



Las abstracciones forman una jerarquía

# Herencia y polimorfismo

# 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**
- ▶ 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
- ▶ Puesto que, usamos **clases** para representar los **conceptos**, la cuestión es **cómo representar las relaciones entre ellos**



# Introducción

---

- ▶ La mente humana **clasifica** los **conceptos** de acuerdo a 2 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
- 



# Introducción



# Herencia

---



**Una clase derivada puede heredar estructura y comportamiento de su(s) clase(s) base (Booch)**

---



# Herencia

---

- ▶ 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 Círculo y Triangulo** de modo que tengan en **común** una clase **FormaGeometricaPlana**
  - ▶ 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**
- 



# Herencia

---

- ▶ 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 sin tener que escribirlo nuevamente; las clases derivadas pueden reutilizar código de la clase base de la jerarquía, sin tener que volver a definirlo en cada una de ellas



# Herencia

---

- ▶ La **herencia** permite declarar abstracciones con **economía de expresión**
- ▶ Sin herencia cada clase sería una unidad independiente, **desarrollada partiendo de cero**; las distintas clases no guardarían relación entre sí
- ▶ 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**





# Clases derivadas

---

- ▶ Considere la construcción de una **aplicación** que maneje **cajas de sección rectangular, de distintos tipos**
- ▶ La clase **Caja**, 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



# Clases derivadas -Ejemplo

---

```
/* * Caja.h
**/
#ifndef CAJA_H_
#define CAJA_H_
class Caja
{
public:
    Caja(double l=1.0, double an=1.0, double al=1.0);
    ~Caja(void);
    double volumen(void) const;
private:
    double largo;
    double ancho;
    double alto;
};
#endif /* CAJA_H_ */
```



# Clases derivadas -Ejemplo

---

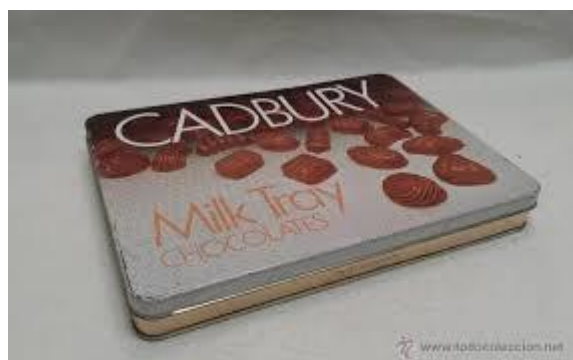
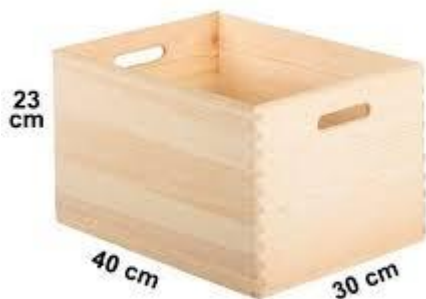
```
#include "caja.h"
#include <iostream>
using std::cout;
using std::endl;
Caja::Caja(double l, double an, double al)
{
    largo=l;
    ancho=an;
    alto=al;
}
double Caja::volumen(void) const
{
    return largo*ancho*alto;
}
Caja::~Caja(void)
{
    cout << "Se invoca al destructor de Caja" << endl;
}
```

---



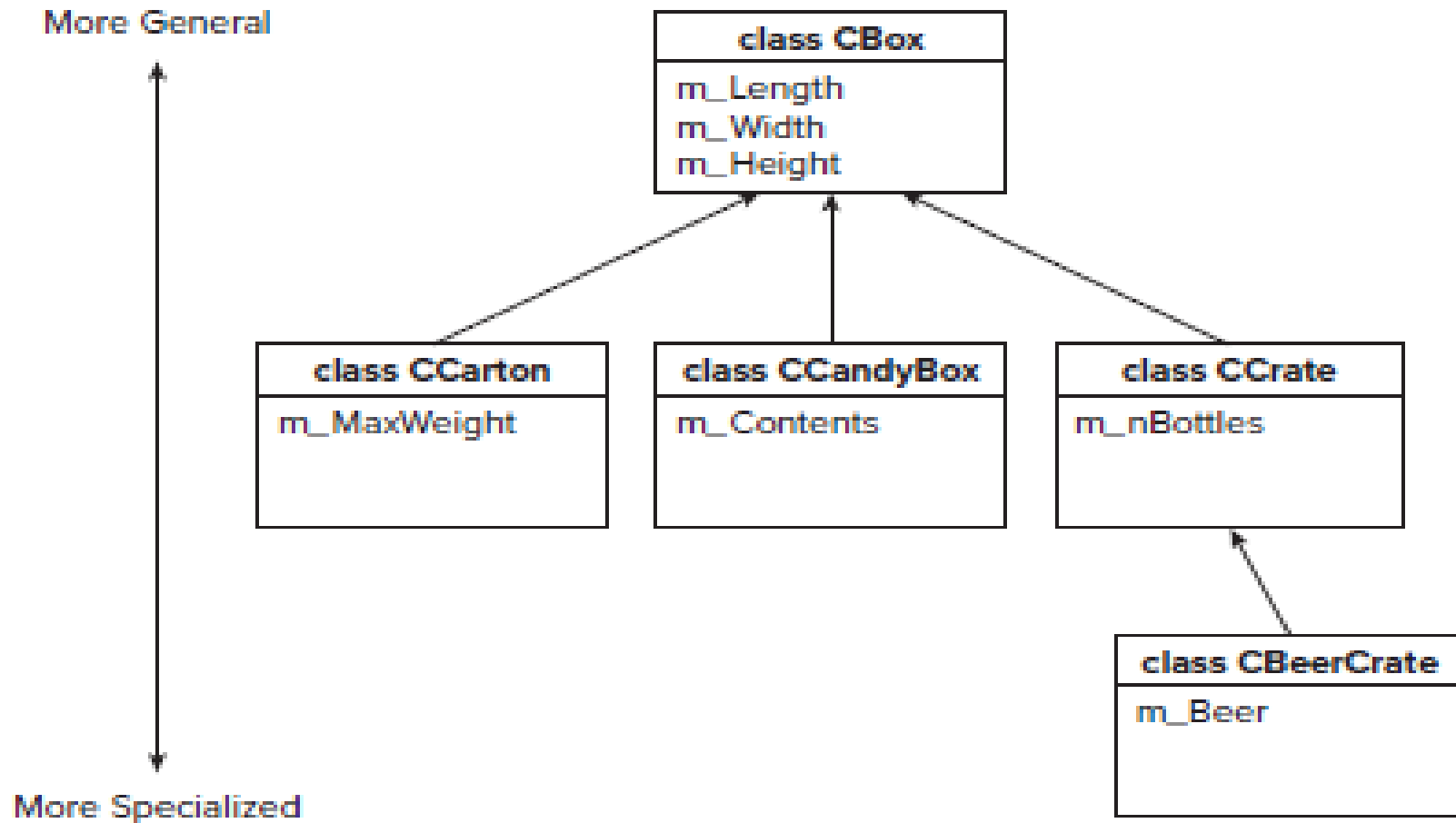
# Clases derivadas -Ejemplo

---



# Clases derivadas -Ejemplo

---



# Clases derivadas -Ejemplo

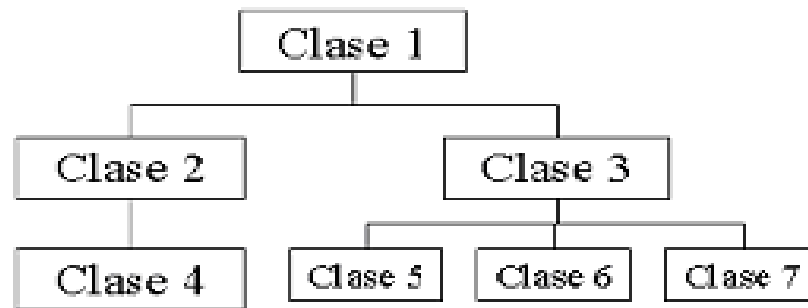
---

- ▶ 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
  - ▶ 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 una flecha desde la clase derivada 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
- 

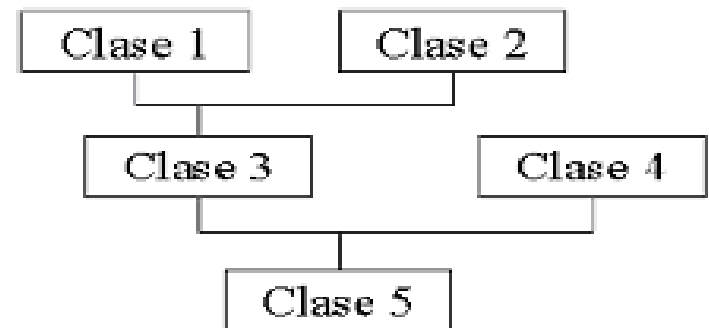


# Jerarquía de clases

- ▶ 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

# Clases derivadas

---

- ▶ Con frecuencia se dice 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**





# Clases derivadas -Ejemplo

---

Caja

largo  
alto  
ancho

CajaBotella

largo  
alto  
ancho  
nrobotellas



# Clases derivadas -Ejemplo

---

- ▶ Derivar **CajaBotellas** de **Caja** de esta forma, hace que, **CajaBotellas** pueda usarse en todos los lugares de un programa **donde sea aceptable Caja (es una Caja)**
  - ▶ Es decir, una **CajaBotellas** 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 (***downcast***)
  - ▶ **CajaBotellas\***  
`cb=dynamic_cast<CajaBotellas*>punterocajabasica;`
- 



# Clases derivadas

---

- ▶ 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**



# Clases derivadas -Ejemplo

---

- ▶ 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 (como si hubiesen sido declarados **public** dentro de la clase derivada)



# Clases derivadas -Ejemplo

---

- ▶ 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**



# Clases derivadas - Ejemplo

---

```
▶ /** CajaBotellas.h*/  
#ifndef CAJABOTELLAS_H_  
#define CAJABOTELLAS_H_  
  
#include "Caja.h"  
class CajaBotellas :public Caja  
{  
public:  
    CajaBotellas(int nro=1); //argumento por defecto  
    double volumen(void) const;  
    ~CajaBotellas(void);  
private:  
    int nrobotellas;  
};  
#endif /* CAJABOTELLAS_H_ */
```

---



# Clases derivadas - Ejemplo

---

```
/*CajaBotellas.cpp*/
#include "CajaBotellas.h"
#include <iostream>
using std::cout;
using std::endl;
CajaBotellas::CajaBotellas(int nro)
{
    nrobotellas=nro;
}
double CajaBotellas::volumen(void) const
{
    return 0.85*Caja::volumen();
//ojo no olvidar :: para invocar volumen() de Caja
}
CajaBotellas::~CajaBotellas(void)
{
    cout << "Se invoca al destructor de CajaBotellas" << endl;}
```

# Clases derivadas - Ejemplo

```
#include <iostream>
```

```
#include "CajaBotellas.h"
```

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

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

```
int main()
```

```
{
```

```
    Caja caja1(4.0, 3.0, 2.0);
```

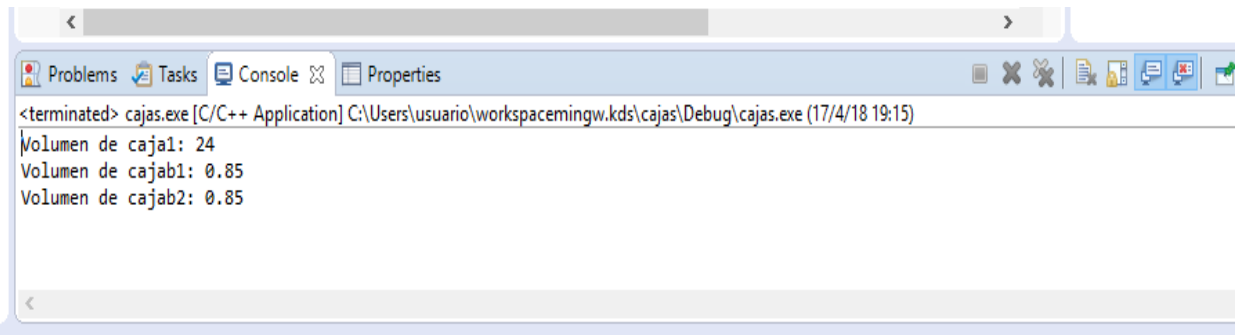
```
    CajaBotellas cajab1;
```

```
    CajaBotellas cajab2(6);
```

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

```
    return 0;
```

```
}
```



The screenshot shows a Visual Studio IDE window with the 'Console' tab selected. The console output displays the results of the program execution: 'Volumen de caja1: 24', 'Volumen de cajab1: 0.85', and 'Volumen de cajab2: 0.85'. The window title bar indicates the file path 'C:\Users\usuario\workspace\mingw.kds\cajas\Debug\cajas.exe' and the timestamp '(17/4/18 19:15)'.

```
<terminated> cajas.exe [C/C++ Application] C:\Users\usuario\workspace\mingw.kds\cajas\Debug\cajas.exe (17/4/18 19:15)  
Volumen de caja1: 24  
Volumen de cajab1: 0.85  
Volumen de cajab2: 0.85
```



# Constructores y destructores

---

- ▶ En el ejemplo de las cajas se invocaba automáticamente el constructor por defecto de la clase base (con los argumentos por defecto)
- ▶ 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



# Constructores y destructores

---

- ▶ 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 signature** 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**



# Constructores y destructores

---

- ▶ 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 explícita al constructor de la misma
- ▶ Se añaden mensajes por pantalla a los constructores y destructores para poder observar la secuencia de creación/destrucción al ejecutar la aplicación de prueba de dichas clases



# Ejemplo

---

```
/** CajaBotellas.h*/  
#ifndef CAJABOTELLAS_H_  
#define CAJABOTELLAS_H_  
  
#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)const;  
    ~CajaBotellas(void);  
private:  
    int nrobotellas;  
};  
#endif /* CAJABOTELLAS_H_ */
```

---



# Ejemplo

---

```
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)
```

```
{
```

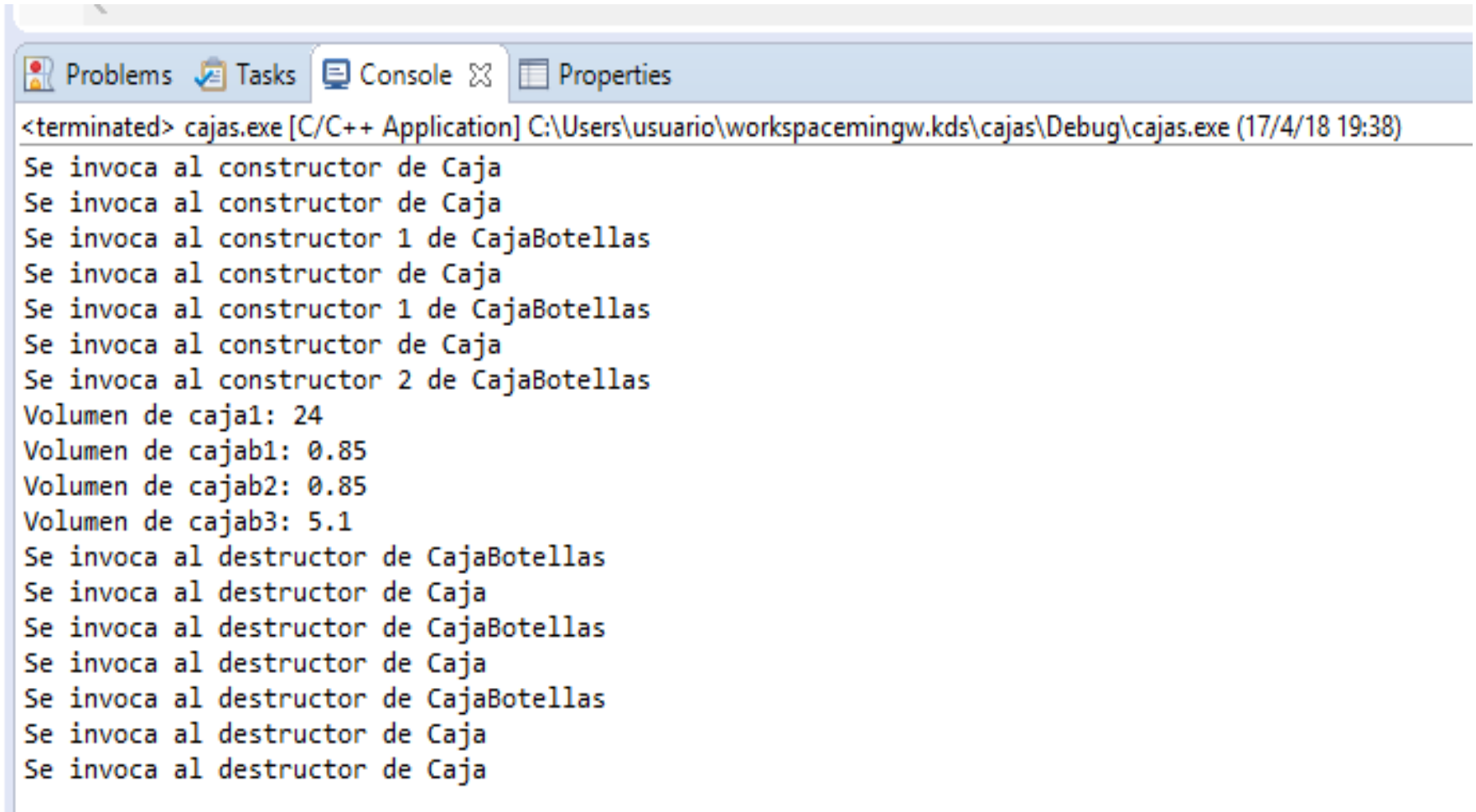
```
    cout << "Se invoca al constructor 2 de  
    CajaBotellas" << endl;
```

```
    nrobotellas=nro;
```

```
}
```



# Ejemplo



```
<terminated> cajas.exe [C/C++ Application] C:\Users\usuario\workspacemingw.kds\cajas\Debug\cajas.exe (17/4/18 19:38)
Se invoca al constructor de Caja
Se invoca al constructor de Caja
Se invoca al constructor 1 de CajaBotellas
Se invoca al constructor de Caja
Se invoca al constructor 1 de CajaBotellas
Se invoca al constructor de Caja
Se invoca al constructor 2 de CajaBotellas
Volumen de caja1: 24
Volumen de cajab1: 0.85
Volumen de cajab2: 0.85
Volumen de cajab3: 5.1
Se invoca al destructor de CajaBotellas
Se invoca al destructor de Caja
Se invoca al destructor de CajaBotellas
Se invoca al destructor de Caja
Se invoca al destructor de CajaBotellas
Se invoca al destructor de Caja
Se invoca al destructor de Caja
```

Se añade `CajaBotellas cajab3(1.0, 2.0, 3.0);`

# 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** 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**
- 



# Funciones miembro Ejemplos

---

- ▶ 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**





# Funciones miembro Ejemplos

---

```
/*se supone que el volumen efectivo usado  
de la caja es el 85% del total*/
```

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



# Funciones virtuales

---

- ▶ 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 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 - Ejemplo

---

- ▶ 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 función **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!!**
- 



# Funciones virtuales - Ejemplo

---

```
void Caja::mostrarVolumen(void) const
```

```
{//en Caja.cpp, añadir al Caja.h el prototipo
```

```
    cout<<endl<<"El volumen de la caja es: "<<volumen();
```

```
}
```

```
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;
```

```
}
```



# Funciones virtuales - Ejemplo

---

 Problems  Tasks  Console  Properties

<terminated> cajas.exe [C/C++ Application] C:\Users\usuario\workspacemingw.kds\cajas\Debug\cajas.exe (17/4/18 20:38)

---

Se invoca al constructor de Caja

Se invoca al constructor de Caja

Se invoca al constructor 2 de CajaBotellas

El volumen de la caja es: 24

El volumen de la caja es: 24

El volumen de la caja es: 24

El volumen de la caja es: 24

Se invoca al destructor de CajaBotellas

Se invoca al destructor de Caja

Se invoca al destructor de Caja



# Funciones virtuales - Ejemplo

---

- ▶ 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**
- 



# Funciones virtuales - Ejemplo

---

- ▶ 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( ) en la clase Caja**, 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 1º 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



# Funciones virtuales - Ejemplo

---

```
class Caja
```

```
{
```

```
public:
```

```
    Caja(double l=1.0, double an=1.0, double  
    al=1.0);
```

```
    ~Caja(void);
```

```
    virtual double volumen(void) const;
```

```
    void mostrarVolumen(void) const;
```

```
private:
```

```
    double largo;
```

```
    double ancho;
```

```
    double alto;
```

```
};
```

---



# Funciones virtuales - Ejemplo

---

 Problems  Tasks  Console  Properties

<terminated> cajas.exe [C/C++ Application] C:\Users\usuario\workspacemingw.kds\cajas\Debug\cajas.exe (17/4/18 20:48)

---

Se invoca al constructor de Caja

Se invoca al constructor de Caja

Se invoca al constructor 2 de CajaBotellas

El volumen de la caja es: 24

El volumen de la caja es: 20.4

El volumen de la caja es: 24

El volumen de la caja es: 20.4

Se invoca al destructor de CajaBotellas

Se invoca al destructor de Caja

Se invoca al destructor de Caja



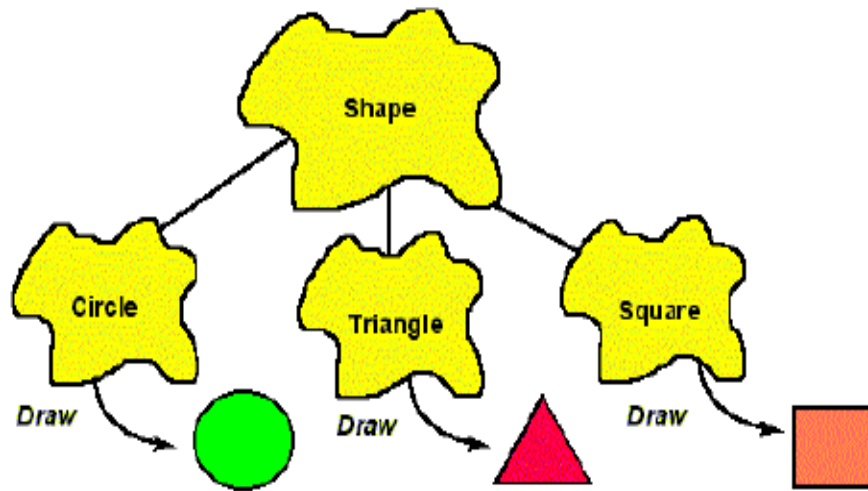
# Funciones virtuales

---

- ▶ 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**
-

# Polimorfismo

---



*Proteo cambia de forma*

# Polimorfismo

---

- ▶ 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**
- ▶ Existe **polimorfismo** cuando interactúan la **herencia** y el **enlace dinámico**



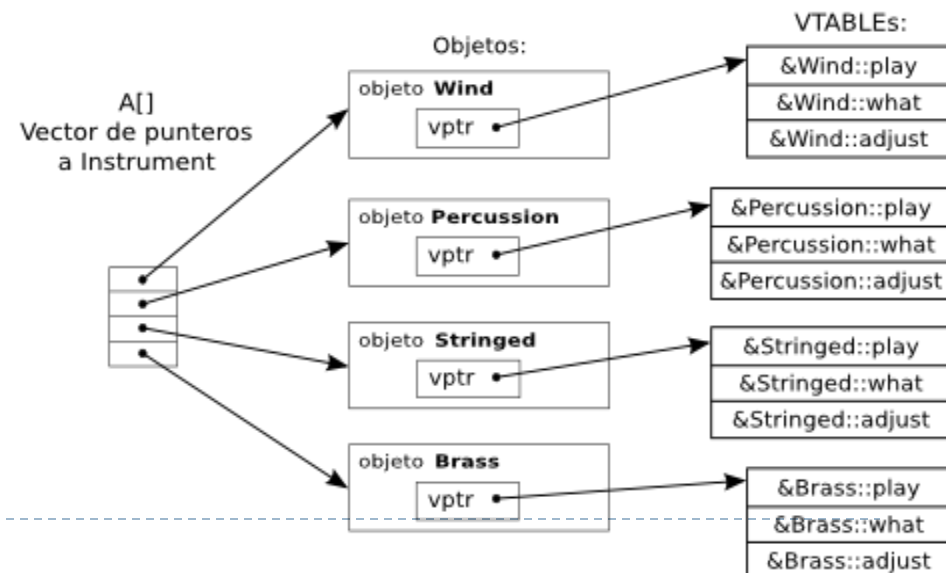
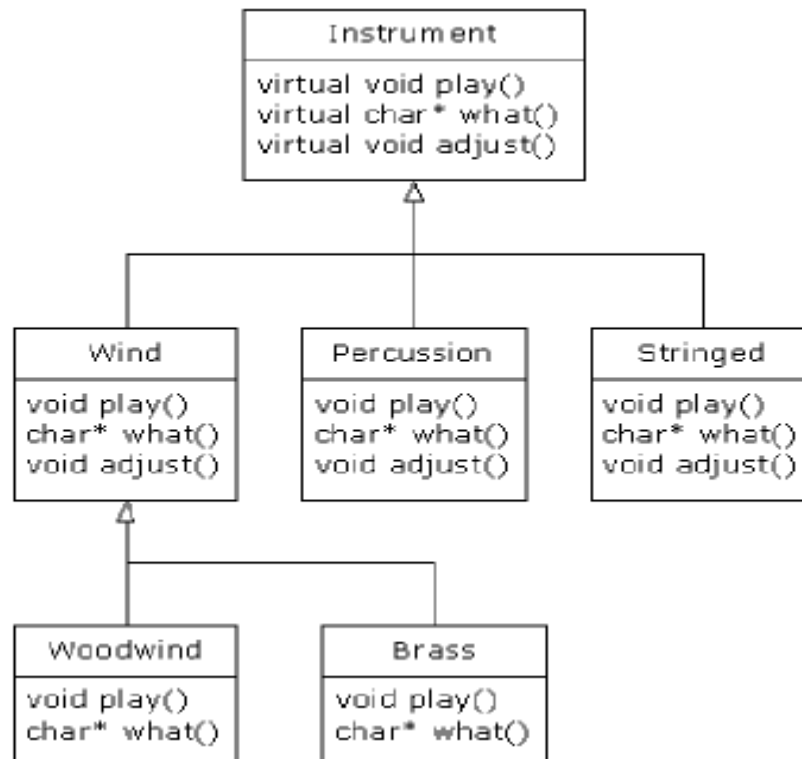
# Polimorfismo

---

- ▶ Para obtener polimorfismo en C++ debe usarse **funciones virtuales**, invocadas a través de **punteros o referencias a objetos de la clase base**
- ▶ 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**



# Polimorfismo



# Polimorfismo

---

- ▶ 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
- 



# 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()` (no miembro) 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()`





# Referencias a objetos y funciones virtuales-

## Ejemplo

---

```
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**



# Funciones virtuales puras y clases abstractas

---

- ▶ 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
  - ▶ Un uso importante de las clases abstractas es proporcionar una **interfaz sin exponer ningún detalle de implementación**
  - ▶ 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**
  - ▶ 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**
- 



# Clases abstractas - Ejemplo

---



# Clases abstractas - Ejemplo

---

- ▶ 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:  $h \cdot \pi \cdot r^2$



# Clases abstractas - Ejemplo

---

```
/** Contenedor.h*/  
#ifndef CONTENEDOR_H_  
#define CONTENEDOR_H_  
  
class Contenedor //contenedor genérico, clase abstracta  
{//interfaz pura, no tiene campos  
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*/  
};  
#endif /* CONTENEDOR_H_ */
```

---



# Clases abstractas - Ejemplo

---

```
/** Contenedor.cpp */
#include "Contenedor.h"
#include <iostream>
using std::cout;
using std::endl;
void Contenedor::mostrarVolumen() const
{
    cout<<endl
    <<"El volumen del contenedor es: "<<volumen()<<endl;
}
Contenedor::~Contenedor(void)
{
    cout<<"Invocado el destructor de Contenedor"<<endl;
}
```



# Clases abstractas - Ejemplo

---

```
/** Caja.h*/
#ifndef CAJA_H_
#define CAJA_H_

#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;
    virtual ~Caja(void);
protected: //ahora los hago protected
    double largo;
    double ancho;
    double alto;
};
```

---



# Clases abstractas - Ejemplo

```
/** Caja.cpp */  
#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;}
```

# Clases abstractas -Ejemplo

---

```
/** Lata.h*/  
#ifndef LATA_H_  
#define LATA_H_  
  
#include "Contenedor.h"  
  
class Lata :public Contenedor  
{  
public:  
    Lata(double al=4.0, double d=2.0);  
    virtual double volumen() const;  
    virtual ~Lata(void);  
protected:  
    double alto;  
    double diametro;  
};  
  
#endif /* LATA_H_ */
```



# Clases abstractas -Ejemplo

---

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

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

double Lata::volumen() const
{
    return M_PI* diametro*diametro*alto;
}

Lata::~Lata(void)
{
    cout<<"Invocado destructor de Lata"<<endl;
}
```



# Clases abstractas - Ejemplo

---

```
/** CajaBotellas.h*/
#ifndef CAJABOTELLAS_H_
#define CAJABOTELLAS_H_

#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;
};

#endif /* CAJABOTELLAS_H_ */
```

---



# Clases abstractas - Ejemplo

---

```
/** CajaBotellas.cpp */
#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;
}
```

---



# Clases abstractas - Ejemplo

---

```
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;
}
```



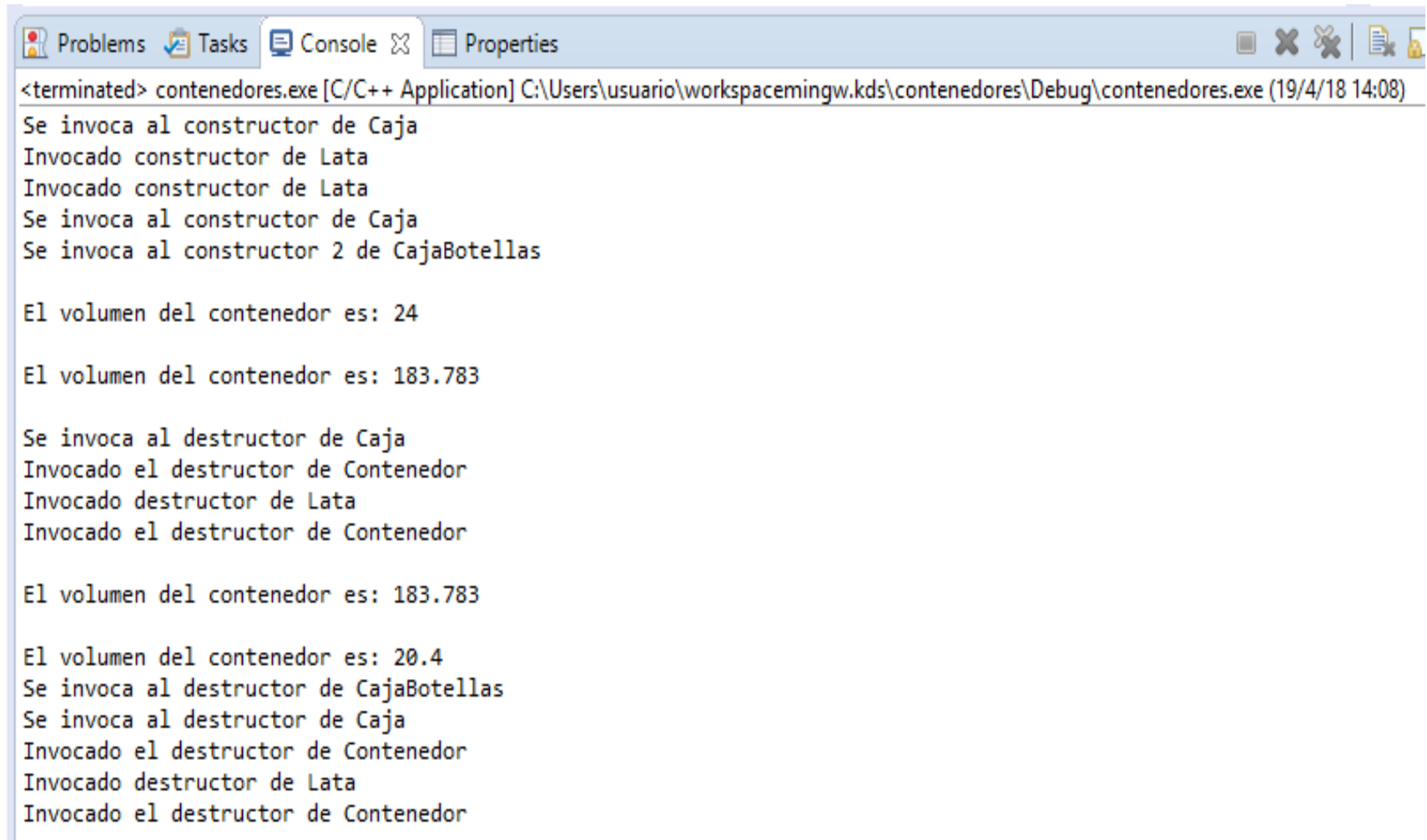
# Clases abstractas - Ejemplo

---

```
#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;}
```

---

# Clases abstractas - Ejemplo



```
<terminated> contenedores.exe [C/C++ Application] C:\Users\usuario\workspacemingw.kds\contenedores\Debug\contenedores.exe (19/4/18 14:08)
Se invoca al constructor de Caja
Invocado constructor de Lata
Invocado constructor de Lata
Se invoca al constructor de Caja
Se invoca al constructor 2 de CajaBotellas

El volumen del contenedor es: 24

El volumen del contenedor es: 183.783

Se invoca al destructor de Caja
Invocado el destructor de Contenedor
Invocado destructor de Lata
Invocado el destructor de Contenedor

El volumen del contenedor es: 183.783

El volumen del contenedor es: 20.4
Se invoca al destructor de CajaBotellas
Se invoca al destructor de Caja
Invocado el destructor de Contenedor
Invocado destructor de Lata
Invocado el destructor de Contenedor
```