Universidad Nacional de Rosario
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Escuela de Ingeniería Electrónica
Cátedra de Informática Aplicada

Programación Orientada a Objetos en C++

Clases y Objetos en C++

Introducción

Las variables de los tipos fundamentales de datos no son suficientes para modelar adecuadamente objetos del mundo real. Por ejemplo, no se puede modelar una caja mediante un `int`, pero si se definen las variables largo, ancho y alto para representar las dimensiones de una caja, se las puede juntar en una estructura de datos llamada `Caja`. A continuación es posible definir variables de este nuevo tipo de la misma manera que con variables de tipos básicos. Se pueden crear, manipular y destruir tantos objetos de tipo `Caja` como se quiera [4]. De esta manera puede verse cómo C++ incorpora la noción de **clase** del paradigma de Orientación a Objetos. La palabra `class` es la palabra claave para implementar este concepto.

Veamos cómo se puede definir una clase que representa cajas:

```
class Caja {
      double largo;
      double ancho;
      double alto;
};
```

Las variables que se definen como parte de la clase se llaman **datos miembro** de la clase.
Se puede hacer una declaración de una variable de esta clase, digamos `cajaGrande`, que representa una instancia de tipo `Caja` como la siguiente:

```
Caja cajaGrande;
```

Una vez que se ha definido la clase `Caja`, las declaraciones de variables de este tipo son estándares. Estas variables son instancias de la clase y se las llama **objetos**.

Clases

La definición de una clase es la especificación de un nuevo tipo de dato. Puede contener elementos que pueden tener variables tanto de los tipos básicos como de otros tipos definidos por el usuario. Pueden ser elementos simples o arreglos, punteros, arreglos de punteros, etc. Además una clase puede contener funciones que operan sobre los objetos de esa clase accediendo a sus elementos. De esta manera, una clase combina la definición de los datos que componen un objeto y los medios para manipularlos.

Los datos y funciones de una clase son llamados **miembros** de la clase. Las funciones miembro, a veces, también son llamadas métodos. A los datos miembro se los suele llamar campos.

Cuando se define una clase, no se define un dato, sino qué significa el nombre de la clase, en qué consiste un objeto de esa clase y qué operaciones pueden realizarse sobre los objetos de esa clase [4].

Definición de una clase:

Veamos nuevamente la clase mencionada anteriormente, la clase de las cajas. Se utilizó la palabra clave `class` de la siguiente manera:

```
class Caja {
      double largo;
      double ancho;
      double alto;
};
```

El nombre de la clase aparece siguiendo la palabra clave `class`, y los tres datos miembro se declaran entre llaves. La definición de la clase completa debe terminar con punto y coma. Los nombres de todos los miembros de la clase son locales a la clase, por lo que se puede utilizar los mismos nombres en cualquier parte del programa sin causar inconvenientes.

Control de acceso en una clase:

Se puede especificar que los miembros de una clase sean `public`, `private` o `protected`. Por defecto los miembros de una clase son `private`.

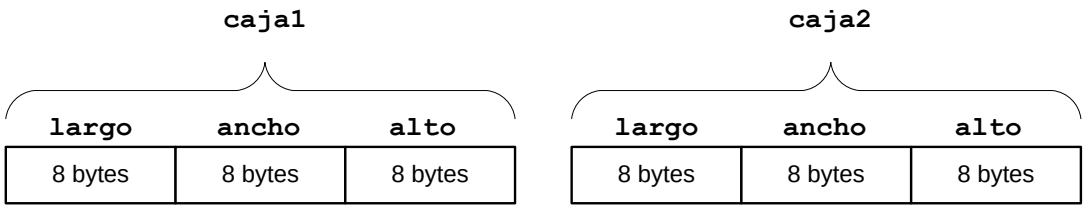
Recordemos que cuando se define una clase, que es un tipo de datos, no se declara ningún objeto del tipo de la clase. Cuando hablamos de acceso a un miembro de la clase, por ejemplo el alto, estamos hablando acerca del acceso al miembro de datos de un objeto particular, que debe ser definido en algún momento.

Declaración de objetos de una clase:

Los objetos de una clase se declaran de exactamente la misma manera que los tipos básicos. Se pueden declarar objetos de la clase `Caja` con las siguientes sentencias:

```
Caja caja1;
Caja caja2;
```

Cada objeto de la clase `Caja` (`caja1` y `caja2`), tiene sus propios datos miembro [4]. Esto se muestra en la figura:



Los campos miembro no están inicializados, contienen basura, por lo que se necesita alguna forma de acceder a ellos.

Funciones miembro de una clase:

Una función miembro de una clase es una función que tiene su definición o su prototipo dentro de la definición de la clase.
Las funciones miembro siempre tienen acceso a todos los campos de la clase. Por ejemplo, si deseamos proveer a la clase `Caja` de una función que retorne el volúmen de la caja desde el cual se la invoca, podemos escribir:

```
class Caja {
      double largo;
      double ancho;
      double alto;

public:
      double Volumen() {
            return largo * ancho * alto;
      }
};
```

En la definición de la clase, contamos con la sección de los miembros privados (`private`), que es el acceso por defecto y la sección de los miembros públicos (`public`), que son los que aparecen después de la etiqueta indicadora.
Si la definición de una clase miembro se desea realizar fuera de la definición de la clase, se debe poner el prototipo de la función dentro de la clase. Como la definición de la función aparece afuera de la clase, debe haber una manera de decirle al compilador que esa función pertenece a la clase. Esto se realiza anteponiendo al nombre de la función el nombre de la clase, y separando ambos por el **operador de resolución de alcance**, '**::**'. Lo ilustramos en el siguiente ejemplo:

```
class Caja {
      double largo;
      double ancho;
      double alto;

public:
      double Volumen();
};

Caja::Volumen() {
      return largo * ancho * alto;
}
```

Esto produce el mismo efecto que el ejemplo anterior, pero no genera exactamente el mismo programa. En el segundo caso, todos los llamados a la función son tratatos de la manera habitual. Sin embargo, si la definición de la función está dentro de la definición de la clase, el compilador considera implícitamente a la misma como una **función inline** [4]. Con una función inline el compilador expande el código del cuerpo de la función en el lugar del llamado a la misma. Esto hace que el código sea más rápido.

Constructores

En algunos casos, inicializar todos los campos de una clase, demanda grandes porciones de código. Además, como los campos son privados, no se tiene acceso a ellos desde fuera de la clase.
Un constructor de clase es una función especial que crea nuevos objetos. Por consiguiente, un constructor provee la oportunidad de inicializar los objetos al momento de ser creados, asignándoles valores válidos. Una clase puede contener varios constructores permitiendo crear objetos de varias maneras.
Los constructores siempre tienen el mismo nombre que la clase en la cual están definidos. Además, los constructores no tienen valor de retorno. El mismo no es necesario, ni se permite. A continuación vemos un ejemplo con un constructor para la clase `Caja`. Por lo tanto ya estamos en condiciones de ver un programa completo:

```
#include <iostream>
using std::cout;
using std::endl;

class Caja {
      double largo;
      double ancho;
      double alto;

public:
      Caja (double lar, double an, double al) {
            largo = lar;
            ancho = an;
            alto = al;
      }
      double Volumen();
};


int main() {
      Caja unaCaja(0.58, 0.25, 0.55);
      cout << "Volúmen de la caja: " << unaCaja.Volumen();
}
```

Observemos que al invocar la función `Volumen` para la instancia de `Caja unaCaja` se utiliza el operador punto ('.') llamado **operador de selección directa de miembros** [2].

Un constructor que requiere un único argumento no necesita ser llamado explícitamente. Por ejemplo:

```
Caja v = 0.8;
```

es lo mismo que

```
Caja c(0.8);
```

El constructor por defecto

Si no se desea inicializar los campos en un constructor, se puede definir un constructor por defecto, también llamado constructor **noarg**:

```
Caja () {};
```

El mismo puede ser definido por el programador, pero en el caso de que no se provea ningún constructor en una clase, el compilador generará un constructor por defecto [4]. En cualquiera de los dos casos, es el constructor que se invoca en una sentencia como la que vimos anteriormente:

```
Caja caja1;
```

Asignación de parámetros por defecto

De la misma manera que con las funciones ordinarias, se pueden establecer valores por defecto en funciones miembro y constructores [4]. Por ejemplo:

```
Caja (double lar, double an, double al = 0.8) {
     largo = lar;
     ancho = an;
     alto = al;
}
```

Listas de inicialización

Puede utilizarse una técnica alternativa para inicializar campos en un constructor llamada **Listas de inicialización**, cuyo formato puede verse en el siguiente ejemplo:

```
Caja (double lar, double an, double al = 0.8): alto(al), ancho(an),
longitud(logt) {}
```

Es muy común, que utilizando este estilo, el cuerpo del constructor quede vacío.

El constructor de copia por defecto

El constructor de copia por defecto (generado por el compilador) es invocado cuando se quiere inicializar un objeto a partir de otro. Por ejemplo:

```
Caja unaCaja(0.58, 0.25, 0.55);
Caja otraCaja = unaCaja;
```

En este caso se inicializa un nuevo objeto llamado `otraCaja` con las mismas dimensiones que el objeto `unaCaja`. En las clases que poseen punteros o arreglos como campos miembro, el constructor de copia por defecto no funciona como se espera, pues los punteros se copian y los punteros del nuevo

objeto quedan apuntando a los miembros del primero. En este caso, el programador debe proveer un constructor de copia.
Cuando se implementa un constructor de copia, el parámetro debe ser una referencia constante:

```
Caja(const Caja& unaCaja);
```

Acceso a campos miembro privados

Como habíamos mencionado, desde afuera de una clase no se puede acceder a sus miembros privados. A la vez, observamos que en un diseño orientado a objetos, el encapsulamiento es una característica esencial. Debemos proteger los campos en la medida de lo posible, pero esto no significa que deban ser secretos. En la sección pública de la clase, se pueden definir funciones para controlar el acceso a los datos privados. Estas funciones tienen como tipo de retorno el mismo tipo del campo miembro en cuestión, se las llama con el prefijo `get` y no reciben ningún argumento. El cuerpo consiste simplemente en retornar el valor actual del atributo. En el siguiente ejemplo se muestra cómo se permite conocer el estado del campo `alto` desde fuera de la clase:

```
double getAlto() {
      return alto;
}
```

En general, se definen funciones similares para cada campo que se quiera hacer disponible desde fuera de la clase sin perjudicar la seguridad de la misma.

Modificación de campos miembro privados

Si se permite setear o modificar valores de campos privados, se procede de manera similar, mediante la definición de funciones. Estas últimas no retornan ningún valor, se las llama con el prefijo `set` y reciben como argumento el valor que será asignado al campo miembro o una referencia del mismo tipo. El cuerpo consiste simplemente en asignar al campo en cuestión, el valor recibido como argumento. En el siguiente ejemplo se muestra cómo se permite modificar el estado del campo `alto` desde fuera de la clase:

```
void setAlto(double al) {
      alto = al;
}
```

Funciones amigas

Existen algunas cirscunstancias donde se desea que algunas funciones que no son miembros de una clase, tengan acceso a todos los miembros privados de la misma [4]. Tales funciones son llamadas **amigas** de la clase. Se las declara anteponiendo la palabra clave **friend**.
Se debe tener en cuenta que si bien el prototipo de las funciones amigas se declara dentro de la definición de la clase, éstas no son miembros de la misma, por lo tanto los atributos de acceso no se aplican sobre ellas.

El puntero this

Cuando se ejecuta una función miembro, ésta contiene un puntero oculto llamado `this`, que apunta al objeto mediante el cual se hizo la llamada a la función. Cuando durante la ejecución se accede a un campo miembro de una clase, por ejemplo `alto`, en realidad la referencia es `this->alto`. El nombre del puntero `this` es agregado por el compilador a los miembros en las funciones. También se puede hacer mención explícita a este puntero [4].

Miembros estáticos de una clase

Los datos y las funciones miembro de una clase pueden ser declarados `static`.
Cuando se declara como estático un campo miembro, éste se crea una única vez y es compartido por todos los objetos de la clase. Cada objeto posee su propia copia de cada uno de los campos miembro ordinarios, pero existe sólo una instancia de los campos miembro estáticos, independientemente de la cantidad de objetos de la clase que sean creados [4]. Un uso común de un dato miembro estático es para contar cuántos objetos de una clase existen. Por ejemplo, se puede declarar en la sección pública de una clase:

```
static int cantCajas;
```

No se puede inicializar un dato miembro estático en la definición de la clase. Por ello, la sentencia debe estar fuera de la clase, por ejemplo de la siguiente manera:

```
int Caja::cantCajas = 0;
```

Se puede hacer referencia a un campo miembro estático desde una instancia de la clase o desde la clase misma:

```
cout << unaCaja.cantCajas;
cout << Caja::cantCajas;
```

Los campos estáticos de una clase se crean automáticamente cuando el programa comienza y se inicializan con el valor 0, es por ello que existen a pesar de que no se instancie ningún objeto de la clase.

Cuando se declara como estática una función, se la hace independiente de cualquier objeto de la clase. La ventaja es que existe y puede ser invocada antes de que se instancie cualquier objeto. Estas funciones, al igual que los campos miembro estáticos, pueden llamarse desde un objeto instanciado o a partir del nombre de la clase.

Destructores

Un destructor es una función miembro que destruye un objeto cuando ya no se lo necesita o cuando queda fuera de alcance. Tiene el mismo nombre que la clase pero precedido con tilde (~). No retorna ningún valor y tampoco toma ningún parámetro, por lo que sólo puede existir un único destructor por clase [4].

Si no se define un destructor para una clase, el compilador siempre genera un destructor por defecto, pero éste no elimina objetos que han alocado memoria con el operador `new`. Por eso se debe definir el destructor con el correspondiente operador `delete`.
Por ejemplo, si un constructor de una clase `ClaseA` es:

```
ClaseA(const char * texto = "Un mensaje") {
      mensaje = new char[strlen(texto) + 1];
      strcpy(mensaje, texto);
}
```

Entonces el destructor puede ser:

```
~ClaseA() {
      delete[] mensaje;
}
```

Sobrecarga de operadores

La sobrecarga de operadores permite que los operadores estándares de C++ puedan programarse para operar con los tipos de datos o clases definidas por el usuario. No está permitido inventar nuevos operadores ni cambiar la precedencia de los existentes. Obviamente, se espera que el comportamiento de un operador sobrecargado sea consistente con el uso normal que se le da al mismo y razonablemente intuitivo [4].

Sólo algunos operadores no pueden sobrecargarse: (`::, ?:, ., sizeof y .*`)

Ejemplo:
Sobrecargamos el operador < (menor) para la clase `Caja`, considerando que 'esta' caja es menor que otra si tiene menor volúmen.

```
bool Caja::operator< (const Caja& unaCaja) const {
      return this->Volumen() < unaCaja.Volumen();
}
```

Aquí `operator` es una palabra clave, que junto con el operador (separado por espacio o no) definen el nombre de la función. Notar que la función se define `const` porque no modifica los datos miembro de la clase. También el argumento debe ser una referencia constante.
El operador < es binario infijo, es decir, toma un operando a la izquierda y un operando a la derecha. En este caso, podremos escribir

```
if (caja1 < caja2) cout << "La caja1 es menor que la caja2 ";
```

cuya expresión entre paréntesis es equivalente a:

```
caja1.operator<(caja2)
```

Funciones de sobrecarga miembros de la clase

La sobrecarga de un operador binario mediante una función miembro, como en el ejemplo anterior, determina implícitamente el operando de la izquierda como el objeto que hace la llamada (`this`). Ejemplos de estas sobrecargas son: asignación, comparaciones contra sí, incremento/decremento, etc.

Funciones de sobrecarga no miembros de la clase

Si se desea sobrecargar un operador en el cual el primer operando no es el puntero `this`, la función de sobrecarga debe ser una función ordinaria o una función amiga. Esta última es de utilidad cuando se necesita acceder a los miembros privados de la clase, por ejemplo cuando se sobrecarga <<.

Clases derivadas

Introducción

C++ proporciona construcciones del lenguaje que soportan directamente la idea del diseño de sistemas que indica: las clases deben usarse para modelar conceptos en el mundo del programador y de la aplicación [1].

Un concepto no existe en forma aislada. Coexiste con otros próximos a él y, gran parte de su fuerza radica en las relaciones entre ellos. Si se piensa en un auto, rápidamente se lo asocia con las nociones de: rueda, motor, conductor, peatón, camión, ambulancia, carreteras, gasolina, exceso de velocidad, etc. Puesto que, usamos clases para representar los conceptos, la cuestión es cómo representar las relaciones entre ellos. [1]

La mente humana [2] clasifica los conceptos de acuerdo a dos dimensiones: pertenencia y variedad. Puede decirse que un taxi es un tipo especial de auto (relación de variedad o, en inglés, una relación de tipo *is a*) y que, una rueda es parte de un auto (relación de pertenencia o, en inglés, una relación de tipo *has a*). C++ permite implementar ambos tipos de relaciones. La combinación de ambos tipos de relaciones es potente: la relación de pertenencia permite el agrupamiento físico de estructuras relacionadas lógicamente y la de variedad o herencia permite que estos grupos de aparición frecuente se reutilicen con facilidad en distintas abstracciones.

La noción de clase derivada y los mecanismos del lenguaje asociados a la misma sirven para expresar relaciones jerárquicas, es decir, para caracterizar aspectos comunes entre las clases. Por ejemplo, los conceptos de círculo y triángulo están relacionados por cuanto ambos son formas, o sea, tienen en común el concepto de forma. Así pues, debe definirse explícitamente las clases Circulo y Triangulo de modo que tengan en común una clase FormaGeometricaPlana. Si se representara un círculo y un triángulo, en un programa, sin que intervenga la noción de forma geométrica plana se perdería algo esencial. [1]. La herencia implica una relación de generalización/especialización en la que, una clase derivada especializa el comportamiento o estructura más general de sus clases bases [3]. Realmente esta es la piedra de toque de la herencia: si B no es un tipo de A, entonces B no debería heredar de A. Las clases bases representan abstracciones generalizadas y, las clases derivadas representan especializaciones en las que, los datos y funciones miembros de la clase base, sufren añadidos, modificaciones o incluso ocultaciones.

Una de las ventajas del mecanismo de derivación de clases es la posibilidad de reutilizar código si tener que escribirlo nuevamente; las clases derivadas pueden utilizar código de la clase base de la jerarquía, sin tener que volver a definirlo en cada una de ellas.

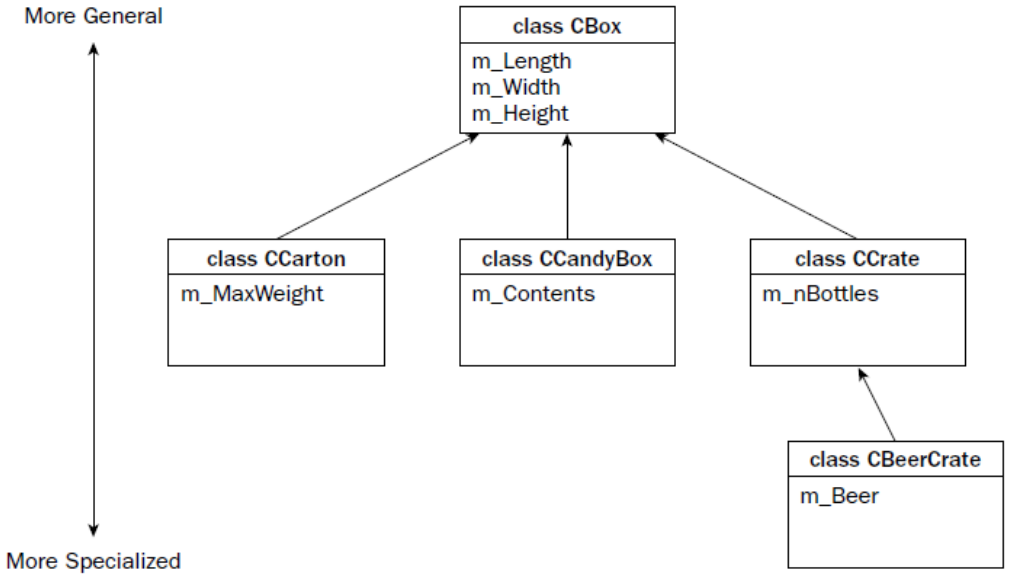
La herencia permite declarar abstracciones con economía de economía de expresión. El ignorar las jerarquías de clases que existen en un diseño, puede conducir a diseños deformados y poco elegantes; sin herencia cada clase sería una unidad independiente, desarrollada partiendo de cero; las distintas clases no guardarían relación entre sí puesto que, el desarrollador de cada una de ellas proporcionaría funciones miembros según se le viniese en ganas [3]. Toda consistencia entre clases es el resultado de una disciplina por parte de los programadores. La herencia posibilita la definición de nuevo software de la misma forma en que se presenta un concepto a un recién llegado: comparándolo con algo que le resulta familiar.
Se explorará en este capítulo las implicancias de esta sencilla idea, que es una de las bases de la programación orientada a objetos.

Clases derivadas

Considere la construcción de una aplicación que maneje cajas de distintos tipos. La clase Caja [4], define una caja en términos de sus dimensiones más un conjunto de funciones públicas que podrían aplicarse a objetos de tipo Caja para resolver problemas asociados a las mismas. La misma tendría este aspecto:

```
//Archivo cabecera Caja.h
#pragma once
class Caja
{
public:
	Caja(double l=1.0, double an=1.0, double al=1.0);
	double volumen(void);
	~Caja(void);
private:
	double largo;
	double ancho;
	double alto;
};

//Archivo Caja.cpp
#include "caja.h"
#include <iostream>
using std::cout;
using std::endl;

Caja::Caja(double l, double an, double al)
{
	largo=l;
	ancho=an;
	alto=al;
}

double Caja::volumen(void)
{
	return largo*ancho*alto;
}

Caja::~Caja(void)
{
//cout << "Se invoca al destructor de Caja" << endl;
}
```

En el mundo real hay muchos tipos de cajas de base rectangular; se las podría diferenciar por la clase de material que contienen, aquél del que están hechas y en muchas otras formas. Se puede definir una clase particular de caja de base rectangular con las características genéricas de todas ellas: alto, ancho, largo y, añadir algunas características adicionales al tipo de caja básica para diferenciarlas del resto. Probablemente habrá nuevas cosas para hacer con este tipo de cajas que no se pueda hacer con otras o, algunas acciones necesitarán redefinirse.
Para representar todo esto puede definirse un tipo genérico de cajas de base rectangular con las características básicas y especificar otras clases de cajas como especializaciones de ellas. En la figura se ilustra un ejemplo de estas relaciones que podrían definirse entre diferentes clases de cajas de base rectangular.



La clase CCrate (representa una caja de botellas) *deriva* de la clase CBox (representa una caja genérica) y, a la inversa, CBox es la *clase base* de CCrate. La clase CCrate tendrá los miembros propios de la clase CBox (alto, ancho, largo) además de los propios (nrobotellas, representando la cantidad de botellas que puede contener la caja). A menudo se representa gráficamente la derivación mediante un puntero desde la clase derivada (tal como se ve en la figura) hasta su clase base, para indicar que la clase derivada se refiere a su base (y no al contrario). A medida que nos movemos hacia abajo en el diagrama, las cajas se vuelven más especializadas
Con frecuencia se dice [1] que una clase derivada hereda todas las propiedades de su clase base, por lo que la relación suele llamarse *herencia*. Lo mismo ocurre con las funciones miembros, con algunas restricciones, entre otras: la clase derivada no hereda el constructor y destructor de la clase base como tampoco el operador de asignación sobrecargado en dicha clase ni funciones y/o datos miembros estáticos de la clase base; las clases derivadas tendrán sus propias versiones de constructor, destructor y operador de asignación sobrecargado.
Cuando se dice que estas funciones no se heredan no significa que no existen como miembros en un objeto de la clase derivada; ellas existen aún para la parte de la clase base que conforma un objeto de la clase derivada.
A veces se denomina a la clase base *superclase* y a la clase derivada *subclase*. Sin embargo, esta terminología induce a error a quien observa que los datos de un objeto de clase derivada son un superconjunto de los datos de un objeto de la clase base. Una clase derivada es mayor que su clase base en el sentido que, usualmente contiene más datos y proporciona más funciones que esta última.

En una implementación gráfica muy conocida y eficiente de la noción de clase derivada, un objeto de la clase derivada, se representa como uno de la clase base añadiendo al final los atributos que pertenecen específicamente a la clase derivada.
Por ejemplo:

Caja

largo alto ancho

CajaBotella

largo alto ancho nrobotellas

Derivar CajadeBotellas de Caja de esta forma, hace que, CajadeBotellas pueda usarse en todos los lugares de un programa donde sea aceptable Caja. Es decir, una CajadeBotellas es (también) una Caja por lo que, puede usarse un puntero CajadeBotellas* como uno Caja* sin conversión explícita de tipos. Sin embargo una Caja no es necesariamente una CajadeBotellas por lo que, no puede usarse un puntero Caja* como uno CajadeBotellas*, la conversión tiene que ser explícita (*cast*).
Dicho de otro modo, un objeto de una clase derivada puede tratarse como un objeto de su clase base, cuando se manipula a través de punteros y referencias. Lo contrario no es cierto. Esta propiedad es muy utilizada para construir listas de objetos heterogéneos (conectados mediante relaciones de herencia). Sólo hay que tener en cuenta que, de este modo sólo puede referirse a miembros de la clase base (funciones y datos). Si mediante el puntero de tipo de la clase base se hace referencia a miembros (funciones y datos) que figuran sólo en la clase derivada, el compilador informará de un error de sintaxis [5].
Para derivar la clase CajaBotellas de Caja, debemos agregar la directiva #include para el archivo cabecera Caja.h, debido a que la clase Caja está en el código de la clase derivada. El nombre de la clase base aparece después del nombre de la clase derivada CajaBotellas y separada por :, sino se especifica nada más, el compilador supone que el estatus de los miembros heredados en la clase derivada es privado: `class CajaBotellas :public Caja`
En este caso y en todos los que trabajemos, supondremos que el especificador de acceso para la clase base es public; todos los miembros heredados y especificados originalmente como public en la clase base, tiene el mismo nivel de acceso en la clase derivada.
En caso de jerarquía de clases de un nivel, a esta clase base (en este ejemplo Caja) se la llama clase base directa y, en el caso de tener una jerarquía de clases (como se verá en un ejemplo posterior con la clase Contenedor) de más de un nivel, se conoce como clase base indirecta, a aquélla que no aparece en la lista de derivación luego de los dos puntos.
Se añade un nuevo miembro en CajaBotellas para representar la cantidad de botellas que puede contener la caja, la cual es inicializada en el constructor:

```
//Archivo de cabecera CajaBotellas.h
#pragma once
#include "caja.h"
```

```
class CajaBotellas :
	public Caja
{
public:
	CajaBotellas(int nro=1);
	double volumen(void);

	~CajaBotellas(void);
private:
	int nrobotellas;
};

//archivo CajaBotellas.cpp
#include "CajaBotellas.h"
#include <iostream>
using std::cout;
using std::endl;

CajaBotellas::CajaBotellas(int nro)
{
	nrobotellas=nro;
}

double CajaBotellas::volumen(void)
{
	return 0.85*Caja::volumen();//ojo no olvidar :: para invocar volumen() de Caja
}
CajaBotellas::~CajaBotellas(void)
{
	//cout << "Se invoca al destructor de CajaBotellas" << endl;
}
```

En el siguiente código se utilizan ambas clases:

```
#include <iostream>
#include "CajaBotellas.h"
using std::cout;
using std::endl;
int main()
{
	Caja caja1(4.0, 3.0, 2.0);
	CajaBotellas cajab1;
	CajaBotellas cajab2(6);
	cout<<"Volumen de caja1: "<<caja1.volumen()<<endl
		<<"Volumen de cajab1: "<<cajab1.volumen()<<endl
		<<"Volumen de cajab2: "<<cajab2.volumen()<<endl;
	return 0;
}
```

Funciones miembro

La solución más limpia para el diseño de jerarquías de clases, es aquélla donde la clase derivada usa sólo los miembros públicos de su clase base, para acceder a los miembros privados de dicha clase base.
Igualmente existe la posibilidad de utilizar miembros protegidos usando la palabra reservada protected. Un miembro protegido es como un miembro público para la clase derivada pero, es como un miembro privado para todas las demás clases. Se debe acceder a dichos miembros protegidos únicamente a través de una referencia de la clase derivada donde se esté utilizando.

Los miembros de una clase derivada [5] pueden hacer referencia a miembros públicos y protegidos de la clase base usando directamente los nombres de los mismos; no es necesario utilizar el operador de resolución de ámbito.
En la clase CajaBotellas obsérvese que se usa :: el operador de resolución de ámbito en la función miembro volumen(), porque la misma ha sido redefinida; esta reutilización de nombres es habitual para añadir más funcionalidad en las clases derivadas y evitar reescribir código; se invoca la versión de la clase base, para que lleve a cabo parte de la nueva tarea. Si se omitiesen los :: el programa caería dentro de una secuencia de llamadas recursivas infinitas. La redefinición de un método no hace desaparecer al original, sin embargo, una función miembro redefinida oculta todas las funciones miembros heredadas con el mismo nombre.

```
//se supone que el volumen efectivo usado de la caja es el 85% del total
double CajaBotellas::volumen(void)
{
	return 0.85*Caja::volumen();//ojo no olvidar :: para invocar volumen() de Caja
}
```

Constructores y destructores

Algunas clases derivadas necesitan constructores. Si la clase base tiene un constructor hay que invocarlo y si, dicho constructor necesita argumentos, hay que proporcionarlos.
Aunque los constructores de la clase base no se heredan, son usados para crear la parte heredada de la clase base, de un objeto de la clase derivada y, esta tarea es responsabilidad del constructor de la clase base.
En el ejemplo de las cajas se invocaba automáticamente el constructor por defecto de la clase base (con los argumentos por defecto).
Para hacer utilizable la clase derivada CajaBotellas se contempla la posibilidad de especificar las dimensiones de la caja de botellas, además del número de botellas que pueda contener. Dicho constructor invoca explícitamente al constructor de la clase base para dar valores iniciales a los datos miembros que heredó de la clase base Caja; la invocación se realiza al final de la signatura del constructor de la clase derivada CajaBotellas luego de añadir :, puede observarse que esto coincide con la forma que puede utilizarse para inicializar datos miembros en un constructor, es decir, al respecto la clase base actúa exactamente igual que un miembro de la clase derivada.

```
//archivo de cabecera de Caja
#pragma once
#include "caja.h"
class CajaBotellas :
	public Caja
{
public:
	CajaBotellas(int nro=1);
	CajaBotellas(double l, double an, double al, int nro=1);
	double volumen(void);

	~CajaBotellas(void);
private:
	int nrobotellas;
};

#include "CajaBotellas.h"
```

```
#include <iostream>
using std::cout;
using std::endl;

CajaBotellas::CajaBotellas(int nro)
{
	cout << "Se invoca al constructor 1 de CajaBotellas" << endl;
	nrobotellas=nro;

}

CajaBotellas::CajaBotellas(double l, double an, double al, int nro):Caja(l, an, al)
{
	cout << "Se invoca al constructor 2 de CajaBotellas" << endl;
	nrobotellas=nro;
}

double CajaBotellas::volumen(void)
{
	return 0.85*Caja::volumen();//ojo no olvidar :: para invocar volumen() de Caja

}

CajaBotellas::~CajaBotellas(void)
{
	cout << "Se invoca al destructor de CajaBotellas" << endl;
}
```

Sino se invoca explícitamente al constructor de la clase base, el compilador dispone que al ejecutarse el código, se invoque al constructor por defecto de la clase base (*no-arg*); sino existe este tipo de constructor, el compilador indica un error.
Un constructor de la clase derivada puede especificar inicializadores sólo para sus propios miembros, no puede inicializar directamente miembros de su clase base directa, eso es responsabilidad de la invocación explicita al constructor de la misma.
Los objetos de una clase se construyen de abajo arriba: primero la base, luego los miembros y a continuación la clase derivada. Se destruyen en el orden contrario: primero la clase derivada, luego los miembros y a continuación la base.
Se añaden mensajes por pantalla a los constructores y destructores para poder observar esta secuencia al ejecutar la aplicación de prueba de dichas clases.

```
#include <iostream>
#include "CajaBotellas.h"
using std::cout;
using std::endl;

int main()
{
	Caja caja1(4.0, 3.0, 2.0);
	CajaBotellas cajab1;
	CajaBotellas cajab2(6);
	CajaBotellas cajab3(1.0, 2.0, 3.0);
	    cout<<"Volumen de caja1: "<<caja1.volumen()<<endl
	       <<"Volumen de cajab1: "<<cajab1.volumen()<<endl
	       <<"Volumen de cajab2: "<<cajab2.volumen()<<endl
	      <<"Volumen de cajab3: "<<cajab3.volumen()<<endl;
	return 0;
}
```

Constructor de copia

La copia de los objetos de una clase la definen el constructor de copia y las asignaciones.
Recuerde al definir un constructor de copia en cualquier clase que, debe recibir como parámetro una referencia a un objeto de dicha clase, para evitar un número infinito de invocaciones al mismo; lo que ocurriría si el argumento se pasa por valor y, por tanto, debe construirse una copia del mismo.
En la clase CajaBotellas se incluye un constructor de copia que se encarga en primer lugar de invocar explícitamente al constructor de copia de la clase base Caja.

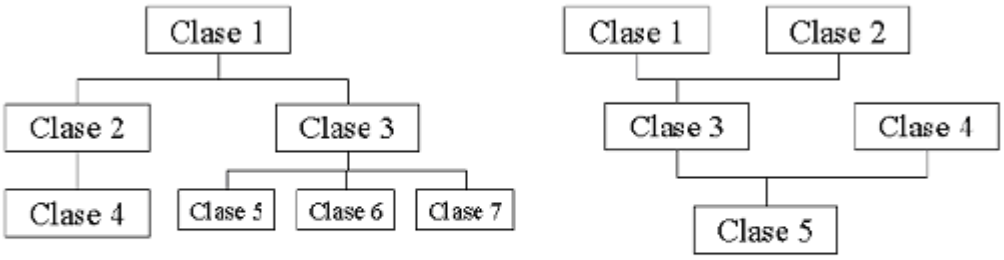
```
#pragma once
class Caja
{
public:
	Caja(double l=1.0, double an=1.0, double al=1.0);
	Caja(const Caja& c);
	double volumen(void);
	~Caja(void);
private:
	double largo;
	double ancho;
	double alto;
};
#include <iostream>
#include "caja.h"
using std::cout;
using std::endl;


Caja::Caja(double l, double an, double al)
{
	largo=l;
	ancho=an;
	alto=al;
	cout << "Se invoca al constructor de Caja" << endl;
}


Caja::Caja(const Caja& c)
{
	largo=c.largo;
	ancho=c.ancho;
	alto=c.alto;
	cout<<"Invocado el constructor por copia de Caja"<<endl;
}

double Caja::volumen(void)
{
	return largo*ancho*alto;
}

Caja::~Caja(void)
{
cout << "Se invoca al destructor de Caja" << endl;
}

#pragma once
#include "caja.h"
```

```
class CajaBotellas :public Caja{
public:
	CajaBotellas(int nro=1);
	CajaBotellas(double l, double an, double al, int nro=1);
	CajaBotellas(const CajaBotellas& cb);
	double volumen(void);

	~CajaBotellas(void);
private:
	int nrobotellas;
};

#include "CajaBotellas.h"
#include <iostream>
using std::cout;
using std::endl;

CajaBotellas::CajaBotellas(int nro)
{
	cout << "Se invoca al constructor 1 de CajaBotellas" << endl;
	nrobotellas=nro;

}

CajaBotellas::CajaBotellas(double l, double an, double al, int nro):Caja(l, an, al)
{
	cout << "Se invoca al constructor 2 de CajaBotellas" << endl;
	nrobotellas=nro;
}

CajaBotellas::CajaBotellas(const CajaBotellas& cb):Caja(cb)
{
cout<<"Invocado constructor por copia de CajaBotellas"<<endl;
nrobotellas=cb.nrobotellas;
}

double CajaBotellas::volumen(void)
{
	return 0.85*Caja::volumen();//ojo no olvidar :: para invocar volumen() de Caja

}

CajaBotellas::~CajaBotellas(void)
{
	cout << "Se invoca al destructor de CajaBotellas" << endl;
}

#include <iostream>
#include "CajaBotellas.h"
using std::cout;
using std::endl;

int main()
{
	Caja caja1(4.0, 3.0, 2.0);
	CajaBotellas cajab1;
	CajaBotellas cajab2(6);
	CajaBotellas cajab3(1.0, 2.0, 3.0);
	CajaBotellas cajab4(cajab3); //usa constructor por copia
```

```
            cout<<"Volumen de caja1: "<<caja1.volumen()<<endl
            <<"Volumen de cajab1: "<<cajab1.volumen()<<endl
            <<"Volumen de cajab2: "<<cajab2.volumen()<<endl
            <<"Volumen de cajab3: "<<cajab3.volumen()<<endl;
            return 0;
}
```

Jerarquía de clases

La jerarquía de clases conectadas mediante relaciones de herencia forma estructuras de apariencia arborescente.
Una clase puede heredar datos y funciones miembros de una o más clases base, aquí sólo consideraremos la herencia simple (heredar de una única clase base) y no la múltiple.



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

Herencia Múltiple: Las clases derivadas tienen varias clases base

Figura 4: Herencia simple y herencia múltiple.

Tipos de campos

Una forma de determinar el tipo de un objeto perteneciente a una jerarquía de clases mediante herencia es incorporar una sentencia switch que luego permita invocar la acción apropiada. Esta solución expone a una variedad de problemas potenciales: el programador podría olvidarse de incluir una prueba de tipos cuando es obligatoria hacerla o de evaluar todos los casos posibles; al modificar un programa basado en este tipo de solución, agregando nuevos tipos, el programador podría olvidar insertar los nuevos casos en dicha sentencia. Cada adición o eliminación de una clase requiere [5] la modificación de todos los switch del programa; rastrear estas instrucciones puede ser un proceso que consuma mucho tiempo y propenso a errores. Veremos una solución más adecuada.

Funciones virtuales

Se añade a la clase Caja una función miembro mostrarVolumen( ) que se encarga de invocar a la función volumen( ) y mostrar en pantalla el valor calculado por dicha función. Esta función, por pertenecer a la interfaz pública de esta clase base es heredada por las clases derivadas de la misma. En la funcion main( ) de una clase de prueba, se invoca dicha función para mostrar los datos de una Caja y de una CajaBotellas, de las mismas dimensiones (en caso de CajaBotellas el volumen debería ser 85% menor que la Caja de mismas dimensiones). Se observa que al ser invocada produce los mismos resultados.

```
void Caja::mostrarVolumen(void) const
{
	cout<<endl
		<<"El volumen de la caja es: "<<volumen();
}

#include <iostream>
#include "CajaBotellas.h"
using std::cout;
using std::endl;


int main()
{
	Caja caja1(4.0, 3.0, 2.0);
	CajaBotellas cajab1(4.0, 3.0, 2.0);
	Caja* cajap=0; //puntero nulo a la clase base Caja
	caja1.mostrarVolumen();
	cajab1.mostrarVolumen();//sino es virtual se llama volumen de la clase Caja
	cajap=&caja1; //puntero a objeto de la clase base
	cajap->mostrarVolumen();
	cajap=&cajab1; //puntero a objeto de la clase derivada
	cajap->mostrarVolumen();
	cout<<endl;
	return 0;
}
```

La razón es que, mostrarVolumen() es una función miembro de la clase base y, en tiempo de compilación, se resuelve la invocación a volumen( ) que se realiza en el cuerpo de dicha función (enlace estático o temprano por *early binding*) y siempre se invoca a la versión de la clase base.
Para obtener el comportamiento esperado, la invocación a la función volumen( ) correcta (de la clase base o de alguna clase derivada), determinada por el tipo de objeto asociado, debería resolverse en tiempo de ejecución, es decir, debería usarse un enlace dinámico (o tardío por *late binding*) y no, arbitrariamente fijado por el compilador antes de ejecutar la aplicación donde figura este código. C++ provee la forma de resolver este tema: las funciones virtuales.
Una función virtual es una función en una clase base que se declara usando la palabra reservada virtual. Si existe, en alguna clase derivada otra definición para dicha función declarada como virtual en la clase base, le indica al compilador que no debe usar enlace estático para la misma, se determinará en tiempo de ejecución qué versión se invoca. A menudo a las funciones virtuales se les llama métodos.
Si se agrega la palabra virtual a la definición de la función volumen( ) la aplicación funciona correctamente. No es esencial agregar la palabra virtual a la versión de la función volumen( ) que está en la clase derivada CajaBotellas; sólo es obligatorio hacerlo en la clase en la que se declara por primera vez Simplemente se recomienda hacerlo para facilitar la lectura de la definición de las clases derivadas e informar cuáles funciones son virtuales y por tanto serán seleccionadas dinámicamente en tiempo de ejecución.

```
#pragma once
class Caja
{
public:
	Caja(double l=1.0, double an=1.0, double al=1.0);
	Caja(const Caja& c);
	virtual double volumen(void) const;//se usa virtual en la declaración de la clase
	void mostrarVolumen(void) const
	~Caja(void);
```

```
protected:
	double largo;
	double ancho;
	double alto;
};

#include <iostream>
#include "caja.h"
using std::cout;
using std::endl;


Caja::Caja(double l, double an, double al)
{
	largo=l;
	ancho=an;
	alto=al;
	cout << "Se invoca al constructor de Caja" << endl;
}


Caja::Caja(const Caja& c)
{
	largo=c.largo;
	ancho=c.ancho;
	alto=c.alto;
	cout<<"Invocado el constructor por copia de Caja"<<endl;
}

double Caja::volumen(void) const
{
	return largo*ancho*alto;
}
void Caja::mostrarVolumen(void) const
{
	cout<<endl
		<<"El volumen de la caja es: "<<volumen();
}
Caja::~Caja(void)
{
cout << "Se invoca al destructor de Caja" << endl;
}

#pragma once
#include "caja.h"
class CajaBotellas :
	public Caja
{
public:
	CajaBotellas(int nro=1);
	CajaBotellas(double l, double an, double al, int nro=1);
	CajaBotellas(const CajaBotellas& cb);
	virtual double volumen(void) const;
	~CajaBotellas(void);
private:
	int nrobotellas;
};

#include "CajaBotellas.h"
#include <iostream>
```

```
using std::cout;
using std::endl;

CajaBotellas::CajaBotellas(int nro)
{
	cout << "Se invoca al constructor 1 de CajaBotellas" << endl;
	nrobotellas=nro;

}

CajaBotellas::CajaBotellas(double l, double an, double al, int nro):Caja(l, an, al)
{
	cout << "Se invoca al constructor 2 de CajaBotellas" << endl;
	nrobotellas=nro;
}

CajaBotellas::CajaBotellas(const CajaBotellas& cb):Caja(cb)
{
cout<<"Invocado constructor por copia de CajaBotellas"<<endl;
nrobotellas=cb.nrobotellas;
}
double CajaBotellas::volumen(void) const
{
	return 0.85*largo*ancho*alto;

}

CajaBotellas::~CajaBotellas(void)
{
	cout << "Se invoca al destructor de CajaBotellas" << endl;
}
```

Para que una función se comporte como virtual en una clase derivada, debe tener el mismo nombre, lista de parámetros y tipo de retorno que la versión de la clase base. Si la función en la clase base es const, en la clase derivada también lo deberá ser. Si estos requisitos no se cumplen, el mecanismo de función virtual no funcionará y se usará enlace estático en tiempo de compilación.
Se puede usar una función virtual en una clase aunque no se derive ninguna otra clase de ella. Tampoco es necesario que una clase derivada proporcione una versión de una función definida como virtual en su clase base, sólo debe proporcionarse una versión adecuada si es necesaria sino, se usa la versión de la clase base.
Las funciones virtuales constituyen un mecanismo extraordinariamente poderoso; cuando se usan de manera adecuada proporcionan un buen grado de estabilidad a un programa en evolución. El término polimorfismo es una de las características más importantes de la programación orientada a objetos. Mediante el polimorfismo, un nombre (tal como la declaración de una variable) puede denotar objetos de muchas clases diferentes, relacionadas por una clase base común; cualquier objeto denotado por este nombre es, por tanto, capaz de responder a algún conjunto común de operaciones de distintas formas [3]. Existe polimorfismo cuando interactúan la herencia y el enlace dinámico.
Para obtener polimorfismo en C++ debe usarse funciones virtuales, invocadas a través de punteros o referencias. Cuando se manipulan los objetos directamente (en lugar de hacerlo de esta forma indirecta) su tipo exacto es conocido en tiempo de compilación y no es necesario polimorfismo en tiempo de ejecución.
La invocación a funciones que están redefinidas en clases derivadas, pero usando el operador de resolución de ámbito (::) asegura que no se usa el mecanismo de las funciones virtuales, siempre se invoca a la versión de la clase base.

Cada clase que utiliza funciones virtuales tiene un vector de punteros, uno por cada función virtual, llamado *v-table*. En el caso que una clase derivada no tenga versión propia de una función definida como virtual, el puntero apuntará a la versión de la clase más próxima en la jerarquía de clases, que tenga una definición propia de dicha función virtual. Es decir se busca primero, al invocar una función virtual, en la propia clase, luego en la clase anterior en la jerarquía y así, subiendo hasta encontrar una clase que tenga la definición de la función buscada. Cada objeto que se crea, de alguna clase que tenga funciones virtuales, contiene un puntero oculto a la *v-table* de su clase.
Las funciones virtuales no pueden declararse static puesto que, carecen del puntero this y, las funciones virtuales lo necesitan para la mecánica de su funcionamiento.

Punteros a objetos y funciones virtuales

A un puntero a un objeto de una clase base puede también asignársele un objeto de la clase derivada. Por tanto puede usarse, punteros a objetos de la clase base conjuntamente con funciones virtuales, para obtener distintos comportamientos dependiendo del tipo real de objeto al que se apunta.
En el ejemplo de las cajas ahora el método mostrarvolumen() se invoca mediante punteros a objetos de la clase base Caja:

```
#include <iostream>
#include "CajaBotellas.h"
using std::cout;
using std::endl;

int main()
{
	Caja caja1(4.0, 3.0, 2.0);
	CajaBotellas cajab1(4.0, 3.0, 2.0);
	Caja* cajap=0; //puntero nulo a la clase base Caja
	caja1.mostrarVolumen();
	cajab1.mostrarVolumen();//sino es virtual se llama volumen de la clase Caja
	cajap=&caja1; //puntero a objeto de la clase base
	cajap->mostrarVolumen();
	cajap=&cajab1; //puntero a objeto de la clase derivada
	cajap->mostrarVolumen();
	cout<<endl;
	return 0;
}
```

Ahora el mecanismo de funciones virtuales garantiza que se invoque la versión correcta de volumen( ).

Referencias a objetos y funciones virtuales

Análogamente a los punteros, las referencias a objetos de la clase base pueden utilizarse para referirse a objetos de clases derivadas. Este hecho combinado con el uso de funciones virtuales hará que, en tiempo de ejecución, se invoque la versión correcta de dicha función virtual.
Con respecto al ejemplo de las cajas, se añade en el archivo de prueba de las clases creadas, aparte de main( ) otra función Output( ) que toma como argumento una referencia a objetos de la clase base Caja, dicha función utiliza esta referencia para invocar a la función virtual volumen( )

```
#include <iostream>
#include "CajaBotellas.h"
using std::cout;
using std::endl;
```

```
void Output(const Caja& c); //prototipo de función para ver referencia +func.virtuales

int main()
{
	Caja caja1(4.0, 3.0, 2.0);
	CajaBotellas cajab1(4.0, 3.0, 2.0);
	Caja* cajap=0; //puntero nulo a la clase base Caja
	caja1.mostrarVolumen();
	cajab1.mostrarVolumen();//sino es virtual se llama volumen de la clase Caja
	cajap=&caja1; //puntero a objeto de la clase base
	cajap->mostrarVolumen();
	cajap=&cajab1; //puntero a objeto de la clase derivada
	cajap->mostrarVolumen();
	Output(caja1);
	Output(cajab1);
	cout<<endl;
	return 0;
}

void Output(const Caja& c){
	c.mostrarVolumen();
}
```

Funciones virtuales puras y clases abstractas

Muchas clases tales como FormaGeometricaPlana representan conceptos abstractos para los cuales no pueden existir objetos, sólo tienen sentido para derivar de ellas formas geométricas planas concretas, tales como círculos. Determinadas operaciones, como calcular área, sólo tienen sentido para formas geométricas concretas. Para este tipo de funciones la solución es usar funciones virtuales puras. Una función virtual pura se logra mediante el inicializador =0.

Una clase con, al menos una función virtual pura, se denomina clase abstracta, no pudiéndose crear objetos de la misma; sólo puede utilizarse como interfaz y clase base para otras derivadas de ellas. No puede utilizarse una clase abstracta como tipo a pasar como parámetro o como tipo devuelto por una función. Sí, está permitido definir punteros o referencias que, posteriormente serán inicializados con objetos de clases derivadas concretas.

Una función virtual pura que no esté redefinida en una clase derivada, permanece como función virtual pura por lo que, dicha clase derivada se convierte también en clase abstracta. Un uso importante de las clases abstractas es proporcionar una interfaz sin exponer ningún detalle de implementación.

Al ejemplo de las cajas se le añade una clase base abstracta Contenedor puesto que contiene el método virtual puro volumen( ). Ahora la clase Caja, un tipo particular de contenedor, se deriva de dicha clase base abstracta; en Caja el método volumen( ) está perfectamente definido puesto que se crearán objetos de dicha clase. Se define también una clase Lata, que es otro tipo de Contenedor y, también se define el volumen( ) de acuerdo a la fórmula: $h\pi r^2$

```
#pragma once
class Contenedor //contenedor genérico, clase abstracta
{
public:
	virtual double volumen() const=0;//virtual sólo en declaración
	void mostrarVolumen() const;
	//al usar punteros a la clase base no se invocan a los destructores correctos
	virtual ~Contenedor(void);//para invocar el destructor correcto agrego virtual

};
```

```
#include "Contenedor.h"
#include <iostream>
using std::cout;
using std::endl;

void Contenedor::mostrarVolumen() const
{
	cout<<endl
		<<"El volumen de la caja es: "<<volumen()<<endl;
}
Contenedor::~Contenedor(void)
{
	cout<<"Invocado el destructor de Contenedor"<<endl;
}


#pragma once
#include "Contenedor.h"
class Caja: public Contenedor
{
public:
	Caja(double l=1.0, double an=1.0, double al=1.0);
	Caja(const Caja& c);
	virtual double volumen(void) const;
	~Caja(void);
	protected:
	double largo;
	double ancho;
	double alto;

};

#include <iostream>
#include "Caja.h"
using std::cout;
using std::endl;



Caja::Caja(double l, double an, double al)
{
	largo=l;
	ancho=an;
	alto=al;
	cout << "Se invoca al constructor de Caja" << endl;
}

Caja::Caja(const Caja& c)
{
	largo=c.largo;
	ancho=c.ancho;
	alto=c.alto;
	cout<<"Invocado el constructor por copia de Caja"<<endl;
}

double Caja::volumen(void) const
{
	return largo*ancho*alto;
}
```

```
Caja::~Caja(void)
{
cout << "Se invoca al destructor de Caja" << endl;
}
```

```
#pragma once
#include "contenedor.h"
class Lata :
	public Contenedor
{
public:
	Lata(double al=4.0, double d=2.0);
	virtual double volumen() const;
	~Lata(void);
protected:
	double diametro;
	double alto;
};
```

```
#include "Lata.h"
#include <iostream>
using std::cout;
using std::endl;
const double PI = std::atan(1.0)*4;


Lata::Lata(double al, double d):alto(al), diametro(d)
{
	cout<<"Invocado constructor de Lata"<<endl;
}

double Lata::volumen() const
{
	return PI* diametro*diametro*alto;
}
Lata::~Lata(void)
{
cout<<"Invocado destructor de Lata"<<endl;
}
```

```
#pragma once
#include "caja.h"
class CajaBotellas :
	public Caja
{
public:
	CajaBotellas(int nro=1);
	CajaBotellas(double l, double an, double al, int nro=1);
	CajaBotellas(const CajaBotellas& cb);
	virtual double volumen(void) const;//agrego para usar ref constante en Output

	~CajaBotellas(void);
private:
	int nrobotellas;
};
```

```
#include "CajaBotellas.h"
#include <iostream>
```

```
using std::cout;
using std::endl;

CajaBotellas::CajaBotellas(int nro)
{
	cout << "Se invoca al constructor 1 de CajaBotellas" << endl;
	nrobotellas=nro;

}

CajaBotellas::CajaBotellas(double l, double an, double al, int nro):Caja(l, an, al)
{
	cout << "Se invoca al constructor 2 de CajaBotellas" << endl;
	nrobotellas=nro;
}

CajaBotellas::CajaBotellas(const CajaBotellas& cb):Caja(cb)
{
cout<<"Invocado constructor por copia de CajaBotellas"<<endl;
nrobotellas=cb.nrobotellas;
}

double CajaBotellas::volumen(void) const
{
	return 0.85*largo*ancho*alto;

}



CajaBotellas::~CajaBotellas(void)
{
	cout << "Se invoca al destructor de CajaBotellas" << endl;
}

#include <iostream>
#include "CajaBotellas.h"
#include "Lata.h"
using std::cout;
using std::endl;



int main(void)
{
	//puntero a la clase base abstracta Contenedor que apunta a un objeto Caja nuevo
	Contenedor* pc1=new Caja(2.0, 3.0, 4.0);
	//puntero a la clase base abstracta Contenedor que apunta a un objeto Lata nuevo
	Contenedor* pl1=new Lata(6.5, 3.0);
	Lata l1(6.5, 3.0); //crea otra lata igual a la anterior
	CajaBotellas cb(2.0, 3.0, 4.0);//crea caja de botellas
	pc1->mostrarVolumen();
	pl1->mostrarVolumen();
	cout<<endl;
	//limpia el espacio asignado dinámicamente
	delete pc1;
	delete pl1;
	//inicializa pc1 con la dirección de la lata l1
	pc1=&l1;
	pc1->mostrarVolumen();
```

```
        //ahora el puntero pc1 apunta a la dirección de CajaBotellas cb
        pc1=&cb;
        pc1->mostrarVolumen();
        return 0;
}
```

Referencias

[1] Stroustrup, Bjarne, El lenguaje de programación C++, 3.a edición. Addison-Wesley, (1998).
[2] Escuela Superior de Ingenieros Industriales de San Sebastián, UNIVERSIDAD DE NAVARRA, Aprenda C++ como si estuviera en primero. Disponible en: http://mat21.etsii.upm.es/ayudainf/aprendainf/Cpp/manualcpp.pdf.
[3] Booch, G., Análisis y Diseño Orientado a Objetos con Aplicaciones, Addison Wesley (1999).
[4] Horton, Ivor, Ivor Horton's Beginning Visual C++ 2008, Wiley Publishing / WROX Programmer to Programmer, (2008).
[5] H.M. Deitel, P.J. Deitel, Como programar en C/C++ (2ª ed), Prentice-Hall, (1995).