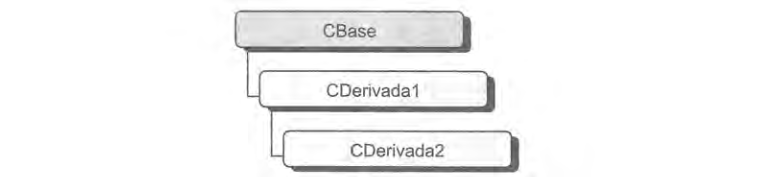
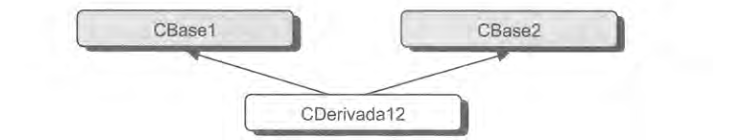
Herencia múltiple

Una clase derivada puede tener una o más clases base directas. Cuando una clase tiene solo una clase, la relación de herencia se denomina como *herencia simple*. En tal caso, el objeto de la clase derivada se compone de los miembros heredados de la clase base y de los propios de la clase derivada.



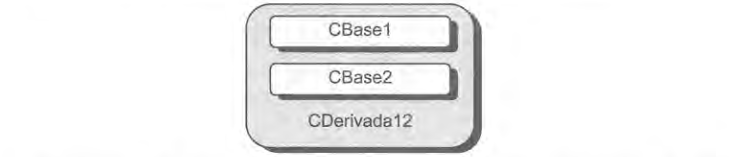
Por el contrario, cuando una clase tiene más de una clase base directa, según muestra el gráfico siguiente, se tiene el caso de *herencia múltiple*:



Esta jerarquía de clases vista desde C++ se implementaría así:

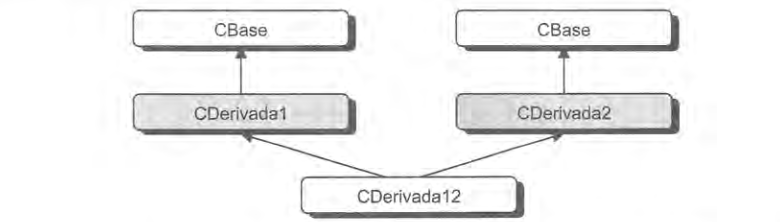


En este caso, un objeto de CDerivada12 estará formado por los miembros heredados de CBase1, más los miembros heredados de CBase2, más sus propios miembros. La herencia múltiple permite que clases hermanas (CBase1 y CBase2 se denominan clases hermanas respecto de CDerivada12) compartan información sin que exista una dependencia con respecto a una clase base común.



El acceso a un miembro de CBase1, de CBase2 o de CDerivada12 se hace exactamente igual que en la herencia simple. En cuanto a los métodos virtuales, trabajan en la forma acostumbrada, no obstante, al final de este apartado haremos alguna observación para tener en cuenta.

Por otra parte, el hecho de que podamos especificar más de una clase base implica la posibilidad de tener otra clase dos veces (nos referimos a la clase base de éstas que actúan ahora como base en la derivación múltiple); en este caso se dice que la clase base está replicada. Por ejemplo:



En este ejemplo, tanto la clase CDerivada1 como CDerivada2 se han derivado de la misma clase, CBase, por lo que Cderivada12 heredará tanto los miembros de CDerivada1 como de CDerivada2, lo que implica heredar dos veces los miembros de CBase.

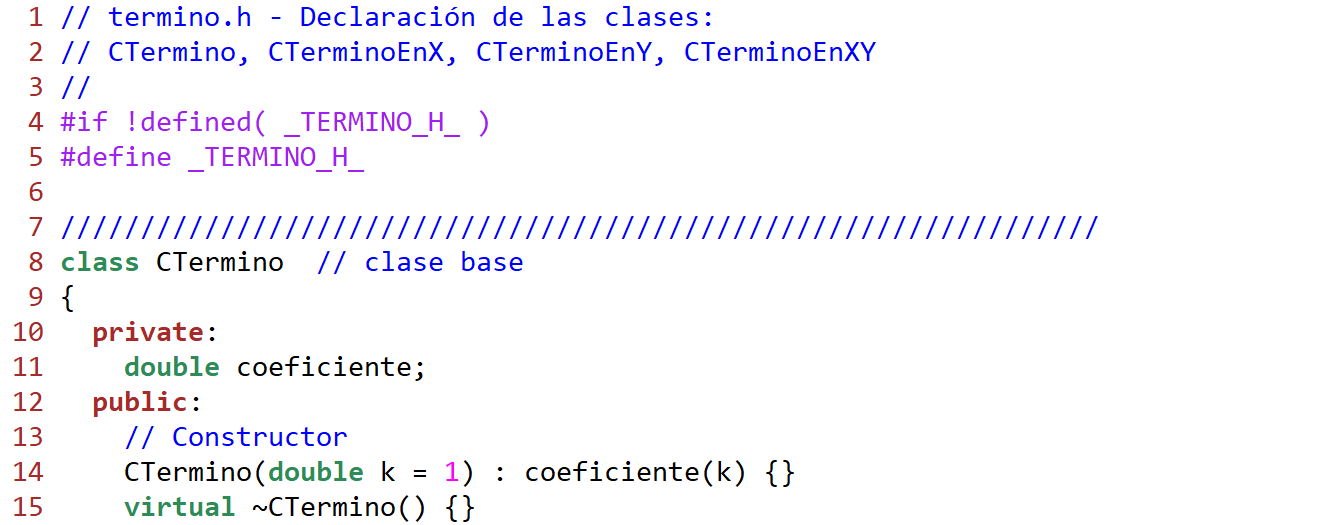
Un ejemplo práctico podría ser una jerarquía de clases que permita crear términos de un polinomio dependiente de dos variables x e y. Por ejemplo:

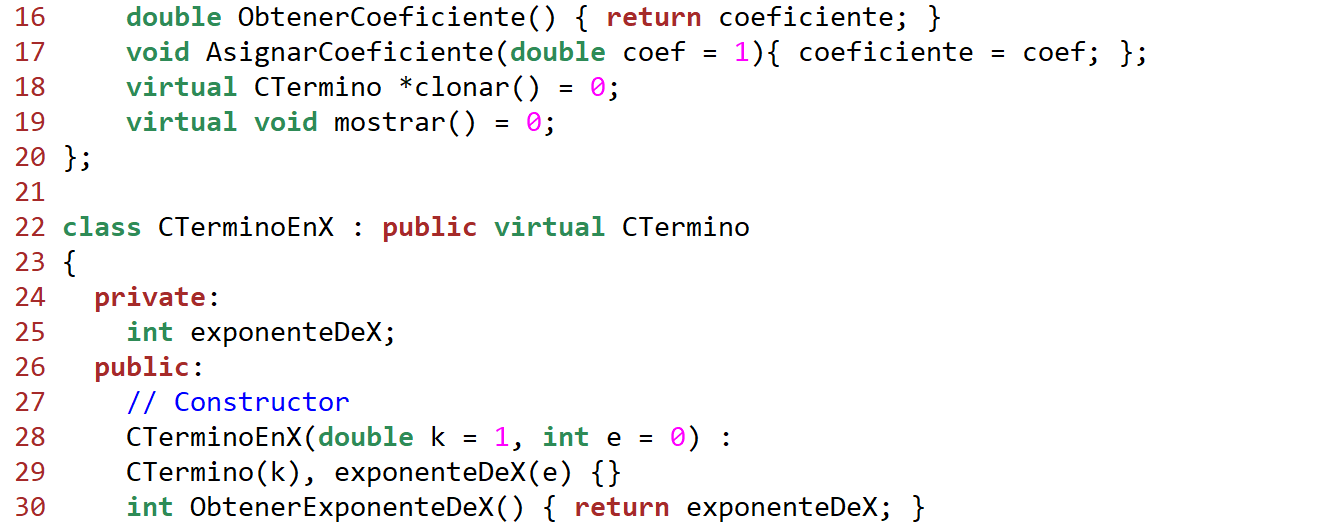


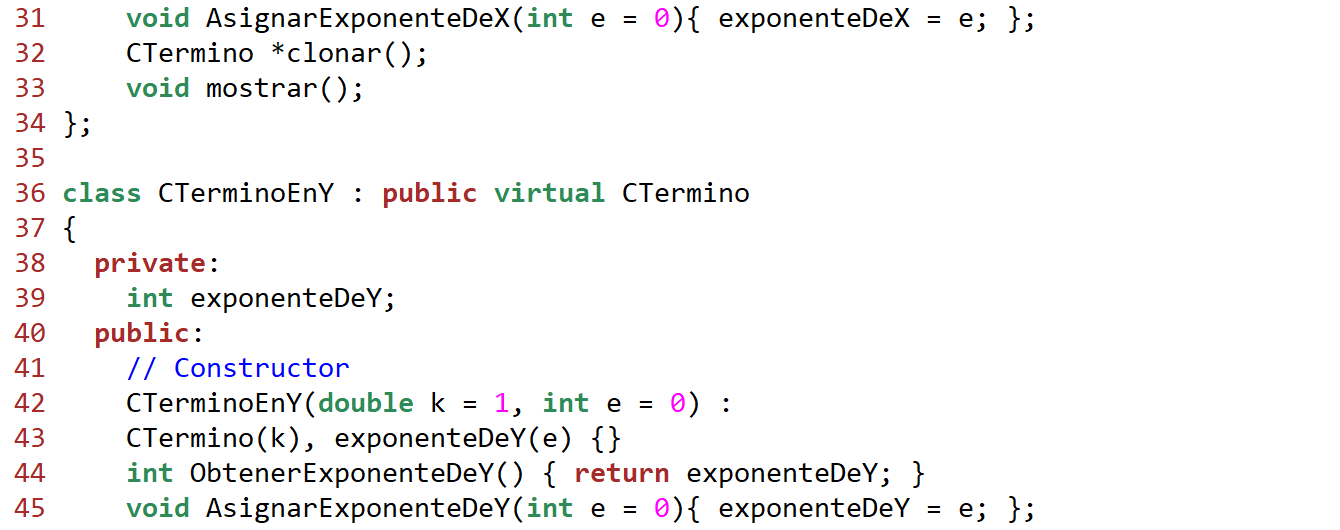
La clase base de esta jerarquía podría ser CTermino con un dato miembro que almacene el coeficiente de un término cualquiera. De ella se derivarían CTerminoEnX, con un dato miembro que almacene el exponente de x, y CTerminoEnY, con un dato miembro que almacene el exponente de y, y de éstas dos se derivaría CTerminoEnXY. De esta forma, un objeto de la clase CTermino puede representar un término independiente, un objeto de la clase CTerminoEnX puede representar a un término en x, un objeto de la clase CTerminoEnY puede representar a un término en y, y un objeto de la clase CTerminoEnXY puede representar a un término en xy.

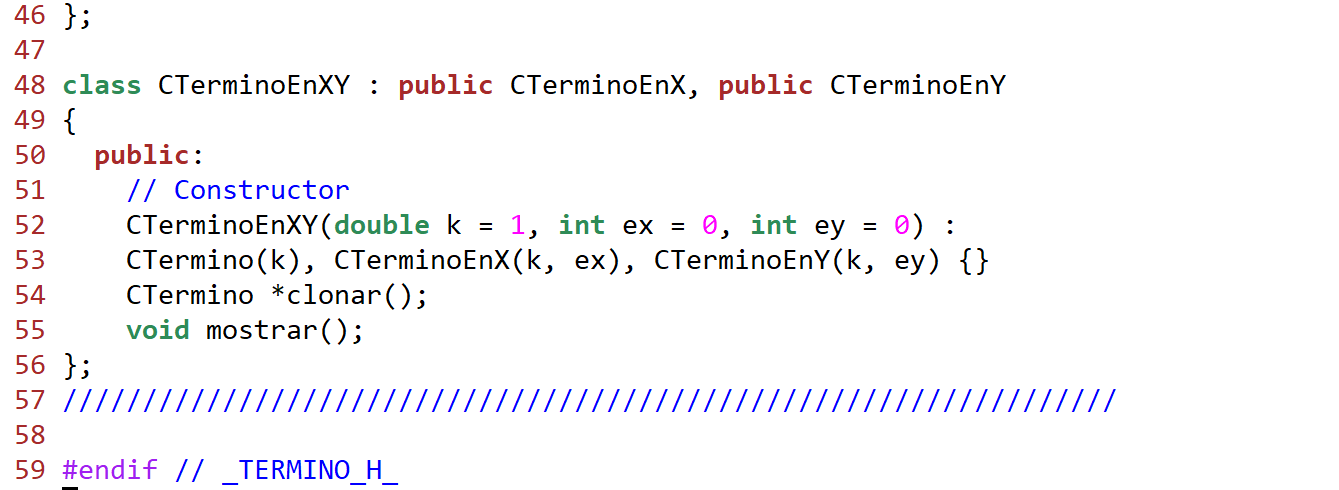
Si ahora suponemos que inicialmente ya existían las clases CTerminoEnX y CTerminoEnY, cabe destacar el papel tan importante que hace la herencia múltiple en la fusión de clases existentes. En definitiva, ésta es la aplicación más común de la herencia múltiple.

Apoyándonos en este ejemplo práctico, vamos a implementar una jerarquía de clases igual a la mostrada en la figura anterior:



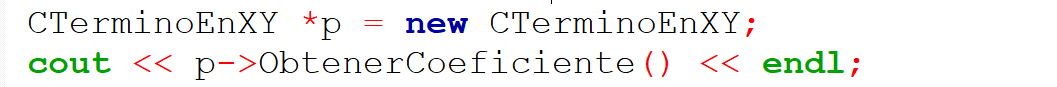




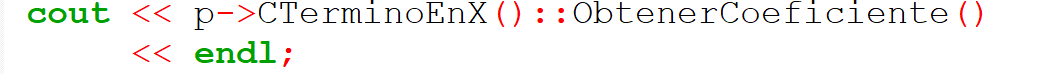


Archivo termino.h

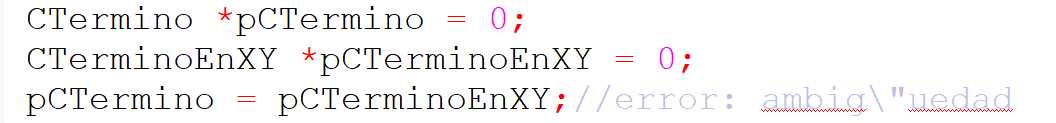
El hecho de que cada objeto de CTerminoEnXY contenga dos copias de los miembros de CTermino implica que las referencias a los mismos producirán errores de ambigüedad. Por ejemplo, la siguiente sentencia produce un error de ambigüedad (el error de ambigüedad se produce si las clases CTerminoEnX y CTerminoEnY se derivan solo como public de la clase base CTermino, y no como public virtual de la clase base CTermino):



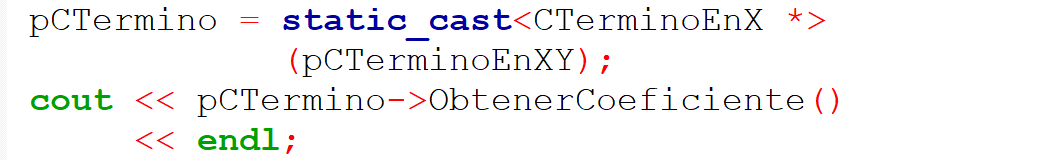
Lo que sucede es que el compilador no puede decidir qué copia de coeficiente tiene que utilizar: la copia heredada de CTermino a través de CTerminoEnX o la heredada a través de CTerminoEnY. Especificar explícitamente la copia a utilizar, por ejemplo:



Tampoco soluciona el problema. Otros problemas de ambigüedad pueden surgir por otras causas y generalmente la solución será utilizar el operador de resolución de ámbito, o simplemente ejecutar una conversión cast. Por ejemplo, suponga que deseamos convertir un apuntador a un objeto de la clase CTerminoEnXY en un apuntador a la clase CTermino. Si procedemos como se indica a continuación, se producirá un error:



Una vez más, el compilador no puede decidir si la conversión la realiza a través de CTerminoEnX o a través de CTerminoEnY. Para deshacer intentar resolver la ambigüedad, se prueba utilizar una conversión cast, por ejemplo, así:

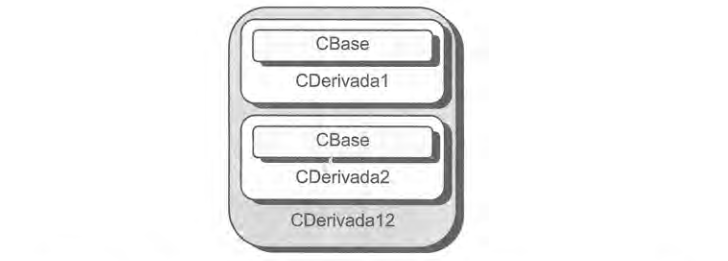


Pero el compilador mingw-g++ que utiliza Codeblocks sigue marcando error.

Lo expuesto hasta ahora demuestra que una clase base que se replique igual que CTermino no se debería utilizar fuera de sus clases derivadas inmediatas (por ejemplo, CTermino no debería utilizarse fuera de CTerminoEnX o de CTerminoEnY), y cuando se haga desde cualquier punto donde se vea más de una copia de la base, los accesos a la funcionalidad de la misma deben calificarse explícitamente para resolver la ambigüedad.

# Clases base virtuales

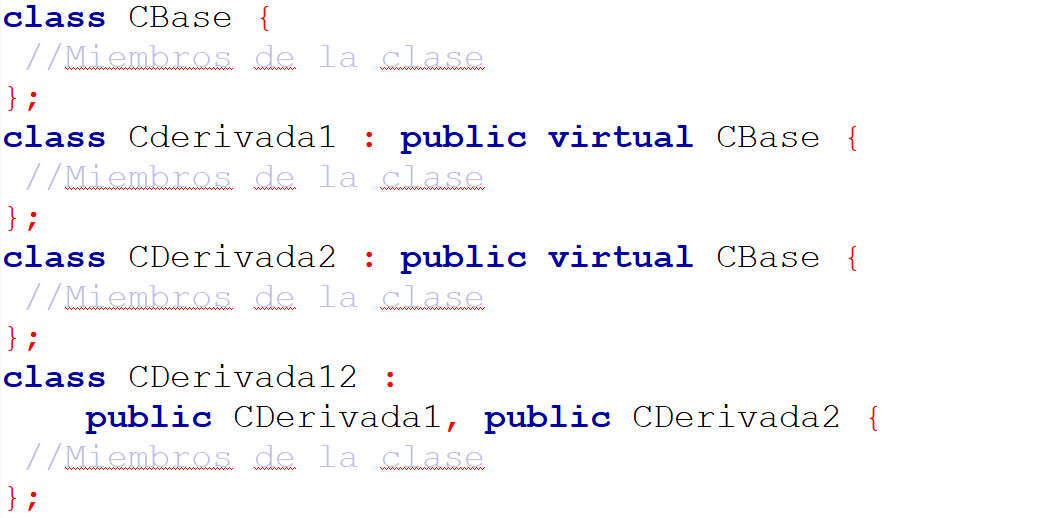
En la jerarquía de clases presentada anteriormente, el mecanismo de herencia normal hace que la clase derivada CDerivada12 herede dos veces los miembros de la clase CBase: una a través de la clase CDerivada1 y otra a través de la clase CDerivada2. Esto implica que cuando creemos un objeto CDerivada12, éste contendrá dos subobjetos de CBase (subobjetos: áreas de memoria para soportar la clase base), lo que origina, no solo los errores de ambigüedad comentados anteriormente, sino un derroche de espacio. De acuerdo con los expuesto, podemos imaginarnos un objeto Cderivada12 según se muestra en la figura siguiente.

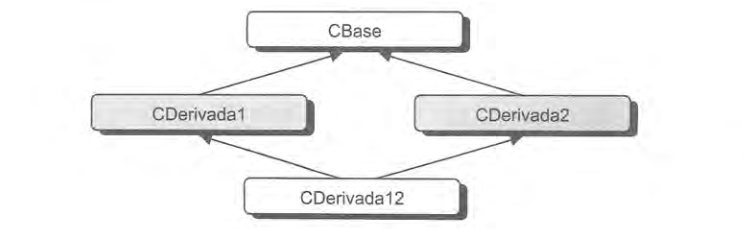


Para evitar que la clase base común, se replique en la clase más derivada (CBase se replica en CDerivada12), es necesario poner en marcha el mecanismo de herencia virtual. Esto hará que la clase base común pase de ser clase base replicada a clase base virtual.

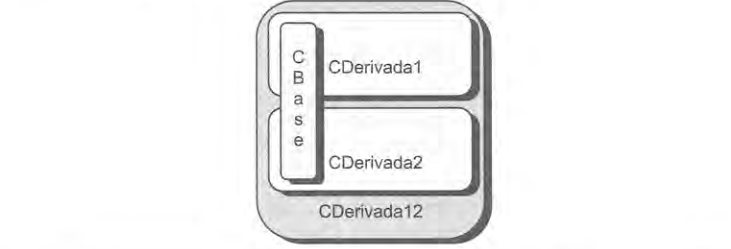
¿Cómo se activa el mecanismo de herencia virtual? Este mecanismo se pone en marcha cuando, en un proceso de derivación, la clase base común se declara **virtual**, lo que asegurará que solo se utilizará un subobjeto de la misma.

Por ejemplo, para evitar que CBase se replique en la clase CDerivada12, en el proceso de derivación la declararemos **virtual**, así:



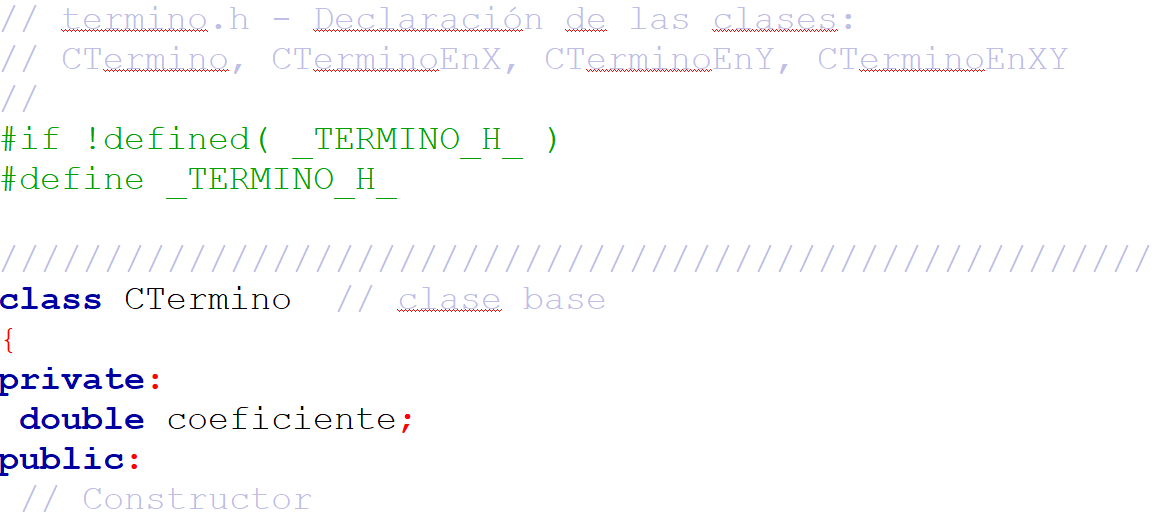


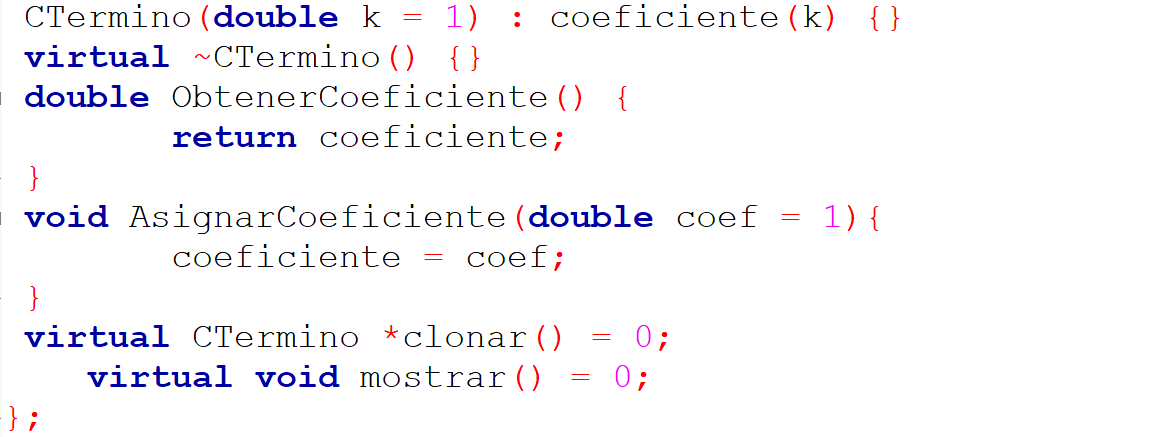
En un proceso de herencia múltiple, toda clase base que se declare virtual será representada en las derivadas mediante un solo subobjeto, y cada clase no declarada virtual tendrá su propio subobjeto:

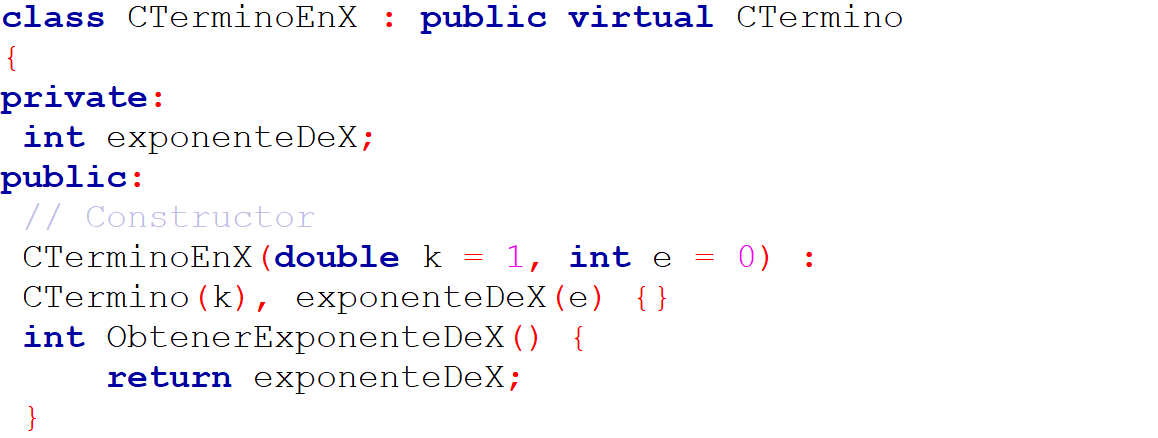


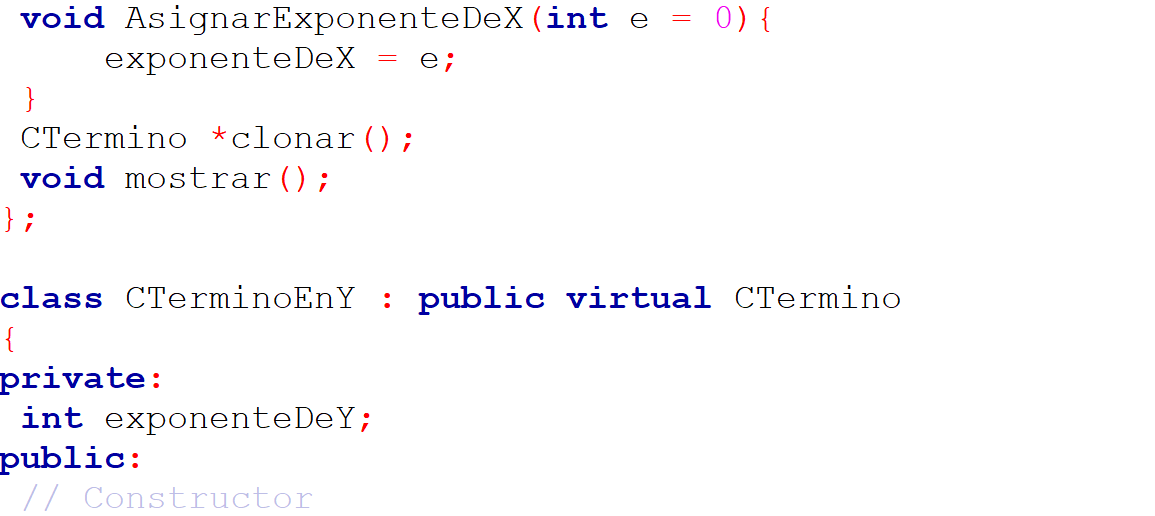
Cuando se construye un objeto completo (un objeto de la clase más derivada), el constructor de una *clase base virtual* es siempre llamado antes que los constructores de las clases base no virtuales, independientemente del orden en el que se hayan especificado en la lista de iniciación. Esto es, cualquier subobjeto virtual se inicia antes que otro no virtual. Los destructores son siempre invocados en el orden inverso.

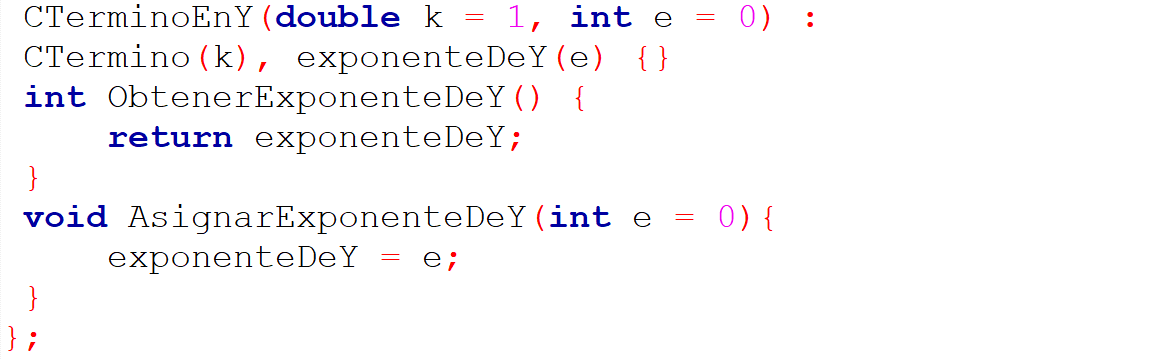
La siguiente versión del programa anterior demuestra con claridad como afrontar un caso como el expuesto.

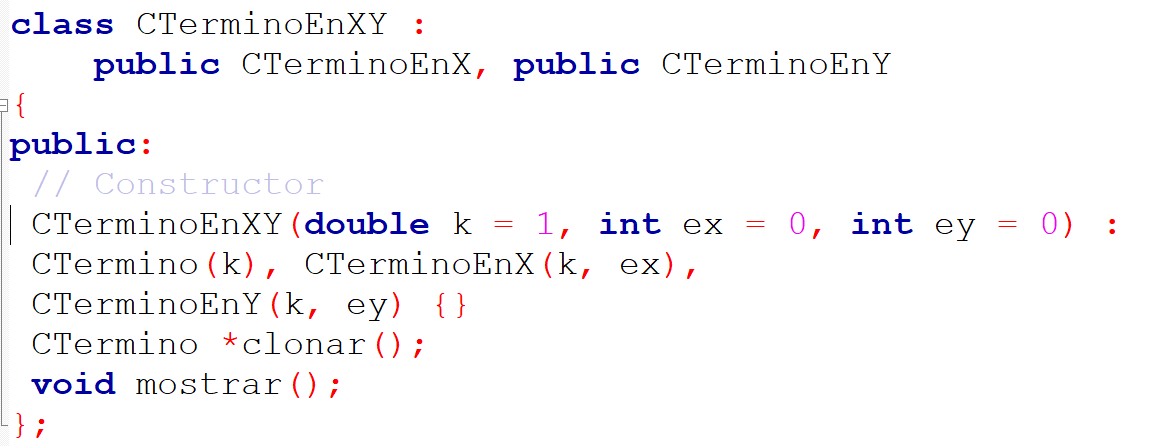




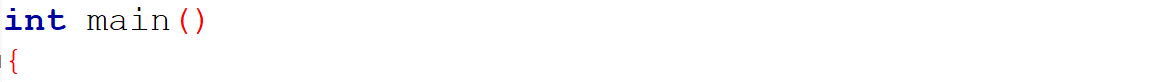


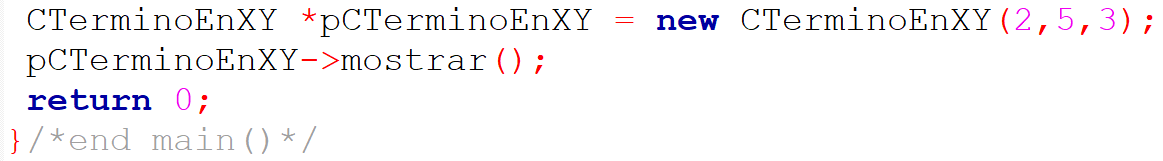






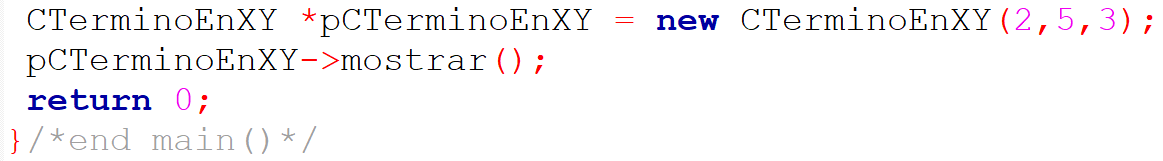
Jerarquía de clases con herencia múltiple y clase base virtual





Obsérvese que al derivar las clases *CTerminoEnX* y *CTerminoEnY*, se declara la clase *CTermino* **virtual**. Esto asegura que el compilador solamente pasará a la clase *CTerminoEnXY* una copia de *CTermino*.

Véamos ahora, cómo se construye un objeto completo. Por ejemplo, cuando se ejecuta la sentencia:



se invoca al constructor de CTerminoEnXY para construir un objeto dinámico de esta clase, el cual llama para su ejecución primero al constructor de CTermino, después al constructor de CTerminoEnX, después al constructor de CTerminoEnY y por último se ejecuta el cuerpo del constructor CTerminoEnXY; esto es, el constructor de la *clase base virtual* es llamado antes que los constructores de las clases base no virtuales. Los destructores son ejecutados en el orden inverso. Como consecuencia, en la lista de iniciadores del constructor CTerminoEnXY se ha especificado CTermino(k); de no hacerlo así, *coeficiente* sería iniciado con el valor por omisión.

Para poder observar el orden de ejecución de los constructores y de los destructores (ojo, no confunda el orden en el que se llaman a los constructores con el orden en el que se ejecuta el cuerpo de cada uno de ellos), puede hacer que éstos muestren un mensaje cuando se ejecuten. En este caso, si ejecuta el código de la función **main** anterior, podrá observar una salida similar a la mostrada a continuación:





Si la clase base no fuera virtual, el iniciador CTermino(k) del constructor CTerminoEnXY no tendría sentido, porque CTermino no es una clase base directa.

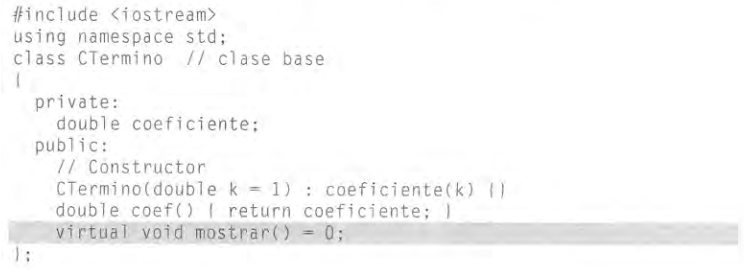
# Redefinición de métodos de base virtuales

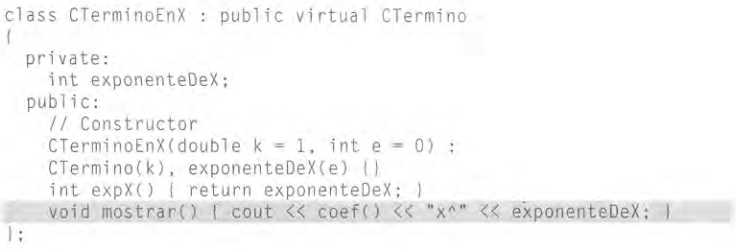
Un método virtual de una clase base virtual puede ser redefinido en una clase derivada directa o indirecta. Por ejemplo, volviendo al esquema general anterior, si CBase declara un método virtual, éste puede ser redefinido por CDerivada1, por CDerivada2 o por CDerivada12.

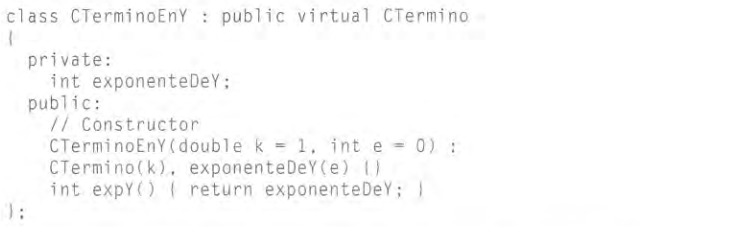
Cuando la interfaz de una clase base virtual proporciona varios métodos virtuales, no todos tienen que ser redefinidos por todas las clases derivadas, sino que clases derivadas diferentes pueden redefinir métodos diferentes. Por ejemplo, supongamos que CBase declara los métodos virtuales *mvb1* y *mvb2*; CDerivada1, podría redefinir *mvb1* y CDerivada2 *mvb2*.

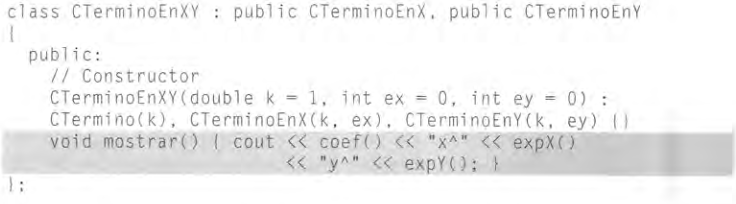
¿Qué sucede si clase derivadas diferentes redefinen el mismo método? Esto es válido si y sólo si cada una de las clases queda bien formada. En una clase bien formada, por cada método virtual declarado en esa clase o en cualquiera de sus clases base directas o indirectas, hay un único método que redefine ese método en cada una de las otras clases que lo redefinen.

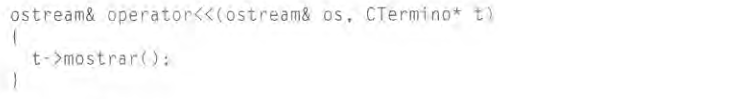
Por ejemplo, si CBase declara un método virtual mvb, que es redefinido por CDerivada1 y CDerivada2, sería un error que no fuera redefinido también por CDerivada12, ya que, por haber heredado esta clase mvb dos veces, no habría en ella un único método que redefine mvb en cada una de las otras clases base directas o indirectas que lo redefinen. A continuación, se muestra otra versión del programa anterior con la que se puede experimentar lo expuesto:

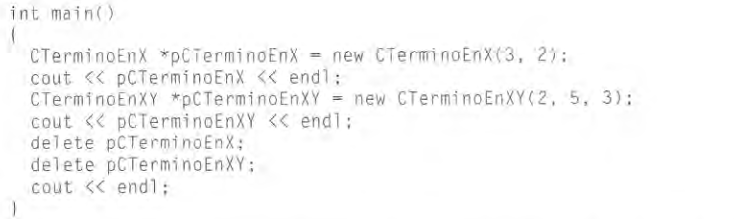












Este ejemplo constituye una jerarquía de clases con una clase base virtual polimórfica CTermino, de la que se derivan dos clases hermanas, CTerminoEnX y CTerminoEnY, de las que se deriva otra clase CTerminoEnXY. La clase CTermino se ha declarado abstracta incluyendo el método virtual puro mostrar, y si analizamos sus clases derivadas directas o indirectas, todas están bien formadas. La clase CTerminoEnY también es abstracta porque no redefine el método mostrar. Esto se ha hecho así porque para crear términos dependientes de una variable ya tenemos la clase CTerminoEnX.