4. POLIMORFISMO

En programación orientada a objetos se denomina polimorfismo a la capacidad que tienen los objetos de una clase de responder al mismo mensaje o evento en función de los parámetros utilizados durante su invocación. Un objeto polimórfico es una entidad que puede contener valores de diferentes tipos durante la ejecución del programa.

Dicho de otra forma, el polimorfismo consiste en conseguir que un objeto de una clase se comporte como un objeto de cualquiera de sus subclases, dependiendo de la forma de llamar a los métodos de dicha clase o subclases. Una forma de conseguir objetos polimórficos es mediante el uso de punteros a la superclase. De esta forma podemos tener dentro de una misma estructura (arrays, listas, pilas, colas, ...) objetos de distintas subclases, haciendo que el tipo base de dichas estructuras sea un puntero a la superclase. Otros lenguajes nos dan la posibilidad de crear objetos polimórficos con el operador new. La forma de utilizarlo, por ejemplo en java, sería:

```
Superclase sup = new (Subclase);
```

En la práctica esto quiere decir que un puntero a un tipo puede contener varios tipos diferentes, no solo el creado. De esta forma podemos tener un puntero a un objeto de la clase Trabajador, pero este puntero puede estar apuntando a un objeto subclase de la anterior como podría ser Márketing, Ventas o Recepcionistas (todas ellas deberían ser subclase de Trabajador).

El concepto de polimorfismo se puede aplicar tanto a funciones como a tipos de datos. Así nacen los conceptos de *funciones polimórficas* y *tipos polimórficos*. Las primeras son aquellas funciones que pueden evaluarse o ser aplicadas a diferentes tipos de datos de forma indistinta; los *tipos polimórficos*, por su parte, son aquellos tipos de datos que contienen al menos un elemento cuyo tipo no está especificado.

Clasificación

Se puede clasificar el polimorfismo en dos grandes clases:

- Polimorfismo dinámico (o polimorfismo paramétrico) es aquél en el que el código no incluye ningún tipo de especificación sobre el tipo de datos sobre el que se trabaja. Así, puede ser utilizado a todo tipo de datos compatible.

- Polimorfismo estático (o polimorfismo *ad hoc*) es aquél en el que los tipos a los que se aplica el polimorfismo deben ser explicitados y declarados uno por uno antes de poder ser utilizados.

El polimorfismo dinámico unido a la herencia es lo que en ocasiones se conoce como programación genérica.

También se clasifica en herencia por redefinición de métodos abstractos y por método sobrecargado. El segundo hace referencia al mismo método con diferentes parámetros.

Otra clasificación agrupa los polimorfismo en dos tipos: Ad-Hoc que incluye a su vez sobrecarga de operadores y coerción, *Universal* (inclusión o controlado por la herencia, paramétrico o genericidad).

Polimorfismo. Métodos virtuales

- El polimorfismo indica que una variable pasada o esperada puede adoptar múltiples formas. Cuando se habla de polimorfismo en programación orientada a objetos se suelen entender dos cosas:

  1. La primera se refiere a que se puede trabajar con un objeto de una clase sin importar de qué clase se trata. Es decir, se trabajará igual sea cual sea la clase a la que pertenece el objeto. Esto se consigue mediante jerarquías de clases y clases abstractas.

  2. La segunda suele referirse a la posibilidad de declarar métodos con el mismo nombre que pueden tener diferentes argumentos dentro de una misma clase.

- La capacidad de un programa de trabajar con más de un tipo de objeto se conoce con el nombre de polimorfismo
- Hasta ahora la herencia se ha utilizado solamente para heredar los miembros de una clase base, pero también existe la posibilidad de que un método de una clase derivada se llame como método de la clase base pero tenga un funcionamiento diferente.
- Por ejemplo, tomemos la clase disco_musica que hereda de la clase base disco:

```
class disco{
protected:
int capacidad;
char * fabricante; int num_serie; public:
disco (int c, char * f, int n){ capacidad = c; strcpy(fabricante, f); num_serie=n;}
void imprimir_capacidad(){
cout << capacidad; }
void imprimir_fabricante(){
cout << fabricante; }
void imprimir_num_serie(){
cout << num_serie; }
};

class disco_musica : public disco{
private:
char * tipo; //CD,vinilo, mini-disc, etc. cancion * lista_canciones;//Suponemos la
//existencia de una clase "cancion". public:
disco_musica (int c, char* f, int n, char* t) : disco (c,f,n){
strcpy(tipo,t); lista_canciones=NULL;}
};
```

- Podemos redefinir el método imprimir del ejemplo anterior de la siguiente forma:

```
void imprimir_fabricante(){
cout <<  "Este disco de musica ha
sido grabado por: "  << fabricante; }
//fabricante ha de ser protected!
```

- Si tenemos, por ejemplo,

```
void main(){
disco_musica d1(32,"EMI",423,"CD"); disco * ptr = & d1;
d1.imprimir_fabricante();
//Invoca al de la clase derivada.
ptr -> imprimir_fabricante();
//Invoca al de la clase base.
}
```

- Cuando se invoca un método de un objeto de la clase derivada mediante un puntero a un objeto de la clase base, se trata al objeto como si fuera de la clase base.

- Este comportamiento puede ser el deseado en ciertos casos, pero otras veces tal vez se desee que el comportamiento de la clase desaparezca por completo. Es decir, si el objeto de la clase disco_musica se trata como un objeto de la clase disco, se quiere seguir conservando el comportamiento de la clase disco_musica. Para ello se utiliza la palabra reservada **virtual**, si se quiere que el comportamiento de un método de la clase disco desaparezca se ha de declarar el método como **virtual.**

- Si queremos modificar la clase disco para que se pueda sobrecargar su comportamiento cuando se acceda a la clase disco_musicaa través de un puntero a disco pondríamos:

```
class disco{
    protected:
    int capacidad;
    char * fabricante; int num_serie; public:
    disco (int c, char * f, int n){ capacidad = c; strcpy(fabricante, f); num_serie=n;}
    void imprimir_capacidad(){
    cout << capacidad; }
    virtual void imprimir_fabricante(){
    cout << fabricante; }
    void imprimir_num_serie(){
    cout << num_serie; }
};
```

- Así, la clase disco ahora sobre el mismo extracto de programa anterior produce otro efecto:

```
void main(){
disco_musica d1(32,"EMI",423,"CD"); disco * ptr = & d1;
d1.imprimir_fabricante();
```

```
  //Invoca al de la clase derivada. ptr -> imprimir_fabricante();
  //Invoca al de la clase derivada.
}
```

- En este caso el objeto será siempre el de la clase derivada independientemente de cómo se pase el objeto a otras funciones. A esto es a lo que suele referir el término polimorfismo.
- Los métodos virtuales heredados son también virtuales en la clase derivada, por lo que si alguna clase hereda de la clase derivada el comportamiento será similar al indicado.

- Los únicos métodos que no pueden ser declarados como virtuales son los **constructores**, los métodos estáticos, y los operadores **new** y **delete**.

Clase virtual pura

Hay veces en las que no va a ser necesario crear objetos de la clase base, o simplemente no se desea que quien utilice la clase pueda crear objetos de la clase base. Para ello existen lo que suele llamarse en POO clases abstractas. Esta clase define el interfaz que debe tener una clase y todas las clases que heredan de ella. En C++ el concepto de clases abstractas se implementa mediante **funciones virtuales puras.** Estas funciones se declaran igual que cualquier otra función anteponiendo la palabra **virtual** y añadiendo al final de la declaración =0. Para estas funciones no se proporciona implementación. En C++ la clase abstracta debe tener uno o más métodos virtuales puros.

- Dada una clase abstracta, no se pueden crear objetos de esa clase base. Se pueden crear punteros que a objetos de la clase base abstracta que realmente apunten a objetos de la clase derivada.

Ejemplos/Ejercicios:

1) Dadas las clases:

```
class Persona {
    public:
      Persona(char *n) { strcpy(nombre, n); }
      char *VerNombre(char *n) { strcpy(n, nombre);
      return n;}
    protected:
      char nombre[30];
};

class Empleado : public Persona {
    public:
      Empleado(char *n) : Persona(n) {}
      char *VerNombre(char *n) { strcpy(n, "Emp: "); strcat(n, nombre); return n;}
};

class Estudiante : public Persona {
    public:
      Estudiante(char *n) : Persona(n) {}
      char *VerNombre(char *n) { strcpy(n, "Est: "); strcat(n, nombre); return n;}
};
```

¿Qué salida tiene el programa siguiente?:

```
int main()
{
        char n[40];
        Persona *Pepito = new
      Estudiante("Jose");
        Persona *Carlos = new
      Empleado("Carlos");
        cout << Carlos->VerNombre(n) <<
      endl;
        cout << Pepito->VerNombre(n) <<
      endl;
        delete Pepito; delete Carlos; system("pause");
        return 0;
}
```

Solución:

La salida es: Carlos
Jose
Presione cualquier tecla para continuar . . .

Vemos que se ejecuta la versión de la función
"VerNombre" que hemos definido para la clase base

2) Si modifcamos en el ejemplo anterior la declaración de la clase base "Persona“ como:

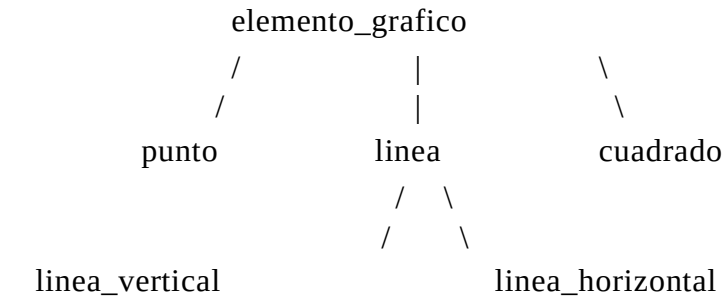
```
class Persona {
    public:
      Persona(char *n) { strcpy(nombre, n); }
      virtual char *VerNombre(char *n) { strcpy(n, nombre); return n;}
    protected:
      char nombre[30];
};
```

¿Qué salida tiene el programa anterior?

Solución:

La salida es como ésta:
Emp: Carlos
Est: Jose
Presione cualquier tecla para continuar . . .
Ahora, al llamar a "Pepito->VerNombre(n)" se invoca a
la función "VerNombre" de la clase "Estudiante", y al llamar a "Carlos->VerNombre(n)" se invoca a la función de la clase "Empleado"

3)Se quiere crear una jerarquía de elementos gráficos como en la siguiente figura:



Crear las clases necesarias para poder pintar los elementos gráficos de la jerarquía. Para completar la funcionalidad proporcionada por esta jerarquía, crear un array de objetos de la clase elementos_graficos y haga que se muestre por pantalla.

```
class elemento_grafico{
public:
virtual void pinta()const =0;
};

class punto : public elemento_grafico{
public:
void pinta()const{
cout << ".";}
};
class linea: public elemento_grafico{
protected:
    int longitud;
public:
    linea (int longitud):linea(longitud){}
};

    class linea_vertical: public linea {
    public:
    linea_vertical(int longitud):linea(longitud){}

    void pinta()const{
      for(int i=0; i<longitud; i++)
            cout << "|" << endl;}
};
```

```
class linea_horizontal: public linea {
  public:
  linea_horizontal(int longitud)
    :linea(longitud){}
  void pinta()const{
    for(int i=0; i<longitud; i++)
        cout << "-";}
};

class cuadrado: public elemento_grafico{
protected: int lado; public:
cuadrado(int lado):lado(lado){}
void pinta() const {
//pintar lado superior for(int i=0; i< lado; i++)
    cout << "-";
   cout << endl;
//pintar lados laterales for(int i=0; i<lado-2; i++){
     cout << "|";
     for(int j=0; j<lado-2;j++)
       cout << " ";
     cout << "|";
     cout << endl;
}
//pintar lado inferior
for (int i=0; i<lado; i++){
cout << "-"; }
}
};
```

Crear un array de punteros a elemento_grafico que referencien objetos de las clases derivadas distintos.

```
void main (){
    elemento_grafico * lista[4];

    lista[0]  =  new  punto;
    lista[1]  =  new  linea_horizontal(10);
    lista[2]  =  new  linea_vertical(5);
    lista[3]  =  new  cuadrado(6);
```

```
        for(int i=0; i<4; i++){ lista[i]->pinta(); cout << endl;
                }
        for (int i=0; i<4; i++)
                delete lista[i];
}
```

Este programa crea un array de punteros a elementos elemento_grafico, que debido a la herencia pueden apuntar a cualquier objeto de una clase que heredase directa o indirectamente de la clase elemento_grafico. A continuación, se llama al método pinta()particular de cada objeto, para conseguirlo el método pinta() de la clase elemento_grafico se ha declarado como **virtual.** Además, ya no se pueden crear objetos de la clase elemento_grafico, ya que la clase elemento_grafico es abstracta, por ser su método pinta() **virtual puro.**

Ejemplo de polimorfismo

En este ejemplo haremos uso del lenguaje C++ para mostrar el polimorfismo. También se hará uso de las funciones virtuales puras de este lenguaje, aunque para que el polimorfismo funcione no es necesario que las funciones sean virtuales puras, es decir, perfectamente el código de la clase "superior" (en nuestro caso Empleado) podría tener código

```
class Empleado {
protected:
    static const unsigned int SUELDO_BASE = 700; // Supuesto sueldo
    base para todos

public:
    /* OTROS MÉTODOS */
    virtual unsigned int sueldo() = 0;
};
class Director : public Empleado {
public:
    /* OTROS MÉTODOS */
    unsigned int sueldo() { return SUELDO_BASE*100; }
};
class Ventas : public Empleado {
private:
    unsigned int ventas_realizadas; // Contador de ventas realizadas
por el vendedor

public:
    /* OTROS MÉTODOS */
    unsigned int sueldo() { return SUELDO_BASE +
    ventas_realizadas*60;
}
};
```

```
class Mantenimiento : public Empleado {
public:
    /* OTROS MÉTODOS */
    unsigned int sueldo() { return SUELDO_BASE + 300; }
};

class Becario : public Empleado {
private:
    bool jornada_completa; // Indica si el becario trabaja a jornada
    completa

public:
    /* OTROS MÉTODOS */
    unsigned int sueldo() {
        if (jornada_completa) return SUELDO_BASE/2;
        else return SUELDO_BASE/4;
    }
};

/* AHORA HAREMOS USO DE LAS CLASES */
int main() {
    Empleado* e[4]; // Punteros a Empleado
    Director d;
    Ventas v; // Estas dos las declararemos como objetos normales en
    la
 pila

    e[0] = &d; // Asignamos a un puntero a Empleado la dirección de n
objeto del tipo Director
    e[1] = &v; // Lo mismo con Ventas
    e[2] = new Mantenimiento();
    e[3] = new Becario();

    unsigned int sueldo = 0;
    for (int i = 0; i < 4; ++i) sueldo += e[i]->sueldo();

    cout << "Este mes vamos a gastar " << sueldo << " dinero en
    sueldos" << endl;
}
```

Clases polimórficas

Sinopsis

En la introducción a la POO se señaló que uno de sus pilares básicos es el **polimorfismo**; la posibilidad de que un método tenga el mismo nombre y resulte en el mismo efecto básico pero esté implementado de forma distinta en clases distintas de una jerarquía. En este capítulo y siguiente describiremos los mecanismos que utiliza C++ para hacer esto posible. La explicación involucra varios conceptos estrechamente relacionados entre si: el de **función virtual**, el **enlazado dinámico** y el de **clase polimórfica** que analizamos en este capítulo.

Clases polimórifcas

Como se verá muy claramente en el ejemplo que sigue, las técnicas de POO necesitan a veces de clases con una interfaz muy genérica que deben servir a requerimientos variados (de ahí el nombre "polimórficas"), por lo que no es posible establecer "a priori" el comportamiento de algunos de sus métodos, que debe ser definido más tarde en las clases derivadas. Dicho en otras palabras: en ciertas súper-clases, denominadas **polimórficas**, solo es posible establecer el prototipo para algunas de sus funciones-miembro, las definiciones deben ser concretadas en las subclases, y generalmente son distintas para cada descendiente [1]. Estas funciones-miembro son denominadas **virtuales** y C++ establece para ellas un mecanismo de enlazado especial (dinámico) que posibilita que las diversas subclases de la familia ofrezcan diferentes comportamientos (diferencias que dan lugar precisamente al comportamiento polimórfico de la súper clase).

Para que una clase sea **polimórfica** debe declarar o heredar al menos una **función virtual** o **virtual pura** (aclararemos este concepto más adelante. En este último caso la clase polimórfica es además **abstracta**. Así pues, **clases polimórficas** son aquellas que tienen una interfaz idéntica, pero son implementadas para servir propósitos distintos en circunstancias distintas, lo que se consigue incluyendo en ellas unos métodos especiales (virtuales) que se definen de forma diferente en cada subclase. Así pues, el polimorfismo presupone la existencia de una súper-clase polimórfica de la que deriva más de una subclase.

Generalmente las clases polimórficas no se utilizan para instanciar objetos directamente [2], sino para derivar otras clases en las que se detallan los distintos comportamientos de las funciones virtuales. De esta segunda generación si se instancian ya objetos concretos. De esta forma, las clases polimórficas operan como interfaz para los objetos de las clases derivadas.

Es clásico el caso de la clase Poligono, para manejar Cuadrados, Círculos, Triángulos, etc.

```
class Poligono {
  Punto centro;
  Color col;
  ...
```

```
 public:
 point getCentro() { return centro; }
 void putCentro(Punto p) { centro = p; }
 void putColor(Color c) { col = c; }
 void mover(Punto a) { borrar(); centro = a; dibujar(); }
 virtual void borrar();
 virtual void dibujar();
 virtual void rotar(int);
 ...
};
```

Esta clase es polimórfica; Punto y Color son a su vez sendas clases auxiliares. Los métodos borrar, dibujar y rotar (que se declaran **virtuales**) tienen que ser definidos más tarde en las clases derivadas, cuando se sepa el tipo exacto de figura, ya que no se requiere el mismo procedimiento para rotar un triángulo que un círculo. De esta clase no se derivan objetos todavía.

Para definir una figura determinada, por ejemplo un círculo, lo primero es derivar una clase concreta de la clase genérica de las figuras (en el ejemplo la clase Circulo). En esta subclase, que en adelante representará a todos los círculos, debemos definir el comportamiento específico de los métodos que fueron declarados "virtuales" en la clase base.

```
class Circulo : public Poligono {   // la clase circulo deriva de Poligono
  int radio;        // propiedad nueva (no existe en la superclase)
  public:
    void borrar() { /* procesos para borrar un círculo */ }
    void dibujar() { /* procesos para dibujar un círculo */ }
    void rotar(int) {}             // en este caso una función nula!
    void putRadio(int r) { radio = r; }
};
```

El último paso será instanciar un objeto concreto (un círculo determinado). Por ejemplo:

```
Circulo circ1;
Punto centr1 = {1, 5};
circ1.putCentro(centr1);
circ1.putColor( rojo );
circ1.putRadio( 25 );
circ1.dibujar();
```

Veremos que, además de la herencia y las funciones virtuales, hay otro concepto íntimamente ligado con la cuestión: las **funciones dinámicas**, aunque esta es una particularidad de C++Builder y como tal no soportado por el Estándar.

Funciones virtuales

Sinopsis:

Las **funciones virtuales** permiten que clases derivadas de una misma base (clases hermanas) puedan tener diferentes versiones de un método. Se utiliza la palabra-clave **virtual** para avisar al compilador que un método será polimórfico y que en las clases derivadas existen distintas definiciones del mismo. En respuesta, el "Linker" utiliza para ella una técnica especial, **enlazado retrasado**. La declaración de **virtual** en un método de una clase, implica que esta es **polimórfica**, y que probablemente no se utilizará directamente para instanciar objetos, sino como super-clase de una jerarquía.

Observe que esta posibilidad, que un mismo método puede exhibir distintos comportamientos en los descendientes de una base común, es precisamente lo que posibilita y define el polimorfismo. En estos casos se dice que las descendientes de la función virtual **solapan** o **sobrecontrolan** ("Override") la versión de la superclase, pero esta versión de la superclase puede no existir en absoluto. Es probable que en ella solo exista una declaración del tipo: vitual void foo(); sin que exista una definición de la misma. Para que pueda existir la declaración de un método sin que exista la definición correspondiente, el método debe ser **virtual puro** (un tipo particular dentro de los virtuales).

Utilizaremos la siguiente terminología: "función sobrecontrolada" o "solapada", para referirnos a la versión en la superclase y "función sobrecontroladora" o "que solapa" para referirnos a la nueva versión en la clase derivada. Cualquier referencia al **sobrecontrol** o **solapamiento** ("Overriding") indica que se está utilizando el mecanismo C++ de las funciones virtuales. Parecido pero no idéntico al de **sobrecarga**; conviene no confundir ambos conceptos. Más adelante intentamos aclarar sus diferencias.

Sintaxis

Para declarar que un método de una clase base es **virtual**, su prototipo se declara como siempre, pero anteponiendo la palabra-clave **virtual** [3], que indica al compilador algo así como: "Será definido más tarde en una clase derivada". Ejemplo:

```
virtual void dibujar();
```

Cuando se aplica a métodos de clase [6], el especificador **virtual** debe ser utilizado en la declaración, pero **no** en la definición si esta se realiza offline (fuera del cuerpo de la clase). Ejemplo:

```
class CL {
  ...
  virtual void func1();
  virtual void func2();
  virtual void func3() { ... }    // Ok. definición inline
};
virtual void CL::func1() { ... }  // Error!!
void CL::func2() { ... }          // Ok. definición offline
```

Descripción

Al tratar del enlazado se indicaron las causas que justifican la existencia de este tipo de funciones en los lenguajes orientados a objeto. Para ayudarnos a comprender el problema que se pretende resolver con este tipo de funciones, exponemos una continuación del ejemplo de la clase Poligono al que nos referimos al tratar de las **clases polimórficas**:

```
class Color { public: int R, G, B; };   // clase auxiliar
class Punto { public: float x, y; };    // clase auxiliar

class Poligono {        // clase-base (polimórfica)
  protected:            // interfaz para las clases derivadas
  Punto centro;
  Color col;
  ...
  public:               // interfaz para los usuarios de poligonos
  virtual void dibujar() const;
  virtual void rotar(int grados);
  ...
};

class Circulo : public Poligono {       // Un tipo de polígono
  protected:
  int radio;
  ...
  public:
  void dibujar() const;
  void rotar(int) { }
  ...
};

class Triangulo : public Poligono {     // Otro tipo de pológono
  protected:
  Punto a, b, c;
  ...
  public:
  void dibujar() const;
  void rotar(int);
  ...
};
```

Lo que se pretende con este tipo de jerarquía es que los usuarios de subclases manejen los distintos polígonos a través de su interfaz pública (sus **miembros públicos**), mientras que los implementadores (creadores de los diversos tipos de polígonos que puedan existir en la aplicación), compartan los aspectos representados por los **miembros protegidos**.

Los **miembros protegidos** son utilizados también para aquellos aspectos que son dependientes de la implementación. Por ejemplo, aunque la propiedad color será compartida por todas las clases de polígonos, posiblemente su definición dependa de aspectos concretos de la implementación, es decir, del concepto de "color" del sistema operativo, que probablemente estará en definiciones en ficheros de cabecera.

Los miembros en la superclase (polimórfica) representan las partes que son comunes a todos los miembros (a todos los polígonos) pero en la mayoría de los casos no es sencillo decidir cuales serán

estos miembros (propiedades o métodos) compartidos por todas las clases derivadas. Por ejemplo, aunque se puede definir un centro para cualquier polígono, mantener su valor puede ser una molestia innecesaria en la mayoría de los polígonos y sobre todo en los triángulos, mientras es imprescindible en los círculos y muy útil en el resto de polígonos equiláteros.

Ejemplo-1

```
#include <iostream>
using namespace std;

class B {               // L.4:  Clase-base
  public: virtual int fun(int x) {return x * x;}   // L.5 virtual
};
class D1 : public B {     // Clase derivada-1
  public:  int fun (int x) {return x + 10;}        // L.8 virtual
};
class D2 : public B {     // Clase derivada-2
  public:  int fun (int x) {return x + 15;}        // L.11 virtual
};

int main(void) {          // =========
  B b; D1 d1; D2 d2;      // M.1
  cout << "El valor es: " << b.fun(10)     << endl;  // M.2:
  cout << "El valor es: " << d1.fun(10)    << endl;  // M.3:
  cout << "El valor es: " << d2.fun(10)    << endl;  // M.4:
  cout << "El valor es: " << d1.B::fun(10) << endl;  // M.5:
  cout << "El valor es: " << d2.B::fun(10) << endl;  // M.6:
  return 0;
}
```

Salida:

```
El valor es: 100
El valor es: 20
El valor es: 25
El valor es: 100
El valor es: 100
```

Comentario

Definimos una clase-base B, en la que definimos una función virtual fun; a continuación derivamos de ella dos subclases D1 y D2, en las que definimos sendas versiones de fun que solapan la versión existente en la clase-base.

Observe que según las reglas §4.1 y §4.2 indicadas más adelante, las versiones de fun en L.8 y L.11 son virtuales, a pesar de no tener el declarador **virtual** indicado explícitamente. Estas versiones son un caso de polimorfismo, y solapan a la versión definida en la clase-base.

La línea M.1 instancia tres objetos de las clases anteriormente definidas. En las líneas M.2 a M.4 comprobamos sus valores, que son los esperados (en cada caso se ha utilizado la versión de fun adecuada al objeto correspondiente). Observe que la elección de la función adecuada **no** se realiza por el análisis de los argumentos pasados a la función como en el caso de la sobrecarga (aquí los argumentos son iguales en todos los casos), sino por la "naturaleza" del objeto que invoca la función; esta es precisamente la característica distintiva del del polimorfismo.

Las líneas M.5 y M.6 sirven para recordarnos que a pesar del solapamiento, la versión de fun de la superclase sigue existiendo, y es accesible en los objetos de las clases derivadas utilizando el sobrecontrol adecuado.

> **Nota**: la utilización del operador **::** de acceso a ámbito anula el mecanismo de funciones virtuales, pero es aconsejable no usarlo en demasía, pues conduce a programas más difíciles de mantener. Como regla general, este tipo de calificación solo debe utilizarse para acceder a miembros del subobjeto heredado como es el caso del ejemplo.

Ejemplo-2

Hagamos ahora de abogados del diablo compilando el ejemplo anterior con la única modificación de que el método fun de **B** no sea **virtual**, sino un método normal. Es decir, la sentencia L.5, quedaría como:

```
public: int fun(int x) {return x * x;}    // L.5b no virtual!!
```

En este caso, comprobamos que el resultado coincide exactamente con el anterior, lo que nos induciría a preguntar ¿Para qué diablos sirven entonces las funciones virtuales?.

Ejemplo-3

La explicación podemos encontrala fácilmente mediante una pequeña modificación en el programa: en vez de acceder directamente a los miembros de los objetos utilizando el selector directo de miembro **.** en las sentencias de salida, realizaremos el acceso mediante punteros a las clases correspondientes. De estos punteros declararemos dos tipos: a la superclase y a las clases derivadas (por brevedad hemos suprimido las sentencias de comprobación M.5 y M.6).

El nuevo diseño sería el siguiente:

```
#include <iostream>
using namespace std;

class B {               // L.4: Clase-base
  public: virtual int fun(int x) {return x * x;}  // L.5 virtual
};
class D1 : public B {     // Clase derivada-1
  public: int fun (int x) {return x + 10;}        // L.8 virtual
};
class D2 : public B {     // Clase derivada-2
  public: int fun (int x) {return x + 15;}        // L.11 virtual
```

```
};

int main(void) {          // =========
  B b; D1 d1; D2 d2;      // M.1
  B* bp = &b; B* bp1 = &d1; B* bp2 = &d2;          // M.1a
  D1* d1p = &d1; D2* d2p = &d2;                    // M.1b
  cout << "El valor es: " << bp->fun(10) << endl;   // M.2a:
  cout << "El valor es: " << bp1->fun(10) << endl;  // M.3a:
  cout << "El valor es: " << bp2->fun(10) << endl;  // M.3b:
  cout << "El valor es: " << d1p->fun(10) << endl;  // M.4a:
  cout << "El valor es: " << d2p->fun(10) << endl;  // M.4b:
  return 0;
}
```

Salida:

```
El valor es: 100
El valor es: 20
El valor es: 25
El valor es: 20
El valor es: 25
```

Comentario

La sentencia M.1a define tres punteros a la clase-base. No obstante, dos de ellos se utilizan para señalar objetos de las sub-clases. Esto es típico de los punteros en jerarquías de clases. Precisamente se introdujo esta "relajación" en el control de tipos, para facilitar ciertas funcionalidades de las clases polimórficas.

La sentencia M.1b define sendos punteros a subclase, que en este caso si son aplicados a entidades de su mismo tipo.

Teniendo en cuenta las modificaciones efectuadas, y como no podía ser menos, las nuevas salidas son exactamente análogas a las del ejemplo inicial.

Ejemplo-4

Si suprimimos la declaración de **virtual** para la sentencia L.5 y volvemos a compilar el programa, se obtienen los resultados siguientes:

```
El valor es: 100
El valor es: 100
El valor es: 100
El valor es: 20
El valor es: 25
```

Observamos como, en ausencia de enlazado retrasado, las sentencias M.3a y M.3b, que acceden a métodos de objetos a través de punteros a la superclase, se refieren a los métodos heredados (que se definieron en la superclase). En este contexto pueden considerarse equivalentes los siguientes pares de expresiones:

```
bp1->fun(10)  ==  d1.B::fun(10)
bp2->fun(10)  ==  d2.B::fun(10)
```

Hay ocasiones es que este comportamiento no interesa. Precisamente en las clases abstractas, en las que la definición de B::fun() no existe en absoluto, y expresiones como M.3a y M.3b conducirían a error si fun() no fuese declarada virtual pura en **B**.

Ejemplo-5

Presentamos una variación muy interesante del primer ejemplo en el que simplemente hemos eliminado la línea 11, de forma que no existe definición específica de fun en la subclase D2.

```
#include <iostream>
using namespace std;

class B {                // L.4:  Clase-base
  public:  virtual int fun(int x) {return x * x;}   // L.5 virtual
};
class D1 : public B {     // Clase derivada
  public:  int fun (int x) {return x + 10;}         // L.8 virtual
};
class D2 : public B { };  // Clase derivada

int main(void) {          // =========
  B b; D1 d1; D2 d2;      // M.1
  cout << "El valor es: " << b.fun(10)     << endl;  // M.2:
  cout << "El valor es: " << d1.fun(10)    << endl;  // M.3:
  cout << "El valor es: " << d2.fun(10)    << endl;  // M.4:
  cout << "El valor es: " << d1.B::fun(10) << endl;  // M.5:
  cout << "El valor es: " << d2.B::fun(10) << endl;  // M.6:
  return 0;
}
```

Salida:

```
El valor es: 100
El valor es: 20
El valor es: 100
El valor es: 100
El valor es: 100
```

Comentario

El objeto d2 no dispone ahora de una definición específica de la función virtual fun, por lo que cualquier invocación a la misma supone utilizar la versión heredada de la superclase. En este caso las invocaciones en M.4 y M.6 utilizan la misma (y única) versión de dicha función.

No confundir el mecanismo de las **funciones virtuales** con el de **sobrecarga** y **ocultación**. Sea el caso siguiente:

```
class Base {
  public:
  void fun (int);    // No virtual!!
  ...
};
class Derivada : public Base {
  public:
  void fun (int);    // Oculta a Base::fun
  void fun (char);   // versión sobrecargada de la anterior
  ...
};
```

Aquí pueden declararse las funciones void Base::fun(int) y void Derivada::fun(int); incluso sin ser virtuales. En este caso, se dice que void Derivada::fun(int) oculta cualquier otra versión de fun(int) que exista en cualquiera de sus ancestros. Además, si la clase Derivada define otras versiones de fun(), es decir, existen versiones de Derivada::fun() con diferentes definiciones, entonces se dice de estas últimas son **versiones sobrecargadas** de Derivada::fun(). Por supuesto, estas versiones sobrecargadas deberán seguir las reglas correspondientes.

Ejemplo-6

Para ilustrar el mecanismo de ocultación en un ejemplo ejecutable, utilizaremos una pequeña variación del ejemplo anterior (Ejemplo-3):

```
#include <iostream>
using namespace std;

class B {               // L.4: Clase-base
  public: virtual int fun(int x) {return x * x;}    // L.5 virtual
};
class D1 : public B {    // Clase derivada-1
  public: int fun (int x, int y) {return x + 10;}   // L.8 NO virtual
};
class D2 : public B {    // Clase derivada-2
  public: int fun (int x) {return x + 15;}          // L.11 virtual
};

int main(void) {         // =========
  B b; D1 d1; D2 d2;     // M.1
  B* bp = &b; B* bp1 = &d1; B* bp2 = &d2;           // M.1a
  D1* d1p = &d1; D2* d2p = &d2;     // M.1b
  cout << "El valor es: " << bp->fun(10) << endl;   // M.2a:
  cout << "El valor es: " << bp1->fun(10) << endl;  // M.3a:
  cout << "El valor es: " << bp2->fun(10) << endl;  // M.3b:
  cout << "El valor es: " << d1p->fun(10) << endl;  // M.4a:
  cout << "El valor es: " << d2p->fun(10) << endl;  // M.4b:
  return 0;
}
```

En este caso nos hemos limitado a añadir un segundo argumento a la definición de D1::fun de la clase derivada-1 (L8). Como consecuencia, la nueva función **no** es virtual, ya que no cumple con las condiciones exigidas. El resultado es que en las instancias de D1, la definición B::fun de la superclase queda ocultada por la nueva definición, con la consecuencia de que la sentencia M.4a, que antes de la modificación funcionaba correctamente, produce ahora un error de compilación porque los argumentos actuales (un entero) no concuerdan con los argumentos formales esperados por la función (dos enteros).

Además el compilador nos advierte de la ocultación mediante una advertecia; en Borland C++: 'D1::fun(int,int)' hides virtual function 'B::fun(int)' [5].

Cuando se declare una función como **virtual** tenga en mente que:

- Solo pueden ser métodos (funciones-miembro).
- No pueden ser declaradas **friend** de otras clases.
- No pueden ser miembros estáticos
- Los constructores **no** pueden ser declarados virtuales
- Los destructores sí pueden ser virtuales.

Como hemos comprobado en el Ejemplo-5, las funciones virtuales no necesitan ser redefinidas en todas las clases derivadas. Puede existir una definición en la clase base y todas, o algunas de las subclases, pueden llamar a la función-base.

Para redefinir una función virtual en una clase derivada, las declaraciones en la clase base y en la derivada deben coincidir en cuanto a número y tipo de los parámetros. Excepcionalmente pueden diferir en el tipo devuelto; este caso es discutido más adelante.

Una función virtual redefinida, que **solapa** la función de la superclase, sigue siendo virtual y **no** necesita el especificador **virtual** en su declaración en la subclase (caso de las declaraciones de L.8 y L.11 en el ejemplo anterior). La propiedad **virtual** es heredada automáticamente por las funciones de la subclase. Aunque si la subclase va a ser derivada de nuevo, entonces **sí** es necesario el especificador.

Ejemplo:

```
class B {       // Superclase
   public:
   virtual int func();
};
...
class D1 : public B {     // Derivada
  public:
  int fun ();             // virtual por defecto
```

```
};
class D2 : public B {     // Derivada
  public:
  virtual int fun ();     // virtual explícita
};
class D1a : public D1 {   // Derivada
  public:
  int fun ();             // No virtual!!
};
class D2a : public D2 {   // Derivada
  public:
  int fun ();             // Ok Virtual!!
  int fun (char);         // No virtual!!
};
```

Como puede verse, de la simple inspección de las dos últimas líneas, no es posible deducir que el método fun de la clase **D2a** es un método virtual. Esto representa en ocasiones un problema y puede ser motivo de confusión, ya que es muy frecuente que las definiciones de las superclases se encuentren en ficheros de cabecera, y el programador que utiliza tales superclases para derivar versiones específicas debe consultar dichos ficheros. Sobre todo, porque como se indicó en el epígrafe anterior, la declaración en la subclase debe coincidir en cuanto a número y tipo de los parámetros. En caso contrario se trataría de una nueva definición, con lo que estamos ante un caso de sobrecarga y se ignora el mecanismo de enlazado retrasado [8].

Invocación de funciones virtuales

Lo que realmente caracteriza a las funciones virtuales es la forma muy especial que utiliza el compilador para invocarlas; forma que es posible gracias a su tipo de enlazado (dinámico). Por lo demás, el mecanismo externo (la sintaxis utilizada) es exactamente igual que la del resto de funciones miembro. He aquí un resumen de esta sintaxis:

```
class CB {              // Clase-base
   public:
   void fun1(){...}     // definición-10
   void virtual fun2(){...}    // definición-20
   ...
};
class D1 : public CB {  // Derivada-1
   public:
   void fun1() {...}    // definición-11
   void fun2() {...}    // definición-12
};
class D2 : public CB {  // Derivada-2
   public:
   void fun1() {...}    // definición-21
   void fun2() {...}    // definición-22
};
...
```

```
CB obj; D1 obj1;  D2 obj2;     // se instancian objetos de las clases
...
obj.fun1();            // invoca definición-10
obj.fun2();            // invoca definición-20
obj1.fun1();           // invoca definición-11
obj1.fun2();           // invoca definición-12
obj1.CB::fun1();       // invoca definición-10
obj1.CB::fun2();       // invoca definición-20
obj2.fun1();           // invoca definición-21
obj2.fun2();           // invoca definición-22
obj2.CB::fun1();       // invoca definición-10
obj2.CB::fun2();       // invoca definición-20
```

Observe que la dierencia entre las invocaciones obj.fun1() y obj1.CB::fun1(), estriba en que la misma función se ejecuta sobre distinto juego de variables. Por contra, en obj.fun1() y obj.fun2(), funciones distintas se ejecutan sobre el mismo juego de variables.

```
CB* ptr = &obj;        // se definen punteros a los objetos
D1* ptr1= &obj1;
D2* ptr2= &obj2;
...
ptr->fun1();           // invoca definición-10
ptr1->fun1();          // invoca definición-11
ptr2->fun1();          // invoca definición-21
```

Cuando desde un objeto se invoca un método (virtual o no) utilizando el nombre del objeto mediante los operadores de acceso a miembros, **directo .** o **indirecto ->**), se invoca el código de la función correspondiente a la clase de la que se instancia el objeto. Es decir, el código invocado solo depende del tipo de objeto (y de la sintaxis de la invocación). Esta información puede conocerse en tiempo de compilación, en cuyo caso se utilizaría **enlazado estático**. En otros casos este dato solo es conocido en tiempo de ejecución, por lo que debería emplearse **enlazado dinámico**.

Recuerde que cualquiera que sea el mecanismo para referenciar al objeto que invoca a la función (operador de acceso directo -1- o indirecto -2-):

```
obj.fun1()      // -1-
ptr->fun1()     // -2-
```

la función conoce cual es el objeto que la invoca -que juego de variables debe utilizar- a través del argumento implícito **this**.

Invocación en jerarquías de clases

Sea **B** es una clase base, y otra **D** derivada públicamente de **B**. Cada una contiene una función virtual **vf**, entonces si **vf** es invocada por un objeto de **D**, la llamada que se realiza es **D::vf**(), incluso cuando el acceso se realiza vía un puntero o referencia a la superclase **B**. Ejemplo:

```
#include <iostream>
using namespace std;
```

```
class C {               // Clase-base
  public: virtual int get() {return 10;}
};
class D : public C {    // Derivada
  public: virtual int get() {return 100;}
};

int main(void) {        // =========
  D d;
  C* cptr = &d;
  C& cref = d;
  cout << d.get()     << endl;
  cout << cptr->get() << endl;
  cout << cref.get()  << endl;
  return 0;
}
```

Salida:

```
100
100
100
```

Resulta muy oportuno aquí volver al caso mostrado al tratar el uso de punteros en jerarquías de clases, donde nos encontrábamos con una "sorpresa" al acceder al método de un objeto a través de un puntero a su superclase (repase el ejemplo original para situarse en la problemática que allí se describe).

Como ya anunciábamos en el referido ejemplo, basta una pequeña modificación en la definición de la función f en la superclase **B** para evitar el problema. En este caso basta la definición de dicha función como **virtual**. Observe la diferencia de salidas en ambos casos y como el nuevo diseño es en lo demás exactamente igual al original.

```
#include <iostream.h>

class B {             // Superclase (raíz)
  public: virtual int f(int i) { cout << "Funcion-Superclase "; return i; }
};
class D : public B {  // Subclase (derivada)
  public: int f(int i) { cout << "Funcion-Subclase "; return i+1; }
};

int main() {          // ==========
  D d;               // instancia de subclase
  D* dptr = &d;       // puntero-a-subclase señalando objeto
  B* bptr = dptr;     // puntero-a-superclase señalando objeto de subclase

  cout << "d.f(1)     ";
  cout << d.f(1) << endl;
  cout << "dptr->f(1) ";
```

```
  cout << dptr->f(1) << endl;  // acceso mediante puntero a subclase
  cout << "bptr->f(1) ";       // acceso mediante puntero a superclase
  cout << bptr->f(1) << endl;
}
```

Salida:

```
d.f(1)     Funcion-Subclase 2
dptr->f(1) Funcion-Subclase 2
bptr->f(1) Funcion-Subclase 2
```

Tabla de funciones virtuales

Las funciones virtuales pagan un tributo por su versatilidad. Cada objeto de la clase derivada tiene que incluir un puntero (**vfptr**) a una tabla de direcciones de funciones virtuales, conocida como **vtable** [2]. La utilización de dicho puntero permite seleccionar desde el objeto, la función correspondiente en tiempo de ejecución. Como resultado, el mecanismo de invocación de estas funciones (enlazado dinámico) es mucho menos eficiente que el de los métodos normales (enlazado estático), por lo que debe reservarse su utilización a los casos estrictamente necesarios.

Es fácil poner en evidencia la existencia del puntero **vfptr** mediante un sencillo experimento [7] que utiliza el operador **sizeof** :

```
struct S1 {
  int n;
  double get() { return n; } // método auxiliar normal
};

struct S2 {
  int n;
  virtual double get() { return n; } // método virtual
};

...

size_t tamS1 = sizeof(S1);  // -> 4
size_t tamS2 = sizeof(S2);  // -> 8
```

Comentario

El resultado del tamaño de ambos tipos es respectivamente 4 y 8 en cualquiera de los compiladores comprobados: Borland C++ 5.5 y gcc-g++ 3.4.2-20040916-1 para Windows. La diferencia de 4 Bytes obtenida corresponde precisamente a la presencia del mencionado puntero oculto.

> **Nota**: Tenga en cuenta que, en ambos casos, el tamaño resultante puede venir enmascarado por fenómenos de alineamiento interno. De forma que los tamaños respectivos y sus diferencias, generalmente no coincidirán con los valores teóricos que cabría esperar.

Función virtual pura

En ocasiones se lleva al extremo el concepto "virtual" en la declaración de una súper clase ("esta función será redefinida más tarde en las clases derivadas"), por lo que en ella solo existe una declaración de la función, relegándose las distintas definiciones a las clases derivadas. Entonces se dice que esta función es **virtual pura**. Esta circunstancia hay que advertirla al compilador; es una forma de decirle que la falta de definición no es un olvido por nuestra parte (de lo contrario el compilador nos señala que se nos ha olvidado la definición); esto se hace igualando a cero la declaración de la función [1]:

```
virtual int funct1(void);      // Declara función virtual
virtual int funct2(void) = 0;  // Declara función virtual pura
```

Como hemos señalado, la existencia de una función **virtual** basta para que la clase que estamos definiendo sea **polimórfica**. Si además igualamos la función a cero, la estaremos declarando como función **virtual pura**, lo que automáticamente declara la clase como **abstracta**

Es muy frecuente que las funciones virtuales puras se declaren además con el calificador **const**, de forma que es usual encontrar expresiones del tipo:

```
virtual int funct2(void) const = 0;  // Declara función virtual pura y constante
```

Ejemplo

El ejemplo que sigue es una modificación del anterior, en el que declaramos la función fun de la clase-base B como virtual pura, con lo que podemos omitir su definición.

```
#include <iostream>
using namespace std;

class B {              // L.4:  Clase-base
  public:  virtual int fun(int x) = 0;       // L.5 virtual pura
};
class D1 : public B {  // Clase derivada
  public:  int fun (int x) {return x + 10;}  // L.8 virtual
};
class D2 : public B {  // Clase derivada
  public:  int fun (int x) {return x + 15;}  // L.11 virtual
};
int main(void) {       // =========
  D1 d1; D2 d2;        // M.1

  cout << "El valor es: " << d1.fun(10)    << endl;  // M.3:
  cout << "El valor es: " << d2.fun(10)    << endl;  // M.4:
  return 0;
}
```

Salida:

```
El valor es: 20
El valor es: 25
```

Comentario

El resto del programa se mantiene prácticamente igual al modelo anterior, con la salvedad de que ahora en M1 no podemos instanciar un objeto directamente de la superclase B; la razón es que la superclase está incompleta (falta la definición de fun). Si lo intentáramos, el compilador nos mostraría un error señalando que no se puede instanciar un objeto de la clase B y que dicha clase es **abstracta**; además las clases abstractas no son instanciables por definición.

En lo que respecta a las salidas, comprobamos que son los valores esperados.

Una declaración de función **no** puede ser al mismo tiempo una definición y una declaración de virtual pura. Ejemplo:

```
struct Est {
   virtual void f() { /* ... */ } = 0;    // Error!!
};
```

La forma legal de proporcionar una definición es:

```
struct Est {
  virtual void f(void) = 0;    // declara f virtual pura
};
virtual void Est::f(void) {    // definición posterior de f
   /* código de la función f  */
};
```

Un mundo de excepciones

Seguramente el lector que haya llegado hasta aquí experimente una cierta perplejidad (que comparto). En el párrafo §7 decimos que se utiliza el recurso de declarar una función virtual pura para poder omitir la definición, y a continuación, en el párrafo §7.2 exponemos la forma legal de proporcionarla... El lector puede comprobar que esta aparente contradicción es asumida por el compilador sin protestas. En efecto, considere el ejemplo siguiente en el que modificamos el anterior añadiendo una definición a la función virtual pura fun.

```
#include <iostream>
using namespace std;

class B {               // L.4:  Clase-base
  public:  virtual int fun(int x) = 0;       // L.5 virtual pura
};
int B::fun(int x) {return x * x}            // L.7 definición de fun

class D1 : public B {   // Clase derivada
  public:  int fun (int x) {return x + 10;}  // L.9 virtual
};

class D2 : public B {   // Clase derivada
  public:  int fun (int x) {return x + 15;}  // L.12 virtual
};
```

```
int main(void) {        // =========
  D1 d1; D2 d2;         // M.1

  cout << "El valor es: " << d1.fun(10)    << endl;  // M.3:
  cout << "El valor es: " << d2.fun(10)    << endl;  // M.4:
  cout << "El valor es: " << d1.B::fun(10) << endl;  // M.5:
  cout << "El valor es: " << d2.B::fun(10) << endl;  // M.6:
  return 0;
}
```

Salida:

```
El valor es: 20
El valor es: 25
El valor es: 100
El valor es: 100
```

Comentario

En M1 seguimos sin poder instanciar un objeto de la superclase B (porque es **abstracta**). En cambio, ahora podemos insertar las líneas M5 y M6 (que ya utilizamos en la primera versión del ejemplo). La razón es que ahora invocamos las versiones de fun (que sí está definida) heredadas de la superclase en los objetos d1 y d2.

Tipos devueltos por las funciones virtuales

Generalmente, cuando se redefine una función virtual no se puede cambiarse el tipo de valor devuelto. Para redefinir una función virtual en alguna clase derivada, la nueva función debe coincidir exactamente en número y tipo con los parámetros de la declaración inicial (la "firma" -Signature- de ambas funciones deben ser iguales). Si no coinciden en esto, el compilador C++ considera que se trata de funciones diferentes (un caso de sobrecarga) y se ignora el mecanismo de funciones virtuales.

> **Nota**: para prevenir que puedan producirse errores inadvertidos, el compilador C++ GNU dispone de la opción **-Woverloaded-virtual**, que produce un mensaje de aviso, si se redefine un método declarado previamente **virtual** en una clase antecesora, y no se cumple la condición de igualdad de firmas.

No obstante lo anterior, hay casos en que las funciones virtuales redefinidas en clases derivadas devuelven un tipo diferente del de la función virtual de la clase base. Esto es posible solo cuando se dan **simultáneamente** las dos condiciones siguientes [4]:

- La función virtual sobrecontrolada devuelve un puntero o referencia a clase base [1].
- La nueva versión devuelve un puntero o referencia a la clase derivada [2].

§8.1 Ejemplo

```
struct X {};           // clase base.
struct Y : X {};       // clase derivada (:public X por defecto).
struct B {             // clase base.
  virtual void vf1();  // L.4:
  virtual void vf2();
  virtual void vf3();
  void f();
  virtual X* pf();     /* L.8: [1] devuelve puntero a clase base,
                          esta función virtual puede ser sobrecontrolada */
};
class D : public B {    // clase derivada
  public:
  virtual void vf1();  // L.12: Especificador virtual, legal pero redundante
  void vf2(int);       /* L.13: No virtual, oculta B::vf2()
                          dado que usa otros argumentos */
// char vf3();         // L.14: Ilegal! cambia solo el tipo devuelto!
  void f();           // L.15: privativa de D (no virtual)
  Y*   pf();          /* L.16: [2] función sobrecontrolante; difiere solo
                         en el tipo devuelto. Devuelve puntero a subclase */
};
void extf() {
  D d;                // d objeto de la clase D (instancia)
  B* bp = &d;         /* L.20: Conversión estándar D* a B*
                         Inicializa bp con la tabla de funciones del objeto d.
                         Si no existe entrada para una función en dicha tabla,
                         utiliza la función de la tabla de la clase B */
  bp–>vf1();          // invoca D::vf1
  bp–>vf2();          // invoca B::vf2 (D::vf2 tiene diferentes argumentos)
  bp–>f();            // invoca B::f (not virtual)
  X* xptr = bp–>pf();  /* invoca D::pf() y convierte el resultado
                          en un puntero a X */
  D* dptr = &d;
  Y* yptr = dptr–>pf(); /* inicializa yptr; este puntero invocará a D::pf()
                           No se realiza ninguna conversión */
}
```

L.12: La versión D::vf1 es **virtual** automáticamente, dado que devuelve lo mismo y tiene los mismos parámetros que su homónima en la superclase B. El especificador **virtual** puede ser utilizado en estos casos pero no es estrictamente necesario a no ser que se vayan a derivar nuevas subclases de la clase derivada.

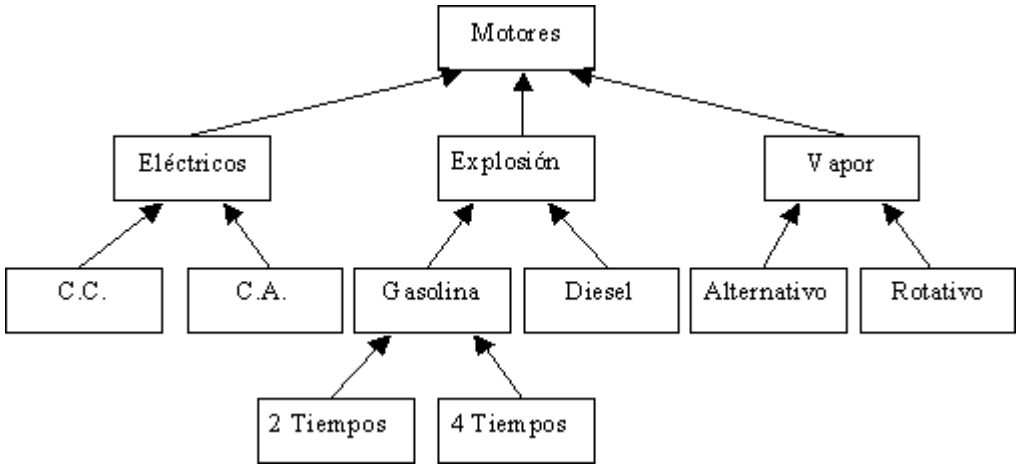
Clases abstractas

Sinopsis

La abstracción es un recurso de la mente (quizás el más característico de nuestra pretendida superioridad respecto del mundo animal). Por su parte, los lenguajes de programación permiten expresar la solución de un problema de forma comprensible simultáneamente por la máquina y el humano. Constituyen un puente entre la abstracción de la mente y una serie de instrucciones ejecutables por un dispositivo electrónico. En consecuencia, la capacidad de abstracción es una característica deseable de los lenguajes artificiales, pues cuanto mayor sea, mayor será su aproximación al lado humano. Es decir, con la imagen existente en la mente del programador. En este sentido, la introducción de las **clases** en los lenguajes orientados a objetos ha representado un importante avance respecto de la programación tradicional y dentro de ellas, las denominadas **clases abstractas** son las que representan el mayor grado de abstracción.

De hecho, las **clases abstractas** presentan un nivel de "abstracción" tan elevado que no sirven para instanciar objetos de ellas. Representan los escalones más elevados de algunas jerarquías de clases y solo sirven para derivar otras clases, en las que se van implementando detalles y concreciones, hasta que finalmente presentan un nivel de definición suficiente que permita instanciar objetos concretos. Se suelen utilizar en aquellos casos en que se quiere que una serie de clases mantengan una cierta característica o interfaz común. Por esta razón a veces se dice de ellas que son pura interfaz.

Resulta evidente en el ejemplo de la figura que los diversos tipos de motores tienen características diferentes. Realmente tienen poco en común un motor eléctrico de corriente alterna y una turbina de vapor. Sin embargo, la construcción de una jerarquía en la que todos motores desciendan de un ancestro común, la clase abstracta "Motores", presenta la ventaja de unificar la interfaz. Aunque evidentemente su definición será tan "abstracta", que no pueda ser utilizada para instanciar directamente ningún tipo de motor. El creador del lenguaje dice de ellas que soportan la noción de un concepto general del que solo pueden utilizarse variantes más concretas [2].



Clases abstractas

Una **clase abstracta** es la que tiene al menos una **función virtual pura** (como hemos visto, una **función virtual** es especificada como "pura" haciéndola igual a cero.

Nota: recordemos que las clases que tienen al menos una función virtual (o virtual pura) se denominan **clases polimórficas**. Resulta por tanto, que todas las clases abstractas son también polimórficas, pero no necesariamente a la inversa.

Reglas de uso:

- Una clase abstracta solo puede ser usada como clase base para otras clases, pero **no** puede ser instanciada para crear un objeto 1.
- Una clase abstracta **no** puede ser utilizada como argumento o como retorno de una función 2.
- Si puede declararse punteros-a-clase abstracta 3 [1].
- Se permiten referencias-a-clase abstracta, suponiendo que el objeto temporal no es necesario en la inicialización 4.

Ejemplo

```
class Figura {       // clase abstracta (CA)
  point centro;
    ...
  public:
  getcentro() { return center; }
  mover(point p) { centro = p; dibujar(); }
  virtual void rotar(int) = 0;   // función virtual pura
  virtual void dibujar() = 0;    // función virtual pura
  virtual void brillo() = 0;     // función virtual pura
    ...
};
...
Figura x;           // ERROR: intento de instanciar una CA.1
Figura* sptr;       // Ok: puntero a CA. 3
Figura f();         // ERROR: función NO puede devolver tipo CA. 2
int g(Figura s);    // ERROR: CA NO puede ser argumento de función 2
Figura& h(Figura&);  // Ok: devuelve tipo "referencia-a-CA" 4
int h(Figura&);      // Ok: "referencia-a-CA" si puede ser argumento 4
```

Suponiendo que **A** sea una clase abstracta y que **D** sea una clase derivada inmediata de ella, cada función virtual pura **fvp** de **A**, para la que **D** no aporte una definición, se convierte en función virtual pura para **D**. En cuyo caso, **D** resulta ser también una clase abstracta.

Por ejemplo, suponiendo la clase Figura definida previamente:

```
class Circulo : public Figura { // Circulo deriva de una C.A.
  int radio;        // privado por defecto
  public:
```

```
  void rotar(int);   // convierte rotar en función no virtual
};
```

En esta clase, el método heredado Circulo::dibujar() es una función virtual pura. Sin embargo, Circulo::rotar() no lo es (suponemos que definición se efectúa off-line). En consecuencia, Circulo es también una clase abstracta. En cambio, si hacemos:

```
class Circulo : public Figura { // Circulo deriva de una C.A.
  int radio;
  public:
  void rotar(int);   // convierte rotar en función no virtual pura
  void dibujar();    // convierte dibujar en función no virtual pura
};
```

la clase Circulo deja de ser abstracta.

Las funciones-miembro pueden ser llamadas desde el constructor de una **clase abstracta**, pero la llamada directa o indirecta de una función virtual pura desde tal constructor puede provocar un error en tiempo de ejecución. Sin embargo, son permitidas disposiciones como la siguiente:

```
class CA {        // clase abstracta
  public:
  virtual void foo() = 0;  // foo virtual pura
  CA() {         // constructor
    CA::foo();  // Ok.
};
...
void CA::foo() {  // definición en algún sitio
  ...
}
```

La razón es la ya señalada, de que la utilización del operador **::** de acceso a ámbito anula el mecanismo de funciones virtuales.