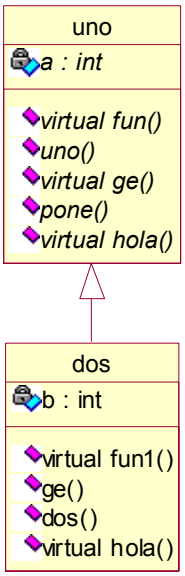
16.- **Las funciones virtuales no tienen que ser necesariamente sobre-escritas.**

La declaración de una función virtual forza al compilador a generar vectores a través del apuntador **vptr** para la invocación de las funciones. La palabra reservada, **virtual** le dice al compilador que la función puede ser sobre-escrita en una clase base. ¿Qué sucede si no hay una clase base?, o qué sucede si no hay clases derivadas?. Una función virtual, no necesariamente tiene que ser sobre-escrita.

En la jerarquía de clases del programa PLCP96, al ser puesta en memoria la clase **dos**, ésta contenía completamente a un objeto de la clase **uno** en ella. El **vptr** para las 2 clases fué diferente. En cualquier caso, en la clase **uno** se paso por alto el hecho que fué usada como una clase base para otras clases. Una función virtual no tiene que ser sobre-escrita en una clase derivada. De hecho, una función virtual declarada en una clase no tiene que ser definida, en cuyo caso, es una función virtual pura. Ciertamente, una función virtual pura no puede ser invocada directamente, si no se quiere ocasionar un error en tiempo de ejecución.

Las clases, en librerias comerciales de clases, con frecuencia tienen muchas de sus funciones miembro declaradas como **virtual**, de tal manera de permitir al usuario la opción de poder sobre-escribirlas, lo que aumenta la flexibilidad en su manejo, mejorando su comportamiento en tiempo de ejecución.

Considere una clase derivada, la cual sobre-escribe solamente algunas de las funciones virtuales en su clase base. Para ver como maneja el compilador estas funciones virtuales, analicemos el siguiente árbol de herencia.



El árbol de herencia de la figura anterior es implementado de acuerdo al siguiente código.

```
// Programa PLCP97.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;
    }
    virtual void ge() {
        cout << "Función uno::ge()" << endl;
    }
    virtual void hola() {
        cout << "Función uno::hola()" << 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;
    }
    virtual void hola() {
        cout << "Función dos::hola()" << endl;
    }
};


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

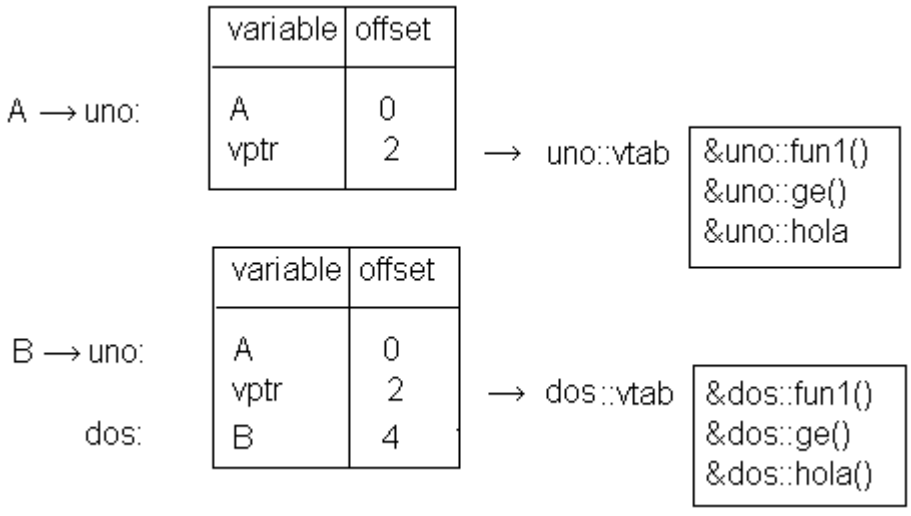
```
void main()
{
    uno A(4);
    dos B(6, 8);

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

OBSERVACIONES.

La salida nos resulta:

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

La clase **dos** no sobre-escribe todas las funciones virtuales definidas en la clase **uno**, ésto origina que el compilador genere una tabla de vectores **vtab**, no usual para la clase **dos**, de tal forma de generar la corrección necesaria en el mecanismo virtual. En la siguiente figura se muestra como el Borland C++ coloca en memoria los objetos




Analizando los punteros en **dos::vtab**, es de esperarse solo ver punteros a dos::fun1() y a dos::hola(), ya que son las únicas funciones virtuales declaradass en la clase **dos**. Sin embargo, se puede apreciar que el compilador también generó un puntero a la función dos::ge(), ya que el puntero no fue sobre-escrito en la clase **dos**. Esto ocasiona que uno::ge() se convierta en la función por default para **dos**, también, esto propicia que

todas las entradas en uno::vtab y dos::vtab ocurrán en el mismo orden y con el mismo offset, garantizando la llamada a la función correcta cuando se da una referencia a la clase **uno** o a la clase **dos**.

Podemos ver en la salida del programa anterior que en ambos casos se llamó a la función uno::ge(), la cual no fué sobre-escrita en la clase **dos**.

Si observamos el contenido de la pila:

| offset | obj. | direc. | cont. | vtab | Dir. ini. | fun. virtuales |
|---|---|---|---|---|---|---|
| 2 | | FFF4 | 0004 | | | |
| | | | | DS:012D | CS:044A | uno::hola() |
| | | | | DS:012B | CS:03CA | uno::ge() |
| 0 | A | FFF2 | 0129 | DS:0129 | CS:040C | uno::fun1() |
| 4 | | FFF0 | 0008 | | | |
| 2 | | FFEE | 0006 | | | |
| | | | | DS:0127 | CS:03EB | dos::hola() |
| | | | | DS:0125 | CS:03CA | dos::ge() |
| 0 | B | FFEC | 0123 | DS:0123 | CS:0376 | dos::fun1() |

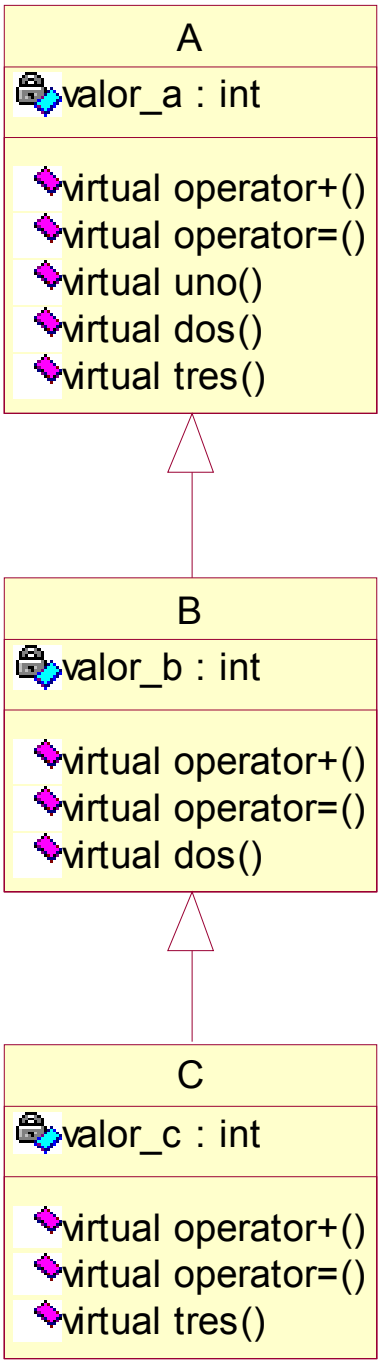
Las funciones son llamadas de la siguiente forma:

```
fun1()    call [BX]
ge()      call [BX+02]
hola()    call [BX+04]
```

17.- **Operadores Virtuales.**

Cuando los operadores se implementan como funciones miembro o funciones **friend**, es posible declararlos virtuales.

Como un ejemplo, considere el árbol de herencia, en él cual cada clase utiliza 2 operadores miembro virtuales.


A
valor_a : int
virtual operator+()
virtual operator=()
virtual uno()
virtual dos()
virtual tres()
B
valor_b : int
virtual operator+()
virtual operator=()
virtual dos()
C
valor_c : int
virtual operator+()
virtual operator=()
virtual tres()

Cada clase en el árbol de herencia usa operadores sobrecargados, como se muestra en el siguiente código.

```
// Programa PLCP98.CPP
// Operadores virtuales sobrecargados.
#include <iostream.h>
class A {
protected:
    int valor_a;
public:
    A(int i) { valor_a = i; }
    virtual A& operator+(A&);
    virtual A& operator=(A&);
    virtual int uno() { return valor_a; }
    virtual int dos() { return 0; }
    virtual int tres() { return 0; }
    void muestra() {
        cout << "Valor_a = " << valor_a
             << endl << endl;
    }
};


class B : public A {
protected:
    int valor_b;
public:
    B(int i, int j) : A(i) { valor_b = j; }
    virtual A& operator+(A&);
    virtual A& operator=(A&);
    virtual int dos() { return valor_b; }
    void muestra() {
        cout << "Valor_a = " << valor_a << endl;
        cout << "Valor_b = " << valor_b << endl
             << endl;
    }
};


class C : public B {
    int valor_c;
public:
    C(int i, int j, int k) : B(i, j) { valor_c = k; }
    virtual A& operator+(A&);
    virtual A& operator=(A&);
    virtual int tres() { return valor_c; }
    void muestra() {
        cout << "Valor_a = " << valor_a << endl;
```

```
        cout << "Valor_b = " << valor_b << endl;
        cout << "Valor_c = " << valor_c << endl
             << endl;
    }
};

A& A::operator+(A &t)
{
    valor_a += t.uno();
    return *this;
}

A& A::operator=(A &t)
{
    valor_a = t.uno();
    return *this;
}

A& B::operator+(A &t)
{
    valor_a += t.uno();
    valor_b += t.dos();
    return *this;
}

A& B::operator=(A &t)
{
    valor_a = t.uno();
    valor_b = t.dos();
    return *this;
}

A& C::operator+(A &t)
{
    valor_a += t.uno();
    valor_b += t.dos();
    valor_c += t.tres();
    return *this;
}

A& C::operator=(A &t)
{
    valor_a = t.uno();
    valor_b = t.dos();
    valor_c = t.tres();
    return *this;
}
```

```
void suma_obj(A& obj1, A& obj2)
{
    obj1 = obj1 + obj2;
}


void main()
{
    A a(10);
    B b(20, 30);
    C c(40, 50, 60);

    a.muestra();
    b.muestra();
    c.muestra();

    suma_obj(a, b);
    suma_obj(b, c);
    suma_obj(c, b);

    a.muestra();
    b.muestra();
    c.muestra();

    a = a + c;
    c = c + a;

    a.muestra();
    c.muestra();
}
```

OBSERVACIONES.

Note el uso de la palabra **protected** en las clases **A** y **B**, lo cual se hizo para poder tener acceso a los miembros privados de estas clases a través de la clase derivada **C**.

Si observamos la pila:

| offset | obj. | direc. | cont. | vtab | Dir. ini. | fun. virtuales | |
|---|---|---|---|---|---|---|---|
| 2 | | FFF4 | 000A | | | | |
| | | | | DS:0108 | CS:069F | A::tres() | |
| | | | | DS:0106 | CS:06AB | A::dos() | |
| | | | | DS:0104 | CS:0672 | A::uno() | |
| | | | | DS:0102 | CS:02DF | A::operator=() | |
| 0 | a | FFF2 | 0100 | DS:0100 | CS:02C2 | A::operator+() | |

| offset | obj. | direc. | cont. | vtab | Dir. ini. | fun. virtuales | |
|---|---|---|---|---|---|---|---|
| 4 | | FFF0 | 001E | | | | |
| 2 | | FFEE | 0014 | | | | |
| | | | | DS:00FE | CS:069F | B::tres() | |
| | | | | DS:00FC | CS:0681 | B::dos() | * |
| | | | | DS:00FA | CS:0672 | B::uno() | |
| | | | | DS:00F8 | CS:0323 | B::operator=() | |
| 0 | b | FFEC | 00F6 | DS:00F6 | CS:02FC | B::operator+() | |
| 6 | | FFEA | 003C | | | | |
| 4 | | FFE8 | 0032 | | | | |
| 2 | | FFE6 | 0028 | | | | |
| | | | | DS:00F4 | CS:0690 | C::tres() | * |
| | | | | DS:00F2 | CS:0681 | C::dos() | |
| | | | | DS:00F0 | CS:0672 | C::uno() | |
| | | | | DS:00EE | CS:037B | C::operator=() | |
| 0 | c | FFE4 | 00EC | DS:00EC | CS:034A | C::operator+() | |

NOTA: De las funciones, son las únicas definidas en cada clase.

Se usan (*//*) para especificar comentarios a la salida.

La salida nos resulta:

```
valor_a = 10        // Valores de inicialización
```

```
Valor_a = 20
Valor_b = 30

Valor_a = 40
Valor_b = 50
Valor_c = 60        // Valores de inicialización.

Valor_a = 30        // Resultado después de llamar a suma_obj()
                    // a = a + b;  10 + 20 = 30
Valor_a = 60        // b = b + c;  20 + 40 = 60
Valor_b = 80                      30 + 50 = 80

Valor_a = 100       // c = c + b;  40 + 60 = 100
Valor_b = 130                     50 + 80 = 130
Valor_c = 60        // Resultado después de llamar a suma_obj()

Valor_a = 130       // a = a + c;   30 + 100 = 130

Valor_a = 230       // c = c + a;  100 + 130 = 230
Valor_b = 130                                 130
Valor_c = 60                                   60
```

El código anterior muestra algunos puntos interesantes.

1.- Todos los operadores sobrecargados son declarados con un parámetro del tipo A&. Esto es necesario ya que a través de este tipo de parámetro pueden seguir la trayectoria desde la clase A.
El mecanismo virtual es utilizado solamente cuando una función virtual en una clase derivada tiene el mismo nombre y tipo de parámetros que la función virtual de la clase base.

2.- El código en main() muestra como se pueden pasar objetos como argumentos a una función aplicando el concepto de polimorfismo a los objetos dentro de la función global **suma_obj(A&, A&)**.

3.- Un uso explicito de operadores virtuales se muestra en las lineas:

a = a + c;
c = c + a;

Las operaciones realizadas se muestran en el lado derecho de cada salida, usando el símbolo de (*//*), para denotar que esta información fué colocada solo como explicación a la salida y no forma parte de la salida en sí.

## 18.- **Función virtual vs función no virtual.**

Se puede declarar cualquier función miembro como virtual con las siguientes restricciones.

a) Constructores.
b) Funciones miembro **static**.

Considerando estas restricciones, podemos pensar que cualquier otro tipo de función puede ser declarado como virtual, esto es cierto en parte, sin embargo, al considerar el mecanismo del enlace tardío que va involucrado, nos hace reflexionar en que debemos de considerar el uso de funciones virtuales, solamente cuando éste es necesario. Sin embargo, la decisión al considerar si una función virtual es necesaria no siempre es obvio. Por ejemplo, se puede pensar en descartar en el uso de funciones virtuales a aquellas clases que no son diseñadas para derivar desde ellas otras clases, sin embargo, no se puede saber que requerimientos se tendrán en el futuro.

A continuación se trata de plantear algunos lineamientos para seleccionar entre el uso de funciones virtuales y funciones no virtuales.

Se deberá considerar el uso de funciones virtuales al menos en los siguientes casos.

1) En el diseño de clases que se van a encontrar en la parte superior del árbol o muy cerca de ella.

2) Para funciones que describen los atributos de una clase y que dependen de la estructura o tipo de una clase.

3) Para funciones donde se implementa entrada y salida para una clase.

4) Para funciones que tienen acciones definidas solamente para una clase específica.

Como un ejemplo, el cual puede servir como referencia, para definir un criterio de selección en un caso en particular, podemos considerar una base de datos, semejante como la clase **Objeto_DB**, cada clase en el árbol de herencia puede ser aumentada con funciones para identificar un objeto, de tal manera de poder leer información de él desde una terminal, de tal forma que sea capaz de contestar a preguntas de usuarios, etc. Considere el dotar a la clase raiz **Objeto_DB** con las siguientes funciones y operadores.

* mostrar()

* iden()

* por_autor()

* fecha()

* imprime()

* lee_desde()

* exhibe_a()

* arbol_ancestros()

Algunas de estas, pueden ser candidatos para un ambiente polimórfico y otras no.

Analicemos cada una de ellas.

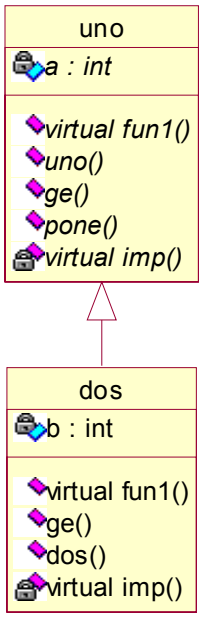
| | |
|---|---|
| muestra() | La función muestra() imprime el contenido de una clase sobre la terminal del usuario. Cada clase tiene una estructura interna diferente que la de sus ancestros, por lo tanto, de acuerdo a los lineamientos 1, 2, y 3 esta función necesita ser polimórfica. |
| iden() | Cada clase es un nuevo tipo, por lo que deberá regresar de manera correcta su identificador, de acuerdo al lineamiento 2, se recomienda declararla como virtual. |
| por_autor() | El objeto regresado por esta función es una cadena declarada en la clase **Objeto_DB**. El nombre del autor no esta en función del tipo de objeto, entonces, no requiere un medio polimórfico. |
| fecha() | La fecha de creación de una pieza de arte es probablemente no dependiente del tipo de objeto considerado, sin embargo, no se puede estar absolutamente seguro de este hecho. Esta función regresa el año como un valor entero. Pero que pensar de objetos con un período de creación de varios años?. |
| imprime() | Usando el lineamiento 3, esta función usada para la salida de una clase, siempre deberá ser virtual. |
| lee_desde() | Usando el lineamiento 3, esta función usada para la entrada de una clase, siempre deberá ser virtual. |

| | |
|---|---|
| exhibe_a() | Esta función deberá probablemente describir el museo u otro lugar en el cual un objeto es exhibido. Este situación probablemente es dependiente de una clase, por lo que seguramente se deberá requerir una función virtual. |
| arbol_ancestros() | Esta función puede ser usada como una función de rastreo (debugging) para indicar la clase base de una función. Por lo que deberá soportar herencia. Para ello se usa una función polimórfica. |

19.- **Las funciones virtuales también pueden ser privadas**.

El hecho de definir una función virtual por lo general, con el objetivo de crear una interface de clase para el usuario consistente y polimórfica, no significa, que las funciones virtuales deben ser siempre accesibles por el usuario. Si se tiene una clase en la cual una función virtual es diseñada para ser invocada exclusivamente desde otra función miembro, la función virtual puede ser declarada privada. En el siguiente árbol de herencia se ilustra esta situación.

Herencia de clases usando funciones virtuales privadas




A continuación se muestra la implementación de este árbol de herencia.

```
// Programa PLCP99.CPP
```

```
#include <iostream.h>

class uno {
    int a;
    virtual void imp() {
        cout << "Función uno::imp()"
             << ", LLamada via uno::fun1()" << ", "
             << "a² = " << a*a << endl << endl;
    }
public:
    uno(int x = 0) { a = x; }
    virtual void fun1() {
        imp();
        cout << "Función uno::fun1()" << ", "
             << "a = " << a << endl;
    }
    void ge() {
        cout << "Función uno::ge()" << endl;
    }
    int pone() { return a; }
};

class dos : public uno {
    int b;
    virtual void imp() {
        cout << "Función dos::imp()"
             << ", LLamada via dos::fun1()" << ", "
             << "a² = " << pone()*pone() << ", "
             << "b² = " << b*b << endl << endl;
    }
public:
    dos(int x = 0, int y = 0) : uno(x) { b = y; }
    virtual void fun1() {
        imp();
        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);
    cout << endl << endl;
    fun_uno(B);
}
```

OBSERVACIONES.

La salida nos resulta:

```
Función uno::imp(), LLamada via uno::fun1(), a² = 16

Función uno::fun1(), a = 4
Función uno::ge()


Función dos::imp(), LLamada via dos::fun1(), a² = 36, b² = 64

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

Tanto la clase **uno** como la clase **dos**, tienen una función virtual privada **imp()**, la cual no puede ser llamada de manera directa por un usuario de la clase uno, o de la clase dos, solo pueden ser accesadas por medio de la función virtual fun1(), virtual también de ambas clases.

El declarar una función virtual privada no va a producir ningún error, sin embargo hay que hacer notar que el no declarar la función imp() como virtual, va a producir el mismo resultado. Esto se debe a que una vez definida en tiempo de ejecución la función virtual correspondiente, ésta llama a la función miembro privada de su clase, sea o no virtual.