Técnico superior en desarrollo de videojuegos

Programación 1

Unidad 3

Polimorfismo y clases abstractas

Polimorfismo

- Definición
- Implementación del Polimorfismo
- Polimorfismo en C++ - Ejemplo de Funciones Virtuales
- Polimorfismo - funcionamiento
- Clases Abstractas Funciones virtuales puras. Ejemplos en C++
- Casting (conversión descendente downcasting)
- Destructores Virtuales
- Resumen

Polimorfismo

En un sentido literal, *polimorfismo* significa la cualidad de tener más de una forma. En el contexto de POO, el polimorfismo se refiere al hecho de que una misma operación puede tener diferente comportamiento en diferentes objetos

Polimorfismo

- En otras palabras, diferentes objetos reaccionan al mismo mensaje de modo diferente.
- Por ejemplo, consideremos la operación sumar. En un lenguaje de programación el operador **“+”** representa la suma de dos números (x+y) de diferentes tipos: enteros, coma flotante.
- Además se puede definir la operación de sumar dos cadenas: *concatenación,* mediante el operador suma.
- Por lo tanto el mensaje es el mismo **“+”**sumar y el resultado varía según el tipo: entero o cadena

Polimorfismo

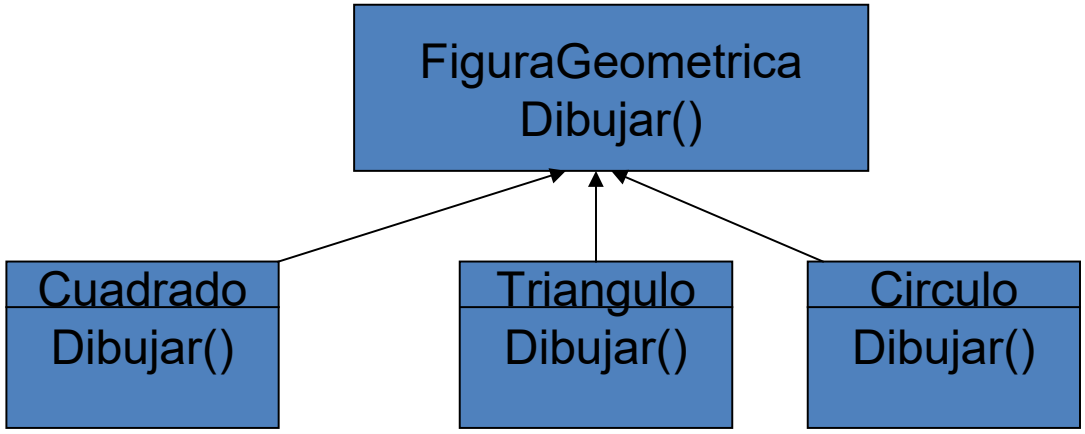
- Permite que un objeto pueda ser visto de diferentes formas.
- Permite que objetos de diferentes clases sean vistos de la misma forma.
- C++ permite el *polimorfismo;* la habilidad que objetos de diferentes clases relacionadas por la herencia respondan de manera diferente al mismo mensaje (es decir, a una llamada de una función miembro). El mismo mensaje enviado a muchos tipos diferentes de objetos toma "muchas formas"; de aquí el término polimorfismo. El mensaje enviado no necesita saber a que clase pertenece el objeto.

Polimorfismo

- Por ejemplo, si la clase **Rectángulo** deriva de **FiguraGeométrica**, entonces un **objeto Rectángulo** *es* una versión más específica de un objeto **FiguraGeométrica.**
  (Rectangulo **Es Un** FiguraGeométrica)
  Una operación (por ejemplo, el cálculo del perímetro o el dibujo del mismo) que puede realizarse sobre un objeto **FiguraGeométrica** también puede realizarse sobre un objeto **Rectángulo**.

Polimorfismo

- De modo similar, supongamos un número de figuras geométricas que responden todas al mensaje dibujar.
- Cada objeto reacciona a este mensaje mostrando su figura en una pantalla. Obviamente, el mecanismo real para visualizar los objetos difiere de una figura a otra, pero todas las figuras realizan esta tarea en respuesta al mismo mensaje(dibujar).
- Cuadrado, Triangulo y Circulo heredan de la clase base FiguraGeométrica, ya que cumplen con la sentencia "**Es Un**"



Polimorfismo

- Al polimorfismo se lo define como reutilización de interfaces.
- Se basa en enviar el mismo mensaje a objetos de diferentes clase y cada uno reacciona al mismo en forma diferente.
- Las clases de estos objetos heredan todos de una misma clase base.
- Este comportamiento se decide en tiempo de ejecución.
- Se denomina a este proceso **ligadura tardía** o **late binding ya que es en tiempo de ejecución y no en tiempo de compilación que se decide el llamado al método correspondiente**.

Implementación del Polimorfismo

Para implementarlo es necesario:

1. La utilización de punteros de clase base que reciben la dirección de un objeto de clase derivada de la misma(puntero polimorfico). Llamado **conversión ascendente(Upcasting).**

2. La existencia de funciones virtuales en clase base.

```
Base* p= new Derivada;
```

p es un puntero polimórfico

Polimorfismo

- Con las funciones virtuales y el polimorfismo, el programador puede manejar generalidades y dejar que el ambiente en tiempo de ejecución se ocupe de las particularidades. El programador puede manejar una amplia variedad de objetos para que se comporten de manera apropiada, sin siquiera tener que conocer los tipos (clases) de esos objetos.
- El polimorfismo promueve la extensibilidad: el software escrito para invocar un comportamiento polimórfico se escribe de manera independiente de los tipos de los objetos a los que se envían los mensajes. Entonces, los nuevos tipos de objetos que pueden responder a mensajes existentes pueden agregarse en un sistema, sin tener que modificar el sistema base. Con excepción del código nuevo que genera instancias de nuevos objetos, los programas no necesitan recompilarse.

Ejemplo de Funciones Virtuales


```
#include<iostream.h>
class Base {
public:
   void mostrar( ){cout <<"estoy en base";}
   virtual void mostrar2(){cout <<"estoy en base 2";}
};
class Derivada: public Base{
public:
   void mostrar(){ cout<<"estoy en derivada";}
   virtual void mostrar2(){cout <<"estoy en derivada 2";}
};
void main(){
Base *pb;    // puntero de clase Base

     pb=new Derivada;

     pb->mostrar();

     pb->mostrar2();

     if (pb) delete pb;
}
```

La palabra virtual indica la posibilidad de utilizar polimorfismo

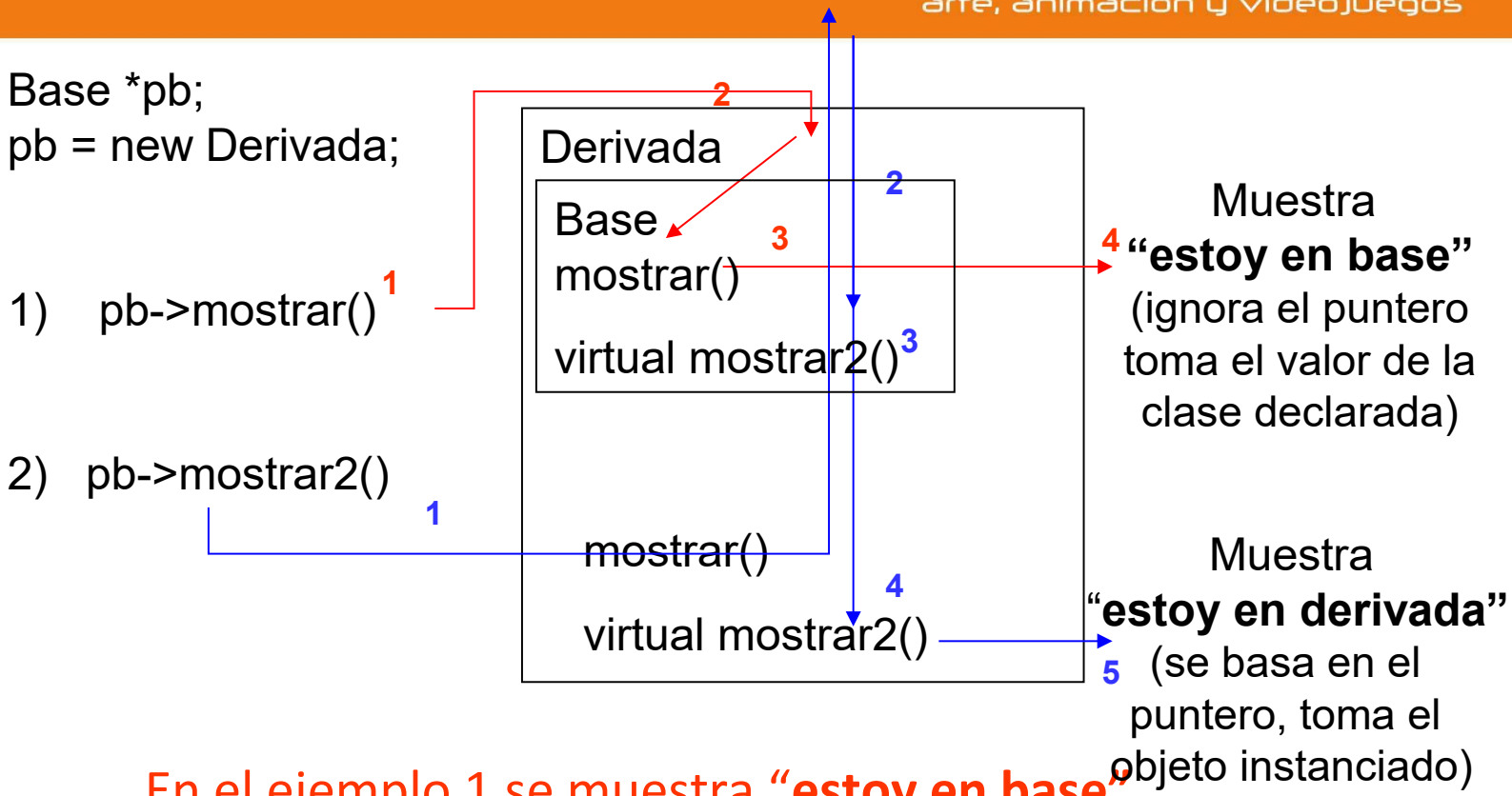
Observar que el método virtual debe llamarse igual y tener los mismos parámetros del mismo tipo en base y en derivada

Asigno la dirección de un objeto de derivada en uno de base(upcasting)

Ignora el contenido del puntero, por lo que se ejecuta el método mostrar de base(se basa en la clase declarada)

Se basa en el contenido del puntero, por lo que se ejecuta el método mostrar de derivada(se basa en el objeto instanciado)

Funcionamiento



En el ejemplo 1 se muestra "**estoy en base**" porque no es virtual el método

En el ejemplo 2 se muestra "**estoy en derivada**" porque es virtual y existe en derivada

Clases Abstractas
Funciones virtuales puras

Una clase abstracta define una interfaz para los diferentes miembros de una jerarquía de clases. La clase abstracta contiene funciones virtuales puras (solo declaradas) que se implementaran en las clases derivadas. Todas las funciones de la jerarquía pueden utilizar esta misma interfaz, a través del polimorfismo.

Aunque no podemos instanciar objetos de clases base abstractas, podemos declarar punteros hacia clases base abstractas. Tales punteros pueden entonces utilizarse para permitir manipulaciones polimórficas de los objetos de clases derivadas(conversión ascendente), cuando dichos objetos son instanciados a partir de clases concretas.

Clases Abstractas
Funciones virtuales puras

- Consideremos aplicaciones del polimorfismo y de las funciones virtuales. Un administrador de pantalla necesita desplegar muchos objetos de diferentes clases, incluso nuevos tipos de objetos que se agregarán al sistema, incluso después de que se haya escrito el administrador de pantalla. El sistema puede necesitar desplegar varios objetos (es decir, la clase base es Window) como ser:

un marco de ventana, etiquetas, cuadros de texto, botones, líneas y otras (cada clase de objetos se deriva de la clase base Window).

Clases Abstractas
Funciones virtuales puras

- Un administrador de pantalla utiliza punteros de la clase base (Window) para administrar todos los objetos a desplegar. Para dibujar cualquier objeto (independientemente del nivel en el que aparezca ese objeto en la jerarquía de herencia), el administrador utiliza un puntero de clase base hacia el objeto, y simplemente envía un mensaje **dibujar** hacia él.
- La función dibujar se declaró como virtual pura en la clase base Window y se ignoró en cada una de las clases derivadas. Cada objeto Window sabe cómo dibujarse a sí mismo ya que reescribió la función según su comportamiento.
- El administrador de pantalla no tiene que preocuparse por el tipo de cada objeto, o si el objeto es de un tipo que ha visto antes; simplemente le dice a cada objeto que se dibuje a sí mismo.

Polimorfismo en C++ Ejemplo de Funciones Virtuales Puras


```
#include<iostream.h>
// es una clase abstracta
class FiguraGeometrica {
public:
        virtual void dibujar()=0;
};
class Cuadrado: public FiguraGeometrica{
public:
        void dibujar( ){ cout<<“soy un cuadrado"<< endl;}
};
class Triangulo: public FiguraGeometrica{
public:
        void dibujar( ){ cout<<“soy un triangulo"<< endl;}
};
class Circulo: public FiguraGeometrica{
public:
        void dibujar( ){ cout<<“soy un circulo"<< endl;}
};
```

Cuando una Clase contiene **al menos un método = 0**, la clase se transforma en una **Clase Abstracta**
Y el método debe ser codificado obligatoriamente en la clase derivada

Polimorfismo en C++ Ejemplo de Funciones Virtuales Puras


```
void main(){
Cuadrado*  c=new Cuadrado(); // instancia objeto clase base
Triangulo*  t=new Triangulo(); // instancia objeto clase derivada
Circulo* ci=new Circulo(); // instancia objeto clase derivada
FiguraGeometrica *pb[3];
//arreglo de punteros de base
int i;
  // error de compilación
  // FiguraGeometrica b;
  clrscr();
  pb[0]=c;
  pb[1]=t;
  pb[2]=ci;
  for (i=0; i<3;i++)
            pb[i]->dibujar();
...
```

Una clase Abstracta **no** puede ser instanciada y para utilizarla se debe crear una clase Derivada y escribir el código del método que es virtual puro

Le asigna un puntero de clase derivada a uno de clase base
Esto lo permite el compilador ya que un objeto de derivada es un objeto de base (**upcasting**)

En tiempo de ejecución según el puntero elige a que función llamar.
Se denomina **Late Binding** o **ligadura tardía**.

Polimorfismo en C++ Ejemplo de Funciones Virtuales Puras


```
#define TAM 4
void main(){
Window *pb[TAM]; //arreglo de punteros de Window
for (i=0; i<TAM;i++)
      pb[i]=NULL;

pb[0]= new Marco; // objeto del tipo marco
pb[1]= new CuadroTexto;
pb[2]= new Etiqueta;
pb[3]= new Boton;

for (i=0; i<TAM;i++)
      if(pb[i]!=NULL)
            pb[i]->dibujar();
…

for (i=0; i<TAM;i++)
      if(pb[i]!=NULL)
        delete pb[i];
}
```

Se dibujan todos los elementos de la ventana porque el mensaje es el mismo **dibujar()** pero cada uno de los objetos reacciona diferente.

La elección de cada método dibujar se realiza en tiempo de ejecución (**late binding**)

Polimorfismo ejemplo


```
#include<iostream.h>
class Base {
public:
   void mostrar( ){cout <<"estoy en base"<<endl;}
   virtual void mostrar2(){cout <<"estoy en base";}
};

class Derivada: public Base{
public:
void mostrar( ){cout <<"estoy en derivada"<<endl;}
virtual void mostrar2(){cout <<"estoy en derivada";}
};

void main(){
Derivada *d = new Derivada;
Base *pb;
      pb = d;
      pb->mostrar();
      d->mostrar();
      if(d) delete d;
}
```

Algunas veces se define un **método no virtual** en una clase base y se ignora en una clase derivada.

Si se llama a dicha función miembro a través de un puntero de clase base hacia el objeto de clase derivada, se utiliza la versión de la clase base.

Si se llama a la función miembro a través de un puntero de clase derivada, se utiliza la versión de la clase derivada.

Éste ultimo es un comportamiento no polimórfico.

Polimorfismo - downcasting


```
#include<iostream.h>
class Base {
public:
   void mostrar( ){cout <<"estoy en base";}
   virtual void mostrar2(){cout <<"estoy en base";}
};
class Derivada: public base{
public:
void mostrar( ){cout <<"estoy en derivada";}
virtual void mostrar2(){cout <<"estoy en derivada";}
void mostrar3(){ cout <<"otro mensaje de derivada";}
};

void main(){
Base *pb = new Derivada;
      // pb->mostrar3(); error de compilación
     (  (Derivada *) pb )->mostrar3();
      if(d)delete d;
}
```

Algunas veces se necesita ejecutar un **método** de una clase **derivada** que no existe en la clase **base** con un **puntero de base**. Si se llama a dicha función miembro a través de un puntero de clase base dará un **error de compilación** ya que no encontrará el método en la clase base.
Para solucionar este problema se cambia el tipo de puntero a través de un **casting (conversión descendente o downcasting**) a la clase derivada como indica el

Se convierte el puntero de base en uno de derivada solamente para esta instrucción.

Ejemplo random


```
#define TAM 3
class FiguraGeometrica {
public: virtual void dibujar()=0; // es una clase abstracta
};
class Cuadrado: public FiguraGeometrica {
public: void dibujar(){ cout<<"estoy en cuadrado"<< endl;}
};
class Triangulo: public FiguraGeometrica {
public: void dibujar() { cout<<"estoy en triangulo"<< endl;}
};
class Circulo: public FiguraGeometrica {
public: void dibujar(){ cout<<"estoy en circulo"<< endl;}
};
void main(){
FiguraGeometrica* pb[TAM]={NULL,NULL,NULL};
      srand(time(0));
      for (int i=0;i<TAM;i++) {
          switch( rand()%TAM ) {
                case 0:pb[i] = new Cuadrado; break;
                case 1:pb[i] = new Triangulo; break;
                case 2:pb[i] = new Circulo;
      }
      }
      for (int i=0; i<TAM;i++) {
          if (pb[i]) pb[i]->dibujar( );
      }
      for (int i=0; i<TAM;i++)
          if (pb[i]) delete pb[i];
}
```

En este ejemplo observamos que cada vez que se ejecuta, sale por pantalla un mensaje diferente . Esto se debe a que el **random** crea objetos diferentes en cada corrida, pero gracias al polimorfismo el **mensaje** es el mismo sin interesar de que clase es cada objeto

Destructores Virtuales

- Cuando se utiliza el polimorfismo para procesar objetos asignados de una manera dinámica a una jerarquía de clase, puede ocurrir un problema. Si un objeto (con un destructor no virtual) se destruye explícitamente, aplicando el operador **delete** a un puntero de clase base hacia el objeto, se llama a la función destructora de clase base (que coincida con el tipo del puntero) sobre el objeto. Esto ocurre independiente del tipo del objeto al que se refiere el puntero de clase base, e independiente del hecho de que el destructor de cada clase tiene un nombre diferente.
- Existe una solución sencilla para este problema; declarar un destructor de clase base **virtual**. Esto hace que todos los destructores de clases derivadas sean virtuales, aunque no tengan el mismo nombre que el destructor de clase base. Ahora, si se destruye explícitamente a un objeto de la jerarquía, aplicando el operador **delete** a un puntero de clase base que apunta hacia un objeto de clase derivada(upcasting), **se llama al destructor de la clase apropiada**. Recuerde, cuando se destruye un objeto de clase derivada, la parte de la clase base correspondiente al objeto de la clase derivada también se destruye; el destructor de clase base siempre se ejecuta después del destructor de clase derivada.

Destructores Virtuales

```
class Base{
public:
Base( ){ }
~Base( ){ cout<<“me voy de base”;}
};
class Derivada: public base{
public:
Derivada( ){ }
~Derivada( ){
      cout<<“me voy de derivada”;}
};
void main(){
Base *pb=new Derivada();
delete pb;
}
```

Nunca se ejecuta el destructor de derivada ya que el puntero es de Base(upcasting) y no es una función virtual. Se ignora el puntero de derivada

```
class Base{
public:
 Base( ){ }
virtual ~Base( ){ cout<<“me voy de base”;}
};
class Derivada: public base{
public:
  Derivada( ){ }
~Derivada( ){
      cout<<“me voy de derivada”;}
};
void main( ){
Base *pb=new Derivada();
delete pb;
}
```

La solución es colocar un destructor virtual en la clase base.
Entonces sí se ejecuta el destructor de la Derivada y luego el de Base

Polimorfismo - Resumen

- Con las funciones virtuales y el polimorfismo, se hace posible diseñar e implementar sistemas que sean más fácilmente extensibles. Los programas pueden escribirse para procesar objetos de clases que pueden no existir cuando el programa está en desarrollo.
- La programación polimórfica con funciones virtuales puede eliminar la necesidad del switch lógico (solución sin polimorfismo). El programador puede utilizar el mecanismo de una función virtual para desarrollar la lógica equivalente, con lo que se evitan los tipos de errores generalmente asociados con el switch lógico. El código que toma decisiones sobre los tipos de objetos y las representaciones indica un diseño de clase pobre.
- Si es necesario, las clases derivadas pueden proporcionar sus propias implementaciones de una función virtual de clase base, pero si no lo es, se utiliza la implementación de la clase base.

Polimorfismo - Resumen

- Existen muchas situaciones en las que es útil definir clases para las que el programador nunca intenta crear instancias de ningún objeto. Dichas clases se conocen como clases **abstractas**. Estas se utilizan sólo como clases base, por lo que normalmente nos referiremos a ellas como clases base abstractas(interfases). Ningún objeto de una clase abstracta puede instanciarse en un programa.
- Las clases cuyos objetos pueden instanciarse se conocen como clases concretas.
- Una clase se vuelve abstracta declarando una o más funciones virtuales como puras. Una función virtual pura es aquella que tiene un inicializador=0 en su declaración.
- Si una clase se deriva de una clase con una función virtual pura, sin suplir la definición de esa función virtual pura en la clase derivada, entonces esa función virtual permanece pura en la clase derivada. Como consecuencia, la clase derivada también es una clase abstracta.
- C++ permite el polimorfismo; la habilidad de los objetos de diferentes clases relacionadas por la herencia de responder de manera diferente a la misma llamada a la función miembro.

Polimorfismo - Resumen

- El polimorfismo se implementa a través de funciones virtuales.
- Cuando se hace una solicitud a través de un puntero de clase base para utilizar una función virtual, C++ elige la función correcta en la clase derivada asociada con el objeto.
- Por medio de las funciones virtuales y el polimorfismo, una llamada a una función miembro puede ocasionar diferentes acciones, de acuerdo con el tipo del objeto que recibe la llamada.
- Aunque no podemos instanciar objetos de clases base abstractas, podemos declarar punteros hacia ellas. Tales punteros pueden utilizarse para permitir manipulaciones polimórficas de objetos de clases derivadas, cuando dichos objetos se instancian a partir de clases concretas.
- Por lo general, nuevos tipos de clases se añaden a los sistemas. Las nuevas clases son alojadas por medio de la vinculación dinámica (también conocida como vinculación tardía). El tipo de un objeto no necesita conocerse en tiempo de compilación, para que una llamada a una función virtual se compile. En tiempo de ejecución, se hace que la llamada a una función virtual coincida con la función miembro del objeto que la recibe.

Polimorfismo - Resumen

- La vinculación dinámica permite a los fabricantes de software independientes distribuir software sin revelar secretos del propietario. Las distribuciones de software pueden consistir solamente en archivos de encabezado y en archivos de objetos. No es necesario revelar el código fuente. Los desarrolladores de software pueden entonces utilizar la herencia para derivar nuevas clases a partir de aquellas provistas por los fabricantes. El software que funciona con las clases de los fabricantes independientes de software continuara funcionando con las clases derivadas, y utilizara (a través de la vinculación dinámica) las funciones sustituidas provistas en estas clases.
- La vinculación dinámica requiere que, en tiempo de ejecución, la llamada a la función miembro virtual se dirija hacia la versión de la función virtual apropiada para la clase. Una tabla de funciones virtual llamada ***vtable*** se implementa como un arreglo que contiene puntero a las funciones. Cada clase con funciones virtuales tiene una ***vtable***. Para cada función virtual en la clase, la ***vtable*** tiene una entrada que contiene un puntero de función hacia la versión de la función virtual a utilizar para un objeto de esa clase. La función virtual a utilizar para una clase en particular podría ser la función definida en esa clase, o podría ser una función heredada directa o indirectamente desde una clase base más arriba en la jerarquía.

Polimorfismo - Resumen

- Cuando una clase base proporciona una función miembro virtual, las clases derivadas pueden pasar por alto a la función virtual, pero no tienen que hacerlo. Entonces, una clase derivada puede utilizar una versión de una clase base correspondiente a una función miembro, y esto se indicaría en la ***vtable***.
- Cada objeto de una clase con funciones virtuales contiene un puntero a la ***vtable*** para esa clase. El puntero de la función adecuada en la ***vtable*** se obtiene y se desreferencia para completar la llamada en tiempo de ejecución. Esta búsqueda en la ***vtable*** *y* la desreferencia de un puntero requieren una sobrecarga nominal en tiempo de ejecución, normalmente menor que el mejor código cliente escrito.
- Declarar el destructor de la clase como virtual, si la clase contiene funciones virtuales. Esto hace que todos los destructores de clases derivadas sean virtuales, aunque no tengan el mismo nombre que el destructor de la clase base. Si un objeto de la jerarquía se destruye explícitamente, aplicando el operador delete a un puntero de clase base hacia un objeto de clase derivada, se llama al destructor de la clase apropiada.