utiliza el operador flecha (→)*

Diferencias con struct

- *En struct, si no se indica nada, sus miembros son públicos.*
- *En class, si no se indica nada, sus miembros son privados.*

Las estructuras en C están mucho más limitadas con respecto a las de C++. En C no es posible declarar funciones miembro de una estructura, entre otras cosas.

Programa de ejemplo:

```
#include <iostream>

using namespace std;


enum Sexo {SIN_DATOS, MACHO, HEMBRA};

// Cambiar la palabra class por struct y el programa funcionará igual
class Animal {
   private:
      int edad;
      Sexo sexo;
   public:
      void setEdad (int  edad) { this->edad = edad; }
      void setSexo (Sexo sexo) { this->sexo = sexo; }
      int  getEdad ()          { return edad;       }
      Sexo getSexo ()          { return sexo;       }
};

int main () {
    Animal mi_perro;
    Animal *Tobi;

    Tobi = &mi_perro;

    mi_perro.setEdad (2);
    Tobi->setSexo (MACHO);

    cout << "Mi perro tiene " << Tobi->getEdad() << " años" << endl;
    cout << "Mi perro es " << mi_perro.getSexo() << endl;

    return 0;
}
```

El proceso de crear objetos de una clase se denomina instanciar. Por tanto se dice que cada objeto es una instancia de la clase a la que pertenece.
En el ejemplo anterior, el objeto mi_perro es una instancia de la clase Animal.

En POO, instanciar se asemeja mucho a declarar una variable en PE.
Ej:

```
Animal  mi_perro;               // Instanciación de un objeto en POO
int     numero;                 // Declaración de una variable en PE

// Con punteros
Animal  *mi_gato = new Animal;  // POO
int     *num     = new int;     // PE
```

Especificadores de acceso

Los miembros de una clase pueden ser

- *private: puede ser accedido sólo por los miembros y amigos (friend) de la clase donde está declarado.*
- *protected: puede ser accedido sólo por los miembros y amigos (friend) de la clase donde está declarado, y por los miembros y amigos (friend) de las clases derivadas de esta.*
- *public: puede ser accedido sin restricciones.*

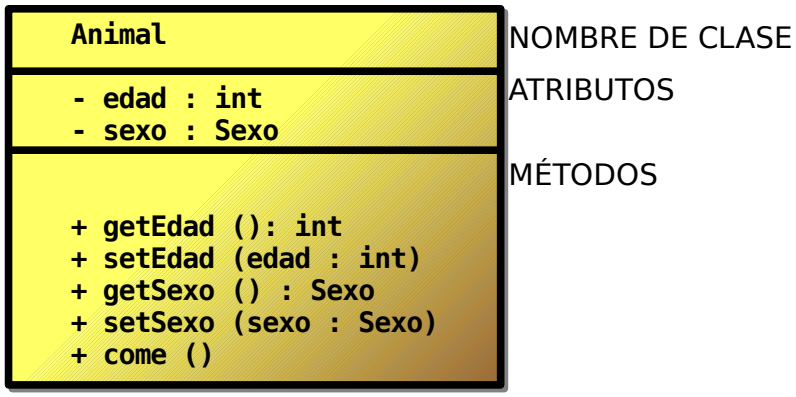
Ocultación

*La encapsulación permite la **ocultación** de información y su acceso controlado mediante una **interfaz** pública. Así, normalmente, el estado de un objeto representado mediante sus propiedades suele permanecer oculto, y sólo es posible modificarlo haciendo uso de los métodos públicos que definen el comportamiento del objeto (lo que podemos hacer con él).*

La interfaz de una clase (y por tanto cualquiera de sus objetos) la forman todos sus métodos públicos (y de forma poco frecuente alguna propiedad pública). Abajo se indica de forma sombreada la interfaz de la clase Animal.

```
class Animal {
   private:
      int edad;
      Sexo sexo;
   public:
      void setEdad (int  edad) { this->edad = edad; }
      void setSexo (Sexo sexo) { this->sexo = sexo; }
      int  getEdad ()          { return edad;       }
      Sexo getSexo ()          { return sexo;       }
};
```

Representación en UML (Universal Modeling Language) de la clase Animal.



Constructores

A menudo cuando se crea un objeto es necesario realizar ciertas operaciones de inicialización. Esto puede hacerse automáticamente mediante un método constructor. El constructor no tiene tipo de retorno y su nombre es idéntico a la clase. El constructor puede tener parámetros o no tenerlos.
Pueden existir varios constructores mediante el uso de sobrecarga de funciones.

El constructor permite que cada vez que se declare un objeto se ejecuten las operaciones de inicialización indicadas dentro de él.

```
Animal ()  { cout  << "Nuevo animal "; }
```

Destructor

Cuando un objeto se destruye a menudo es necesario hacer alguna operación de "limpieza": liberar memoria reservada dinámicamente o liberar otro tipo de recursos. Esto puede hacerse automáticamente mediante un método destructor.
El destructor no tiene tipo de retorno y su nombre es idéntico a la clase anteponiendole una tilde. El destructor nunca tiene parámetros.
Sólo existe un destructor.

El destructor permite que cada vez que se destruya un objeto se ejecuten las operaciones de "limpieza" indicadas dentro de él.

```
~Animal () { cout << "Desaparece animal ";}
```

Copia de objetos

Un objeto puede ser copiado de dos formas:

- *Inicialización: (constructor de copia)*
  - *al pasarlo como argumento de una función*
  - *al ser devuelto por una función*
- *Asignación: (operador de asignación de copia)*

```
Animal (const Animal&);             // constructor de copia
Animal& operator= (const Animal&);  // operador de asignación de copia
```

Si el programador no crea las funciones miembro anteriores entonces el compilador las crea implícitamente. En total son 4:

- Constructor
- Destructor
- Constructor de copia
- Operador de asignación de copia

Amigos

*Un amigo (**friend**) de una clase es una función, operador o clase que no es miembro de la clase pero que se le permite hacer uso de los miembros privados y protegidos de la clase.*

```
class Animal {
   private:
      int edad;
      Sexo sexo;
   public:
      void setEdad (int  edad) { this->edad = edad; }
      void setSexo (Sexo sexo) { this->sexo = sexo; }
      int  getEdad ()          { return edad;       }
      Sexo getSexo ()          { return sexo;       }

      friend void cambiarEdad (Animal *a, int annos);
};

void cambiarEdad (Animal *a, int annos) { a->edad = annos; }
```

Puntero this

*El puntero **this** es una palabra reservada que representa un puntero al objeto mismo. Al tratarse de un puntero es necesario utilizar el operador flecha (→) para acceder a cada miembro del objeto. Se utiliza a menudo para distinguir entre variables miembro del objeto y parámetros que tienen el mismo nombre.*
Ej.:

```
void setEdad (int edad)   { this->edad = edad; }
```

this->edad (variable miembro del objeto)
edad (parámetro pasado a la función)

Miembros estáticos

Una clase puede contener miembros estáticos, tanto datos como funciones.

Los miembros estáticos de datos también se conocen como variables de clase, debido a que sólo existe una copia compartida por todos los objetos de la misma clase. Dichas variables existen aunque no exista ningún objeto declarado. Se comportan como si de variables globales se tratase (salvo restricciones de visibilidad).

```
static int numAnimales;
```

Por ello para evitar que sean inicializadas varias veces por cada objeto creado, no se permite su definición dentro de la clase, debiendo realizarse esta fuera de ella. Dentro de la clase sólo se declara la variable.

```
int Animal::numAnimales = 0;   // !!! aquí no se indica el especificador static
```

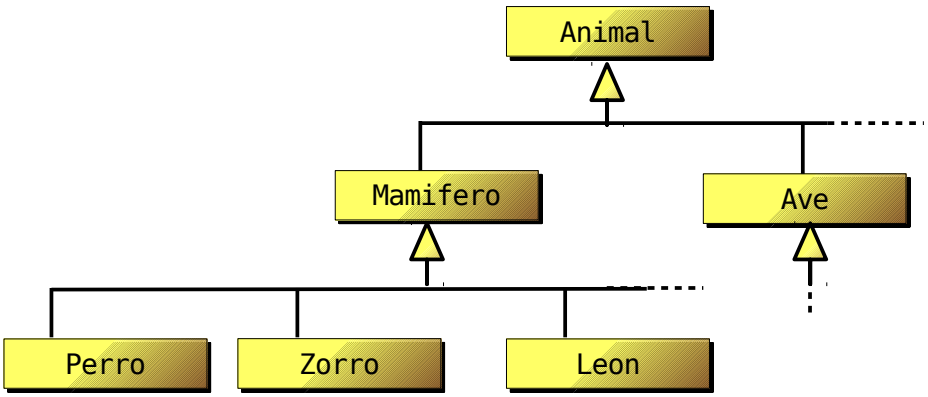
También es posible hacer uso de funciones estáticas. Al igual que las variables se comportan como si se tratase de funciones globales (salvo restricciones de visibilidad) y existen aunque no se declare ningún objeto de dicha clase. Por este motivo tienen dos limitaciones:
- *no pueden hacer uso de las variables no-estáticas de la clase (sólo var. estáticas).*
- *no se permite el uso del puntero this dentro de una función estática.*

Herencia

Jerarquía de clases

Si observamos el programa del apartado anterior, pronto nos daremos cuenta de que para representar a mi_perro existen pocas propiedades para indicar su estado y también existen pocos métodos para indicar su comportamiento.

*Hubiera sido más aconsejable utilizar una clase más especializada como Mamífero o Perro. A menudo las bibliotecas poseen toda una **jerarquía de clases**, en la cual existe una (o varias) **clase base** y distintas **clases derivadas**. Se dice que una clase es la clase base de otra cuando se halla en un nivel superior de la jerarquía. Se dice que una clase es una clase derivada de otra cuando se halla en un nivel inferior de la jerarquía.*
Mamifero es la clase base de Perro, Zorro y Leon. Mamifero, al igual que Ave, es a su vez es una clase derivada de Animal.



La clase base representa la descripción más general de un conjunto de rasgos. La clase derivada hereda esos rasgos y añade propiedades y métodos que le son propios. Por ejemplo la clase Mamifero podría ser:

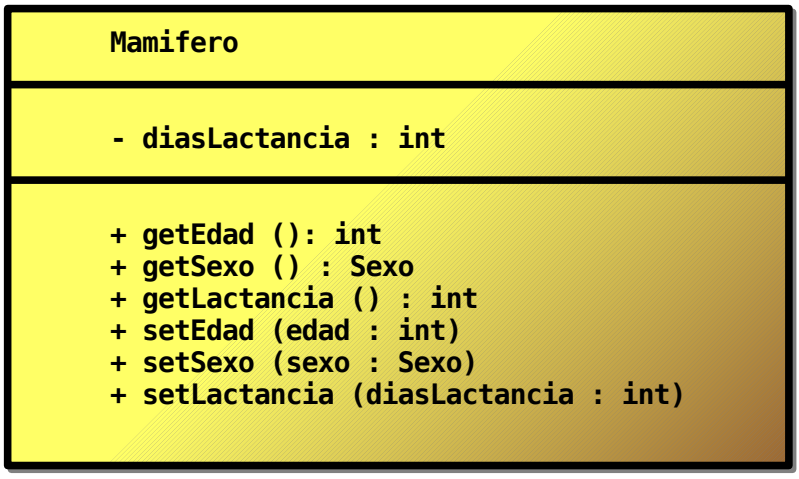
```
class Mamifero : public Animal {
  private:
    int diasLactancia;
  public:
    void setLactancia (int diasLactancia)
                              {this-> diasLactancia = diasLactancia; }
    int getLactancia ()       { return periodoLactancia; }
};
```

*El organizar una biblioteca mediante una jerarquía de clases tiene su utilidad debido a que nos permite trabajar en cada momento con la clase más adecuada, además de permitir la **reusabilidad** del código, permitiendo al programador si así lo desea ampliar dicha biblioteca creando clases que hereden de las ya existentes. Por ejemplo podría crear dos clases llamadas Perro_domestico y Perro_salvaje que heredaran de Perro.*

Se dice que una clase es una **clase abstracta** si no permite crear objetos de dicha clase.
Se dice que una clase es una **clase final** si no se permiten crear clases derivadas de ella.

Normalmente las clases de lo más alto de la jerarquía son clases abstractas mientras que las clases de lo más bajo de la jerarquía suelen ser clases finales. Las clases finales suelen darse en Java.
En C++ no existen clases finales propiamente dichas.

La clase Mamifero en UML:



Observa como ha heredado de Animal los métodos
–getEdad
–setEdad
–setSexo
–getSexo
Aunque también hereda de Animal las propiedades
–edad
–sexo
éstas siguen siendo privadas para Animal, con lo cual la única forma de que la clase Mamífero pueda acceder a ellas es mediante los métodos anteriores.

Una clase derivada hereda todos los miembros de la clase base excepto:

- *su constructores y destructor*
- *su operador de asignación de copia*
- *sus amigos (friend)*

Aunque los constructores y destructor de la clase base no se heredan, su constructor por defecto (constructor sin parámetros) y su destructor es siempre llamado cuando se crea o destruye un objeto de la clase derivada.

La clase Mamifero hereda todos los miembros de la clase Animal. Sin embargo las propiedades edad y sexo de la clase Animal permanecen ocultas, incluso para las clases derivadas.
*Los miembros públicos siguen siendo públicos en la clase derivada pues así lo hemos decidido con el especificador de acceso public (**class** Mamifero : **public** Animal).*
*En lugar de public puede escribirse **private** o **protected.***

Especificadores de acceso

Cuando se declara una clase derivada, el especificador de acceso a la clase base puede ser:

- *public: los miembros públicos de la clase base son accesibles como miembros públicos de la clase derivada y los miembros protegidos de la clase base son accesibles como miembros protegidos de la clase derivada*
  public → public
  protected → protected

- *protected: los miembros públicos y protegidos de la clase base son accesibles como miembros protegidos de la clase derivada*
  public → protected
  protected → protected

- *private: los miembros públicos y protegidos de la clase base son accesibles como miembros privados de la clase derivada.*
  public → private
  protected → private

Si no existe especificador de acceso para la clase base, se supone public cuando la clase derivada es una estructura (struct) y private cuando la clase derivada es una clase (class).

```
class Mamifero : Animal { /* ... */ };  // privado por defecto

struct Mamifero : Animal { /* ... */ }; // público por defecto
```

Redefinición de métodos

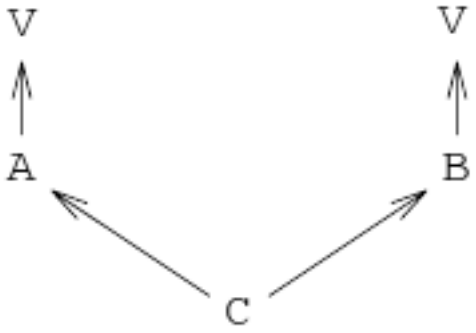
*Es posible **sobreescribir** o **redefinir** métodos de la clase base dentro de la clase derivada.*

Por ejemplo si tenemos una clase Leon que hereda de Mamífero que a su vez hereda de Animal, entonces disponemos de un método come(). Podemos sobreescribir o redefinir dicho método para que se adapte mejor a nuestra clase.

```
void Animal::come () {}

void Leon::come () { cout << “El león come gacelas”; }
```

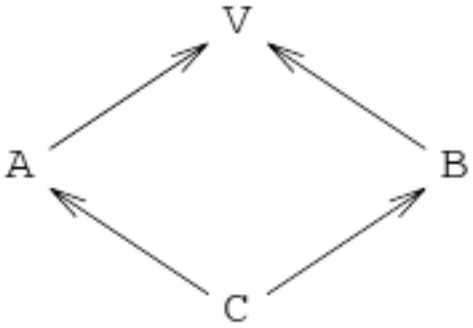
Herencia múltiple y clases virtuales

Una clase puede heredar de más de una clase. La clase C deriva de A y B. A su vez A y B derivan de V. Esto puede llevar a que C posea dos copias de la clase V.




```
class V { /* ... */ };
class A : public V { /* ... */ };
class B : public V { /* ... */ };
class C : public A, public B { /* ... */ };
```

Una clase puede ser heredada como virtual. Esto es muy útil, ya que permite que las clases derivadas sólo posean una copia de la clase base.




```
class V { /* ... */ };
class A : virtual public V { /* ... */ };
class B : virtual public V { /* ... */ };
class C : public A, public B { /* ... */ };
```

Se utiliza está técnica cuando realizamos herencia múltiple y no deseamos que haya copias duplicadas de las clases base lo cual produciría ambigüedad al tener métodos duplicados con el mismo nombre.

Polimorfismo

Permite realizar acciones distintas (aunque parecidas) sobre distintos tipos de datos a través de un nombre común. Se lleva a la práctica la técnica de "Una interfaz, múltiples métodos".

Ello facilita enormemente la programación, cuando se utiliza correctamente, al verse reducida de manera considerable la cantidad de nombres que el programador debe recordar.

La resolución de dichos nombres, es decir la asociación de dicho nombre con el método adecuado, puede hacerse en tiempo de compilación o durante la ejecución del programa.

Polimorfismo en tiempo de compilación

Es posible utilizar el mismo nombre de método o el mismo operador para realizar la "misma" operación sobre objetos diferentes. Esto se hace mediante:

- *Sobrecarga de funciones*
- *Sobrecarga de operadores*

Sobrecarga de funciones

C++ permite la sobrecarga de funciones, es decir varias funciones pueden tener el mismo nombre pero recibir distinto tipo y/o número de parámetros. Sin embargo no su pueden declarar dos funciones que sólo se diferencien en el tipo devuelto, pues el compilador no sabrá decidir cual de las dos utilizar.

```
// Sin parámetros
Animal:: Animal () { cout << "\n-> Animal "; numAnimales++; }

// Tres parámetros
Animal:: Animal (string id, Sexo sexo = SIN_DATOS, int edad = 0) {
  this->id   = id;
  this->sexo = sexo;
  this->edad = edad;
}
```

A menudo se sobrecargan los constructores para facilitar la inicialización de un objeto de diversas formas. El destructor no puede sobrecargarse.
También es posible otros método de una clase.

Sobrecarga de operadores

En C/C++ existen numerosos operadores de diversos tipos. Los operadores operan sobre operandos que les son "conocidos". Por ejemplo el operador binario << opera en C sobre 2 operandos enteros o compatibles (uno a cada lado del operador), desplazado el primer operando el número de bits indicados por el segundo operando.

```
int  num = 2;    // ...000010

num = num << 3;  // num = 16 →  ...010000
```

En C++ a este operador se le ha añadido nueva funcionalidad sobrecargándolo para que pueda trabajar con flujos de datos convirtiéndolo en un operador de extracción para salida de datos. El primer operando es un flujo de salida y el segundo puede ser cualquier tipo básico, string y ciertos objetos.

```
cout << "Hola";
```

En C++ los operandos tienen su equivalente en forma de función. La línea anterior también puede escribirse:

```
cout.operator<< ("Hola");
```

A pesar de la potencia del operador << y de la cantidad de tipos que reconoce como segundo operador, no es capaz de reconocer los objetos de nueva creación. Si declaramos la clase Animal y creamos objetos de dicha clase, el operador << no sabrá como mostrarlos por pantalla. Por dicho motivo deberemos sobrecargar dicho operador añadiéndole así mayor funcionalidad.

```
ostream& operator<< (ostream& , const Animal&);

ostream& operator<< (ostream &out, const Animal &a) {
  string s;
  if (a.sexo)
    if (a.sexo==MACHO)  s="Macho";  else  s="Hembra";
  else
    s = "Sin datos";

  out << endl << "DATOS DEL ANIMAL" << endl;
  out << "Nombre : " << a.id << endl;
  out << "Sexo   : " << s << endl;
  out << "Edad   : " << a.edad << endl;
  return out;
}
```

El operador << es uno de los operadores más frecuentemente sobrecargados en C++

Polimorfismo en tiempo de ejecución

Básicamente se consigue con punteros o referencias a clase base. Para ello vamos a ver primero lo que son miembros virtuales pues ellos nos permiten crear clases polimórficas y abstractas, condición necesaria para poder utilizar un puntero a clase base apropiadamente.

Miembros virtuales

*Una clase que hereda o declara una función virtual se denomina **clase polimórfica**.*
*Una clase que hereda o declara una función virtual pura se denomina **clase abstracta**.*

Una clase polimórfica es instanciable, es decir pueden crearse objetos de dicha clase.
Una clase abstracta no es instanciable, no pueden crearse objetos de dicha clase.

*La técnica de **clase abstracta** proporciona la noción de un concepto general, como una figura, de la cual solo variantes más concretas como un círculo o un cuadrado, pueden ser usadas realmente. Una clase abstracta puede ser usada también para definir una **interface** de la cual cada clase derivada proporcionará una implementación.*

No se pueden crear objetos de una clase abstracta. Dicho tipo de clase tiene dos utilidades:
- *sirve de "plantilla" para sus clases derivadas*
- *un puntero a dicha clase puede apuntar a cualquier clase derivada.*

En C++ se emplean las clases abstractas para implementar los interfaces.
En Java y en C# existe la palabra clave "interface".
Una clase abstracta donde todas sus funciones fuesen virtuales puras sería equivalente a una interface de Java.

Punteros a clase base abstracta o polimórfica

Un puntero a la clase base abstracta o polimórfica puede apuntar a cualquiera de sus clases derivadas.

Es posible utilizar un puntero o una referencia a una clase base abstracta o polimórfica para acceder a los miembros comunes de cualquier clase derivada. Sólo es posible acceder a los miembros de las clases derivadas que se hallen declarados en la clase base.
Si queremos acceder a los miembros que cada clase derivada ha ido añadiendo debemos de hacer un casting.

```
Animal *a = new Leon;
a->setEdad (7);          // Bien. Miembro de Animal y Leon
a->setRugido ("GRRR");   // Mal. No es miembro de Animal.
a->ruge();               // Mal. No es miembro de Animal.

Leon *l = dynamic_cast <Leon *> (a);
l->setRugido ("GRRRR"); // Bien
l->ruge();               // Bien
```

El moldeo (casting) desde una clase base a una clase derivada a menudo se llama molde hacia abajo (downcast) debido a la convención de dibujar los árboles de herencia creciendo desde la raíz hacia abajo
Del mismo modo, un molde (cast) desde una clase derivada a una clase base se denomina molde hacia arriba (upcast)

*Un **dynamic_cast** requiere un puntero o una referencia a un tipo polimórfico para hacer un molde hacia abajo (downcast)*

Su mayor utilidad reside en el uso de funciones que tienen como parámetro un puntero o una referencia a la clase base. A dichas funciones se le puede pasar un puntero o referencia de cualquiera de sus clases derivadas, realizándose un conversión implícita de clase derivada a clase base (upcast) para adaptarse al parámetro de la función.
Dentro de la función, el acceso a los métodos comunes de clase base y derivada se hace directamente. Para acceder a los métodos específicos de cada clase derivada es necesario convertir el puntero o referencia a clase base a un puntero o referencia a clase derivada (downcast)

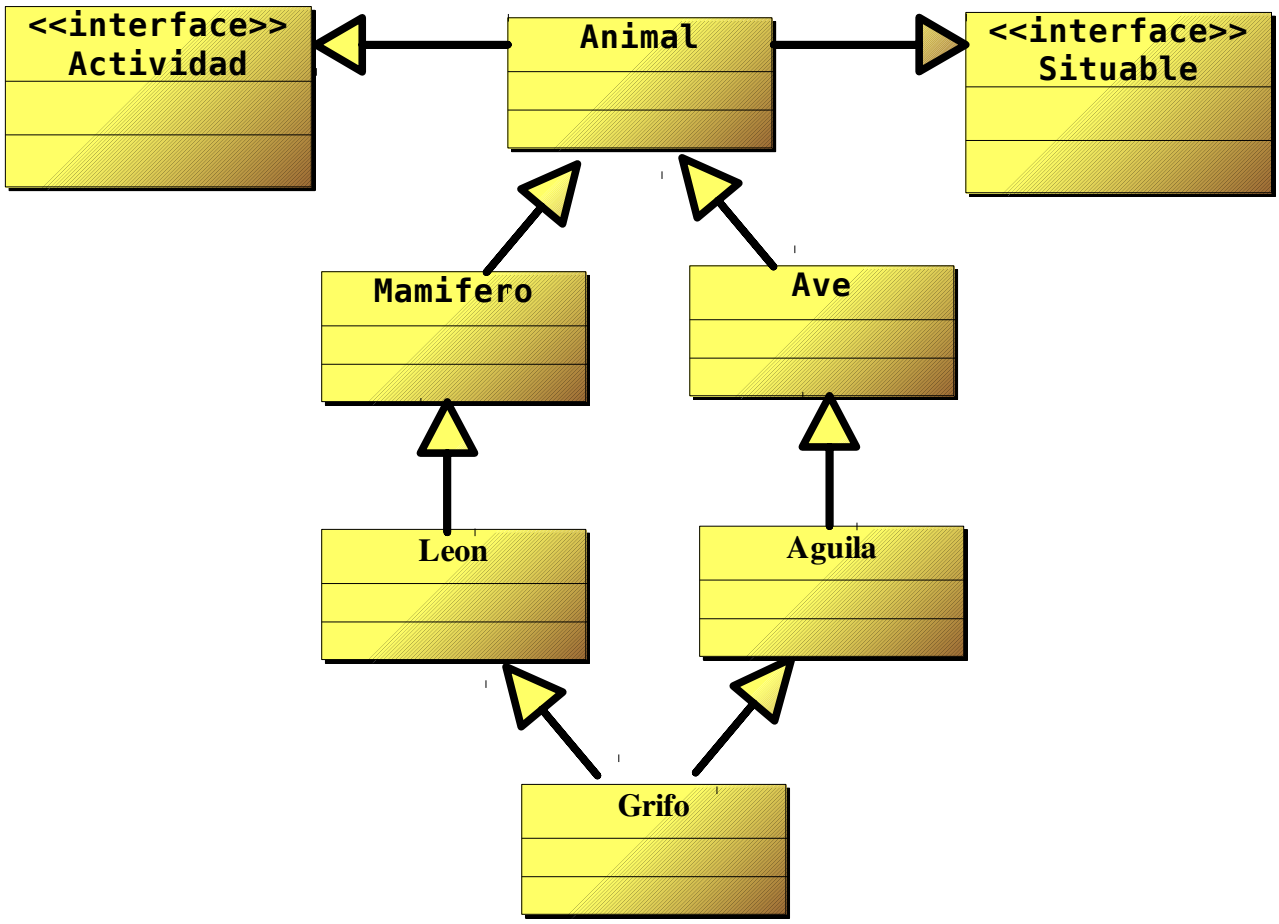
```
Leon   leopoldo;
Aguila agueda;

alojar (leopoldo);
alojar (agueda);


void alojar (Animal &a){
    if (typeid(a)==typeid(Leon)){
      Leon &leon = dynamic_cast<Leon &> (a);
      leon.situar ("Sector 1");
      leon.ruge ();              // Miembro de Leon
      return;
    }
    if (typeid(a)==typeid(Aguila)) {
      Aguila &aguila = dynamic_cast<Aguila &> (a);
      aguila.situar ("Sector 2");
      aguila.chilla ();             // Miembro de Aguila
      return;
    }
}
```

Ejemplo final

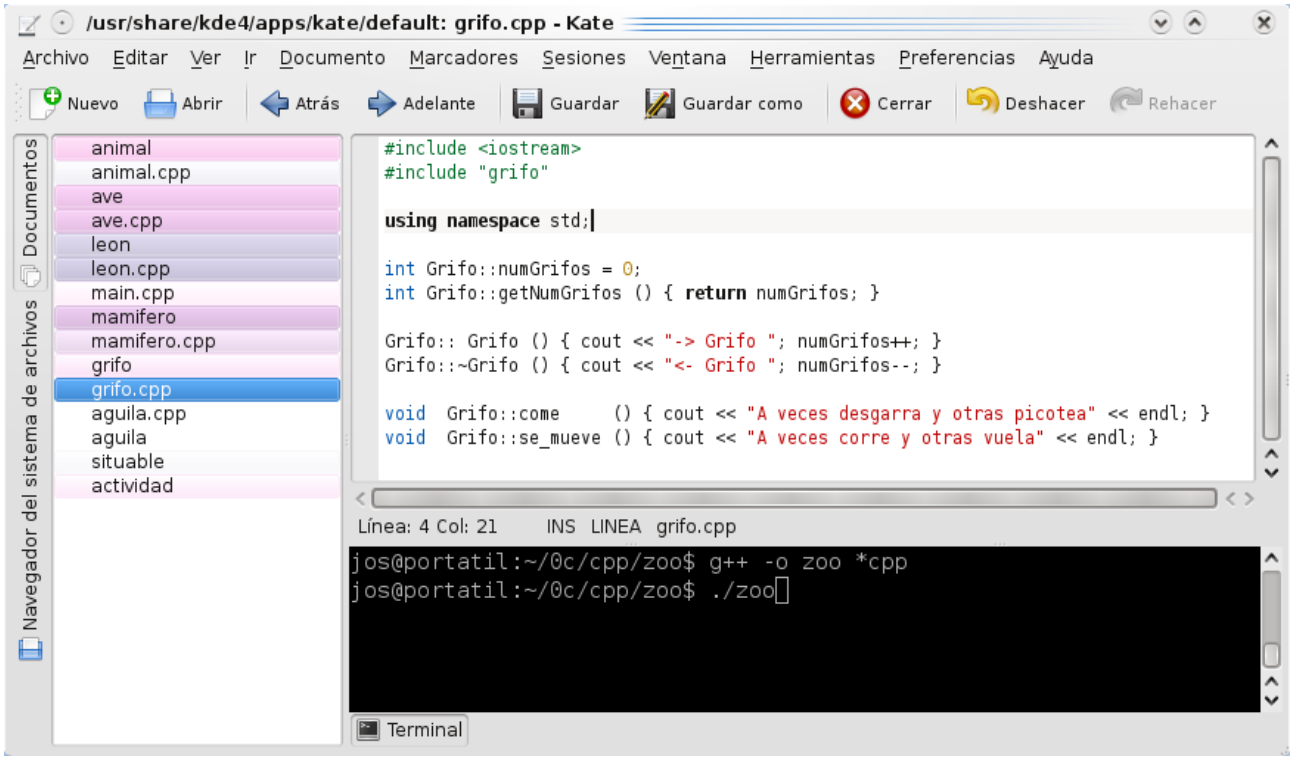
Archivos

Definición	Implementación
actividad	
situable	
animal	animal.cpp
mamifero	mamifero.cpp
ave	ave.cpp
leon	leon.cpp
aguila	aguila.cpp
grifo	grifo.cpp
	main.cpp

Diagrama UML de clases



Entorno de trabajo



Compilación

```
g++ *cpp -o zoo
```

Ejecucion

```
./zoo
```

Código

Clases abstractas (Interfaces)

actividad

```
#ifndef ACTIVIDAD
#define ACTIVIDAD


//// Clase abstracta
class Actividad {
  public:
    // Métodos virtuales puros
    virtual void come     () = 0;
    virtual void se_mueve () = 0;
};

#endif // ACTIVIDAD
```

situable

```
#ifndef SITUABLE
#define SITUABLE

// Las 2 líneas siguientes son necesarias para tipo string
#include <string>
using namespace std;

//// Clase abstracta
class Situable {
  protected:
    string zona;
  public:
    // Métodos virtuales puros
    virtual void   situar    (string  zona  =  "Sin zona"  ) = 0;
    virtual string situacion () = 0;
};

#endif // SITUABLE
```

Clase polimórfica

animal

```
#ifndef ANIMAL
#define ANIMAL

#include <iostream>
#include <string>
#include "actividad"
#include "situable"

using namespace std;

enum Sexo  {SIN_DATOS, MACHO, HEMBRA};


//// Clase polimórfica
class Animal: public Actividad, public Situable {
  private:
    static int numAnimales;

  protected:
    string id;
    Sexo sexo;
    int edad;

  public:
    static int getNumAnimales ();

     Animal ();
    ~Animal ();
     Animal (string id, Sexo sexo = SIN_DATOS, int edad = 0);

    void   situar    (string  zona  =  "Sin zona" );
    string situacion ();
    virtual void   come     ();    // Función virtual no pura. Clase polimórfica
    virtual void   se_mueve ();    // Función virtual no pura. Clase polimórfica

    void   setSexo (Sexo sexo);
    Sexo   getSexo ();
    void   setEdad (int edad);
    int    getEdad ();

    friend ostream& operator<< (ostream& , const Animal&);

};

#endif // ANIMAL
```

animal.cpp

```
#include <iostream>
#include "animal"

using namespace std;


// Miembros estáticos
int Animal::numAnimales = 0;
```

```
int Animal::getNumAnimales () { return numAnimales; }


// Constructores y destructor
Animal:: Animal () { cout << "\n-> Animal "; numAnimales++; }
Animal::~Animal () { cout << "<- Animal \n"; numAnimales--;}
Animal:: Animal (string id, Sexo sexo, int edad)  { numAnimales++;  this->id = id;
this->sexo = sexo;  this->edad = edad;}

// Métodos redefinidos
void   Animal::come      () {}
void   Animal::se_mueve  () {}
void   Animal::situar    (string  zona) { this->zona = zona; }
string Animal::situacion () { return zona; }


// Otros métodos
void Animal::setEdad (int edad)  { this->edad = edad;}
int  Animal::getEdad ()          { return edad; }
void Animal::setSexo (Sexo sexo) { this->sexo = sexo; }
Sexo Animal::getSexo ()          { this->sexo = sexo; }


// Sobrecarga de operador amigo <<
// No se indica el resolutor de ámbito ::
// pues la funcion no es miembro de la clase, sólo amiga
ostream& operator<< (ostream &out, const Animal &a) {
  string s;
  if (a.sexo)
    if (a.sexo==MACHO)  s="Macho";  else  s="Hembra";
  else
    s = "Sin datos";

  out << endl << "DATOS DEL ANIMAL" << endl;
  out << "Nombre : " << a.id << endl;
  out << "Sexo   : " << s << endl;
  out << "Edad   : " << a.edad << endl;
  return out;
}
```

Clases abstractas

mamifero

```
#ifndef MAMIFERO
#define MAMIFERO

#include "animal"


class Mamifero : public virtual Animal {
  private:
    int diasLactancia;
    static int numMamiferos;

  public:
    static int getNumMamiferos ();

     Mamifero ();
```

```
    ~Mamifero ();
     //Mamifero  (string id, Sexo sexo, int edad);

     void setLactancia (int diasLactancia);
     int getLactancia ();

     virtual void amamantar (int cachorros) = 0;
};

#endif // MAMIFERO
```

mamifero.cpp

```
#include <iostream>
#include "mamifero"

using namespace std;

// Miembros estáticos
int Mamifero::numMamiferos = 0;   // Inicialización
int Mamifero::getNumMamiferos () { return numMamiferos; }

// Definicion de métodos normales
Mamifero:: Mamifero () { cout << "-> Mamífero "; numMamiferos++; }
Mamifero::~Mamifero () { cout << "<- Mamífero "; numMamiferos--; }


// Otros métodos
void Mamifero::setLactancia (int diasLactancia)
                                    { this->diasLactancia = diasLactancia; }
int  Mamifero::getLactancia ()      { return diasLactancia; }
```

ave

```
#ifndef AVE
#define AVE

#include "animal"

class Ave : public virtual Animal {
  private:
    int diasIncubacion;
    static int numAves;

  public:
    static int getNumAves ();

     Ave ();
    ~Ave ();
    //Ave (string id, Sexo sexo, int edad);

    void setIncubacion (int diasIncubacion);
    int  getIncubacion ();

    virtual void incubar (int huevos) = 0;

};
```

```
#endif // AVE
```

ave.cpp

```
#include <iostream>
#include "ave"

using namespace std;

// Inicialización de miembros estáticos
int Ave::numAves = 0;
int Ave::getNumAves () { return numAves; }

// Definicion de métodos normales
Ave:: Ave () { cout << "-> Ave "; numAves++; }
Ave::~Ave () { cout << "<- Ave "; numAves--; }


// Otros métodos
void Ave::setIncubacion (int diasIncubacion)  { this->diasIncubacion = diasIncubacion; }
int  Ave::getIncubacion ()                    { return diasIncubacion; }
```

Clases "finales"

leon

```
#ifndef LEON
#define LEON

#include "mamifero"

// Las 2 líneas siguientes son necesarias para tipo string
#include <string>
using namespace std;


class Leon : public Mamifero {
  private:
    static int numLeones;
    string rugido;

  public:
    static int getNumLeones ();

     Leon ();
    ~Leon ();
     Leon (string id, Sexo sexo, int edad);

    void come     ();
    void se_mueve ();

    void setRugido (string rugido);
    void ruge ();

    void amamantar (int cachorros);
```

```
};

#endif // LEON
```

leon.cpp

```
#include <iostream>
#include <string>
#include "leon"

using namespace std;

// Miembros estáticos
int Leon::numLeones = 0;
int Leon::getNumLeones () { return numLeones; }

// Constructores y destructor
Leon:: Leon () { cout << "-> León "; numLeones++; }
Leon::~Leon () { cout << "<- León "; numLeones--; }
Leon:: Leon (string id, Sexo sexo, int edad) : Animal (id, sexo, edad) {  cout << "->
León "; numLeones++; }

// Métodos redefinidos
void Leon::come     ()  { cout << "\nEl león desgarra la presa"; }
void Leon::se_mueve ()  { cout << "\nEl león corre"; }
void Leon::amamantar (int cachorros) { cout << "\nAmamanta a " << cachorros << "
leones"; }

// Otros métodos
void Leon::setRugido (string rugido) { this->rugido = rugido; }
void Leon::ruge () { cout << endl << rugido << endl; }
```

aguila

```
#ifndef AGUILA
#define AGUILA

#include "ave"

// Las 2 líneas siguientes son necesarias para tipo string
#include <string>
using namespace std;


class Aguila : public Ave {
  private:
    static int numAguilas;
    string chillido;

  public:
    static int getNumAguilas ();

     Aguila ();
    ~Aguila ();
     Aguila  (string id, Sexo sexo , int edad);
```

```
    void come    ();
    void se_mueve();

    void setChillido (string chillido);
    void chilla ();

    void incubar (int huevos);


};

#endif // AGUILA
```

aguila.cpp

```
#include <iostream>
#include <string>
#include "aguila"

using namespace std;

// Miembros estáticos
int Aguila::numAguilas = 0;
int Aguila::getNumAguilas () { return numAguilas; }

// Constructores y destructor
Aguila:: Aguila () { cout << "-> Aguila "; numAguilas++; }
Aguila::~Aguila () { cout << "<- Aguila "; numAguilas--; }
Aguila:: Aguila (string id, Sexo sexo, int edad) : Animal (id, sexo, edad) {  cout << "-
> Aguila "; numAguilas++;  }

// Métodos redefinidos
void Aguila::come     () { cout << "\nEl aguila picotea la presa"; }
void Aguila::se_mueve () { cout << "\nEl aguila planea"; }
void Aguila::incubar (int huevos) { cout << "\nEl aguila incuba " << huevos << "
huevos"; }

// Otros métodos
void Aguila::setChillido (string chillido) { this->chillido = chillido; }
void Aguila::chilla () { cout << endl << chillido << endl; }
```

grifo

```
#ifndef GRIFO
#define GRIFO

#include "leon"
#include "aguila"


class Grifo : public Leon, public Aguila {
  private:
    static int numGrifos;
```

```
  public:
    static int getNumGrifos ();

     Grifo ();
    ~Grifo ();
     Grifo (string id, Sexo sexo, int edad);

    void  come     ();
    void  se_mueve ();

};

#endif // GRIFO
```

grifo.cpp

```
#include <iostream>
#include "grifo"

using namespace std;

// Miembros estáticos
int Grifo::numGrifos = 0;
int Grifo::getNumGrifos () { return numGrifos; }

// Constructores y destructor
Grifo:: Grifo () { cout << "-> Grifo "; numGrifos++; }
Grifo::~Grifo () { cout << "<- Grifo "; numGrifos--; }
Grifo:: Grifo (string id, Sexo sexo, int edad) : Animal (id, sexo, edad) {  cout << "->
Grifo "; numGrifos++;  }

// Métodos redefinidos
void  Grifo::come     () { cout << "\nEl grifo desgarra y picotea la presa"; }
void  Grifo::se_mueve () { cout << "\nEl grifo corre y vuela"; }
```

Bibliografía

C

- *Kernighan and Ritchie: The C Programming Language (Prentice-Hall, 1978, ISBN 0–13–110163–3, © 1978 AT&T).*
- *ISO/IEC 9899:1990*
- *ISO/IEC 9899:1999*

C++

- *Bjarne Stroustrup: The C++ Programming Language (2nd edition, Addison-Wesley Publishing Company, ISBN 0–201–53992–6, © 1991 AT&T)*
- *ISO/IEC 14882:1998*
- *ISO/IEC 14882:2003(E)*
- *http://www.cplusplus.com/doc/tutorial*
- *Aprenda C++ como si estuviera en primero. Universidad de Navarra.*