UNIDAD 10

Polimorfismo.

1.- **Introducción.**

En el C++ una función polimórfica puede asociarse con más de una función, lo cual se determina en el momento de que un objeto es pasado como argumento a la función polimórfica, ésto significa que existe un mecanismo diferente al de ANSI C para realizar una llamada de este tipo. En C++, una llamada a una función, solo es **indicada** en el código fuente, sin especificar de manera precisa la función a ser llamada. A este proceso se le conoce como **Enlace tardío**. En la mayoria de los lenguajes tradicionales, tales como C y Pascal, el compilador siempre llama de manera específica a una función particular, basandose en el código fuente. El enlazador entonces llama al identificador que representa a la función y le asigna una dirección física. A este proceso se le conoce como **Enlace inmediato**, ya que el identificador es asociado con una dirección física antes del tiempo de ejecución, ésto se hace durante la compilación y el enlazado.

El problema en el **enlace inmediato** es que el programador deberá predecir que objeto deberá ser usado en la llamada a cada función y para cada situación en particular. Esta forma de hacer una llamada a una función, no solo es limitada, sino en ocasiones imposible de predecir. Desplazando el proceso de enlace hasta en tiempo de ejecución, el código es el encargado de seleccionar el identificador de la función, ésto es la dirección de la función a ejecutarse. Si comparamos este proceso con el tradicional, nos damos cuenta que el e**nlace inmediato** es más rápido, ya que en el **enlace tardío**, existe un retardo, ocasionado al determinar en tiempo de ejecución el pase de argumentos al stack, en el proceso de llamada a la función, y el limpiado del stack al salir de la función, sin embargo, su flexibilidad compensa este hecho.

El problema con el **enlace tardío** es obiamente su eficiencia en tiempo de ejecución ya que el código deberá determinar en tiempo de ejecución cual función va a invocar, y entonces invocarla. Algunos lenguajes como Smalltalk usan exclusivamente el enlace tardío, lo que lo hace un lenguaje extremadamente potente, pero lento. En el polo opuesto, ANSI C usa **enlace inmediato**, es muy rápido pero carece de flexibilidad.

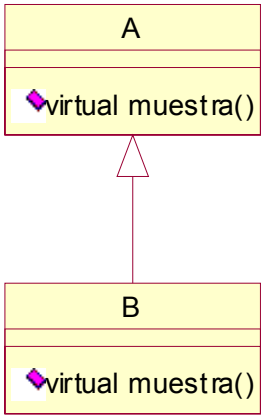
2.- **C++ es un lenguaje híbrido.**

C++ no es un lenguaje procedural tradicional como PASCAL, ni tampoco es un lenguaje puro, Orientado a Objetos. C++ es un lenguaje híbrido, usa ambos tipos de enlace; **enlace inmediato** y **enlace tardío**, brindando al programador las mejores características de cada uno de ellos, siendo éste el que selecciona el método a utilizar. Cuando se escribe código en donde la llamada a una función se pueda definir desde el inicio se recomienda el **enlace inmediato**, para situaciones más complejas se puede seleccionar el **enlace tardío**, logrando un compromiso entre flexibilidad y velocidad.

3.- **Funciones virtuales.**

En C++, para especificar **enlace tardío** en una función se hace declarandola como **virtual**. El **enlace tardío** tiene sentido en C++ solamente para objetos que son parte de una herencia de clases. Si se declara una función **virtual** en una clase la cual no es usada como clase base, es sintacticamente correcto, pero resulta en una innecesaria sobrecarga en tiempo de ejecución.

Por ejemplo, en el siguiente árbol de herencia se muestra la declaración de una función como **virtual**.




```
// Programa PLCP87.CPP
// Funciones virtuales.
#include <stdio.h>

class A {
public:
    virtual void muestra() {
        puts("\nclase A");
    }
};


class B : public A {
public:
    virtual void muestra() {
        puts("\nclase B");
    }
};
```

```
void imprime(A* a)
{
    a -> muestra();
}


void main()
{
    A *a = new A;
    B *b = new B;

    a -> muestra();
    b -> muestra();

    imprime(a);
    imprime(b);

    delete a;
    delete b;
}
```

OBSERVACIONES.

La salida nos resulta:

clase A

clase B

clase A

clase B

El comportamiento del polimorfismo de la función miembro **muestra()** tanto en la clase **A** como en la clase **B** no es obvio cuando solo se esta viendo la función main(), sucede lo mismo en la función **imprime()**, en la cual es imposible predecirlo por el solo hecho de observar su código.

Debido al mecanismo virtual definido en las funciones de la clase **A** y en las de la clase **B**, es posible llamar a la función **A::muestra()** o a la función **B::muestra()**, dependiendo del tipo de objeto con que se haga referencia a las funciones.

Si modificamos el programa, eliminando el mecanismo virtual, de acuerdo al siguiente código:

```
// Programa PLCP87A.CPP
// Funciones virtuales.
#include <stdio.h>

class A {
public:
    void muestra() {
        puts("\nclase A");
    }
};


class B : public A {
public:
    void muestra() {
        puts("\nclase B");
    }
};


void imprime(A* a)
{
    a -> muestra();
}


void main()
{
    A *a = new A;
    B *b = new B;

    a -> muestra();      // Usa A::muestra()
    b -> muestra();      // Usa B::muestra()

    imprime(a);          // Usa A::muestra()
    imprime(b);          // Usa B::muestra()

    delete a;
    delete b;
}
```

OBSERVACIONES.

La salida nos resulta:

clase A

clase B

clase A

clase A

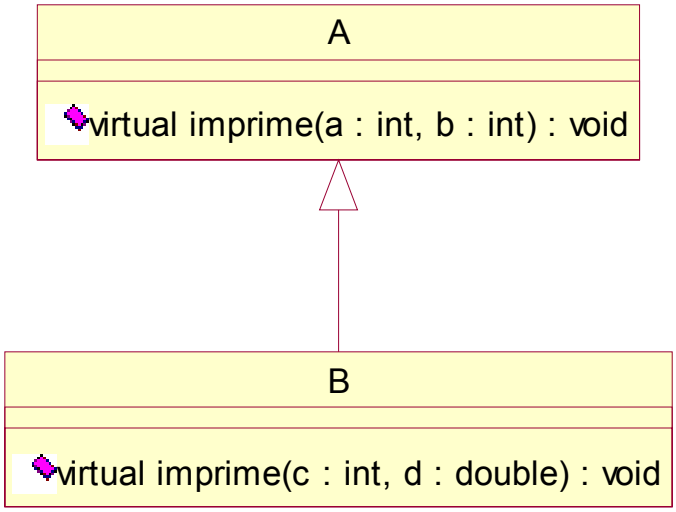
Como podemos ver, al eliminar el mecanismo virtual, solo se llama a la función **A::muestra()** de la clase **A**, ya que en este caso opera la conversión de tipos y B* es convertido a A* y entonces es llamada la función A::muestra() en cualquier caso.

Al declarar una función como virtual, no significa que va a ser sobre-escrita por la de una clase derivada, lo que significa es que el polimorfismo se va a propagar desde la parte superior del árbol de herencia hacia abajo, la condición para que ésto suceda es que cada clase deberá declarar la misma función como virtual.

4.- **Sobre-escritura de una función.**

En el ejemplo del programa PLCP87.CPP, la función virtual B::muestra() es seleccionada de manera dinámica en tiempo de ejecución dentro de la función ::imprime(). Podemos entonces decir que la función B::muestra() virtualmente sobre-escribe a la función A::muestra(). Una función declarada en una clase derivada sobre-escribe una función virtual declarada en una clase base, solamente si tiene el mismo nombre y usa el mismo número y tipos de argumentos como la función virtual de la clase base. En el caso donde un argumento es diferente, entonces, es considerada como una función diferente, y no ocurre sobre-escritura.

Como muestra considere el árbol de herencia siguiente:



Su código se muestra a continuación.

```
// Programa PLCP88.CPP
// Demuestra uso de funciones virtuales con argumentos
// diferentes.
#include <stdio.h>

class A {
public:
    virtual void imprime(int a, int b);
};


class B : public A {
public:
    virtual void imprime(int a, double b);
};


void A::imprime(int a, int b)
{
    printf("\na = %d, b = %d\n", a, b);
}


void B::imprime(int a, double b)
{
    printf("\na = %d, b = %0.2f\n", a, b);
}
```

```
void muestra(A *a)
{
    a -> imprime(3, 6); // Siempre invoca a A::imprime(int, int)
}



void main()
{
    A *a = new A;
    B *b = new B;

    muestra(a);     // Usa A::imprime(int, int)
    muestra(b);     // Usa A::imprime(int, int)

    delete a;
    delete b;
}
```

OBSERVACIONES.

La función **muestra(A* a)** siempre invoca a la misma función ya que la función B::imprime(int a, double b) contiene diferentes argumentos a la función A::imprime(int a, int b), no generandose el proceso de sobre-escritura aunque ambas hayan sido declaradas virtuales.

Si una función en una clase derivada no es declarada como virtual, no sobre-escribe una función virtual de la clase base. Al declarar una función como virtual propicia que pueda ser sobre-escrita por funciones de subsecuentes clases derivadas, pero no se garantiza que ésto ocurra.

La salida nos resulta:

a = 3, b = 6
a = 3, b = 6

Como podemos ver, siempre es llamada la función de la clase base, ya que el compilador supone que son funciones diferentes, y no pone a funcionar el mecanismo virtual.

Al darse cuenta el compilador que se ha establecido el mecanismo para sobre-escritura de la función de la clase base, y éste no se va dar, genera una advertencia indicando tal hecho. NOTA, el programa fué compilado en Borland C++ Ver 3.1.

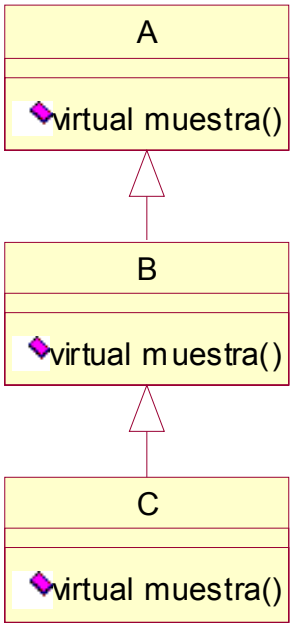
La advertencia es la siguiente:

"B::imprime(int, double) hides virtual function A::imprime(int, int)"

5.- **Funciones Virtuales Núlas.**

En una clase derivada no es necesario implementar una función la cual es declarada como virtual en una clase base. Sin embargo, si una función virtual deberá ser accesible desde una clase derivada a través del mecanismo virtual, es necesario que se establesca la trayectoria de acceso desde la clase base, hacia abajo hasta la clase derivada donde se encuentra la función virtual que se desea ejecutar. Definiendo una **función virtual nula** se puede lograr este propósito.

Análicemos el siguiente árbol de herencia.




```
// Programa PLCP89.CPP
// Funciones virtuales núlas.
```

```
#include <stdio.h>

class A {
public:
    virtual void muestra() { puts("\nClase A"); }
};


class B : public A {
public:
    virtual void muestra() {}
};


class C : public B {
public:
    virtual void muestra() { puts("\nClase C"); }
};


void muestra1(A *a)
{
    a -> muestra();
}



void main()
{
    A *a = new A;
    B *b = new B;
    C *c = new C;
    muestra1(a);        // Usa A::muestra()
    muestra1(b);        // No hace nada
    muestra1(c);        // Usa C::muestra()
}
```

OBSERVACIONES.

La salida nos resulta:

Clase A

Clase C

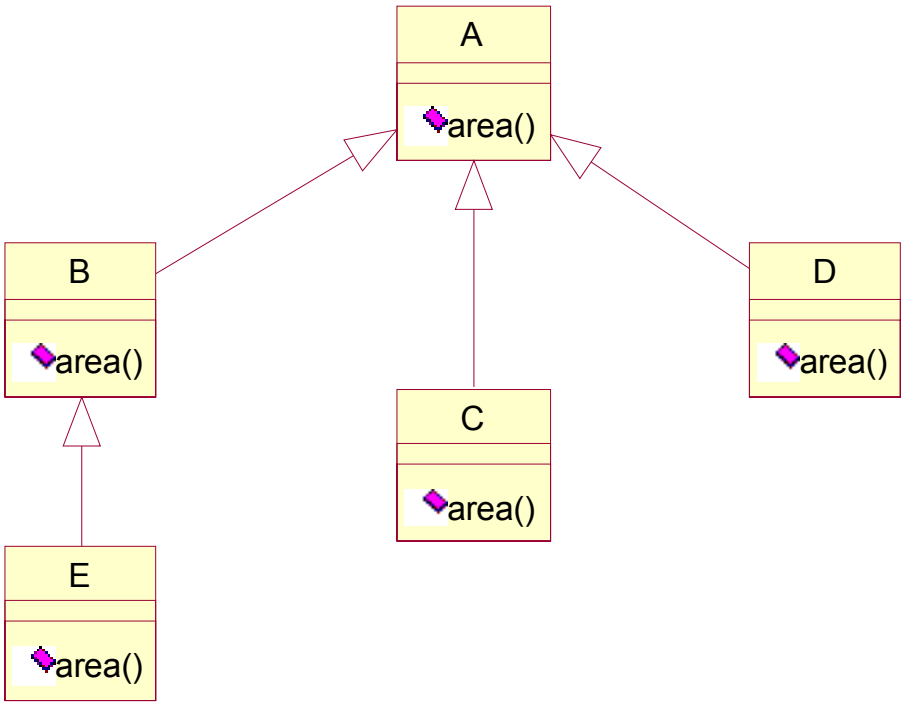
La función virtual B::muestra() se declara como función virtual nula ya que la clase B no requiere de dicha función, pero si es necesario declararla en la clase B de tal manera que sirva como interface para su manejo en la clase C desde la clase A, mediante el uso de la llamada a la función:

muestra1(c);

Entonces, con ello se tiene acceso a la función C::muestra() siguiendo la trayectoria de la función virtual, partiendo de un objeto de la clase A.

6.- **Proporcionando una interface de usuario para clases.**

Las funciones virtuales se usan con herencia de clases. Cuando se tienen varias clases, en las cuales han sido definidas funciones conceptualmente similares, es posible aplicar el mecanismo virtual. Análicemos la siguiente jerarquía de clases.



En el árbol de jerarquía anterior, no todas las clases necesitan de la función area(). Para tener acceso a la función E::area() por medio de una referencia a la clase A, entonces, tanto la clase A como la clase B deberán declarar la función area() como

virtual. Debido a que estas clases presumiblemente no tienen la necesidad de usar tal función, la función area() se debe implementar como **función virtual nula**. En la práctica los miembros de una herencia de clases, sobre todo aquellos que se encuentran en la parte superior del árbol de clases, cuentan con funciones virtuales núlas, las cuales, aunque no hacen nada, son funciones virtuales núlas, declaradas con el solo propósito de proporcionar una interface clara para las clases que se encuentran en la parte inferior del árbol de herencia.

7.- **Clases Abstractas**.

Algunas clases que se encuentran en la parte superior del árbol de clases o en los niveles superiores, normalmente tienen más de una función virtual. Estas funciones son utilizadas para definir una o más interfaces de usuario cuando se maneja la herencia de clases. Esta interface nos permite manejar el concepto de **enlace tardio**, él cual se usa de manera extensiva en este tipo de programación. Con ello, dejamos al polimorfismo el trabajo de distinguir las funciones individuales definidas en cada clase a través del mecanismo virtual.

Considere a una clase que solo tiene funciones virtuales nulas. Tendrá sentido instanciar la clase?, probablemente no, ya que la clase no hace nada, pero el compilador si nos va a permitir hacer ésto. Entonces para que nos sirve este tipo de objeto, el cual usa espacio de memoria, sin proporcionar ningún servicio.

C++ rstringe el uso de tales clases, declarandolas **abstractas**. Entonces el intento de instanciar una clase abstracta es considerada por el compilador como un error. Antes de poder declarar a una clase como **abstracta**, ésta deberá tener cuando menos una función virtual. Para declarar una clase como **abstracta**, además de contar por lo menos con una función virtual, ésta deberá se tratada como una función virtual pura, lo que requiere la siguiente notación en C++.

```
virtual void muestra() = 0;  // Función Virtual pura.
```

Lo anterior no debe ser confundido con el concepto de función virtual nula.

```
virtual void muestra() {}      // Función virtual nula.
```

La notación para declarar una función virtual pura es fácil de recordar si se considera la notación de asignación a cero para indicar que no existe una definición

para la función. Es ilegal instanciar una clase **abstracta**. Sin embargo, si esta permitido declarar un puntero a una clase **abstracta**. Este hecho abre la puerta a la manipulación indirecta de una clase abstracta, lo cual puede traer problemas, ya que al intentar invocar una función virtual pura propicia un error en tiempo de ejecución. Veamos un ejemplo de código que causa este tipo de error.

```
// Programa PLCP90.CPP
// Invocación de una función virtual pura.
// NOTA: Este programa causa problemas al intentar correrlo.

class uno {
public:
    virtual void muestra() = 0; // Función virtual pura.
};

void main()
{
    uno *X = (uno *)malloc(sizeof(uno));

    X -> muestra();
}
```

OBSERVACIONES.

Este programa compila y enlaza sin ningún problema, sin embargo, al intentar correrlo va a producir problemas debido a la definición de una función virtual pura. El error que generó fué que al correr cicló la computadora.

Si ahora usamos el operador new para solicitar memoria al heap.

```
// Programa PLCP90A.CPP
// Invocación de una función virtual pura.
// Este programa causa problemas al intentar correrlo.

class uno {
public:
    virtual void muestra() = 0; // Función virtual pura.
};

void main()
{
```

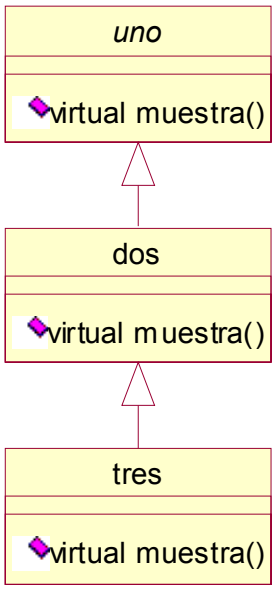
```
    uno *X = new uno;

    X -> muestra();
}
```

OBSERVACIONES.

Este código va a generar el siguiente error en tiempo de compilación:

Cannot create instancia of abstract class 'uno' in function main.

Las clases abstractas mejoran los beneficios que nos brindan las clases derivadas, lo cual sugiere que las clases abstractas deberán de encontrarse en la raiz o cerca de ésta en el árbol de jerarquía. Pueden existir varias clases abstractas en el mismo árbol de herencia. Veamos un ejemplo del uso adecuado de una clase abstracta.



(la clase abstracta, se indica su nombre en cursivas)

Veamos su implementación.

```
// Programa PLCP91.CPP
// Uso de clases abstractas en un árbol de herencia.
# include <stdio.h>
```

```
class uno {
public:
    virtual void muestra() = 0; // Función virtual pura.
};                              // Hace abstracta la clase.

class dos : public uno {
public:
    virtual void muestra() {
        puts("\nClase dos");
    }
};


class tres : public dos {
public:
    virtual void muestra() {
        puts("\nClase tres");
    }
};



void muestra1(uno *A)
{
    A -> muestra();
}


void main()
{
    dos *b  = new dos;
    tres *c = new tres;
    muestra1(b);            // usa dos::muestra();
    muestra1(c);            // usa tres::muestra();
}
```

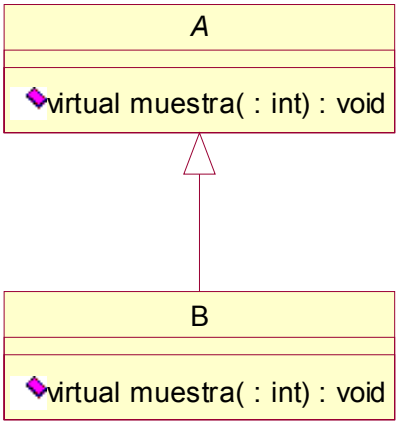
OBSERVACIONES.

La clase abstracta **uno** es usada exclusivamente como una clase base para otras clases, la función ::muestra1(uno *) determina en tiempo de ejecución la función a llamar, basandose en el tipo de objeto pasado a ella.

El hecho de derivar una clase a partir de una clase abstracta, no garantiza que la clase pueda ser instanciada. Si esta clase derivada contiene también una función virtual pura, entonces esta clase es también abstracta. C++ exige que sean sobre-escritas todas las funciones virtuales puras en las clases derivadas, si no se hace se va a generar un error en tiempo de compilación, ya que no fué posible la propagación del mecanismo virtual en las clases derivadas. Si una clase derivada de una clase abstracta, es abstracta en si misma, deberá tener por lo menos una función virtual pura. Esto implica que es posible sobre-escribir una función virtual pura con una función virtual pura.

Falla en la sobre-escritura de una función virtual pura conduce a errores de compilación. El siguiente ejemplo ilustra algunos de los problemas que deberán ser solventados, si una clase derivada no pudo sobre-escribir una función virtual pura, en la clase base.

El árbol de herencia resulta:



En el árbol se muestra una clase B derivada de una clase A, la cual contiene una función virtual pura, la cual no fué sobre-escrita. A continuación se muestra el código.

```
// Programa PLCP92.CPP
//Funciones virtuales puras.
#include <iostream.h>

class A {
public:
    virtual void muestra() = 0;
};
```

```
class B : public A {
    int valor;
public:
    virtual void muestra(int i) {
        valor = i;
    }
    void imprime() {
        cout << "Valor = " << valor << endl;
    }
};


void main()
{
    int y = 10;
    B b;

    b.muestra(y);
    b.imprime();
}
```

OBSERVACIONES.

En el ejemplo anterior, podemos pensar que el programador paso por alto el parámetro de la función B::muestrea(int x), el compilador si detectó esta situación y envió el siguiente mensaje de error:

Cannot create instancia of abstrac class 'B' in function main().

Ya que la clase B deberá heredar la función A::muestra(), el error se produce ya que B no sobre-escribe la función A::muestra(). Si se declarara otra clase llamada C después de la clase B, y en ella si se define correctamente la función C::muestra(), no podrá ejecutarse el mecanismo virtual, ya que en B se interrumpió éste.

NOTA: Las funciones virtuales puras deberán ser sobre-escritas en las clases derivadas para que no se produsca este tipo de error.

8.- **Limitaciones de las funciones virtuales.**

1) No es posible usar funciones virtuales como funciones típicas de ANSI C ya que éstas solo se pueden manejar dentro de las clases. Las funciones globales no

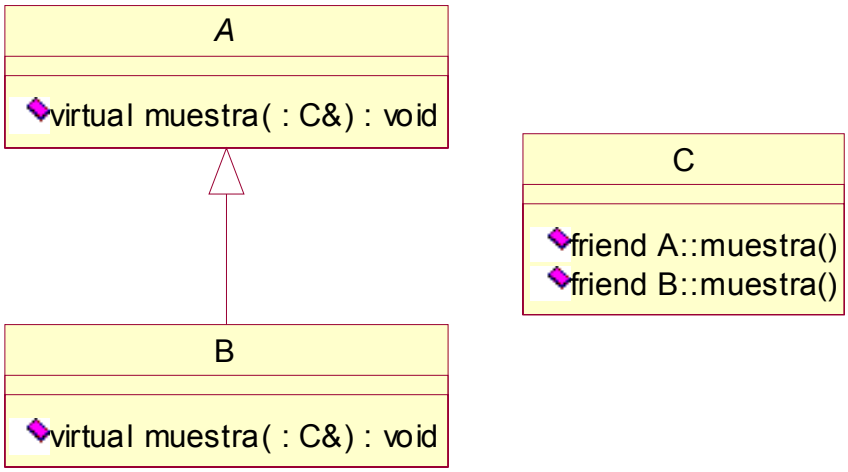
pueden ser declaradas virtuales, solo se puede declarar virtual a las funciones miembro de una clase.

2) Las funciones virtuales, nunca pueden ser declaradas **static** ya que las funciones declaradas **static** se pueden invocadar sin la necesidad de hacer referencia a un objeto en particular. El mecanismo de invocación a una función virtual usa un puntero (vptr) para localizar la función virtual que esta siendo llamada por un determinado objeto, en tiempo de ejecución. Como las funciones **static** no cuentan con un puntero **this** no pueden manejar una trayectoria para su invocación como una función virtual.

9.- **Funciones virtuales friend.**

La idea de declarar una función virtual como **friend**, suena medio raro, sin embargo, ésto es posible si la función declarada como **friend** es una función miembro de una clase determinada.

En el siguiente ejemplo se ilustra este tipo de funciones.



A continuación se presenta el código.

```
// Programa PLCP93.CPP
// Funciones virtuales friend
#include <stdio.h>

class C;

class A {
```

```
    int a;
public:
    A(int i) { a = i; }
    virtual void muestra(C&);
    int dice() { return a; }
};


class B : public A {
    int b;
public:
    B(int i, int j) : A(i) {
        b = j;
    }
    virtual void muestra(C&);
};


class C {
    int a, b, c;
public:
    C(int i, int j, int k) {
        a = i;
        b = j;
        c = k;
    }
    friend void A::muestra(C&);
    friend void B::muestra(C&);
};

void A::muestra(C &t)
{
    printf("\nClase A miembro a = %d", a);
    printf("\nClase C miembro a = %d", t.a);
    printf("\nClase C miembro b = %d", t.b);
    printf("\nClase C miembro c = %d\n", t.c);
}

void B::muestra(C &t)
{
    printf("\nClase A miembro a = %d", dice());
    printf("\nClase B miembro b = %d", b);
    printf("\nClase C miembro a = %d", t.a);
    printf("\nClase C miembro b = %d", t.b);
    printf("\nClase C miembro c = %d\n", t.c);
}
```

void main()

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

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

OBSERVACIONES.

La salida nos resulta:

```
Clase A miembro a = 10
Clase C miembro a = 40
Clase C miembro b = 50
Clase C miembro c = 60

Clase A miembro a = 20
Clase B miembro b = 30
Clase C miembro a = 40
Clase C miembro b = 50
Clase C miembro c = 60
```

En la salida podemos apreciar que cada objeto nos muestra los valores de inicialización, en ambos bloques, tenemos disponibles los valores del objeto **c** perteneciente a la clase **C**.

Es necesario notar que **friend** es un atributo resuelto en tiempo de compilación, mientras el mecanismo virtual es una secuencia llevada a cabo en tiempo de ejecución. Por lo tanto, la misma función puede ser declarada tanto **friend** como **virtual**, cada atributo es resuelto en diferente tiempo, y el atributo **friend** no causa sobrecarga en tiempo de ejecución.