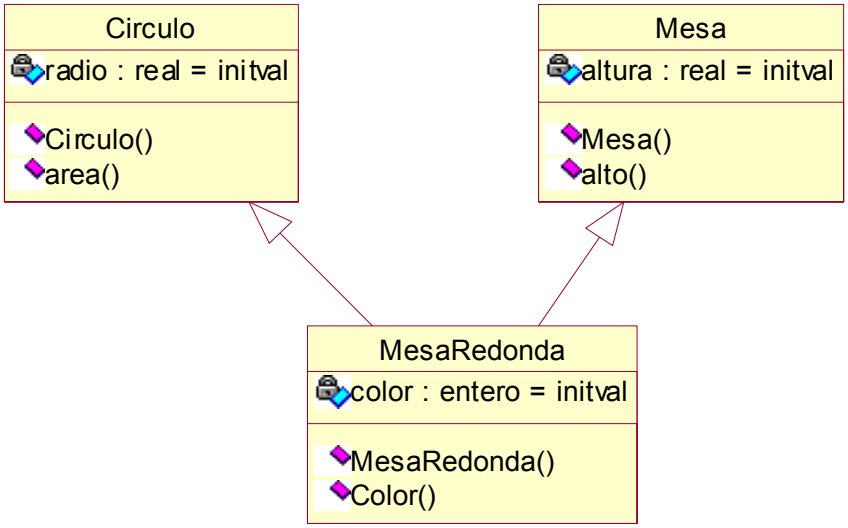
5.- **Herencia Múltiple.**

Un hecho natural es que una persona tenga más de un pariente mayor, esta situación también se puede dar en la herencia de clases, naturalmente este tipo de herencia involucra un mayor grado de complejidad en el lenguaje, sin embargo, el beneficio logrado es substancial. Considere por ejemplo, la clase llamada **MesaRedonda** la cual tiene las propiedades de una **Mesa** y también tiene las propiedades de un **Circulo**, de acuedo a la siguiente estructura.



Veamos ahora la estructura de clases en C++.

```
// Programa PLCP84.CPP
// Herencia Múltiple.
#include <math.h>
#include <stdio.h>

class Circulo {
    float radio;
public:
    Circulo(float r) { radio = r; }
    float area() { return  M_PI*radio*radio; }
};


class Mesa {
    float altura;
public:
    Mesa(float a) { altura = a; }
    float alto() { return altura; }
};
```

```
class MesaRedonda : public Mesa, public Circulo {
    int color;
public:
    MesaRedonda(float r, float a, int c);
    int Color() { return color; }
};

MesaRedonda::MesaRedonda(float r, float a, int c) :
Circulo(r), Mesa(a)
{
    color = c;
}


void main()
{
    MesaRedonda mesa(15.0, 3.0, 5);

    printf("\nLas propiedades de la mesa son:");
    printf("\nAltura = %0.2f", mesa.alto());
    printf("\nArea   = %0.2f", mesa.area());
    printf("\nColor  = %d", mesa.Color());
}
```

OBSERVACIONES.

La salida nos resulta:

Las propiedades de la mesa son:
```
Altura = 3.00
Area   = 706.86
Color  = 5
```

En este caso podemos observar que la clase **MesaRedonda** hereda las propiedades tanto de la clase **Circulo** como de la clase **Mesa**, ésto nos reduce bastante la declaración de la clase **MesaRedonda**. Podemos observar en la declaración de la clase **MesaRedonda** que no solo hay que declarar de que clases hereda características la clase derivada, también es necesario declarar las especificaciones de acceso de cada clase base. Estas especificaciones se usan de la misma manera que en la herencia simple, y no tienen que ser las mismas para todas las clases base.

En el ejemplo anterior, en la función main(), son invocadas tres funciones miembro desde l clase **MesaRedonda**, estas son; MesaRedonda::Alto(), MesaRedonda::area() y MesaRedonda::Color(), note que en ningún caso se indica de que clase son heredadas dichas funciones, y cuales no lo son, todas se tratan como si

fueran funciones miembro de la clase **MesaRedonda**. Esta forma de referirse a miembros heredados de superclases es una extensión natural de la herencia simple.

5.1) Declarando una clase derivada con multiples clases base

La declaración de la clase **MesaRedonda** nos muestra la sintaxis usada; la lista de clases base se indica después del nombre de la clase derivada y de 2 puntos (:), con su especificador de acceso y separada cada clase por una coma (,).

Estas clases base son llamadas **clases base directas**, para distinguirlas de las clases base indirectas, las cuales son clases base de una clase base directa (clases abuelas, diriamos). La clase **Mesa** y **Circulo** son clases base directas de la clase derivada **MesaRedonda**, pero si las clases **Mesa** y **Circulo** tuvieran alguna clase base, éstas se llamarían clases base indirectas de la clase **MesaRedonda**. Esta distinción entre clases base directa e indirecta es importante, ya que el compilador aplica diferentes reglas para cada tipo.

Una clase puede tener tantas clases base como se quiera, pero ésto nos incrementa la complejidad. Como un punto de referencia podemos tomar la implementación de las librerias de C++, en donde, cuando se usa herencia multiple se usan a lo más dos clases base.

5.2) Invocando a los constructores de la clase base.

De la misma forma que con herencia simple, los constructores en multiples clases base son invocados antes del constructor de la clase derivada. El orden de declaración especifica el orden de invocación. Por ejemplo, consideremos el caso para la clase **MesaRedonda**.

```
class MesaRedonda : public Mesa, public Circulo {
    int color;
public:
    MesaRedonda(float r, float a, int c);
    int Color() { return color; }
};
```

Cuando se hace la instanciación de esta clase, el orden de llamada de los constructores de las clases base es el siguiente:

Mesa::Mesa(float r);

Circulo::Circulo(float a);

MesaRedonda::MesaRedonda(int c);

En la definición del constructor de la clase **MesaRedonda**, se pasaron 2 argumentos a las clases base (r, a).

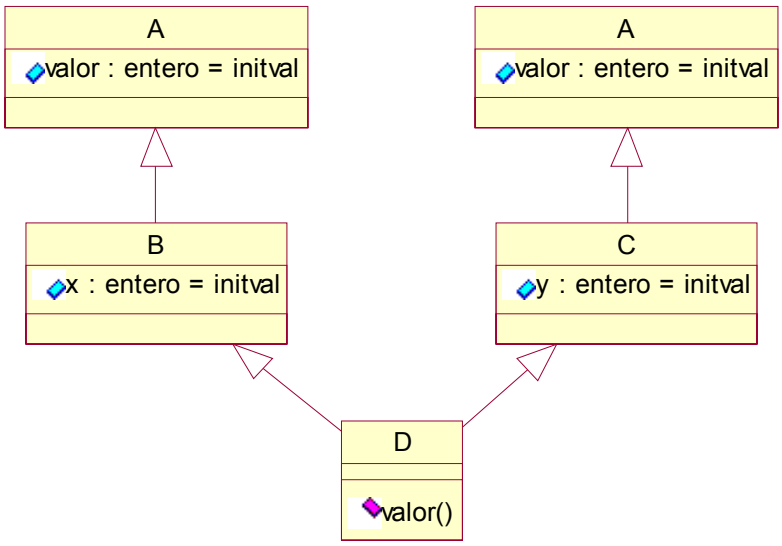
```
MesaRedonda::MesaRedonda(float r, float a, int c) :
Circulo(r), Mesa(a)
{
     color = c;
}
```

El orden de llamada a estos constructores puede causar confusión, ya que el orden de llamada es definido en la declaración de la clase, NO en la definición del constructor, donde el orden de especificación de los constructores no importa, el orden especificado de llamada a los constructores de las clases base es aparente.

En el caso de definir constructores **virtuales**, estos son llamados antes de cualquier constructor no **virtual**.

5.3) Usando Clases Base Virtuales.

Las clases Base Virtuales solo se usan en el contexto de herencia múltiplo. Dada la complejidad que puede aparecer en las relaciones de clase, en la definición del árbol de herencia múltilpe, hay situaciones en las cuales el programador requiere contar con un cierto nivel de control en la forma en que las clases base heredan sus atributos. Considere, por ejemplo, el siguiente árbol de herencia.



La clase **D** tiene a la clase **A** como una clase base, el problema es que existen 2 clases **A** diferentes que aparecen como clases base de la clase **D**, donde cada una de las cuales tiene sus propios datos, aunque en realidad es solo una clase **A**, pero al considerarla como clase base de 2 clases derivadas diferentes, el compilador maneja 2 copias de la clase **A**. En la vida real, esta situación será equivalente a que una persona tenga 2 abuelos, los cuales no solo son genelos, sino también tienen el mismo nombre. ¿Cómo lidear con esta situación?. Consideremos el siguiente código:

```
// Programa PLCP85.CPP
// Herencia múltiple, clase repetidas
#include <iostream.h>

class A {
public:
    int valor;
};

class B : public A {
public:
    int x;
};

class C : public A {
public:
    int y;
};

class D : public B, public C {
public:
    int Valor() { return valor; }
};


void main()
{
    D d;
    d.C::valor = 5;
    D *dp = new D;
    int w = dp -> B::valor = 10;
    cout << "Valor de v = " << d.Valor() << endl;
    cout << "Valor de w = " << w << endl;
    delete dp;
}
```

OBSERVACIONES.

En este caso la sentencia de acceso al dato miembro en la clase **D** es ambigua. Borland C++ nos genera el siguiente mensaje de error.

"Member is ambiguous in 'A::valor' and 'A::valor'
in function D::Valor()"

El compilador no sabe a que copia de **valor** se esta haciendo referencia, entonces, deberemos aplicar el operador de resolución de ámbito en la función de la clase **D::Valor()**.

```
int Valor() { C::valor; }
```

Una función fuera de la clase **D** podrá también accesar cada uno de los datos miembro usando el operador de resolución de ámbito combinado con un identificador del objeto.

NOTA: En el programa anterior aparece también una advertencia, por el aparente no uso del objeto **d** en la función main().

El código a continuación nos muestra una solución al problema, de acuerdo como se planteó anteriormente.

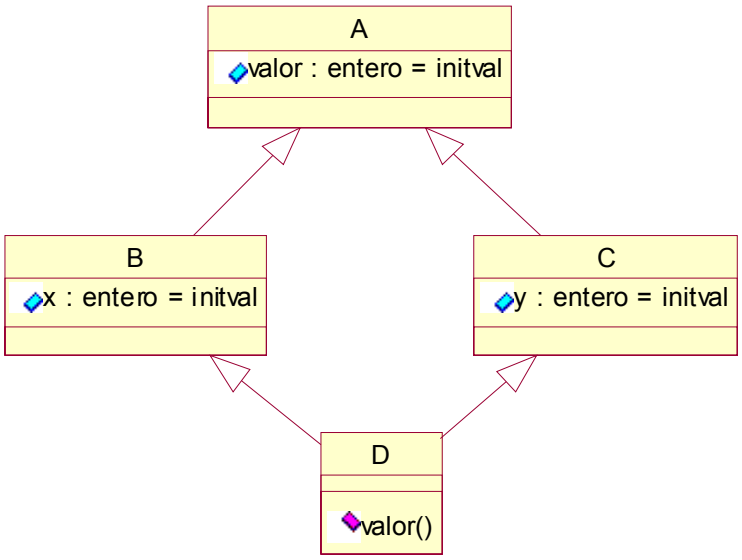
```
// Programa PLCP85.CPP
// Herencia múltiple, clase repetidas
#include <iostream.h>

class A {
public:
    int valor;
};

class B : public A {
public:
    int x;
};

class C : public A {
public:
    int y;
};

class D : public B, public C {
public:
    int Valor() { return C::valor; }
};
```

```
void main()
{
    D d;
    d.C::valor = 5;
    D *dp = new D;
    int w = dp -> B::valor = 10;
    cout << "Valor de v = " << d.Valor() << endl;
    cout << "Valor de w = " << w << endl;
    delete dp;
}
```

OBSERVACIONES.

NOTA: En este programa ya no se genera ningún error ni advertencia.

En la clase **D** se hace una clara distinción de la rama de donde viene la clase base **A**, ésto es, se hace la distinción que viene por el lado de la superclase **C**. También es necesario indicarlo cuando se inicializa el dato miembro **valor**. Si se usa la clase base **B** o se usa la clase base **C**. Si se usa la clase base **C**, y a esta copia de **valor** se le asigna un 5, como se definió en la función **int D::Valor()** que se esta derivando de la clase **C**, es posible usar esta función para mostrar el dato asignado. Esta estructura es más o menos compleja, se puede simplificar un poco redefiniendo la estructura de clase como sigue.



El tener múltiples copias de la misma clase base, en un árbol de herencia no es solo confuso sino también involucra un derroche de memoria.

Declarando la clase base **A** como **virtual** resuelve este problema, forzando al compilador a manejar solo una copia de la clase base en las declaraciones de las clases derivadas. Esto nos lleva también a la redefinición de la función de **Valor()** de la clase **D**.

El código para esta estructura lo escribimos de la forma:

```
// Programa PLCP85A.CPP
// Herencia múltiple, clase repetidas
#include <iostream.h>

class A {
public:
    int valor;
};

class B : public virtual A {
public:
    int x;
};

class C : public virtual A {
public:
    int y;
};


class D : public B, public C {
public:
    int Valor() { return valor; }
};


void main()
{
    D d;
    d.valor = 5;
    D *dp = new D;
    int w = dp -> valor = 10;
    cout << "Valor de v = " << d.Valor() << endl;
    cout << "Valor de w = " << w << endl;
    delete dp;
}
```

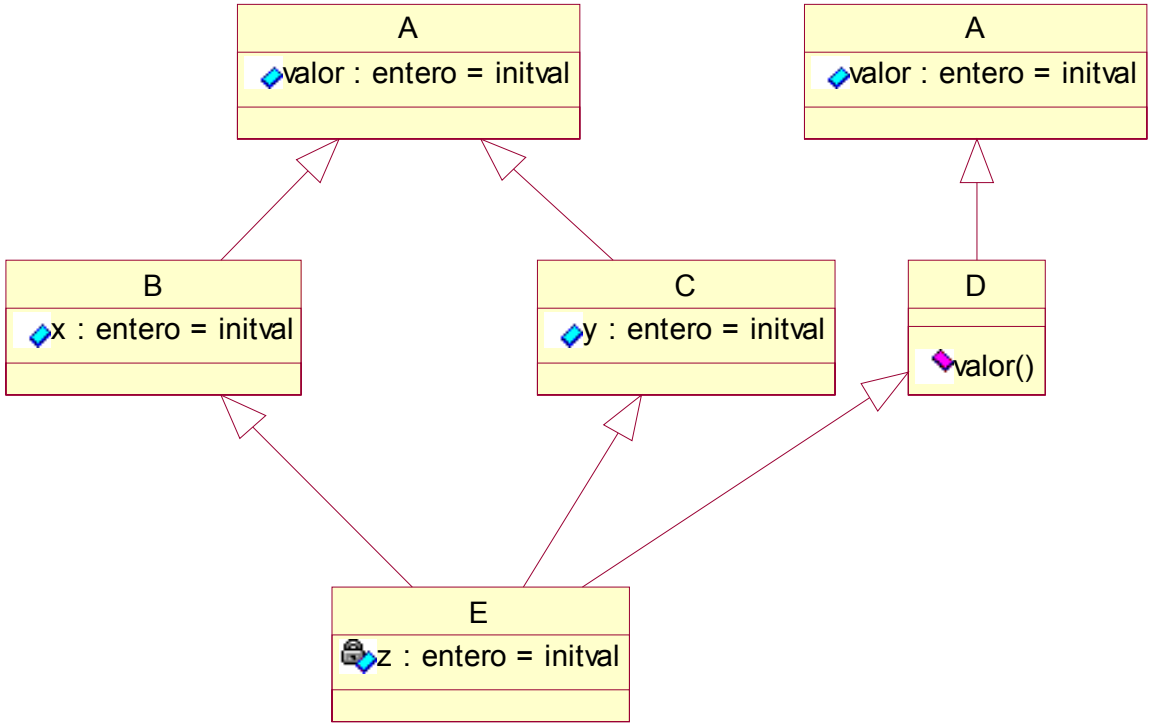
OBSERVACIONES.

Al haber definido como virtual la clase base **A**, tanto en la declaración de la clase **B** como en la declaración de la clase **C**, se establece la estructura de clases arriba indicada, y la estructura del programa vuelve a la normalidad, note que en este caso por simplicidad se manejaron los datos miembro como públicos.

5.4) Usando clases virtuales y no virtuales juntas.

Una clase puede tener clases base virtuales y no virtuales en cualquier orden y combinación.

Si tenemos el siguiente árbol de clases.



El código para dicho árbol de clases resulta de la siguiente forma:

```
class A {};
class B : public virtual A {};
```

```
class C : public virtual A {};
class D : public A {};
class E : public B, public C, public D {};
```

La clase **E** maneja 2 diferentes copias de la clase **A**. Este hecho no genera un error en C++, pero puede causar confusión en los usuarios. De acuerdo a la declaración de la clase **E**, el orden de llamada de los constructores es el siguiente: A, B, C, D, E. Primero se construye la clase base **virtual**, en seguida, las demás clases base. El orden en que las clases virtuales son construidas corresponde al orden de aparición en la declaración de la clase. Después que todas las clases base virtuales son construidas, se llaman a los constructores de las clases no virtuales, en el orden en que aparecen en la declaración de la clase.

5.5) Invocando a los destructores.

Un destructor solo puede invocarse de manera indirecta, por medio del operador **delete**. La secuencia de llamada de los destructores, es el inverso en que fueron llamados los constructores.

5.6) Usando conversión de tipos.

Cuando se usa herencia simple, se puede convertir, por ejemplo, un puntero de una clase derivada en un puntero de una clase base, recuerde que lo contrario no es posible.

Con herencia múltiple, las cosas son más dificiles, la conversión de punteros y referencias no siempre es posible, ya que se puede presentar ambigüedad.

Consideremos nuestro primer ejemplo de herencia múltiple.

```
// Programa PLCP85B.CPP
// Herencia múltiple, clase repetidas, conversión de tipos.
#include <iostream.h>

class A {
public:
    int valor;
};

class B : public A {
public:
    int x;
};

class C : public A {
```

```
public:
    int y;
};

class D : public B, public C {
public:
    int Valor() { return C::valor; }
};

void main()
{
/* ---------------------------------------  GENERA ERROR
    A *pa1 = new D;                          No se puede hacer
    int w = pa1 -> valor = 10;               esta conversión.
   ---------------------------------------- */
    A *pa2 = new B;
    int x = pa2 -> valor = 20;
    A *pa3 = new C;
    int y = pa3 -> valor = 30;

//  cout << "Valor de w = " << w << endl;    GENERA ERROR
    cout << "Valor de x = " << x << endl;
    cout << "Valor de y = " << y << endl;
//  delete pa1;                              GENERA EEROR
    delete pa2;
    delete pa3;
}
```

OBSERVACIONES.

Cuando el compilador ve la sentencia:

A *pa1 = new D;

El compilador se da cuenta que la clase **D** tiene 2 diferentes versiones de la clase **A**, declarando la clase **A** como virtual, se puede prevenir este problema, ya que entonces, solo una copia de la clase **A** es incluida en la clase **D**.

Veamos como resulta el código.

```
// Programa PLCP85C.CPP
// Herencia múltiple, clase repetidas, conversión de tipos.
// uso de clases base virtuales.
#include <iostream.h>
```

```
class A {
public:
     int valor;
};

class B : public virtual A {
public:
     int x;
};

class C : public virtual A {
public:
     int y;
};

class D : public B, public C {
public:
     int Valor() { return valor; }
};


void main()
{
     A *pa1 = new D;
     pa1 -> valor = 10;
     A *pa2 = new B;
     pa2 -> valor = 20;
     A *pa3 = new C;
     pa3 -> valor = 30;

     cout << "Valor de pa1 -> valor = " << pa1 -> valor << endl;
     cout << "Valor de pa2 -> valor = " << pa2 -> valor << endl;
     cout << "Valor de pa3 -> valor = " << pa3 -> valor << endl;

     delete pa1;
     delete pa2;
     delete pa3;
}
```

OBSERVACIONES.

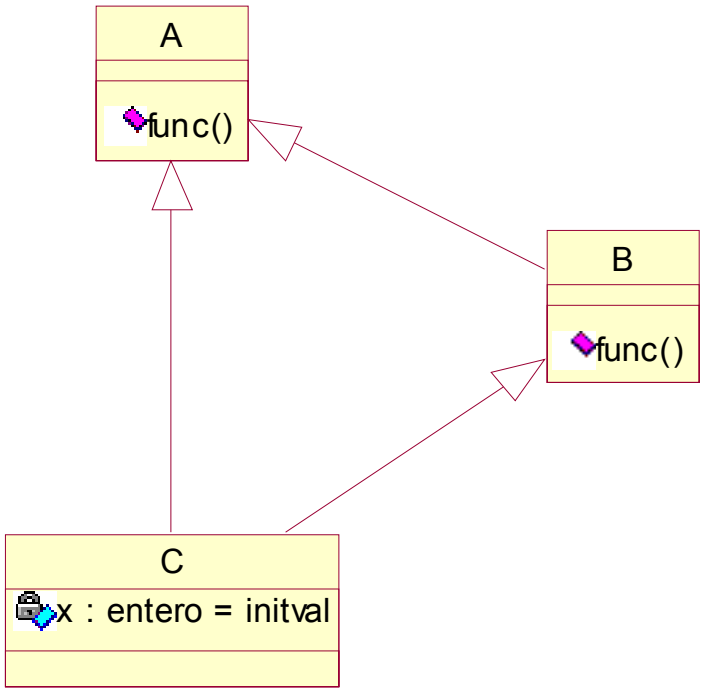
Ahora ya no hay ningún problema ya que solo existe una copia de la clase base **A**.

La salida nos resulta:

```
Valor de pa1 -> valor = 10
Valor de pa2 -> valor = 20
Valor de pa3 -> valor = 30
```

5.7) Búsqueda de una clase en el árbol de clases.

Cuando es invocada una función dentro de una clase derivada, el compilador busca en el árbol de clases, aquellas a las que se tiene acceso, siguiendo ciertas reglas. Si la función no es encontrada en la clase desde donde es invocada, el compilador la busca hacia arriba a través del árbol de herencia. Esto sucede en tiempo de compilación, no en tiempo de ejecución. Si varias clases tienen una función con el mismo nombre, C++, de acuerdo a ciertas reglas determina cual función esta siendo llamada. Considere el siguiente árbol de herencia.



A continuación se presenta la implementación del árbol de herencia anterior..

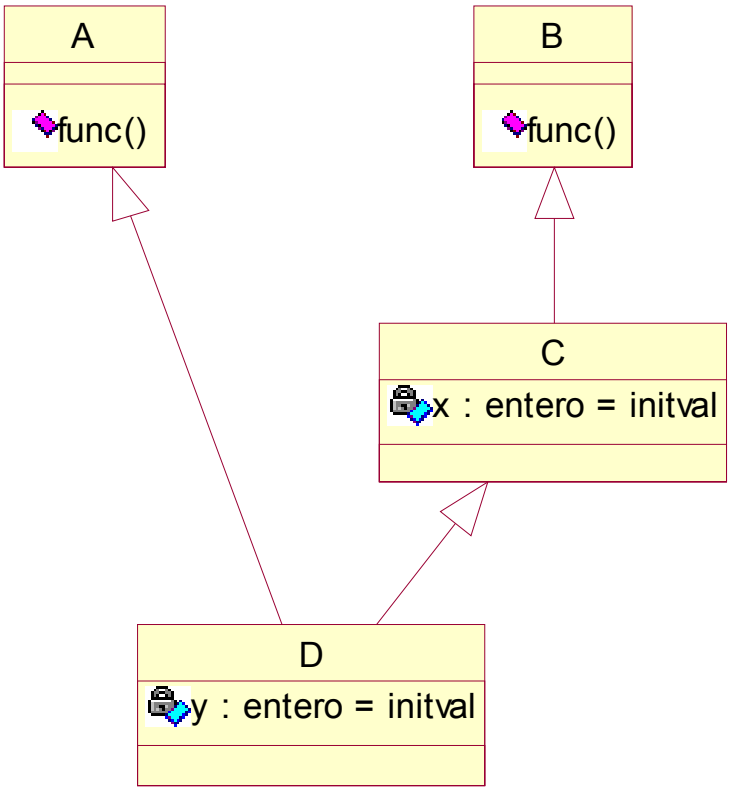
```
// Programa PLCP86.CPP
// Trayectoria de busqueda de funciones
// en el árbol de clases.
#include <iostream.h>
```

```
class A {
public:
    int func() { return 1; }
};


class B : virtual public A {
public:
    int func() { return 2; }
};


class C : public virtual A, public B {
    int x;
};

void main()
{
    C c;
    int i = c.func();                   // Llama a B::func();
    cout << "i = " << i << endl;

    int j = c.A::func();                // Llama a A::func();
    cout << "j = " << j << endl;
}
```

OBSERVACIONES.

En este caso el compilador sigue la regla de **dominación**. El compilador se da cuenta que en ambas clases **B** y **A**, se encuentra una función llamada **func()**. La regla de **dominación** le dice que si 2 clases contienen la función buscada, y si una clase es derivada de la otra, la clase derivada **domina**.

En el ejemplo anterior la clase **B** satisface este criterio, si no existe una función dominante, el compilador genera un error de ambigüedad. En el siguiente árbol se presenta este error de ambigüedad, ya que no hay una clase **dominante**.



Su codificación resulta:

```
// Programa PLCP86A.CPP
// Ambigüedad en la regla de dominación
#include <iostream.h>

class A {
public:
     func() { return 1; }
};

class B {
public:
     func() { return 2; }
};


class C : public B {
     int x;
};
```

```
class D : public C, public A {
     int y;
};


void main()
{
     D d;
     int i = d.func();

     cout << "i = " << i << endl;
}
```

OBSERVACIONES.

Al compilar el programa anterior el compilador nos va a generar el siguiente error.

"Member is ambiguous; 'B::func' and 'A::func' en function main()"

5.8) Usando el operador de resolución de ámbito con herencia múltiple.

El error en el programa anterior se puede corregir re-escribiendo main() para incluir el operador de resolución de ámbito.

```
void main()
{
     D d;
     int i = d.A::func();
     int j = d.B::func();

     cout << "i = " << i << endl;
     cout << "j = " << j << endl;
}
```

La corrección de main(), se hace inclusive para la llamada a las 2 funciones miembro **func()**.

Esto es, usando el operador de resolución de ámbito para especificar de manera explicita al compilador que función deseamos llamar, ya que en este caso, no puede aplicar la regla de dominación.

La declaración de varias clases base con el mismo identificador público, no ocasiona un error. Esto lo permite el compilador al declarar los identificadores, sin embargo si se puede producir un problema al accesarlos.

El acceso a los identificadores se determina solo cuando el código ejecutable necesita ser generado, entonces solo se puede usar un identificador a la vez. Por ejemplo, suponga que un identificador es declarado y éste puede ser ambiguo en el árbol de herencia, en estos casos, el error es detectado solamente si el programador escribe código que intente accesar el identificador ambiguo, sin el uso del operador de resolución de ámbito apropiado.