10.- **Constructores Virtuales.**

El C++ no se permite el uso de constructores virtuales. Esta limitación es impuesta más bien por dificultades técnicas de implementación que por dificultades de concepto. El problema es este; La invocación de una función virtual requiere de la definición de un **vptr** en la tabla de apuntadores virtuales **vtab**. Sin embargo, antes de que un objeto sea construido en memoria no existe tal tabla. Por otro lado la idea de constructores virtuales es confusa, pero tal vez sea necesario que figure en el lenguaje de alguna manera, por lo pronto, no esta disponible.

11.- **Destructores virtuales.**

C++ permite la declaración de un destructor virtual. Esto es porque un destructor es invocado justo antes de que un objeto desaparesca de memoria, el **vptr** va a estar disponible para su uso. Hay que hacer notar que el **vtab** es definido por clase, no por objeto, por lo tanto **vtab** va a estar presente después de ser destruido un objeto, pero no hay acceso a él. Por otro lado, como un destructor no puede tener argumentos, solo se puede declarar virtual un destructor por clase.

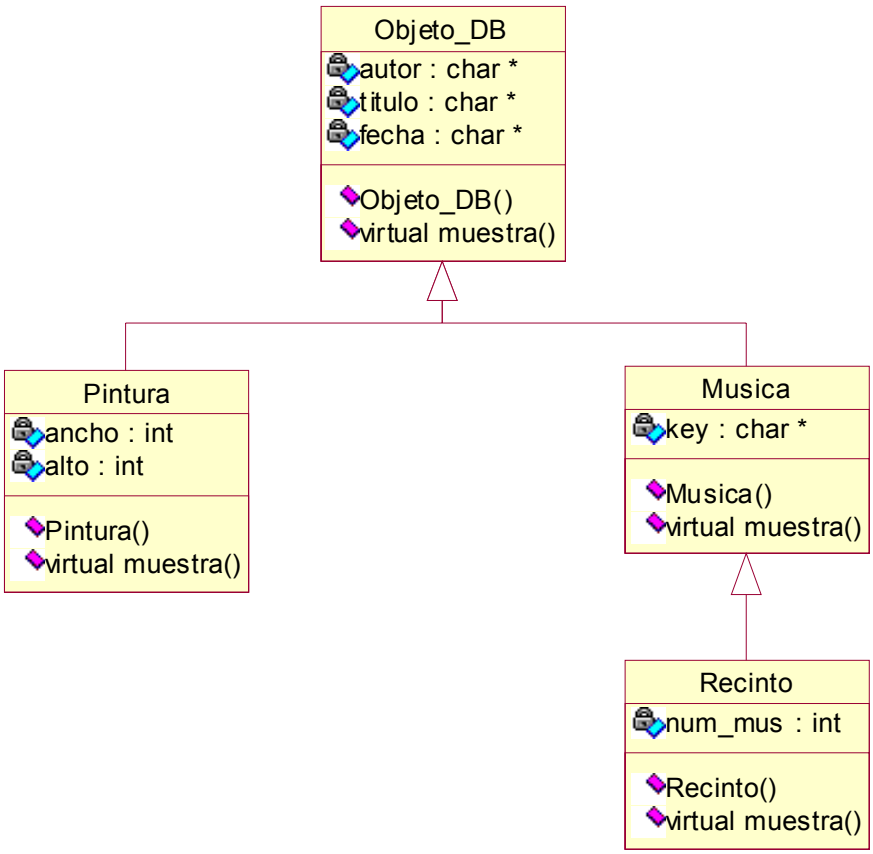
12.- **Un ejemplo de Polimorfismo**.

El polimorfismo no solo hace más simples los programas, también los hace más flexibles y robustos. En esta sección se va a presentar un programa el cual usa funciones virtuales, lo que nos va a permitir ilustrar la potencia del polimorfismo.

Considere una serie de clases usadas para la implementación de una base de datos. Las clases se organizan en una jerarquía, con la intención de permitir al programador accesar cualquier objeto en la base de datos, sin conocer todos los objetos involucrados.

Cualquier acceso a la base de datos inicia en el nodo más alto en el árbol. Cada canto en la base de datos tiene un autor y una fecha. Otra información puede variar a través de las clases, por ejemplo, el método usado para mostrar información acerca de un objeto. Esta función muestra(), es una buena candidata para aplicar polimorfismo, por lo tanto se implementa usando el mecanismo virtual.

A continuación se presenta el árbol de herencia.




```
// Programa PLCP95.H
// Un ejemplo de polimorfismo.

#ifndef DB_H_
#define DB_H_

#include <stdio.h>
#include <string.h>

class Objeto_DB {
    char autor[50];
    char titulo[50];
    char fecha[50];
public:
    Objeto_DB(char *, char *, char *);
    virtual void muestra();
};
```

```
class pintura : public Objeto_DB {
    int ancho;
    int alto;
public:
    pintura(char *, char *, char *, int, int);
    virtual void muestra();
};

class musica : public Objeto_DB {
public:
    musica(char *, char *, char *, char *);
    virtual void muestra();
};


class recinto : public musica {
    int num_mus;
public:
    recinto(char *, char *, char *, char *, int);
    virtual void muestra();
};

#endif // Clase DB_H_
```

A continuación se presenta la implementación, en un archivo diferente.

```
// Programa PLCP95.CPP
// Implementación de la clase Objeto_DB
#include "PLCP95.H"

Objeto_DB::Objeto_DB(char *quien, char *que, char *cuando)
{
    strcpy(autor, quien);
    strcpy(titulo, que);
    strcpy(fecha, cuando);
}

void Objeto_DB::muestra()
{
    printf("\nAutor  : %s", autor);
    printf("\nTitulo : %s", titulo);
    printf("\nFecha  : %s", fecha);
}
```

```
pintura::pintura(char *autor, char *titulo, char *fecha,
                 int w, int h) : Objeto_DB(autor, titulo, fecha)
{
    ancho = w;
    alto =  h;
}


void pintura::muestra()
{
    Objeto_DB::muestra();
    printf("\nTipo   : Pintura");
    printf("\nTamaño : Ancho = %d,  Altura = %d",
             ancho, alto);
}


musica::musica(char *autor, char *titulo, char *fecha, char *k)
        : Objeto_DB(autor, titulo, fecha)
{
    strcpy(key, k);
}


void musica::muestra()
{
    Objeto_DB::muestra();
    printf("\nTipo   : Musica");
    printf("\nLlave  : %s", key);
}


recinto::recinto(char *autor, char *titulo, char *fecha,
                 char *key, int tam) : musica(autor, titulo,
fecha, key)
{
    num_mus = tam;
}


void recinto::muestra()
{
    musica::muestra();
    printf("\nOtro   : Recinto de música %d músicos", num_mus);
}
```

Por último mostramos la aplicación, en otro archivo.

```
// Programa PLCP95A.CPP
// Aplicación
#include <iostream.h>
#include "PLCP95.H"

void muestra_dato(Objeto_DB &d)
{
    d.muestra();
}

void main()
{
    musica sinfonia("Beethoven, Ludwing van",
                    "Sinfonia # 6", "1824", "D menor");
    pintura pinta("da Vinci, Leonardo", "Mona Lisa",
                    "1503", 24, 36);
    recinto opus("Mosart, Wolfgang Amadeus", "Hoffmeister",
                    "1786", "d menor", 4);

    muestra_dato(sinfonia);
    cout << endl;
    muestra_dato(pinta);
    cout << endl;
    muestra_dato(opus);
}
```

OBSERVACIONES.

La salida nos resulta:

Autor : Beethoven, Ludwing van
Titulo : Sinfonia # 6
Fecha : 1824
Tipo   : Musica
Llave : D menor

Autor : da Vinci, Leonardo
Titulo : Mona Lisa
Fecha : 1503
Tipo   : Pintura
Tamaño : Ancho = 24, Altura = 36

Autor : Mosart, Wolfgang Amadeus
Titulo : Hoffmeister
Fecha : 1786
Tipo : Musica
Llave : d menor
Otro : Recinto de música 4 músicos

La clase base comparte las características comunes de todas las clases.

La clase musica hereda de la clase base y agrega las características de **tipo** y **llave**.

La clase pintura agrega el tamaño; **ancho** y **alto**.

La clase recinto agrega número de músicos; **num_mus**.

Usando la función **muestra()** desplegamos toda la información almacenada en la base de datos, de acuerdo a la clase de objeto especificado.

El polimorfismo nos permite una referencia genérica a través de la estructura del programa, por la cual podemos llamar a la función **muestra_dato**(Objeto_DB &D), a partir de la clase base, y seguir la trayectoria hasta la clase derivada especificada como argumento. Este método de trabajo es altamente recomendado en C++, ya que nos simplifica bastante el código.

13.- **Resolución de ámbito en el polimorfismo**.

Existe una gran libertad en la llamada a una función virtual. Una función virtual puede ser invocada por casi cualquier función que tenga los privilegios de acceso necesarios. Una función virtual en una clase derivada, tiene la capacidad de invocar una función virtual de la clase base con el mismo nombre sin producir un lazo infinito y propiciar la caida del sistema. Para invocar una función virtual de la clase base desde una clase derivada, es necesario usar el operador de resolución de ámbito (**::**), de tal manera de indicar de manera explicita la clase de la cual estamos requiriendo sea ejecutada su función virtual. En este caso el operador de ámbito le dice al compilador que no use el mecanismo virtual en esta llamada.
Un ejemplo se presentó en el programa anterior, en la función:

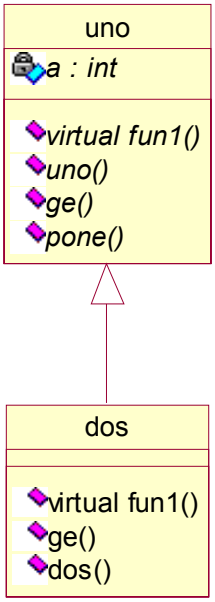
```
void muestra_dato(Objeto_DB &d)
{
    d.muestra();
}
```

Consideremos el caso en que **d** se refiere a un objeto del tipo **musica**. La función **music::**muestra() sobre-escribe la función de la clase base **Objeto_DB**::muestra(), con ello, la tabla de funciones viertuales convierte la llamada a la función virtual **Objeto_DB**::muestra(), en la llamada a la función virtual **musica**::muestra(). La cual aparece dentro del cuerpo de la función **musica::**muestra(). Esto deberá causar un ciclo infinito, ya que la llamada a la función **Objeto_DB**::muestra() va a causar de nuevo la llamada a la función **musica**::muestra(). Al usar el operador de resolución de ámbito, forza al compilador a generar en tiempo de compilación la llamada a la función, con lo cual se salta el mecanismo virtual, que ocurre en tiempo de ejecución. Evitandose entrar en un ciclo infinito de llamadas.

14.- **Uso de funciones virtuales con funciones NO virtuales.**

Usando funciones virtuales y funciones no virtuales juntas en una jerarquía de clases, puede causar confusión de manera inicial para el novicio. Para ejemplificar lo anterior, considere el siguiente árbol de herencia.

Una herencia de clases usando funciones virtuales y no virtuales juntas.



El árbol de clases anterior, lo implementamos como sigue:

```
// Programa PLCP96.CPP
```

```
#include <iostream.h>

class uno {
    int a;
public:
    uno(int x = 0) { a = x; }
    virtual void fun1() {
        cout << "Función uno::fun1()" << ", "
             << "a = " << a << endl;
    }
    void ge() {
        cout << "Función uno::ge()" << endl;
    }
    int pone() { return a; }
};


class dos : public uno {
    int b;
public:
    dos(int x = 0, int y = 0) : uno(x) { b = y; }
    virtual void fun1() {
        cout << "Función dos::fun1()" << ", "
             << "a = " << pone() << ", b = " << b << endl;
    }
    void ge() {
        cout << "Función dos::ge()" << endl;
    }
};


void fun_uno(uno &x)
{
    x.fun1();
    x.ge();
}

void main()
{
    uno A(4);
    dos B(6, 8);

    fun_uno(A);
    fun_uno(B);
}
```

OBSERVACIONES.

La salida nos resulta:

```
Función uno::fun1(), a = 4
Función uno::ge()
Función dos::fun1(), a = 6, b = 8
Función uno::ge()
```

Podemos observar que en el caso de la función virtual se inicia la trayectoria en la clase base **uno**, hasta la clase derivada, por ello, al llamar a un objeto de la clase **uno**, se llama a la función virtual de la clase **uno**, y al llamar a un objeto de la clase **dos** inicia buscando la función virtual a partir de la clase base, hasta la clase derivada, y ejecuta esta verisión de la función.

En el caso de llamar a una función no virtual, como el parámetro de la función **fun_uno()** es del tipo de la clase base, no hay una trayectoria a seguir, entonces, llama de manera inmediata a la función definida en la clase base.

En el siguiente ejemplo se muestra una forma de accesar a la función **dos::ge()**.

```
// Programa PLCP96A.CPP
#include <iostream.h>

class uno {
    int a;
public:
    uno(int x = 0) { a = x; }
    virtual void fun1() {
        cout << "Función uno::fun1()" << ", "
             << "a = " << a << endl;
    }
    void ge() {
        cout << "Función uno::ge()" << endl;
    }
    int pone() { return a; }
};


class dos : public uno {
    int b;
public:
    dos(int x = 0, int y = 0) : uno(x) { b = y; }
    virtual void fun1() {
        cout << "Función dos::fun1()" << ", "
             << "a = " << pone() << ", b = " << b << endl;
    }
```

```
    void ge() {
        cout << "Función dos::ge()" << endl;
    }
};


void fun_uno(uno &x)
{
    x.fun1();
    x.ge();
}

void fun_dos(dos &y)
{
    y.fun1();
    y.ge();
}

void main()
{
    uno A(4);
    dos B(6, 8);

    fun_uno(A);
    fun_uno(B);
    cout << endl << endl;
    fun_dos(B);
}
```

OBSERVACIONES.

Ahora la salida nos resulta:

```
Función uno::fun1(), a = 4
Función uno::ge()
Función dos::fun1(), a = 6, b = 8
Función uno::ge()


Función dos::fun1(), a = 6, b = 8
Función dos::ge()
```

La solución es contar con una función desde donde de manera explicita se llame a la clase **dos**, este es el caso de la función **fun_dos**(dos &y).

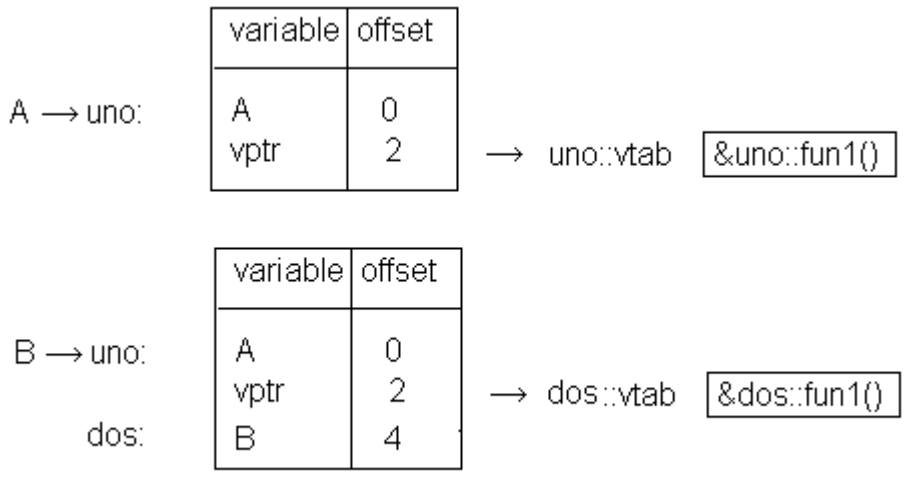
15.- Una vista en memoria de las estructuras **vptr** y **vtab**.

Es fácil entender como el compilador localiza las funciones virtuales en tiempo de ejecución considerando la estructura de la memoria usada en los objetos de las clases que involucran funciones virtuales.

En el ejemplo anterior, la llamada a la función **fun_uno**(uno&), mediante un objeto de la clase **dos**, resultó en la llamada a las funciones **dos::**fun1() y **uno**::ge(). Esto sucede ya que en la llamada a la función:

fun_uno(B)

la referencia a **dos&** es convertido de manera implicita a **uno&**, entonces, el compilador se da cuenta que **uno**::ge() no es una función virtual, por lo tanto, genera código para llamar a la función uno::ge() directamente. La llamada a la función uno::fun1() en la función **fun_uno(uno&)** se maneja de manera diferente ya que el compilador se da cuenta que la función fun1() es declarada como virtual en la clase **uno**, lo cual provoca la generación de código para el uso del mecanismo virtual. Las dos funciones virtuales son puestas en memoria como sigue:



En tiempo de ejecución, el vector **vptr** indica a que función se tiene acceso, dependiendo del tipo de objeto pasado como argumento. En el caso de ser, por ejemplo, un objeto de la clase dos, la función a ser llamada, considerando la función virtual fun1() de la clase anterior, será la función: **dos::fun1()**. El vector **vptr**, de la clase dos se almacena en el mismo offset que el **vptr** de la clase uno, entonces, la función **fun_uno(uno&)** tiene acceso a una función virtual dependiendo del tipo de objeto que

sea pasado como argumento a dicha función, Note sin embargo, que los **vptr's** para la clase **uno** y para la clase **dos** manejan diferentes tablas, **vtab**.

Si observamos la pila mediante un depurador, veremos lo siguiente para el programa anterior:

offset	obj	direc.	cont.	vtab	Dir. ini.	fun. virtuales
2		FFF4	0004			
0	A	FFF2	010F	DS:010F	CS:0463	uno::fun1()
4		FFF0	0008			
2		FFEE	0006			
0	B	FFEC	010D	DS:010D	CS:040F	dos::fun1()

Si observamos el código, vemos que las funciones virtuales son llamadas mediante la instrucción: call [BX]. Lo que significa que la dirección de ejecución puede ser manejada durante la ejecución del programa, y no necesariamente en la compilación.