HERENCIA

Necesidad de la herencia

La mente humana clasifica los *conceptos* de acuerdo a dos dimensiones: ***pertenencia*** y ***variedad***. Se puede decir que el Ford Fiesta es un tipo de coche (*variedad* o, en inglés, una relación del tipo ***is a***) y que una rueda es parte de un coche (*pertenencia* o una relación del tipo ***has a***). Antes de la llegada de la ***herencia***, en C ya se había resuelto el problema de la ***pertenencia*** mediante las ***estructuras***, que podían ser todo lo complejas que se quisiera. Con la ***herencia***, como se va a ver en este capítulo, se consigue clasificar los tipos de datos (*abstracciones*) por ***variedad***, acercando así un paso más la programación al modo de razonar humano.

Definición de herencia

La herencia, entendida como una característica de la programación orientada a objetos y más concretamente del C++, permite ***definir una clase modificando una o más clases*** ya existentes. Estas modificaciones consisten habitualmente en ***añadir nuevos miembros*** (variables o funciones), a la clase que se está definiendo, aunque también se puede ***redefinir*** variables o funciones miembro ya existentes.

La clase de la que se parte en este proceso recibe el nombre de ***clase base***, y la nueva clase que se obtiene se denomina ***clase derivada***. Ésta a su vez puede ser ***clase base*** en un nuevo proceso de derivación, iniciando de esta manera una ***jerarquía de clases***. De ordinario las ***clases base*** suelen ser ***más generales*** que las ***clases derivadas***. Esto es así porque a las clases derivadas se les suelen ir añadiendo características, en definitiva variables y funciones que diferencian concretan y particularizan.

En algunos casos una clase no tiene otra utilidad que la de ser ***clase base*** para otras clases que se deriven de ella. A este tipo de ***clases base***, de las que no se declara ningún objeto, se les denomina ***clases base abstractas*** (***ABC, Abstract Base Class***) y su función es la de agrupar miembros comunes de otras clases que se derivarán de ellas. Por ejemplo, se puede definir la clase ***vehiculo*** para después derivar de ella ***coche***, ***bicicleta***, ***patinete***, etc., pero todos los objetos que se declaren pertenecerán a alguna de estas últimas clases; no habrá vehículos que sean sólo ***vehículos***. Las características comunes de estas clases (como una variable que indique si está parado o en marcha, otra que indique su velocidad, la función de arrancar y la de frenar, etc.), pertenecerán a la ***clase base*** y las que sean particulares de alguna de ellas pertenecerán sólo a la ***clase derivada*** (por ejemplo el número de platos y piñones, que sólo tiene sentido para una ***bicicleta***, o la función ***embragar*** que sólo se aplicará a los vehículos de motor con varias marchas).

Este mecanismo de ***herencia*** presenta múltiples ventajas evidentes a primera vista, como la ***posibilidad de reutilizar código*** sin tener que escribirlo de nuevo. Esto es posible porque todas las ***clases derivadas*** pueden utilizar el código de la ***clase base*** sin tener que volver a definirlo en cada una de ellas.

VARIABLES Y FUNCIONES MIEMBRO PROTECTED

Uno de los problemas que aparece con la ***herencia*** es el del control del ***acceso a los datos***. ¿Puede una función de una ***clase derivada*** acceder a los datos ***privados*** de su ***clase base***? En principio una clase no puede acceder a los datos privados de otra, pero podría ser muy conveniente que una ***clase derivada*** accediera a todos los datos de su ***clase base***. Para hacer posible esto, existe el tipo de dato

protected. Este tipo de datos es ***privado*** para todas aquellas clases que no son derivadas, pero ***público*** para una ***clase derivada*** de la clase en la que se ha definido la variable como ***protected***.

Por otra parte, el proceso de herencia puede efectuarse de dos formas distintas: siendo la clase base ***public*** o ***private*** para la clase derivada. En el caso de que la clase base sea ***public*** para la clase derivada, ésta hereda los miembros ***public*** y ***protected*** de la clase base como miembros ***public*** y ***protected,*** respectivamente. Por el contrario, si la clase base es ***private*** para la clase derivada, ésta hereda todos los datos de la clase base como ***private***. La siguiente tabla puede resumir lo explicado en los dos últimos párrafos.

Tipo de dato de la clase base	Clase derivada de una clase base **public**	Clase derivada de una clase base **private**	Otras clases sin relación de herencia con la clase base
Private	*No accesible directamente*	*No accesible directamente*	*No accesible directamente*
Protected	*Protected*	*Private*	*No accesible directamente*
Public	*Public*	*Private*	*Accesible mediante operador (.) o (->)*

Tabla 1: Herencia pública y privada.

Como ejemplo, se puede pensar en dos tipos de cuentas bancarias que comparten algunas características y que también tienen algunas diferencias. Ambas cuentas tienen un ***saldo***, un ***interés*** y el ***nombre*** del titular de la cuenta. La ***cuenta joven*** es un tipo de cuenta que requiere la ***edad del propietario***, mientras que la ***cuenta empresarial*** necesita el ***nombre de la empresa***. El problema podría resolverse estableciendo una clase base llamada ***C_Cuenta*** y creando dos tipos de cuenta derivados de dicha clase base.

Para indicar que una ***clase deriva de otra*** es necesario indicarlo en la ***definición de la clase derivada***, especificando el modo -***public*** o ***private***- en que deriva de su ***clase base***:

```
class Clase_Derivada : public o private Clase_Base
```

De esta forma el código necesario para crear esas tres clases mencionadas quedaría de la siguiente forma:

```
#include <iostream.h>
class C_Cuenta {
    // Variables miembro
    private:
        char   *Nombre;    // Nombre de la persona
        double Saldo;      // Saldo Actual de la cuenta
        double Interes;    // Interés aplicado
    public:
        // Constructor
        C_Cuenta(const char *unNombre, double unSaldo=0.0, double unInteres=0.0)
        {
            Nombre = new char[strlen(unNombre)+1];
            strcpy(Nombre, unNombre);
            SetSaldo(unSaldo);
            SetInteres(unInteres);
        }
        // Destructor
        ~Cuenta()
            { delete [] Nombre; }
        // Métodos
        inline char *GetNombre()
            { return Nombre; }
        inline double GetSaldo()
            { return Saldo; }
```

```
        inline double GetInteres()
            { return Interes; }
        inline void SetSaldo(double unSaldo)
            { Saldo = unSaldo; }
        inline void SetInteres(double unInteres)
            { Interes = unInteres; }
        inline void Ingreso(double unaCantidad)
            { SetSaldo( GetSaldo() + unaCantidad ); }
        friend ostream& operator<<(ostream& os, C_Cuenta& unaCuenta)
        {
            os << "Nombre=" << unaCuenta.GetNombre() << endl;
            os << "Saldo=" << unaCuenta.GetSaldo() << endl;
            return os;
        }
};

class C_CuentaJoven : public C_Cuenta {
    private:
        int Edad;
    public:
        C_CuentaJoven(     // argumentos del constructor
            const char *unNombre,
            int        laEdad, double
            unSaldo=0.0, double
            unInteres=0.0)
                 : C_Cuenta(unNombre, unSaldo, unInteres)
            // se llama al constructor de la clase base en la línea previa.
        {
            Edad = laEdad;
        }
};
class C_CuentaEmpresarial : public C_Cuenta {
    private:
        char *NomEmpresa;
    public:
        C_CuentaEmpresarial(    // argumentos del constructor
            const char *unNombre,
            const char *laEmpresa,
            double     unSaldo=0.0,
            double     unInteres=0.0)
                 : C_Cuenta(unNombre, unSaldo, unInteres)
        // se llama al constructor de la clase base en la línea previa.
        {
            NomEmpresa = new char[strlen(laEmpresa)+1];
            strcpy(NomEmpresa, laEmpresa);
        }
        // Cuando una variable de este tipo se destruye se llamará
        //   primero el destructor de CuentaEmpresarial y posteriormente se
        //   llama automáticamente el destructor de la clase base.
        ~C_CuentaEmpresarial()
             { delete [] NomEmpresa; }
};

void main()
{
    C_CuentaJoven c1("Igor", 18, 10000.0, 1.0);
    C_CuentaEmpresarial c2("Juan", "MicroComputers Corp." ,10000000.0);
    // Ambas cuentas pueden llamar métodos definidos previamente
    cout << c1;
    cout << c2;
}
```

Si un miembro heredado se ***redefine*** en la clase derivada, el nombre redefinido oculta el nombre heredado que ya queda invisible para los objetos de la clase derivada.

Hay algunos elementos de la clase base que ***no pueden ser heredados***:

Constructores
Destructores
Funciones ***friend***
Funciones y datos estáticos de la clase
Operador de asignación (=) sobrecargado

Constructores de las clases derivadas: inicializador base

Un objeto de la ***clase derivada*** contiene todos los miembros de la ***clase base*** y todos esos miembros deben ser inicializados. Por esta razón el ***constructor de la clase derivada*** debe llamar al ***constructor de la clase base***. Al definir el ***constructor de la clase derivada*** se debe especificar un ***inicializador base***.

Como ya se ha dicho las clases derivadas ***no heredan los constructores*** de sus clases base. El ***inicializador base*** es la forma de llamar a los ***constructores de las clases base*** y poder así inicializar las variables miembro heredadas. Este ***inicilizador base*** se especifica poniendo, a continuación de los argumentos del constructor de la clase derivada, el carácter dos puntos (:) y el nombre del constructor de la clase o las clases base, seguido de una lista de argumentos entre paréntesis.

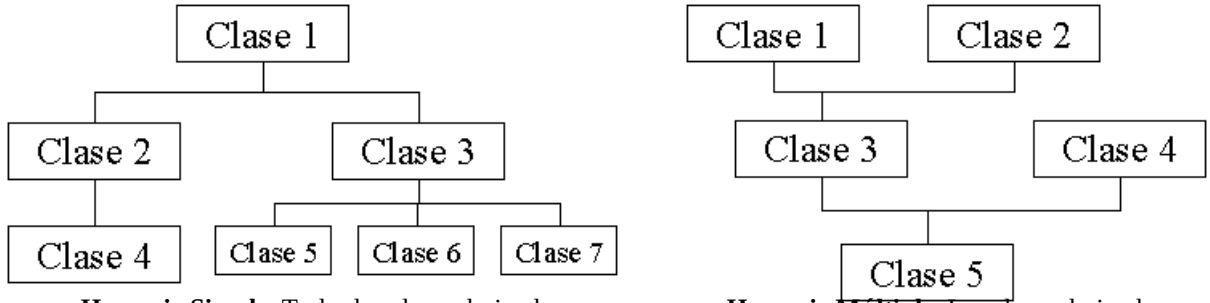
El ***inicializador base*** puede ser omitido en el caso de que la clase base tenga un ***constructor por defecto***. En el caso de que el constructor de la clase base exista, al declarar un objeto de la clase derivada se ejecuta primero el constructor de la clase base.

En el ejemplo del apartado anterior ya se puede ver como se llama al ***constructor de la clase base*** desde el ***constructor de la clase derivada***:

```
C_CuentaJoven(const char *unNombre, int laEdad, double unSaldo=0.0,
    double unInteres=0.0) : C_Cuenta(unNombre, unSaldo, unInteres)
```

Herencia simple y herencia múltiple

Una clase puede heredar variables y funciones miembro de una o más clases base. En el caso de que herede los miembros de una única clase se habla de ***herencia simple*** y en el caso de que herede miembros de varias clases base se trata de un caso de ***herencia múltiple***. Esto se ilustra en la siguiente figura:



Herencia Simple: Todas las clases derivadas tienen una única clase base

Herencia Múltiple: Las clases derivadas tienen varias clases base

Figura 4: Herencia simple y herencia múltiple.

Como ejemplo se puede presentar el caso de que se tenga una clase para el manejo de los datos de la empresa. Se podría definir la clase ***C_CuentaEmpresarial*** como la herencia múltiple de dos clases base: la ya bien conocida clase ***C_Cuenta*** y nueva clase llamada ***C_Empresa***, que se muestra a continuación:

```
class C_Empresa {
    private:
        char *NomEmpresa;
    public:
        C_Empresa(const char*laEmpresa)
        {
            NomEmpresa = new char[strlen(laEmpresa)+1];
            strcpy(NomEmpresa, laEmpresa);
        }
        ~C_Empresa()
            { delete [] NomEmpresa; }
        // Otros métodos ...
};

class C_CuentaEmpresarial : public C_Cuenta, public C_Empresa {
    public:
        C_CuentaEmpresarial(
            const char *unNombre,
            const char *laEmpresa,
            double     unSaldo=0.0,
            double     unInteres=0.0
        ) : C_Cuenta(unNombre, unSaldo, unInteres), C_Empresa(laEmpresa)
        // se llama a los constructores de las clases base en la línea previa
        {
            // Constructor
        }
        // Otros métodos
};
```

Clases base virtuales

Al utilizar la herencia múltiple puede suceder que, indirectamente, una clase ***herede varias veces*** los miembros de otra clase, tal como se ve en la figura 5.

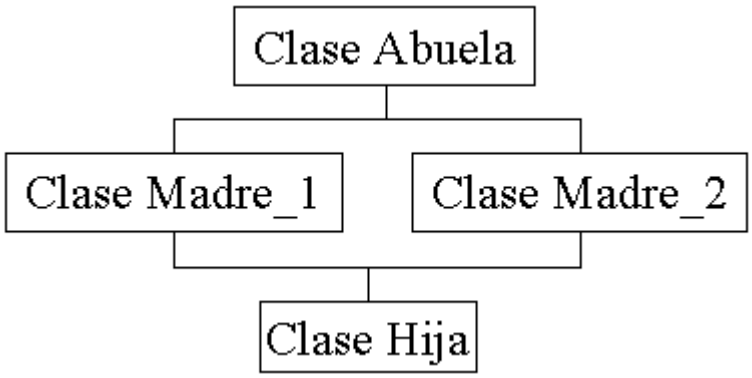


Figura 5: Necesidad de las clases base virtuales.

Si la clase ***Madre_1*** y la clase ***Madre_2*** heredan los miembros de la clase ***Abuela*** y la clase ***Hija*** hereda, a su vez, los miembros de las clases ***Madre_1*** y ***Madre_2***, los miembros de la clase ***Abuela*** se encontrarán duplicados en la clase ***Hija***. Para evitar este problema las clases ***Madre_1*** y ***Madre_2*** deben derivar de la clase ***Abuela*** declarándola ***clase base virtual***. Esto hace que los miembros de una clase de ese tipo se hereden tan sólo una vez. Un ejemplo de declaración de una ***clase base virtual*** es el que se presenta a continuación:

```
class Madre_1 : virtual public Abuela {
...
}
```

Conversiones entre objetos de clases base y clases derivadas

Es posible realizar ***conversiones*** o ***asignaciones*** de un ***objeto de una clase derivada*** a un ***objeto de la clase base***. Es decir se puede ir de lo más particular a lo más general, aunque en esa operación se pierda información, pues haya variables que no tengan a qué asignarse (el número de variables miembro de una clase derivada es mayor o igual que el de la clase de la que deriva).

Por el contrario las ***conversiones*** o ***asignaciones en el otro sentido***, es decir de lo más general a lo más particular, ***no son posibles***, porque puede suceder que no se disponga de valores para todas las variables miembro de la clase derivada.

Así pues, la siguiente asignación sería correcta:

```
Objeto_clase_base = Objeto_clase_derivada          // Asignación válida
```

mientras que esta otra sería incorrecta:

```
Objeto_clase_derivada = Objeto_clase_base          // Asignación incorrecta
```

En el siguiente ejemplo se pueden ver las distintas posibilidades de asignación (más bien de ***inicialización***, en este caso), que se presentan en la clase ***C_CuentaEmpresarial.***

```
void main()
{
    // Válido
    C_CuentaEmpresarial *c1 = new C_CuentaEmpresarial("Juan",
          "Jugos SA", 100000.0, 10.0);

    // Válido. Se utilizan los valores por defecto
    C_Cuenta *c2 = new C_CuentaEmpresarial("Igor", "Patata CORP");

    // NO VÁLIDO
    C_CuentaEmpresarial *c3 = new C_Cuenta("Igor", 100.0, 1.0);
    // ...
}
```

De forma análoga, se puede guardar la dirección almacenada en un ***puntero a una clase derivada*** en un ***puntero a la clase base***. Esto quiere decir que se puede hacer referencia a un ***objeto de la clase derivada*** con su dirección contenida en un ***puntero a la clase base***.

Al igual que sucede con los nombres de los objetos, en principio cuando se hace referencia a un objeto por medio de un puntero, ***el tipo de dicho puntero determina la función miembro que se aplica***, en el caso de que esa función se encuentre definida tanto en la clase base como en la derivada. En definitiva, ***un puntero a la clase base puede almacenar la dirección de un objeto perteneciente a una clase derivada***. Sin embargo, se aplicarán los métodos de la clase a la que pertenezca el puntero, no los de la clase a la que pertenece el objeto.

POLIMORFISMO

Polimorfismo, por definición, es la ***capacidad de adoptar formas distintas***. En el ámbito de la *Programación Orientada a Objetos* se entiende por ***polimorfismo*** la capacidad de ***llamar a funciones distintas con un mismo nombre***. Estas funciones pueden actuar sobre objetos distintos dentro de una jerarquía de clases, sin tener que especificar el tipo exacto de los objetos. Esto se puede entender mejor con el ejemplo de la figura 6:

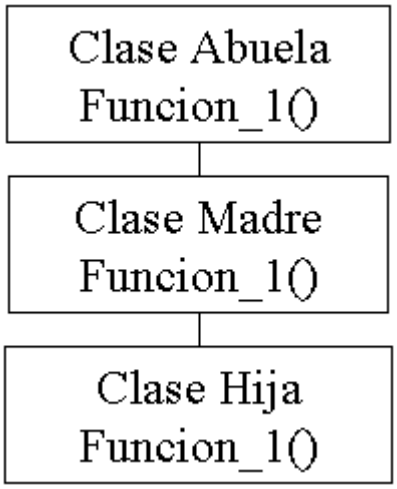


Figura 6: Funciones virtuales

En el ejemplo que se ve en la figura 6 se observa una jerarquía de clases. En todos los niveles de esta jerarquía está contenida una función llamada ***Funcion_1()***. Esta función no tiene por qué ser igual en todas las clases, aunque es habitual que sea una función que efectúe una operación muy parecida sobre distintos tipos de objetos.

Es importante comprender que ***el compilador no decide en tiempo de compilación*** cuál será la función que se debe utilizar en un momento dado del programa. Esa decisión se toma ***en tiempo de ejecución***. A este proceso de decisión en tiempo de ejecución se le denomina ***vinculación dinámica o tardía,*** en oposición a la habitual ***vinculación estática o temprana***, consistente en decidir en tiempo de compilación qué función se aplica en cada caso.

A este tipo de funciones, incluidas en varios niveles de una jerarquía de clases con ***el mismo nombre pero con distinta definición***, se les denomina ***funciones virtuales***. Hay que insistir en que la definición de la función en cada nivel es distinta.

El ***polimorfismo*** hace posible que un usuario pueda ***añadir nuevas clases*** a una jerarquía sin ***modificar o recompilar*** el código original. Esto quiere decir que si desea añadir una nueva clase derivada es suficiente con establecer la clase de la que deriva, definir sus nuevas variables y funciones miembro, y compilar esta parte del código, ensamblándolo después con lo que ya estaba compilado previamente.

Es necesario comentar que las ***funciones virtuales*** son *algo menos eficientes* que las funciones normales. A continuación se explica, sin entrar en gran detalle, el funcionamiento de las ***funciones virtuales***. Cada clase que utiliza ***funciones virtuales*** tiene un ***vector de punteros***, uno por cada ***función virtual***, llamado ***v-table***. Cada uno de los punteros contenidos en ese vector apunta a la ***función virtual*** apropiada para esa clase, que será, habitualmente, la ***función virtual*** definida en la propia clase. En el caso de que en esa clase no esté definida la ***función virtual*** en cuestión, el puntero de ***v-table*** apuntará a la ***función virtual*** de su clase base más próxima en la jerarquía, que tenga una definición propia de la ***función virtual***. Esto quiere decir que buscará primero en la

propia clase, luego en la clase anterior en el orden jerárquico y se irá subiendo en ese orden hasta dar con una clase que tenga definida la función buscada.

Cada objeto creado de una clase que tenga una ***función virtual*** contiene un puntero oculto a la ***v-table*** de su clase. Mediante ese puntero accede a su ***v-table*** correspondiente y a través de esta tabla accede a la definición adecuada de la ***función virtual***. Es este trabajo extra el que hace que las funciones virtuales sean menos eficientes que las funciones normales.

Como ejemplo se puede suponer que la ***cuenta_joven*** y la ***cuenta_empresarial*** antes descritas tienen una forma distinta de abonar mensualmente el interés al saldo.

En la ***cuenta_joven***, no se abonará el interés pactado si el saldo es inferior a un límite.

En la ***cuenta_empresarial*** se tienen tres cantidades límite, a las cuales se aplican factores de corrección en el cálculo del interés. El cálculo de la cantidad abonada debe realizarse de la siguiente forma:

1. Si el saldo es menor que 50000, se aplica el interés establecido previamente.
2. Si el saldo está entre 50000 y 500.000, se aplica 1.1 veces el interés establecido previamente.
3. Si el saldo es mayor a 500.000, se aplica 1.5 veces el interés establecido previamente.

El código correspondiente quedaría de la siguiente forma:

```
class C_Cuenta {
    // Variables miembro
    private:
        double Saldo;     // Saldo Actual de la cuenta
        double Interes;   // Interés calculado hasta el momento, anual,
                          //   en tanto por ciento %
    public:
        //Constructor
        C_Cuenta(double unSaldo=0.0, double unInteres=4.0)
        {
            SetSaldo(unSaldo);
            SetInteres(unInteres);
        }
        // Acciones Básicas
        inline double GetSaldo()
            { return Saldo; }
        inline double GetInteres()
            { return Interes; }
        inline void SetSaldo(double unSaldo)
            { Saldo = unSaldo; }
        inline void SetInteres(double unInteres)
            { Interes = unInteres; }
        void Ingreso(double unaCantidad)
            { SetSaldo( GetSaldo() + unaCantidad ); }
        virtual void AbonaInteresMensual()
        {
            SetSaldo( GetSaldo() * ( 1.0 + GetInteres() / 12.0 / 100.0) );
        }
        // etc...
};
```

```
class C_CuentaJoven : public C_Cuenta {
    public:
        C_CuentaJoven(double unSaldo=0.0, double unInteres=2.0,
             double unLimite = 50.0E3) :
                  C_Cuenta(unSaldo, unInteres)
        {
            Limite = unLimite;
        }
        virtual void AbonaInteresMensual()
        {
            if (GetSaldo() > Limite)
                SetSaldo( GetSaldo() * (1.0 + GetInteres() / 12.0 / 100.0) );
            else
                SetSaldo( GetSaldo() );
        }
    private:

        double Limite;
};

class C_CuentaEmpresarial : public C_Cuenta {
    public:
        C_CuentaEmpresarial(double unSaldo=0.0, double unInteres=4.0)
            : C_Cuenta(unSaldo, unInteres)
        {
            CantMin[0] = 50.0e3;
            CantMin[1] = 500.0e3;
        }
        virtual void AbonaInteresMensual()
        {
            SetSaldo( GetSaldo() * (1.0 + GetInteres() * CalculaFactor() / 12.0 / 100.0 ));
        }
        double CalculaFactor()
        {
            if (GetSaldo() < CantMin[0])
                return 1.0;
            else if (GetSaldo() < CantMin[1])
                return 1.1;
            else return 1.5;
        }
    private:

        double CantMin[2];
};
```

La idea central del ***polimorfismo*** es la de ***poder llamar a funciones distintas aunque tengan el mismo nombre, según la clase a la que pertenece el objeto al que se aplican***. Esto es imposible utilizando ***nombres de objetos***: siempre se aplica la función miembro de la clase correspondiente al nombre del objeto, y esto se decide en tiempo de compilación.

Sin embargo, ***utilizando punteros puede conseguirse el objetivo buscado***. Recuérdese que un ***puntero a la clase base*** puede contener ***direcciones de objetos*** de cualquiera de las ***clases derivadas***. En principio, el tipo de puntero determina también la función que es llamada, pero ***si se utilizan funciones virtuales es el tipo de objeto el que apunta el puntero lo que determina la función que se llama***. Esta es la esencia del ***polimorfismo***.

Implementación de las funciones virtuales

Una ***función virtual*** puede ser llamada como una función convencional, es decir, utilizando ***vinculación estática***. En este caso no se están aprovechando las características de las ***funciones virtuales***, pero el programa puede funcionar correctamente. A continuación se presenta un ejemplo de este tipo de implementación que no es recomendable usar, ya que utilizando una función convencional se ganaría en eficiencia:

```
Clase_1 Objeto_1;              // Se definen un objeto de una clase
Clase_1 *Puntero_1;            //   y un puntero que puede apuntarlo
float   variable;

Puntero_1 = &Objeto_1;
variable = Objeto_1.Funcion_1( );     // Utilización de vinculación estática
variable = Puntero_1->Funcion_1( );   //   con funciones virtuales. Absurdo
```

En el ejemplo anterior en las dos asignaciones a *variable*, las funciones que se van a utilizar se determinan en ***tiempo de compilación***.

A continuación se presenta un ejemplo de utilización correcta de las ***funciones virtuales***:

```
Clase_Base      Objeto_Base;
Clase_Derivada  Objeto_Derivado;
Clase_Base      *Puntero_Base_1;
Clase_Base      *Puntero_Base_2;
float           variable;

…
Puntero_Base_1 = &Objeto_Base;      // El puntero a la clase base puede
                                    //   apuntar a un objeto de la clase base
Puntero_Base_2 = &Objeto_Derivado;  //   o a un objeto de la clase derivada
…
variable = Puntero_Base_2->Funcion_1( );    // Utilización correcta
                                            //   de una función virtual
```

En este nuevo ejemplo se utiliza ***vinculación dinámica***, ya que el ***Puntero_Base_2*** puede apuntar a un ***objeto de la clase base*** o a un ***objeto de cualquiera de las clases derivadas*** en el momento de la asignación a ***variable***, en la última línea del ejemplo. Por eso, es necesariamente en tiempo de ejecución cuando el programa decide cuál es la ***Funcion_1*** concreta que tiene que utilizar.

Esa ***Funcion_1*** será la definida para la clase del ***Objeto_Derivado*** si está definida, o la de la ***clase base más próxima*** en el orden jerárquico que tenga definida esa ***Funcion_1***.

Funciones virtuales puras

Habitualmente las ***funciones virtuales*** de la clase base de la jerarquía no se utilizan porque en la mayoría de los casos ***no se declaran objetos de esa clase***, y/o porque todas las clases derivadas tienen su propia definición de la ***función virtual***. Sin embargo, incluso en el caso de que la ***función virtual*** de la clase base no vaya a ser utilizada, debe declararse.

De todos modos, ***si la función no va a ser utilizada no es necesario definirla***, y es suficiente con ***declararla como función virtual pura***. Una función virtual pura se declara así:

```
virtual funcion_1( ) const=0;    //Función virtual pura
```

La única utilidad de esta declaración es la de posibilitar la ***definición de funciones virtuales en las clases derivadas***. De alguna manera se puede decir que la definición de una función como virtual pura hace necesaria la definición de esa función en las clases derivadas, a la vez que imposibilita su utilización con objetos de la clase base.

Al definir una función como ***virtual pura*** hay que tener en cuenta que:

No hace falta definir el código de esa función en la clase base.

No se pueden definir objetos de la clase base, ya que no se puede llamar a las ***funciones virtuales puras***.

Sin embargo, ***es posible definir punteros a la clase base, pues es a través de ellos como será posible manejar objetos de las clases derivadas***.

Clases abstractas

Se denomina ***clase abstracta*** a aquella que ***contiene una o más funciones virtuales puras***. El nombre proviene de que ***no puede existir ningún objeto de esa clase***. Si una clase derivada no redefine una ***función virtual pura***, la clase derivada la hereda como ***función virtual pura*** y se convierte también en ***clase abstracta***. Por el contrario, aquellas clases derivadas que redefinen todas las ***funciones virtuales puras*** de sus clases base reciben el nombre de ***clases derivadas concretas***, nomenclatura únicamente utilizada para diferenciarlas de las antes mencionadas.

Aparentemente puede parecer que carece de sentido definir una clase de la que no va a existir ningún objeto, pero se puede afirmar, sin miedo a equivocarse, que la ***abstracción*** es una herramienta imprescindible para un correcto diseño de la ***Programación Orientada a Objetos***.

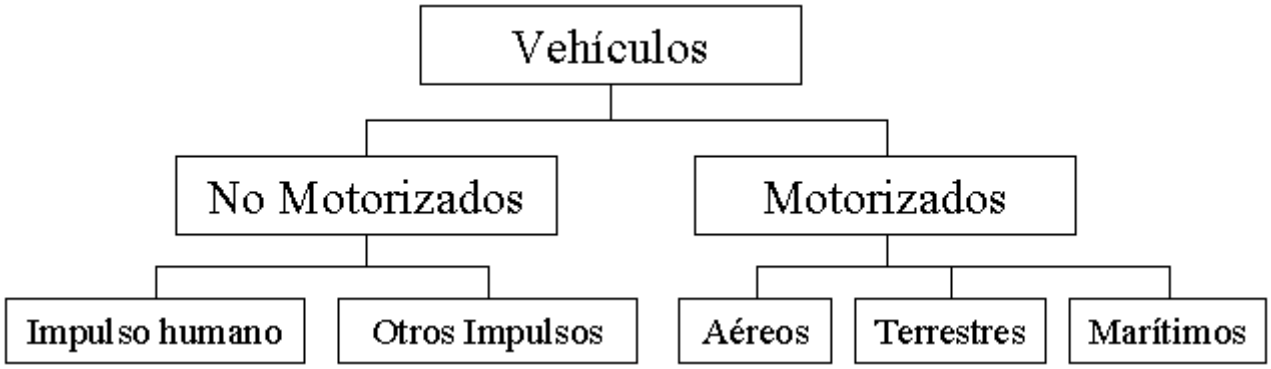


Figura 7. Clases base virtuales.

Está claro que la jerarquía que se presenta en la figura 7 no es suficiente, porque un avión y un helicóptero, o un patinete y una bicicleta, serían objetos de la misma clase. Pero lo que se pretende ilustrar es la necesidad de una clase ***vehículo*** que englobe las características comunes de todos ellos (peso, velocidad máxima, …), aunque no exista ningún objeto de esa clase, ya que cualquier vehículo en el que se piense, podrá definirse como un objeto de una clase derivada de la primera clase base.

Habitualmente las clases superiores de muchas jerarquías de clases son ***clases abstractas*** y las clases que heredan de ellas definen sus propias ***funciones virtuales***, convirtiéndose así en ***funciones concretas***.

Destructores virtuales

Como norma general, ***el constructor de la clase base se llama antes que el constructor de la clase derivada***. Con los ***destructores***, sin embargo, ***sucede al revés***: el destructor de la clase derivada se llama antes que el de la clase base.

Por esa razón, en el caso de que se borre, aplicando ***delete***, un ***puntero a un objeto de la clase base*** que apunte a un ***objeto de una clase derivada***, se llamará al ***destructor de la clase base***, en vez de al ***destructor de la clase derivada***, que sería lo adecuado. La solución a este problema consiste en ***declarar como virtual el destructor de la clase base***. Esto hace que automáticamente los ***destructores de las clases derivadas sean también virtuales***, a pesar de tener nombres distintos.

De este modo, al aplicar ***delete*** a un ***puntero de la clase base*** que puede apuntar a un objeto de ese tipo o a cualquier objeto de una clase derivada, ***se aplica el destructor adecuado*** en cada caso. Por esta razón es conveniente declarar ***un destructor virtual en todas las clases abstractas***, ya que aunque no sea necesario para esa clase, sí puede serlo para una clase que derive de ella.

Este problema no se presenta con los ***constructores*** y por eso no existe ningún tipo de constructor virtual o similar