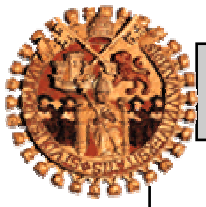


Herencia en C++

Generalidades sobre el concepto de herencia

Se entiende por herencia el proceso por el que un objeto puede tomar características de otro objeto[1]. La herencia se puede usar de dos formas:

1. Cuando una clase escrita no llega a cubrir las necesidades que exige un determinado problema, se puede retocar el código cambiando las características que se requieren, e ignorando el resto. Esto permite una programación rápida. Esto es una forma de **reutilización del código**.
2. Se expresa un determinado problema como una jerarquía de clases, donde existe una clase base de la que se derivan otras subclases. La interfaz de la clase base es común a todas las clases derivadas.

La herencia facilita la construcción de clases, una de las propiedades más interesantes de la filosofía de la **POO**.

Cuando una clase hereda de otra clase se denomina ***clase derivada***, y a veces ***clase hija, subclase*** o ***clase descendiente***. La clase de la cual hereda recibe el nombre de ***clase base***, en ocasiones también se la conoce como ***clase padre, superclase*** o ***clase ascendiente***[2]

Cualquier clase puede ser una clase base. Es más una clase puede ser clase base para más de una clase derivada. A su vez, una clase derivada puede ser clase base de otras clases.

Se recuerda que los elementos de una clase se pueden dividir en tres grupos:

- **public:** Accesibles desde cualquier función del programa.
- **private:** Sólo pueden acceder a ellos funciones miembros de la clase, o funciones afines.
- **protected:** Sólo pueden acceder a ellos las funciones miembro de la clase, de clases derivadas o las funciones afines.

La siguiente tabla recoge todas las posibles combinaciones de los accesos a las secciones de una clase:

Especificador de acceso	Desde la propia clase	Desde las clases derivadas	Desde el exterior
public	Si	Si	Si
protected	Si	Si	No
private	Si	No	No

[1] En **C++** sólo se pueden heredar clases, no funciones ordinarias ni variables.
[2]Los términos más tradicionales en **C++** son clase base y clase derivada, y son los que se van a utilizar en este documento mayoritariamente.

Cuando se hereda, las clases derivadas reciben las características de la clase base, pudiendo añadir su personalidad propia en forma de nuevas características, o modificando las heredadas. El compilador hace una copia de la clase base en la clase derivada, permitiendo al programador añadir o modificar miembros sin que la clase base sufra alteración alguna.

Hasta el momento se ha vendido como gran ventaja de los mecanismos de herencia la **reutilización del código**, que permite que un programador pueda utilizar una clase como clase base de otras nuevas clases, con la característica añadida que no hace falta comprender el fuente de la clase base, sólo hace falta saber lo que hace. Pero la herencia tiene otra interesante característica, la **extensibilidad**. Esta propiedad permite que los programas sean fácilmente ampliables, así de una clase base se pueden derivar varias clases que tengan un interfaz común, pero su realización y las acciones que llevan a cabo sean diferentes, así el programa principal controlará un grupo de estos objetos, puede utilizar una función miembro a cualquier objeto, pero el efecto será diferente, dependiendo de las subclases específicas[3].

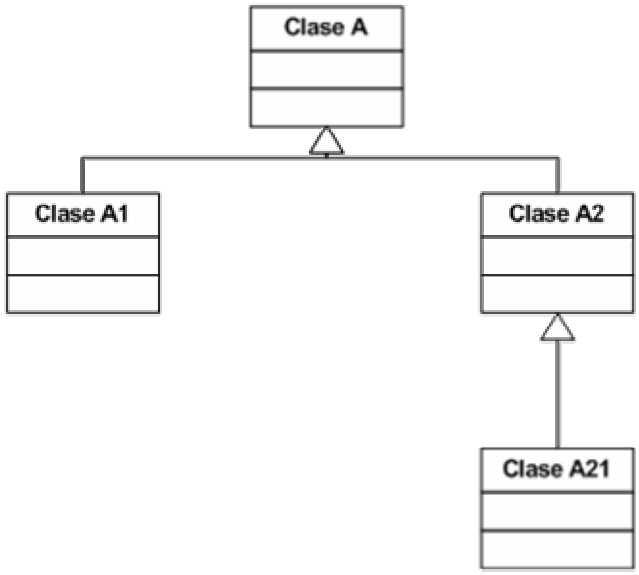
Resumiendo las dos grandes ventajas del mecanismo de herencia en **C++** son:

- Reutilización del código
- Extensibilidad

Dentro de los mecanismos de herencia se van a diferenciar dos tipos:

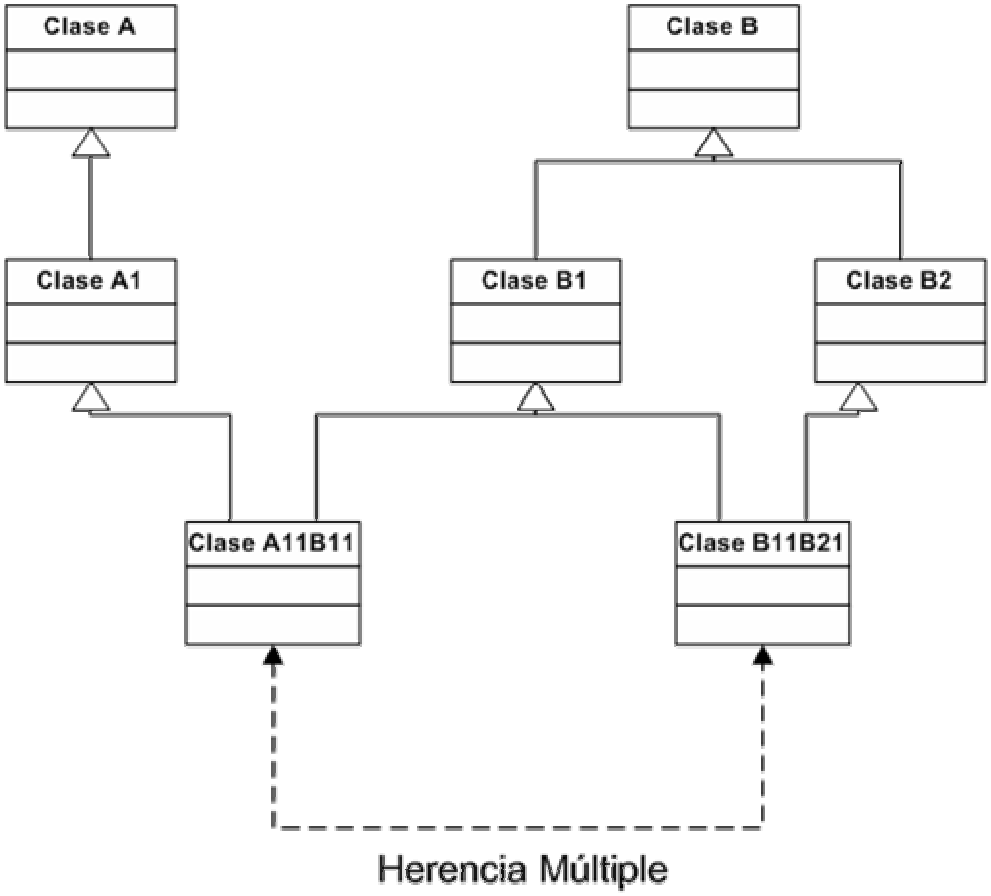
- **La Herencia Simple.** Una clase derivada tiene un sólo padre.
- **La Herencia Múltiple.** Una clase derivada hereda de más de una clase base.

A continuación se presenta una representación gráfica de ambos tipos de herencia.



Herencia Simple

[3] Se está haciendo una referencia implícita al otro gran concepto que se estudiará más adelante: el polimorfismo



Herencia Múltiple

Herencia Simple

Es cuando una clase derivada hereda de una única clase, es decir una clase derivada sólo tiene un padre o ascendiente. Por su parte una clase base puede tener tantos descendientes como sean necesarios sin limitación alguna. Es un sistema jerárquico en forma arborescente, similar a la estructura de directorios de algunos sistemas operativos[4].

La forma general de la herencia en **C++** es:

```
class <nombre_clase_derivada>: [<acceso>] <nombre_clase_heredada> {
          //  Cuerpo de la declaración de la clase
};
```

El *nombre_clase_heredada* se refiere a una clase base declarada previamente. Ésta pude estar ya compilada, o puede que se declare en el mismo programa que la derivada, en este segundo caso se debe declarar la clase base antes de la derivada, o al menos declarar el nombre de la clase base como una referencia anticipada.

[4] Como **MSDOS** o **UNIX**

El ***acceso*** puede ser **private, protected** o **public**. Si se omite se supone que el acceso es **private**[5], de forma que si se quiere dar un acceso **public** o **protected** se debe hacer explícitamente.

Los elementos **private** de la *clase base* son inaccesibles para la *clase derivada*, sea cual se el ***acceso***.

Si el ***acceso*** es **public**, todos los elementos **public** y **protected** de la *clase base* seguirán siendo **public** y **protected** respectivamente en la *clase derivada.*

Si el ***acceso*** es **private**, entonces todos los elementos **public** y **protected** de la *clase base* pasarán a ser elementos **private** de la *clase derivada*.

Si el ***acceso*** es **protected** todos los miembros **public** y **protected** de la *clase base* se convierten en elementos **protected** de la *clase derivada*.

En la siguiente tabla se resumen los especificadores de acceso:

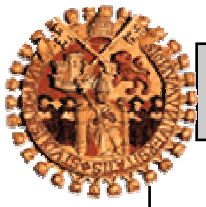
Acceso	Descripción
public	Los miembros públicos de la clase base son miembros públicos de la clase derivada. Los miembros protegidos de la clase base son miembros protegidos de la clase derivada. Los miembros privados de la clase base no son accesibles para la clase derivada.
private	Los miembros públicos de la clase base son miembros privados de la clase derivada. Los miembros protegidos de la clase base con miembros privados de la clase derivada. Los miembros privados de la clase base no son accesibles para la clase derivada.
protected	Los miembros públicos de la clase base son miembros protegidos de la clase derivada. Los miembros protegidos de la clase base son miembros protegidos de la clase derivada. Los miembros privados de la clase base no son accesibles para la clase derivada.

Se puede añadir a la *clase derivada* datos y funciones miembro. Dentro de las funciones miembro de la clase derivada se puede llamar a las funciones miembro y manejar los datos miembro que estén en la sección pública y protegida de la *clase base.*

En una clase derivada se heredan todos los datos miembro de la clase base excepto los estáticos.

Algunas funciones miembro no se heredan de forma automática. Éstas son los constructores, el destructor, las funciones amigas, las funciones estáticas de la clase, y el operador de asignación sobrecargado.

[5] Si en lugar de trabajar con clases, se estuviera trabajando con estructuras, el acceso por defecto sería **public**



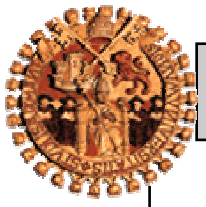
Se va a ilustrar lo que se ha dicho hasta el momento de la herencia en los siguientes ejemplos:

```
// Programa: Herencia Simple
// Fichero:  HER_SIM1.CPP

#include <iostream.h>

// Se define una clase sencilla
class Primera {
protected:
   int i, j;
public:
   Primera() {i=j=0;}
   Primera(int a, int b) {i=a; j=b;}
   void Entra_ij(int a, int b) {i=a; j=b;}
   void Muestra_ij (void) {cout << "\n" << i << " " << j;}
};


// Se tiene otra clase, Segunda, que es derivada de la clase Primera
// Las variables i, j de Primera pasan a ser miembros protected de
// Segunda, ya que el acceso es public
class Segunda: public Primera {
   int k;
public:
   void Entra_k(int a) {k=a;}
   void Muestra_k (void) {cout << "\n" << k;}
};


// Se tiene otra clase, Tercera, que es derivada de la clase Segunda
// Las variables i, j de Primera pasan a ser miembros protected de
// Tercera, ya que el acceso es public. Sin embargo no tiene acceso a k de
// Segunda, ya que esta variable es private.
class Tercera: public Segunda {
public:
   void f (void) { i=j=2;}
};


void main (void)
{
   Primera P(1,2);
   Segunda S;
   Tercera T;

   S.Entra_ij(3,4);
   S.Entra_k(5);

   P.Muestra_ij();      // Saca 1 2
   S.Muestra_ij();      // Saca 3 4
   S.Muestra_k();       // Saca 5

   T.f();
   T.Muestra_ij();      // Saca 2 2
```

```
   T.Entra_k(3);
   T.Muestra_k();      // Saca 3

   S.Muestra_k();     // Saca 5
   T.Entra_ij(5,6);
   T.Muestra_ij();     // Saca 5 6
   P.Muestra_ij();     // Saca 1 2
}
```

Como **Segunda** declara que **Primera** es pública, entonces los elementos **protected** de **Primera** pasan a ser elementos **protected** de **Segunda**, con lo que pueden ser heredados por **Tercera.**
Veamos que ocurre si se cambia si **Segunda** tiene un acceso **private** a **Primera**.

```
// Programa: Herencia Simple
// Fichero:  HER_SIM2.CPP

#include <iostream.h>

// Se define una clase sencilla
class Primera {
protected:
   int i, j;
public:
   Primera() {i=j=0;}
   Primera(int a, int b) {i=a; j=b;}
   void Entra_ij(int a, int b) {i=a; j=b;}
   void Muestra_ij (void) {cout << "\n" << i << " " << j;}
};

// Se tiene otra clase, Segunda, que es derivada de la clase Primera
// Las variables i, j de Primera pasan a ser miembros private de
// Segunda, ya que el acceso es private.
class Segunda: private Primera {
   int k;
public:
   void Entra_k(int a) {k=a;}
   void Muestra_k (void) {cout << "\n" << k;}
};


// Se tiene otra clase, Tercera, que es derivada de la clase Segunda
// Las variables i, j de Primera son inaccesibles por
// Tercera ya que en Segunda son privadas
class Tercera: public Segunda {
public:
   void f (void) {
   // i=j=2;      // Esto ya no es posible
   }
};


void main (void) {
   Primera P(1,2);
   Segunda S;
```




```
    Tercera T;

    // S.Entra_ij(3,4);         // La función Entra_ij es un miembro
                                // público de Primera que pasa a ser un
                                // miembro privado de Segunda, y ya no se
                                // puede llamar desde el exterior

    S.Entra_k(5);

    P.Muestra_ij();             // Saca 1 2
    // S.Muestra_ij();          // Lo mismo ocurre con la función Muestra_ij

    S.Muestra_k();              // Saca 5

    T.f();
    // T.Muestra_ij();          // Esta función se hereda como privada en
                                // Segunda y Tercera. Sólo es pública desde
                                // la clase Primera

    T.Entra_k(3);
    T.Muestra_k();              // Saca 3

    S.Muestra_k();              // Saca 5
}
```

Ventajas e inconvenientes de la derivación Privada y Protegida

Cuando se utiliza el especificador de acceso **private** o el especificador de acceso **protected** en la herencia, se está asegurando que sólo las partes públicas de la clase derivada podrán ser utilizadas por el resto del programa, y por las otras clases derivadas que se derivan a partir de ésta.

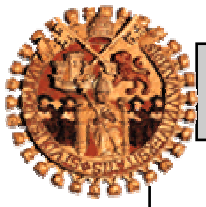
El acceso **private** "corta" el acceso, no la herencia, a partir de la clase derivada. El acceso **protected** "corta" el acceso, no la herencia, desde el exterior, pero no desde las clases derivadas.

Los inconvenientes llegan porque al utilizar estos tipos de accesos se está complicando el árbol de herencia, alcanzando una mayor complejidad la determinación por parte del programador, o de alguien que lea el programa, del derecho de acceso a los miembros de cada clase.

Estos tipos de derivaciones se emplean muy poco, hay que ser muy cuidadoso cuando se utilizan, y tener unos buenos motivos. Se puede considerar similar a tener un miembro que es una instancia de otra clase diferente.

Constructores y Destructores en la herencia

Los constructores y destructores no son heredados por las clases derivadas. Sin embargo, una instancia de una clase derivada contendrá todos los miembros de la clase base, y éstos deben ser iniciados. En consecuencia, el constructor de la clase base debe ser llamado por el constructor de la clase derivada.



Cuando se escribe un constructor para la clase derivada, se debe especificar una forma de inicio para la base. La forma de hacerlo es utilizar una sintaxis similar a la empleada con los miembros de inicio para los objetos miembro de una clase. Esto es, se coloca **:** después de la lista de argumentos del constructor de la clase derivada, y seguidamente el nombre de la clase base y la lista de argumentos.

```
// Programa: Constructores en la herencia
// Fichero:  CONSTR.CPP

#include <iostream.h>

class B {
   int a, b, c;
public:
   B(){a=b=c=0;}

   B( int a1, int a2, int a3 )
   {
      a=a1; b=a2; c=a3;
   }

   void DarValores( int a1, int a2, int a3 )
   {
      a=a1; b=a2; c=a3;
   }

   void MostrarEnteros (void)
   {
      cout << "\n" << a << " " << b << " " << c;
   }
};


class C: public B {
   float x, y;
public:
   C():B(0,0,0) {x=y=0.0;}

   C(float a1, float a2, int a3, int a4, int a5) : B(a3, a4, a5)
   {
      x=a1; y=a2;
   }

   void MostrarFloat(void)
   {
      cout << "\n" << x << " " << y;
   }
};

void main()
{
   C b, c(1, 2, 3, 4, 5);
   b.MostrarEnteros();
   b.MostrarFloat();
   c.MostrarEnteros();
   c.MostrarFloat();
```

```
}
```

Cuando se declara un objeto de una clase derivada, el compilador ejecuta el constructor de la clase base en primer lugar, y después ejecuta el constructor de la clase derivada. Si la clase derivada tuviera objetos miembro, sus constructores se ejecutarían después del constructor de la clase base, pero antes que el constructor de la clase derivada. Por lo que respecta a los destructores en la jerarquía de clases, se llaman en orden inverso de la derivación, es decir de la clase derivada a la clases base. La última clase creada es la que se destruye en primer lugar.

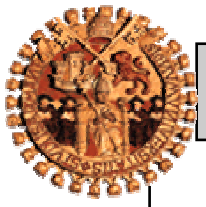
Se puede omitir el iniciador de la clase base siempre que la clase base tenga un constructor por defecto.

Se pueden especificar miembros de inicio, y un iniciador de la clase base, si es que la clase derivada tiene objetos miembro. Sin embargo, no se puede definir miembros de inicio para objetos miembro definidos en la clase base. Vamos a ver esto con un ejemplo:

```
// Programa: Constructores en la herencia
// Fichero:  CONSTR1.CPP

#include <iostream.h>
#include <string.h>

class A {
   unsigned y;
public:
   A ( unsigned a1 ) : y(a1) {}

   ~A() { cout << "\nDestructor de A\n";}

   void saca_y (void) {cout  << y;}
};

class B {
   char *s;
public:
   B ( char *msg )
   {
      s = new char [strlen (msg) + 1];
      strcpy ( s, msg );
   }

   ~B()
   {
      delete [] s;
      cout << "\nDestructor de B\n";
   }

   void msg (void ) { cout << "\n" << s;}
};

class C {         // Esta clase tiene un objeto miembro de la clase A
protected:
   int a, b, c;
   A a1;
public:
```

```
   C():a1(0){a=b=c=0;}

   C( int a1, int a2, int a3, unsigned a4 ) : a1(a4)
   {
      a=a1; b=a2; c=a3;
   }

   ~C() { cout << "\nDestructor de C\n";}

   void DarValores( int a1, int a2, int a3 )
   {
      a=a1; b=a2; c=a3;
   }

   void MostrarEnteros (void)
   {
      cout << "\n" << a << " " << b << " " << c << " " ;
      a1.saca_y();
   }
};

class D: public C {//Clase derivada de C, tiene un objeto miembro de B
   float x, y;
   B b1;
public:
   D():C(0,0,0,0),b1("") {x=y=0.0;}

   D(float a1,float a2,int a3,int a4,int a5,unsigned a6,char *m) :
       C(a3, a4, a5, a6), b1(m)
   {
      x=a1; y=a2;
   }

   ~D() { cout << "\nDestructor de D\n";}

   void MostrarFloat(void)
   {
      cout << "\n" << x << " " << y;
      b1.msg();
   }
};

void main()
{
   D d1, d2(1, 2, 3, 4, 5, 6, "Hola");

   d1.MostrarEnteros();
   d1.MostrarFloat();

   d2.MostrarEnteros();
   d2.MostrarFloat();
}
```

Es interesante destacar de este programa tres aspectos:



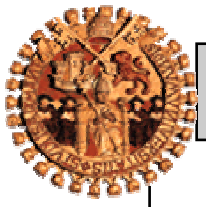
1. El lugar donde se encuentran los miembros que inician los objetos miembro de las clases **C** y **D**.
2. El orden en que se llaman los constructores de las clases al crear un objeto de la clase **D**: en primer lugar el de la clase **A**, seguidamente el de la clase **C**, a continuación el de la clase **B**, y por último el constructor de la clase **D**.
3. El orden en que se llaman los destructores: en primer lugar el de la clase **D**, luego el de la clase **B**, a continuación el de la clase **C**, y por último el de la clase **A.**

Conversiones entre las clases base y derivadas

Dado que una *clase derivada* se puede entender como un superconjunto de la *clase base*, que va a contener todos los miembros de la *clase base*, una instancia de la *clase derivada* se puede emplear de forma automática como si fuera una instancia de la *clase base*. El caso contrario no es cierto, no se puede tratar un objeto de la *clase base* como si fuera un objeto de una *clase derivada,* ya que el objeto de la *clase base* no tiene todos los miembros de la *clase derivada*.

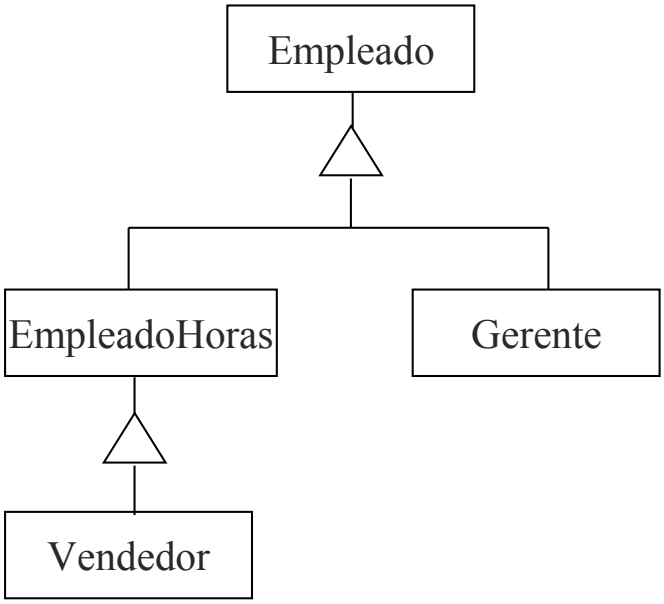
El siguiente programa muestra como funciona el tema de las conversiones de un objeto de una clase derivada a uno de la clase base.

```
// Programa: Herencia Simple: Conversiones entre clases
// Fichero:  HER_SIM3.CPP

#include <iostream.h>

class Primera {
protected:
   int i, j;
public:
   Primera() {i=j=0;}
   Primera(int a, int b) {i=a; j=b;}
   void Entra_ij(int a, int b) {i=a; j=b;}
   void Muestra_ij (void) {cout << "\n" << i << " " << j;}
};

class Segunda: public Primera {
   int k;
public:
   void Entra_k(int a) {k=a;}
   void Muestra_k (void) {cout <<  " " << k;}
};

void main (void)
{
   Primera P(1,2);
   Segunda S;

   S.Entra_ij(3,4);
   S.Entra_k(5);
   P.Muestra_ij();      // Saca 1 2
   S.Muestra_ij();      // Saca 3 4
   S.Muestra_k();       // Saca 5
   P=S;                 // Correcto
   P.Muestra_ij();      // Saca 3 4
}
```



Un punto mucho más interesante es la conversión de un puntero a una clase derivada, en un puntero a una clase objeto.

Para ver este caso, se va a presentar el siguiente ejemplo. Se tiene la siguiente escala de empleados, que se va a representar como una estructura de clases derivadas[6].




```
// Programa: Empleados
// Fichero:  EMPLE1.CPP

#include <iostream.h>
#include <string.h>

class Empleado {
protected:
   char nombre[30];
public:
   Empleado();
   Empleado(char *nom);
   char *Nombre() const;
};

class aHoras: public Empleado {
protected:
   float horas;
   float salario;
public:
   aHoras ( char *nom );
   void   FijaSalario ( float s );
   void   FijaHoras   ( float h );
```

[6] Dado que cualquier clase, incluso aquellas que se derivan de otra, pueden ser clases bases de nuevas derivaciones, se crea lo que en **POO** se denomina ***Jerarquía de Clases***.

```
   void   SacaNombre  (void) const;
};

class Gerente: public Empleado {
   float salario_semanal;
public:
   Gerente ( char *nom );
   void FijaSalario (float s);
};

class Vendedor: public aHoras {
   float comision;
   float ventas;
public:
   Vendedor (char *nom );
   void FijaComision ( float c );
   void FijaVentas   ( float v );
   void SacaComision (void);
};

// Funciones miembro de la clase Empleado
Empleado::Empleado () {
   memset ( nombre, ' ', 30);
}

Empleado::Empleado( char * nom) {
   strncpy (nombre, nom, 30);
}

char *Empleado::Nombre() const {
   return (char *)nombre;
}


// Funciones miembro de la clase aHoras
aHoras::aHoras ( char *nom ):Empleado (nom) {
   horas=0.0; salario=0.0;
}

void aHoras::FijaSalario (float s) {
   salario=s;
}

void aHoras::FijaHoras   (float h) {
   horas=h;
}

void aHoras::SacaNombre  (void) const {
   cout << "\nEl nombre del trabajador es: " << nombre;
}


// Funciones miembro de la clase Gerente
Gerente::Gerente ( char *nom ):Empleado (nom) {
   salario_semanal=0.0;
}
```

```
void Gerente::FijaSalario (float s) {
   salario_semanal=s;
}


// Funciones miembro de la clase Vendedor
Vendedor::Vendedor (char *nom ):aHoras(nom) {
   comision=0.0;
   ventas=0.0;
}

void Vendedor::FijaComision (float c) {
   comision=c;
}

void Vendedor::FijaVentas   (float v) {
   ventas=v;
}

void Vendedor::SacaComision (void) {
   cout << "\n" << comision;
}

void main(void)
{
   Empleado *ptr_emp;
   aHoras   a1("Juan Antonio P.");
   Gerente  g1("Alberto Lamazares");
   Vendedor v1("Ana Elena");

   ptr_emp=&a1;
   cout << "\n" << ptr_emp->Nombre();

   ptr_emp=&g1;
   cout << "\n" << ptr_emp->Nombre();

   ptr_emp=&v1;
   cout << "\n" << ptr_emp->Nombre();
}
```

Se puede utilizar un puntero a **Empleado** para apuntar directamente a una instancia de **aHoras,** a una instancia de **Gerente**, o a una instancia de **Vendedor.** Sin embargo, si la definición de la clase **Vendedor** hubiera sido:

```
class Vendedor : aHoras { ...
```

para poder con un puntero a **Empleado** apuntar a una instancia de la clase **Vendedor** se hubiera tenido que hacer el siguiente *casting.*

```
ptr_emp=(Empleado *)&v1;
```

esto es debido a que cuando en la herencia se utiliza el método de acceso **private** o **protected** no se puede convertir de forma implícita un puntero de la clase derivada a un puntero de la clase base.

Cuando se hace referencia a un objeto a través de un puntero, el tipo de puntero determina que funciones miembro se pueden llamar. Si se utiliza un puntero a una clase base para apuntar a un objeto de una clase derivada, sólo se podrán utilizar las funciones definidas en la clase base. Por ejemplo, estudiar el siguiente programa principal.

```
void main(void)
{
   Vendedor v1("Ana Elena"), *v2_ptr;
   aHoras *a1_ptr;

   v2_ptr=&v1;
   a1_ptr=&v1;

   a1_ptr->FijaHoras(40.0);      // Se llama a aHoras::FijaHoras
   v2_ptr->FijaSalario(6.0);     // Se llama a aHoras::FijaSalario
// a1_ptr->FijaVentas(1000.0);  // Error:no existe aHoras::FijaVentas
   v2_ptr->FijaVentas(1000.0);  // Se llama a Vendedor::FijaVentas
}
```

Ambos punteros **v2_ptr** y **a1_ptr** apuntan al mismo objeto **Vendedor**. No se puede llamar a la función **FijaVentas** a través de **a1_ptr** porque la clase **aHoras** no tiene definida una función que se llame así.

El caso contrario, que un puntero de una clase derivada apunte a un objeto de la clase base, se puede hacer mediante un *casting* apropiado, pero es muy peligroso y no es recomendable hacerlo, ya que no se está seguro a lo que se apunta. Así el siguiente caso:

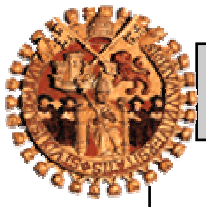
```
void main(void)
{
   aHoras a1("Pepe Pérez");
   Empleado *ptr = &a1;
   Vendedor *v1;

   v1=(Vendedor *) ptr;             // Lícito pero incorrecto
   cout << "\n" << v1->Nombre();    // Saca Pepe Pérez
   v1->FijaComision(1.5);           // La clase aHoras no tiene
                                    // el miembro FijaComision
   v1->SacaComision();              // Saca basura
}
```

Ambigüedades en la Herencia Simple

Una clase derivada puede tener un miembro con el mismo nombre que un miembro de la clase base. Cuando ocurre esto, y se referencia este nombre que está duplicado, el compilador asumirá que se desea acceder al miembro de la clase derivada.
Este hecho se conoce como anulación, invalidación o suplantación[7] del miembro de la clase base por el miembro de la clase derivada.

Para acceder al miembro de la clase base se deberá utilizar el operador de resolución de ámbito.

[7] Es la traducción al castellano del término inglés ***overriding***, que es el que se emplea para denominar este efecto



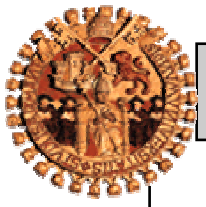
Ejemplo:

```
// Programa: Empleados
// Fichero:  AMBIGU.CPP

#include <iostream.h>

class Base {
protected:
   int j;
public:
   Base (int x):j(x){}

   void Calcula (void) {cout << "\n" << j*j;}
};

class Derivada: public Base {
   int j;
public:
   Derivada (int y, int z) : j(y), Base(z) {}

   void Calcula (void) {cout << "\n" << j*Base::j;}
};

void main()
{
   Derivada d(4,8);
   d.Calcula();          // Saca 32
   d.Base::Calcula();    // Saca 64
}
```

En el ejemplo se ha visto como un dato miembro ha sido anulado. Esto mismo puede ocurrir con las funciones miembro, es decir, una clase derivada además de incluir nuevos métodos, se pueden redefinir métodos que estén en la clase base. Esto se hace con la idea de aumentar la utilidad de estas funciones o de cambiar el sentido de éstas. A este fenómeno se le conoce como **redefinición** o **anulación** de funciones y se logra sobrecargando funciones miembro de la base. Una función miembro redefinida oculta todas las funciones miembro heredadas con el mismo nombre.

Cuando una función existe en la clase base y en la clase derivada, se ejecutará la función de la clase derivada, ya que ésta anula la de la clase base, para acceder a la función definida en la clase base se debe de utilizar el operador de resolución de ámbito **::**.

Mediante la **redefinición** de funciones se está entrando en lo que se denomina ***programación incremental***. Dado que una subclase hereda toda la representación de su clase base, se puede anular de una forma selectiva algunos o todos los métodos de la clase base, reemplazándolos por unos métodos mejorados. Esta es una potente propiedad que da una nueva dimensión a la programación, y hace que después de trabajar en **C++**, los programadores encuentren a otros lenguajes del tipo de **C** o **Ada,** como lenguajes insulsos.



Herencia Múltiple

Una clase puede tener más de una clase base. Esto significa que una clase puede heredar de dos o más clases. A este fenómeno se le conoce como **Herencia Múltiple**[8].

La sintaxis de la herencia múltiple es una extensión de la utilizada para la herencia simple. La manera de expresar este tipo de herencia es mediante una lista de herencia, que consta de las clases de las que se hereda separadas por comas. La forma general es:

```
class < nombre_clase_derivada > : < lista_de_herencia > {
      // Cuerpo de la clase
};
```

Vamos a ilustrar la herencia múltiple con un sencillo ejemplo.

```
// Programa: Herencia Múltiple
// Fichero:  HER_MUL1.CPP

#include <iostream.h>

class Base1 {
protected:
   int b1;
public:
   void Inicia_B1(int v) { b1=v; }
};

class Base2 {
protected:
   int b2;
public:
   void Inicia_B2(int v) { b2=v; }
   int  Ret (void) { return b2; }
};

class Der: public Base1, public Base2 {
public:
   void print (void) { cout << "\nb1 = " << b1 << "\tb2 = " << Ret(); }
};

void main (void)
{
   Der d;

   d.Inicia_B2(78);
   d.Inicia_B1(34);
   d.print();                  // Saca 34 y 78
}
```

[8] La *herencia múltiple*, o *derivación de clases con clases base múltiples*, está permitida a partir de la versión **2.0** de **C++**

Este es un ejemplo muy sencillo en el que una clase, **Der**, hereda de otras dos clases **Base1, Base2**. En este programa no hay complicaciones de ningún tipo, y no es muy interesante.

En la herencia múltiple las reglas de inicio de los miembros son una extensión de los vistos en la herencia simple. De igual forma, los mecanismos de herencia de los miembros están gobernados por las declaraciones de **private, protected** y **public** en las clases.

El orden en el que se especifican las clases bases[9] en la lista de herencia sólo afecta al orden en que se van a invocar los constructores, y destructores de las clases base desde los constructores y destructores de la clase derivada.

Vamos a ver a continuación otro ejemplo de herencia múltiple, en el que las clases tengan constructores.

```
// Programa: Herencia Múltiple
// Fichero:  HER_MUL2.CPP

#include <iostream.h>

class A {
protected:
   int a;
public:
   A (int valor) {
   cout << "\nConstructor de A.";
   a=valor;
   }

   ~A () {  cout << "\nDestructor de A."; }
};

class B {
protected:
   int b;
public:
   B (int valor) {
   cout << "\nConstructor de B.";
   b=valor;
   }

   ~B () {  cout << "\nDestructor de B."; }
};


// C hereda de A y de B
class C: public A, public B {
public:
   C(int v1, int v2) : B(v2), A(v1) {
      cout << "\nConstructor de C.";
   }
```

[9] No está permitido que una misma clase base aparezca más de una vez en la lista de herencia. Es que sea una clase base directa más de una vez para una clase derivada, sin embargo pueden aparecer múltiples ocurrencias de la clase base de forma indirecta, a través de una clase base directa.

```
   ~C () {  cout << "\nDestructor de C."; }

   int operar (void) { return a*a+b*b; }
};

void main (void)
{
   C obj(4, 5);
   cout << "\n" << obj.operar();
};
```

La salida del programa anterior sería la siguiente:

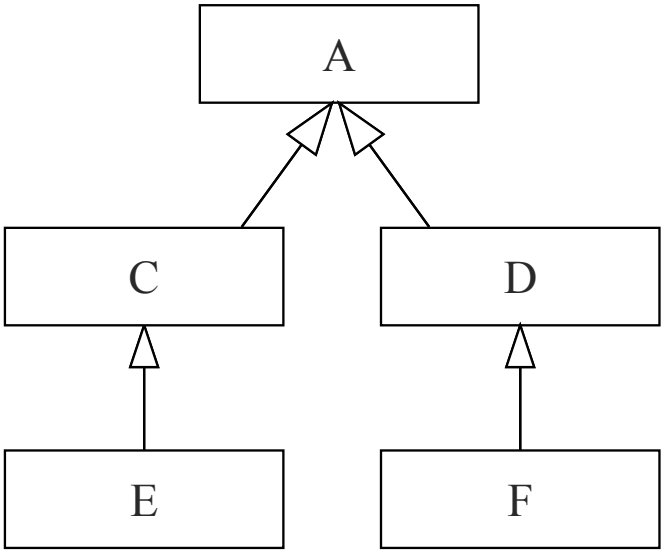
```
Constructor de A
Constructor de B
Constructor de C
41
Destructor de C
Destructor de B
Destructor de A
```

De lo que se deduce que las clases bases se construyen en el orden en que aparecen en la declaración de **C**. Una vez que las clases han recibido los valores iniciales, se ejecuta el constructor de la clase derivada.

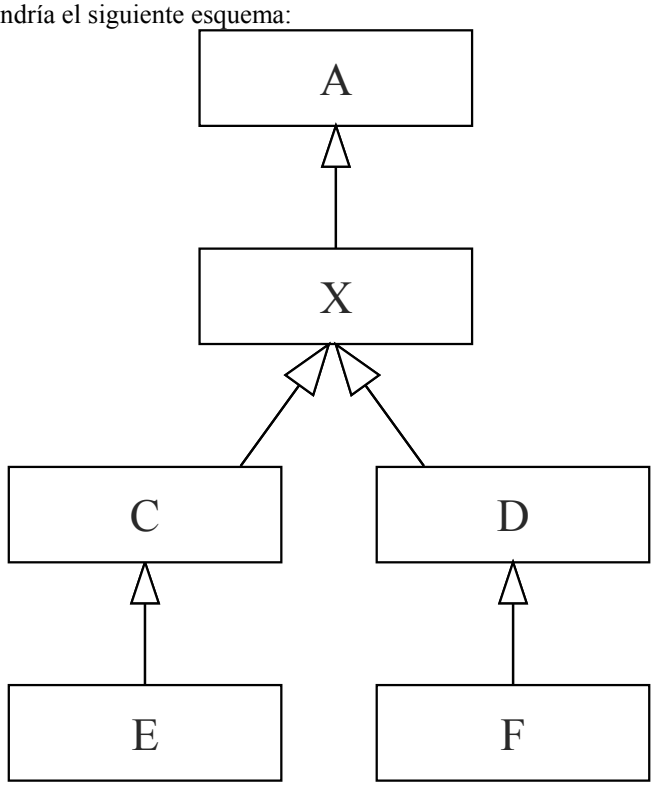
Mientras que ninguna de las clases base tenga constructor, o que todas las clases bases tengan un constructor por defecto, la clase derivada no tiene porque tener constructor. De todas formas es una buena práctica de programación que todas las clases tengan un constructor para de esta forma contar con un mecanismo que permita pasar argumentos a los constructores de sus clases bases.

En cuanto a los destructores, decir que al igual que en la herencia simple, se llaman en orden inverso a la derivación.

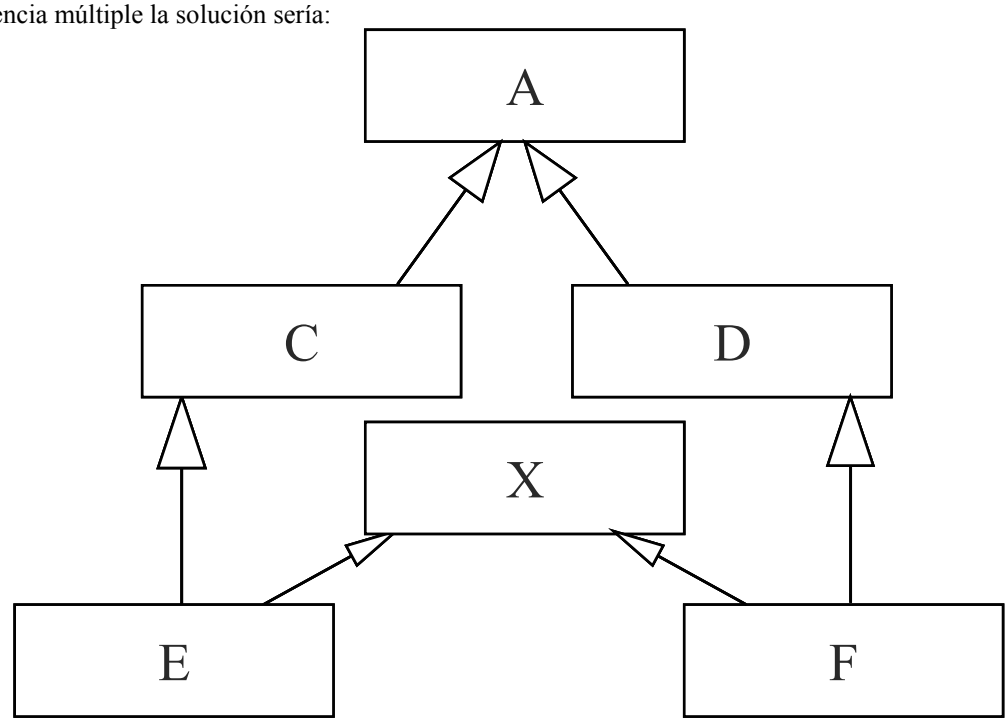
Los mecanismos de herencia múltiple permiten al programador situar la información donde sea necesario. En contra, la herencia simple puede situar parte de los datos donde no se requieran. Supóngase que se tiene el siguiente árbol de jerarquía, y sólo se desea que existan datos de otra clase **X** en las clases **E** y **F**.



Con herencia simple se tendría el siguiente esquema:



Con herencia múltiple la solución sería:



La herencia múltiple es más apropiada para definir objetos compuestos o una combinación de objetos. Mientras que la herencia simple es más útil para definir objetos que son más especializaciones de objetos generales.

Ambigüedades en la Herencia Múltiple

Supongamos que modificamos el programa **HER_MUL1.CPP** de la siguiente forma:

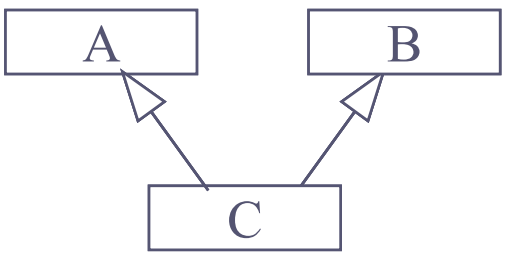
```
// Programa: Herencia Múltiple
// Fichero:  HER_MUL3.CPP

#include <iostream.h>

class Base1 {
protected:
   int b1;
public:
   void Inicia(int v) { b1=v; }
};

class Base2 {
protected:
   int b2;
public:
   void Inicia(int v) { b2=v; }
   int  Ret (void) { return b2; }
```

```
};

class Der: public Base1, public Base2 {
public:
   void print (void) { cout << "\nb1 = " << b1 << "\tb2 = " << Ret(); }
};
void main (void)
{
   Der d;

   d.Inicia(78);  // Ambigüedad. No compilaría
   d.Inicia(34);  // Ambigüedad. No compilaría
   d.print();     // Saca 34 y 78
}
```

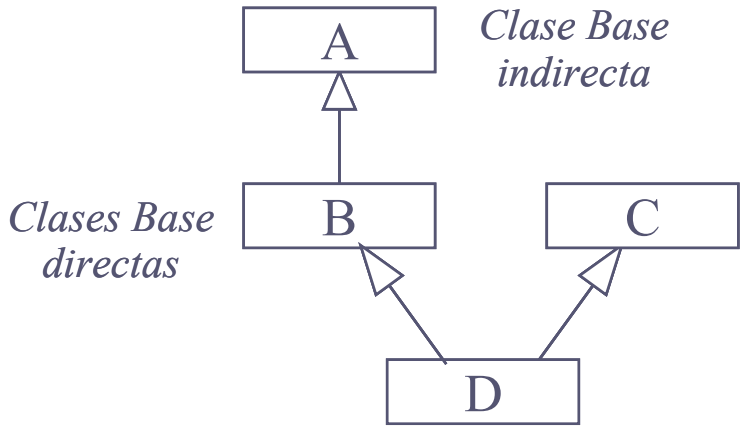
Ahora se tiene que tanto la clase **Base1** como la clase **Base2** cuentan con una función miembro con el mismo nombre **Inicia**. La clase **Der** hereda estas dos funciones y no tiene una función miembro **Inicia** que las anule, así cuando se produce la llamada **d.Inicia(78)** el compilador no sabe como actuar, ya que existe una ambigüedad. Para solventarla se debe recurrir al operador de resolución de ámbito. Así la función **main** que sería correcta es:

```
void main (void)
{
   Der d;

   d.Base1::Inicia(78);
   d.Base2::Inicia(34);
   d.print();            // Saca 78 y 34
}
```

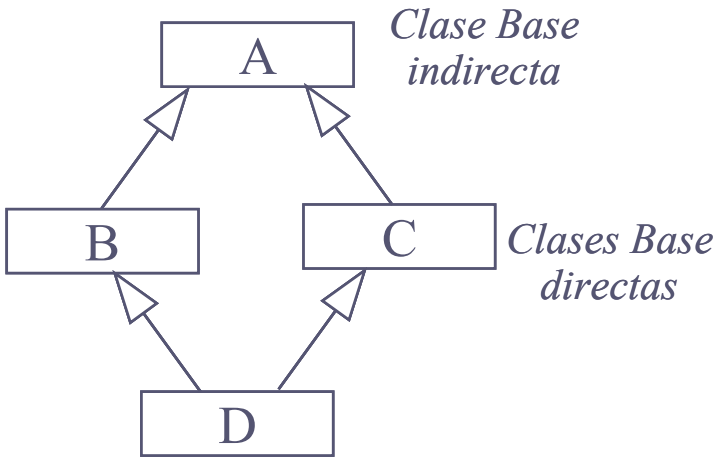
La herencia múltiple puede introducir más ambigüedades. Según se ha visto en el apartado anterior, en la lista de herencia una clase base solamente puede especificarse una sola vez de forma directa[10] y se pueden ver con sólo examinar la declaración de la base. Pero de forma indirecta[11] una clase base puede aparecer varias veces, ocurre cuando se oculta en el interior de una clase base que se ha heredado. De forma gráfica:



[10] Y se denominan clases base directas
[11] Y se denominan clases base indirectas



La ambigüedad se produce cuando se heredan dos clases base, que a su vez contienen ambas a la misma clase indirecta, produciéndose la circunstancia de que la nueva clase base contiene dos veces los objetos de la clase base indirecta. Gráficamente la situación se puede representar:



Ambigüedad

Así en el siguiente ejemplo:

```
#include <iostream.h>

class A {
protected:
   int i;
};

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

```
public:
   int Retorna (void) { return i;}
};
void main (void)
{
   D d1;
   int j;
   j=d1.Retorna();
}
```

El compilador no puede conocer cual es la copia de **i** a la que se está haciendo referencia, y por ejemplo el compilador **Borland C++ 4.02** daría el siguiente error:
Error NONAME00.CPP 11: Member is ambiguous: 'A::i' and 'A::i' in function D::Retorna()

Para solucionar esta ambigüedad se recurrirá al operador de resolución de ámbito dentro de la función **Retorna()** de la siguiente forma:

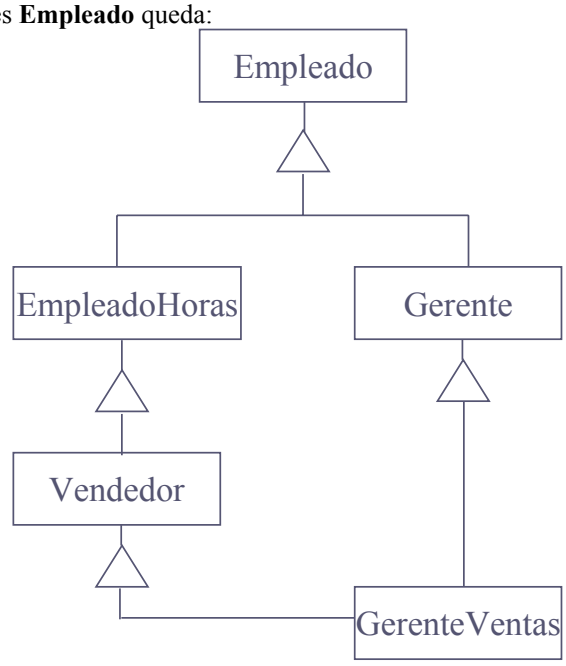
```
int Retorna (void) {return C::i;}
```

Se va a ampliar la jerarquía de clases **Empleado** añadiéndole una nueva clase que se denomine **GerenteVentas** y cuya definición es la que sigue:

```
class GerenteVentas: public Vendedor, public Gerente {
public:
   GerenteVentas(char *nom);
};
```

la clase **GerenteVentas** contiene de forma indirecta a la clase **Empleado** dos veces, una a través de **Vendedor** y otra a través de **Gerente**.

Entonces la jerarquía de clases **Empleado** queda:



Si se tiene una función **main()** como la que sigue, habrá ambigüedad y el programa no compilará:

```
void main(void)
{
   GerenteVentas gv("Marcelino Ostroski");
   cout << "\n" << gv.Nombre();     // Error
}
```

Para solucionar el problema debemos recurrir al operador de resolución de ámbito, como se muestra a continuación:

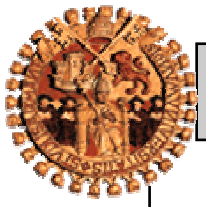
```
void main(void)
{
   GerenteVentas gv("Marcelino Ostroski");

   cout << "\n" << gv.Gerente::Nombre();
   // Esto sería equivalente
   // cout << "\n" << gv.Vendedor::Nombre();

}
```

Si una clase actúa como clase base de forma indirecta más de una vez, habrá ambigüedad cuando se produzcan conversiones entre la clase base y la clase derivada. Si se tiene el siguiente fragmento de código en la función **main()**:

```
void main(void)
{
   Empleado *ptr_emp;
   GerenteVentas *ptr_gv, gv("Pepe");

   ptr_gv=&gv;
   ptr_emp=ptr_gv;

   cout << "\n" << ptr_emp->Nombre();
}
```

De nuevo existe ambigüedad. Para solucionarlo debemos recurrir a un casting.

```
void main(void)
{
   Empleado *ptr_emp;
   GerenteVentas *ptr_gv, gv("Pepe");

   ptr_gv=&gv;
   ptr_emp=(Gerente *)ptr_gv;
   cout << "\n" << ptr_emp->Nombre();
}
```



Dado que una clase no tiene que tener repetidas las variables miembro, no debe tener más de una copia de una clase base. Para evitar esta duplicación se debe definir la clase base como una **clase base virtual**. Para hacer esto se debe utilizar la palabra reservada **virtual** en aquellas clases que hereden de forma directa la clase base. Ver el siguiente ejemplo:

```
// Programa: Empleados
// Fichero:  BASEVIR.CPP

#include <iostream.h>

class A {
public:
   int v1;
};
class B : virtual public A {
public:
   float v2;
};

class C : virtual public A {
public:
   float v3;
};

class D : public B, public C {
public:
   char v4;
};

void main()
{
   D o;

   o.v4='a';
   o.v3=3.14;
   o.v2=1.7;
   o.v1=9;

   cout << o.v1 << " " << o.v2 << " " << o.v3 << " " << o.v4 << endl;
}
```

Así se pueden eliminar las ambigüedades en la herencia múltiple. Pero en algunos casos el orden de llamada a los constructores puede producir confusión, debido a que **C++** sigue el siguiente orden de iniciación:

1. Todas las clases virtuales se inician primero, es decir sus constructores se ejecutan en primer lugar.
El constructor de cada clase virtual se llama sólo una vez. Si existen varias clases virtuales, sus constructores se ejecutan en el orden en que han sido declarados.

2. Las clase base no virtuales se inician en el orden que aparecen en la declaración de clases.
Después del paso anterior se ejecutan los constructores de las clases base no virtuales en el orden en que aparecen en la lista de herencia.

3. Por último se ejecuta el constructor de la clase derivada.

Estas reglas se aplican recursivamente.

La secuencia seguida para llamar a los destructores es la inversa de la que se ha expuesto para los constructores.

Así el programa completo de la jerarquía de clases **Empleado** utilizando clases base virtuales es:

```
// Programa: Empleados
// Fichero:  EMPLE4.CPP

#include <iostream.h>
#include <string.h>

class Empleado {
protected:
   char nombre[30];
public:
   Empleado();
   Empleado(char *nom);
   char *Nombre() const;
};

class aHoras : virtual public Empleado {
protected:
   float horas;
   float salario;
public:
   aHoras (char *nom);
   void   FijaSalario (float s);
   void   FijaHoras   (float h);
   void   SacaNombre  (void) const;
};

class Gerente : virtual public Empleado {
   float salario_semanal;
public:
   Gerente (char *nom);
   void FijaSalario (float s);
};

class Vendedor : public aHoras {
   float comision;
   float ventas;
public:
   Vendedor (char *nom);
   void FijaComision (float c);
   void FijaVentas   (float v);
   void SacaComision (void);
};

class GerenteVentas : public Vendedor, public Gerente {
public:
   GerenteVentas(char *nom);
};
```

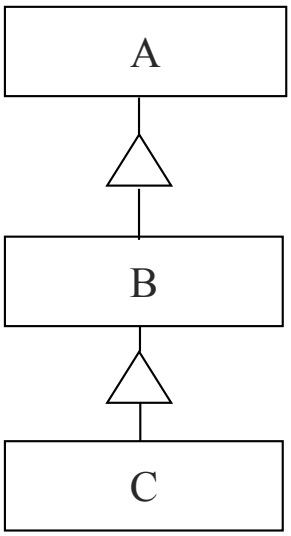
```
// Funciones miembro de la clase Empleado
Empleado::Empleado ()
{
   memset (nombre, ' ', 30);
}

Empleado::Empleado(char * nom)
{
   strncpy (nombre, nom, 30);
}

char *Empleado::Nombre() const
{
   return (char *)nombre;
}


// Funciones miembro de la clase aHoras
aHoras::aHoras (char *nom):Empleado (nom)
{
   horas=0.0; salario=0.0;
}

void aHoras::FijaSalario (float s)
{
   salario=s;
}

void aHoras::FijaHoras   (float h)
{
   horas=h;
}

void aHoras::SacaNombre  (void) const
{
   cout << "\nEl nombre del trabajador es: " << nombre;
}


// Funciones miembro de la clase Gerente
Gerente::Gerente (char *nom):Empleado (nom)
{
   salario_semanal=0.0;
}

void Gerente::FijaSalario (float s)
{
   salario_semanal=s;
}
```

```
// Funciones miembro de la clase Vendedor
Vendedor::Vendedor (char *nom):aHoras(nom)
{
   comision=0.0;  ventas=0.0;
}

void Vendedor::FijaComision (float c)
{
   comision=c;
}

void Vendedor::FijaVentas (float v)
{
   ventas=v;
}

void Vendedor::SacaComision (void)
{
   cout << "\n" << comision;
}


// Funciones miembro de la clase GerenteVentas
GerenteVentas::GerenteVentas(char *nom): Vendedor(nom),
                                         Gerente(nom),
                                         Empleado(nom){}


void main(void)
{
  GerenteVentas *p_gv, gv1("Pepe");
  Empleado *p_emp;

  p_gv  = &gv1;
  p_emp = p_gv;

  cout << p_emp->Nombre() << endl;
}
```

Fijarse en el constructor de la clase **GerenteVentas**, como las clases virtuales se inician en primer lugar si no se pone el miembro que inicie **Empleado(**nom**)**, se inicia la clase Empleado con basura en la variable **nombre.** El lugar en el que se ponga este miembro que inicie es independiente, ya que se ejecutará en primer lugar, así en el ejemplo está puesto detrás de los otros miembros de inicio, pero se llama de todas formas en primer lugar.

Reglas de Dominio

En una jerarquía de herencia, la **regla de dominio** se emplea para resolver ambigüedades. Si una clase base y una clase derivada tienen un miembro con el mismo nombre, el que esté en la clase derivada es el dominante.

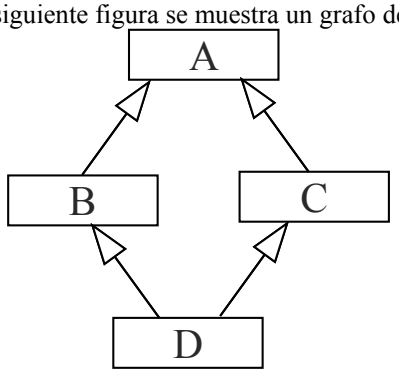



```
void A::func()
```

```
void B::func()
```

En esta figura se muestra una jerarquía de clases en la que la clase base **A** y la clase derivada **B** tienen un miembro con el mismo nombre. Así en el contexto de los objetos de la clase **B**, cuando se invoque a **func()** se refiere a **B::func()**, debido a que **B::func()** predomina sobre **A::func()**. De igual manera sucedería si se estuviese en el contexto de los objetos de la clase **C.**

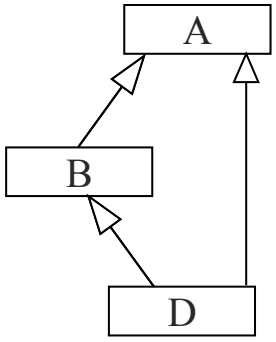
En la herencia múltiple el problema del dominio está relacionado con el uso o no de clases virtuales, así en la siguiente figura se muestra un grafo de herencia múltiple, con las definiciones de clase.




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

Cuando se invoca a una función de una clase derivada, el compilador la busca utilizando las clases, en el árbol de herencia, que sean accesibles siguiendo las reglas específicas. Si la función no se encuentra en la clase, se invocan y el compilador empieza a buscarlas en el árbol de herencia. Esta búsqueda se realiza en tiempo de compilación, no en tiempo de ejecución. Si diferentes clases tienen el mismo nombre de función, **C++** determina que función debe llamar.

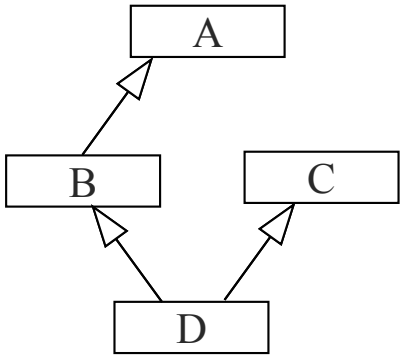
Ejemplo 1:
Considerar el siguiente árbol de herencia, con una función **func()** en las clases **A** y **B**.




```
// Programa: Reglas de dominio
// Fichero:  EJEM1.CPP

class A {
public:
   float func() { return 3.1416; }
};

class B : virtual public A {
public:
   float func() { return 2.778;  }
};

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

void main(void) {
   C c1;
   float j=c1.func();  // Se llama a B::func()
}
```

Se llama a la función **B::func()** como consecuencia de la regla de dominio, que señala que si dos clases contienen la función buscada, y una clase se deriva de la otra, la clase derivada predomina.

Ejemplo 2:
Considerar el siguiente árbol de herencia.

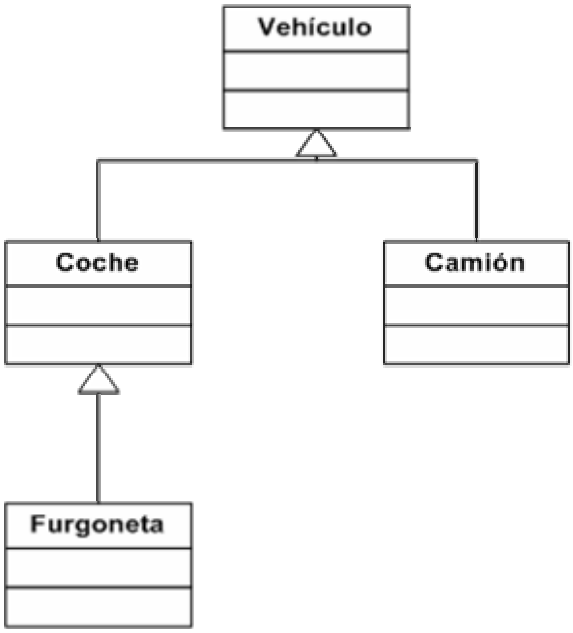


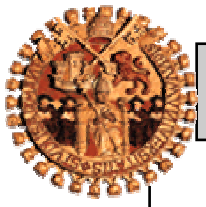
```
// Programa: Reglas de dominio
// Fichero:  EJEM2.CPP

class A {
public:
   float func() { return 3.1416;}
};

class C {
public:
   float func() { return 2.778;}
};

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

void main(void) {
   D d1;
   float j=d1.func();   // Error
}
```

En este ejemplo se tiene una situación en la que la regla de dominio no se puede aplicar, y se tiene un error por ambigüedad.

Ejercicios

1. Realizar una jerarquía de clases de forma que se represente el siguiente esquema.



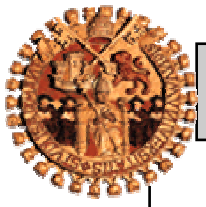


2. Encontrar los errores de este programa:

```
// Programa: Empleados
// Fichero:  EJER6-2.CPP

#include <iostream.h>

class Base
{
     int j;
  public:
     void Base (int x):j(x){}

  void Calcula (void)
  {
     cout << "\n" << j*j;
  }
};


class Derivada : public Base
{
     int j;
  public:
     Derivada (int y, int z) : j(z), Base(y) {}

  void Calcula (void)
  {
     cout << "\n" << j*Base::j;
  }
};

void main()
{
   Derivada d(4,8);

   d.Calcula();         // Saca 32
   d.Base::Calcula();   // Saca 64
}
```

3. Si empleamos la jerarquía de clases realizada en los programas EMPLE1.CPP y EMPLE2.CPP. ¿Cuál sería la salida del siguiente programa?

```
void main(void)
{
   aHoras a1("Juan Gómez");
   Empleado &e1=a1;

   cout << e1.Nombre() << "\n";

   e1.FijaSalario(9);
}
```



4. En un capítulo anterior se propuso un ejercicio para crear un programa para manejar una pila de números enteros. Vamos a suponer que se realizó un clase muy simple como la que se muestra a continuación:

```
class Pila {
   int *p;    // Pila
   int top;   // Cima
   int size;  // Tamaño
public:
   Pila(int s=TOPE)  // Constructor
   {
     if (s<=0)
     {
        error("Error en la creación de la pila.\n");
     }

     p = new int[s];
     size=s;
     top=0;
   }

   ~Pila()/  / Destructor
   {
      delete p;
   }

   void Pon (int elemento)
   {
      p[top++]=elemento;
   }

   int Quita (void)
   {
      return p[- -top];
   }

   int Size(void)
   {
      return size;
   }
};
```

Se quiere reutilizar este código, de forma que mediante una clase derivada, llámese **Pila2**, se detecte el desbordamiento y el vaciamiento de la pila.