Clases abstractas, Métodos Virtuales y Polimorfismo en C++

Ing. Pablo Novara (zaskar_84@yahoo.com.ar)
Facultad de Ingeniería y Ciencias Hídricas
Universidad Nacional del Litoral

Última revisión: 02/09/2014

En este documento se pretende explicar claramente y con ejemplos cortos el funcionamiento y la implementación del polimorfismo en C++, así como también introducir sus conceptos y definiciones esenciales. Sin embargo, para entender conceptos específicos de clases abstractas y métodos virtuales en C++ debemos tener presente algunas cuestiones básicas relacionadas principalmente a los mecanismos de herencia.

Para empezar se debe notar que **cuando se crea una instancia de una clase, se reserva memoria para sus atributos**. Los métodos no ocupan lugar en una instancia. Esto significa que si tenemos la clase Base:

```
class Base {
     float f;
     int i;
public:
     Base (float x=0, int y=0) { f=x; i=y; }
     float VerF() { return f; }
     float VerI() { return i; }
};
```

al crear una instancia:

```
Base b; // se crea una instancia
```

se reserva en memoria lugar para un flotante y un entero. El tamaño de la instancia es 8 bytes (4 del flotante + 4 del entero). Al crear una instancia, además de reservar la memoria, también estamos invocando al constructor de la clase, que podría, eventualmente, inicializar esa memoria con los valores que correspondan.

El código de los métodos está en el ejecutable y es el mismo para cualquier instancia (se puede imaginar como una función que recibe además de los parámetros declarados, un argumento más, implícito, que es el puntero this). Es decir, que los métodos no son datos que se guardan en la memoria con cada instancia, sino que son algoritmos que dicen cómo operar con los verdaderos datos, los atributos.

En segundo lugar, es necesario dejar en claro que **al crear un puntero a un objeto de tipo Base, no se está creando ninguna instancia, sino solamente una "flecha"** que eventualmente servirá para apuntar a una instancia. Hay dos formas de hacer que apunte efectivamente a una instancia: obteniendo la dirección de una existente con el operador &, o creando una nueva dinámicamente con el operador new:

```
Base *p; // se crea la flecha, pero no hay instancia
p = new Base(3.2,5); // se crea la instancia y se la apunta
```

En este caso, p apunta al comienzo de una instancia de la clase Base. Por ser un puntero de tipo Base, p espera encontrar en esa dirección un flotante y un entero. Al llamar a los métodos de la clase Base con el puntero p, se utilizarán los 8 bytes que comienzan donde apunta p como si fueran un flotante y un entero:

```
cout<<p->VerF(); // muestra 3.2
cout<<p->VerI(); // muestra 5
```

Supongamos que ahora tenemos también una clase Hija que hereda de la clase Base:

```
class Hija:public Base {
     double d;
public:
     Hija(float x, int y, double z) : Base(x,y) { d=z; }
     VerDouble();
};
```

Esto significa que ahora podemos declarar 2 tipos de objetos: objetos de tipo Hija y objetos de tipo Base. Los objetos de tipo Base tienen sus propios métodos y atributos. **Los objetos de tipo Hija, tienen todos los métodos y atributos de Base, y además agregan algunos propios.** Esto significa que cuando construimos una instancia de Hija estamos construyendo una primer parte que es igual a una instancia de Base (más aún, se utilizan los mismo constructores) y luego agregando los atributos propios de la clase hija.

En memoria, al crear una instancia de Hija, se organizan primero los datos heredados de la clase Base y luego los datos propios de la clase Hija. Esto quiere decir que una instancia de la clase hija ocupa 16 bytes: los primeros 8 corresponden al flotante y al entero que hereda de Base, los 8 siguientes al double d.

Notar que el constructor de la clase Hija recibe tres datos: dos de ellos se pasan al constructor de la clase Base para que inicialice f e i, y el tercero lo usa para luego inicializar d.

Debido a esta organización de la memoria, **si miramos solamente la primer parte de una instancia de la clase Hija se ve igual que una instancia de la clase Base**. Esto permite realizar dos operaciones importantes: asignar una instancia de tipo Hija a una de tipo Base, o apuntar a una instancia de tipo Hija con un puntero de tipo Base.

Al asignar a una instancia de Base una de Hija se copia la primer parte de la instancia de Hija en la memoria donde está la instancia de Base:

```
Base b(1.1,2); // instancia de Base
```

```
Hija h(0,0,0); // instancia de Hija
b=h; // asigna instancia de Hija en instancia de Base
cout<<b.VerI()<<b.VerF(); // muestra 00
```

Dicho de otra forma, dado que una instancia de Base solo tiene lugar para un flotante y un entero (8 bytes), es imposible copiar allí toda una instancia de Hija (16 bytes), pero como la primer mitad de la instancia de Hija es "compatible" (también tiene un flotante y un entero), se puede copiar esa mitad, y eso es lo que ocurre en C++[1].

Lo que realmente sucede es que C++ trata a la instancia de Hija como si fuera momentáneamente una instancia de Base (hay un casting implícito). El caso opuesto (asignar una instancia de Base a una instancia de Hija) no es correcto, dado que la clase Base no tiene todos los elementos necesarios para asignar en Hija, y por lo tanto no se puede ver a una instancia de Base como una instancia de Hija.

Al apuntar a una instancia de Hija con un puntero a Base se puede usar dicho puntero para operar con los métodos de Base sobre la parte de la instancia que fue heredada de Base:

```
Hija h(1.1,2,3.33);
Base *p=&h;
```

Esto quiere decir que **se puede llamar a VerF() y VerI()** con este puntero:

```
cout<<p->VerF(); // muestra 1.1
cout<<p->VerI(); // muestra 2
```

ya que estos métodos operan sobre una clase Base, y la primer parte de h (los primeros 8 bytes, donde apunta p) es precisamente igual a una instancia de la clase Base (hay un flotante y un entero).

Sin embargo, **no es posible llamar a VerD()** con el puntero p porque p es de tipo Base, y Base no tiene ningún método VerD(). Es decir, que **el tipo que usamos para declarar el puntero determina qué métodos se pueden llamar**, no importa a qué apunte luego.

De esta forma, si se quiere llamar un método a través de p (puntero a la clase Base), éste (el método) tiene que estar declarado sí o sí en la clase Base. Por ejemplo, si se tiene en ambas clases un método MostrarDatos para que muestre todos los atributos por pantalla:

```
class Base {
     float f;
     int i;
public:
     Base (float x=0, int y=0) { f=x; i=y; }
     void MostrarDatos() { cout<<f<<" "<<<i<<endl; }
};
class Hija:public Base {
     double d;
public:
     Hija(float x, int y, double z) : Base(x,y) { d=z; }
     void MostrarDatos() { cout<<f<<" "<<<i<<" "<<d<<endl; }
```

1 Si bien esto es posible, en general no es útil ni recomendable. No tiene sentido pensando en el principio de ocultación y en la propiedad de los atributos (en el sentido de quien es dueño y responsable), poder seccionar el objeto de esta forma.

```
};
```

el método de la clase Base muestra solo el flotante y el entero, y el método de la clase Hija muestra además del flotante y el entero, también el double.

La pregunta obligada es: **¿Cómo saber a cuál de los dos se llama?. La respuesta es simple: basta con identificar el tipo de objeto o el tipo de puntero con el cual se lo llama**:

```
Base b(1.1,2); // instancia de Base
Base *pb = &b; // puntero a Base
Hija h(3.3,4,5.55); // instancia de Hija
Hija *ph = &h; // puntero a Hija
b.MostrarDatos(); // llama al método de Base
pb->MostrarDatos(); // llama al método de Base
h.MostrarDatos(); // llama al método de Hija
ph->MostrarDatos(); // llama al método de Hija
```

Si el objeto o el puntero son de tipo Base, se llama al método de Base y se muestra el flotante y el entero pero no el doble. Si el objeto o el puntero son de tipo Hija se llama al método de Hija y se muestran los 3 valores. Esto es así sin importar si se ha copiado en la instancia de Base una instancia de Hija, o si el puntero a Base en realidad apunta a una instancia de Hija:

```
Hija h(3.3,4,5.55);
Base b;
b=h;
b.MostrarDatos() // se llama al método de Base
Base *p;
p=&h;
p->MostrarDatos() // se llama al método de Base
```

La palabra clave virtual permite alterar este comportamiento. Normalmente al compilador le basta con saber el tipo del puntero para saber a cual método llamar (y esto se resuelve en tiempo de compilación). Cuando se agrega la palabra virtual se le está avisando que esto podría no ser suficiente. **Si la clase Base tiene métodos virtuales, a la hora de llamar a uno de esos métodos con un puntero, se verificará si en verdad está apuntando a una instancia de Base, o si en realidad está apuntando a una instancia de Hija**, **y se invocará al método que se corresponda con el verdadero contenido de la memoria**, y no con el tipo de puntero (y esto se resuelve en tiempo de ejecución). Es decir, si modificamos la clase base agregando el calificativo virtual al método MostrarDatos:

```
class Base {
     float f;
     int i;
public:
     Base (float x=0, int y=0) { f=x; i=y; }
     virtual void MostrarDatos() { cout<<f<<" "<<<i<<endl; }
};
```

el siguiente código cambia su significado:

```
Hija h(1.1,2,3.33);
Base *pb = &h;
```

```
pb->MostrarDatos(); // Se llama al método de Hija
```

Ahora, al llamar a MostrarDatos, se muestran los 3 valores, porque el compilador reconoce que aunque el puntero es de tipo Base, en realidad apunta a una instancia de Hija.

Se debe notar que **este comportamiento sólo puede obtenerse mediante el uso de punteros**[2], ya que si se asignan objetos "reales" (instancias del objeto, no punteros) de distinto tipo, como se explicó anteriormente sólo se copian en realidad las partes comunes. Es decir, al declarar una instancia de tipo Base, reservamos memoria para sus atributos (8 bytes, un flotante y un entero) y al asignarle una instancia de Hija, solo se copia la primer parte de Hija (los primeros 8 bytes), pues no hay lugar donde copiar la segunda parte (el double d). Por esto, al invocar a un método con la instancia de Base, no hay duda de que sólo se puede llamar al método de Base, ya que no se tiene más información que esa, no se tiene la parte que añade Hija. En cambio, cuando se trata de punteros, sí, ya que el puntero sólo apunta al comienzo, y por esto puede apuntar a bloques de memoria de distintos tamaños sin problemas.

Notar además que al invocar desde un método de una clase a otro método de la misma clase, estamos en condiciones de utilizar polimorfismo, ya que la invocación se realiza implícitamente con el puntero this:

```
class ProcesaVector {
     int *v, n; // v es un arreglo dinámico de tamaño n
public:
     virtual void Leer(int &x) = 0;
     virtual void Mostrar(const int &x) = 0;
     void LeerTodos() { for(int i=0;i<n;i++) Leer(v[i]); }
     void MostrarTodos() { for(int i=0;i<n;i++) Mostrar(v[i]); }
     void Procesar() {/**el verdadero trabajo de esta clase**/}
};
class PVConsola : public ProcesaVector {
     void Leer(int &x) override { cin>>x; }
     void Mostrar(const int &x) override { cout<<x<<endl; }
};
int main() {
     PVConsola v(10);
     v.LeerTodos();
     v.Procesar();
     v.MostrarTodos();
```

En este ejemplo, la clase ProcesaVector toma n enteros uno por uno, luego los procesa, y finalmente los escribe. Esta clase no define de donde se leen y ni donde se escriben los datos, solo cómo se procesan. Por el contrario, la clase hija PVConsola define solo las acciones de lectura y escritura (en este caso utilizando la consola), mientras que heredan de forma transparente el procesamiento. De esta manera, esta jerarquía de clases separa en dos niveles bien diferenciados, grupos tareas igualmente diferenciados, por un lado el procesamiento de la información, por otro lado las operaciones de entrada salida. Se dice que la clase base define una interfaz para las operaciones de entrada/salida, que deberá implementar la clase hija. Más ejemplo de este uso se pueden encontrar en algunos entornos de diseño

2 Una alternativa válida es utilizar alias (por ejemplo, pasaje por referencia en argumentos de funciones), pues este mecanismo, a bajo nivel es equivalente al uso de punteros.

de interfaces gráficas, que generan automáticamente código orientado a objetos para las iterfases diseñadas de forma visual. Por ejemplo, en wxFormBuilder se diseña visualmente una ventana eligiendo qué controles tendrá, con qué propiedades, de qué forma se distribuirán en la misma etc. Esta herramienta luego genera el código fuente de una clase que representa la ventana diseñada. Para que el usuario pueda luego programar independientemente los eventos (por ejemplo programar cómo responderá la ventana ante un click en un botón de la misma) el código generado tiene métodos virtuales para los mismos. Así, el programador trabaja sobre una clase hija, dejando de lado y relativamente desacoplado (esto es, fácilmente reemplazable) el código de la parte visual.

Hasta este punto se ha mostrado que agregando virtual a un método de una clase, se obliga al compilador a "prestar atención" a qué apunta en realidad un puntero del tipo de esa clase. Hay que notar que esto genera un problema importante para el compilador. Si el método no es virtual, el método al que se llama cuando se ejecuta el programa es siempre el mismo (porque el tipo de variable es fijo), y por lo tanto el compilador puede determinarlo a la hora de generar el ejecutable sin necesidad de efectivamente ejecutarlo. En cambio, si el método es virtual, el método a llamar puede cambiar de una ejecución a otra:

```
int n;
cin>>n;
Base *p;
if (n==1)
     p=new Base(1.1,2);
else
     p=new Hija(3.3,4,5.55);
p->MostrarDatos();
```

En este ejemplo, si el usuario lo ejecuta e ingresa 1 se llama al método de Base, pero si lo ejecuta e ingresa 2 se llama al método de Hija. Por esto, el compilador no puede decidir al compilar, sino que el programa final lo tendrá que decidir en cada ejecución. Esto involucra una tarea adicional que debe hacer el programa cada vez que se ejecuta, que consiste en mantener la llamada "tabla de métodos virtuales". El mecanismo exacto no es fijo, y distintos compiladores pueden implementarlo de formas diferentes, pero en general el trabajo y la memoria extra no son significativos[3]. Esto quiere decir que la pérdida de eficiencia es imperceptible, y que la ganancia en la legibilidad y flexibilidad del código y del programa la justifican.

Volviendo a la declaración de los métodos virtuales. Hay ocasiones en qué se declara una clase base y una o más clases heredadas, y el programa utiliza tanto la clase base como la clase heredada (como los ejemplos vistos hasta ahora). Sin embargo, en otras situaciones, puede que el programa sólo utilice instancias de las clases hijas. En estos casos, la clase base sólo sirve para colocar el código común a todas las clases hijas y para declarar los métodos virtuales:

```
class Base {
public:
```

3 El mecanismo más usual consiste en colocar en cada clase que utilice métodos virtuales un puntero static a una tabla con punteros a métodos que dice cuales son las implementaciones que realmente debe utilizar. Hay entonces una tabla y un puntero por cada clase (no por cada objeto), y el costo de una llamada se incremente en un nivel de indirección, por lo que resulta despreciable (si no se considera el impacto sobre las optimizaciones de inlining).

```
        virtual void Saludar() {} // este método nunca se invoca
};
class HijaIngles : Public Base {
public:
        void Saludar() { cout<<”Hello World!”<<endl; }
};
class HijaEspaniol : Public Base {
public:
        void Saludar() { cout<<”Hola Mundo!”<<endl; }
};
int main() {
        cout<<”Elija el Idioma (1=Ingles, 2=Español):”;
        int n;
        cin>>n;
        Base *p;
        if (n==1) p=new HijaIngles();
        else p=new HijaEspaniol();
        p->Saludar();
        delete p;
        return 0;
}
```

En este ejemplo, nunca se crea una instancia de Base. Siempre se crean instancias de HijaIngles o HijaEspaniol. Base se usa solo para declarar el puntero. De esta forma, la diferencia entre un idioma y otro está en la creación de la instancia (el new dentro del if/else), pero a la hora de utilizar dicha instancia (invocar a Saludar) no hay diferencia. Para que el programa compile sin errores, el ejemplo debe implementar el método Saludar() de la clase Base, a pesar de que no se invoque y no haga nada (notar las llaves vacías {}). Si se omiten las llaves el compilador dirá que no puede terminar de enlazar el ejecutable porque no encuentra la implementación de ese método. Existe una alternativa, que consiste en avisarle al compilador que no busque esta implementación, ya que no existe. Para ello, se iguala el método a 0:

```
class Base {
public:
        virtual void Saludar() =0;
};
```

Se dirá entonces que el método Saludar de la clase Base, además de ser virtual, es virtual “puro”. Esto significa que no tiene implementación. Esto tiene dos consecuencias: no se puede instanciar la clase Base, y se debe implementar el método obligatoriamente en la clase Hija.

La primera significa que ahora definitivamente no podremos crear una instancia de la clase Base, porque esta clase está incompleta. Es decir, al igualar el método a 0 indicamos que no se va a implementar, por lo tanto a la clase “le falta algo”. Esto impide compilar un programa que declare una instancia de esa clase. Sin embargo, no impide declarar el puntero, ya que el puntero es solo una flecha, que se puede usar para apuntar a otra clase que sí esté completa. A una clase que nunca se instancia se la llama clase “abstracta”.

La segunda implica que para que una clase hija esté completa y pueda ser instanciada, debe implementar obligatoriamente el método virtual puro. Si el método fuera virtual pero no puro (tuviera una implementación en Base), la clase hija podría no implementarlo y utilizar el que heredó de Base. Si el método es virtual puro la implementación ya no es opcional, pues si la clase hija no lo implementa también estará incompleta (también será abstracta).

Finalmente, es conveniente analizar el caso en que se quiere utilizar polimorfismo con muchos objetos, ya que este es el caso más común. Ejemplos de esto se pueden encontrar en:

- Juegos: una clase Personaje genérica tiene métodos para moverse, atacar, etc. Luego el juego presenta diferentes tipos de personajes (arqueros, caballeros, princesas, magos, ogros, etc) que se mueven o atacan de forma diferente. Un ejemplo puede ser el ajedrez, donde cada jugador tiene diferentes tipos de Piezas (alfil, caballo, rey, reina, peón, torre) que se mueven de forma diferente.
- Interfaces visuales: Las bibliotecas para generar interfaces de ventanas suelen colocar dentro de una ventana instancias de una clase genérica, por ejemplo Componente. Luego, cada tipo de Componente (botón, cuadro de texto, imagen, lista desplegable, etc) se dibuja de forma diferente, reacciona a los clicks del ratón de forma diferente, etc.

En estos casos y muchos más, se requiere generar un arreglo de objetos que trabajen polimorficamente. Si para utilizar polimorfismo con una instancia, declaramos un puntero, para hacerlo con N instancias, debemos declarar un arreglo de N punteros:

```
Base *A[4]; // arreglo de 4 punteros
A[0]=new HijaIngles();
A[1]=new HijaEspañol();
A[2]=new HijaAleman();
A[3]=new HijaFrances();
for (int i=0:i<4;i++)
     A[i]->Saludar();
```

Notar que un arreglo estático de punteros se declara igual que un arreglo estático de objetos, pero agregando un asterisco(*). Al declarar un arreglo estático de 4 punteros, estamos reservando memoria para las 4 flechas, pero ninguna de ellas apunta a una verdadera instancia. Por eso, en el ejemplo, luego de declarar el arreglo se hace un new por cada elemento.

Si queremos que el arreglo sea dinámico (es decir, no fijar su tamaño en 4, sino tomar su dimensión de una variable), debemos declarar un puntero a puntero, y hacer un primer new para reservar memoria para los N punteros, y luego N news para crear las instancias. Imaginemos que tenemos que hacer un arreglo dinámico de flotantes, donde la cantidad la ingresa el usuario:

```
int N;
cin>>N;
float *A = new float[N];
…
delete [] A;
```

Para hacer un arreglo dinámico de punteros a Base hay que aplicar exactamente el mismo esquema, reemplazando el tipo float por el tipo Base*:

```
int N;
```

```
cin>>N;
Base* *A = new Base*[N];
```

de esta forma se obtiene la declaración de puntero a puntero (doble asterisco). Luego de reservar la memoria para los N punteros, hay que crear los N objetos:

```
for (int i=0;i<N;i++) {
     int lang;
     cin>>lang;
     if (lang==1) A[i]=new HijaIngles();
     else if (lang==2) A[i]=new HijaEspañol();
     else if (lang==3) A[i]=new HijaAleman();
     else if (lang==4) A[i]=new HijaFrances();
}
```

y cuando ya no se utilice, se debe liberar la memoria, tanto de las instancias, como del arreglo:

```
for (int i=0;i<N;i++)
     delete A[i]; // elimina cada instancia
delete [] A; // elimina el arreglo de punteros
```

Resumiendo:

- Para que podamos plantear la pregunta ¿a qué método llamar?, tiene que haber una relación de herencia donde tanto la clase hija como la clase base tengan exactamente el mismo método (esto es, mismo nombre y mismos argumentos).
- Para que exista la posibilidad de optar por varios métodos debemos trabajar con punteros, declarando un puntero de un tipo (base) y haciendo que apunte a una instancia de otro tipo (hija).
- Si el método no es virtual, el método que se llama se determina con el tipo del puntero en tiempo de compilación. Si el método sí es virtual, el método que se llama se determina con el tipo de instancia que hay guardada en la memoria en tiempo de ejecución.
- Si nunca se van a invocar algunos métodos virtuales de la clase base, se puede igualar a 0 dichos métodos, convirtiéndolos así en métodos virtuales puros, y generando una obligación para las clases heredadas.
- Cuando una clase tiene al menos un método virtual puro, ya no puede ser instanciada, y se dice que es abstracta. Sólo sirve como base para las clases hijas, y para declarar punteros.